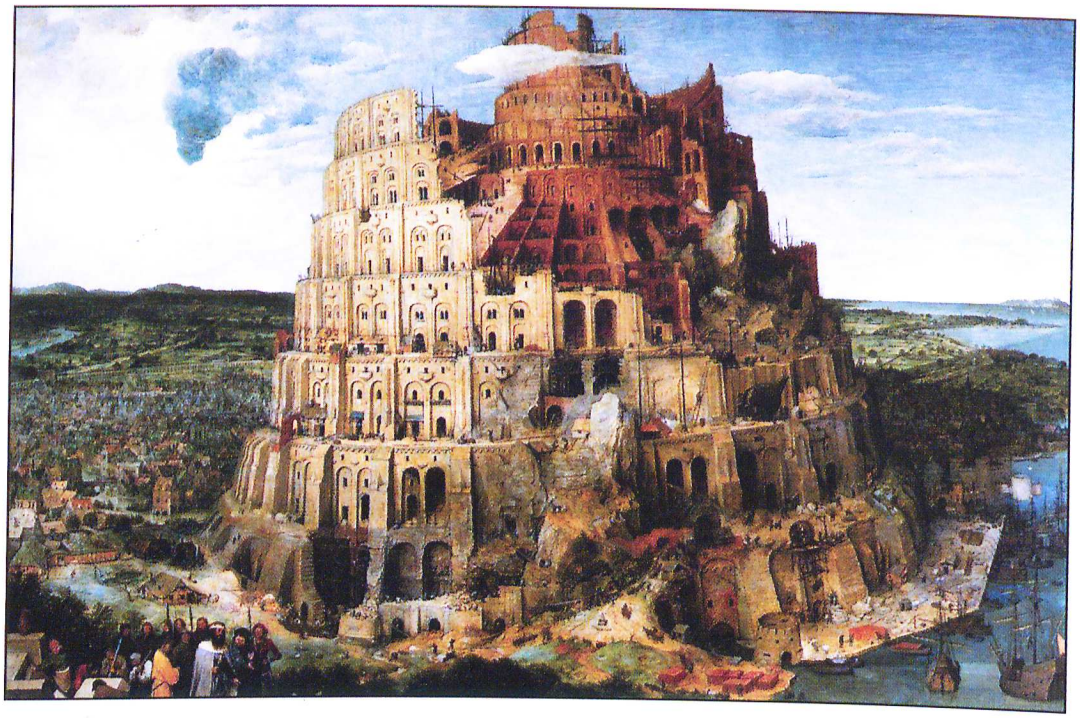
Szerkesztette
Nyékyné Gaizler Judit

Programozási nyelvek



KISKAPU

Szerzők

Ásványi Tibor
Balázs Iván József
Csizmazia Albert
Csizmazia Balázs
Csontos Péter
Fodor Szabina
Hegedűs Hajnalka
Horváth Zoltán
Juhász András
Kozsik Tamás
Kovács D. Lehel István
Lakatos Attila
Marcinkovics Tamás
Nohl Attila Rajmund
Nyékyné Gaizler Judit
Porkoláb Zoltán
Pécsy Gábor
Tejfel Máté
Sergyán Szabolcs
Zsók Viktória

Programozási nyelvek

Szerkesztő
Nyékyné Gaizler Judit

Kiskapu Kft.
Budapest, 2003.

A könyv az Oktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolított felsőoktatási tankönyvtámogatási program keretében jelent meg.



A kiadás munkatársai:

Alkotó szerkesztő:
Nyékyné Gaizler Judit

Lektorok:
Kozma László
Csörnyei Zoltán (2. és 16. fejezet)
Pásztorné Varga Katalin (17. fejezet)

© Ásványi Tibor, Balázs Iván József, Csizmazia Albert, Csizmazia Balázs, Csontos Péter, Fodor Szabina, Hegedűs Hajnalka, Horváth Zoltán, Juhász András, Kozsik Tamás, Kovács D. Lehel István, Lakatos Attila, Marcinkovics Tamás, Nohl Attila Rajmund, Nyékyné Gaizler Judit, Porkoláb Zoltán, Pécsy Gábor, Tejfel Máté, Sergyán Szabolcs, Zsók Viktória

Felelős kiadó a Kiskapu Kft. ügyvezető igazgatója

© 2003 Kiskapu Kft. Minden jog fenntartva

1081 Budapest, Népszínház utca 31.
Telefon: (+36 1) 303-9119, (+36 1) 477-0443
Fax: (+36 1) 303-1619
Honlap: `http://www.kiskapu.hu/`
Elektronikus cím: `kiskapu@kiskapu.hu`

Felelős szerkesztő: Szy György
Tördelés: Lőrentey Károly
Borítóterv: Zsédely Teréz
Anyanyelvi lektor: Rézműves László

ISBN: 963 9301 47 7

Nyomás és kötés: Debreceni Kinizsi Nyomda
Felelős vezető: Bördős János Igazgató

Áttekintés

Bevezetés • *2*

1. Programozási nyelvek tervezése • *7*
2. Lexikális elemek • *29*
3. Vezérlési szerkezetek, utasítások • *45*
4. Hatókör, élettartam • *103*
5. Típusok • *119*
6. Típuskonstrukciók • *167*
7. Alprogramok • *203*
8. Kivételkezelés • *277*
9. Modulok • *315*
10. Absztrakt adattípusok • *333*
11. Típussal való paraméterezés • *353*
12. Objektumorientált programozás • *391*
13. A programhelyesség-ellenőrzés nyelvi elemei • *459*
14. Párhuzamos nyelvi elemek • *501*
15. Programkönyvtárak • *561*
16. A funkcionális programozási nyelvek elemei • *589*
17. Logikai programozás • *637*
18. Aspektusorientált programozás • *685*
19. Programozási nyelvek rövid leírásai • *697*
20. Függelék • *711*

Irodalomjegyzék • *718*
Tárgymutató • *736*

Tartalomjegyzék

Bevezetés • *2*

1. Programozási nyelvek tervezése • *7*

Fodor Szabina

1.1. **Történeti áttekintés** • *7*
A korai nyelvek • *8*
A szoftverfejlesztés kialakulása • *9*
1.2. **A programozási nyelvek szerepe** • *11*
1.3. **A programozási nyelvek tervezésének alapelvei** • *13*
A „jó" programozási nyelv tulajdonságai • *13*
A nyelvtervezés lépései • *18*
1.4. **Programozási nyelvek csoportosítása** • *18*
Imperatív, procedurális nyelvek • *18*
Applikatív, funkcionális nyelvek • *19*
Szabály alapú, logikai nyelvek • *20*
Objektumorientált nyelvek • *20*
1.5. **A programozási nyelvek szabványosítása** • *21*
1.6. **Nemzetközi programozás** • *24*
1.7. **A programozási környezet nyelvre gyakorolt hatása** • *25*
Önálló fordítási egységek • *25*
Tesztelés és nyomkövetés • *26*

2. Lexikális elemek • *29*

Nyékyné Gaizler Judit

2.1. **A jelkészlet** • *29*
Az ASCII kód • *31*
Az EBCDIC kód • *31*
Az ISO 8859 család • *32*
A Unicode szabvány • *32*

2.2. **A programozási nyelvek jelkészletei** • *33*
2.3. **Azonosítók** • *35*
A használható karakterek • *36*
A megengedett szintaxis • *36*
A kis- és a nagybetűk megkülönböztetése • *37*
Hosszúsági megkötések • *37*
A fenntartott szavak • *37*
2.4. **Literálok** • *39*
Számliterálok • *39*
Karakterek és karaktersorozatok • *41*
2.5. **Megjegyzések** • *42*

3. **Vezérlési szerkezetek, utasítások** • *45*

Csizmazia Albert, Csizmazia Balázs

3.1. **A programozó feladata** • *45*
Mondatszerű leírás • *46*
Folyamatábrák • *47*
D-diagramok • *48*
Blokkdiagramok • *49*
Struktogramok • *50*
3.2. **Az assembly nyelvű megoldás** • *50*
A Pascal nyelvű program • *51*
A Motorola 68000-es család programozása • *51*
Az MC 68000-es assembly kód • *56*
Az assembly és a Pascal nyelvű kód összevetése • *57*
3.3. **Egy elemi megközelítés** • *58*
A while-programok elemei • *59*
Magasabb szintű műveletek • *59*
3.4. **Vezérlési megközelítések** • *61*
Imperatív programozási nyelvek • *61*
Deklaratív és funkcionális nyelvek • *61*
Párhuzamos végrehajtás • *62*
Eseményvezérelt programozás • *62*
3.5. **A vizsgált programozási nyelvek** • *63*
Mondatszerű algoritmusleírás: COBOL • *63*
Strukturált programozás: a Pascal nyelv • *65*
Hordozható assembly: a C nyelv • *66*
Minden objektum: a Smalltalk • *67*
Egyéb vizsgált programnyelvek • *67*

3.6. **Értékadás, aritmetikai utasítások** • *68*
A COBOL eszköztára • *68*
Egyszerű értékadás: a Pascal nyelv • *69*
Értékadás a C nyelvben • *69*
A Smalltalk megoldása • *70*
A többszörös értékadás és a CLU • *70*
Az értékadás szerepe a programokban • *70*
Az üres utasítás • *71*
3.7. **Szekvencia és blokk utasítás** • *71*
Blokk utasítás a Pascalban • *72*
Szakítás a Pascal hagyományaival: az Ada nyelv • *73*
A C nyelvcsalád jellemzői • *74*
A Smalltalk blokk utasítása • *74*
3.8. **Feltétel nélküli vezérlésátadás** • *75*
A COBOL eszközei • *76*
Feltétel nélküli vezérlésátadás a Pascalban • *76*
Modula-3: búcsú a GOTO-tól • *77*
Különleges vezérlési utasítások a C-ben • *77*
A Java újdonságai • *78*
3.9. **Elágazási szerkezetek** • *79*
Elágazás a COBOL-ban • *80*
Feltételes utasítás a Pascalban • *81*
Többirányú elágazás a Pascalban • *82*
Biztonságos elágazások: a Modula-3 újításai • *82*
Biztonságos CASE a Modula-3-ban • *84*
A C nyelv elágazási szerkezetei • *84*
Többágú elágazás a C-ben • *85*
Többágú elágazás a C#-ban • *85*
Feltételes utasítás a Smalltalkban • *86*
3.10. **Ciklusszerkezetek** • *87*
A COBOL ciklusai • *87*
Ciklusutasítások a Pascalban • *88*
Modula-3: biztonságos ciklusok • *90*
loop–end–exit ciklusok • *91*
Az Ada nyelv eszközei • *91*
Ismétlési szerkezetek a C és a Java nyelvben • *92*
A C# nyelv újdonságai • *93*
Bejárók (iterátorok) • *94*
Ciklusutasítás a Smalltalkban • *97*
3.11. **Önhívó kód (rekurzió)** • *98*
3.12. **Összefoglalás** • *99*

4. Hatókör, élettartam • *103*

Balázs Iván József, Porkoláb Zoltán

4.1. **A memóriaterülek típusai** • *104*
A statikus memóriaterület • *104*
A dinamikus memóriaterület • *104*
Az automatikus memóriaterület • *105*
Egy egyszerű példa • *105*
4.2. **Hatókör** • *106*
Globális hatókör • *108*
Fordítási egység mint hatókör • *108*
Függvény és kódblokk mint hatókör • *109*
Egy típus mint hatókör • *110*
4.3. **Élettartam** • *110*
4.4. **Példák** • *113*

5. Típusok • *119*

Pécsy Gábor

5.1. **Mi a típus?** • *119*
Típus a fordítóprogram szemszögéből • *119*
Típus a programozó szemszögéből • *120*
A programozási nyelvek típusossága • *122*
Típuskonverzió • *123*
5.2. **Típusok osztályozása** • *127*
Típusosztályok • *128*
Attribútumok az Ada nyelvben • *128*
5.3. **Skalár típusosztály** • *129*
Ábrázolás • *129*
Műveletek • *130*
Az Ada skalár típusai • *130*
5.4. **Diszkrét típusosztály** • *131*
Felsorolási típusok • *131*
Egész típusok • *133*
Csellengők • *138*
5.5. **Valós típusosztály** • *140*
Típusértékhalmaz • *140*
Műveletek • *141*
Programozási nyelvek • *142*
5.6. **Mutató típusok** • *143*
Memóriagazdálkodás • *144*
Típusértékhalmaz • *146*
Műveletek • *148*
Alprogramra hivatkozó mutató • *151*
Nyelvi specialitások • *152*

5.7. **Kifejezések** • *155*
Kifejezések szerkezete • *155*
Kifejezések kiértékelése • *157*
5.8. **Egyéb nyelvi specialitások** • *161*
5.9. **Összefoglalás** • *162*

6. Típuskonstrukciók • *167*

Pécsy Gábor

6.1. **Direktszorzat típus** • *170*
Típusértékhalmaz • *170*
Műveletek • *171*
A direktszorzat ábrázolása • *174*
Egyedi nyelvi elemek • *176*
6.2. **Unió típus** • *178*
Típusértékhalmaz • *178*
Műveletek • *179*
Unió jellegű típuskonstrukciók • *180*
6.3. **Iterált típus** • *187*
6.4. **Vektor** • *188*
Típusértékhalmaz • *188*
Műveletek • *189*
Nyelvi specialitások • *190*
Általánosítás több dimenzióra – tömbök • *193*
6.5. **Halmaz** • *195*
Típusértékhalmaz • *195*
Műveletek • *196*
6.6. **Egyéb iterált konstrukciók** • *196*
6.7. **Összefoglalás** • *198*

7. Alprogramok • *203*

Kozsik Tamás

7.1. **Az alprogramok hatása a szoftverminőségre** • *204*
7.2. **Eljárások és függvények** • *205*
Nyelvek, ahol nincs különbség az eljárások és függvények között • *206*
Nyelvek, ahol van különbség az eljárások és függvények között • *208*

7.3. **Az alprogramok és hívások szerkezete** • *209*
Mi lehet paraméter vagy visszatérési érték? • *209*
Az alprogramok specifikációja • *216*
Alprogramok törzse • *222*
Az alprogramok meghívása • *227*
Rekurzív alprogramok • *232*
Az alprogramok deklarációja • *233*
Makrók és inline alprogramok • *235*
Alprogram típusok • *236*
7.4. **Paraméterátadás** • *237*
Paraméterátadási módok • *237*
A paraméterátadási módok összehasonlítása • *245*
A lehetőségek néhány programozási nyelvben • *247*
7.5. **Az alprogramok környezete** • *253*
Külön fordíthatóság • *254*
Beágyazhatóság • *255*
Statikus és dinamikus hatókör • *257*
A változók élettartama • *260*
7.6. **Az alprogramnevek túlterhelése** • *261*
Operátorok túlterhelése • *264*
7.7. **Az alprogramok megvalósítása** • *264*
Paraméterként átadott alprogramok megvalósítása • *267*
7.8. **Iterátorok** • *269*
7.9. **Korutinok** • *272*
7.10. **Összefoglalás** • *274*

8. **Kivételkezelés** • *277*

Nohl Attila Rajmund

8.1. **Bevezetés** • *277*
A kivételkezelés alapfogalmai • *277*
Miért jó a kivételkezelés? • *279*
A kivételkezelések összehasonlításának szempontjai • *285*
8.2. **A kivételkezelés kezdetei** • *285*
Egy utasítás kivételkezelése: FORTRAN • *285*
Több utasítás kivételkezelése: COBOL • *286*
Dinamikus kivételkezelés: PL/I • *286*

8.3. **Fejlett kivételkezelés** • *288*
Statikus kivételkezelés: CLU • *288*
Kivételek terjedése: Ada 95 • *289*
Kivételosztályok megjelenése: C++ • *291*
Helyességbizonyítás és kivételkezelés: Eiffel • *292*
A *finally* blokk: Modula-3 • *294*
Ellenőrzött kivételek: Java • *295*
A Delphi kivételkezelése • *296*
Egymásba ágyazott kivételek: C# • *297*
Kivételkezelés függvényekkel: Common Lisp • *297*
Új megoldások: Perl • *299*
8.4. **Összefoglaló** • *301*
8.5. **Példaprogramok kivételkezelésre** • *302*
C++ • *302*
Java • *304*
Ada 95 • *306*
Eiffel • *308*

9. **Modulok** • *315*

Lakatos Attila

9.1. **A moduláris tervezés** • *315*
A moduláris tervezés kritériumai • *315*
A modularitás alapelvei • *316*
9.2. **A modulok nyelvi támogatása** • *320*
Alacsonyszintű programnyelvek • *320*
Procedurális programnyelvek • *321*
Az Ada nyitott modulok támogatása • *323*
Objektumorientált programnyelvek • *328*
Funkcionális programnyelvek • *329*

10. **Absztrakt adattípusok** • *333*

Pécsy Gábor

10.1. **Típuskonstrukciók és adatabsztrakció** • *333*
10.2. **Elvárások a programozási nyelvekkel szemben** • *334*
10.3. **Modulokra bontás** • *335*
10.4. **Egységbe zárás** • *335*
10.5. **Az ábrázolás elrejtése** • *335*
Átlátszatlan típus a C-ben • *336*
Ada típusok privát nézete • *338*
CLU absztrakt adattípusok • *338*
Láthatósági szintek • *339*

10.6. A specifikáció és implementáció szétválasztása • *340*
10.7. A modulfüggőség kezelése • *342*
10.8. Konzisztens használat • *343*
10.9. Általánosított programsémák • *344*
10.10. A programozási nyelvek értékelése • *347*

11. Típussal való paraméterezés • *353*

Hegedűs Hajnalka, Marcinkovics Tamás, Nyékyné Gaizler Judit, Kozsik Tamás

11.1. Polimorfizmus • *359*
Parametrikus polimorfizmus • *360*
Altípusos polimorfizmus • *364*
Túlterheléses polimorfizmus • *368*
Típuskényszerítéses polimorfizmus • *370*
A polimorfizmus megvalósítása monomorf nyelvekben • *371*
11.2. A sablon-szerződés modell • *373*
11.3. Sablonparaméterek • *376*
Típus és típusosztály • *376*
Alprogram • *378*
Objektum • *379*
Modul • *379*
11.4. Példányosítás • *381*
Explicit példányosítás • *382*
Igény szerinti példányosítás • *382*
Lusta példányosítás • *382*
Sablonparaméterek megfeleltetése • *383*
Specializáció • *384*
11.5. Sablonok és öröklődés • *385*
11.6. Összegzés • *387*

12. Objektumorientált programozás • *391*

Kovács D. Lehel István, Sergyán Szabolcs, Nyékyné Gaizler Judit

12.1. Az osztály és az objektum • *391*
Az osztályok és objektumok az egyes nyelvekben • *393*
12.2. Jelölési módok, diagramok • *400*
Osztálydiagram • *400*
Objektumdiagram • *400*
A példányosítás ábrázolása • *400*
12.3. Objektumok létrehozása és megszüntetése • *400*
12.4. Egységbezárás • *405*
12.5. Adatrejtés, interfészek • *406*
Friend metódusok, osztályok • *409*

12.6. **Osztályadat, osztálymetódusok** • *411*
12.7. **Öröklődés** • *415*
Adatrejtés és öröklődés • *423*
Polimorfizmus és dinamikus kötés • *426*
Absztrakt osztályok • *437*
A közös ős • *440*
Többszörös öröklődés • *441*
Interfészek • *450*
Beágyazott osztályok, tagosztályok • *454*
12.8. **Összefoglalás** • *455*

13. **A programhelyesség-ellenőrzés nyelvi elemei** • *459*

Juhász András

13.1. **A helyességi specifikációs nyelv** • *463*
Az Eiffel és az elsőrendű logika • *464*
Példa • *469*
Elő- és utófeltételek • *470*
Osztályinvariáns • *476*
Ciklusok • *478*
A check utasítás • *482*
Öröklődés • *483*
13.2. **A programhelyesség definiálása az Eiffelben** • *487*
Az attribútumok helyessége • *487*
A metódusok helyessége • *487*
Az osztályok helyessége • *490*
A programok helyessége • *491*
13.3. **A programhelyesség-ellenőrzés problémái** • *492*
Függőségek • *492*
Helyességi specifikációs nyelv • *493*
13.4. **A *VEREM* osztály** • *495*

14. **Párhuzamos nyelvi elemek** • *501*

Zsók Viktória (1., 2–3. alfejezet), Tejfel Máté (1., 4–6. alfejezet)

14.1. **A párhuzamos nyelvek általános jellemzése** • *505*
14.2. **Ada** • *514*
Taszkok • *514*
A taszk specifikációja és törzse • *515*
Taszk típusok, taszk objektumok • *517*
Taszkok elindítása, végrehajtása • *518*
Taszkok lezárása (terminálása) • *519*
A taszkok attribútumai • *519*
Entry, entry-hívás, **accept** utasítás • *520*
Belépésipont-családok • *521*

A **select** utasítás • *522*
Szelektív hívásfogadás • *522*
Feltételes hívásfogadás • *523*
A **terminate** alternatíva • *524*
A **delay** és az **abort** utasítás • *524*
Az **else** alternatíva • *524*
A feltételes **entry** hívás • *526*
Időhöz kötött **entry** hívás • *526*
Kivételkezelés • *526*
Példák • *527*
14.3. **CSP** • *532*
14.4. **Occam** • *534*
14.5. **Java** • *537*
14.6. **C alapú könyvtárak** • *543*
MPI • *543*
PVM • *553*

15. **Programkönyvtárak** • *561*

Csontos Péter, Lakatos Attila

15.1. **Követelmények a programkönyvtárakkal szemben** • *562*
A jó könyvtárfejlesztő tulajdonságai • *562*
Alapvető minőségi követelmények • *563*
A programkönyvtárakra vonatkozó egyedi követelmények • *566*
A követelmények teljesítésének feltételei • *569*
15.2. **Objektumorientált programkönyvtár-tervezés** • *569*
Osztályhierarchia • *570*
Az osztályok mérete • *573*
A szolgáltatások mérete • *576*
Az osztályok fajtái • *578*
15.3. **Új paradigmák** • *582*
15.4. **Általános programkönyvtárak** • *583*
Adatszerkezetek • *583*
I/O • *584*
Memóriakezelés • *584*
15.5. **Programkönyvtárak életciklusa** • *584*
Tervezési fázis • *585*
Megvalósítási fázis • *586*
Karbantartási fázis • *586*

16. A funkcionális programozási nyelvek elemei • *589*

Horváth Zoltán *16.9. alfejezet: Zsók Viktória, Tejfel Máté, Horváth Zoltán*

16.1. **Bevezetés** • *589*
A funkcionális programozási stílus • *590*
Funkcionális program felépítése és kiértékelése • *591*
A modern funkcionális nyelvek főbb jellemzői • *593*
A funkcionális nyelvek rövid áttekintése • *596*
16.2. **Egyszerű funkcionális programok** • *597*
Egyszerű függvények definíciója • *597*
Feltételek megadása • *597*
Mintaillesztés • *598*
16.3. **Függvények típusa, magasabbrendű függvények** • *599*
Egyszerű típuskonstrukciók • *600*
Lokális deklarációk • *605*
Egy érdekesebb példa: királynők a sakktáblán • *607*
16.4. **Típusok és osztályok** • *609*
Polimorfizmus, típusosztályok • *609*
Algebrai adattípus • *612*
Típusszinonimák • *614*
Származtatott típusok • *614*
Típuskonstrukció osztályok • *615*
16.5. **Modulok** • *615*
Absztrakt algebrai adattípus • *616*
16.6. **Frissíthető változók, monádok, mellékhatás** • *619*
Egyszeresen hivatkozott változók • *619*
Monádok • *620*
Frissíthető változók • *621*
16.7. **Interaktív funkcionális programok** • *622*
16.8. **Kivételkezelés** • *624*
16.9. **Párhuzamos kiértékelés és elosztott programok** • *625*
Párhuzamos és elosztott programozás Concurrent Cleanben • *626*
Distributed Haskell with Ports • *630*
A JoCaml párhuzamos és osztott nyelvi elemei • *632*

17. Logikai programozás • *637*

Ásványi Tibor

17.1. **Logikai programok** • *638*
Tények • *639*
Szabályok • *640*
Rekurzív szabályok • *644*

17.2. **Bevezetés a Prolog nyelvbe** • *645*
17.3. **A logikai programok objektumai** • *646*
17.4. **Listakezelés rekurzív LP módszerekkel** • *647*
A rekurzív keresés • *649*
Az eredmény fokozatos közelítése • *649*
Az akkumulátor módszer • *650*
Az általánosítás módszere • *651*
17.5. **A Prolog gép** • *652*
Tiszta Prolog programok végrehajtása • *652*
Az illesztés algoritmusa • *653*
NSTO programok • *654*
Első argumentum indexelés • *656*
Utolsó hívás optimalizáció • *657*
17.6. **Vezérlésmódosítás a Prologban** • *658*
Feltételes célok • *658*
Tagadás • *660*
17.7. **A Prolog metalogikai predikátumai** • *662*
Aritmetika • *662*
Termek típusa és összehasonlítása • *664*
Termműveletek • *665*
17.8. **Prolog műveleti jelek** • *666*
17.9. **A Prolog logikán kívüli predikátumai** • *668*
Programbetöltés • *669*
Bemenet és kimenet • *669*
Önmódosító programok • *670*
17.10. **Prolog célok megoldásainak összegyűjtése** • *672*
17.11. **Kivételkezelés a Prologban** • *673*
17.12. **Prolog modulrendszerek** • *675*
A név alapú modulrendszer • *675*
Az eljárás alapú modulrendszer • *675*
A forrásmodul megváltoztatása • *677*
17.13. **Kitekintés** • *679*
Ajánlott irodalom • *679*
Klasszikus LP-kiterjesztések • *680*
Az ötödik generációs számítógépek és programjaik • *680*
Újabb irányzatok • *681*
17.14. **Összefoglalás** • *682*

18. Aspektusorientált programozás • *685*

Csontos Péter, Kozsik Tamás

18.1. Az AOP áttekintése • *686*
Aspektusok és komponensek • *687*
Aspektusleíró nyelvek • *688*
Aspektusszövők • *688*
18.2. Az AspectJ bemutatása • *689*
Az AspectJ elemei és főbb tulajdonságai • *689*
Egy rövid AspectJ példa • *690*
Fejlesztőeszközök és rokon nyelvek • *690*
18.3. Az AOP-vel rokon paradigmák és megvalósításaik • *691*
Vonatkozások többdimenziós szétválasztása (MDSC) • *691*
Adaptív programozás (AP) • *692*
Kompozíciós szűrők (CF) • *693*
Generatív programozás (GP) • *693*
Intencionális programozás (IP) • *694*
További ígéretes kezdeményezések • *695*

19. Programozási nyelvek rövid leírásai • *697*

Csontos Péter et al.

19.1. Ada 95 • *697*
19.2. ALGOL 60 • *698*
19.3. ALGOL 68 • *698*
19.4. BASIC • *699*
19.5. BETA • *699*
19.6. C • *699*
19.7. C++ • *700*
19.8. C# • *700*
19.9. Clean • *701*
19.10. CLU • *701*
19.11. COBOL • *701*
19.12. Delphi • *702*
19.13. Eiffel • *702*
19.14. FORTRAN • *703*
19.15. Haskell • *703*
19.16. Java • *704*
19.17. LISP • *704*
19.18. Maple • *705*
19.19. Modula-2 • *705*
19.20. Modula-3 • *705*
19.21. Objective-C • *706*
19.22. Pascal • *706*
19.23. Perl • *706*

19.24. PHP • *707*
19.25. PL/I • *707*
19.26. Python • *707*
19.27. SIMULA 67 • *708*
19.28. Smalltalk • *708*
19.29. SML • *708*
19.30. SQL • *709*
19.31. Tcl • *709*

20. Függelék • *711*

Irodalomjegyzék • *718*

Tárgymutató • *736*

Bevezetés

A programozási nyelvekről sokan hiszik, hogy csak jelölésformát adnak a program leírásához, s nem veszik figyelembe – esetleg nem is tudnak róla –, hogy a *magasszintű* vagy *felhasználó központú nyelvek* eszközeikkel segítik a program bonyolultságának kezelését is. A különböző nyelvek lehetőségeikkel különböző programozási szemléleteket sugallnak, ezért is nagyon veszélyes az a sok helyen ma is elterjedt gyakorlat, amikor a programozás módszertanának oktatása az egyes programozási nyelveken keresztül, és nem azoktól függetlenül történik – ekkor ugyanis csak egy leszűkített kép alakulhat ki a programírás lehetőségeiről.

A programozás célja a *jó minőségű programtermék* előállítása, ezért a programozás oktatásának a feladat, és a feladatot megoldó program minél általánosabb megfogalmazásából kell kiindulnia [Fót 83], s emellett, erre építve kell megismertetnie a programozókat a különböző konkrét nyelvi eszközökkel, amelyek támogatják elgondolásai megvalósításában. Ugyanakkor azonban, ahogy helytelen a módszertant egy-egy konkrét nyelven keresztül tanítani, hasonlóan zsákutcába vezet, ha azt mondjuk, hogy a használt programozási nyelv nem is fontos a módszertan szempontjából. Ez olyan – ahogy Bertrand Meyer írja [Mey 00] –, mint „a madár szárnyak nélkül". A gondolat nem választható el a kifejezés lehetőségeitől. Nem véletlen, hogy a programozásban egy nyelv sem lett egyeduralkodóvá, ahogy az sem, hogy egyre újabb és újabb programozási nyelveket terveznek, amelyek eszközeikkel egyre jobban támogatják a különböző módszertani elgondolások, követelmények gyakorlatba való átültetését.

A programozási nyelvek tervezőinek három problémával kell megküzdeniük [Hor 84]:

- A nyelv által biztosított ábrázolásmódnak egyszerre kell illeszkednie. a számítógéphez és az alkalmazásokhoz.
- A nyelvnek jó jelölésrendszert kell biztosítania az algoritmus leírásához.
- A nyelv eszközként kell szolgáljon a program bonyolultságának kezeléséhez.

A szoftverminőség szempontjai

A szoftver – termék, s mint minden termékre, erre is megfogalmazhatunk – ahogy meg is fogalmaztak sokan (például Meyer, Horowitz, Liskov [Mey 00, Hor 84, Lis 86]) – különböző minőségi jellemzőket, követelményeket. A programozási módszertan egyik legfőbb célja, hogy elvi eszközt adjon a kezünkbe a jó minőségű programtermék létrehozásához. A programozási nyelv tervezését, illetve a már meglévő nyelvek értékelését feltétlenül befolyásolják módszertani megfontolásaink.

A következőkben először Bertrand Meyer munkája [Mey 00] alapján áttekintjük a „jó" szoftver jellemzőit. Ezután foglalkozunk csak a módszert támogató nyelvi eszközökkel – számos programozási nyelven keresztül.

A szoftver minőségét számos tényező befolyásolja. Ezek egy részét – ilyenek például a megbízhatóság, a gyorsaság, vagy a könnyű használhatóság – alapvetően a program felhasználója érzékeli. Más jellemzők – például, hogy egyes részeit milyen könnyű egy más, de hasonló feladat megoldásához újra felhasználni – a programfejlesztőket érintik.

Helyesség • A programtermék *helyessége* azt fejezi ki, hogy a program pontosan megoldja a feladatot, megfelel a kívánt specifikációnak. Ez az első és legfontosabb követelmény, hiszen ha egy program nem úgy működik, ahogy kellene, akkor az egyéb szempontok nyilván nem sokat számítanak.

Ami itt alapvető: a pontos és minél teljesebb specifikáció.

Megbízhatóság • Egy programot *megbízhatónak* nevezünk, ha helyes, és abnormális – a specifikációban nem leírt – helyzetekben sem történik katasztrófa, hanem valamilyen „ésszerű" működést produkál.

A fenti meghatározás jól tükrözi, hogy a megbízhatóság messze nem olyan precíz fogalom, mint a helyesség. Mondhatnánk ugyan, hogy ha precízebb, teljesebb specifikációt készítettünk volna, akkor ez pontosan a helyességet jelentené, de a gyakorlat azt mutatja, hogy mindig vannak olyan esetek, amelyeket egy specifikáció nem ír le explicit módon. Ezért kiemelt jelentőségű a programtermék minősége szempontjából a megbízhatóság.

Karbantarthatóság • A *karbantarthatóság* azt mutatja, hogy milyen könnyű a programterméket a specifikáció változtatásához igazítani.

A felhasználó gyakran kéri a programtermék továbbfejlesztését, módosítását, új szabályozókhoz igazítását. Egyes felmérések szerint a szoftverköltségek 70 %-át szoftver karbantartásra fordítják, így érthető, hogy ez a követelmény a program minőségét komolyan befolyásolja. (Ennek a követelménynek nagy programok, programrendszerek írásánál van jelentősége, hiszen kis programoknál általában semmilyen változtatás nem túl bonyolult.)

A karbantarthatóság növelése szempontjából a két legfontosabb alapelvnek a tervezési egyszerűséget és a decentralizációt (minél önállóbb modulok létrehozását) tekinthetjük.

Újrahasznosíthatóság • Az *újrahasznosíthatóság* vagy *újrafelhasználhatóság* a szoftvertermékek azon képessége, hogy egészben vagy részben újra felhasználhatók új alkalmazásokban.

Ez más, mint a karbantarthatóság, hiszen ott ugyanazon specifikáció módosult, míg most azt a tapasztalatot szeretnénk hasznosítani, hogy a szoftverrendszerek sok eleme közös mintákat követ, hogy elkerülhessük a már megoldott problémák újrakidolgozását.

Ez a kérdés különösen fontos, nem is elsősorban egy egyedi programtermék előállításánál, hanem ha a szoftverkészítés általános optimalizációját tekintjük, hiszen minél inkább lehetőségünk van arra, hogy újrahasznosítható elemek segítségével oldjuk meg feladatainkat, annál több energiánk marad a többi minőségi jellemző javítására is (azonos költség mellett).

Kompatibilitás • A *kompatibilitás* (összeegyeztethetőség) azt mutatja meg, milyen könnyű a szoftvertermékeket egymással kombinálni.

A programokat általában nem „légüres térben" fejlesztjük, a hatékonyságot nagyságrendekkel növelheti, ha a programok egymással összekapcsolhatók. (A programok közti kommunikáció például valamilyen szabvány alapján történik, mint a Unixban.)

Egyéb jellemzők • A programtermék minőségi jellemzői közül fontos odafigyelnünk a program hordozhatóságára, hatékonyságára, emberközeliségére, tesztelhetőségére, áttekinthetőségére stb. is.

A program *hordozhatósága* azt mutatja, mennyire könnyű a programot más géphez, kiépítéshez vagy operációs rendszerhez – általában más fizikai környezethez – átalakítani.

A program *hatékonysága* a futási idővel és a felhasznált memória méretével arányos – minél gyorsabb, illetve minél kevesebb memóriát használ, annál hatékonyabb. (Ezek a követelmények sokszor egymás ellen hatnak, a gyorsabb futást sokszor nagyobb memóriaigény ellensúlyozza, és fordítva.)

A program *emberközelisége*, barátságossága a felhasználó számára rendkívül fontos: ez megköveteli, hogy az adatbevitel logikus és egyszerű, az eredmények formája áttekinthető legyen.

A *tesztelhetőség*, *áttekinthetőség* a program karbantartói, fejlesztői számára fontos szempontok, ezek nélkül a program megbízhatósága sem garantálható.

A programok tervezési szempontjai

Az előző követelmények egy része – a helyesség és a megbízhatóság javítása – elsősorban a specifikációs eszközök fejlesztését igényli. Minél könnyebb belátni, hogy a programnyelvi leírás tényleg a specifikációnak megfelelő megvalósítás, annál könnyebben írhatunk helyes, illetve megbízható programokat. Itt a legnagyobb szerepe a specifikáció (típusinvariáns, elő- és utófeltételek) leírását lehetővé tevő programnyelvi eszközöknek van – ezt támogatja például az Eiffel [Mey 00].

A követelmények másik csoportjának – gondoljunk itt elsősorban az újrahasznosíthatóság, karbantarthatóság, kompatibilitás szempontjaira – megvalósítását legjobban azzal támogathatjuk, ha programjainkat egymástól minél inkább független, jól definiált kapcsolatban álló programegységekként tervezzük meg. Ez az úgynevezett *moduláris tervezés* alapja. (A modul itt nem programnyelvi fogalom, hanem a tervezés egysége.) Ezt a kérdést bővebben a 9. fejezetben tárgyaljuk.

Célunk, hogy bemutassuk, a különböző programozási nyelvek tervezői milyen eszközöket választottak arra, hogy a professzionális programozókat támogassák a jó minőségű, megbízható szoftver készítésében.

A programozási nyelvek eszközeinek tanulmányozása

Természetes kérdés, hogy miért nem elég, ha ismerünk *egy* programozási nyelvet, mire is jó, ha külön foglalkozunk a programozási nyelvek lehetséges eszközeivel? A következőkben – elsősorban Robert W. Sebesta [Seb 02] munkája alapján – megpróbáljuk összefoglalni az ebből származó előnyöket:

A kifejezőerő növekedése • Gondolkodásunkat, absztrakciós készségünket nagyban befolyásolják az általunk használt nyelv lehetőségei. Azt tudjuk kifejezni, amire vannak szavaink. Hasonlóan, a programok fejlesztése, a megoldások tervezése során a programozókat komolyan segítheti, látókörüket tágíthatja a különféle programozási nyelvi eszközök ismerete. Ez még akkor is igaz, ha egy adott nyelvet kell használjanak, hiszen a jó elvek bármely környezetben alkalmazhatók.

A megfelelő programozási nyelv kiválasztása • Számos programozó csak egy, esetleg két nyelven keresztül tanult programozni. Mások olyan régi programozási nyelveket ismernek, amelyek ma már elavultnak számítanak, és nem jártasak a modern nyelvek eszközeiben. Ez azt eredményezheti, hogy amikor egy új feladatnál több programozási nyelv közül választhatnának, nem a legmegfelelőbbet választják – hiszen nem ismerik a többi nyelv lehetőségeit. Ha ezek a programozók ismernék az elérhető nyelvek egyedi jellemzőit, sokkal jobb helyzetben lennének a döntést meghozatalakor.

Az új eszközök könnyebb elsajátítása • A minőségi programozás állandó tanulást igényel. Újabb és újabb programozási nyelvek jelennek meg. Minél inkább ismerjük a programozási nyelvek alapvető elveit, annál könnyebben tanuljuk meg ezeket, annál könnyebben tartunk lépést a fejlődéssel.

A könyvben az egyes nyelvi elemekhez kapcsolódóan a legtöbb példát az Ada, a C/C++ és a Java nyelvekből vettük, alig van olyan fejezet (kivéve természetesen a funkcionális és a logikai programozás eszközeit bemutatókat), ahol ezek a nyelvek ne szerepelnének majdnem minden alfejezetben.

Könyvünkkel elsősorban a felsőfokú tanulmányaik keretében programozási nyelvekkel foglalkozó egyetemi és főiskolai hallgatók, valamint a számítástechnikai és informatikai szakemberek munkáját szeretnénk segíteni. Bizonyos mennyiségű informatikai előismeret szükséges a könyv megértéséhez: feltételezzük, hogy olvasóink legalább néhány kisebb programozási feladatot megoldottak már valamilyen programozási nyelven.

Szeretnénk tevőlegesen részt vállalni a magyar szaknyelv népszerűsítésében, így, ahol csak tudtunk, igyekeztünk a szakkifejezésekben magyar szavakat használni.

Köszönetnyilvánítások

A szerzők ezúton köszönik meg az Oktatási Minisztérium Felsőoktatási tankönyvpályázatának támogatását.

Megköszönik Kátai Imrének és Kozma Lászlónak, hogy a könyv megírása során szabadon felhasználhatták, és fel is használták az ELTE TTK Informatikai Tanszékcsoportjának infrastruktúráját. Enélkül ez a könyv nem készülhetett volna el.

Megköszönik az ELTE programtervező matematikus és informatika doktori iskolás hallgatóinak, hogy hatékonyan közreműködtek a könyv anyagának előkészítésében és a sajtóhibák számának csökkentésében.

Egy ilyen nagy terjedelmű könyvben – annak ellenére, hogy a szerzők és a szerkesztő a legjobb tudásuk szerint állították össze – előfordulhatnak hibák. Megkérjük az Olvasót, ha ilyet talál, akkor ezt írja meg elektronikus levélben a *pnymegj@elte.hu* címre. A szerzők minden konstruktív kritikát örömmel fogadnak.

1 Programozási nyelvek tervezése

Ebben a fejezetben először egy rövid áttekintést adunk a programozási nyelvek kialakulásáról. Próbálunk választ adni arra, hogy milyen tulajdonságok tesznek sikeressé, illetve ítélnek kudarcra egy nyelvet. Megvizsgáljuk továbbá, hogy a programozási nyelvek számának ugrásszerű emelkedésével hogyan változtak a programkészítés alapelvei, majd a programozási nyelvek egy módszertani-történeti csoportosítását adjuk. Végül elemezzük, hogy a programozási környezetek változása, fejlődése milyen hatást gyakorolt a programozási nyelvekre.

Mindmáig a programozási nyelvek ezreit alkották meg: már 1969-ben, a számítógépek mindennapi eszközzé válása előtt is 120 széles körben használt nyelv létezett [Sam 69]. A legtöbb programozó azonban munkája során alig néhányat használ ezekből. Kérdésként merülhet fel, hogy mi az értelme újabb nyelvek megismerésének és elsajátításának. Nos, a nyelvek alaposabb ismerete lehetővé teszi a feladathoz jobban illeszkedő nyelv kiválasztását, megkönnyíti az új programozási nyelvek elsajátítását, továbbá hatékonyabb algoritmusok készítését teszi lehetővé azáltal, hogy növeli a programozók „szókincsét".

Az elkészített program minőségét alapvetően befolyásoló tényezők a programhelyesség, a program megbízhatósága, bővíthetősége, újrafelhasználhatósága, hordozhatósága, hatékonysága, tesztelhetősége, emberközelisége, karbantarthatósága és könnyű programozhatósága. Ezek egy részét, például a megbízhatóságot, a gyorsaságot vagy a könnyű használhatóságot alapvetően a program használója érzékeli. Más jellemzők, mint az, hogy a program egyes részeit milyen könnyű egy más, de hasonló feladat megoldásához újra felhasználni a programfejlesztőket érintik. A helyesen megválasztott programozási nyelv nagymértékben segíti jó minőségű programok elkészítését.

1.1. Történeti áttekintés

A legkorábbi ismert algoritmusok időszámításunk előtt 1500–3000 évvel keletkeztek, ezeket az ókori Mezopotámia területén agyagtáblába vésve találták meg. Az első algoritmikus módszerek egyenletmegoldások voltak, amelyek leírásaiban konkrét számok szerepelnek és az eljárás végét az „íme ez az eljárás" jelzi [Sam 69].

Az első mai értelemben vett programot Ada Byron készítette Babbage analitikus gépére. A program a mai értelemben inkább gépi kód volt, bár a leírásban előfordul a változók fogalma (bemeneti-, munka- és eredményváltozók) és a változók indexelése, ami megadja a felhasználás számát, illetve jelzi a változó üres voltát.

1.1.1. A korai nyelvek

Az első gyakorlatban is használt programozási nyelvet, a Plankalkült Konrad Zuse [Zus 72] 1944-ben alkotta meg a Z4 elnevezésű számítógép számára. A Plankalkül nyelvet számítási leírások formalizált ábrázolására tervezték. Példaként nézzük meg az $x^2 + bx + c = 0$ másodfokú egyenlet megoldását Plankalkül nyelven! A két gyököt jelölje x_1 és x_2. A $b = V1$ és $c = V2$ jelölés mellett a program a következő:

$$\frac{V1}{2} = Z1$$
$$Z1 \cdot Z1 = Z2$$
$$Z2 - V2 = Z3$$
$$\sqrt{Z3} = Z4$$
$$Z1 \cdot (-1) = Z5$$
$$Z5 + Z4 = R1(x_1)$$
$$Z5 - Z4 = R2(x_2)$$

Zuse a változókat három típusba sorolta: bemeneti változók (V, mint Variablen); köztes értékek (Z, mint Zwischenwerte); illetve eredmény értékek (R, mint Resultatswerte). A számokat 32 biten ábrázolta, ebből 22 bit szolgált a mantissza, 7 bit pedig a karakterisztika kódolására. A fennmaradó 3 jelző bit az előjelet mutatta, illetve azt, hogy a szám valós, képzetes vagy nulla. Zuse ötleteit később az ALGOL 58 fejlesztői használták fel.

A hőskor számítógépeit dugaszolt kapcsolótáblákkal vagy gépi kódban programozták. 1946 végén jelentek meg a mnemonikus kódok, majd 1947-ben a szimbolikus nyelvek, az autokódok, amivel a programírás hatékonysága ugrásszerűen megnövekedett. Magas szintű programozási nyelvek készítésére 1950 után tették meg az első erőfeszítéseket. Az IBM az 1950-es években John Backus vezetésével létrehozott egy, a nyelvek fordításának lehetőségeit kutató csoportot. 1954-ben tették közzé annak a nyelvnek a leírását, amelyet FORmula TRANslator néven fejlesztettek és amely FORTRAN 0 [Wex 81] néven vált ismertté. A FORTRAN 0 még nem a mai FORTRAN nyelv, mivel például a feltételes utasítás `IF` ⟨*feltétel*⟩ `TRUE, FALSE` alakú volt, ahol a `TRUE` és a `FALSE` két utasításcímkét helyettesített. A változók nevét két karakterre korlátozták, ami ekkor újításnak számított. A napjainkban ismert FORTRAN nyelvet [Sam 69] 1957-ben mutatták be először, és szinte azonnal népszerűvé vált – ebben szerepet játszott az is, hogy az IBM támogatásával a fordítóprogramhoz ingyen hozzá lehetett jutni.

A programozási nyelvek fejlődésében azonban az ALGOL 60 [Dij 62] lett az igazi áttörés, ahol megjelentek a mai nyelvek főbb tulajdonságai. A tervezők

- bizonyos azonosítókat fenntartottak és ezeket kiemelték, amivel egy jól olvasható nyelvet hoztak létre,
- kikötötték, hogy minden változó típusát explicit módon kell megadni,
- általánosabb iterációs utasítás alkalmazására adtak alkalmat,

- bevezették a blokkszerkezetet, megengedve a szoros társítást a változók és az azokat használó programszövegek között,
- megengedték karakter típusú állandók használatát,
- megvalósították a rekurzív eljárások kezelését,
- bevezették a név szerinti hívást, az értékparamétereket és a dinamikus tömbhatárokat.

Az ALGOL 60 fejlesztésében is részt vevő John McCarthy az MIT-n később kifejlesztett egy merőben új szemléletű, szimbolikus kifejezések egyszerű használatát segítő nyelvet, a LISP-et (LISt Processing) [Sam 69], amely a mesterséges intelligencia kutatásában meghatározó szerephez jutott. A nyelv újdonsága volt a műveleti jelek előtagként való használata (prefix jelölés), illetve a számítási hulladékok összegyűjtése. A tömör és tiszta szemantika és a rekurzív feltételes kifejezések használata hosszú távon biztosította a nyelv népszerűségét.

1964-ben a gépidő jobb kihasználása érdekében a Dartmouth Egyetem két professzora elkészítette a BASIC (Beginners All-purpose Standard Instruction Code) megvalósítását. 1975-ben jelent meg a Tiny BASIC, mely egy 4 kB memóriájú, Intel 8080 alapú gépen (Altair) működött. 1994-ben megjelent a Visual BASIC, amely a Microsoft alkalmazások bővítőnyelve lett [Wex 81].

A hatvanas évek elejére egyre több magas szintű programozási nyelv jött létre és vált általánosan elérhetővé. A számítógépek használata így mind szélesebb körben terjedhetett el a műszaki, tudományos, üzleti és adatfeldolgozási feladatok megoldásában.

1.1.2. A szoftverfejlesztés kialakulása

Az IBM-nek az 1960-as években indított nyelvtervezési kampánya fokozatosan meghozta az eredményét. Elkészült a PL/I nyelv [Luc 69], amelyet az IBM 360-as gépcsalád nyelvének terveztek. A nyelv egyik újdonsága a kivételkezelés volt, ami lehetővé tette a normális végrehajtás mellett a vezérlésátadást például alul- vagy túlcsordulás esetén, illetve nullával való osztásnál (részletesebben lásd még a 8. fejezetben). A másik új nyelvi elem a több feladat egyidejű kezelése (multitasking), ami a párhuzamosság támogatásának első formája volt [Sam 69, Wex 81].

Először az amerikai hadügyminisztérium fogalmazta meg a beágyazott számítógéprendszerek programozását segítő nyelvek fejlesztésének igényét. Az új kívánalmaknak megfelelő programozási nyelv vázlatát STRAWMAN-nek (szalmabáb) nevezték el. A nyelvtervezetet később többször felülvizsgálták, és a végső változat (IRONMAN, azaz vasbáb) megvalósítására pályázatot írtak ki, amelyet a Cii-Honeywell Bull csoport nyert meg. A csoport által javasolt nyelvtervezetből (STEELMAN, azaz acélbáb) származó nyelvet Ada [Don 80] névre keresztelték. Az Ada az 1980-as évek közepének legfejlettebb nyelve lett, bár szerepe ma nem akkora, mint várták.

Dennis Ritchie 1972-ben fogott neki a C nyelv kidolgozásának, melynek legfontosabb tervezési szempontja a hatékonyság volt. 1980-ban Bjarne Stroustrup bevezette a C-be az osztály fogalmát, és ezzel létrehozta a C++ nyelvet [Sam 69]. 1983-ban a Microsoft és a

A korai években		
Alkalmazás	**Fontosabb nyelvek**	**További nyelvek**
Üzleti	COBOL	Assembly
Tudományos	FORTRAN	ALGOL, BASIC, APL
Rendszer	Assembly	JOVIAL, Forth
Mesterséges intelligencia	LISP	SNOBOL
Napjainkban		
Alkalmazás	**Fontosabb nyelvek**	**További nyelvek**
Üzleti	COBOL, C++, Java	C, PL/I, 4GLs
Tudományos	FORTRAN, C++, Java	
Rendszer	C, C++, Java	Ada, Modula, Assembly
Mesterséges intelligencia	LISP, Prolog	
Kiadói tevékenység	TEX, PostScript, szövegszerkesztők	
Végrehajtás	UNIX shell, Tcl, Perl, JavaScript	AWK, Marvel, SED
Új paradigmák	ML, Smalltalk	Eiffel

1.1. táblázat: Különböző alkalmazási területeken leggyakrabban használt programozási nyelvek

Digital Research elkészítette az első C fordítót mikroszámítógépre és ugyanezen év júliusában Rick Mascetti kidolgozta a C++ első megvalósítását.

Az 1990-es évek elején a már meglévő nyelvek szabványosítása került előtérbe. Megszülettek a nyelvek ANSI (az American National Standards Institute nevének rövidítéséből) változatai, amelyek a rengeteg géptípus között teszik hordozhatóvá a programokat, új nyelvi elemként pedig megjelentek az objektumorientált megközelítést követő programozási elemek.

Az 1990-es évek elejére egyre valószínűbbé vált, hogy a következő nagy terület, amelyre a mikroprocesszorok komoly befolyást gyakorolnak, az intelligens elektronikus eszközök lesznek. A Sun Microsystems-nél ezt felismerve létrehozták az úgynevezett Green projektet, amelynek egyik eredménye egy C és C++ alapú programozási nyelv, amelyet alkotója, James Gosling, az irodája ablakából látható tölgy után Oaknak (angolul tölgy) nevezett el. Később kiderült, hogy már létezik ilyen nevű programozási nyelv, így a nyelvet Javara keresztelték át. Az intelligens elektronikus eszközök piaca nem fejlődött olyan gyorsan, ahogyan azt eleinte hitték, ezért 1993-ban a Green projektet már le akarták állítani, amikor a Világháló (World Wide Web) robbanásszerű fejlődésnek indult. A Sun szakemberei azonnal felismerték a lehetőséget, hogy a Java segítségével az addig statikus internetes lapokat dinamikus tartalommal lehet ellátni. Először a Netscape, majd a Microsoft is beépítette a Java-támogatást internetböngészőjébe, ami tovább segítette a Java elterjedését. Így a Java mai sikere nem kis részben a Világhálónak köszönhető. A Sun Microsystems a kezdetektől fogva kihasználta az Internetben rejlő lehetőségeket a nyelv elterjedésének érdekében. Ezt példázza az is, hogy a programíráshoz szükséges fejlesztői környezet ingyenesen letölthető [Pra 01].

Az 1.1. táblázatban áttekintést adunk arról, hogy a különböző alkalmazások a több száz programozási nyelv közül melyiket használták, illetve használják napjainkban is.

1.2. A programozási nyelvek szerepe

A programozási nyelvek létrehozásának alapvető célja a programok hatékony végrehajthatóságának biztosítása. Akárcsak a természetes nyelvek, a programozási nyelvek is folyamatosan keletkeznek, fejlődnek és egyszer feledésbe merülnek. A kezdeti időkben használt APL, PL/I és SNOBOL4 napjainkra szinte teljesen eltűntek. A 1960-as években sikeres ALGOL helyét a 1970-es években átvette a Pascal [Hoa 74a], melyet később a C++-szal helyettesítettek. Néhány, az 1970-es években kifejlesztett nyelvet napjainkban is használnak, de időszakonként átdolgozták és új elemekkel egészítették ki őket. Az újabb nyelvek (például C++, Java, ML) egyértelműen tükrözik az évek során a nyelvek százainak használatából és tervezéséből leszűrt tapasztalatokat.

A következőkben a programozási nyelveket alapvetően befolyásoló tényezők közül sorolunk fel néhányat:

A számítógép adottságai • A számítógépek az 1950-es évekbeli kicsi, lassú és költséges elektroncsöves gépekből napjainkra olcsó és hatékony szuperszámítógépekké fejlődtek. Egyidejűleg operációs rendszer helyett szoftverfelületeket illesztettek be a programozási nyelvek és a gép hardvere közé.

Mind a hardver fejlődése, mind az újabb szoftverfelületek lényegesen befolyásolták a magas szintű nyelvek felépítését és a beépített tulajdonságok felhasználásának költségét.

Alkalmazások • A számítógépek költségeinek csökkenésével azok felhasználása gyorsan túlnőtte a kezdeti katonai, tudományos és ipari alkalmazásokat. Ezáltal igény jelentkezett a számítógépes játékokra, valamint az emberi élet szinte valamennyi területét átölelő számítógépes alkalmazásokra.

Programozási módszerek • A programozási nyelvek fejlődését nagyban befolyásolta a programozási módszerek és programozási környezetek fejlődése.

Végrehajtási módszerek • A végrehajtási módszerek fejlődése befolyásolta a nyelvekben támogatott sajátosságok megválasztását.

Elméleti tanulmányok • A formális matematika módszerei lehetőséget adnak a programozási nyelvek felépítésének és működésének elméleti vizsgálatára. Ezen kutatások elmélyítették a különböző nyelvi elemek és tulajdonságok megértését és így jelentősen hozzájárultak a nyelvek felépítésének, az azok által támogatott tulajdonságoknak és lehetőségeknek az optimalizálásához.

Szabványosítás • A különböző számítógépek és rendszerek szükségessé tettek olyan szabványosított nyelveket, amelyeket könnyen lehet különböző rendszerekre telepíteni és amelyek megengedik a programok egyik gépről a másikra való átvitelét.

Az 1.2. táblázatban röviden összefoglaltuk a 20. század második felének legfontosabb nyelveit és az egyes korokat meghatározó technikai tényezőket.

Kor	Hatás és új technológia
1951–1955	*Hardver:* vákuum-csöves gépek *Módszer:* Assembly, alapvető koncepció: alprogram, adat struktúra *Nyelvek:* a kifejezés fordítók kísérleti használata
1956–1960	*Hardver:* mágnes szalagos tárolás, alapmemória, tranzisztoros áramkörök *Módszer:* BNF nyelvtan, kódoptimalizáció, értelmezők, dinamikus tároló eljárások, lista-feldolgozás *Nyelvek:* FORTRAN, ALGOL, LISP
1961–1965	*Hardver:* kompatibilis környezetek, mágnes lemezes tárolás *Módszer:* szintaxis-vezérelt fordítók *Nyelvek:* COBOL, ALGOL 60, SNOBOL, JOVIAL
1966–1970	*Hardver:* növekedő méretek és sebesség, csökkenő költségek, mikroprogramozás, integrált áramkörök *Módszer:* idő megosztásos rendszerek, fordítók optimalizációja *Nyelvek:* APL, FORTRAN 66, COBOL 65, ALGOL 68, SNOBOL4, BASIC, SIMULA 67, ALGOL W
1971–1975	*Hardver:* miniszámítógépek, félvezető memóriák *Módszer:* programbizonyítás, strukturált programozás *Nyelvek:* Pascal, COBOL 74, PL/I (standard), C, Scheme, Prolog
1976–1980	*Hardver:* mikroszámítógépek, párhuzamos számítás *Módszer:* adat absztrakció, formális szemantika, valós idejű programozási módszerek *Nyelvek:* Smalltalk, Ada, FORTRAN 77, ML
1981–1985	*Hardver:* személyi számítógépek, munkaállomások, videójátékok, lokális hálózatok, ARPANET *Módszer:* objektumorientált programozás, interaktív környezetek, szintaxis-vezérelt szerkesztők *Nyelvek:* Turbo Pascal, Smalltalk-80, Prolog használata, Ada 83, PostScript
1986–1990	*Hardver:* Mikroszámítógépek éve, RISC, Internet *Módszer:* Kliens/szerver *Nyelvek:* FORTRAN 90, C++, SML
1991–1995	*Hardver:* Nagyon gyors nem drága munkaállomások és számítógépek, párhuzamos architekturák, hang, videó, fax, multimédia *Módszer:* nyílt rendszerek *Nyelvek:* Ada 95, Tcl, Perl, HTML
1996–2000	*Hardver:* számítógép olcsó, mindennapi eszközzé válik, WWW, kábel alapú otthoni hálózatok, gigabájtos tárolók *Módszer:* elektronikus kereskedelem *Nyelvek:* Java, JavaScript, XML

1.2. táblázat: A programozási nyelveket befolyásoló néhány tényező.

1.3. A programozási nyelvek tervezésének alapelvei

Bár az eddig említett nyelvek mindegyikének vannak hátrányai, mégis sikeresek lettek több száz más nyelvvel szemben, amelyeket megterveztek, egy ideig használták őket, majd feledésbe merültek. A nyelvek sikerének, illetve kudarcának külső, a nyelvtől független okai is lehetnek. Az Egyesült Államokban például a kormány erősen támogatta a COBOL és az Ada nyelvek használatát. A FORTRAN sikerében szerepet játszik, hogy az IBM nagy erőfeszítéseket tett a nyelv hatékonyabb megvalósításának és részletes dokumentációjának elkészítésére. A LISP és a Pascal sikere a gyakorlati használaton kívül annak is köszönhető, hogy széles körben használták őket a programozási nyelvek elméletének tanítására.

1.3.1. A „jó" programozási nyelv tulajdonságai

A külső befolyásoló tényezőkön kívül végső soron azonban a programozók határozzák meg, hogy mely nyelvek lesznek sikeresek, és melyek nem. Egy programozási nyelv alkalmazhatóságának számos objektív feltétele és szempontja létezik, amelyeket a nyelv tervezésénél mindenképpen figyelembe kell venni. A következőkben felsorolást adunk a programozási nyelvekkel kapcsolatban felmerülő leggyakoribb elvárásokról. A helyes programtervezés alapelveit először Barbara Liskov fogalmazta meg 1973-ban [Lis 73]. Az alábbiakban ezen alapelveket tárgyaljuk a programozás témakörében nyert újabb tapasztalatok tükrében.

Világosság, egyszerűség és egységesség

A jó programozási nyelvek fogalmi keretet szolgáltatnak az algoritmikus gondolkodásnak, azaz a nyelvnek a konkrét kódolási szakasz előtt is segítenie kell a programozót. A nyelvnek egy egyszerű, világos és egységes fogalomhalmazt kell biztosítania, amelyet a programfejlesztés során építőkövekként lehet használni.

A nyelvek szintaxisa (a nyelvi elemek sorbarendezési szabályainak összessége) nagymértékben leegyszerűsítheti a programok írását, tesztelését, későbbi újramegértését és módosítását. A programok olvashatósága alapvető fontosságú. Bizonyos nyelven (például APL) írt programok kódja például olyan összetett és nehezen érthető, hogy néhány hónap elteltével maga a program készítője is nehezen tudja átlátni azt. A nyelvtan másik kulcsfontosságú követelménye, hogy a nyelvi elemek jól elkülöníthetők legyenek. Lássunk egy jellemző példát PL/I nyelven e szabály megsértésére:

```
A=B=C;
```

Az utasítás nyilvánvaló értelmezése lenne, hogy `B` változónak adjuk `C` értékét, és `A`-nak `B` értékét, vagyis hogy a példa egy többszörös értékadó utasítást jelentene. Egy ilyen többszörös értékadó utasítást azonban a PL/I nyelvben nem így, hanem úgy kellene írni, hogy `A,B=C;`. A fenti példa utasítása valójában az `A=(B=C);` feltételes értékadást jelenti, amely `A` változónak igaz vagy hamis értéket fog adni, attól függően, hogy `B` és `C` egyenlők-e vagy sem. A nyelvtervezés hibája az ilyen megtévesztő utasítások használata.

Ortogonalitás

Az ortogonalitás a nyelvek azon tulajdonsága, hogy a nyelv néhány alapvető tulajdonsággal rendelkezik. Ezen tulajdonságok mindegyike külön-külön is érthető, de együttes használatukkor is értelmes kifejezéseket kapunk, mivel a kifejezések nem hatnak egymásra. Példaként tegyük fel, hogy egy nyelv tartalmaz egy kifejezést, amely létrehoz egy értéket és egy feltételt, amely egy tetszőleges kifejezésről igaz vagy hamis érteket ad vissza válaszul. Ez a két tulajdonság ortogonális, ha a kifejezést használhatjuk és kiszámíthatjuk a feltételes kifejezésben.

Ha a nyelv tulajdonságai ortogonálisak, akkor a nyelv könnyen tanulható lesz és könnyű programot írni benne, mivel kevesebb kivételt kell megtanulni, és kevesebbre kell emlékezni. Az ortogonalitásnak azonban negatív hatása is van, mivel a nyelvek fordítóprogramjai logikailag zavaros vagy kevéssé hatékony kifejezéskombinációkat is hiba nélkül lefordítanak.

Az alkalmazások természetessége

A programozási nyelvnek olyan nyelvtanra van szüksége, amely megengedi és támogatja, hogy a program felépítése tükrözze a programot alkotó algoritmus logikai összefüggéseit. Ideális esetben a programtervezés eredményét közvetlenül a programozási nyelven lehetne értelmezni. Például a szekvenciális, párhuzamos és logikai algoritmusok eltérő szerkezetekkel ábrázolják a programokat.

A tökéletes nyelvnek tehát az alkalmazásnak megfelelő adatszerkezeteket, műveleteket, ellenőrző szerkezeteket, és ezekhez kapcsolódó természetes nyelvtant kellene követnie. Sok ezer speciális programozási nyelv elkészítését éppen ez az igény ösztönözte. A Perl nyelv például jelentősen megkönnyíti a programozó munkáját a szövegfeldolgozást igénylő alkalmazásokban azáltal, hogy a hasítótáblák létrehozása és kezelése rendkívül egyszerű.

Az absztrakció támogatása

Az alkalmazáshoz legtermészetesebben illeszkedő nyelveknél is tekintélyes rés marad az absztrakt adatszerkezetek és a nyelvben létező adattípusok és műveletek között. A C nyelv alkalmas lehet órarend-beosztást készítő program írására, de az absztrakt fogalmak (hallgató, előadás, tanár stb.) és az absztrakt műveletek (hallgató beosztása adott előadásra, előadáshoz terem rendelése) bizonyosan hiányoznak.

A programozói munka alapvető feladata megtervezni, majd megvalósítani a feladat megoldásához szükséges absztrakt adattípusokat és műveleteket. Ideális esetben a nyelv olyan adatszerkezeteket és hozzájuk tartozó műveleteket nyújt, amelyek segítenek meghatározni és elkészíteni a program által megkívánt absztrakt adatokat és műveleteket. Mind az Ada, mind a C++ nyelvek tervezésénél fontos szempont volt, hogy a korábbi nyelveknek (például Pascal és C) az absztrakció terén mutatkozó hiányosságait ellensúlyozzák.

Megbízhatóság

A programok minőségének fontos összetevője a programok megbízható működése. Több különböző módszer létezik annak igazolására, hogy a program helyesen hajtja végre az igényelt feladatokat. A programok helyes működésének ellenőrzése lehet az „asztali" ellenőrzés (a programszöveg olvasása és ellenőrzése) illetve a tesztelés, amikor különböző bemenő adatok mellett ellenőrizzük a kimenő értékek helyességét. Utóbbi esetben is szükség van azonban a programozó „kézi" beavatkozására, hiszen a legtöbb esetben az automatikus ellenőrzés csak a hiba tényét tárja fel, annak forrását a programozónak kell megtalálnia.

A nyelvek egyszerű nyelvtana és szemantikája nagyban megkönnyítheti a programok helyes működésének ellenőrzését. A jó programnyelv már a hibakeresés előtt is könnyen átlátható kódot eredményez, ezáltal csökkentve az előforduló hibák gyakoriságát. Az „asztali" hibakeresés, illetve a tesztelés során észlelt hibák okának felderítése során szintén nagy segítséget jelent, ha a program szerkezete jól átlátható, így a hibák forrása könnyen felfedezhető. Egy látszólag ártalmatlan nyelvi szabály súlyos és nehezen észrevehető programhibát eredményez például a PL/I nyelvben. A PL/I minden azonosítója, amely egy `END` és egy pontosvessző között jelenik meg, automatikusan minden olyan eljárás vagy kódblokk végét létrehozza, amelyet ez a név azonosít, és bármely olyan benne foglalt programrészt is, amely nem rendelkezik saját `END`-del. Ez igen nehezen észrevehető logikai hibákat eredményez, mivel ha egy `END` figyelmetlenségből kimarad, akkor a megfelelő elrendezés automatikusan lezáródik anélkül, hogy programozót erre bármi is figyelmeztetné.

A fentieken kívül további ellenőrzési lehetőség a program helyes működésének formális matematikai módszerekkel való igazolása. Bár más szinten (az ellenőrzés matematikai összetettsége, illetve az ellenőrző program bonyolultsága szintjén), de itt is nagy jelentősége van a programnyelv logikai és szintaktikai tisztaságának [Fri 01, Hor 87].

A programozási környezet jelentősége

A programozási nyelvek felépítése mellett további szempontok is befolyásolják a nyelvek hasznosságát. Megfelelő programozási környezet egy gyengébb nyelvet sikerre vihet azzal, hogy megkönnyíti a programnyelv használatát a strukturálisan megalapozott nyelvekkel szemben. Számos, a nyelv felépítésétől független tényező létezik, amely a nyelv sikerét befolyásolja. Ezek közé tartozik például a nyelv hatékony, jól dokumentált megvalósítása, speciális szerkesztők és tesztelő csomagok elérhetősége stb. A Smalltalkot például kimondottan egy programozási környezet köré tervezték, amely menükkel, eszközcsomaggal dolgozott a Smalltalk nyelven írt programokon [Fri 01, Hor 87].

A programok hordozhatósága

A magas szintű nyelvek számára fontos kritérium, hogy a program (vagy legalábbis a forráskód) gépről gépre átvihető legyen. Ezt az igen értékes tulajdonságot azonban roppantul

nehéz elérni. A gép aritmetikája, szókészlete igen nagy akadálya a géptől való függetlenségnek. Azon nyelvek (például Ada, FORTRAN, C és Pascal), amelyeknek széles körben elérhető, szabványos, számítógéptől független tulajdonsággal rendelkező leírása létezik, hatékony alapját képezhetik a géptől független programok készítésének.

A Java nyelv készítői más utat választottak: a Java rendszerfüggetlenségét a nyelv interpretált volta biztosítja. A Java programot nem az operációs rendszer, hanem mindig egy Java interpreter program, a Java Virtuális Gép hajtja végre, így nemcsak a Java forráskód, de a lefordított program is hordozható lesz. Mindennek azonban az az ára, hogy minden egyes operációs rendszerre előzetesen el kell készíteni a Java Virtuális Gépet [Fri 01, Hor 87].

A használat költsége

A programhasználat költsége számos tényező bonyolult és nehezen számszerűsíthető együtteséből adódik. Mivel azonban egy program költséghatékonysága alapvetően befolyásolja annak használhatóságát, az alábbiakban tekintsük át, milyen összetevőkből adódik egy program, illetve programnyelv költsége.

A program végrehajtásának költsége • A programozás kezdeti éveiben a legfontosabb kérdés a program végrehajtásának költsége volt. Kutatások folytak az ideális regiszterhasználat, az optimalizált kódgenerálók tervezéséről. A hatékony kódgenerálás mind a mai napig inkább művészet, mint tudomány. Lényeges technikája a hatékony tárgykód elérésének az egyszerűségre való törekvés és az, hogy a programozó minél több információt adjon meg a változókról és azok lehetséges értékeiről. Minél több az információ, a kód annál hatékonyabb, ennek viszont a programozó munkaideje látja a kárát.

Napjainkra azonban a nagy teljesítményű számítógépek használatával a programok sebességének és hatékony futásának kérdése a legtöbb esetben háttérbe szorult.

A fordítás költsége • A fordítási idő optimális esetben nem jelent jelentős költséget, mivel a program élettartamának döntő részét a már lefordított program használata adja. A fordítás időigénye jelentős tényező lehet azonban a program írása, fejlesztése, tesztelése során. Egy további speciális eset a programozó hallgatók oktatása, ahol a fordítás ideje szintén jelentős tényezővé válik, egyrészt a gyakrabban felmerülő hibák miatt, illetve azáltal, hogy a program egyszeri bemutatásra, nem pedig a lefordítás utáni tartós használatra készül.

Az 1960-as években sok kutatás folyt a nyelvi elemzőkkel kapcsolatosan. A kutatók olyan módszereket javasoltak, mint a rekurzív lebontás, a műveleti jelek kiértékelési sorrendje, az egyszerű precedencia (LR(K)). A kifinomult elemzési módszerek azonban nem jelentenek tökéletes megoldást, mivel az elemzőnek a hibás programokat is kezelnie kell [Hor 87].

A program létrehozásának, tesztelésének, használatának költsége • A problémaosztályhoz jól illeszkedő nyelvet választva a feladat megoldásának megtervezése, kódolása, tesztelése és a kód módosítása minimális programozói időt és energiát követel. A költségek ezen kategóriája a Perl nyelv esetében igen alacsony lehet, a program végrehajtása azonban jóval hosszabb ideig is tarthat, mintha más nyelven valósítanánk meg a feladatot [Hor 87].

A program fenntartásának költsége • Számos tanulmány megmutatta, hogy a hosszú ideig használt programok esetében a legnagyobb költséget nem a program tervezése, kódolása és tesztelése jelenti, hanem a program élettartama alatti fejlesztés és fenntartás. Ez magában foglalja a program élettartama alatt talált hibák kijavítását, valamint az operációs rendszerek, hardver változásai miatt szükségessé váló kódváltoztatásokat. A program teljes élettartamára számított költséget nagyban lecsökkenti egy olyan nyelv választása, amelyik megkönnyíti az évek során szükségessé váló módosításokat, javításokat.

Néhány lehetséges megoldás

A fent tárgyalt tulajdonságok leírása közben is utaltunk arra, hogy bizonyos tulajdonságok ellentmondanak egymásnak, illetve a tulajdonságok előnyeik mellett hátrányokat is magukban foglalnak. A következőkben egy-egy adott tulajdonságot mint önálló programozástechnikai szempontot vizsgálunk.

Megbízhatóság • Léteznek kifejezetten diagnosztikai, vizsgáló fordítóprogramok, amelyeket a program megbízhatóságának biztosítására fejlesztettek ki. Az ilyen fordítóprogramoknak is vannak korlátaik és nem helyettesíthetik a magában a nyelvben levő megbízhatóságot. A diagnosztikai fordítóprogramok rendszerint nem hatékonyak, különösen nagyméretű programok futtatásakor, így a valós életben lehetetlen diagnosztizáló fordítóprogramok használata. Gondot jelent továbbá, hogy a diagnosztikai, illetve a „hagyományos" fordítóprogram a nyelvnek különböző formáit fogadja el. Így előfordulhat, hogy egy program hibátlanul fut a diagnosztikai fordítóprogramon, de hibát eredményez egy másik fordítóprogramon, illetve fordítva.

Gyors fordítás • Ezen cél elérésére ajánlott módok egyike az, hogy lehetőség legyen a program különböző részeinek külön-külön történő fordítására. Ebben az esetben az egyik programozó a szoftverrendszer egyik részét fejleszti, és annak tárgykódját hozza létre. A másik programozó használhatja ezt a tárgykódot, miközben a rendszernek egy másik részét hozza létre. A „részenkénti fordítás" lehetőségét több programozási nyelv is biztosítja, többek között a FORTRAN, a C, a C++ és a Java. Az elkülönített fordítás hiánya komoly problémát jelentett az ALGOL 60 esetében: egyetlen apró változtatás esetén is újra kellett fordítani a teljes rendszert. Hosszú programoknál ez különösen jelentős időveszteséget és költségnövekedést jelent. A részenkénti fordítás lehetőségét sem sikerült minden esetben tökéletesen megoldani. Példa erre a FORTRAN-ban biztosított részenkénti fordítás, ahol egy egyszer már létrehozott tárgykódot nem lehet ellenőrizni abban az esetben, ha egy hívó abba belépett. Egy FORTRAN eljárásnak például, amelynek mind egész, mind valós paraméterei is vannak, nincs módja ellenőrizni, hogy egy hívás a megfelelő típusú értékeket, vagy akár csak a megfelelő számú értékeket szolgáltatja-e.

Hatékony tárgykód • Elérhető-e hatékony tárgykód egy optimalizáló fordítóprogrammal? Igen, de az ilyen fordítás igen lassú, és előfordul, hogy az optimalizált változat az eredetitől eltérő eredményt ad. Nincs lehetőség továbbá a programhibák nyomkövetésére, és a forrás kismértékű változtatásainak hatására az optimalizáció megszűnik.

1.3.2. A nyelvtervezés lépései

Mint az előzőekből is kitűnik, a nyelvtervezés bonyolult, többszörösen összetett feladat, ahol sok szempontot kell párhuzamosan figyelembe venni. A különböző célok és felhasználói csoportok más és más szempont szerinti választást igényelnek. A sikeres nyelvfejlesztés 10 lépése Horowitz szerint a következő [Hor 87]:

1. Válasszunk egy alkalmazási területet és néhány jól körülírt tervezési célkitűzést!
2. Válasszunk olyan kis tervezői bizottságot, amilyet csak lehet!
3. Bocsássuk ki a nyelv első változatát kisszámú érdeklődő számára!
4. Vizsgáljuk felül a nyelvdefiníciót!
5. Hozzunk létre egy fordítóprogram-prototípust!
6. Adjuk meg a nyelvi szemantika formális definícióját!
7. Vizsgáljuk felül ismét a nyelvdefiníciót!
8. Hozzunk létre egy világos és egyértelmű nyelvleírást, és adjuk ki azt!
9. Biztosítsunk egy professzionális fordítóprogramot, és azt széles körben osszuk ki!
10. Írjunk világos és érthető kézikönyvet a nyelv használatáról!

1.4. Programozási nyelvek csoportosítása

A nyelvfejlesztés folyamatának fejlődésével és a programozási nyelvek számának ugrásszerű emelkedésével jelentősen változtak a programkészítés alapelvei is. A különböző programozási paradigmák alapján a nyelvek módszertani-történeti csoportokba oszthatók, melyek változását az 1.1. ábra mutatja be.

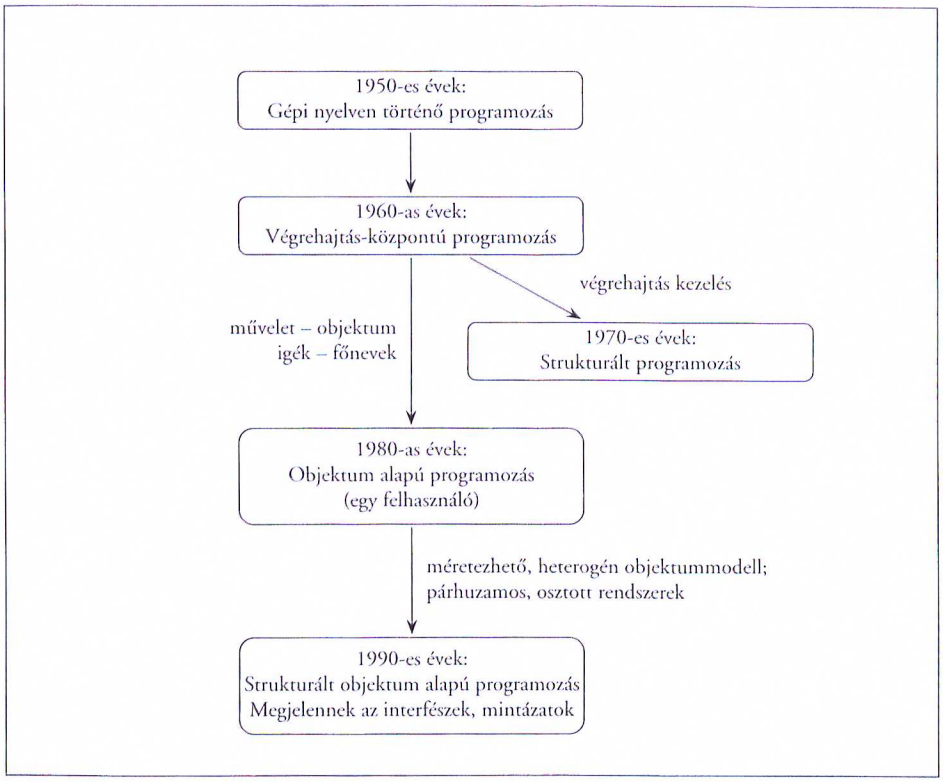
Napjaink programozási nyelveit négy alapvető számítási modell jellemzi: az imperatív, az applikatív, a szabály alapú, illetve az objektumorientált programozás. A következőkben ezeket ismertetjük röviden [ACM 89].

1.4.1. Imperatív, procedurális nyelvek

Történetileg a legelső megközelítés, amely az 1960-as, 1970-es években volt népszerű. A z ilyen nyelven írt program utasítások és eljáráshívások sorozata, amelyek sorrendben hajtódnak végre. Az állítások végrehajtása egy vagy több érték megváltozását jelenti a memóriában. Az imperatív nyelveken megírt programok szerkezete a következő általános formában írható le:

```
utasítás1;
utasítás2;
...
```

Az utasítások paramétereik segítségével hatnak egymásra, a változóban való érték tárolásával a benne tárolt előző érték elvész. A programozás így megfelelő memória-állapotok létrehozását jelenti, amely végül a megoldáshoz vezet. A legtöbb modern procedurális nyelv tartalmaz továbbá számos funkcionális elemet is. Procedurális nyelv többek között a C, a C++, a FORTRAN, az ALGOL, a PL/I, a Pascal és a COBOL [Fri 01].



1.1. ábra: Programozási megközelítés fejlődése [Weg 87]

1.4.2. Applikatív, funkcionális nyelvek

A funkcionális programozás alapelvét követve a funkcionális nyelvet tisztán függvényekre alapozzuk, műveletenként csupán a függvény-összehasonlítást, a rekurziót és a feltételes kifejezést használva. A függvény értékeit kizárólag az argumentumok értékei határozzák meg, így a függvény hívása meghatározott argumentumok használatával mindig ugyanazt az értéket hozza létre. Ezt a tulajdonságot hivatkozási átláthatóságnak nevezzük. A programok formája általában a következő:

$$\texttt{függvény}_n(\ldots\ \texttt{függvény}_2(\texttt{függvény}_1()))$$

A paraméterek kiértékelése szempontjából a funkcionális nyelveket két csoportra osztjuk. A régebbi (LISP, T, Scheme) funkcionális nyelveknél a hívás szerint történik a paraméterek kiértékelése, míg az újabb logikai nyelveknél (ML, Miranda) esetén igény szerint („lusta" kiértékelés). A program objektumok, típusok és függvények statikus definíciójának nem rendezett sorozata. A „futtatható program" helyett gyakori a „kiértékelendő kifejezés" szóhasználat (részletesebben lásd a 16. fejezetben).

1.4.3. Szabály alapú, logikai nyelvek

A logikai nyelvek interaktív nyelvek, amelyek számításaikat a formális logika és a halmazelmélet szabályait követve végzik. A logikai program alapszerkezete a kifejezés. A kifejezés lehet állandó, változó vagy összetett kifejezés. A szabály egy függvénynevet és egy vagy több kifejezést, mint paramétert tartalmaz. A tények 0 változós szabályoknak tekinthetők. A logikai program szabályok véges készlete. A program feladata egy olyan eset keresése, amely az adott kérdésre a programból logikailag következik. A logikai nyelven írt programok formája általában a következő:

```
megengedő feltétel1 -> akció1
megengedő feltétel2 -> akció2
                ...
megengedő feltételn -> akción
```

Ebben az osztályban a legismertebb nyelv a Prolog, de más nyelvek (például HASL, FUNLOG, Templog, Uniform) is léteznek, amelyek a szabály alapú logikai megközelítést használják. A programozás a lehetséges feltételek mátrixának felépítéséből és a megfelelő cselekvés megadásából áll. A BNF elemző és a YACC (Yet Another Compiler Compiler) is ezt a módszert használják [Fri 01].

1.4.4. Objektumorientált nyelvek

Az objektumorientált nyelvekben a program egymással kölcsönhatásban álló objektumok összessége. Az objektumok típusosak, és magukban foglalják a típusazonosítóikat. Minden függvénynek többféle definíciója lehetséges, melyeket ebben a terminológiában metódusoknak nevezünk (részletesebben lásd még a 12. fejezetben). Az objektumorientált megközelítés használata a programozási nyelvekkel szemben újabb és újabb követelményeket támasztott. Történetileg a megközelítés alkalmazásával először az objektum alapú, majd az osztály alapú, és végül az objektumorientált nyelvek csoportja jelent meg [Weg 87, Weg 89].

Objektum alapú nyelvek • Az objektum alapú nyelvek támogatják az objektumot, mint nyelvi eszközt. Az osztály, mint fogalom még nincs kialakítva. Az objektumokat modulokkal (csomagokkal) lehet ábrázolni. Ebbe a csoportba tartozik az Ada és a Modula.

Osztály alapú nyelvek • A fejlődés következő lépése az osztály alapú nyelvek (például CLU) kialakulása, amelyekben megjelennek az osztályok, és ezek lesznek a modularizáció alapjai is, nincs azonban osztályhierarchia.

Objektumorientált nyelvek • Napjainkban a legfontosabbak az objektumorientált nyelvek, ahol az objektumok osztályokhoz tartoznak és az osztályok közötti hierarchiát az öröklődés határozza meg, mely nyelvi elemként is létezik. Ebben a csoportba tartozik a SIMULA, a Smalltalk, az Owl és a Java.

Aktor alapú nyelvek • Az elosztott rendszereken történő objektumorientált programozásban általánosan használt objektummodell az aktor modell. Az aktor modellben a központi fogalom az objektum, az osztály fogalma hiányzik. Az objektumok leírása egy kódtörzs, ami a beérkező üzenetekre adandó válaszokat definiálja. Az aktor modellben az üzenetváltás aszinkron és egyirányú, nincs implicit szinkronizáció. A változók elérése indirekt módon történik, azaz a változókat – a metódusokhoz hasonlóan – üzeneteken keresztül tudjuk megváltoztatni, illetve az értéküket lekérdezni. Az aktor számítási modellt használó nyelveket aktor alapú nyelveknek nevezik. Ebbe a csoportba tartozik többek között az ACT-1, a Rosette és a SALSA.

1.5. A programozási nyelvek szabványosítása

A programozási nyelvek széles körben való elterjedése, a nagy biztonságot követelő (például banki, katonai) alkalmazások, a programok hosszú élettartama és a különböző fejlesztői felületek párhuzamos léte egyre jobban igényelték a programozási nyelvek szabványosítását. A szabványok jelentőségét és azok ismeretének szükségességét jól példázza az alábbi C programkód:

```
int i; i = (1 && 2) + 3;
```

A fenti kód a legtöbb nyelvben hibát eredményezne az egész és a logikai típusok keveredése miatt. A C nyelvben azonban nem létezik külön logikai típus, az igaz-hamis megkülönböztetés az egész típuson belül történik úgy, hogy a hamis állítás 0, az igaz állítás 1 (vagy bármely nem 0) értéknek felel meg. A fenti példa tehát érvényes C kód, amely *i*-nek 4-es értéket ad.

A fentiekhez hasonló kérdések tisztázására a következő általános megoldások állnak rendelkezésre:

1. Olvassuk el a változók és értékadások definícióját a nyelv kézikönyvében.
2. Írjunk egy egyszerű programot, amely teszteli a fenti utasítást.
3. Nézzük meg a nyelv szabványos leírásában a meghatározásokat és kiértékelési szabályokat.

Hasonló problémák megoldására valószínűleg a második megoldási módszer a legnépszerűbb. Mind az első, mind a második megoldás hátránya azonban, hogy a kapott válasz a nyelv egy adott megvalósításához van kötve, így nem lehetünk biztosak abban, hogy más fordítót használva is ugyanazt az eredményt kapjuk. A probléma megoldását a nyelvek szabványos leírása jelenti. Napjainkban a legtöbb programozási nyelv rendelkezik szabványos definícióval, amelyhez minden fordítónak ragaszkodnia kell. A szabványok alapvetően két csoportra oszthatók:

Szabadalmaztatott szabványok • Ezek egy vállalat vagy cég által kifejlesztett és birtokolt nyelvek szabványos leírását adják meg. Ebben az esetben a szabadalmi jogvédelem az, ami kikényszeríti a szabvány mindenki által való következetes és egyértelmű használatát.

Konszenzus szabványok • Olyan szabványos leírások, amelyeket bizonyos szervezetek a résztvevő felek egyetértésével adnak ki. Miután a konszenzus szabványok a felek közti párbeszédben, azok önkéntes és aktív hozzájárulásával alakulnak ki, a konszenzus szabványok lényegesen kisebb ráfordítással képesek megteremteni az egyes nyelvek egységes használatát. Ezek a szabványok képezik az alapját a legtöbb nyelv különböző megvalósításai közti egyöntetűségnek.

Minden ország rendelkezik egy vagy több olyan szervezettel, amely szabványok létrehozására jogosult. A legismertebb ilyen szervezetek az egyesült államokbeli American National Standards Institute (ANSI), a National Institute of Standards and Technology (NIST) és az Institute of Electrical and Electronic Engineers (IEEE), valamint az angol British Standards Institute (BSI) és a nemzetközi International Standards Organization (ISO). Ezek a szervezetek gyakran együttesen dolgoznak egyes szabványok kidolgozásán [Cez 95].

A szabványok kialakítása hasonló folyamatot követ a különböző szervezetekben. Egy adott ponton a szervezet eldönti, hogy egy adott nyelvnek szüksége lenne szabványos leírásra. A szabvány alapvető körvonalait egy önkéntesekből álló munkacsoport dolgozza ki, majd azt egy érdekelt csoport elé terjesztik, akik megszavazzák a javaslatot. A folyamat során felmerülő nézeteltérések egyben lehetőséget teremtenek a szabvány használatával kapcsolatos problémák felismerésére.

Bár a folyamat elméletileg jó, a szabványkészítés során a szakmai szempontok mellett gyakran politikai szempontokat is figyelembe kell venni. A nyelv fordítóprogramjait áruló cégnek például jelentős érdekeltsége van a szabványosítás folyamatában, pontosabban abban, hogy a végső szabvány minél jobban hasonlítson az általuk forgalmazott fordító által használt nyelvdefinícióhoz.

A szabványkészítés során szükséges megegyezések jelentőségét jól illusztrálja a FORTRAN nyelv 1977-es szabványosítása. A szabványnak meg kellett volna határoznia, hogy egy szövegből hogyan lehet kiválasztani egy azon belüli karakterláncot (például az `M="abcdefghi"` szövegből a `"cdef"` részt). Bár szakmai szempontból a felmerülő lehetőségek (például `M[3:6]`, `M[3:4]` stb.) bármelyike megfelelő lett volna, a feleknek nem sikerült egyöntetűen megegyezniük, így a függvény definíciója végül kimaradt a szabványból.

Annak ellenére, hogy a szabványokat szakmai szempontok mellett politikai döntések is színesítik, azok továbbra is igen hasznos dokumentumok. A szabványok optimális alkalmazásához a következőket kell figyelembe venni:

Időszerűség • A nyelvi szabvány elkészítésének időzítése a nyelv fejlődésének korai szakaszában a legmegfelelőbb. Ekkor ugyanis valamekkora tapasztalat már rendelkezésre áll a nyelv használatával kapcsolatban, ugyanakkor nincsenek még mélyen bevéződött és egymással nem kompatibilis mechanizmusok, melyeken egy szabvány később már nehezen tudna változtatni. A korai fejlődés szakaszában tehát egy szabvány nagymértékben hozzájárul az adott nyelv későbbi egységes és széles körű elterjedéséhez. Ennek az ellenkezője történt a FORTRAN nyelvvel, melynek 1966-ban készült el a szabványos leírása, amikor számos nem kompatibilis fordító már létezett; vagy az Ada esetében, ahol az alapvető szabványos nyelvi definíció 1983-ban készült el, míg az első hatékony fordítóra egészen 1987-ig kellett várni.

Megfelelés • Egy program megfelel a szabványos leírásnak, ha csak a szabványban szereplő nyelvi elemeket használja. A szabványnak megfelelő fordító képes lefordítani bármilyen, a szabványnak megfelelő programot, és a kapott program mindig a helyes eredményt adja. A megfelelés definíciója nem foglalkozik azonban a fordítókba gyakran beépített kiegészítésekkel, azaz olyan tulajdonságok meglétével, amelyek nincsenek benne a nyelv szabványos definíciójában. A programozónak részletesen kell tehát ismernie mind a szabványt, mind a fordítóprogramot, hogy tudja, hogy programjának mely részei lennének képesek hibátlan tárgykódot eredményezni más fordítókon, illetve más programozók kezében is.

Az alábbiakban egy, a nem szabványos nyelvhasználat miatt kiszámíthatatlan futású Pascal programot mutatunk be:

```
program abnormalis (in, out);
    procedure S;        (* 1. pont *)
    begin
        write('Rossz');
    end;
    procedure T;        (* 2. pont *)
        procedure U;
        begin
            S           (* 3. pont *)
        end;
        procedure S;    (* 4. pont *)
        begin
            write('Jo');
        end;
    begin
        U;
    end;
begin
  T
end.
```

A fenti abnormális programnak Pascal-fordítótól függően a következő három különböző kimenete lehet:

1. A fordító „hibás program" üzenettel leáll, mivel a 3. pontban az *S* eljáráshívás a 4. pontban definiált eljárásra utal.

2. Az egymenetű fordítók azonban a 3. pontban történő *S* eljáráshívást hibásan a 2. pontban definiált eljárásnak értelmezik, annak ellenére, hogy az a láthatóság körén kívül esik (a láthatósággal részletesen a a 4. fejezetben foglalkozunk).

3. A harmadik lehetőség az, hogy a fordítás után a program – helyesen – a 3. pontban definiált *S* eljárást hajtja végre.

Az 1984-es Pascal szabvány értelmében az első (hibaüzenet) az egyedüli helyes lehetőség; a harmadik – bár ésszerű, de – szabálytalan fordítás, míg a második egyszerűen rossz. A program abnormális működését ki tudjuk azonban küszöbölni, ha a 2. pontba beillesztjük a következő sort, amely jelzi a fordítónak, hogy az S eljárás leírása később következik [Fri 01].

procedure *S;* **forward;**

Elavulás • A programfejlesztői tudás fejlődése, az újabb hardver- és programozási környezetek újabb és újabb nyelvi elemek meglétét követelik. A nyelv szabványosítása után néhány évvel a nyelvi elemek és definíciók elavultnak hatnak. Az eredeti (1966-os) FORTRAN szabvány meglehetősen elmaradott, mivel nincs lehetőség blokkstruktúrák létrehozására, nem tartalmaz típusokat és sok más hasznos nyelvi elemet.

Megoldás lehet, hogy bizonyos időközönként (5–10 évenként) felülvizsgálják és kiegészítik a nyelvi szabványokat. A megoldás azonban problémákat rejt magában, mivel a legtöbb szabvány feltétele, hogy a korábbi hasonló szabványokkal kompatibilis legyen. A nyelv így meglehetősen sok elavult elemet kell, hogy tartalmazzon. Talán ennél is súlyosabb probléma, hogy a régi megoldások hátráltathatják a hatékony programok tervezését. Lássuk példaként a FORTRAN `EQUIVALENCE` állítását.

```
EQUIVALENCE(A, I)
A=A+1
I=I+1
```

Legyen `A` valós és `I` egész. Az `EQUIVALENCE` utasítás hatására a program ugyanazt a tárolóhelyet jelöli ki mindkét változó számára. Az `A` esetében az `1` hozzáadása valósként történik, míg az `I` esetében egészként. A legtöbb számítógépen a valós és egész számok ábrázolása különbözőképpen történik, így az eredmény megjósolhatatlan.

Az ehhez hasonló problémák elkerülésére kialakult az obsolete (elavult) és a deprecated (érvénytelenített) kifejezések használatának gyakorlata. Az elavultként jelzett nyelvi elemet valószínűleg a következő szabvány nem fogja támogatni, míg az érvénytelenített tulajdonság a következő szabványban elavult jelzőt fog kapni. Így lehetőség van a nyelv használóit előre figyelmeztetni, hogy melyek azok a nyelvi elemek, amelyeket a jövőben a nyelv már nem fog tartalmazni.

1.6. Nemzetközi programozás

A világ globalizációja, a piacok nemzetközivé válása és a Világháló megköveteli, hogy a nyelvek alkalmasak legyenek különböző országok kívánalmaihoz illeszkedni. A felhasználók joggal várják, hogy a szoftver különböző változatai – bizonyos határok között – különbözőképpen viselkedjenek az aktuális felhasználó igényeinek megfelelően. Mindezek megvalósítására kidolgozták a nemzetközi programozás elvét (internationalization vagy i18n). Ez azt követeli meg, hogy a program globális része független legyen a leendő felhasználó nyelvétől és országától, az országra jellemző egyedi tulajdonságokat pedig külön adjuk meg,

például erőforrásfájlokban, adatbázisokban. Az alábbiakban felsorolunk néhány, a nemzetközi programozás szempontjából alapvető tulajdonságot [Fri 01]:

- Rugalmas szövegösszehasonlítási sorrend alkalmazása, azaz tetszőleges nyelven írt szöveget az adott nyelv ábécéjének és szokásainak megfelelően kell rendezni.
- Az országra jellemző dátum- és időformátum használata.
- A különböző időzónák kezelése.
- A helyi számformátum és pénznem alkalmazása.

1.7. A programozási környezet nyelvre gyakorolt hatása

Az előzőekben már utaltunk arra, hogy a programozási környezetek változása, fejlődése a programozási nyelveket sem hagyta hatás nélkül. A programozási környezetek általában magukba foglalnak programozást támogató eszközöket, amely szerkesztést, nyomkövetést, tesztadatok készítését teszik lehetővé. Minden egyes segédeszköz egy önálló program, amelyet a programozó a programkészítés különböző lépéseiben hívhat segítségül. A programozási környezetek elsősorban a különálló fordítást lehetővé tevő, valamint a tesztelést, nyomkövetést segítő nyelvi elemek megjelenésének fejlődésére voltak hatással.

1.7.1. Önálló fordítási egységek

A programok fejlesztésekor alapvető követelmény, hogy több programozói csoport egymással párhuzamosan tudjon egy adott feladat megoldásán dolgozni, azaz a nyelv lehetőséget teremtsen önállóan fordítható egységek létrehozására. Ekkor azonban a fordítónak nincs megfelelő információja a programegységek között megosztott adatokról. A probléma megoldására a következő három megoldás a legelterjedtebb:

1. A nyelv megkívánhatja, hogy a szükséges információkat újra deklaráljuk az adott fordítási egységben (például FORTRAN).
2. A fordító speciális fordítási sorrendet határozhat meg, azaz a meghívandó alprogramnak már lefordítva kell lennie, amikor a meghívásra sor kerül (például Ada és néhány Pascal-fordító).
3. A specifikációt tartalmazó könyvtárnak jelen kell lennie (például Java, C++).

Az önálló fordítási egységek jelenléte szükségessé tette az egyes fordítási egységekben használható nevek szabályozását is. Az egyértelmű programkód igényének kielégítésére az egyes nyelvek a következő három módszert használják:

1. Minden megosztottan használt névnek egyéninek kell lennie. A programozók feladata, hogy ezt megfelelően biztosítsák.
2. A nyelvek szigorú szabályokat használhatnak a kifejezések láthatóságára, azaz ha egy alprogramot egy másik alprogram magában tartalmaz, akkor csak a legkülső programban használt nevek lesznek más, külön fordított alprogramok számára láthatók. (Ezt az elvet követi például az Ada, a C, és a Pascal.)

3. A nevek felismerhetővé válhatnak azáltal is, hogy definíciójukat egy külső könyvtárban egyértelműen megadjuk. Az objektumorientált nyelveknél ez az öröklődés alapvető módszere. Ha egy külső alprogram által definiált osztályt visszünk át az alprogramunkba, akkor az ezen osztály által definiált objektumok is ismertté válnak. A nyelvek ilyenkor gyakran lehetőséget adnak a nevek túlterhelésére is, azaz több objektumnak is megegyezhet a neve. Mindaddig, amíg a fordító el tudja dönteni, hogy melyik az éppen hivatkozott objektum, nincs szükség semmilyen változtatásra a meghívó programban [Fri 01].

1.7.2. Tesztelés és nyomkövetés

A legtöbb nyelv tartalmaz tesztelést és nyomkövetést segítő elemeket, amelyekre a következőkben mutatunk néhány gyakori megvalósítást:

Végrehajtó nyomkövetés • A Prolog, a LISP és sok más interaktív nyelv lehetőséget biztosít bizonyos változók futás alatti nyomkövetésére. Amikor a vizsgált változó új értéket kap, a program végrehajtása megszakad és egy kijelölt alprogram (leggyakrabban kiíró utasítás) hívódik meg.

Töréspont (breakpoint) • Az interaktív programozási környezetben használt programozási nyelvek gyakran adnak lehetőséget arra, hogy program bizonyos részein töréspontot helyezzünk el. Amikor a program végrehajtása eléri a megadott töréspontot, a végrehajtást megszakítjuk, és a vezérlés visszakerül a programozóhoz. A programozó megvizsgálhatja vagy módosíthatja a változók értékét, illetve a töréspontól újraindíthatja a program végrehajtását.

Megerősítés (assertion) • A megerősítés egy olyan feltételes kifejezés, amelyet mint önálló állítást illeszthetünk a forráskódba. Ilyen lehet például a következő:

assert *(X > 0* **and** *A = 1)* **or** *(X = 0* **and** *A > B + 10)*

Az állítás egy olyan feltételt ír le, amellyel a változóknak a program e sorában rendelkezniük kell a program folytatásához. Ha a megerősítés engedélyezett, a fordító a feltétel ellenőrzésének megfelelő kódot illeszt a programba. Ha a végrehajtás közben a feltétel nem teljesül, akkor a végrehajtás megszakad és egy kivételkezelő eljárás hívódik meg. A program nyomkövetése után az megerősítést letilthatjuk és így a fordító nem illeszt ellenőrző kódot a programba. Ez a hasznos fogalom sok nyelvben létezik, többek között a C++-ban [Fri 01].

Feladatok

1. Válasszon ki egy programozási nyelvet és soroljon fel minél több olyan nem szabványos nyelvi elemet, amelyet az ön fordítóprogramja elfogad.

2. Válasszon egy rövid, imperatív nyelven (például C, Pascal vagy FORTRAN) elkészített programot és írja át úgy, hogy egy applikatív nyelvet használ!

3. Hány különböző lehetőséget tud felsorolni arra, hogy egy adott változó értékét eggyel megnövelje C programozási nyelven (például $x = x + 1$)? Gondolja át a különböző lehetőségek előnyeit és hátrányait a C nyelvtervezése szempontjából!

4. Tekintse azt az egyszerű programozási nyelvet, melynek állításai a következők (a és b tetszőleges egész változókat jelölnek):

$a = b$	az a változó értékének átállítása a b értékére
$a = a + 1$	az a változó értékének eggyel történő megnövelése
$a = a - 1$	az a változó értékének eggyel történő csökkentése
ha $a = 0$, **akkor** L	ha az a értéke nulla, akkor a program végrehajtása az L címkén folytatódik
ha $a > 0$, **akkor** L	ha az a értéke pozitív, akkor a program végrehajtása az L címkén folytatódik
halt	a program végrehajtásának befejezése

Ezen a programozási nyelven például a következőképpen néz ki az $a = a + b$ érték kiszámítása:

```
L:  a = a + 1
    b = b - 1
    ha b > 0, akkor L
    halt
```

Oldja meg az alábbi feladatokat az előbbiekben megadott nyelven:

a) Írjon programot, mely adott a és b értékekre kiszámolja az $x = a + b$ kifejezést.

b) Írjon programot, mely adott a és b értékekre kiszámolja az $x = a \cdot b$ kifejezést.

c) Írjon programot, mely adott a, b, c és d értékekre kiszámolja az $x = a \times b$ és $y = c \times d$ kifejezéseket.

2 Lexikális elemek

A különböző programozási nyelveken írt programok közös jellemzője, hogy valamilyen rögzített jelkészlet elemeiből álló jelsorozatok. Ezeknek a jelsorozatoknak a szerkezetét az adott programozási nyelv lexikális szabályai határozzák meg. Az alapvető nyelvi egységek az úgynevezett lexikális elemek, *ezekből épülnek fel a programegységek. Ebben a fejezetben megvizsgáljuk, hogy a lexikális elemek milyen jelkészletből épülhetnek fel, mennyire szabványosították ezt az egyes nyelvek esetében, mi jellemzi a programozási nyelvek azonosítóit, a használható szám-, karakter- és szövegliterálokat. Külön érdemes megemlíteni a programszövegben elhelyezhető megjegyzések formáját, hiszen ez is befolyásolhatja a program megbízhatóságát.*

A programok egy vagy több *fordítási egységből* állnak. A fordítási egységek *lexikális elemek* sorozatai[1]. A lexikális elemek megadott szabályoknak megfelelő karaktersorozatok, amelyeket egymástól határolójelek választanak el. Lexikális elem a határoló, az azonosító, a szám-, a karakter- és a szövegliterál, valamint a megjegyzés.

2.1. A jelkészlet

A programjainkban használható jelkészlet általában nem csak a programszöveg, hanem a használható adatok megadása szempontjából is fontos. A programok hordozhatósága szempontjából kiemelt jelentősége van a használható karakterkészlet szabványosításának.

A számítógépekben és a köztük lévő adatforgalomban az adatok belső ábrázolása általában úgynevezett *oktetek* segítségével történik. Egy oktet egy egész érték 0 és 255 között, amely megadható decimális, oktális vagy hexadecimális formában. Az okteteket gyakran bájtnak is hívják, de érdemes odafigyelni a különbségre: bár egy oktetet nyolc bittel (vagyis egy bájttal) ábrázolunk, többnyire nincs szükség a bitek szintjére[2].

Különböző megállapodások vannak arra vonatkozóan, hogy egy oktet, vagy oktetek egy sorozata hogyan ábrázolja az adatokat. Itt párhuzamosan élnek az oktetek, illetve az általuk tartalmazott bitek számára hivatkozó elnevezések. Például 4 egymást követő oktet (32 bit) gyakran alkot valamely szabványnak megfelelően egy valós számot, az ASCII szabvány egy, a Unicode szabvány pedig két oktetes (16 bit, 2 bájt) megvalósítást használ a karakterek kódolására.

A karakterekkel kapcsolatos információk szempontjából fontos, hogy különbséget tegyünk a *karakterkészlet*, a *karakterkód* és a *karakterkódolás* között. Ezeket a definíciókat Jukka Korpela karakterekről írt tanulmánya [Kor 02] alapján tárgyaljuk.

A *karakterkészlet* egyszerűen különböző karakterek egy halmaza. Nem teszünk fel semmit a számítógép belső karakterábrázolásáról. A készlet önmagában még rendezést sem té-

[1] A fordítási egységeket bővebben lásd a 4. fejezetben.

[2] Mindazonáltal, mivel időnként használják az „előjelbit-halmaz" (sign bit set) vagy az „elsőbit-halmaz" (first bit set) kifejezéseket, nem árt tudni, hogy ez annyit jelent, hogy az oktet értéke nagyobb, mint 127. Ezeket az okteteket időnként negatív számként értelmezik, ami a legkülönfélébb problémákat okozhatja.

telez fel a karakterek között, ezt is külön kell megadnunk. A karakterkészletet a szokásos módon úgy adjuk meg, hogy megadjuk a karakterek neveit és a karakter megjelenítését látható formában[3]. Például:

```
EXCLAMATION     !
QUESTION_MARK   ?
SEMICOLON       ;
```

A karakterkészlet megadásánál használt elnevezések, vagyis a karakterek nevei inkább a karakterekhez rendelt azonosítók, nem pedig a karakterek definíciói. A nevek többnyire csak az `'A'...'Z'` tartományba eső betűket, valamint helyközöket és aláhúzásjeleket tartalmaznak. Ugyanannak a karakternek különböző nevei lehetnek a különböző karakterkészlet-definíciókban. Általában feltételezhetjük, hogy a név sugall valamifajta általános jelentést, és utal a használat hatáskörére; mindenesetre ezzel legyünk óvatosak, mert egy karakter használati lehetőségei sokkal tágabbak lehetnek, mint amit a neve mutat.

A *karakterkód* egy leképezés, amely – gyakran táblázatos formában – kölcsönösen egyértelmű megfeleltetést ad a karakterkészlet karakterei és a nemnegatív egészek egy halmaza között. Ez azt jelenti, hogy egy egyedi számkódot, egy *kódpozíciót* rendel a karakterkészlet minden karakteréhez. A kód egészében tekinthető úgy, mint egyetlen tábla (függetlenül attól, hogy egy vagy több táblával lett megadva), melynek a kódpozíciók az indexei. A kódpozíció szinonimájaként gyakran használják a *kódszám*, *kódérték*, *kódhalmaz-érték*, vagy egyszerűen a *kód* kifejezéseket is[4].

A *karakterkódolás* egy *algoritmus* karakterek digitális formában történő megadására. Ez a karakterek kódszámainak sorozatait oktetek sorozatára képezi le. A legegyszerűbb esetben minden karakter a 0–255 intervallum egy egész értékére lesz leképezve, a karakterkódnak megfelelően, és ezt használjuk, mint oktetet. Ez természetesen csak legfeljebb 256 karakterre használható.

Noha egy karakterkód-tábla közvetetten egy karakterkészletet is meghatároz, sőt a karakterkódolást is gyakran egy karakterkód (és az általa meghatározott karakterkészlet) segítségével adják meg, logikai szinten *elsődleges* a karakterkészlet, amivel meghatározzuk karakterek egy halmazát. Ezután következik a karakterkód megadása, amivel számértékeket rendelünk a karaktereinkhez – például az ISO 10646 karakterkódban az `'a'` és az `'ä'` és a `'‰'` (ezrelék jel) karaktereknek megfelelő számkódok a 97, a 228 és a 8240. A karakterkódolás adja meg, hogy a karakterkódokat hogyan képezzük le oktetek sorozatára. Így például az ISO 10646 egy lehetséges kódolásában minden karakter kódolásához két oktetet használva a kódolási algoritmus az `'a'`, az `'ä'` és a `'‰'` karaktereknek a (0, 97), a (0, 228) és a (32, 48) oktetpárokat felelteti meg. Gondot jelenthet, hogy egyes kifejezéseket többféle értelemben is használnak, így például a *karakterhalmaz* jelentheti a karakterkészletet, de a karakterkódot is, sőt időnként valamilyen karakterkódolást is[5].

[3] Érdemes megjegyezni, hogy a karakterkészlet tartalmazhat olyan karaktereket, amelyek bizonyos megjelenítéseknél ugyanúgy néznek ki, mégis logikailag különbözőnek tekintünk. Ilyenek például a latin nagy A, a cirill nagy А és a görög nagybetűs Alfa (Α).

[4] Nem feltétlenül szükséges, hogy a karaktereknek megfelelő nemnegatív egészek egymást követő elemekből álljanak. Valójában a legtöbb karakterkódban vannak „lyukak", azaz olyan kódpozíciók, amelyeket vezérlőkarakterekhez rendelnek vagy későbbi használatra szánnak.

[5] A *charset* szót sajnos sok esetben „kódolás" értelemben használják, mely könnyen félreértéseket okozhat.

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

2.1. táblázat: Az ASCII grafikus karakterei. A legelső elem a szóköz (space).

A leggyakrabban használt nemzetközileg elfogadott és szabványosított karakterkódok az ASCII, az EBCDIC, az ISO 8859-1, az ISO 8859-2 és a Unicode. Az egyes szabványok kialakításában komoly szerepe volt annak, hogy egyre nőtt az igény a sajátos nemzeti karakterek használata iránt.

2.1.1. Az ASCII kód

Ahhoz, hogy megértsük, miért is volt az ASCII bevezetésének 1963-ban olyan nagy jelentősége, jó, ha tudjuk, hogy előtte a különböző számítógépek nem tudtak kommunikálni egymással. Minden gyártónak megvolt a saját módszere az ábécé betűinek, a számoknak és a vezérlőkódoknak az ábrázolására. „Több, mint 60 különböző módon ábrázolták a karaktereket a számítógépekben. Ez valódi Bábel torony volt." mondja Bob Bemer [Bra 99], aki tevékenyen közreműködött az ASCII kialakításában, és az „ASCII atyja"-ként is ismert.

Az ASCII, ami az *American Standard Code for Information Interchange* (amerikai szabványkód az információcserére) rövidítése, „közös nevezője" ma már olyan számítógépeknek, amelyekben egyébként semmi közös nincs. Több, mint két évbe telt, míg az Amerikai Szabványügyi Hivatal, az ANSI (American National Standard Institute) által javasolt kódkészlet kialakult. Mára ez az egyik legelterjedtebb kódolás. Annyira elterjedt, hogy sok esetben az „ASCII fájl" egyszerűen szöveges (azaz nem bináris) állományt jelent, még akkor is, ha a karakterkód valami más! Számos karakterkód részhalmazként tartalmazza az ASCII-t. Az ASCII első 32 kódpozíciója és a 127-es kód vezérlőkaraktereket (*control characters*) ír le, mint például a *soremelés* (LF) és az *escape* (ESC). A tényleges karakterkészlet, ami a nyomtatható karakterekből áll, a 2.1. táblázatban látható. (A teljes kódtábla a függelékben található a 711. oldalon.)

Az eredeti szabványban a 128–255 közötti értékeket nem használták, majd később, amikor ez a választék szűkösnek bizonyult, különböző kiterjesztéseket hoztak létre. Egy elterjedt kiterjesztés a függelékben a 713. oldalon található. Az ASCII kiterjesztésének tekinthető például az ISO 8859-1 vagy Latin-1 kódkészlet és az ISO 8859-2 vagy Latin-2 kódkészlet is. Az eredeti ASCII-re gyakran mint US-ASCII-re hivatkoznak.

2.1.2. Az EBCDIC kód

Az ASCII-n kívül nagyon elterjedt az EBCDIC (*Extended Binary Coded Decimal Interchange Code*) kódolás is, amit az IBM már a A 60-as évek óta használ. Érdekessége, hogy a 0–255 tartományban több üres hely is található, valamint az is, hogy például az `'a'...'z'`

vagy az 'A'...'Z' betűk kódjai, bár követik az ábécében megszokott rendezést, de nem feltétlenül következnek folyamatosan, egymás után. Sőt, előfordul, hogy egyéb karakterek kódjai is „beékelődnek" közéjük. Például az 'r' karakter kódja 153, ezután 7 üres hely következik, majd a 161-es kódú '~' után folytatódik az ábécé az 's' karakternek megfeleltetett 162-es kóddal. Ennek a kódolásnak legalább 6 egymástól kicsit különböző formája létezik. A talán leggyakrabban használt teljes kódtáblát a függelék 714. oldalán bemutatjuk.

2.1.3. Az ISO 8859 család

A különböző nyelvekben használt különleges karakterek iránti igény hozta létre az 1980-as évek közepén az ISO 8859-1, ISO 8859-2 stb. kódtáblákat, melyek az ASCII által nem használt 128–255 közötti értékekhez rendeltek különböző speciális – elsősorban grafikus – karaktereket.

Az ISO 8859-1 vagy Latin-1 szabvány az Európai Unió országaiban talán a legelterjedtebb 8 bites karakterkódolási szabvány. A nyugat-európai ékezetes betűk nagy részét tartalmazza, például az 'ö'-t, az 'ô'-t, a 'ð'-t, valamint még egyéb jeleket is: '©', '¿', '<', 'ï', '£' stb. Nem tartalmazza viszont – többek között – a magyar nyelvben fontos 'ő' és 'ű' karaktereket. A teljes Latin-1 táblázat megtalálható a 712. oldalon.

A különleges magyar karaktereket ('ő', 'ű') az ISO 8859-2 vagy Latin-2 szabványba vették fel. Ugyancsak itt található a közép- és kelet-európai ékezetes betűk nagy része, mint például az 'Ł', vagy az 'š'. A Latin-1-ben lévő '©', '¿', '<', 'ï', '£' stb. karaktereknek viszont itt már nem jutott hely. Nekünk, magyaroknak szerencsénk van, mert ha Latin-2 kódolással írt magyar szöveget Latin-1 kódolás szerint próbálunk elolvasni, ez – bár nem szép – de érthető lesz, csak az 'ő', 'ű' betűk helyett 'õ', illetve 'û' lesz látható. (A csehek vagy lengyelek már sokkal rosszabbul jártak.) A teljes Latin-2 táblázatot lásd a 712. oldalon.

Az ISO 8859-3 vagy Latin-3 a dél-európai nyelvek (például a máltai, de ide tartozik az eszperantó is) szükségleteit fedezi, míg az ISO-8859-4 vagy Latin-4 az észak-európai nyelvek (észt, litván, lett, grönlandi, lapp) leírásánál használt különleges betűk kódolását adja. Külön kódtábla, az ISO 8859-5, segíti a cirill betűket használó bolgár, belorusz, orosz, szerb stb. nyelveken írókat, ismét másik az alap arab jelkészlet használóit (ISO 8859-6), a görög (ISO-8859-7), a héber (ISO-8859-8) nyelv leírását stb.

Akit bővebben érdekel az ISO 8859 kódolás, figyelmébe ajánljuk Roman Czyborra *The ISO 8859 Alphabet Soup* című tanulmányát [Czy 98a].

2.1.4. A Unicode szabvány

Többnyelvű környezetekben az előző szabványok által okozott problémákat szeretné kiküszöbölni az *ISO 10646* 1993-ban bevezetett szabványa[6]. Definiálja az UCS-t (Universal Character Set), az *univerzális karakterhalmazt*, mely egy hatalmas, és egyre növekvő karakterkészlet és karakterkód. Több tízezer karaktert határoztak meg benne egységes elvek

[6] A szabvány száma emlékeztet a 646-ra, az ASCII-nek megfelelő ISO szabványra.

szerint, minden karakternek egyértelmű, szabványos nevet adtak. Tartalmazza az előzőekben említett karakterkészletek közül részhalmazként az ASCII-t és a Latin-1-et, a közép- és kelet-európai ékezetes karaktereket, a nemzetközi fonetikai ábécét (International Phonetic Alphabet – IPA), a görög, cirill, grúz, örmény, héber, arab karaktereket, az indiai földrész nyelveinek karakterkészleteit, a kínai/japán/koreai íráshoz szükséges jeleket, a matematikai műveleti jeleket, a különféle rajzolást segítő karaktereket, de még a csekkek feldolgozásánál használt speciális OCR karaktereket is[7]. A halmaz méretéből következik, hogy legtöbb esetben a karakterek más kódpozíción találhatók, mint az előző szabványokban.

A *Unicode* a Unicode Consortium szabványa, mely teljesen kompatibilis az ISO 10646-ban definiált karakterkészlettel és kóddal[8]. Meghatároz ugyanakkor kódolásokat is: kiinduláskor még azt javasolták, hogy a karakterkódok lehetséges értékeit az addigi 8 bites megvalósítás által meghatározott 0–255 értékintervallumról terjesszék ki egy 16 bites megvalósítás 0–65535 értékeire. Ezt használja például a Windows NT. Erről az eredeti elképzelésről mára kiderült, hogy sok is, meg kevés is, hiszen az első 256 karakterhez – és ezek elég gyakoriak – elég lenne 8 bit is, másrészt még a 16 bites megvalósítás sem elegendő a világban felmerülő igények kódolására. Ennek megfelelően kialakították a különböző szabványos Unicode transzformációs formátumokat, az UTF-8-at, az UTF-16-ot stb. a karaktersorozatok rendszerfüggetlen kódolására. (Bár ezeket az angol nyelvű irodalomban „charset"-eknek hívják, ez kódolási algoritmust és nem karakterkészletet jelent!) Az elektronikus levelezésben 1999 óta az Internet Mail Consortium ajánlására szabvány az UTF-8.

A gyakorlatban az ISO 10646 helyett inkább a Unicode név használata terjedt el, a különböző leírásokban általában az ISO szabvánnyal egyenértékűként használják az utóbbi nevet. A Unicode kódolásról bővebben olvashatunk Roman Czyborra *Unicode Transformation Formats: UTF-8 & Co.* című tanulmányában [Czy 98b].

2.2. A programozási nyelvek jelkészletei

A korai programozási nyelvek nagy része a lexikális elemekben még csak az angol ábécé betűit, a számjegyeket és néhány speciális karaktert engedett meg, mint például a zárójelek, a pontosvessző, stb.

Mostanában egyre nagyobb hangsúlyt kap az az igény, hogy a programok képesek legyenek kezelni a különböző nemzetek által használt speciális karaktereket is, sőt adott esetben legyen mód a karakterek rendezésének a konkrét nyelvben meghatározott ábécé-sorrend szerinti definiálására is.

A programozási nyelvek jelkészleteinek értékelésénél figyeljük meg, hogyan, milyen szinten felelnek meg a következő kérdéseknek:

- Milyen pontosan határozták meg a nyelv definíciójában a használható jelkészletet?
- Milyen jelek szerepelhetnek a programszövegben?

[7] Az egyes karaktercsoportokról részletesebben lásd Jukka Korpela összeállítását [UCS 02]. A kelet-ázsiai írások kódolási problémáiról érdemes elolvasni Steven J. Searle *A Brief History of Character Codes in North America, Europe, and East Asia* című írását is [Sea 02].

[8] Eredetileg az ISO 10646 Basic Multilingual Plane (BMP) készlettel.

Pascal

A *Pascal* 1990-es szabványa (ISO 7185:1990) még csak minimális követelményeket támaszt, mint például: „a `'0'...'9'` rendezett és folytonos kell legyen", „az `'A'...'Z'` (ha van) rendezett, de nem feltétlenül folytonos", az `'a'...'z'` ugyanígy. Nincs előírva a lehetséges karakterek halmaza sem, és megengedett, hogy különböző megvalósítások különböző kódolást használjanak [Cat 01]. Javasolják az ISO 646-t, de nem teszik kötelezővé.

Ez a hiányosság komolyan befolyásolhatja a Pascal nyelven írt programok hordozhatóságát. Gondoljuk meg, hogy ha például egy programozó ír egy ciklust, ami végigmegy `'a'`-tól `'z'`-ig az ábécé betűin, vagy összehasonlítja az `'A'` és az `'1'` kódját a karaktereken definiált rendezés szerint, akkor a programja másképpen viselkedhet, attól függően, hogy EBCDIC vagy ASCII kódolást használ-e.

Ada

Érdemes megfigyelni, milyen komoly változáson ment át az *Ada* szabványa. Már 1983-ban szigorúan rögzítették ugyan a használható kódolást, de még csak az ASCII kódra (ISO 646). Az 1995-ös szabvány előírásai szerint azonban már az Ada 95 program forrásszövegében általában az ISO-10646-1 szabványban felsorolt 65536 karakter bármelyikét használhatjuk, de – ahogy látni fogjuk – például azonosítókban csak az ISO-8859-1 szabvány által is definiált 193 grafikus karakter fordulhat elő.

C++

A *C++* 1998-as szabványa (ISO/IEC 14882:1998) – a forrásszövegben alapvetően csak 96 karaktert (a vízszintes és függőleges tabulátoron, a lapdobás, újsor és szóköz karaktereken kívül 91 grafikus karaktert), vagyis egy viszonylag szűk karakterkészletet engedélyez. Pontosan előírja ugyanakkor a karakterek kódolását: az értékeket az ISO 10646 azon részhalmazából kell venni, amely az ASCII-t tartalmazza. Ez komoly előrelépés az 1990-es szabványhoz képest, ahol még nem volt előírás a használandó kódra. Lehetőség nyílik ezen kívül úgynevezett univerzális karakternevek használatára is.

Java és C#

A *Java* és a *C#* programozási nyelvek specifikációja már megengedi, hogy a programok forráskódjában tetszőleges Unicode karakterek szerepeljenek. Új elemként jelenik meg a *Unicode szekvencia*, mellyel Unicode karaktereket ábrázolhatunk: `'\uXXXX'`, ahol minden `X` egy-egy hexadecimális számjegy. A C# tervezői már a kódolásra is tesznek javaslatot, az UTF-8 támogatását ajánlják.

A határolójelek

A legtöbb programozási nyelvben – például a Pascalban, a C++-ban, az Adában, a Javában – a határolójelek közé tartozik a szóköz, a tabulátor és a sorvége karakter.

	Pascal	C++	Java	Ada 95	CLU	Eiffel
szóköz, tab, sorvége	Igen	Igen	Igen	Igen	Igen	Igen
Egyetlen karakter	& ' ( ) + - * / : . ; < = >	& % ^ ' ( ) + - { } | ~ [ ] \ " * / : . ; , < = > ! ?	( ) { } [ ] ; , . + - * / : < = > ! ~ ? : & | ^ %	& ' ( ) * + , - . / : ; < = > |	( ) { } [ ] : ' . $ " , \ + - * / < = > ~ & |	; , : . ! = ( ) [ ] { } " + - $ % /
Karakter-sorozat	:= >= <= <> << ** ..	-> ++ -- .* ->* == <= >= && || << >> <<= >>= != .. += -= &= ^= |= :: *= /=	== <= >= << != && || ++ -- >> >>> += -= *= /= &= |= ^= %= <<= >>= >>>=	=> .. ** := /= >= <= << >> <> --	:= <= >= ~< ~<= ~= ~>= ~> || ** //	-- !! /= -> .. << >> := ?=

2.2. táblázat: Határolójelek táblázata.

A korai programozási nyelveknél – ilyen például a FORTRAN vagy az ALGOL 68 –, előfordul, hogy a szóköz nem számít határolójelnek. Nézzük meg egy rövid példán, hogy ez a látszólag apró döntés, kombinálva a változók automatikus deklarálásával, milyen komolyan befolyásolhatja a programok megbízhatóságát:

Tegyük fel, hogy a `DO 10 I=1.5` FORTRAN nyelvű kódrészletnél ciklust szerettünk volna kódolni, de a vessző helyett véletlenül pontot írtunk. Sajnos így a fordító nem jelez hibát, a nyelv szabályai alapján nem is jelezhet, hanem úgy tekinti, hogy a `DO10I` nevű változónak értékül adjuk az `1.5`-öt.

A határolójel állhat egy karakterből, vagy lehet karakterek sorozata, erre a 2.2. táblázat mutat néhány lehetőséget.

2.3. Azonosítók

A változók, típusok, alprogramok stb. neveként programjainkban *azonosítókat* használunk. Érdemes áttekinteni, hogy a különböző programozási nyelvekben milyen választ adtak a következő kérdésekre:

- Milyen karakterek használhatók az azonosítók leírására?
- Mi az azonosító megengedett szintaxisa?
- Megkülönbözteti-e a nyelv a kis- és nagybetűket?
- Van-e hosszúsági megkötés az azonosítókra?
- Vannak-e kötött szavak (kulcsszavak), melyek nem használhatók a programozó által definiált módon?

2.3.1. A használható karakterek

A legtöbb programozási nyelvben az angol ábécé betűit (`'A'...'Z'`, `'a'...'z'`) és a számjegyeket (`'0'...'9'`) használhatjuk az azonosítókban.

Ennyit és nem többet tesz lehetővé például a Pascal 1990-es ISO 7185 szabványa. Az azonosítók jobb tagolása érdekében ezt a készletet bővítették először számos programozási nyelvnél az aláhúzásjel (`'_'`) alkalmazásának lehetőségével, ahogy az Extended Pascal (ISO 10206), a CLU, az Eiffel, vagy a C++ (ISO/IEC 14882:1998) nyelveknél láthatjuk.

Különbséget jelent a megengedett szintaxisban, hogy az aláhúzásjellel a betűk halmazát bővítik, vagy további lehetséges megengedett karakterként kezelik.

Bizonyos nyelveknél speciális karakterek adhatnak további információt az azonosító által meghatározott változóról az azonosító elején vagy végén. Így például Perlben a `$`, `@`, vagy `%` jel adja meg az azonosító elején, hogy az adott név skalár, számmal indexelt, illetve asszociatív tömb-e. BASIC-ben az azonosító utolsó karaktere utalhat a változó típusára.

Az azonosítók nemzeti nyelven való megadása felé az első lépést az Ada 95 tervezői tették meg: azonosító-betűnek nevezték el az ISO 10646 BMP Row 00 azon karaktereit, amelyeknek a neve „Latin Capital Letter"-rel és „Latin Small Letter"-rel kezdődik, és ezek is használhatók, így például a `Távolság` vagy a `Sebesség` érvényes azonosítók. Ada 95 azonosítókban nem használhatók viszont az `'ő'`, vagy az `'ű'` karakterek.

Az igazi nagy lépést a Java tette meg: betűnek, illetve számjegynek tekintenek, – és így azonosítókban használhatónak – minden jelet, amely valamely nemzet számára betű, illetve számjegy, legyen az például a cirill, görög, héber, arab, kínai, thaiföldi, indiai, japán, grúz, örmény, vagy akár a magyar ábécé eleme[9]. Így a Javában érvényes azonosító az „$\alpha\beta\gamma$" vagy a `Felső` is. Ezt az elvet követik a C# azonosítói is.

2.3.2. A megengedett szintaxis

A különböző programozási nyelvekben az azonosítók leggyakrabban betűvel kezdődhetnek, s ezt betűk, illetve számjegyek követhetik. A szokásos BNF szintaxisleírás szerint a képzési szabály:

⟨*betű*⟩ {⟨*betű*⟩ | ⟨*számjegy*⟩}

Így képezhetünk érvényes azonosítókat például az ALGOL 60-ban, a FORTRAN-ban, a Pascalban, a Modulában, a C-ben vagy a C++-ban. Érdemes megemlíteni, hogy az előzőek közül a Pascalban, a C-ben és a C++-ban a betűk halmazát bővítették az `'_'` jellel. A Javában, ahogy már említettük, a betűk halmaza még sokkal nagyobb, és itt is beletartozik az `'_'` jel is. Így például Javában az `El_Niño` szabályos azonosító.

Más nyelvekben, mint például az Adában, bár a betűk halmaza itt is nagyobb, az `'_'` jel nem tartozik a betűk közé, s noha szintén használható, de csak az azonosítón belül, a hosszú nevek tagolási lehetőségeként, s két `'_'` jel nem jöhet egymás után, vagyis itt a szintaxis:

⟨*betű*⟩ { ['_'] ⟨*betű*⟩ | ['_'] ⟨*számjegy*⟩ }

[9] A teljes specifikációhoz lásd [Jav 03a].

2.3.3. A kis- és a nagybetűk megkülönböztetése

A kis- és nagybetűk megkülönböztetésének kérdése már a legkorábbi programozási nyelveknél is felmerült.

A korai nyelvek egy része (FORTRAN, COBOL) csak a nagybetűket fogadta el – a lyukkártyákra csak ilyet lehetett rögzíteni. A későbbi változatok fordítói már megengedik a kisbetűk egyenértékű használatát is, és egyszerűen nagybetűkké alakítják azokat fordítás közben.

A programozási nyelvek egyik vonulatának tervezői úgy vélték, hogy az a tervezési szempont, amely kimondja, hogy ugyanúgy megjelenített dolgoknak ugyanaz kell legyen a jelentése, megköveteli, hogy a kis- és nagybetűs írásmód ne jelentsen különbséget, tehát a *kutya*, a *KUTYA*, vagy a *Kutya* ugyanarra az egyedre vonatkozzon. Ezt a nézetet vallották például a Pascal, az Ada, vagy akár a Visual BASIC tervezői. Ez utóbbiakról érdekességként jó tudni, hogy az azonosítóikban szereplő ékezetes karakternél is ismerik a kisbetűk nagybetűs párját, tehát például *KÖRTE* és *körte* is ugyanazt jelöli.

Más tervezők – az ALGOL 60-nal és ALGOL 68-cal kezdve, a C-vel, C++-szal, Javával folytatva, – azt gondolták, hogy célszerű minden betűt különbözőnek tekinteni. A nevek esetleges keveredéséből adódó nehézségeket a C-ben még ki tudjuk küszöbölni, ha csak kisbetűs azonosítókat használunk. A Javában azonban ez a probléma már nem kerülhető meg, hiszen a szabványos könyvtárban is vegyesen alkalmazzák a kis- és nagybetűket. Például az egészeket karakterlánccá alakító művelet neve *parseInt*, és fordításkor a *ParseInt* vagy *parseint* alakokat a rendszer nem ismeri fel.

Szokatlan megoldást választott Bertrand Meyer, az Eiffel tervezője. Itt ugyanis, bár a kis- és a nagybetűk egyenértékűek, használhatjuk ugyanazt a nevet egy objektumra és a típusára, így például megengedett az *alma: ALMA* deklaráció [Mey 91].

2.3.4. Hosszúsági megkötések

A legkorábbi programozási nyelvek egykarakteres neveket használtak. Ez természetes módon adódott, hiszen a matematikában általánosan elterjedt, hogy a változókat `a`-val, `b`-vel, `x`-szel vagy `y`-nal jelöljük, és az első programozók általában matematikusok voltak.

A FORTRAN 66-ban, de még a FORTRAN 77-ben is legfeljebb 6 karakter hosszú azonosítóval dolgozhattunk, ezt a FORTRAN 90-ben legfeljebb 31 jelre növelték. Az újabb programozási nyelveknél – például a C++ 1998-as szabványában, az Ada 95-ben vagy a Javában és a C#-ban – már általában nincs hosszúsági megszorítás, az azonosító akármilyen hosszú lehet. Előfordul, hogy a fordítók adnak valamilyen korlátot, hogy a használt szimbólumtábla mérete ne legyen túl nagy, de ez az „átlagos" programozó számára többnyire nem jelent gondot.

2.3.5. A fenntartott szavak

A programozási nyelvekben általában bizonyos szavakhoz speciális jelentést rendelnek. Ezeket használjuk például a különböző utasításoknál, és segítségükkel tagoljuk a programszöveget is. Így az elágazások gyakran az `if` szóval kezdődnek, a ciklusok a `while` vagy `for` szóval stb.

Ezeket a szavakat *kulcsszavaknak* nevezzük, és a ma használatos programozási nyelvek többségében – Pascal, C, C++, Java, Perl, Ada 95, C#, Eiffel, Delphi stb. – ezek úgynevezett *fenntartott szavak* is, vagyis azonosítóként nem használhatók[10].

A fenntartott szavak hiánya áttekinthetetlen, nagyon nehezen karbantartható programokat eredményezhet, komoly értelmezésbeli problémákat okozva a nem megfelelően megírt programkód egy esetleges későbbi módosításakor. A FORTRAN-ban például, ha a REAL kulcsszó egy utasítássor elején áll és egy azonosító követi, ez azt jelzi, hogy azon a helyen egy REAL típusú változót deklarálunk. Ha ugyanezt a REAL szót az értékadás jele követi, akkor a fordító változónévnek tekinti:

```
REAL ALMA
REAL = 2.5
```

A program további részében könnyen összekeverhető a REAL szó kulcsszóként, illetve változónévként való használata [Seb 02].

Számos nyelvben vannak úgynevezett *előre definiált* nevek is, melyek valamilyen értelemben a kulcsszavak és a fenntartott szavak között helyezkednek el. Például az Ada 95 nyelvben nem nyelvi kulcsszóként, hanem a Standard könyvtár részeként adták meg az *Integer*, *Float*, *Boolean* alapvető beépített típusokat, vagy a *True*, *False* stb. neveket. Így lehetőségünk van arra is, hogy ezeket átdefiniáljuk. Az átdefiniálás lehetősége bizonyos esetekben előnyös, hiszen a

type *Integer* **is range** *–999_999..+999_999;*

használata után programunk független lesz a konkrét számítógépes környezettől. Más esetekben jobb, ha óvatosak vagyunk, félrevezető lehet például a programszöveg későbbi olvasója számára a

True: Integer;

jellegű deklaráció.

A C# érdekessége, hogy a kulcsszavak az @ bevezető jellel használhatók azonosítóként is (például: **@bool**). Ez a jel „normál" azonosítónak nem lehet első karaktere. Ezt a lehetőséget a programok hordozhatósága, illetve „átjárhatósága" miatt vezették be. Az @ előtagú szimbólumok neveit a fordító valójában @ nélkül tárolja. Így attól függetlenül, hogy egy másik nyelv nem tartalmazza például a **sealed** kulcsszót és ott egy programban azonosítóként használják, a C#-ban lehetőség van az azonosító közvetlen használatára.

[10] A szakirodalom egy részében standard szónak nevezik azt, amit mi kulcsszónak és kulcsszónak hívják a más célra nem használható, fenntartott szavakat (lásd például [Csö 96]).

2.4. Literálok

A programjainkban előforduló számokat, karaktereket és szövegeket literáloknak nevezzük.

2.4.1. Számliterálok

A számliterálokban a gyakorlatban csak a `'0'...'9'` és az `'A'...'F'` „számjegyek" szerepelhetnek, bár tagolásra bizonyos nyelvekben használhatjuk az `'_'` (aláhúzás) jelet is, amelynek a szám értéke szempontjából nincs jelentősége.

A számok szerkezete

A számliterálokban alapvetően megkülönböztetjük az egész és a valós számokat.

Egész számok • A decimális *egészek* általában számjegyek sorozatai, szintaxisuk BNF-leírása a következő:

$$[\,'+'\,|\,'-'\,]\ \langle számjegy\rangle\ \{\langle számjegy\rangle\}$$

Például: `-123` vagy `456789`. Az Ada 95-ben, a Perlben vagy az Eiffelben ezt írhatjuk a `456_789` alakban is. Az Eiffelben a szabályok még azt is előírják, hogy ha szerepel `'_'` jel, akkor *pontosan* 3 számjegy kell álljon minden `'_'` jel jobb oldalán.

A C, C++, Java vagy C# nyelvekben az **int** és **long** típusok különbségét az egész szám után írt *l* vagy *L* betű használatával jelölhetjük. Például a *2* típusa **int**, míg *2L* típusa **long** lesz. A **byte** és a **short** jelölésére nincs hasonló jelzés. Kifejezésekben ilyenkor explicit típuskényszerítés alkalmazható. Hasonlóan az *u*, illetve az *U* utótag használható az **unsigned** (előjel nélküli) típusok jelölésére.

Valós számok • A valós számokat többnyire az különbözteti meg az egész számoktól, hogy tizedespontot (`.`) tartalmaznak. A programjainkban használható *valós számok* általános szerkezete:

$$[\,'+'\,|\,'-'\,]\ \langle számjegy\rangle\ \{\langle számjegy\rangle\}\ \text{'.'}\ \langle számjegy\rangle\ \{\langle számjegy\rangle\}\ [\,\langle kitevő\text{-}rész\rangle\,]$$

A kitevőt általában egy `E` betű vezeti be, utána esetleg írhatunk előjelet, majd a kitevőértéket. Ennek jelentése: előjel × szám × $10^{\text{előjel}_k \times \text{kitevő}}$. Például a `-123.456E+3` forma a `-123456.0` értékű valós számot jelöli.

A fentiekhez majdnem minden nyelv ad valami specialitást:

Pascal • A Pascalban a kitevő rész csak valós számot jelölhet, így például *12E+3* valós, a *12000.0* értéket jelöli, bár nincs benne tizedespont [Cat 01].

Ada 95 • Az Ada 95-ben egészekre is van exponenciális alak, ahol a kitevő nem lehet negatív. Itt `1E6` az 1 000 000 értékű egész. Valósak exponenciális alakjában negatív kitevő is szerepelhet [Ada 95].

Eiffel • Az Eiffelben elegendő egy számjegy és a tizedespont, nem szükséges a tizedespont mindkét oldalára számjegy [Mey 91]. (Például `-1.`)

C++, Java • A C++-ban és a Javában a valós literál alapértelmezés szerint **double** típusú (*-12.3*, *12.3e4*) hacsak nem szerepel utánuk egy *f* vagy *F* betű, ami **float** típusúként történő deklarációt jelent (*3.141592f*, *2.9e-3f*). A **double** típust egy *d* vagy *D* betűvel ki is lehet emelni [C++ 98, Nyé 01]. Bevezetik a *NaN* (Not a Number) értéket is. Ez a *0.0/0.0* eredménye is.

Java • A Javában a *nulla* értéket előjellel is lehet deklarálni: *0.0*, illetve *-0.0*. Ezek egyenlőek ugyan, de néhány helyzetben más eredményt adnak, például: *1.0/0.0* eredménye a pozitív végtelen (*POSITIVE_INFINITY*), de *1.0/-0.0* a negatív végtelen (*NEGATIVE_INFINITY*) lesz. A számításokban a *POSITIVE_INFINITY* és a *NEGATIVE_INFINITY* a matematikában megszokott módon viselkednek: véges szám hozzáadása vagy kivonása nem változtatja meg az értéküket, a kettő szorzata pedig a *NEGATIVE_INFINITY*-t adja. Összegük viszont *nem definiált*. [Nyé 01]

A megengedett alapok

A programozási nyelvek nagy részében a valós számokat decimális alakban, de az egészeket a decimálison kívül hexadecimális, esetleg oktális, bináris alakban is meg lehet adni. Vannak olyan nyelvek is, amelyekben ezt a lehetőséget kibővítették a valós számok más alapon való megadásával, illetve más alapok használatának megengedésével is. A megengedett számjegyek halmaza természetesen a számrendszertől függ, így például az oktálisban csak a `'0'...'7'` számjegyek, a hexadecimálisban a `'0'...'9'` számjegyeken kívül az `'A'...'F'` is használhatók „kiterjesztett" számjegyként stb. Az oktális és hexadecimális megadási módot bitminták kezeléséhez ajánlják, számokként értelmezve meglepetések érhetik a programozót, ahogy ezt például a C++ és a Java nyelveknél láthatjuk is.

Pascal • A Pascalban a decimális egészeken kívül használhatunk hexadecimális egész számokat is a számjegyek elé írt *$* jel segítségével. Egy hexadecimális egész szám legfeljebb 8 jegyű lehet. Például a *+124* vagy a *−5* decimális egészek, míg a *$1*, a *$A00* vagy a *−$FFFF* hexadecimális egészek. A valósakat csak tízes számrendszerben adhatjuk meg. [Cat 01]

Ada 95 • Az Ada 95-ben a számrendszer alapszáma kettőtől tizenhatig bármelyik szám lehet, és a valós számok is felírhatók más alapú számként. Ha a számokat decimálistól különböző alapon szeretnénk megadni, ennek alapformája a következő:

⟨*alap*⟩ `'#'` ⟨*szám*⟩ `'#'` [⟨*kitevő*⟩]

A számrendszer alapszáma megelőzi a szám számjegyeit, amelyek sorozatát egy-egy # jel nyitja és zárja le. Az alapszám és a kitevő mindig decimális alakban adott [Nyé 98].

Például:

2#1011# —— *Bináris egész szám.*
16#F.FF#E2 —— *Hexadecimális valós exponenciális alak.*
2#1.1111_1111_111#E11 —— *A kitevőhöz tartozó alap a számrendszer alapszáma,* —— *tehát a két utóbbi érték egyaránt 4095.0.*

C, C++ • A C-ben vagy a C++-ban a lebegőpontos számokat csak decimálisan lehet megadni. Az egészeknél viszont a szám kezdő karakterei határozzák meg a számrendszert, amelyben azt értelmezni kell:

- A *0* kezdetű szám oktális, például: *0377*,
- A *0x* kezdetű szám hexadecimális, például: *0xFF*, *0xCOBOL*,
- Az összes többi számliterál decimális számrendszerben lesz értelmezve. Például: *255*.

Ha például az általunk használt számítógépen az `int` kettes komplemensben, 16 biten van ábrázolva, akkor számként értelmezve a *0xffff* értéke -1. Ha viszont több bitet használunk az egészek ábrázolására, akkor a *0xffff* értéke 65 535 lesz [C++ 98].

Java • A Javában a számokat a C-hez hasonlóan adhatjuk meg. Az értelmezési szabályok kimondják, hogy az oktális és a hexadecimális esetben a számjegyeket 3, illetve 4 bitre kell leképezni, balról nullákkal 32, illetve 64 bitre feltölteni, s az eredményt kettes komplemens kódban kell értelmezni [Jav 03b]. Így például, $0xFFFFFFFF = -1_{10}$, míg ugyanez az Adában: $16\#FFFFFFFF\# = (2^{32} - 1)_{10} = 4_294_967_295_{10}$

Mathematica • Érdekességként megemlítjük, hogy a Mathematicában 2 és 36 között minden alap megengedett, vagyis a `'0'...'9'` számjegyeken kívül kiterjesztett számjegyként az angol ábécé 26 betűje is használható.

A tízes alapú számrendszerben megadott `n` egész szám `a` alapú ($a \leq 36$) számrendszerbeli alakját a `BaseForm[n, a]` adja, míg fordítva: az `a^^n` kifejezés az `a` alapú számrendszerben megadott `n` számot decimális számmá alakítja, például: `30^^Mathematica` = 13 207 019 439 499 570. [Szi 96]

2.4.2. Karakterek és karaktersorozatok

A grafikus karakterkonstansokat leggyakrabban ' jelek (aposztrófok) közé írhatjuk – `'A'` –, bár némely nyelv megengedi közvetlenül a karakterkódok használatát is. Bizonyos nyelvekben, mint például a C-ben, C++-ban, Javában stb. néhány karakternek szabványos neve van, és a `'\'` „escape" karakter segítségével is beírhatók. Példáként nézzük meg a 2.3. táblázatot.

A karaktersorozatok (sztringek) a nyelv által megengedett karakterekből képzett, általában " jelek (macskakörmök) vagy aposztrófok közé írt szövegek. Például: `"Ez egy rövid szöveg"`.

A lehetséges karakterkészletekről bővebben jelen fejezet elején, a különböző programozási nyelvek által támogatott karakter- és karaktersorozat-típusokról bővebben az 5., illetve a 6. fejezetben olvashatunk.

Kód	Leírás
`\n`	újsor
`\t`	tabulátor
`\b`	törlés, egy karakter vissza
`\r`	sorvissza
`\f`	lapdobás
`\\`	backslash
`\'`	aposztróf
`\"`	idézőjel
`\ooo`	karakter oktális számrendszerben (0–377)
`\uhhhh`	Unicode karakter hexadecimálisan (0–0xffff)

2.3. táblázat: Escape sorozatok a Javában.

2.5. Megjegyzések

A programok karbantarthatóságát, áttekinthetőségét jelentősen javítják a programban elhelyezett megjegyzések is. Ezek olyan magyarázó szövegek amelyekben a programozó leírja az egyes részek szerepét, bonyolultabb algoritmus esetén a megoldás fő elvét. A fordítóprogram ezeket a végrehajtható kód készítése során nem veszi figyelembe, ezek sem a szintaktikai helyességet, sem a program jelentését nem befolyásolják.
A különböző nyelvekben szokásos lehetőségek:

Egy speciális oszlopban elhelyezett jeltől a sor végéig • Például ha FORTRAN-ban `'C'` karaktert írunk az első oszlopba, akkor ez a teljes sor megjegyzéssé válik. Ezt a lehetőséget időnként assembly szintű nyelveknél is alkalmazzák. A magasszintű nyelveknél a kötetlen formátum használata miatt ma már nem jellemző. Gondot jelent ugyanis, hogy ez a módszer nem ad olyan lehetőséget, amivel valamely kódrészlet *mellett*, tőle jobbra helyezhetünk el megjegyzéseket.

Speciális jelek a megjegyzés elején és végén • Ez elég gyakori, számos programozási nyelv adja lehetőségként, így akár nagyobb kódrészletek is megjegyzéssé válhatnak. Az ALGOL 60-ban például a **comment** kulcsszótól a ';' (pontosvessző) jelig tart egy megjegyzés. A Pascalban a megjegyzések elejét és végét a `(*` és `*)` vagy `{` és `}` pár jelöli. Ezzel a módszerrel akár egy utasítás közepébe is tehetünk megjegyzést, például:

if *List_mut* = **nil** *(* a lista ures *)* **then** ...

Hasonló igaz a C-re és a C++-ra: itt `/*` és `*/` között helyezhetjük el a magyarázatokat.
Ez a lehetőség is mutatja, hogy a megjegyzések szintaktikai szerkezete befolyásolja programjaink megbízhatóságát is, hiszen egy „megjegyzés-vége" jel hiányának hatására a fordító általában egészen a következőnek megtalált „megjegyzés-vége" jelig figyelmen kívül hagyja a programkódot.

Speciális jel a megjegyzés elején – a megjegyzés vége a sor vége • Ez megbízhatóbbá teszi a programot, védi az előzőekben említett figyelmetlenség következményeitől. Ez az

oka, hogy a C++-ban az előzőek mellé már bevezették ezt a lehetőséget is: // speciális jeltől az aktuális sor végéig is írhatunk megjegyzéseket. Az Ada 95-ben vagy az Eiffelben a programok megbízhatósága kiemelt fontosságú, ezért itt *csak* a -- jeltől a sor végéig tartó megjegyzéseket engedik meg.

Már több programozási nyelvben léteznek a programok úgynevezett *belső dokumentációját* segítő eszközök. Ezek képesek arra, hogy a programszövegnek egy olyan kivonatát hozzák létre, amely a programban használt függvények, eljárások specifikációs sorait és bizonyos – külön erre szánt jelekkel megjelölt – megjegyzéseit tartalmazza.

A Javában ugyanúgy írhatunk megjegyzéseket, mint a C++-ban, de létezik a *dokumentációs megjegyzés* lehetősége is: a

/** dokumentációs megjegyzés */

alakú szövegeket a *javadoc* program segítségével kiszűrhetjük.

Az Eiffelben a környezet ad támogatást a belső dokumentáció készítéséhez, itt az osztályok dokumentációjaként megkaphatjuk az egyes jellemzők specifikációs sora után írt megjegyzéseket, a jellemzők elő- és utófeltételeit, valamint az osztályinvariánst is[11].

A C#-ban a megjegyzések írásához a C++-ban megszokott //, illetve /* ... */ jelek használhatóak, de emellett elhelyezhetők ///... alakú egysoros dokumentációs megjegyzések is. Ezeket a dokumentációs megjegyzéseket az XML szabványnak megfelelően kell (célszerű) írni, hogy a dokumentáció-készítő segédprogram megfelelően tudja kezelni.

Feladatok

1. Bizonyos programozási nyelvek megkülönböztetik a programszövegben a nagy- és a kisbetűket, mások egyenértékűnek tekintik őket. Sorolja fel mindkét lehetőség előnyeit és hátrányait, figyelembe véve a program megbízhatóságát, a típusellenőrzést és a fordítás bonyolultságát. Indokolja, hogy ha új programozási nyelvet tervezne, melyik lehetőség mellett döntene!

2. A következő azonosítók közül melyik a legjobban olvasható, melyiket használná legszívesebben a programjaiban? Indokoljon!

```
BeosztottakMaxSzáma          beosztottak_max_szama
beosztottak_max_száma        BEOSZTOTTAKMAXSZAMA
BEOSZTOTTAKMAXSZÁMA          b_m_sz
BeosztottakMaxSzama          bmsz
```

3. Keressen olyan programozási nyelvet, amely megengedi az egymásba ágyazott megjegyzések használatát! Mi lehet az oka annak, hogy kevés ilyen nyelv van?

4. Milyen helyzetekben indokolt a tízes számrendszeren kívül más számrendszerek használatának lehetősége egészek, illetve valósak esetén?

[11] Ez az osztály úgynevezett *rövid* (*short*) formája, bővebben lásd [Mey 91].

3 Vezérlési szerkezetek, utasítások

Az imperatív programozási nyelvekben utasítások írják le a programok elemi lépéseit. A programozó ezekkel valósítja meg a program állapotterének megváltoztatását. Az imperatív paradigmában a program végrehajtása nem más, mint ezeknek az utasításoknak az egymás utáni végrehajtása. A programozási nyelvek különféle eszközöket biztosítanak ennek a végrehajtási sorrendnek a módosítására. Ebben a fejezetben áttekintjük az imperatív nyelvek vezérlési szerkezeteit, ezek kialakulásának folyamatát, vázolva a történelmi hátteret és egy mögöttük lévő kiszámítási modellt.

Ebben a fejezetben a programozási nyelvekben előforduló vezérlési szerkezetekkel, valamint a nyelvekben előforduló alapvető utasítástípusokkal foglalkozunk. Utasítások írják le a programok elemi lépéseit, míg a vezérlési szerkezetek, amelyek gyakran utasítások formájában testesülnek meg, teszik lehetővé az utasítások végrehajtási sorrendjének leírását. A vezérlés megértéséhez tisztában kell lennünk azzal a ténnyel, hogy a Neumann-elvű számítógépek mikroprocesszorai úgy működnek, hogy a memóriában található gépi kódú utasításokat memóriabeli előfordulási sorrendjükben hajtják végre. Ezt nevezzük *szekvenciális vezérlésnek*, szekvenciális végrehajtási sorrendnek. Ezt a szekvenciális végrehajtási sorrendet négy alapvető vezérlésmódosító utasítással módosíthatjuk. Ezek a módosítók a következők:

- Feltétel nélküli vezérlésátadás
- Feltételes vezérlésátadás
- Alprogramhívás: eljáráshívás, függvényhívás (objektumorientált nyelvekben metódushívás). Egyes – ma már a legtöbb – nyelvekben rekurzióval[1].
- Visszatérés az alprogramokból

Ezeken kívül léteznek más vezérlési szerkezetek is, melyek általában programozásinyelv-függőnek tekinthetők, de ezek kifejezhetők a fent említett négy alapmódosítóval. Ilyen például a többirányú elágazás: egyes nyelvekben a *case*, másokban a *switch* utasítás.

3.1. A programozó feladata

Programozóként dolgozva gyakran teszik fel nekünk azt a kérdést, hogy mivel is foglalkozunk a munkánk során. Erre a kérdésre egy másik programozónak könnyen válaszolhatunk munkánk többé-kevésbé részletes ismertetésével. Egy informatikai ismeretekkel nem rendelkező embernek viszont hiába próbáljuk elmagyarázni azokat a témákat, amikkel foglalkozunk, a reakció gyakran – mondhatjuk, hogy jó esetben – az lesz, hogy a kérdező elismerően bólint, esetleg meglepődik, hogy vajon a társadalomnak erre mi szüksége lehet. Ha belegondolunk, hogy mi is egy programozó, illetve informatikus szakember feladata,

[1] Programkód esetében használják a rekurzió szinonímájaként az önhívó kifejezést. A rekurzió bővebb fogalom, mint az egyszerű önhívás, így ahol ez szükséges, a rekurzív kifejezést használjuk a magyarosabb önhívó kifejezés helyett.

akkor könnyebb megfogalmazást is találhatunk: a gyakorlati életben felmerülő problémák megoldása. Ez a válasz a laikusok számára is érthetővé teszi azt, hogy mivel is foglalkozunk és önmagában adja azt is, hogy erre a munkára a társadalomnak nagy szüksége van. A programozó feladata problémamegoldás: egy meglévő, adott állapotból egy kívánt másik célállapot – a megoldás – elérése. A megoldás megkeresése kötött „játékszabályok" szerint történik. Ezek a szabályok írják le, hogy egy adott állapotból hogyan juthatunk el a megoldás egy következő állapotába. Egy adott problémára, pontosabban problémaosztályra adott megoldásmódot nevezzük algoritmusnak.

Ahhoz, hogy egy algoritmus számítógép által érdemben feldolgozható legyen, többféle követelménynek is meg kell felelnie. Ezek a követelmények a következők:

- Egyértelműen meghatározott lépésekből állhat.
- Lépésenként végrehajthatónak kell lennie.
- Végesnek kell lennie (mind a leírásának, mind pedig a végrehajtásának).
- Mindent leírónak, pontosnak kell lennie: a számítógép csak az általunk leírt lépések pontos végrehajtására képes, ezért ezeket a lépéseket nagyon pontosan kell leírnunk.
- Az algoritmus mindig egy jól meghatározott bemenő adatokkal leírt állapotból egy másik, eredményadataival jól meghatározott végállapotba vezet.

Az algoritmusok leírására az igényektől és a lehetőségektől függően többféle eszközt használhatunk. Ezek közé tartozik a mondatszerű leírás, a folyamatábrákkal történő leírás, a D-diagramokkal történő leírás, a blokkdiagramokkal történő leírás és a struktogramokkal való leírás (a struktogramokat nevezik Nassi-Schneider diagramoknak is). Végül, a fejezetünk szempontjából talán a legfontosabb, és a könyvünkben részletesen körüljárt leírási módszer: a programozási nyelvek alkalmazásával történő szöveges leírási módszer.

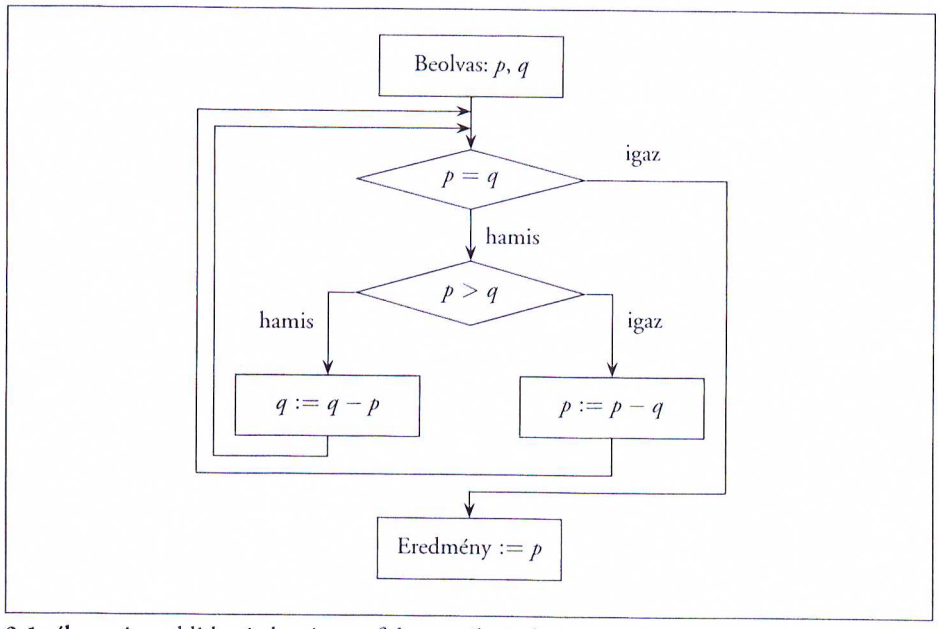
A következőkben áttekintjük ezeknek az eszközöknek az elemi alkalmazását. Az euklideszi algoritmus segítségével két természetes szám legnagyobb közös osztóját határozhatjuk meg. Az algoritmus alapötlete a következő: két szám legnagyobb közös osztója osztója a két szám különbségének is.

3.1.1. Mondatszerű leírás

A mondatszerű leírás során az algoritmus lépéseit a mindennapjainkban is használt mondatformákkal, mondatszerkezettel írjuk le. Ennél a módszernél törődünk a legkevésbé azzal, hogy a mondatok minden egyes szava egy számítógép számára is teljesen érthető legyen, bár a precizitásra itt is törekedni kell. Az euklideszi algoritmus mondatszerű leírása a következőképpen történhet (adott két szám, melyeknek legnagyobb közös osztóját kell meghatározni).

1. Hasonlítsuk össze a két számot.
2. Ha a két szám megegyezik, akkor készen vagyunk: a számok mindegyike az eredeti számok legnagyobb közös osztóját határozza meg.
3. Ha a két szám nem egyezik meg, akkor a nagyobbikból vonjuk ki a kisebbiket.
4. Folytassuk a végrehajtást az 1. lépéstől.

A fenti leírás számunkra kellően pontosan megadja a legnagyobb közös osztó kiszámításának módszerét, viszont egy számítógépnek elég nehéz lenne ezeket a – magyar nyelven –



3.1. ábra: Az euklideszi algoritmus folyamatábrával

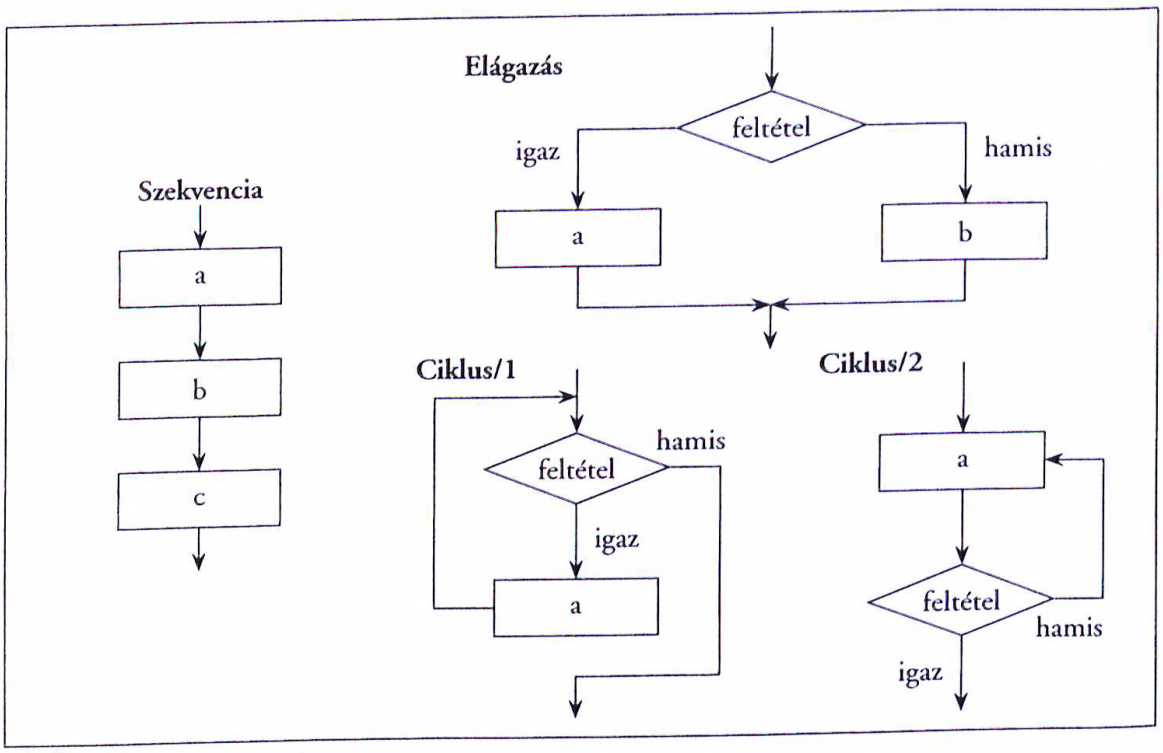
megfogalmazott mondatokat megadva előírni az elvégezni kívánt kiszámítási lépéseket. Az algoritmust átfogalmazhatjuk a maradékképzés műveletének tulajdonságainak ismeretében (a két algoritmus egyenértékűségének bizonyítását az Olvasóra bízzuk gyakorlásként). Az átfogalmazott algoritmus a következőképpen írható le:

1. Osszuk el p értékét q értékével. Az osztási maradékot jelöljük r-rel (ennek 0 és $q - 1$ közé kell esnie).
2. Ha r (a maradék) értéke 0, akkor q a keresett legnagyobb közös osztó. Egyébként a korábbi q-t tegyük p-be, és r-t q-ba.
3. Folytassuk a végrehajtást az 1. lépéstől.

3.1.2. Folyamatábrák

Az algoritmusok papíron történő leírásához kihasználhatjuk a papír egyik kényelmes adottságát: a kétdimenziós ábrázolás lehetőségét. Ez átláthatóbb leírási módot eredményez, viszont a kétdimenziós ábrázolás és az ehhez kötődő geometriai alakzatok elég távol állnak a programozók munkája során alkalmazott programozási nyelvektől.

A folyamatábrákban téglalap alakú dobozokba írjuk a végrehajtási lépéseket, a lépések végrehajtási sorrendjét pedig a dobozok közötti nyilakkal rögzítjük. Feltételekhez kötött elágazásokat általában rombusz alakú dobozba írunk: a dobozba írjuk az elágazási feltételt, míg ebből két nyíl is kivezethet: az egyik nyíl a dobozba írt feltétel teljesülése esetére írja elő a további végrehajtási sorrendet, míg a másik nyíl az ellenkező esetre, vagyis a feltétel nem teljesülése esetére. A nyilakra rá is írjuk, hogy melyik végrehajtási ágat jelölik.



3.2. ábra: D-diagramok

Azzal, hogy ezáltal a programok leírása eléggé terebélyes lesz, könnyen olyan folyamatábrákhoz juthatunk, amelyek nehezen áttekinthetők – mivel kuszák – és ráadásul nehezen ültethetők át konkrét programozási nyelvekre. Az ilyen programok ráadásul nehezen módosíthatók is.

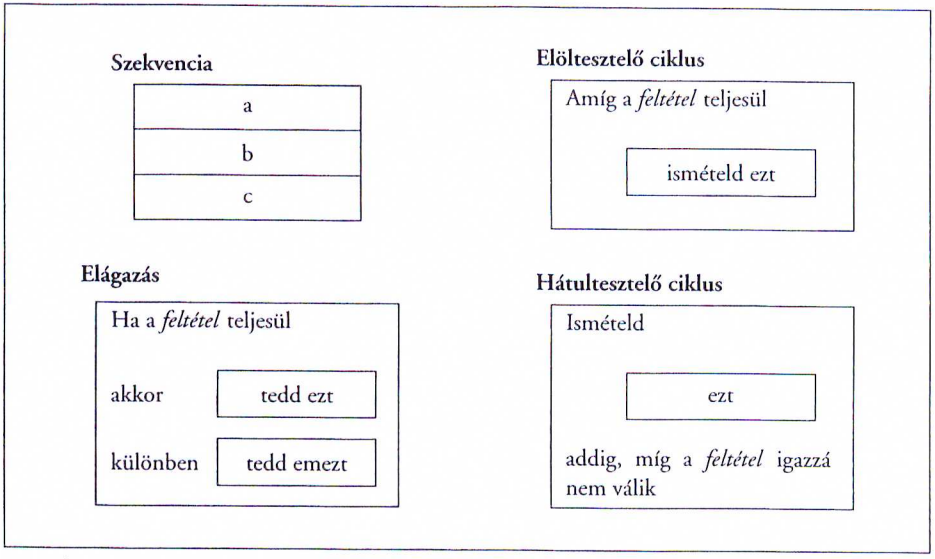
Az euklideszi algoritmus a 3.1. ábrán látható módon írható le folyamatábrával.

3.1.3. D-diagramok

A folyamatábrák kuszaságát megelégelő emberek közül Edgser W. Dijkstra alakított ki egy szűkebb jelölési rendszert, melynek lényege, hogy olyan elemi szerkezetekre szorítkozik, melyeknek minden esetben pontosan egy kimeneti (folytatási) ága van (nyilvánvaló kivételt jelenthet a program vége, ahol a vezérlés megáll). Dijkstra neve alapján ezeket a folyamatábra-elemeket *D-diagramoknak* nevezik.

1. Egy egyszerű művelet egy D-diagram.
2. Ha *A* és *B* D-diagramok, akkor *A* és *B* egymás után végrehajtva (szekvencia) is D-diagram, ha *c* egy feltétel, akkor a *„ha c feltétel igaz, akkor A, egyébként B"*, valamint a *„ha c feltétel igaz, akkor A"*, valamint a *„mindaddig, amíg a c feltétel igaz, hajtsuk végre ciklusban A-t"*, és a *„hajtsuk végre A-t, majd amíg a c feltétel igazzá nem válik, hajtsuk végre ciklusban A-t"* is D-diagramok.
3. Más D-diagram nincs (követelmény: legfeljebb 1 kimenő él minden konstrukcióból).

A D-diagramokat a 3.2. ábrán szemléltetjük. (A ciklusra két változatot is adunk, egy elöltesztelő és egy hátultesztelő változatot.)



3.3. ábra: Blokkdiagram építőelemek

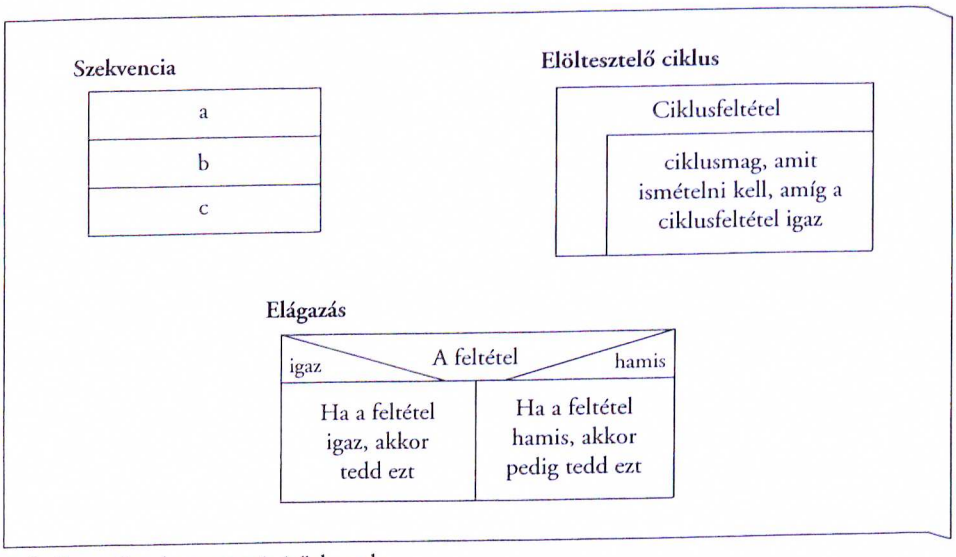
A D-diagramok megszorításaival elérték, hogy a program bonyolultsága, összetettsége valóban arányos lett a program hosszával, és a minták ismertsége miatt a leírás olvashatósága is lényegesen jobb a hagyományos folyamatábrák olvashatóságánál. A programok vezérlési szerkezeteinek ezen elvek szerinti kidolgozását, a program alprogramokra bontását, a kifejezések és a változók által felvehető értékek és a rajtuk elvégezhető műveletek rögzítését együttesen strukturált programozás néven ismerjük.

Böhm és Jacopini [Böh 66] megmutatták, hogy minden – folyamatábrákkal leírható – algoritmus leírható D-diagramokkal is. Ezt a fajta programozási stílust nevezzük **goto** nélküli programozásnak (a **goto** sok programnyelvben a feltétel nélküli vagy feltételes vezérlésátadás leírására szolgáló utasítás).

3.1.4. Blokkdiagramok

Egy egyszerű művelet téglalapba (dobozba) rajzolva egy blokkdiagram. Ezek a dobozok egymás után írva szekvenciálisan, sorban lesznek végrehajtva. Az elágazásnál két dobozt és egy feltételt adunk meg. A feltételtől függ, hogy melyik doboz tartalma lesz végrehajtva. A feltétel hamis volta esetére nem kötelező dobozt megadni (ezt jelzi a kipontozottan bekeretezett doboz). Az egész elágazást egy újabb dobozba tesszük, ami bárhol használható, ahová egy dobozt rakhatunk. Ciklus esetén egy belső doboz tartalmazza az ismételten végrehajtani kívánt tevékenységet (ciklusmagot) leíró dobozt. Ezt betesszük egy nagyobb, külső dobozba, amely az ismétlési feltételt, a ciklusfeltételt tartalmazza.

A blokkdiagramoknál a 3.3. ábrán látható szerkezeti elemeket használjuk az algoritmusok leírására.



3.4. ábra: Struktogram építőelemek

3.1.5. Struktogramok

A struktogramokat is az algoritmusleírás eszközeként használhatjuk. Ez is a tevékenységek téglalapokba írására, illetve a téglalapokból programszerkezetek képzésére épül. A tevékenységeket tartalmazó téglalapok könnyen egymásba ágyazhatók, így a beágyazással könnyen szemléltethető a program szerkezete (hasonlóan a blokkdiagramokhoz). Előnyük a blokkdiagrammal szemben annyi, hogy kötöttebb formájuk miatt könnyen előállíthatók programmal vagy szövegszerkesztővel.

A struktogramoknál a 3.4. ábrán látható szerkezeti elemeket használjuk az algoritmusok leírására.

3.2. Az assembly nyelvű megoldás

A programozási nyelveket gyakran csoportosítják alacsonyszintű, illetve magasszintű programozási nyelvek csoportjaiba. Az alacsonyszintű nyelveket általában egy-egy számítógép vagy számítógép-család igényeinek megfelelően alakítják ki: itt az utasítások az adott számítógép-család mikroprocesszorának utasításkészletéből állnak össze. Ezeket a nyelveket nevezzük assembly nyelveknek. Ezeknek az utasításai közvetlen megfeleltethetők egy-egy gépi kódú utasításnak. A fordítási feladatot, vagyis az assembly nyelven megírt programból egy gépi kód megfelelő előállítását az assembler végzi. A magasszintű – algoritmikus – nyelvek utasításkészlete független egy adott számítógép-architektúra utasításkészletétől: végrehajtásuk előtt egy fordítóprogramnak kell őket assembly kódra, illetve gépi kódra fordítani.

A következőkben bemutatjuk az euklideszi algoritmus Pascal nyelvű változatát az ebből előállított Motorola 68000-es mikroprocesszor-család assembly utasításaival. A Moto-

rola 68000-es mikroprocesszor választása önkényes, jelenleg az Intel 80386-os mikroprocesszora és annak utódai (Pentium család) a legelterjedtebbek. A választásunk azért esett a 68000-es mikroprocesszorra, mert ennek utasításkészlete kialakításában nagyobb részt vállaltak a szoftveres szakemberek, így az utasítások még viszonylag magasabb szintűek, bonyolultabb feladatok megoldására alkalmasak, mint az Intel 80386-os mikroprocesszornál. Míg a 68000-es mikroprocesszor-család az Intel 80386 családdal együtt kezd kiszorulni a piacról (ezek úgynevezett CISC processzorok, Complex Instruction Set Computer – összetett utasításkészletű processzorok) és helyükön egyre több helyen RISC (Reduced Instruction Set Computer), csökkentett utasításkészletű processzorokat használunk, mint amilyen a MIPS vagy az IBM PowerPC architektúrája. A legújabb Pentium processzorok is RISC architektúrájúak. Az Intel 80386-ossal való kompatibilitás egy, a processzorban elhelyezett processzor-emulációs blokkban lesz megvalósítva. Ennek feladata az összetettebb 80386-os utasítások RISC alaputasításokra bontása. Mindezt úgy végzi, hogy a programozó ebből semmit se lásson, ha erre nincs szüksége. Megjegyezzük, hogy ezeknek a RISC processzoroknak a lehetőségeit, hatékony elemeit a programozók általában közvetlenül nem használják ki, ugyanis nagyon ritka az, hogy valaki ezekre assembly nyelvű vagy gépi kódú programokat írjon, mivel az utasítások adta lehetőségek optimális kihasználása ember által nehezen átlátható. Ezért ezekre az architektúrákra egy optimalizáló fordítóprogram hatékonyabb kódot tud készíteni, mint egy speciális ismeretekkel fel nem vértezett programozó.

3.2.1. A Pascal nyelvű program

Most tekintsük röviden át az euklideszi algoritmus – maradékokkal átfogalmazott – Pascal nyelvű megvalósítását:

```
procedure lnko(p, q: integer; var eredmeny: integer);
    var maradek: integer;
begin
    maradek := p mod q;
    while maradek <> 0 do
    begin
        p := q;
        q := maradek;
        maradek := p mod q
    end;
    eredmeny := q
end;
```

3.2.2. A Motorola 68000-es család programozása

A Motorola 68000-es assembly kód jobb érthetősége érdekében most röviden bemutatjuk a Motorola 68000-es mikroprocesszor család főbb jellemzőit, programozói modelljét és néhány jellemzőbb utasítástípusát. Ez egyben példa arra, hogy milyen jellegű utasításokkal találkozhatunk assembly nyelvű programok kódolásakor.

Megjegyezzük, hogy a leírás nem teljes. Célunk nem a Motorola 68000-es család referenciájának elkészítése volt – ez amúgy is ingyen letölthető a Motorola honlapjáról. Célunk sokkal inkább betekintést nyújtani egy nagyobb arányban szoftver-szakemberek által tervezett mikroprocesszor nyújtotta néhány lehetőségbe.

A Motorola 68000-es család több processzortípust is tartalmaz: ezek belül általában 32 bitesek, azaz 32 bites adatokkal dolgoznak, de például a 68000-es változat kifelé 16, a 68008-as változat pedig kifelé nyolcbites rendszerbuszokhoz illeszkedik (ezekhez sokkal olcsóbb a perifériák és egyéb kiegészítő eszközök kifejlesztése). A 68020-as egy igazi 32 bites mikroprocesszor, lapozásos virtuális memóriával. A 68030-as egy második generációs, a 68040-es egy harmadik generációs 32 bites mikroprocesszor. Ezeknek az utasításai lényegében összeegyeztethetők egymással. Mi a tárgyalásunkkor a közös őst, a 68000-est vesszük alapul, de ugyanezek elmondhatók a későbbi változatokról is.

A Motorola 68000-es modellje két működési módot, és ennek megfelelően két regisztercsoportot tartalmaz: felhasználói módban a felhasználói programok által elérhető regisztereket, illetve felügyelői (supervisor, avagy védett) módban ezeken kívül további regisztereket biztosít – de a speciális, csak felügyelői módban elért regisztereket általában csak az operációs rendszerek készítői használják. A továbbiakban csak a felhasználói móddal foglalkozunk.

A processzor gyors elérésű adattárként 16 darab 32 bites általános célú regisztert biztosít (D0-D7, A0-A7). Az előbbiek adatműveletekre vannak felkészítve, míg utóbbiakkal inkább memóriarészeket címezhetünk és a címzéssel kapcsolatos műveleteket használhatjuk (az A7-es regiszter a végrehajtási verem mutatója). Van egy 32 bites programszámláló, ami az éppen végrehajtás alatt álló utasítás memóriabeli címét tartalmazza. Továbbá van egy *feltételbiteket tartalmazó regiszter*, ami 16 bites, de felhasználói módban csak az alsó 8 bitje érhető el. Ez tartalmazza a legutóbb végrehajtott művelet eredményének jellemzőit (nulla volt-e a végeredmény vagy negatív volt-e, történt-e aritmetikai túlcsordulás vagy keletkezett-e nem nulla átvitel bit aritmetikai művelet során). Ezeket a feltételes vezérlésátadó utasítások használhatják vezérlésátadási feltételként. A processzorcsaládhoz tartozik egy lebegőpontos aritmetikai társprocesszor (MC 68881, illetve MC 68882). Ez további regisztereket biztosít a lebegőpontos aritmetikai műveletekhez, de ezekkel szintén nem foglalkozunk.

Az adatregiszterek használhatók 8, 16, illetve 32 bites regiszterekként is. A konkrét méretet az utasítások mögé egy pont után illesztett méretazonosítóval adhatjuk meg (`B` a bájtot, `W` a 16 bites hosszt, az `L` pedig a 32 bites hosszat jelöli). Például a regiszterek értékét feltöltő `MOVE` utasítás 8 bites érték esetén `MOVE.B` alakú (ilyenkor a maradék 24 bit értékét az utasítás nem módosítja, nem nullázza), 16 bites érték esetén `MOVE.W` alakú. Az optimalizáló fordítók ezt kihasználva, illetve azt, hogy sok helyen elegendő a 16 bites adatok kezelése, gyakran tesznek egy 32 bites regiszter alsó, illetve felső 16 bitjébe különböző 16 bites válto-

zókat és az alsó és felső 16 bit tartalmának felcserélésével (erre használható a SWAP assembly utasítás) mindig a megfelelő értékhez engednek hozzáférni[2].

A 8 bites, feltételbiteket is tartalmazó regiszter az elágazás-utasítások számára érdekes. Egy értékeket összehasonlító utasítás után például átadhatjuk a vezérlést egy másik programrészletnek, ha a két érték egyenlő volt. A feltétel nélküli vezérlésátadási utasítás a BRA, míg az értékek egyenlősége esetén történő – feltételes – vezérlésátadásra a BEQ utasítást használjuk (az utasítás nevének első betűje azonosítja azt, hogy vezérlésátadásról van szó – a branch angolul elágazást jelent, az azt követő két jel pedig a vezérlésátadás feltételét azonosítja; például az EQ az egyenlőséget, az NE a nem egyenlőt, a GT a „nagyobb, mint" feltételt). A vezérlésátadó utasítások is gyorsabban lesznek végrehajtva, ha kisebb az átugrandó távolság. Ezt az ugró utasítás mögé egy pont után írt S betű (short, azaz rövid) jelzi. Viszont ez nem használható, ha nagyobb kódrészleteket akarunk átugrani (több, mint 255 bájtot).

Utasítások

A felhasználói módban végrehajtható utasítások a következő csoportokra oszthatók:

Adatmozgató utasítások:

MOVE: adatmozgatás regiszterek és memória, illetve regiszterek között

MOVEQ: gyors adatmozgatás (8 bites állandót 32 bites regiszterbe mozgat)

MOVEM: egyszerre több regiszter mentése (memóriába, verembe) és visszatöltése onnan

LEA: cím betöltése egy címregiszterbe

PEA: cím ráhelyezése a végrehajtási veremre

LINK: alprogramok kezdetekor a hívási lánc[3] és a lokális változók kezelésére használt utasítások (lefoglal megadott számú bájtot a végrehajtási veremben a lokális változóknak)

UNLK: a LINK-kel lefoglalt területek felszabadítása, hívási láncban, a blokk-struktúrában a tartalmazó (külső) blokk aktuálissá tétele

Egész típusú adatokon végzett aritmetikai műveletek:

ADD: összeadás

SUB: kivonás

MUL: szorzás (MULS az előjeles, MULU az előjel nélküli szorzást jelöli)

DIV: osztás (DIVS az előjeles, DIVU az előjel nélküli osztást jelöli)

CMP: összehasonlítás (az eredménytől függően a feltételbitek be lesznek állítva)

[2] Ezzel a problémakörrel, azaz a regiszter-lefoglalás problémájával részletesen foglalkozik Steven S. Muchnick *Advanced compiler design and implementation* című könyve [Muc 97].

[3] Itt is látszik, hogy a processzor tervezésekor szoftverszempontokat is figyelembe vettek: ezzel az utasítással a blokk-strukturált nyelvek támogatása volt a cél.

Logikai műveletek:

AND: *logikai és* művelet két érték között

OR: *logikai vagy* művelet két érték között

EOR: *logikai kizáró vagy* művelet két érték között

NOT: a logikai tagadás művelete

Eltolási műveletek: (megfelelnek a kettőhatvánnyal osztásnak vagy szorzásnak)

ASL: aritmetikai biteltolás balra

ASR: aritmetikai biteltolás jobbra (a legfelső helyiértéken az előjelbit jön be, vagyis az az érték marad, ami korábban is ott volt)

LSL: logikai biteltolás balra

LSR: logikai biteltolás jobbra (legfelső helyiértéken 0 jön be)

Elforgató műveletek: (hasonlóak az eltoláshoz, de ciklikusak, vagyis az egyik végen kijövő bitek a túloldalon visszajönnek)

ROL: bitenkénti balra forgatás

ROR: bitenkénti jobbra forgatás

SWAP: elforgatás 16 bittel (azaz alsó és felső 16 bit felcserélése)

Bitkezelő utasítások: (adott bit értékének módosítása, beállítása, lekérdezése)

BCHG: adott érték (regiszter) adott bitjének átkapcsolása

BCLR: adott bit 0-ra állítása

BSET: adott bit 1-re állítása

BTST: adott bit értékének lekérdezése és a feltételbitek ennek megfelelő beállítása

BCD aritmetikai utasítások:

ABCD: BCD összeadás, illetve kivonás

Vezérlésátadó utasítások:

B⟨*cc*⟩: feltételes vezérlésátadás ahol ⟨*cc*⟩ egy feltételkód (például EQ az egyenlőség, GT a nagyobb, mint...). Ez egy úgynevezett relatív ugrás – a célpont az aktuális programszámlálótól számított különbségével van megadva.

BRA: feltétel nélküli vezérlésátadás (relatív ugrás – célpont az aktuális programszámlálótól számított különbségével van megadva).

JMP: feltétel nélküli vezérlésátadás, abszolút ugrás, vagyis a célutasítás memóriabeli címét kell megadni benne, nem az átugrandó távolságot.

JSR: feltétel nélküli vezérlésátadás programszámláló mentésével (az aktuális programszámláló a végrehajtási verembe lesz mentve; alprogramok hívásához).

RTS: visszatérés egy JSR-rel meghívott szubrutinból (visszatérési cím a végrehajtási veremből).

NOP: az utasítás nincs hatással a regiszterekre, csupán a processzor egész típusú aritmetikai csővonalát szinkronizálja (az úgynevezett pipeline utasításfeldolgozó csővonaltechnika alkalmazása esetén érdekes).

Címzési módok

Az utasítások rövid áttekintése után megnézzük még a címzési módokat: azt, ahogy az utasítások által feldolgozandó adatok helyét megadhatjuk. A Motorola ebben nagyon rugalmas és gazdag eszköztárat biztosít a programozónak. A processzor jellemzőbb címzési módjai a következők:

- **Közvetlen konstans érték megadása.** Az assembly programban egy # jelet kell írni a konstans elé.
- **Regiszter megadása.** A kívánt regiszter nevét (például A3 vagy D5) kell beírni a programba).
- **Címregiszter által mutatott memóriahelyen levő érték azonosítása.** A programban a címregiszter által mutatott memóriahelyen levő értékre a címregiszter zárójelek közé tett formájával hivatkozhatunk: például `(A0)` az A0 címregiszter által mutatott memóriacímen található értéket azonosítja.
- **Címregiszter által mutatott memóriahelytől adott távolságra levő érték azonosítása.** A programban a címregiszter értékéhez hozzáadott előjeles távolság által meghatározott memóriahelyen levő értékre hivatkozunk. A hivatkozás a címregiszternek a távolságot azonosító érték után zárójelek közé tett formájával történik: például `14(A6)` az A6 címregiszter által mutatott memóriacímtől 14 bájtra található értéket azonosítja.
- **Címregiszter által mutatott értékre hivatkozás, a címregiszter értékének a hivatkozott adat méretével való csökkentésével vagy növelésével.** Az adatot előbb veszi ki a memóriából és csak ezután változtatja meg a címregiszter értékét. Ha 16 bites értéket töltöttünk be a memóriából, akkor 2 bájttal (16 bittel) változtatja a regiszter értékét. Például az `(A7)+` az A7 által mutatott értéket jelöli, majd a memóriahozzáférés után az A7 értékét növeljük. A `-(A7)` az A7 által mutatott értéket jelöli, majd a memóriahozzáférés után az A7 értékét csökkentjük.
- **A programszámláló értékéhez képest viszonyított címzés.** Hasonló változatokkal, mint amit az előbbi pontokban láttunk.
- **Egyéb címzési módok.** A fenti címzési módok tovább kombinálhatók skálázási tényezővel, illetve a memóriában megadott távolsági tényező alkalmazásával. Ezekről a címzési módokról nem írunk. A modern RISC processzor-architektúrák ilyen címzési módokat nem támogatnak, mert kiszámításuk nagyon időigényes. Egy optimalizáló fordító többre megy az elemi címzési módokkal és az aritmetikai utasítások alkalmazásával: mindig csak azt használja, amire valóban szükség van.

A vezérlés témaköréhez tartozik még az úgynevezett *megszakítás-kezelés*. Egy megszakítás a processzor reakciója valamely külső periféria, vagy magának a programnak valamilyen állapotváltozására. Kiválthatja például a gépbe épített óra, egy billentyű megnyomása, de akár egy nullával osztás is. Ha megszakítás lép fel, a processzor felfüggeszti az aktuális programszál végrehajtását és elkezdi végrehajtani a fellépett megszakítást kezelő programkódot. Miután a megszakítás-kezelő programkód lefutott, a processzor folytatja a korábban megszakított programszál végrehajtását. A megszakítás-kezelés nagyon alacsonyszintű eszköz. Ezzel a fejezetben nem foglalkozunk részletesebben. Magasszintű programnyelveknél

vannak magasszintű eszközök a megszakítások kezelésére. Ilyen eszközök például a programozási nyelvek kivételkezelési mechanizmusai.

Ezzel befejeztük a Motorola 68000-es architektúrájának tárgyalását. A célunk nem assembly programozók képzése volt, hanem egy informális kép közlése az Olvasóval azért, hogy pontosabb elképzelése legyen arról, hogy mit biztosítanak a mai modern 32 bites mikroprocesszorok a programozónak mind utasításokban, mind pedig címzési, vezérlési, programozási lehetőségek tekintetében. A csökkentett utasításkészletű (RISC) modern mikroprocesszorok általában kevesebb utasítást és egyszerűbb címzési módokat biztosítanak, viszont ezeknél a processzoroknál nem ritka, hogy 16, 64 vagy 128 regiszter áll a programozó rendelkezésére a memóriahozzáférések számának csökkentése érdekében. De mint azt korábban írtuk, ezeknek a programozását inkább bízzuk a gyártóik által kifejlesztett optimalizáló fordítóprogramokra, és ne mi próbáljunk meg optimális kódot írni. Nagyon nagy a valószínűsége annak, hogy nem tudunk jobb kódot írni (sőt annak is, hogy sokkal rosszabb hatékonyságú kódot írunk), mint amit például egy optimalizáló C-fordító állítana elő a problémát leíró C nyelvű programból.

3.2.3. Az MC 68000-es assembly kód

A bemutatott Pascal kódból egy fordítóprogrammal[4] előállított Motorola 68000-es assembly kód egy ATARI ST számítógépen a következőképpen néz ki. A bemutatott programot az elterjedtebb assemblerekkel fordíthatjuk gépi kódú végrehajtható programmá (egy főprogramot is kell hozzá készítenünk). IBM PC tulajdonosok az Interneten elérhető ATARI ST szimulátorok valamelyikével és a szabadon elérhető assemblerekkel és szerkesztőprogramokkal (ez utóbbiak angol elnevezése linker program, linkage editor) próbálkozhatnak a program lefordításával és életre keltésével.

```
    .globl lnko
lnko:
    ; procedure lnko(p,q: integer; var eredmeny: integer);
    LINK      A6,#0      ; Lefoglalunk 0 bájtot a lokális változóknak.
    ; Ekkor a következő hivatkozásokat használhatjuk:
    ;   (A6)    = Az A6 regiszter régi tartalmát tárolja.
    ;             (zárójelek között egy címregiszter az általa mutatott memóriacím
    ;             tartalmára hivatkozik.)
    ;   4(A6)   = A visszatérési címet tárolja (4 bájtnyira az A6 által mutatott
    ;             memóriacímtől).
    ;   8(A6)   = Az 'eredmeny' nevű formális paraméter címe.
    ;             (Cím szerinti paraméterátadás!)
    ;   12(A6)  = A 'q' paraméter értéke (2 bájtos; a korábbi elemek memóriacímek
    ;             voltak, és ezért foglaltak le 4 bájtot).
    ;   14(A6)  = A 'p' paraméter értéke.
```

[4] A fordítóprogramot – a Pascal nyelv egy szűkített változatára – a fejezet szerzője készítette. A Motorola 68000-es kódgenerátor később lett a programhoz illesztve úgy, hogy kihasználja a processzor adta lehetőségeket.

```
    MOVEM D0-D2,-(A7)     ; A D0, D1, D2 regisztereket a verembe mentjük.
                          ; (Az A7 regiszter a veremmutató.)
    MOVE.W 14(A6),D0      ; A D0 regiszter 16 bites alsó része tartalmazza a
                          ; 'p' paramétert, a gyorsabb elérés érdekében.
    MOVE.W 12(A6),D1      ; A D1 regiszter 16 bites alsó része tartalmazza a
                          ; 'q' paramétert, a gyorsabb elérés érdekében.

__cimke10:
    MOVEQ #0,D2           ; maradek := 0 (#0 a 0 konstans értéket jelöli;
                          ; a D2-ben a maradékot tároljuk.)
    MOVE.W D0,D2          ; maradek := p (csak 16 bitet másol, a maradék
                          ; bitek 0 értékűek maradnak.)
    DIVS  D1,D2           ; maradek := (p mod q, p div q) rendre a
                          ; magasabb, illetve alacsonyabb helyiértéken.
    SWAP D2               ; maradek := (p mod q, p div q) – itt
                          ; megcseréltük az alsó és a felső 16 bitet.
    TST.W D2              ; A 'maradek' változó alacsony helyiértékű
                          ; részének (p mod q) értéke nulla?
    BEQ.S __cimke11       ; Ha igen, ugorjunk a program végére.
    MOVE.W D1,D0          ; p := q
    MOVE W D2,D1          ; q := korábban számított p mod q érték
    BRA.S __cimke10       ; Menjünk vissza a while ciklus elejére.

__cimke11:
    MOVE.L 8(A6),A0       ; A0 := az 'eredmeny' nevű paraméter címe.
    MOVE.W D1,(A0)        ; Mentsük 'q' értékét az 'eredmeny' nevű paraméter
                          ; címére a memóriában.
    MOVEM.L (A7)+,D0-D2   ; Visszaállítjuk a D0, D1, D2 regiszterek tartalmát
                          ; a hívás előtti értékekre
    UNLK A6               ; Visszaállítjuk az A6-os címregiszter eredeti értékét.
    MOVE.L (A7)+,A0       ; A végrehajtási veremből kivesszük a visszatérési címet
                          ; (szokás szerint ez a hívott eljárás feladata).
    ADDA.W #8,A7          ; A végrehajtási veremből kivesszük a paramétereket
                          ; (szokás szerint ez is a hívott eljárás feladata).
    JMP (A0)              ; Visszatérünk az A0 regiszterben tárolt visszatérési címre.
```

3.2.4. Az assembly és a Pascal nyelvű kód összevetése

A Pascal és az assembly nyelvű programok között számos általános érvényű különbséget figyelhetünk meg.

- A Pascal programok (mint más magasszintű nyelveken írt programok) hordozhatók. Míg ezek többnyire módosítás nélkül átvihetők új környezetbe, addig az assembly

nyelven írt programok mikroprocesszorhoz kötöttek: egy új mikroprocesszor-architektúrára való áttéréskor a teljes programot át kell írni az új processzorra. Egyes esetekben a másik processzor szimulációja hardveres úton megoldható, de ezzel nagyon sokat vesztünk a hatékonyságból.

- A gépi nyelvű, assembly programok gépi műveleteket írnak le, például regiszterek és memóriacímek ide-oda mozgatását. A magasszintű nyelvekben ehelyett ennél magasabb szintű absztrakciókat használhatunk. Például a memóriarekesz/regiszter absztrakciójaként a változókat. Magasszintű nyelven megírt programban nem kell foglalkoznunk a futtató számítógép regiszterkezelésével vagy memóriaszervezésével. Ezeket a feladatokat bízzuk a fordítóprogramra.
- Az alacsonyszintű program nehezebben olvasható. A fenti Motorola 68000-es kódrészlet a mellé írt megjegyzéseknek és az általa végrehajtott algoritmus ismertségének köszönhetően érthető volt, viszont ha egy 10 000 soros assembly nyelven megírt, megjegyzésekkel alig ellátott programot kapunk, annak módosítása komoly nehézségeket okozhat.

Végül megemlítjük, hogy az assembly program már szövegesen olvasható, a programozónak sokat mondó elemeket tartalmaz, de végrehajtva ebből – az assembler által előállított gépi kódú program lesz, amit leginkább bitek sorozatával jellemezhetünk. Ennek programozása (a „bitfaragás") nagyon nehéz munka, és az alacsony-, majd a magasszintű programnyelvek megjelenése előtt bizony így programoztak.

3.3. Egy elemi megközelítés

Az algoritmusok leírásának vizsgálatakor matematikusok számos kiszámítási modellt dolgoztak ki. Ezek a módszerek minimális absztrakciót igényelnek mind hardver, mind pedig szoftver oldalról, mégis lehetővé teszik a problémák egyszerű algoritmikus megfogalmazását és leírását. Most ezek közül a módszerek közül mutatunk be egyet, melynek részei:

- Potenciálisan végtelen memória a változók tárolására[5].
- Potenciálisan végtelen nagy memória magának a programnak a tárolására.

Adatként csak – tetszőlegesen nagy – természetes számokkal dolgozunk (beleértve a nullát is, mint természetes számot). Ez nem jelent megszorítást, mivel a legtöbb adattípus valamilyen módon leképezhető erre a halmazra. A be- és kimenettel sem foglalkozunk: feltételezhetjük, hogy az algoritmusok bemeneti és kimeneti értékeit a memóriában tároljuk. Erre a számítási modellre felépítjük az úgynevezett *while-programok* modelljét, majd megnézzük néhány magasabb szintű művelet megvalósíthatóságát ebben a modellben.

[5] A „potenciálisan végtelen" jelző a változókra vagy az őket reprezentáló memóriarekeszekre vonatkozóan azt jelenti, hogy ha megjelöljük a memóriarekeszeket $r_1, r_2, r_3, \ldots r_n$ módon, akkor n – a rendelkezésre álló memóriarekeszek száma – tetszőlegesen nagy értéket felvehet a program igényeinek megfelelően.

3.3.1. A while-programok elemei

A while-programok négyféle alaputasításból állnak:

1. Egy tetszőleges memóriarekesz értékének törlése:

   $x := 0$

2. Egy tetszőleges memóriarekesz értékének növelése:

   $x := x + 1$

3. Egy tetszőleges memóriarekesz értékének csökkentése (megegyezés szerint $x = 0$-ra $x - 1 = 0$ lesz):

   $x := x - 1$

4. Ciklus, amíg az x memóriarekesz értéke egyenlővé nem válik az y rekesz értékével:

   while $x \neq y$ **do**
   ... (A ciklusmag, amit végre kell hajtani.)
   wend

Ezek az alaputasítások nagyon egyszerűek, mégis sok minden kifejezhető velük.

3.3.2. Magasabb szintű műveletek

Ebben a pontban pár példát látunk arra, hogy milyen módon valósíthatunk meg közhasznú és ismert elemeket while-programokként. Ezzel szemléltetjük a while-programok nyelvének nagy kifejezőerejét.

1. Egy tetszőleges pozitív ($n \geq 0$) konstans betöltése egy regiszterbe ($x := n$).

   $x := 0;$
   $x := x + 1;$
   $x := x + 1;$
   ...

   Az $x := x + 1$ értékadást összesen n-szer kell végrehajtanunk.

2. Egy x változó értékének átmásolása egy y változóba ($x := y$).

   $y := 0;$
   while $y \neq x$ **do**
   $y := y + 1;$
   wend

3. Egy x változó megnövelése y változó értékével ($x := x + y$)

   $s := 0;$
   while $s \neq y$ **do**
   $x := x + 1;$ $s := s + 1;$
   wend

4. Összeadás ($z := x + y$)

   $z := x;$
   $z := z + y;$

   Lásd ezekhez a konstans betöltése változóba, illetve a változó növelése példákat.

5. Kivonás ($z := x - y$, ahol az eredmény 0 lesz, ha $x < y$): Az összeadáshoz hasonlóan valósítható meg (lásd annak levezetését), csupán az $x := x + 1$ utasítás helyett $x := x - 1$ kell.

6. Ha van összeadásunk és kivonásunk, akkor velük a szorzás és a maradékképzéses osztás (többszörös összeadással, illetve kivonással) könnyen megvalósítható.

7. A logikai adattípusok értékeit átalakíthatjuk. A 0 és az 1 a lehetséges értékeket jelöli. A 0 jelöli a logikai hamis értéket, míg 1 a logikai igaz értéket.

8. A feltételes utasítás while-programokkal a következőképpen fejezhető ki:
   Az utasítás, amit ki akarunk fejezni:

   if ⟨*kifejezés*⟩ **then** S **end**

   ahol ⟨*kifejezés*⟩ egy logikai kifejezés (értéke 0 vagy 1), S pedig a feltételtől függően végrehajtandó utasítás. A vele egyenértékű while-program a következő (s egy segédváltozó):

   $s :=$ ⟨*kifejezés*⟩;
   while $s \neq 0$ **do**
   S;
   $s := 0$;
   wend;

9. A kibővített feltételes utasítás while-programokkal a következőképpen fejezhető ki:
   Az utasítás, amit ki akarunk fejezni:

   if ⟨*kifejezés*⟩ **then** S_1 **else** S_2 **end**

   ahol ⟨*kifejezés*⟩ egy logikai kifejezés (értéke 0 vagy 1).
   A vele egyenértékű while-program a következő:

   $s_1 :=$ ⟨*kifejezés*⟩;
   $s_2 := s_1$;
   while $s1 \neq 0$ **do**
   S_1;
   $s_1 := 0$;
   wend;
   while $s_2 \neq 1$ **do**
   S_2;
   $s_2 := 1$;
   wend;

10. Nemrekurzív alprogramok hívása az alprogram utasításainak bemásolásával oldható meg. A rekurzív alprogramokat – egy verem segítségével – előbb át kell alakítani nemrekurzívvá, majd az előző – nemrekurzív alprogramokra vonatkozó – esetet kell követni.

3.4. Vezérlési megközelítések

A programok végrehajtása a használt programozási nyelv leírásának megfelelően sokféleképpen történhet. A következőkben bemutatunk néhány *vezérlési megközelítést*, amelyek a mai programozási nyelvekre jellemzők. Fontos látni, hogy ezek közül a megközelítések közül számos kölcsönösen kizárja egymást, míg egyes megközelítések természetesen megférnek egymással. Végül megemlítendő, hogy néhány megközelítés lehetséges összeházasítása a programozási nyelvek témakörének ma is aktív kutatási témáját képezik.

Megjegyezzük, hogy az egyes megközelítésekhez itt csak a vezérlésre vonatkozó jellemzőket foglaljuk össze. Fejezetünkben részletesen foglalkozunk majd az imperatív programozási nyelvek jellemző vezérlési szerkezeteivel, míg a deklaratív (logikai) és funkcionális programozási nyelvekről külön fejezetekben írunk.

3.4.1. Imperatív programozási nyelvek

Az *imperatív programozási megközelítés* a valós számítógépek olyan absztrakciója, amely a regiszterek és a memória fogalmával a Neumann-elvű számítógépek modelljén és a Turing-gépeken alapszik. A változók és az értékadás a memóriatartalom megváltoztatásának programozási absztrakciói. Az imperatív szemléletben egy változó egy memóriarekeszt jelöl ki. Ezeket a változókat névvel látjuk el, hozzájuk értéket rendelhetünk, illetve a hozzájuk rendelt értéket megváltoztathatjuk.

A program változóinak nevei, értékei, a program aktuális végrehajtási pontja (az éppen végrehajtott utasítás) együtt alkotják a *program állapotát*. Egy végrehajtás alatt álló program *állapot-átmenetek sorozatával* jellemezhető (Turing-gép modell). Az egyes állapotok közötti átmenetet értékadások és vezérlési utasítások sorozataként írhatjuk le (lásd [Wir 73] ide vonatkozó részeit).

Ezek alapján mondhatjuk, hogy egy imperatív program állapotokból és az azokat módosító utasításokból áll. Állapotátmenetek – bizonyos tekintetben jellemző – sorozatának absztrakciójaként létrehozhatunk eljárásokat. Az imperatív programozás eljárásokkal együttes használatát a szakirodalom *procedurális programozásnak* nevezi.

3.4.2. Deklaratív és funkcionális nyelvek

Míg az imperatív szemlélet szerint a programok egy bizonyos cél elérését megvalósító utasítások sorozatából állnak, addig a deklaratív, valamint a funkcionális nyelvek ettől lényegesen eltérő szemléletet követnek.

A *deklaratív (és logikai) nyelvekben* az imperatív nyelvekkel szemben nem azt mondjuk meg, hogyan kell egy feladatot megoldani, hanem azt, hogy mi a megoldandó feladat. Ekkor a feladat pontos megfogalmazásán, a specifikáción van a hangsúly, erre kell figyelnünk. A megoldás megkeresését a futtató környezetre bízzuk. Ilyen nyelvek például a Prolog és az SQL, melyekről a könyv más fejezeteiben olvashatunk.

A *funkcionális modellnél* szintén a feladat specifikációjának pontos megadásán van a hangsúly, viszont itt a program egy – matematikai – függvény kiszámítása. A program végeredménye a függvény eredménye. Ebben a modellben az imperatív modellben megszokott egyes elemek, így például a részeredmények kiírása, a függvénykiszámítás mellékhatásaiként tekinthetők. [McC 85] és [Bow 02] sok hasznos információt tartalmaz erről a témakörről.

3.4.3. Párhuzamos végrehajtás

A programok végrehajtása történhet egy végrehajtási szál mentén, de egyes programozási nyelvek lehetővé teszik a végrehajtás párhuzamosítását: ekkor a végrehajtás több szálon történik. A deklaratív nyelveknél ezzel a problémával nem kell foglalkoznunk, mivel a program végrehajtási környezete foglalkozik azzal, hogyan valósítsa meg a megadott feladatot, a párhuzamosítási lehetőségekről legfeljebb tanácsokat adhatunk, például szinkronizációs specifikáció formájában. Az imperatív nyelveknél a párhuzamos végrehajtás könnyebben modellezhető: a különböző programvégrehajtási pontokat tartalmazó állapotokkal együttesen több végrehajtási szálat adhatunk meg.

A vezérlés ekkor általában a programot futtató környezet – az operációs rendszer – felügyeletével és segítségével valósul meg. Az operációs rendszer biztosítja az egyes végrehajtási szálak (végrehajtási állapotok) ütemezését.

A párhuzamos programok elméletéről részletesebben Manna [Man 91] ír. A gyakorlati eszközöket, például a PVM könyvtárat [PVM 02] ismerteti.

3.4.4. Eseményvezérelt programozás

A program futási környezete és a program vezérlési szerkezete közötti kapcsolat tekintetében a 90-es években szemléletváltozást figyelhetünk meg. Ez nem általános érvényű, nem minden programozási nyelvre – programozási környezetre – jellemző, de a vezérlés kapcsán mindenképpen meg kell említeni.

A 90-es évekig az *algoritmikus végrehajtási megközelítés* volt a jellemző. Ennek alapötlete az volt, hogy a program elindítása után bizonyos feladatokat végez el (ez egy program lefordításától kezdődően egy házkezelőségi nyilvántartáson át mindenféle feladat lehet), és ehhez szükség esetén a felhasználótól adatokat kér be azokon a pontokon, ahol erre szükség van.

Az *algoritmikus modellt* követő programkód egy általánosan vázolt szerkezete a következő:

```
Főprogram:
  Változódeklarációk

  Inicializálás elvégzése
  Készítsd elő a feldolgozandó adatokat
  Olvass be a felhasználótól adatokat
  Dolgozd fel az adatokat
  Az eredményt írd ki
  Folytasd a feldolgozást
  Program vége.
```

A 90-es évektől, leginkább a grafikus ablakozó rendszerek elterjedésével főként az interaktív programoknál, az algoritmikus végrehajtási megközelítést fokozatosan felváltja az *eseményvezérelt programozási megközelítés*. Itt a program egy inicializáló, előkészítő rész után (amely magában foglalhatja például egy grafikus felhasználói felület megjelenítését) az operációs rendszertől úgynevezett *események* formájában kap visszajelzést arról, hogy mi törté-

nik a környezetben (például a felhasználó megnyomott egy billentyűt vagy elmozdította az egeret; esetleg egy másik program értesítést küldött egy fájl törléséről).

Az *eseményvezérelt modellt* követő programkód egy általánosan vázolt szerkezete a következő:

```
Főprogram:
  Változódeklarációk

  Inicializálás elvégzése
  Ciklus, amíg az utolsó eseményt be nem olvastuk
    Olvassuk be a következő eseményt az operációs rendszertől
    Dolgozd fel az eseményt annak típusának megfelelően

  Program vége.
```

A két modell közti lényeges különbség a környezet – így az ablakozó rendszer, az operációs rendszer – szolgáltatásainak felhasználásmódja közt keresendő. Az algoritmikus modellben a feladatot megoldó algoritmuson van a hangsúly és a környezetet csak bizonyos szolgáltatások elérése érdekében használjuk fel (ilyen szolgáltatás lehet például felhasználói adat beolvasása a billentyűzetről). Az eseményvezérelt modellben a környezet (például az ablakozó rendszer) szerepe a passzív szolgáltatói szerep mellett kiegészül egy aktív vezérlésmeghatározó szereppel: ez figyeli a felhasználó ténykedését és események formájában értesíti a programokat a felhasználó cselekedeteiről, ezúton irányítva a program globális vezérlését.

A fenti eseményvezérelt programmodellben szemléltetett szerkezet jellemző a legtöbb programozási nyelvben ma alkalmazott eseményvezérlési technikára. Egyes nyelvek azonban a korábban kialakult algoritmikus modellt támogató nyelvi eszközkészlet felhasználásával ettől eltérő megoldási módokat is nyújthatnak az eseményvezérelt programozás támogatására. A 70-es évekre már szabványosított COBOL programozási nyelv a `DECLARATIVES` (lásd a 8.2.2. részt) kivételkezelési mechanizmusával biztosíthatja az eseményvezérlést az egyes eseményekhez rendelt `DECLARATIVES` paragrafusok segítségével.

3.5. A vizsgált programozási nyelvek

A következőkben rövid áttekintést adunk a fejezetben vizsgált, a fejlődést – szerintünk – legjobban bemutató programozási nyelvekről.

3.5.1. Mondatszerű algoritmusleírás: COBOL

A COBOL programozási nyelvet még a blokkstruktúra megjelenése előtt (1966-ban, majd 1974-ben) szabványosították: a mai COBOL programok nagy része az 1974-es COBOL-74 szabvány szerint készült (kisebb részt képviselnek az 1985-ben szabványosított COBOL-85 szabványnak megfelelő programok). A COBOL nyelv szemléletmódját érdemes komolyabban is megvizsgálni egyszerűsége és elterjedtsége miatt (2002-ig bezárólag a szerte a világon használt programok programsorainak több, mint a fele COBOL nyelven készült, és évtizedünkben ez az arány előreláthatólag nem változik lényegesen; esetleg a 60 százalék

feletti részesedés kismértékű csökkenése prognosztizálható). Mi a COBOL-74 szabvány idevonatkozó részeit ismertetjük.

A COBOL program a szabványban rögzített szabályoknak megfelelő angol nyelvenmegfogalmazott *mondatok* sorozatából áll: ezek a nyelv utasításai. Eggyel magasabb szintena COBOL program *paragrafusokra* tagolódik. A paragrafus utasítások (a COBOL szóhasználatának megfelelően mondatok) sorozata. A paragrafus végét a következő paragrafus eleje (vagy egy úgynevezett szekció vége – lásd később) jelzi. A paragrafusok a COBOL nyelvben a többi programozási nyelvben megismert paraméter nélküli eljárásokhoz hasonlíthatók. Végrehajtásuk a paragrafus első mondatán kezdődik és a paragrafus végéig tart (hacsak hívásukkor mást meg nem adtunk – lásd később). A COBOL nem támogatja a rekurziót vagy a paraméterezett paragrafusokat. A COBOL program paragrafusai úgynevezett *szekciókat* alkotnak. Egy szekció is meghívható alprogramként. A szekció végrehajtása a szekció paragrafusainak végrehajtását jelenti, az első paragrafustól kezdve akár több paragrafuson át az `EXIT` utasításig, amelynek elérésével befejeződik a szekció végrehajtása és a vezérlés visszatér a hívás helyére. A COBOL programok szekciók sorozatából állnak.

Egy COBOL program szerkezetét a következő programrészlet szemlélteti:

```
EGYIK SECTION.
* Ez egy szekció a programban
      Utasítások.
      ...
 PARAe1.
* Ez pedig a szekció első paragrafusa.
      Utasítások.
      ...
 PARAe2.
* Ez pedig a szekció következő paragrafusa.
      Utasítások.
      ...
      EXIT
 PARAen.
* Ez pedig a szekció következő paragrafusa.
      Utasítások.
      ...

MASIK SECTION.
* Ez egy szekció a programban
      Utasítások.
      ...
 PARAm1.
* Ez pedig a szekció első paragrafusa.
      Utasítások.
      ...
```

```
PARAm2.
* Ez pedig a szekció következő paragrafusa.
      Utasítások.
      ...
      EXIT
PARAmn.
* Ez pedig a szekció következő paragrafusa.
      Utasítások.
      ...
```

A fenti példában két szekciót írtunk. Ha a vezérlés például a MASIK nevű szekcióra kerül, akkor annak utasításai hajtódnak végre előfordulási sorrendjükben. A vezérlés a PARAm1 nevű paragrafus elérésekor folytatódik annak a paragrafusnak a végrehajtásával, majd a PARAm2 paragrafus végrehajtásával. Ennek végén az *EXIT* utasítás hatására visszatérünk a szekció végrehajtásából a szekcióhívás helyére. Az *EXIT* utáni paragrafusok (a példában ilyen a PARAmn paragrafus) nem lesznek automatikusan végrehajtva. Ezt a paragrafust a szekció többi paragrafusa szükség esetén meghívhatja. Lehetőség van egy szekcióból más szekciók paragrafusainak meghívására is, bár ezzel vigyázni kell, mivel az úgynevezett *overlay* szekciók csak végrehajtásukkor vannak bent a memóriában, és ha megpróbálnánk egy ilyen szekció egy paragrafusát másik szekcióból meghívni, akkor futásidejű hibát jelző üzenetet kapnánk. Ez egykoron a virtuális memória kezelését segítette, ami hasznos volt, ha az operációs rendszer nem támogatta azt és kevés központi memória állt a programozó rendelkezésére. Ma a memóriaárak rohamos csökkenésével ez a megoldás idejétmúlt.

3.5.2. Strukturált programozás: a Pascal nyelv

A Pascal programozási nyelvet Niklaus Wirth a strukturált programozás eszközeinek támogatását szem előtt tartva tervezte meg. A nyelv tervezése 1968-ban kezdődött. Az oktatásban először 1972-ben használták fel az ETH Zürichen. A szabványos Pascal nyelv definícióját 1973-ban publikálták [Jen 74]. A nyelv a strukturált programozás módszereivel elkészített programok kódolására elegendő eszközt biztosított, bár egyes megvalósításai (például a Borland Turbo Pascal környezete vagy az ATARI ST számítógépekre készített ST Pascal/68000) a nyelvi szabványban rögzítetteknél sokrétűbb vezérlési szerkezeteket biztosítanak. Ezeknek a kiegészítő eszközöknek a használata a program hordozhatóságát nagyban gátolja, de mindegyikük helyettesíthető más, szabványos elemmel, illetve elemekkel.

A Pascal nyelvnek illik megemlíteni még egy megvalósítását: az UCSD P-System rendszert. A rendszert a University California San Diego egyetemen fejlesztették ki, még a hetvenes évek végén. A rendszer lényege a rendszerfüggetlensége: a Pascal-fordítóprogram egy úgynevezett *P-kódot* állít elő[6], amit egy úgynevezett P-kód végrehajtó *virtuális gép* hajt

[6] A P-kód és egy erre fordító Pascal-fordítóprogram leírása megtalálható Steven Pemberton és M. C. Daniels *Pascal: The language and its implementation – The P4 System* című könyvében[Pem 82]. A könyvben sorról sorra bemutatott Pascal-fordítóprogram a végrehajtási környezettel együtt – elérhető az Internet számos fordítóprogramokkal foglalkozó honlapján, a rendszer vagy a könyv adataira egy webkeresővel rákeresve. A könyv majd egy évtizeden át alapmű volt a fordítóprogram-készítés gyakorlati oldalával foglalkozó egyetemi és főiskolai kurzusok számára.

végre egy adott hardver-architektúrán. A rendszer Pascal nyelven készült, és teljes értékű eseményvezérelt grafikus felhasználói felülettel és számos hasznos könyvtárral volt ellátva felhasználóbarát programok készítésére. Akkoriban a rendszer elsősorban gyenge teljesítménye miatt nem tudott elterjedni az iparban. Ezzel szemben az egyetemeken sok fantáziát láttak az rendszer nyújtotta lehetőségek oktatási felhasználásában – a futtatásához alkalmas volt az akkortájt egyetemeken igen elterjedt Apple-II számítógépcsalád. Az ötletet a Java programozási nyelvvel kapcsolatban, sokkal hatékonyabb hardverre, a Java fejlesztői a 90-es évek közepén ismét feleleventették – és a megoldás sikeressége és hatékonysága mára már teljesen nyilvánvaló (a P-kód virtuális gépéhez hasonlóan a Java virtuális gép is verem alapú technológiára épül).

A Pascal nyelv egy blokkstruktúrát, strukturált programozási eszközöket támogató nyelv. Számos későbbi programozási nyelvnek őse, így a Modula-3 [Bösz 96] és az Ada [Nyé 98] programozási nyelveknek is. A fejezetben áttekintjük a Pascal vezérlési szerkezeteit, valamint kitérünk a Modula-3 és az Ada nyelvekben bevezetett kiterjesztésekre.

3.5.3. Hordozható assembly: a C nyelv

A Pascal-származék nyelvek néhány ismert képviselője után áttekintjük a C nyelv és leszármazottainak főbb nyelvi elemeit, rajtuk keresztül szemléltetve az egyes nyelvi fogalmak kikristályosodását a 70-es évek elejétől (a C nyelv születésétől) napjainkig (a C#[7] nyelvvel [Sch 02] bezárólag). A C-származék nyelvek vizsgálatának keretében három programozási nyelv ide vonatkozó vezérlési szerkezeteit fogjuk megismerni: a C, a Java és a C# nyelvét[8].

A C mint közös ős választása természetesnek tűnik. A Java nyelv elterjedtsége és a benne megalkotott biztonsági rendszer komoly indok a Java nyelv kiválasztására. A C# nyelv választása elsőre meglepőnek tűnhet a C++ nyelv [Str 00] elterjedtségére való tekintettel. Mégis, a C++ helyett a C# nyelvet választottuk, mivel a C# a C++ vezérlési szerkezeteivel megegyező, illetve annál gazdagabb választékot biztosít a programozó számára a vezérlési szerkezetek terén. Másrészt komoly érvek szólnak a C# nyelv mellett, akár az ipari alkalmazásban is, a C++-hoz viszonyított egyszerűsége miatt. Itt hasonló érveket hozhatunk fel, mint egykoron az Ada nyelvvel kapcsolatban, bár a helyzet itt nem olyan kiélezett, mint az Adánál volt. Az Adát – pontosabban annak elődjeit – az USA kormánya eredetileg egy többlépéses kutatás-fejlesztési verseny keretében fejlesztette ki párhuzamosan több résztvevővel, majd a céljaiknak legmegfelelőbb nyelvet választották győztesnek, és jópár évig egy Ada-fordító megléte, valamint a készülő projektek Ada-alapúsága döntő kritérium volt a kormány által finanszírozott szoftverprojekteknél, vásárlásoknál [Nyé 98]. Mégis, az Ada olyan pszichológiai korlátot emelt a programozók elé azzal, hogy a típus fogalmát a korábbi nyelvekhez képest lényegesen bonyolultabbá tették (ami a biztonságos programozáshoz szükséges volt), valamint sok részlet – például a mutató típusok megfelelőinek, az *access*

[7] A könyvben a C# nyelv elnevezését a név helyes angol kiejtésének megfelelően ragozzuk, ami egyes várakozásokkal ellentétben nem C-hash, hanem C-sharp (magyar kiejtése szí-sárp). A továbbiakban ahol kell, ehhez a szóhoz illesztjük a toldalékokat.

[8] Ide sorolhatjuk a J# nyelvet is, amely a Microsoft legújabb Java-megvalósítása. Mivel a nyelv vezérlés tekintetében megegyezik a JDK 1.1.4-es Java-változattal, ezzel nem foglalkozunk részletesen. Általában a Java nyelvre vonatkozóan leírtak érvényesek erre is.

típusok megvalósításának – elrejtésével nem alakulhatott ki a programozókban egy egységes kép programjuk működéséről. Ezt a helyzetet látva az USA kormányzata a 90-es évek közepén újraértékeltette az Ada kizárólagosságának kérdését a kormányprojektek keretében, és az Ada mellé felvette a C++ nyelvet a „megengedett" programozási környezetek közé.

Sok programozónak azonban a C++ is egyre bonyolultabbá válik: a Microsoft ezt észrevéve és a Java nyelvvel kapcsolatos kudarca[9] folytán kezdte el a C# nyelv tervezését és kivitelezését (a nyelvet nemzetközi szabvánnyá alakítva, így segítve a terjedését). A C# jelenlegi egyetlen komoly hiányossága a C++ nyelvvel szemben a sablon (vagy generic) típusok hiánya (ugyanez elmondható a Javára is). A Java ezirányú bővítése, szabványosítása folyamatban van, míg a C# nyelvnél még várni kell erre.

Száz szónak is egy a vége, a vezérlési szerkezetek tekintetében a C# választása a könyvben a C++ nyelvet teljes mértékben helyettesíti, hisz a C# a vezérlési szerkezeteknek egy, a C++-énál is szélesebb eszköztárát biztosítja.

3.5.4. Minden objektum: a Smalltalk

A Smalltalk nyelvet a Xerox PARC kutatói fejlesztették ki. Objektumorientált nyelv (szóhasználatát tekintve a metódushívás helyett egy adott objektumnak való *üzenetküldésről* beszél): a Smalltalkban minden objektum. Továbbá, mivel egységes rendszert akartak kifejleszteni, az operációs rendszert, a program futtató környezetét is bevették a nyelvbe speciális, előre elkészített objektumok és eszközök formájában, melyek közül a lényegesebbek a következők:

- Memóriakezelés (automatikus szemétgyűjtéssel)
- Fájlkezelés kifejezetten ilyen célú objektumokkal
- Képernyőkezelés – ablakos, egeres grafikus felhasználói felületet biztosítva
- A billentyűzet kezelése a felhasználótól való adatbekérés szabványosítása érdekében
- Valódi objektumorientált nyomkövető rendszer, amelynek segítségével a Smalltalk programok egy objektumként érhetik el önmagukat.

3.5.5. Egyéb vizsgált programnyelvek

A fejezetben számos más nyelvből is veszünk példákat. Ezen nyelvek a következők:

Modula-3 • a Pascal, illetve Modula-2 egyik legkésőbb elkészült kiterjesztése [Bösz 96].

Ada • az USA védelmi minisztériuma által kifejlesztett univerzális nyelv [Nyé 98].

Eiffel • Bertrand Meyer objektumorientált szemléletű, az iparban kevésbé, de még említést érdemlő mértékben elterjedt programozási nyelve.

[9] A Java nyelvet a Microsoft néhány lehetőséggel bővítve támogatta, például az eljárás-paraméterek helyett bevezetett *delegate lehetőséggel*. A Sun azonban a Microsoft és programozóik által hasznosnak tartott bővítésekben a Java hordozhatóságának elvesztését, illetve ennek lehetőségét látta, ezért beperelte a Microsoftot. A perből a Sun került ki nyertesen. Ezután a Microsoft több éves hallgatása következett, míg 2001-re kihozták a C# nyelvet és a *.NET környezetet*, amely kiváló alapot szolgáltatott a C#-nak.

CLU • kifejlesztése az MIT-n történt, Barbara Liskov nevéhez fűződik; felhasználási területei: a programozás-oktatás. Jelentőségeként említik még, hogy szemléletével sokban hozzájárult hozzá az X Window Rendszer kifejlesztéséhez. A nyelvről bővebben [Lis 81] ír.

FORTRAN • numerikus számításokra tervezett nyelv (körülbelül egyidős a COBOL-lal): bár sok még aktív FORTRAN program van, és számos aritmetikai, algebrai és más matematikai könyvtár készült a FORTRAN-hoz, használata egyre kevésbé jellemző [FOR 03].

3.6. Értékadás, aritmetikai utasítások

Ebben a részben áttekintjük a vezérlési szerkezetek fejlődését, néhány kiválasztott programozási nyelvet mint mérföldkövet kiválasztva, az illető nyelv lehetőségeit bemutatva és kiemelve a fontosabb jellemzőket.

3.6.1. A COBOL eszköztára

A COBOL nyelvben eredetileg – a fordítóprogram egyszerű elkészíthetősége érdekében – nem volt olyan típusú értékadó utasítás, amelyben az értékül adandó értéket összetett – például zárójelezett – aritmetikai kifejezés formájában adhattuk volna meg. A probléma megoldását részekre kellett bontani.

Konstans értékek változókba töltését a `MOVE` utasítással végezhettük el. Ugyanez az utasítás szolgált változók értékének más változókba töltésére. Tekintsük a következő példákat:

```
MOVE 23 TO A.
MOVE B TO C.
```

Az első utasítás a `23` numerikus konstanst az `A` változóba helyezi. A második utasítás a `B` változó tartalmát a `C` változóba másolja. A COBOL mondatokat ponttal kell lezárni; ezt láthatjuk a fenti példában is.

Aritmetikai műveletekre négy alapvető utasítást használhatunk:

```
ADD A TO B GIVING C.
SUBTRACT A FROM B GIVING C.
MULTIPLY A BY B GIVING C.
DIVIDE A BY B GIVING C.
```

Ezek jelentése rendre:

- Add hozzá `A`-t a `B`-hez, az eredményt tárold `C`-ben.
- Vond ki `A`-t a `B`-ből, az eredményt tárold `C`-ben.
- Szorozd össze `A`-t `B`-vel, az eredményt tárold `C`-ben.
- Oszd el `A`-t `B`-vel, a hányadost tárold `C`-ben.

A `GIVING` záradékkal lehetett megmondani, hogy az eredményt melyik változóban akarjuk elhelyezni (a példákban mindenütt `C` változót neveztük meg itt). A `GIVING` rész a változónévvel együtt elhagyható: a szabvány ilyenkor is rögzíti, hogy melyik változóban keletkezik az eredmény (például az

```
ADD A TO B.
```

utasítás önmagában az A és B változók összegét a B-be helyezi).

A COBOL újabb változatai rendelkeznek egy összetettebb kifejezések kiértékelésére is alkalmas *COMPUTE* nevű utasítással. Erre példa a következő:

```
COMPUTE A = ( A + B - C / ( A * B ) - A * B ) .
```

A fenti utasítás jobb oldalán megadott kifejezés értéke az A változóba kerül (ezt adtuk meg a bal oldalon). Megjegyezzük, hogy itt is csak a négy alapművelet használható: aritmetikai függvények (például a szinusz vagy koszinusz függvények vagy a gyökvonás nem használható: ezeket elemi lépésekből szükség esetén a programozónak kell megvalósítani, például a megfelelő függvények sorbafejtésével).

3.6.2. Egyszerű értékadás: a Pascal nyelv

A Pascal nyelv értékadásának szintaxisa nagyon egyszerű. Az utasítást a := (azaz *kettőspont-egyenlő*) jel vezeti be. Ennek bal oldalán annak a változónak a nevét kell megadni, amelynek az értékét módosítani akarjuk, jobb oldalán pedig a változó új értékét kell feltüntetni. A Pascal-származék nyelvek mind átvették ezt a jelet az értékadó utasítás jelzésére. Ezeknél a nyelveknél az = (egyenlőség) szolgál két érték egyenlőségének vizsgálatára – összehasonlító operátorként. A Pascal nem engedi meg a többszörös értékadást[10], míg egyes nyelvekben – például a CLU-ban – ez megengedett.

Az értékadás általános alakja:

⟨*változónév*⟩ := ⟨*kifejezés*⟩;

Ahol az utasítás végrehajtásakor először a jobb oldali kifejezés kiértékelésére kerül sor, majd értéke hozzá lesz rendelve a bal oldalon megnevezett változóhoz. Az értékadásnak fontos jellemzője a jobb, és bal oldalán álló érték, illetve változó típusának kompatibilitása, összeegyeztethetősége. Erről a könyvnek az adattípusokkal kapcsolatos részeiben részletesebben olvashatunk.

3.6.3. Értékadás a C nyelvben

Az értékadás a függvényhívással együtt a C-ben és a C-származék nyelvekben a kifejezés-utasítás témakörébe tartozik. Az értékadás olyan művelet, amely a jobb oldalán álló értéket bemásolja a bal oldalon megadott változóba.

Megjegyezzük, hogy a C nyelvtől származtatott nyelvekben az értékadás általában *operátorként* viselkedik (úgynevezett *kifejezés-utasításban* szerepelhet), így ezeknél a nyelveknél a többszörös értékadás természetes nyelvi elem. Szintén a C-származék nyelveknél vehetjük észre, hogy az értékadás a := (kettőspont-egyenlő) jel helyett csupán egy egyenlőségjellel van jelölve. Ezeknél a nyelveknél az egyenlőségvizsgálatot két egyenlőségjellel jelöljük.

Mivel az értékadás operátor, visszatérési értéke is van: a jobboldalán álló kifejezés értéke. Ezáltal az

[10] Többszörös értékadás alatt egy értéknek egyszerre több változóba való eltárolását értjük.

```
A = B = 23;
```

utasítás értékül adja mind az *A* mind pedig a *B* változónak a *23* értéket (lehetőség van tehát többszörös értékadásra). Kötését tekintve ez az utasítás a következővel egyenértékű:

```
A = (B = 23);
```

Ez a gyakorlatban azt jelenti, hogy értékül adjuk a *B*-nek a *23*-at, és lesz az értékadást tartalmazó kifejezés értéke is. A zárójelezett (vagyis legelőször kiértékelt értékadás) kifejezés értéke így *23*, ami az *A* változóban tárolódik.

Ezekben a nyelvekben az értékadás, kifejezésekről lévén szó, egyéb operátorokkal együtt is használható. Például az `A = A + 4;` utasítást írhatjuk a tömörebb `A += 4;` formában is.

3.6.4. A Smalltalk megoldása

A Smalltalk nyelvben az értékadás az egyetlen olyan művelet, amely nem üzenetküldés (metódushívás) formájában valósul meg [Gol 83]. Alakja a Pascal nyelvben megismerthez hasonlít. Az

```
A := 34.
```

utasítás például az `A` változót a `34` objektumhoz köti (minden esetben referencia módosításáról van szó).

A kifejezés-utasítás azonban másképp értelmezendő. Objektumokat és a nekik küldendő üzeneteket kell megadni. Azaz az

```
1 + 2.
```

utasítás az 1 egész szám objektumnak elküldi a + nevű üzenetet (ezt a metódusát hívja meg), az üzenet paramétere pedig a 2. A pont a sor végén a Smalltalk utasításokat lezáró karakter.

3.6.5. A többszörös értékadás és a CLU

A CLU-t, mint kutatási és programozás-oktatási célokra tervezett nyelvet a *többszörös értékadás* kapcsán érdemes megemlíteni, mivel ez a nyelv lehetővé teszi, hogy egy értékadás bal, illetve jobb oldalán több (de a két oldalon ugyanannyi) változót és értéket adjunk meg (a nyelvről részletesebben lásd a [Lis 81] kiadványt).

A CLU-ban az `X` és `Y` változók tartalmának megcserélése egyszerűen leírható a következő utasítással:

```
X,Y := Y,X
```

Az előállított kód helyesen fog működni. A más programozási nyelvekben a változók értékének felcseréléséhez szükséges segédváltozóval a programozónak nem kell törődnie.

3.6.6. Az értékadás szerepe a programokban

Az értékadás bár nagyon egyszerű utasításnak tűnhet, mégis a leglényegesebb művelet korunk programozási nyelveiben: megvizsgáltunk több programozási projektet, melyek forrás-

kódban rendelkezésre álltak, és azt a – nem meglepő – eredményt kaptuk, hogy a projekt forráskódját alkotó utasításoknak több mint fele értékadás! (A vizsgált projektek egy több mint negyedmillió COBOL forrássort tartalmazó alkalmazói programrendszer[11], valamint a Linux operációs rendszer általunk használt változata – egy Linux 2.4-származék és egy multimédia-szerver, illetve egy kódoló-dekódoló program). Annak fényében ez az eredmény nem is meglepő, hogy valójában az értékadással teszünk komoly lépéseket a feladat megoldásának irányába, a ciklus például ezzel szemben csupán egy-egy értékadás vagy alprogram megadott állapottéren, megadott tartományon való többszöri végrehajtását írja elő.

3.6.7. Az üres utasítás

Az *üres utasítás* (ami nem csinál semmit, illetve lehet speciális szerepe, lásd például a `NOP` szerepét a Motorola 68000-es assembly nyelvét ismertető részben) felfogható akár egy olyan értékadásként is, amelynek jobb és bal oldalán ugyanaz a változó van. A legtöbb programozási nyelv azonban biztosít külön üres utasítást: az Adában az utasítás neve **null**, a Pascalban és a C-származék nyelvekben a pontosvessző használható (bár utóbbiakban – C, C++, Java, C# – nincs komoly szerepe). Az Eiffel nyelvben a pontosvessző használható, de nincs valódi szerepe. A COBOL nyelvben az üres utasítást a `NEXT SENTENCE` mondattal jelöljük (jelentése: következő mondat).

Az üres utasításnak első ránézésre nem biztos, hogy látjuk az értelmét (az MC 68000-es `NOP`-ja a processzor pipeline-ját ürítette, de a magasszintű programnyelvekben a programozónak ezzel nem kell törődnie). Annyit azonban meg kell említeni róla, hogy mint utasítás viselhet címkét.

A CLU nyelv tervezői a fentiek alapján alakították ki azt az álláspontjukat, hogy az üres utasítás a CLU-ban felesleges, ezért a CLU nyelvben ilyen nincs is.

3.7. Szekvencia és blokk utasítás

A *szekvencia* az utasítások egymás utáni végrehajtásával kialakuló egyik alapvető vezérlési szerkezet. Ez a gyakorlatban az utasításoknak azok előfordulási sorrendjükben történő végrehajtását jelenti.

A szekvenciát, mint vezérlési szerkezetet a kezdetek óta megtalálhatjuk az összes imperatív programozási nyelvben. Míg az értékadások egy állapotváltozást írnak le a program állapotterén, addig összetettebb előrehaladást (a program állapotának több összetevőjét megváltoztatva) a szekvenciával érhetünk el.

A COBOL nyelvben a szekvencia az utasítások egymás utáni sorokba írásával érhető el. A szekvenciát alkotó egyes utasítások lezárására a pont szolgál. A COBOL követi az „egy sor egy utasítás" elvet azzal a gyengítéssel, hogy lehetővé teszi, hogy egy utasítást több sorba szétszedjünk a program jobb olvashatósága érdekében.

[11] A COBOL programban értékadásnak tekintettük az értékváltoztató aritmetikai utasításokat is, nemcsak az értékmásoló `MOVE` utasítást, mivel a máshol nincsenek ilyen megszorítások az értékadó utasítással kapcsolatban, mint amit a COBOL-ban megismerhettünk.

A legtöbb programozási nyelv a pontosvesszőt használja utasítások szekvenciájánál (látszólag az utasítások végének jelzésére, de mint látni fogjuk, a gyakorlatban ennél összetettebb a helyzet).

A szekvencia mint utasításszerkezet nyelvtörténeti fejlődésénél a következő szempontokat érdemes vizsgálni:

- A szekvencia vagy blokk lehet-e üres?
- A pontosvessző utasításlezáró vagy utasításelválasztó-e?
- Lehet-e blokkutasítást létrehozni?
- Elhelyezhető-e a blokkutasításban deklaráció, és ha igen, akkor bárhol-e vagy csak a blokk deklarációs részében?

Megjegyezzük, hogy ezek általános szempontok. A példaként felhozott programozási nyelvek ide vonatkozó részeinek elemzésekor (a későbbi részekben is) bemutatjuk az adott nyelv lehetőségeit. Megemlítjük az adott nyelvre jellemző elemeket, de azokról a szempontokról nem írunk, amelyek az adott nyelvben nem léteznek vagy nincs értelme róluk beszélni. Ezáltal a könyv olvashatóbb lesz, és mindenütt a lényegre lehet koncentrálni. Világos, hogy ha egy nyelvben nem lehet blokkutasítást létrehozni, akkor nem is lenne értelme arról írni, hogy helyezhetünk-e el benne deklarációkat. Ugyanez vonatkozik a ciklusok és az elágazások később következő tárgyalására is. Az Olvasó a nyelvi elemek részletes ismertetése után meg tudja adni az eldöntendő formában feltett kérdésekre a választ (amely lehetőségről az adott nyelvnél nem írtunk, nem is utaltunk rá más módon, az az illető nyelvben nincs).

A témakör tárgyalása során részletesen ismertetjük az egyes *nyelvcsaládok alapító programnyelveinek*[12] lehetőségeit. Ezeknél a teljességre törekszünk. Majd a nyelvcsalád egy-egy későbbi nyelvénél bemutatjuk a nyelvcsalád eredeti nyelvéhez képest történt újításokat. Ezeknél az újabb nyelveknél csak azokra – a vezérlési szerkezetekre vonatkozó – új elemekre térünk ki, amelyek eltérnek a nyelvcsalád központi, alapító nyelvénél leírtaktól. A nem érintett témáknál az Olvasó feltételezheti, hogy a nyelvcsalád alapítónyelvéhez képest nem történt lényeges változás. Amennyiben az Olvasó egyes nyelvekről részletesebb információkat szeretne olvasni, úgy ajánljuk a megfelelő nyelv hivatkozási kézikönyvében az ide vonatkozó részének az elolvasását. Ezt irodalomként a megfelelő helyeken feltüntettük.

3.7.1. Blokk utasítás a Pascalban

A Pascal nyelv vezérlési szerkezeteit általában – a **repeat** ciklus kivételével – úgy definiálták, hogy a ciklusmagjuk egyetlen utasításból állhat. Ha több utasítást akarunk olyan helyre írni, ahol a szintaktikai szabályok szerint csak egyetlen utasítás állhat, akkor a blokk utasítást kell használni. Ez egy **begin**...**end** utasításpárok közti utasítássorozatot jelöl. A **begin** jelzi a blokk kezdetét, az **end** pedig a blokk végét. Ez a fordítóprogram számára egyetlen utasításnak minősül, hatása pedig a benne felsorolt utasítások sorozatának végrehajtásaként értelmezendő.

A Pascal nyelven és a belőle származtatott nyelveken megírt programok olvasása során feltűnhet a sok pontosvessző, ami ezekben a nyelvekben az utasítások *elválasztására* szolgál,

[12] Ezek a COBOL, a Pascal, és a C nyelvek.

a C-származék nyelvekkel szemben, amelyeknél a pontosvesszőnek utasításlezáró szerepe van. A Pascalban egy blokk utasításba bezárt utasítássorozat utasításait pontosvesszővel kell elválasztani, míg a blokk végét jelző **end** előtti utasítás és az **end** között nem kell, hogy pontosvessző álljon, mivel ott a blokkon belül nincs mit elválasztani. Ha ott mégis pontosvesszőt írnánk, az nem hiba, de tisztában kell lenni annak jelentésével: a fordítóprogram úgy tekinti, hogy a pontosvessző után, de a blokkzáró **end** előtt egy üres utasítás áll, amely nem változtat a program jelentésén.

A blokk utasítás általános alakja:

begin
    ⟨*utasítás*$_1$⟩ ; ⟨*utasítás*$_2$⟩ ; ... ; ⟨*utasítás*$_n$⟩
end

Az utasítások tetszőleges utasítások lehetnek, akár újabb blokkok is. A blokkok a Pascalban tetszés szerint *egymásba ágyazhatók.*

A blokk utasításai szekvenciálisan, előfordulási sorrendjükben hajtódnak végre. A Pascal nem engedi meg tetszőleges blokkon belül lokális változók deklarálását. Erre csak eljárásokban, függvényekben és a főprogramban van lehetőség.

3.7.2. Szakítás a Pascal hagyományaival: az Ada nyelv

A Pascal hagyománnyal való szakítás jele a pontosvessző szerepének megváltozása: az Adában a pontosvessző nem elválasztja az utasításokat, hanem lezárja azokat [Nyé 98]. Vagyis minden utasítás végén kell lenni pontosvesszőnek. Még a strukturált utasításokat záró **end loop** vagy **end if** előtt (és után) is.

Az Adában a blokk utasítás formája is egyértelműbbé vált a Pascal vagy Modula-3 nyelvekhez képest. Nézzük erre a következő példát:

Csere:
declare
    Segedvaltozo: Integer;
begin
    Segedvaltozo := I; I := J; J := Segedvaltozo;
end *Csere;*

Látható, hogy itt a **declare** és a **begin** kulcsszavak között akár újabb változókat, szükség esetén eljárásokat vagy függvényeket deklarálhatunk. Az itt deklarált elemek csak ezen a blokkon belül láthatók. Látható továbbá, hogy a blokkoknak egy előttük álló címkével nevet adhatunk, és ezt a nevet a blokkot záró **end** utasítás mögött is feltüntethetjük. Ez nagyban segíti a program olvashatóságát. Az Ada minden strukturált utasításhoz bevezetett egy külön azt lezáró kulcsszót: például az **if** utasítást az **end if**, a **loop** utasítást az **end loop**, a többirányú elágazást megvalósító **case** utasítást az **end case** szerkezet zárja le. Ez mind a fordítóprogramnak, mind pedig a programozónak egyértelmű, jól olvasható forráskódot eredményez.

Az Adában nagy adatszerkezetek rövid, meghatározott időre történő lefoglalására (a blokk elejétől a végéig) a blokk utasítás nagyon jó eszköz volt[13]. Ennek szerepe ma, a hatékony fordítási módszerek miatt, háttérbe szorult (az adatfolyam-elemzés eredményeképpen a fordítóprogram rendelkezésére állnak a változók helyfoglalási időtartamának minimalizálásához szükséges információk és eszközök, a lefoglalást pedig a fordítóprogramok általában hatékonyabban tudják elvégezni, mint a programozó – nem utolsósorban a részletesebb információk, valamint a célarchitektúra jellemzőinek pontosabb ismerete miatt).

3.7.3. A C nyelvcsalád jellemzői

A C nyelv és a belőle származó nyelvek rendelkeznek blokk struktúrával: a Pascalban használt **begin–end** utasítások helyett nyitó és záró kapcsos zárójelek használatosak. Míg a C nyelvben ennek a szerkezetnek csak utasításcsoportosító szerepe van, addig a C++-ban, a C#-ban vagy a Javában az Adáéhoz hasonló a szerepe. Egy így definiált blokkon belül új deklarációk helyezhetők el, mint az Ada **declare** blokkjában. Az olvashatóság mellett itt is, mint az Adánál, meg kell említeni, hogy a C kivételével mindegyik nyelv (vagyis C++, a C# és a Java) megenged lokális deklarációkat *tetszőleges* blokkon belül, tetszőleges helyen. Az Adával szemben ezek a nyelvek (a C kivételével – ahol a beágyazott blokkok nem tartalmazhatnak deklarációkat) nem bontják fel a belső blokkokat deklarációs részre és (egy ezt követő) utasítástörzsre: deklaráció utasítások közé bárhova írható. Ennek eredményeképpen a blokkban deklarált változók a deklarációjuktól csak a hatáskörük végéig – ami általában az őket tartalmazó blokk vége – foglalják a memóriát. Korábban nagy adatszerkezetek rövid, meghatározott időre történő lefoglalására a blokkon belüli tetszőleges helyen való deklaráció nagyon jó eszköz volt. Ennek szerepe ma, a hatékony fordítási módszerek miatt – ahogy azt az előző pontban leírtuk – háttérbe szorult.

A pontosvessző a C nyelvben, ahogy a Javában és a C++ nyelvben vagy C# nyelvben *utasításlezáró* szerepet tölt be: lezárja az utasításokat, nem pedig elválasztja azokat. Kivétel ezek alól a blokk utasítás, amit ezekben a nyelvekben a Pascalban megismert **begin–end** utasítás-zárójelek helyett használt { és } utasítás-zárójelekkel képezünk. A blokk utasítást nem kell lezárni pontosvesszővel.

3.7.4. A Smalltalk blokk utasítása

A blokk a Smalltalkban rendkívül hatékony eszköz: *végrehajtható kódot reprezentál objektumként.* Nem Smalltalk fogalmi rendszerrel úgy is fogalmazhatnánk, hogy a blokk egy név nélkül definiálható függvény (metódus), amit más objektumoknak átadhatunk üzenetben paraméterként vagy ennek is küldhetünk üzenetet. Persze egy blokknak a Smalltalkban általában van neve. A blokkra nevével később is hivatkozhatunk, ami a megfelelő kódrész végrehajtását vonja maga után:

[13] Ezzel válaszoltunk arra a kérdésre is, hogy az Adában létrehozható blokk utasítás és ezek tartalmazhatnak-e lokális deklarációkat.

Tekintsük a következő példát, egy képernyőre író Smalltalk blokkot:

```
|kiir|
...
kiir := [ :x | Transcript showCR :x ].
```

A blokkdeklarációt szögletes zárójelek közé írjuk, a blokk neve és egy azt követő := jel után. A blokk deklarációja tartalmazza a blokknak küldhető üzenetek paramétereit, majd egy függőleges vonal után a blokk törzsét. A fenti blokk a képernyőt reprezentáló `Transcript` (előre definiált) objektumnak elküldi a `showCR` üzenetet (ez végzi a kiírást), az üzenet paramétereként pedig átadja a kapott paramétert, vagyis hogy mit kell kiírni.

A blokk használata a következőképpen történhet:

```
kiir value:'hello'
```

A Smalltalkban a blokkok használata nagyon sokrétű. Például egy felhasználói felületen megjelent nyomógombot definiálhatunk úgy, hogy átadunk neki egy blokkot paraméterként, melynek a nyomógomb megnyomásakor egy előre megadott nevű, paraméter-típusú üzenetet küld.

3.8. Feltétel nélküli vezérlésátadás

A feltétel nélküli vezérlésátadás a legkorábbi, már a FORTRAN0-ban is bevezetett, és egyben a legvitatottabb vezérlésátadási szerkezet. A FORTRAN0-ban az utasításokat numerikus címkékkel láthattuk el[14], és egy `GOTO` utasítással a program egy tetszőleges pontjáról egy tetszőleges másik (numerikus címkével ellátott) utasítására adhattuk a vezérlést.

Mi minden szól a `GOTO` ellen? Röviden: *sok minden.*

- Korlátlan alkalmazása a modern nyelvekben nemcsak megnehezíti, de lehetetlenné is teszi a fordítóprogram munkáját, mivel nincs értelme az egymástól független blokkok közötti ugrálásnak. Gondoljuk csak el mi lenne ilyenkor a végrehajtási veremmel? Mi legyen a lokális változók értéke? Rekurzív függvények esetén milyen hívási mélységbe ugorjunk?. A program kuszasága (az úgynevezett „spagetti kód") a megérthetőségét nagyban gátolja.
- A `GOTO` aláássa a legtöbb helyességbizonyítási eszköz alkalmazását. A helyességbizonyítási módszerek általában úgy működnek, hogy a programot a vezérlési szerkezeteknél felszabdalják, invariánsokat fogalmaznak meg és ezek segítségével bizonyítják – logikai eszközökkel – a program helyes működését. A `GOTO`-nál és címkéknél nagyon nehéz invariánsokat megfogalmazni (ha ezek bárhol előfordulhatnak), ezért használata nem tanácsos.
- A `GOTO` utasítást tartalmazó programozási nyelvek jelentésének precíz leírása sokkal nehezebb, de azért lehetséges.
- Nincs összhangban az emberi gondolkodással.

[14] Innen jön az eredeti BASIC nyelv azon jellemzője, hogy minden utasítást sorszámmal kell ellátni.

Egyes nyelvekben kényszerből kell `GOTO`-t használni (például ciklusból való kilépésre), de a legtöbb nyelv erre is ad megoldást (lásd később a C-ben a **break**-et).

Megjegyezzük, hogy ezek általános szempontok. A példaként felhozott programozási nyelvek ide vonatkozó részeinek elemzésekor (a későbbi részekben is) bemutatjuk az illető nyelv azon lehetőségeit, amelyek a nyelv részletesen ismertetett elődjétől eltérnek (a Modula-3 esetén ilyen például a Pascal). Bemutatjuk az adott nyelvre jellemző elemeket, de azokról a szempontokról nem írunk, amelyek a nyelvben nincsenek értelmezve, vagy amit a nyelv részletesebben bemutatott elődjénél már ismertettünk.

3.8.1. A COBOL eszközei

A COBOL nyelv a `GOTO` utasítással támogatja a feltétel nélküli vezérlésátadást. A `GOTO` utasítás általános alakja a következő:

`GOTO` ⟨*paragrafusnév*⟩.

Ennek hatására a vezérlés a megadott nevű paragrafusra kerül és ott folytatódik. A COBOL programok megírhatók `GOTO` használata nélkül is, a nyelv így is elég széles eszköztárat biztosít a programozónak tetszőleges programszerkezetek megfogalmazására. A COBOL ezen kívül lehetővé teszi *kiszámított* `GOTO` utasítások használatát is. Ezek általános alakja a következő:

`GOTO` ⟨*paragrafusnevek felsorolása*⟩ `DEPENDING ON` ⟨*változónév*⟩.

Az utasítás végrehajtásakor a `DEPENDING ON` után írt változó értékétől függően vagy az első vagy a második paragrafusnál folytatódik a végrehajtás.

A COBOL nyelv érdekessége az `ALTER` utasítás, amely önmódosító programok készítését támogatja. Az `ALTER` első végrehajtásakor másképp viselkedik, mint a későbbi végrehajtásokkor. Az utasítást nem ismertetjük részletesebben, mivel a legtöbb mai COBOL rendszer ezt a lehetőséget egyáltalán nem vagy csak komoly hatékonysági veszteség árán támogatja. Ennek oka az, hogy az önmódosító kód létrehozásakor módosítani kell a program végrehajtható formáját tartalmazó úgynevezett kódszegmenst, amit a legtöbb operációs rendszer írásvédetten kezel (ez részben védelmi okokkal[15], részben pedig a kódszegmensmegosztással magyarázható: ha egy programnak több példánya is aktív, akkor a futáskor közösen használhatják a megosztott kódszegmenst, csak az adataikat tároló adatszegmenst kell külön létrehozni és kezelni minden programhoz).

3.8.2. Feltétel nélküli vezérlésátadás a Pascalban

A Pascal nyelv tartalmazza a feltétel nélküli vezérlésátadás lehetőségét, a **goto** utasítást. Ehhez a programban *címkéket* kell deklarálni, és az ugrással csak a címkékre ugorhatunk.

[15] Védelmi okok alatt arra gondolunk, hogy egy program még véletlenül se írhassa felül saját kódszegmensét, mivel ez a legtöbb esetben memóriakezelési hibákra vezethető vissza a programban. Ezzel a megoldással a modern operációs rendszerek a programozót támogatják a hibakeresésben.

A Pascal további megszorításokat is tesz, mind nyelvi, mind fordítóprogram-megvalósítási szinteken a **goto** használatával kapcsolatban, ami a blokkstruktúra következményeként is magyarázható.

Eszerint a feltétel nélküli vezérlésátadás csak egy blokkon belül megengedett. Egyes fordítók lehetővé teszik egy blokkból valamelyik azt befoglaló blokkba való kiugrást **goto**-val, de egy beljebb levő blokkon belülre ugrás általában nem megengedett. Ez érthető, hiszen a blokkstruktúrával együtt kezelt végrehajtási verem tartalma blokkon belüli ugrásnál változatlan, külső blokkba ugrásnál szükség szerint csökkenthető, de egy tartalmazott blokkba való ugrásnál a végrehajtási verem bővítendő lenne a belső blokk változóival, ami értelmes módon nem megoldható.

3.8.3. Modula-3: búcsú a GOTO-tól

Végezetül megemlítjük, hogy a Modula-3 nyelvben nincs feltétel nélküli vezérlésátadás: nincs **GOTO** utasítás. A tervezők ezzel pontot tettek arra a vitára, hogy kell-e **GOTO** vagy sem. Véleményük szerint a **GOTO** utasításra nincs szükség.

A Modula-3 nyelvben említésre méltó vezérlésátadó utasítás az eljárásokból és a függvényekből visszatérő **RETURN** utasítás. Ez az utasítás befejezi a hívott alprogram (eljárás vagy függvény) végrehajtását, és visszaadja a vezérlést a hívás helye utáni sorra. A **RETURN** mögött megadható egy kifejezés is, amely függvények esetén értelmezett: a függvény visszatérési értékét lehet vele kijelölni. Emlékeztetünk rá, hogy a Pascal nyelvben nem volt ilyen utasítás: egy alprogram (eljárás, függvény) az utolsó utasításának végrehajtása után fejeződött be; a függvény visszatérési értékét pedig egy értékadással kellett kijelölni, nevezetesen a függvény nevének kellett értékül adni a visszatérési értékét (ami nem vonta maga után a függvényből való kilépést, amire a függvény utolsó utasításának végrehajtása után került sor).

A Modula-3 vezérlésének másik komoly eszköze a kivételkezelés, amivel külön fejezetben foglalkozunk (lásd a 8.3.5. részt). A **GOTO** helyett sok helyen itt is és más nyelvekben is a kivételkezelés használható a feladat megoldására.

3.8.4. Különleges vezérlési utasítások a C-ben

A C nyelv nyelvi szinten támogatja a feltétel nélküli vezérlésátadást, a **goto** utasítást. Ehhez deklarálni kell egy függvényen belül egy címkét, a címke nevének és mögötte egy kettőspontnak a megadásával. A **goto** utasítással lehetőség van erre a címkére ugrani. Mivel a címkék láthatósága az őket tartalmazó függvényre korlátozódnak, csak függvényen belüli vezérlésátadásra van lehetőség.

```
cimke:
  .... utasítások ...
  goto cimke;
```

Amennyiben összetettebb vezérlésátadási igényeink vannak, úgy a C könyvtárában levő *setjmp* és *longjmp* függvényekre van szükségünk. Ezek nem részei a C nyelvnek, de a C könyvtárban általában megtalálhatók.

A *setjmp* elmenti a végrehajtási verem aktuális méretét (de a tartalmát nem másolja át külön helyre!) és elmenti az aktuális programvégrehajtási pontot a paraméterében megadott helyre. A *longjmp* függvény segítségével valósíthatjuk meg a feltétel nélküli vezérlésátadást. Paraméterében egy *setjmp*-vel elmentett állapotleírót kell megadni. A *longjmp* ekkor a végrehajtási vermet a *setjmp*-nél érvényes méretűre méretezi (szükség esetén levesz róla hívási rekordokat), majd a vezérlést átadja a *setjmp* után következő utasításnak. Ez nem közvetlenül a C programozási nyelv része, viszont elterjedtsége miatt és példaként említése és ismerete hasznos. Ezúton ellenőrizetlenül ugyan, de lehetőségünk van függvények közötti ugrásra is.

A **continue** utasítás ciklusokban használható: hatására a ciklusmag végrehajtása megszakad a **continue** utasításnál és a ciklus következő lépése hajtódik végre (ha nem teljesül a kilépési feltétel).

A **break** utasítás a legbelső ciklusból azonnali kilépésre használható. Hatására a vezérlés a ciklust követő utasításnál folytatódik. A **break** utasítást használtuk a **switch** kulcsszóval bevezetett többszörös elágazásból való kilépésre is. Ott is a **switch** blokkot követő utasításon folytatódik a vezérlés.

A **return** utasítással lehet egy függvényből kilépni, vissza a függvényhívás helyére. A **return** utasítás után adhatjuk meg a függvény visszatérési értékét. A visszatérési érték nélküli függvények a többi nyelvben megismert eljárások (melyek deklarációjánál a **void** visszatérési típust adtuk meg). Amennyiben eljárásból térünk vissza, úgy a **return** után nem szabad visszatérési értéket megadni.

3.8.5. A Java újdonságai

A Java programozóknak ugyanazok a vezérlési szerkezetek állnak rendelkezésre, mint amiket a C programnyelvnél láthattunk (a **goto** utasítás kivételével). Lényeges különbség, hogy mivel a Java nyelv ismeri a logikai (**boolean**) adattípust, a feltételeknek mind logikai típusúaknak kell lenniük.

A Java is ismeri a címkézett utasításokat, de ezeket a **break** és **continue** utasításoknál használhatjuk annak megadására, hogy az utasítások melyik ciklusra vonatkoznak (nemcsak a legbelsőre vonatkozhatnak). Ilyenkor a **break**, illetve **continue** utasítások mögött meg kell adni annak az utasításnak (például **for** ciklusnak) a címkéjét, amelyre az adott **break** vagy **continue** vonatkozik.

A Java másik különlegessége a C-vel szemben az *utasítások elérhetőségének vizsgálata*. Például egy végtelen ciklust követően hiába írunk utasításokat, mivel a vezérlés soha sem kerülhet azokra az utasításokra. A Java-fordítóprogram a modern fordítási módszereknek köszönhetően felismeri azokat az utasításokat, amelyekre soha nem adódhat a vezérlés. Ezeknél az utasításoknál a Java fordító hibaüzeneteket ad, figyelmeztetve a programozót egy esetleges logikai vagy algoritmusbeli hibára. Például a

```
while (false) { x = 5; }
```

utasítás fordítási hibát eredményez, mivel a ciklusmagra soha nem adódhat a vezérlés.

vannak magasszintű eszközök a megszakítások kezelésére. Ilyen eszközök például a programozási nyelvek kivételkezelési mechanizmusai.

Ezzel befejeztük a Motorola 68000-es architektúrájának tárgyalását. A célunk nem assembly programozók képzése volt, hanem egy informális kép közlése az Olvasóval azért, hogy pontosabb elképzelése legyen arról, hogy mit biztosítanak a mai modern 32 bites mikroprocesszorok a programozónak mind utasításokban, mind pedig címzési, vezérlési, programozási lehetőségek tekintetében. A csökkentett utasításkészletű (RISC) modern mikroprocesszorok általában kevesebb utasítást és egyszerűbb címzési módokat biztosítanak, viszont ezeknél a processzoroknál nem ritka, hogy 16, 64 vagy 128 regiszter áll a programozó rendelkezésére a memóriahozzáférések számának csökkentése érdekében. De mint azt korábban írtuk, ezeknek a programozását inkább bízzuk a gyártóik által kifejlesztett optimalizáló fordítóprogramokra, és ne mi próbáljunk meg optimális kódot írni. Nagyon nagy a valószínűsége annak, hogy nem tudunk jobb kódot írni (sőt annak is, hogy sokkal rosszabb hatékonyságú kódot írunk), mint amit például egy optimalizáló C-fordító állítana elő a problémát leíró C nyelvű programból.

3.2.3. Az MC 68000-es assembly kód

A bemutatott Pascal kódból egy fordítóprogrammal[4] előállított Motorola 68000-es assembly kód egy ATARI ST számítógépen a következőképpen néz ki. A bemutatott programot az elterjedtebb assemblerekkel fordíthatjuk gépi kódú végrehajtható programmá (egy főprogramot is kell hozzá készítenünk). IBM PC tulajdonosok az Interneten elérhető ATARI ST szimulátorok valamelyikével és a szabadon elérhető assemblerekkel és szerkesztőprogramokkal (ez utóbbiak angol elnevezése linker program, linkage editor) próbálkozhatnak a program lefordításával és életre keltésével.

```
    .globl lnko
lnko:
  ; procedure lnko(p,q: integer; var eredmeny: integer);
  LINK      A6,#0      ; Lefoglalunk 0 bájtot a lokális változóknak.
  ;  Ekkor a következő hivatkozásokat használhatjuk:
  ;   (A6)     = Az A6 regiszter régi tartalmát tárolja.
  ;              (zárójelek között egy címregiszter az általa mutatott memóriacím
  ;              tartalmára hivatkozik.)
  ;   4(A6)    = A visszatérési címet tárolja (4 bájtnyira az A6 által mutatott
  ;              memóriacímtől).
  ;   8(A6)    = Az 'eredmeny' nevű formális paraméter címe.
  ;              (Cím szerinti paraméterátadás!)
  ;   12(A6)   = A 'q' paraméter értéke (2 bájtos; a korábbi elemek memóriacímek
  ;              voltak, és ezért foglaltak le 4 bájtot).
  ;   14(A6)   = A 'p' paraméter értéke.
```

[4] A fordítóprogramot – a Pascal nyelv egy szűkített változatára – a fejezet szerzője készítette. A Motorola 68000-es kódgenerátor később lett a programhoz illesztve úgy, hogy kihasználja a processzor adta lehetőségeket.

- Megadható-e olyan ág, amely akkor hajtódik végre, ha a szelektorkifejezés értéke egyik felsorolt ág értékével sem egyezik?
- Megadhatunk-e azonos értékeket több ágban is, és ha igen, akkor mi ennek a jelentése (behozhat-e nem-determinisztikusságot a programba)?
- Rögzített-e a kiválasztási ágak sorrendje vagy használhatunk-e a leíráskor tetszőleges sorrendet?
- A vezérlés átadódik-e az egyik ág végén a következő kiválasztási ágra? Ez egy nyelvtervezés-filozófiai kérdés is: arról szól, hogy az *egyes ágak valóban önálló programrészeket képviselnek-e*, amelyek közül csak az egyik vagy egyik sem hajtódik végre (ezt látjuk majd a Pascal nyelvnél, de a C#-nál is), *vagy inkább belépési pontot képviselnek*, ahonnan kezdve az egymást követő ágak előfordulási sorrendjükben hajtódnak végre. Ezt a szemléletet képviseli a C, a C++ és a Java nyelv **switch** utasítása, de a C# már nem!
- Mi állhat az egyes kiválasztási, elágazási ágak megadásánál; milyen módon adhatjuk meg a szükséges értékeket?
  * Egyszerre csak egy érték adható meg.
  * Több érték egyenkénti felsorolása is megadható.
  * Megadható-e érték-tartomány a rövidebb leírás érdekében?

Ezek is nagyon általános szempontok. A példaként felhozott programozási nyelvek ide vonatkozó részeinek elemzésekor bemutatjuk az adott nyelv azon lehetőségeit, amelyek a nyelv részletesen ismertetett elődjétől eltérnek. Bemutatjuk az adott nyelvre jellemző elemeket, de azokról a szempontokról nem írunk, amelyek az adott nyelvben nincsenek értelmezve, vagy amit a nyelv részletesebben bemutatott elődjénél már ismertettünk.

3.9.1. Elágazás a COBOL-ban

Elágazások a COBOL-ban az *IF–THEN–ELSE* utasítással kódolhatók. Erre a szerkezetre tekintsük a következő példát.

```
IF ⟨feltétel⟩ THEN
     ⟨utasítás1⟩
     ⟨további utasítások⟩
   ELSE
     ⟨utasítás2⟩
     ⟨utasítás3⟩.
```

Az utasítás jelentése a szokásos: ha az *IF* után megadott feltétel teljesül, akkor a *THEN* utáni utasítások hajtódnak végre az *ELSE*-ig. Ha a feltétel hamis, akkor az *ELSE* utáni utasítások hajtódnak végre (ha az *ELSE* ág létezik; ez elhagyható).

Felhívjuk a figyelmet arra, hogy csak az *IF*-et lezáró legutolsó utasítás végén van mondatvéget jelző pont. A többi utasítást nem szabad ponttal lezárni (különben a COBOL-fordító úgy értelmezi, hogy az IF utasítás a ponttal helytelenül lezárt utasításnál befejeződött). Az *ELSE* rész elhagyható, de ha van *THEN* ág, akkor az nem lehet teljesen üres. Ha

oda mégsem akarunk írni semmit, akkor a `NEXT SENTENCE` mondatot használhatjuk, mint üres utasítást.

3.9.2. Feltételes utasítás a Pascalban

A Pascal kétféle feltételes utasítást biztosít a programozónak. Az egyik a hagyományos **if–then–else** szerkezet, míg a másik a többirányú elágazás egy diszkrét típus értékétől függően (a **case** szerkezet).

Az **if–then–else** szerkezet általános alakja a következő:

if ⟨*logikai kifejezés*⟩
then ⟨*utasítás*$_1$⟩ **else** ⟨*utasítás*$_2$⟩;

Az utasítás jelentése mindenki számára egyértelmű: ha az **if** után álló logikai kifejezés értéke logikai igaz (*true*), akkor a **then** után megadott utasítás hajtódik végre, egyébként pedig az **else** utáni utasítás hajtódik végre. Mind a **then**, mind pedig az **else** után egyetlen utasítás állhat. Amennyiben valamelyik helyre több utasítást akarunk írni, úgy azt egy **begin–end** közé téve egy blokk utasítással megtehetjük.

Megjegyezzük, hogy az **else** ág az ⟨*utasítás*$_2$⟩ résszel együtt elhagyható.

A pontosvessző utasítás elválasztó szerepének megértéséhez tekintsük a következő két utasítást:

if ⟨*feltétel*⟩ **then** ⟨*utasítás*⟩;

valamint az

if ⟨*feltétel*⟩ **then** ; ⟨*utasítás*⟩

Az előbbinek jelentése nyilvánvaló: ha az **if** után álló feltétel teljesül (igaz), akkor a **then** utáni utasítás hajtódik végre. Az utasítás végén látható pontosvessző az **if** szerkezetet választja el a következő (ez esetben üres) utasítástól. A második példában a **then** után egy üres utasítás van, amit pontosvessző választ el a következő utasítástól, ami az **if** utáni feltétel értékétől függetlenül mindenképpen végre lesz hajtva.

Itt hívjuk fel a figyelmet Pascal definíciójának egy másik pontatlanságára (de a C-ről és a C++-ról is elmondható ugyanez), amit a szakirodalom *csellengő else* (angol kifejezéssel *dangling else*) néven ismer. Itt arról van szó, hogy egy **if** utasításba beágyazott **if** utasítás esetén nem mindig látni egyértelműen, hogy egy **else** utasítás melyik elágazáshoz is tartozik. Tekintsük erre a következő példát.

```
if (a <= b)
then
   if (a < b) then Write("a < b")
   else Write("a = b");
```

Ebben a példában a nagy kérdés: melyik **if** utasításhoz tartozik a feltüntetett **else** ág. A beljebb kezdés azt sugallja, hogy a belsőhöz, de a nyelv ezt nem definiálja (a beljebb írást

a programnyelvek különben sem követelik meg a programozótól, az csupán a kód olvashatósága érdekében szükséges). Ilyenkor egyetlen biztos megoldás van: minden utasítást, még az üres utasítást is beleértve külön **begin–end** blokkokban kell elhelyezni, így egyértelművé válik a belső **if** utasításhoz tartozó **else** ág tartalma és egyértelmű lesz az ezt követő, külső **if** utasításhoz tartozó, **else** ág tartalma is. A fenti példában, így a Pascalban mindenhol, az a szabály él, hogy az **else** rész mindig a legbelső **if** utasításhoz tartozik (ugyanez érvényes a C-ben és a C++-ban is).

3.9.3. Többirányú elágazás a Pascalban

Többirányú elágazást a Pascal nyelvben a **case** szerkezettel írhatunk. Az utasítás általános alakja:

case ⟨*diszkrét kifejezés*⟩ **of**
  ⟨$konstans_1$⟩: ⟨$utasítás_1$⟩;
  ⟨$konstans_2$⟩: ⟨$utasítás_2$⟩;
  ⟨$konstans_3$⟩: ⟨$utasítás_3$⟩;
  ...
  ⟨$konstans_n$⟩: ⟨$utasítás_n$⟩;
  else ⟨*utasítás*⟩
end

Az utasítás működése valószínűleg mindenki számára érthető: először a **case** utáni kifejezés értékelődik ki, majd a vezérlés az ott kiszámított értékkel megegyező értékű címkénél folytatódik. Amennyiben a **case** utáni kifejezés értéke egyik felsorolt konstans értékével sem egyezik meg, akkor az **else** utáni utasításra kerül a vezérlés (ez az ág a nyelvben megengedett, de nem kötelező – hiánya esetén a vezérlés egyik ágat sem érintve a következő utasításra adódik át).

A Pascalban a **case** után álló diszkrét (egész vagy felsorolási típusú) kifejezés típusa lehet *integer*, *char*, *boolean* vagy tetszőleges felsorolási vagy tartomány típus. Az alternatívákat leíró konstans értékek közt megadhatjuk konstansok felsorolását, résztartományát is. Például a *23..44* egy szabályos **case** ág szelektora lehet.

Mivel sok fordítóprogram a hatékonyság érdekében egy, a teljes lehetséges értéktartományt tartalmazó ugrási táblát állít elő a végrehajtható kódban, és az ugrási tábla méretének csökkentését a rövidített szóhosszon tárolt relatív ugrási címmel érik el, a **case** szerkezet méretének ez is komoly határt szabhat. A korábban említett Pascal-P4 például nem támogatta az egész típusú érték szerinti **case** elágazást a 16 bites relatív ugrás korlátai miatt. Ott egy konkrétabb, szűkebb értéktartományt kellett megadni az értékkijelöléshez.

3.9.4. Biztonságos elágazások: a Modula-3 újításai

A Modula-3 nyelv a Pascal egyik közvetlen továbbfejlesztése [Bösz 96]: Wirth szemléletének továbbvitele az egyszerűség szem előtt tartásával, a programozó által elkövethető hibák lehetőségének minimalizálásával.

A Pascal nyelv leírásában sok hibalehetőség származott abból a figyelmetlenségből, hogy a programozók elfelejtették egy **if** utasításban vagy egy **while/for** ciklusba írt utasítás új utasításokkal való bővítésekor a **begin–end** utasítás-zárójeleket kitenni. A Modula-3 az ilyen eredetű hibák kiküszöbölésére módosította ezeket a szerkezeteket: egyrészt bevezette az ezeket a szerkezeteket kötelezően lezáró **END** utasítást, másrészt lehetővé tette azt, hogy **BEGIN–END** utasítás-zárójelek nélkül ezekbe a vezérlési szerkezetekbe több utasítást is beírjunk (az utasításokat pontosvesszővel elválasztva).

Az elágazás ilyen módon a következő alakú lett:

IF ⟨*logikai kifejezés*⟩
  THEN ⟨*utasítás*$_1$⟩; ⟨*utasítás*$_2$⟩; ⟨*utasítás*$_3$⟩; ...
  ELSE ⟨*utasítás*$_{E1}$⟩; ⟨*utasítás*$_{E2}$⟩; ⟨*utasítás*$_{E3}$⟩; ...
END;

Látható, hogy itt a **BEGIN–END** utasítás-zárójelek teljesen kihagyhatók.

Bizonyos egymásba ágyazott elágazások egyszerűbb írásmódjára a Modula-3 bevezette az **ELSIF** kulcsszót is. Például a következő utasítás szerkezet egymásba ágyazás esetén olvashatatlan lesz. Ez több elágazási feltételnél könnyen előfordulhat:

IF ⟨*logikai kifejezés*$_1$⟩
  THEN ⟨*utasítások*$_1$⟩
  ELSE IF ⟨*logikai kifejezés*$_2$⟩
      THEN ⟨*utasítások*$_2$⟩
      ELSE ⟨*utasítások*$_3$⟩
    END
END

A Modula-3 nyelvben ezt egyszerűbben is írhatjuk az **ELSE IF** összevonásaként szolgáló **ELSIF** kulcsszóval:

IF ⟨*logikai kifejezés*$_1$⟩ **THEN** ⟨*utasítások*$_1$⟩
ELSIF ⟨*logikai kifejezés*$_2$⟩ **THEN** ⟨*utasítások*$_2$⟩
ELSE ⟨*utasítások*$_3$⟩
END

Az általános alak a következő:

IF ⟨*logikai kifejezés*⟩ **THEN** ⟨*utasítás*$_1$⟩; ⟨*utasítás*$_2$⟩; ...
ELSIF ⟨*logikai kifejezés*⟩ **THEN** ⟨*utasítás*$_{A1}$⟩; ⟨*utasítás*$_{A2}$⟩; ...
ELSIF ⟨*logikai kifejezés*⟩ **THEN** ⟨*utasítás*$_{B1}$⟩; ⟨*utasítás*$_{B2}$⟩; ...
..
ELSE ⟨*utasítás*$_{E1}$⟩; ⟨*utasítás*$_{E2}$⟩; ⟨*utasítás*$_{E3}$⟩; ...
END;

Tetszőlegesen sok **ELSIF** ágat írhatunk. Általános szabályként megfogalmazhatjuk, hogy amennyiben ebben a szerkezetben egyszerre több ág is igaz, akkor az elsőként elő-

forduló igaz ág lesz végrehajtva. Megjegyezzük, hogy ezt a feltételt sok kutató próbálja gyengíteni úgy, hogy nem az első, hanem egy véletlenszerűen kiválasztott igaz értékű ág végrehajtásra kerül sor. Ezzel a programba bizonyos mértékű nem-determinisztikusságot is belevihetünk, illetve párhuzamos végrehajtás esetén szóba jöhetne az összes igaz értékű ág párhuzamos elindítása és végrehajtása. A Modula-3 nyelv ezeket nem támogatja, mi is csak az érdekesség kedvéért említettük meg mint lehetséges kutatási témát az ez iránt érdeklődő Olvasóknak.

Látható, hogy ezzel a lépéssel – azaz minden egyes strukturált utasításhoz, így az IF-hez is, egy, az azt lezáró elemet bevezetve (ami esetünkben az END szócska) – a nyelv tervezői megoldották a csellengő ELSE problémáját is.

3.9.5. Biztonságos CASE a Modula-3-ban

A CASE utasítás hasonló módon, kicsit módosított alakkal, lehetővé teszi a BEGIN–END utasítás-zárójelek elhagyását az egyes ágaknál. Ezzel a CASE általános alakja az alábbi példa szerint módosul:

```
CASE ⟨diszkrét kifejezés⟩ OF
    ⟨konstans1⟩ => ⟨utasítások1⟩; (* A függőleges vonal ennek az ágnak az elején
                                     is előfordulhat, mint a többinél, de semmi
                                     változást nem eredményez a végrehajtásban. *)
  | ⟨konstans2⟩ => ⟨utasítások2⟩;
  | ⟨konstans3⟩ => ⟨utasítások3⟩;
    ...
  | ⟨konstansn⟩ => ⟨utasításokn⟩;
  | ELSE utasítások;
END
```

A konstans részek (⟨*konstans*$_1$⟩... ⟨*konstans*$_n$⟩) tartalmazhatnak egy vagy több konstans értéket, illetve résztartományt.

A konstans értékeknek nem kell lefedniük a diszkrét kifejezés összes lehetséges értékét. Ha sem a diszkrét kifejezés értékének megfelelő konstans, sem pedig az ELSE ág nem fordul elő, az futási hibát eredményez.

3.9.6. A C nyelv elágazási szerkezetei

A C nyelvben az elágazást az **if**–*then*–**else** szerkezettel kódolhatjuk. Ennek általános alakja a következő:

if (⟨*kifejezés*⟩) ⟨*utasítás*⟩;

valamint

if (⟨*kifejezés*⟩) ⟨*utasítás*$_1$⟩; **else** ⟨*utasítás*$_2$⟩;

Jelentése pedig: ha az **if** utáni zárójelek közt álló kifejezés értéke *igaz* (azaz nem nulla), akkor végre lesz hajtva az ⟨*utasítás*$_1$⟩ utasítás. Ha van **else** ág, akkor a kifejezés *hamis* értéke

esetén (azaz ha az nulla) az ⟨*utasítás*$_2$⟩ utasítás lesz végrehajtva. Mindegyik helyen csak egyetlen utasítást írhatunk. Amennyiben valahová több utasítást kell beírnunk, úgy azt kapcsos zárójelekbe zárt, blokkot képző utasításként írhatjuk a programba.

3.9.7. Többágú elágazás a C-ben

A többirányú elágazás a C-ben a **switch** utasítással valósítható meg. Ennek általános alakja a következő példán látható.

```
switch (⟨diszkrét típusú kifejezés⟩) {
    case ⟨érték1⟩: ⟨utasítások1⟩; /* break; */
    case ⟨érték2⟩: ⟨utasítások2⟩; /* break; */
    case ⟨érték3⟩: ⟨utasítások3⟩; /* break; */
    . . .
    default: ⟨utasításD⟩;
}
```

Jelentése a megszokott: a **switch** után zárójelbe tett kifejezés kiértékelődik, és a fordító megkeresi azt a **case** utáni címkét, amely egyenlő a kiszámított értékkel és a vezérlést ott folytatja (ha nincs ilyen címke, akkor a vezérlés a **default** ágra kerül). Felhívjuk a figyelmet arra, hogy a **case** címkéken a vezérlés *továbbcsorog* a következő **case** ágra, hacsak nem írunk valahova egy **break** utasítást, amelynek hatására a vezérlés elhagyja a **switch** blokkot és az azt követő utasításra kerül. Ez sok programhiba forrása lehet. Ezért is írtuk a **break** utasításokat – igaz, megjegyzésként, emlékeztetőnek – a fenti mintába. Egy utasítássorozat ezáltal több **case** ághoz is tartozhat, mint az alábbi példán is látható.

```
switch (⟨diszkrét típusú kifejezés⟩) {
    case ⟨érték1⟩:
    case ⟨érték2⟩: ⟨utasítások2⟩; break;
    case ⟨érték3⟩: ⟨utasítások3⟩; break;
}
```

Ebben a példában a diszkrét típusú kifejezés értéke akár ⟨*érték*$_1$⟩, akár ⟨*érték*$_2$⟩, az ⟨*utasítások*$_2$⟩ rész hajtódik végre, majd a **break** hatására a vezérlés a **switch** blokk utáni utasításra kerül. Amennyiben a diszkrét típusú kifejezés értéke ⟨*érték*$_3$⟩, úgy a vezérlés az ⟨*utasítások*$_3$⟩ utasításrészre kerül. A végén a **break** utasítás felesleges, de megengedett, mivel nincs több ág, nincs mire „továbbcsorogjon" a vezérlés (ugyanakkor a **break** ottani használata biztonságosabb programozási stílust jelent, mivel az elágazásnak egy új ággal való bővítésekor nem felejthetjük el a **break**-et kitenni az előző és egyben utolsó ág végére, mivel az már ott van).

3.9.8. Többágú elágazás a C#-ban

A C# nyelvben a **switch** (többirányú elágazás) utasítás változott a gyakrabban előforduló eset támogatásának irányába: a C#-fordító hibajelzést ad, ha valamelyik **case** ágat nem zártunk le egy **break** utasítással. Ennek oka az, hogy a legtöbb esetben nem akarjuk, hogy

a vezérlés „továbbcsorogjon" a következő ágra, *ez a jellemző*. A fordító ezért megköveteli, hogy minden ág le legyen zárva egy **break** utasítással. Amennyiben mégis azt szeretnénk, hogy a vezérlés a soron következő **case** ágra kerüljön, úgy ezt a **goto** utasítással érhetjük el (amit a C# nyelvbe direkt erre a célra tettek be – miközben a C# nyelvhez legközelebb álló Java nyelvből kihagyták). A **goto** utasítás hasonlóan használható, mint a C nyelvben, de vele kapcsolatban több komoly megszorítást kell megemlíteni:

1. Nem lehet egy blokk belsejébe ugrani (például egy for ciklus ciklusmagjába a ciklusón kívülről).
2. Nem lehet osztályok között „ugrálni".

A C# tervezői egyetlen helyen tartják jogosnak a **goto** használatát: az előbb említett „továbbcsorgás" biztosítására (és ehhez külön címkéket sem kell írni, mivel a **case** ágak címkéit használhatjuk **goto** címkékként). A **goto** más helyen történő használata is megengedett, de a strukturált programozás keretében rendelkezésünkre áll minden eszköz arra, hogy a **goto**-t a programunkban az előbb említett **switch** utasításon kívül máshol ne kelljen használnunk.

3.9.9. Feltételes utasítás a Smalltalkban

A Smalltalkban a feltételes utasítás egy logikai típusú objektumnak küldött üzenet. A logikai típusú objektumok logikai kifejezések eredményeként előállt logikai igaz (`true`) vagy hamis (`false`) értékek (ezek objektumok). Ezek az objektumok az `ifTrue:` és `ifFalse:` üzenetek fogadására vannak „kiképezve".

```
(a > 5) ifTrue: kiir value:'a értéke nagyobb 5-nél'.
```

A fenti utasítás kiértékeli az `(a > 5)` logikai kifejezést, melynek értéke logikai igaz vagy hamis lehet. Ennek az objektumnak ezután elküldünk egy `ifTrue:` üzenetet, mely után egy blokk (egy üzenetküldés) áll. A logikai objektum (az `a > 5` kifejezés értékét ábrázoló objektum) abban az esetben, ha értéke logikai igaz, végrehajtja a paraméterében megadott blokkot, azaz kiírja a korábban deklarált kiíró eljárásunkkal az `a értéke nagyobb 5-nél` szöveget a képernyőre. Az `ifTrue:` ellentéte az `ifFalse:`. Ezekkel lehet a hagyományos `if-then-else` szerkezetet kódolni a Smalltalkban. Tekintsük erre a következő példát.

```
(a > 5) ifTrue: kiir value:'a értéke nagyobb 5-nél'
        ifFalse: kiir value:'a értéke NEM nagyobb 5-nél'.
```

3.10. Ciklusszerkezetek

A ciklusokkal kapcsolatban egy-egy programnyelvben a következő szempontokat vizsgáljuk:

- Készíthető-e előre nem ismert lépésszámú ciklus a programunkban?
  - Ha igen, akkor van-e elöl-, illetve hátultesztelő ciklus?
  - A ciklusfeltétel (kilépési feltétel) logikai típusú kell legyen vagy lehet más típusú is?
- A ciklusmagot külön blokkba kell írni, esetleg külön eljárásba kell írni? (Lásd a COBOL paragrafusait.)
- Készíthető-e előre ismert lépésszámú ciklus a programunkban?
  - Ha igen, milyen típusú lehet a ciklusváltozó?
  - Milyen kifejezések lehetnek a ciklusváltozó kezdőértékét, határát és lépésközét megadó kifejezések?
  - Mikor történik a lépésszám meghatározása?
  - Módosítható-e a ciklusváltozó értéke a ciklusban, és ha igen, akkor ennek milyen hatása van a ciklusra?
  - Mi a ciklusváltozó hatásköre, definiált-e az értéke a ciklusból való kilépéskor?
  - Hányszor lesz kiértékelve a végső érték és a növekmény?
- Van-e általános ciklus a nyelvben (mint például az Ada **loop–end loop** ciklusa)
- Léteznek-e ciklusból kilépő utasítások, melyek a ciklusmagban bárhol elhelyezhetők? (Például a **break** vagy az **exit**.)
- Létezik-e a következő ciklus-iterációt kikényszerítő utasítás? (Mint például a C-ben a **continue**.)
- Készíthetünk-e bejáró (más néven iterátor) ciklusokat? (Mint amilyen például a C# nyelv **foreach** szerkezete.)

Ezek is nagyon általános szempontok. A példaként felhozott programozási nyelvek ide vonatkozó részeinek elemzésekor bemutatjuk az illető nyelv azon lehetőségeit, amelyek a nyelv részletesen ismertetett elődjétől eltérnek. Bemutatjuk az illető nyelvre jellemző elemeket, de azokról újból nem írunk, amelyek az illető nyelvben nincsenek értelmezve, vagy amit a nyelv részletesebben bemutatott elődjénél már ismertettünk.

3.10.1. A COBOL ciklusai

Az alprogramok (eljárások: COBOL szóhasználattal paragrafusok) hívására a *PERFORM*[16] utasítás használatos. Ennek számos formája létezik, ciklus is ezekkel az utasításokkal szervezhető. Paragrafusok hívására használható még a *GOTO* utasítás is (ez a feltétel nélküli vezérlésátadás, mint ismeretes: a COBOL nyelv támogatja ennek használatát).

[16] A *PERFORM* szó magyarul „végezd el"-ként értelmezhető.

A *PERFORM* legegyszerűbb alakja a következő:

```
PERFORM paragrafusnév.
```

Az utasítás hatására a megadott nevű paragrafus utasításai rendre végrehajtódnak, majd a paragrafus utolsó utasításának végrehajtása után a vezérlés a *PERFORM*-ot követő utasításon folytatódik.

Ha egymás utáni paragrafusok sorozatát akarjuk végrehajtani, akkor használhatjuk a

```
PERFORM paragrafusnév1 THRU paragrafusnév2.
```

alakú utasítást. Ez a `paragrafusnév1` paragrafusnál folytatja a végrehajtást, majd az összes ezt követő paragrafus is végrehajtódik a `paragrafusnév2` nevű paragrafussal bezárólag. Nyilvánvaló, hogy a `paragrafusnév2` paragrafusnak a `paragrafusnév1` nevű paragrafus után kell lennie (köztük lehet több másik paragrafus is).

A *PERFORM* utasítás használható ciklusok szervezésére. A legegyszerűbb, rögzített számú végrehajtást előíró *PERFORM* ciklus alakja a következő:

```
PERFORM paragrafusnév n TIMES.
```

Ahol `n` numerikus érték. Az utasítás hatására a megadott nevű paragrafus `n`-szer egymás után végre lesz hajtva.

Kilépési feltételhez kötött ciklus is a *PERFORM* utasítással szervezhető. Ennek alakja a következő:

```
PERFORM paragrafusnév UNTIL feltétel.
```

Jelentése: hajtsd végre a nevével megadott paragrafust, amíg az *UNTIL* után megadott feltétel nem teljesül.

Például a következő példán látható programrészlet addig olvas adatot a `READ1` paragrafusban, amíg (jó adat esetén) az `R-FLAG` 0-ra nem állítódik:

```
MOVE 1 TO R-FLAG.
PERFORM READ1 UNTIL R-FLAG = ZERO.
```

A számlálásos ciklus alakja a COBOL-ban:

```
PERFORM paragrafusnév VARYING ciklusváltozó
        FROM érték BY növekmény UNTIL feltétel.
```

Jelentése: add értékül a ciklusváltozónak a *FROM* után megnevezett értéket, majd hajtsd végre a nevével megnevezett paragrafust, amíg az *UNTIL* után megadott feltétel teljesül. A ciklusváltozó értékét a paragrafus végrehajtása után a növekménnyel módosítjuk.

Látható, hogy a COBOL nyelv megköveteli, hogy a *ciklusmagot külön paragrafusba helyezzük*; abba a paragrafusba, amelynek nevét a *PERFORM* után megadjuk.

3.10.2. Ciklusutasítások a Pascalban

A szabványos Pascal nyelv a programozó számára számos ciklusformát biztosít. A **while** az elöltesztelő ciklusutasítás. A **while** utasítás általános alakja:

while ⟨*logikai kifejezés*⟩ **do** ⟨*utasítás*⟩

Az utasítás jelentése: amíg a **while** után írt logikai kifejezés igaz, addig hajtsd végre a **do** után álló utasítást. A **do** után egyetlen utasítás állhat, több utasítást szükség esetén **begin–end** blokkba ágyazva írhatunk. Megjegyezzük, hogy így végtelen (soha be nem fejeződő) ciklust a következőképpen írhatunk:

while *true* **do** ⟨*utasítás*⟩

A *true* a Pascalban a logikai igaz konstans érték: mindig igaz értékű kifejezés.

A Pascal hátultesztelő ciklus utasítása a **repeat–until** utasításpár. Ennek általános alakja a következő:

repeat
⟨*utasítás*$_1$⟩ ; ⟨*utasítás*$_2$⟩ ; ... ; ⟨*utasítás*$_n$⟩
until ⟨*logikai kifejezés*⟩;

A **repeat–until** között áll a ciklusmag, a ciklusban végrehajtandó utasítások sorozata. Az utasítás úgy működik, hogy először végrehajtódik a ciklusmag, majd ezután ha az **until** után álló logikai kifejezés értéke hamis, akkor a ciklusmag végrehajtása és az azt követő kilépési feltétel-vizsgálat ismét megtörténik. Ez történik egészen addig, amíg az **until** után írt logikai kifejezés nem logikai igaz (*true*) értékkel tér vissza. Ezzel a ciklusutasítással végtelen ciklust a következő formában szervezhetünk:

repeat
⟨*utasítás*$_1$⟩ ; ⟨*utasítás*$_2$⟩ ; ... ; ⟨*utasítás*$_n$⟩
until *false;*

A Pascal eredeti Wirth-féle változata az előre nem ismert lépésszámú ciklusok szervezésére csak a fenti két ciklusformát biztosította. Számos fordítóprogram biztosítja az úgynevezett **do–loop** ciklust, mely több utasítást tartalmazhat a **do** és a **loop** utasítások között (a köztük levő utasítások alkotják a ciklusmagot). A ciklusból kilépni az **exit** utasítással lehet. Megjegyezzük, hogy az eredeti szabványosítás híján ennek az utasításnak több formája is használatos a különböző fordítóprogramoknál.

Hátra maradt még az előre ismert lépésszámú, úgynevezett *számlálásos ciklusok* tárgyalása. A Pascal nyelvben a számlálásos ciklus általános alakja a következő:

for ⟨*ciklusváltozó*⟩ := ⟨*érték*$_1$⟩ **to** ⟨*érték*$_2$⟩ **do** ⟨*utasítás*⟩;

illetve

for ⟨*ciklusváltozó*⟩ := ⟨*érték*$_2$⟩ **downto** ⟨*érték*$_1$⟩ **do** ⟨*utasítás*⟩;

ahol ⟨*érték*$_1$⟩ ≤ ⟨*érték*$_2$⟩ kell legyen.

Az első esetben a ciklusváltozó a ciklusba lépéskor felveszi az ⟨*érték*$_1$⟩ értéket, majd végrehajtja az egyetlen utasításból álló, **do** után megadott ciklusmagot (ami szükség esetén lehet egy blokkutasítás is), majd a ciklusváltozó értéke eggyel nő, és a vezérlés egészen addig a ciklusban marad, amíg túl nem lépjük a **to** után megadott ⟨*érték*$_2$⟩ értéket. A második forma lényegében ugyanígy működik: itt a ciklusváltozó az ⟨*érték*$_2$⟩ értéktől kezdve lefelé kezd el számlálni (−1 lépésközzel), egészen addig bennmaradva a ciklusban, amíg az érték1 értéket el nem érjük (pontosabban: nem lépünk túl azon).

A Pascal számlálásos ciklusának számos, a későbbi nyelvekben javított (vagy másképp megoldott) hiányossága van:

- A ciklusváltozó csak +1 (**to** kulcsszónál) vagy −1 (**downto** kulcsszónál) lépésközzel változhat.
- A ciklusváltozó értéke a ciklus után definiálatlan (értéke a fordítóprogramtól függ, illetve attól, hogy a fordítóprogram milyen kódot állít elő). Ez következménye annak, hogy a ciklusváltozó hatásköre az őt deklaráló teljes blokkra kiterjed, nem korlátozható magára a ciklusra.
- A ciklusváltozó megváltoztatása nem megengedett, de a fordítóprogram nem ellenőrzi ezt.
- A ciklusváltozó határa egyszer, a ciklusba való belépés előtt kerül kiszámolásra, nem pedig minden végrehajtás után. Ez a hatékony kód előállításához nagyon jó, hiszen itt a ciklusmag végrehajtásának a száma előre ismert (a ciklusváltozó értékének módosításával sem befolyásolható), viszont a más nyelvekhez szokott programozónak komoly hibák forrása lehet az a kimondottan rossz szokás, hogy a ciklusváltozó értékét módosítják a cikluson belül.

3.10.3. Modula-3: biztonságos ciklusok

A Pascalból megismert **WHILE** ciklusszerkezet is az elágazásoknál már látott megoldáshoz hasonlóan módosult: nincs szükség az utasításban **BEGIN–END** utasítászáró-jelekre több utasításból álló ciklusmag esetén. Egyszerűen írjuk a ciklusmagot a strukturált ciklusszerkezetbe és zárjuk le azt a végén egy **END** szócskával. A **REPEAT–UNTIL** ciklus változatlanul úgy használható, mint a Pascalban.

A számlálásos **FOR** ciklus formája leegyszerűsödött a **BEGIN–END** utasítás-zárójelek elmaradásával, és egyben a kifejezőereje is megnőtt, mivel a lépésköz tetszőleges lehet (és nincs szükség a negatív vagy pozitív lépésköz alkalmazásakor más ciklusutasítás használatára (lásd **to**, illetve **downto** a Pascalban).

A **FOR** ciklus alakja a Modula-3 nyelvben az alábbi példán látható.

FOR ⟨*ciklusváltozó*⟩ := ⟨*érték*$_1$⟩ **TO** *érték*$_2$ **BY** ⟨*lépésköz*⟩
DO
  ⟨*utasítás*$_1$⟩; ⟨*utasítás*$_2$⟩; ... ⟨*utasítás*$_n$⟩
END

A lényeges változás a **BY** rész megjelenése, ahol a lépésközt adjuk meg (0 lépésköz megadása végtelen ciklust eredményez). Ez a rész a lépésközzel együtt elhagyható. A Modula-3 tervezői továbbá megoldották a Pascal nyelvnél említett ciklusváltozó problémáját is:

- A ciklusváltozó a **FOR** utasításon belül a ciklusmagban látható, arra nézve lokális.
- A ciklusváltozó deklarációja a **FOR** utasításban való előfordulásával (megnevezésével) történik.
- A ciklus befejeződése után a ciklusváltozó megszűnik, így értékére a cikluson kívül nem hivatkozhatunk.
- A ciklusváltozó a cikluson belül nem módosítható. Módosítása fordítási hibát okoz.

3.10.4. loop-end-exit ciklusok

A Modula-3 szabványosította a Pascalnál csak megemlített DO–LOOP–EXIT ciklust.

A ciklus általános alakja az alábbi példán látható.

```
LOOP
    ⟨utasítások⟩
    ... EXIT ...
    ⟨utasítások⟩
END
```

A LOOP és az END közötti utasítások végtelen ciklusban futnak. A kilépés az EXIT utasítással lehetséges. Egy EXIT hatására a ciklus azonnal befejeződik, a végrehajtás pedig az END utáni utasításon folytatódik. Megjegyezzük, hogy az EXIT utasítással bármelyik ciklusból kiléphetünk (akár WHILE ciklusból is), de a strukturált programozás elveinek betartása érdekében ajánlatos az EXIT utasítás használatát a LOOP–END ciklusokra korlátozni! Az EXIT utasítások jellemzően IF utasításokban fordulnak elő a LOOP–END ciklusokon belül: ezzel a kilépést bizonyos feltételekhez köthetjük. Az is lehetséges, hogy a LOOP–END ciklusba egyáltalán nem írunk EXIT utasítást, ez végtelen ciklust eredményez. A végtelen ciklusoknak általában kevés értelme van, de bizonyos esetekben (például egy kommunikációs programban, ahol a feladat kommunikációs csomagok megállás nélküli fogadása és továbbítása) ennek is lehet értelme. Persze ez a ciklus sem fut a végtelenségig, hiszen ha kikapcsoljuk a számítógépet, akkor ez a ciklus is leáll.

3.10.5. Az Ada nyelv eszközei

Hasonló, igaz más alakú vezérlési szerkezeteket vezettek be az Ada programozási nyelvbe is a Pascal ciklusait bővítve. Mivel ezek elsősorban szintaktikai újításokat jelentenek, ezekkel nem foglalkozunk részletesebben (nem ismételjük önmagunkat).

Néhány újdonságot azonban, amit a nyelv tervezői a korábbiakhoz képest jobban vagy kifejezőbben meg tudtak oldani, megemlítünk.

A számlálásos **for** típusú ciklus az Adában hasonló szerepet tölt be, mint a már megismert programozási nyelvekben. Az Ada azonban számos ponton nagyon jól definiálta a számlálásos ciklus egyes jellemzőit. Ezek a pontok a következők:

- A ciklusváltozó a ciklus lokális *konstansa*.
- A változó a ciklusutasításban deklarált, a ciklus értéktartományához legmegfelelőbb, legszűkebb típusúként.
- A ciklusváltozó (és a tartomány, amit befut) diszkrét típusú kell legyen.
- A ciklusváltozó által befutandó értéktartományt csak egyszer, a ciklusba való belépés előtt számítja ki.
- Nyilvánvaló következménye a fentieknek, hogy a ciklusváltozó értékének ciklusból kilépéskori definiáltságáról nincs értelme beszélni, mivel az ott nem látható. Ha a cikluson kívüli blokkban deklarálva van egy azonos nevű változó, azt a ciklusban deklarált ciklusváltozó eltakarja (a cikluson belül arra a kívül deklarált változóra a deklaráló blokk nevével minősítetten hivatkozhatunk, de tisztában kell lenni vele, hogy az egy teljesen más változó).

A számlálásos ciklusutasítás általános alakja a következő:

for ⟨*ciklusváltozó*⟩ **in** ⟨*értéktartomány*⟩ **loop**
⟨*ciklusmag utasításai*⟩
end loop;

Látható, hogy a ciklusváltozónak az általa befutandó értéktartományt kell kijelölni. Ez lehet közvetlen egy létező adattípus (akár az egészek, akár egy felsorolási típus) teljes értékhalmazának a tartománya (az Adában erre a ⟨*típusnév*⟩**'range** módon hivatkozhatunk), vagy lehet egy értékhalmaz valamely részhalmaza: például az *1–N* tartományon (ahol *N* egy pozitív egész) az *1..N* értéktartományt megadva mehetünk végig. A **reverse** kulcsszóval fordított sorrendben mehetünk végig a megadott értéktartomány értékein. Ennek alakja a következő:

for ⟨*ciklusváltozó*⟩ **in reverse** ⟨*értéktartomány*⟩ **loop**
⟨*ciklusmagutasításai*⟩
end loop;

3.10.6. Ismétlési szerkezetek a C és a Java nyelvben

A C nyelvben is megvan a Pascalban már látott elöltesztelő ciklus, melynek formája:

while (⟨*aritmetikai-logikai kifejezés*⟩) ⟨*utasítás*⟩;

Működése: a ciklusban megadott utasítást egész addig végre kell hajtani, amíg a **while** után zárójelben megadott kifejezés értéke igaz (vagyis értéke nem nulla).

A hátultesztelő ciklus a C nyelvben a következőképpen néz ki:

do {
⟨*utasítások*⟩;
} **while** (⟨*feltétel*⟩);

Működése: a ciklusban megadott utasításokat végre kell hajtani. Egyszer mindenképpen, majd egész addig végre kell hajtani a ciklusmagot, amíg a **while** után zárójelben megadott kifejezés értéke igaz (vagyis értéke nem nulla).

A C nyelv is támogatja a számlálásos ciklust. A számlálásos ciklus alakja a következő:

for (⟨*inicializálás*⟩; ⟨*feltétel*⟩; ⟨*ciklusváltozó frissítése*⟩)
⟨*utasítás*⟩;

Ez a következő **while**-ciklussal egyenértékű[17]:

⟨*inicializálás*⟩;
while (⟨*feltétel*⟩) {
⟨*utasítás*⟩;
⟨*ciklusváltozó frissítése*⟩;
}

[17] Kivéve azt az esetet, amikor a program az *utasítás* részben **continue** utasítást hajt végre a feltétel kiértékelése előtt. Ekkor a **while** ciklus egyszerűen a feltételre ugrik, míg a **for** ciklusban még végrehajtja a ciklusváltozó frissítését, és utána megy a feltétel ellenőrzésére.

A **for** utasításban a pontosvesszővel elválasztott 3 elem egy-egy kifejezés lehet. Az inicializálás jellemzően egy értékadás, a feltétel a ciklusban maradás feltételét fogalmazza meg, a ciklusváltozó frissítése rész pedig tetszőleges kifejezés lehet: általában itt egy, a ciklusváltozó értékét növelő vagy éppen csökkentő kifejezés van, de bármilyen kifejezés megengedett.

A **for** típusú ciklussal kapcsolatban megemlítjük, hogy a Java lehetővé teszi, hogy a ciklusváltozót a **for** ciklus inicializáló részében deklaráljuk, ezáltal a változó hatáskörét és élettartamát a ciklusra korlátozhatjuk. Ezzel a ciklusváltozó hatásköre jól meghatározott, értéke pedig a ciklusból való kilépéskor már nem definiált (hiszen az a változó ott már nem látható). Nézzük erre a következő példát:

```
for (int i = 0; i < 100; i++) System.out.println(i);
```

Ezzel szemben ha a ciklusváltozót a cikluson kívül definiáljuk, akkor értéke a ciklusból való kilépéskor is elérhető. Nézzük erre az alábbi példát.

```
int i;
...
for (i = 0; i < 100; i++) System.out.println(i);
/* i értéke a ciklus után 100 lesz */
```

3.10.7. A C# nyelv újdonságai

A C# nyelv vezérlési szerkezetei sok tekintetben a Javáéhoz hasonlíthatók, illetve a C# sok mindent átvett a C nyelvtől.

A C# nyelv a korábban megismert ciklusokon kívül még egy ciklusváltozatot, az úgynevezett *iterátor ciklust* is a rendelkezésünkre bocsátja (bejáró ciklusnak is nevezik). Ezzel lehetőség van egy olyan számlálásos, ciklusváltozóval ellátott ciklus szervezésére, ahol a ciklusváltozó rendre felveszi egy olyan halmaz elemeinek az értékét, mely halmaz megvalósítja az *IEnumerable* interfészt (az interfészekről bővebben lásd a 12.7.6. részt, az *IEnumerable* interfészről pedig lásd a C# nyelv és alapkönyvtár hivatalos leírását).

A megadott célra a **foreach** ciklusutasítás szolgál. Példa a használatára:

```
int[ ] Egészek = { 1, 2, 3, 5 };
foreach (int x in Egészek) {
    Console.WriteLine(x);
}
```

A fenti utasítás hatására az x változó értéke végigmegy az *Egészek* nevű tömb összes elemén és kiírja az értéküket (vagyis a program külön-külön sorba kiírja az 1, 2, 3, 5 számokat). Az *x* változó értékét a ciklus elején deklaráltuk, a cikluson belül nem módosítható. Ha a tömb elemeit módosítani akarnánk, akkor a hagyományos **for** ciklust kell használnunk. Ezt a ciklust nevezzük *iterátor ciklusnak* vagy bejáró ciklusnak.

A tömbök implementálják az *IEnumarable* interfészt, ami annyit jelent, hogy rendelkeznek egy *GetEnumerator* nevű metódussal, ami az *IEnumerator* interfész egy példányát adja vissza. Ezen interfészen keresztül férhetünk hozzá a bejárandó halmaz elemeihez. Az interfész három metódus megvalósítását írja elő (ezek a *MoveNext*, *Current*, *Reset* metódusok). Ezek a metódusok sorolják fel a halmazban levő elemeket. A tömbök automatikusan

implementálják ezt az interfészt, de mi is készíthetünk hasonló szerepű osztályokat, amelyek megvalósítják ezt. A következő példa egy, az összes páros számot felsoroló osztályt mutat be.

```
public class PárosGenerátor : IEnumerable {
    int max;                    // felső határ
    public PárosGenerátor(int max) { this.max = max; }
    public IEnumerator GetEnumerator() { return new Mutató(max); }

    public class Mutató : IEnumerator {
        int max;    int aktuális = -2;
        internal Mutató(int max) { this.max = max; }
        // Aktuális elem lekérdezése.
        public Object Current { get { return aktuális; } }
        // Felsoroló objektum újrainicializálása.
        public void Reset() { aktuális = -2 };
        // A következő páros szám megkeresése.
        public bool MoveNext() {
            if (aktuális > max)
                return false;       // Nincs több max-nál kisebb páros szám.
            aktuális = aktuális + 2;
            // Egy páros számnál kettővel nagyobb szám a következő páros szám.
            return true;            // Találtunk új páros számot.
        }
    }
}
```

Egy adott tartományon belül (esetünkben 0-tól 100-ig) a páros számokon a következő **foreach** utasítással mehetünk végig:

```
foreach (int érték in new PárosGenerátor(100)) {
    Console.WriteLine(érték);
}
```

A fordítóprogram automatikusan létrehozza a szükséges metódushívásokat, egy ezzel egyenértékű **while** utasítás formájában (az ennek megfelelő **while** ciklusban először a generátor lesz inicializálva, majd amíg a *MoveNext* metódus logikai igaz értékkel tér vissza, addig az érték nevű változó felveszi a *Current* metódus által visszaadott aktuális értéket, amit a ciklusmagban tetszőlegesen használhatunk).

Ilyen módon könnyen írhatunk egy olyan ciklust, amely például az összes prímszámon halad végig.

3.10.8. Bejárók (iterátorok)

A C# nyelv példáján láthattuk az bejáróciklusok lényegét. Ebben a részben a bejáró fogalom mélyítéséről lesz szó.

A bejárók és a CLU

Megjegyezzük, hogy a bejáró fogalmát a CLU nyelv már korábban bevezette. A lényege ugyanaz volt, mint amit a C#-ban láttunk: egy programban kiszámított függvény értékkészletén kell végigmenni. A megvalósítás azonban kicsit más. A bejárót itt is egy önálló függvényként írjuk meg. Ez a függvény a C# megoldásához hasonlóan a következő értéket adja vissza – egy kifejezetten erre a célra kialakított `yield` nevű utasítással (a szerepe `return` szerepéhez hasonlítható). Egy teljes példaprogramtól, a CLU – nem idevonatkozó területű, de – nagyon egyéni jelölésrendszerére való tekintettel eltekintünk.

Az alábbi példában látható kódrészlet egy CLU iterátor szerkezetét szemlélteti.

```
minden_prim = iter () yields ( int ) % iterátor-függvény deklarálása
  p : int              % p egy egész típusú változó
  yield(2)             % visszaadjuk az első prímet
  while true do
    % ide jön a következő prímet kereső kódrész
    % (p változóba teszi azt)
    yield (p)          % visszaadjuk a következő prímet
  end
end minden_prim
```

Ezek után a következő utasítással az `i` nevű ciklusváltozó végigmegy az összes prímen:

```
for i : int in minden_prim() do ....
```

A bejáró mint vezérlési szerkezet absztrakció

Láthattuk, hogy a programozási nyelvek egy része biztosítja a bejáróciklus utasítás valamilyen formáját. Ezek a nyelvek lehetővé teszik adatelemek valamilyen módú generálását és feldolgozását a bejáróciklussal. Ehhez egyszerűen meg kell adni az első elem generátorát, illetve egy rákövetkező elem generátorát. A bejáróciklus utasítás ezeknek a felhasználásával tetszőleges adatszerkezet adatelemeit bejárhatja és feldolgozhatja.

Vegyük észre, hogy itt valójában általános algoritmus-osztályokról van szó: a bejárók teremtik meg a kapcsolatot az algoritmusok és az adatelemeket tartalmazó adatszerkezetek (tárolók[18]) között. A bejáró algoritmusnak egy jól meghatározott feladatot kell ellátnia: az adatszerkezetek elemeinek bizonyos sorrenden alapuló bejárását és mindezt anélkül, hogy pontosabban ismernie kellene az éppen bejárt adatszerkezetet. Az éppen bejárt adatszerkezetről csak azt kell tudnia, hogy milyen módon bocsátja rendelkezésünkre egy-egy adatelemét. Az adatszerkezetek készítőinek pedig csak azt kell tudniuk, hogy a bejárók milyen módon – milyen interfészen keresztül – várják a működésükhöz szükséges adatelemeket.

Az előbb használt algoritmus-osztály szó könnyen eszünkbe juttatja az objektum-orientált programozási nyelvek osztályfogalmát, mint az absztrakt adattípusok és az adatabsztrakció központi eszközét. A C++, a C#, a Java és más programozási nyelvek tervezői is ta-

[18] Az angol terminológiában használják rá általánosságban a container (konténer) kifejezést, ez leginkább a C++ nyelvre jellemző, míg a Java környezetekben és a C# esetében a gyűjtemény (collection) kifejezés terjedt el ugyanezen eszköz megnevezésére.

nulmányozták ezt az algoritmus-absztrakciós technikát, és új nyelvi eszközök[19] bevezetése nélkül kidolgoztak egy általánosan használható módszert összetett absztrakt adatszerkezetek tartalmának bejárását végző algoritmusok általánosítására. Ezeket az algoritmusokat bejáróalgoritmusoknak nevezzük (elterjedt kifejezés még az iterátoralgoritmus is). Az említett nyelvek a nyelv specifikációjának részeként, de a szabványos könyvtárba kihelyezve biztosítanak ilyen célú úgynevezett bejáró osztályokat.

A C++ nyelv szabványos könyvtára, az STL a leggazdagabb és legjobban kidolgozott bejáró programszerkezetek tekintetében. A C# és a Java szintén tartalmaznak ilyen elemeket. A Java nyelvben először a *java.util.Enumeration* interfész jelent meg iterátorok nyelvi támogatására, majd a gyűjtemény keretrendszer[20] megjelenésével a Java számos olyan lehetőséggel gazdagodott, amely korábban csak a C++ programozók számára volt elérhető. A Java gyűjtemény keretrendszeréből indultak ki a C# nyelv tervezői is, és egy ahhoz hasonló kifejezőerejű keretrendszert dolgoztak ki.

A C++ nyelv operátor-túlterhelési lehetőségét jól felhasználva a C++ szabványos könyvtárának, az STL-nek a tervezői olyan természetes módon biztosítják a programozónak bonyolult absztrakt adattípusok feldolgozását, ahogyan azt az elemi adattípusoknál is megszokhatták. Például a ++ operátor az egész számok körében az eggyel növelést jelenti: az egész típusú változó tartalmát a rákövetkező egész számra állítjuk. Egy lista esetében a ++ operátor jelentheti egy listabejáró algoritmusban a rákövetkező listaelemre léptetést. Azokban a nyelvekben, ahol nincs lehetőség operátor-túlterhelésre, ott ezeket a lépéseket a megfelelő metódusok meghívásával végezhetjük. Ez néha nehezebben olvasható kódot eredményez, mint az operátor-túlterhelést is használó megoldás.

Tekintsük a következő egyszerű bejárót, mely egy adatszerkezet elemeinek összegzését végzi. A bejáró C++ sablonként van definiálva; az összegzés pontos jelentése és az egyes lépéseknél végrehajtott műveletek attól az adattípustól függenek, amit a sablon paraméterként megadunk.

```
template<class C> typename
C::érték_típus összegez(const C& gy)      // A gy paraméter az elemek gyűjteménye.
{
    typename C::érték_típus összeg = 0;    // Kezdetben az összeget nullázzuk.
    typename C::const_iterator bejaro = gy.begin(); // Kezdés az elején.
    while (bejaro != gy.end()) {           // Amíg el nem érjük a végét.
        összeg = összeg + *bejaro;         // Összeg növelése az aktuális elemmel.
        ++bejaro;
    }
    return összeg;
}
```

Vegyük észre, hogy a * operátor a bejáró indirekcióját jelöli (a bejáró – iterátor – által hivatkozott aktuális elem), míg a ++ operátor a bejáró léptetését jelenti: az *összegez* metódus paraméterében megadott gyűjtemény aktuális elemének a rákövetkezőjére állítását. A

[19] Ez a megoldás csak a nyelv már meglevő utasításaira épül.

[20] Angol elnevezése: collections framework.

rákövetkezési relációt a gyűjtemény *C* osztályá készítője határozza meg. Az *érték_típus* az összegzendő adat típusát adja meg[21], ami jellemzően sablonparaméter. A *const_iterator* típus egy olyan iterátor típusát deklarálja, amelynek használatakor nem változtathatjuk meg a segítségével bejárt gyűjtemény tartalmát, elemeit. Ennek vannak a példában megadott *begin* és *end* metódusai, melyek rendre a bejárási sorrendben legelső és legutolsó elemet adják vissza. Ezekről a C++ nyelvi specialitásokról, a sablonokról és az operátorokról, a 11. és 5. fejezetekben olvashatunk részletesebben.

Az STL a bejárók öt lényeges típusát különbözteti meg (az egyszerű bejárón kívül, amely egy önálló elemre hivatkozik – rákövetkező vagy más vele relációban levő elem definiálása nélkül). A legegyszerűbb bejáró forma az olvasó bejáró és az író bejáró: ezek mindegyike egy menetben teszi lehetővé a gyűjtemény bejárását. Az olvasó bejárónál nem módosíthatjuk a gyűjteménynek a bejáróval elért elemeit (azok tartalmát), míg az író bejáróknál lehetőség van a módosításra (felülírásra, bővítésre), viszont az értékének hivatkozására nincs mód. A gyakorlatban ez a megkülönböztetés azon alapszik, hogy mely operátorokat kötelező definiálni az egyes bejáró-típusoknál. Míg író bejáróknál a **p=* és a *++* operátorokat, addig az olvasóknál az *=*p*, a *->*, *++* és az *==* és *!=* operátorokat kötelező megvalósítani.

Az író és az olvasó bejárók speciális formái az előre haladó iterátorok. Ezek lehetővé teszik a bejárt gyűjtemény elemeinek többszöri bejárását. A kétirányú iterátorok visszafelé haladást is biztosítanak a bejárt gyűjtemény elemei között a *--* operátor definiálásával.

A bejárók legspeciálisabb formája a véletlen elérésű bejáró, mely lehetőséget biztosít relatív címzésre[22], elemek távolságának meghatározására.

A bejárók megvalósításával nem foglalkozunk. Egy szabványos könyvtár specifikációja általában nem foglalkozik azzal, hogy egy adott könyvtári elemet milyen módon implementálnak. A specifikáció csak a kívülről hívható felületet rögzíti. Az már a programozó dolga, hogy egy bejáró megvalósításakor kiválassza a legjobb eszközöket a rendelkezésre álló eszközök közül. Például egy halmaz elemeinek az összes permutációját visszaadó bejárót el lehet készíteni egy szekvenciális alprogramként is, de párhuzamos programozási eszközöket biztosító környezetben jogosan merül fel a párhuzamosan futó programszálként történő megvalósítás lehetősége is (lásd a 14. részt).

3.10.9. Ciklusutasítás a Smalltalkban

A Smalltalkban a ciklusutasítás az elágazásnál megismertekhez hasonlóan kezelhető (elöltesztelő **while** típusú ciklusutasítása van, illetve egy tömbbejáró-ciklusa). Az egyetlen lényeges különbség, hogy a ciklusfeltételt egy blokk kell, hogy kiértékelje, amely minden iteráció után újra kiértékelődik. Ennek a – logikai értéket visszaadó – blokknak a `whileTrue:` üzenetet küldve, paraméterként a ciklusmagot mint blokkot megadva, eljutunk a hagyományos **while** típusú szerkezethez.

[21] Az STL-ben *value_type*-ként szokás erre hivatkozni.

[22] Ennek a bejárótípusnak az elemi adattípusokon megvalósított párját a mutató-aritmetikában találhatjuk meg. Nem véletlen, hogy az ilyen bejáróknál a mutató-aritmetikai műveleteket is mind meg kell valósítani.

Tekintsük erre a következő példát.

```
|deklaraltValtozo|
...
deklaraltValtozo := 1.
[deklaraltValtozo < 10] whileTrue: [
   Transcript showCR: deklaraltValtozo.
   deklaraltValtozo := deklaraltValtozo + 1.
]
```

Vegyük észre, hogy a feltételt nem kerek, hanem szögletes zárójelek közé tettük. Ennek az az oka, hogy ott egy blokkot kell megadni, nem egy logikai értéket. A Smalltalk virtuális gép minden ciklusléptés előtt kiértékeli a megadott blokkot, majd a kiértékelés eredményeként kapott logikai értéknek elküldi a `whileTrue:` üzenetet a beágyazott blokkal (ami kiírja a `deklaraltValtozo` nevű változó értékét, majd megnöveli azt eggyel).

A program kiírja a képernyőre a számokat 1-től 9-ig.

A másik, tömbbejáró ciklus azon az ötleten alapszik, hogy minden tömbnek lehet küldeni egy `do:` üzenetet, amely után egy blokkot kell írni. A blokknak egyetlen paramétert kell várnia (ott kapja meg az aktuális tömbelemet). Az üzenet hatására a megadott tömb összes elemével, mint paraméterrel végre lesz hajtva a `do:` után megadott blokk.

```
|egyTomb|
...
egyTomb := #('egy' '2' 3 4 5 '6' 7.0).
egyTomb do: [ :elem |
   Transcript ShowCR:elem
].
```

A fenti példaprogram az `egyTomb` tömb elemein végigmegy és kiírja azokat. Látható, hogy a tömbelemek tetszőleges típusúak lehetnek, egy tömbben lehetnek akár karakterláncok, numerikus értékek és bármilyen más értékek egyidejűleg. Ennek oka az, hogy az összes lehetséges érték objektum és a tömb az objektumokra hivatkozó referenciákat tárolja, ami minden adattípusnál egyforma (de a referenciával hivatkozott érték típusa már nem feltétlenül!).

3.11. Önhívó kód (rekurzió)

A vezérlési szerkezetek tárgyalásának teljessége érdekében megemlítjük az önhívó kódot, a rekurziót is, mint vezérlésszervezési eszközt (bár valójában ez az eljárás- és függvényhívások témaköréhez tartozik, részletesebb tárgyalását lásd az erről szóló 7. fejezetben). Az önhívó kód lehetősége és működése azon alapszik, hogy egy eljáráson (vagy függvényen) belül az eljárás neve látható, így lehetőség van arra, hogy egy eljárás önmagát hívja meg. Ezt az esetet nevezzük önhívó kódnak, rekurziónak, amikor egy eljárás vagy függvény önmagát hívja meg közvetlenül vagy közvetve (másik eljáráson keresztül).

A Pascal nyelv támogatja az önhívó kódot. A COBOL nyelvben számos más, úgynevezett második generációs nyelvhez – mint például a FORTRAN – hasonlóan, még nem volt lehetőség önhívó függvényhívásra.

Önhívó algoritmusok tervezésekor fontos a végesség, vagyis hogy ne alakulhasson ki olyan eset, amikor egy eljárás vagy függvény vég nélkül hívogatja önmagát. Ezt az önhívó algoritmusok tervezésekor két alapszabály betartásával biztosíthatjuk:

- Minden önhívó algoritmusnak tartalmaznia kell egy nem-önhívó alternatívát, amely lehet üres is. Ez általában a probléma *egyszerű esetének* megoldását tartalmazza.
- Az önhívásokat tartalmazó alternatíváknál gondoskodni kell arról, hogy a hívási lánc *valamikor biztosan elérje* a nem-önhívó alternatívát (az egyszerű esetet).

Tekintsük a klasszikus példát, a faktoriális függvény önhívó definícióját:

$$n! = \begin{cases} 1, & \text{ha } n = 0 \\ n \cdot (n-1)!, & \text{ha } n > 0. \end{cases}$$

Itt az egyszerű eset a 0 faktoriálisának a kiszámítása, ami definíció szerint 1. A másik alternatíva pozitív számok faktoriálisának kiszámítását írja le. Látható, hogy ott a feladatot (egy n szám faktoriálisának a kiszámítását) visszavezetjük egy eggyel kisebb szám $(n-1)$ faktoriálisának a kiszámítására, és ezt megszorozzuk n-nel. A faktoriális fenti definíciója alapján látható, hogy az algoritmus minden pozitív n-re befejeződik: mivel mindig eggyel kisebb szám faktoriálisának a kiszámítása a feladat, ezért előbb-utóbb elérünk oda, hogy 0 faktoriálist kell kiszámolni, ami nem-önhívó módon definiált: értéke 1.

Az önhívó algoritmusok alkalmazásának két jellegzetes esete a következő:

- Egy probléma megoldása elvégezhető ugyanazon algoritmusnak a probléma részproblémáira való alkalmazásával és az eredményt a kapott részeredményekből számíthatjuk ki.
- Egy önmagára hivatkozást is tartalmazó adatszerkezetet kell feldolgoznunk. Például egy fa adatszerkezetet. A fa adatszerkezetnél maradva az egyszerű eset a fa leveleinek a feldolgozása lehet, míg egy teljes fa feldolgozása felbontható például a gyökérelem feldolgozására és a részfák feldolgozására.

Megjegyezzük, hogy az önhívó algoritmusok hiánya a korai nyelvekben sem okozott gondot, mivel minden önhívó algoritmus egy verem adatszerkezet segítségével átalakítható nem-önhívóvá (iteratívvá), egy ciklus segítségével. A segédveremmel az önhívó program végrehajtási vermét utánozzuk (az új hívások paramétereit helyezzük rá, így az a még elvégzendő feladatokat tartalmazza).

3.12. Összefoglalás

A fejezetben áttekintettük az algoritmusleírás alapvető nem formális és formális eszközeit, az alacsonyszintű (assembly) nyelvektől egy példán át egészen a magasszintű programnyelvek vezérlési szerkezeteiig. Néhány ma is elterjedt programozási nyelvet alapul véve megvizsgáltuk az egyes nyelvek konkrét vezérlési szerkezeteit (részletesebben tárgyaltuk a COBOL, a

Pascal, a C nyelveket, illetve vezérlési szempontokból foglalkoztunk még a Modula-3, az Ada, a Java, valamint a C# nyelvekkel).

Látható, hogy a fejlődés óriási – a vezérlés irányítására egyre magasabb szintű eszközöket dolgoztak ki, melyek néhány jellemzőbb képviselőjét a fejezetben részletesen megismerhettük.

A vezérlési szerkezetekkel kapcsolatos vizsgálódásainkat általában több közös szempont alapján végeztük, mely szempontokat a vizsgált vezérlési szerkezet ismertetésekor részletesen megadtuk.

Megjegyezzük, hogy az itt bemutatottakon kívül sok ezer programozási nyelv van, melyeket más-más céllal, más-más fejlesztői csoportok fejlesztettek ki. Közülük sok a célterülettől függő programnyelv. Ezek a programozási nyelvek az általuk megcélzott terület problémakörére adnak megoldást, ennek megfelelően vezérlési szerkezeteik célirányos bővítéseket is tartalmazhatnak.

Példaként két alkalmazás-területet említünk meg: az adatbáziskezelést és a valósidejű alkalmazásokat. Az adatbáziskezelést támogató nyelvek közül például a SuperNOVA vezérlését tekintve illeszkedik az adatbázisok tranzakció alapú szemléletéhez: az alprogramok végrehajtása sikeres vagy sikertelen lehet. A sikeres végrehajtásnál valószínű, hogy az adatbázison elvégzett módosításokat tartalmazó tranzakció végleges (vagyis az adatok az adatbázisban is módosulnak). A nyelv tervezői abból indultak ki, hogy sikertelen végrehajtásnál viszont a tranzakció is visszavonódik: ennek megfelelően – a nem `STATIC` minősítésű globális változók – egy alprogramban bekövetkezett hiba esetén visszakapják az alprogram elején érvényes értékeiket és az alprogram futása sikertelenséget jelezve azonnal befejeződik. Ez jellemző más, úgynevezett negyedik generációs nyelvekre (4GL) is. A másik említett terület a valósidejű alkalmazások területe: egy valósidejű környezetben egy program végrehajtása nem feltétlenül helyes csak azért, mert helyes eredményeket ad vissza. Időkorlátok is vannak, amiket a programnak figyelembe kell vennie. *Egy valósidejű program végrehajtása akkor helyes, ha egy rögzített határidő előtt helyes eredményeket ad.* Ezekben a nyelvekben a fejezetben bemutatottaknál sokkal egyszerűbb (ennél szűkebb) vezérlési szerkezetek vannak, amelyek alapján a fordítóprogram az utasítások megfelelő ütemezésével helyes kódot állíthat elő, vagy a fordítás sikertelen lehet azért, mert a fordítóprogram szerint nincs olyan ütemezés, amely időben adna vissza helyes eredményt.

Feladatok

1. Mutassuk meg, hogy a fejezetben a legnagyobb közös osztó kiszámítására megadott két módszer egyenértékű!

2. A saját számítógépünk assembly nyelvét felhasználva valósítsuk meg a legnagyobb közös osztó kiszámítását végző programot. Hasonlítsuk össze a saját számítógépünk assembly nyelvének szerkezetét a könyvben bemutatott Motorola 68000-es assembly nyelvvel!

3. Egyes programnyelvek utasítás- és azonosítókészlete érzékeny a kis- és nagybetűk különbségére, míg más nyelveknél ez nincs így. Hasonlítsa össze a kis- és nagybetűket

megkülönböztető nyelveket az azokat nem megkülönböztető nyelvekkel! Az összehasonlításnál térjen ki a biztonságos programozás és a nyelvhez készíthető fordítóprogram bonyolultságára.

4. Készítsen Pascal programot, amely kiírja önmaga forráskódját!

5. Vizsgálja meg a fejezetben tárgyalt vezérlési szerkezetek redundanciáját. Mely vezérlési szerkezetek hagyhatók el úgy, hogy az adott programozási nyelv kifejezőereje ne csökkenjen?

6. Mi a véleménye az önmódosító kód alkalmazásáról?

7. Készítsen Pascal nyelvű programot, mely a képernyőre kirajzol egy sakktáblát. A sakktábla négyzet alakú, oldalhosszát - ami esetünkben nemcsak 8 lehet - a program a felhasználótól kérje be! Ügyeljünk arra, hogy csak pozitív oldalhosszúságú sakktáblát engedjünk kirajzolni.

8. Vizsgálja megy egy saját programjában az egyes vezérlési szerkezetek, utasítások arányát egymáshoz és a program összes utasításához viszonyítva. Az összehasonlításba vegye bele az értékadó utasításokat is!

9. Készítsen önhívó függvényt egy egészeket tartalmazó tömb elemeinek összes lehetséges permutációinak meghatározására.

10. Készítsen iteratív (nem-önhívó) függvényt egy egészeket tartalmazó tömb elemeinek összes lehetséges permutációinak meghatározására.

11. Mit ír ki az alábbi C# programrészlet?

```
int i = 2;
switch (i) {
  case 0:
    goto case 2;
  case 1:
    goto case 2;
  case 2:
    System.Console.WriteLine("i < 3");
    break;
  case 3:
    System.Console.WriteLine("i = 3");
    break;
  default:
    System.Console.WriteLine("i > 3");
    break;
}
```

4 Hatókör, élettartam

A különböző paradigmákat képviselő, különböző lehetőségeket nyújtó, különböző eszközöket adó nyelvekkel és környezetekben dolgozva egy program tervezésekor, készítésekor, karbantartásakor egyaránt fontos szempont a változók és típusok hatóköre és élettartama.

Egy objektum hatóköre, láthatósági köre (scope) az a része a programnak, ahonnan az objektum (az adott néven) elérhető.

Egy objektum élettartama (life, lifespan) a program futási folyamatának az illető objektum keletkezése és megszűnte közé eső része, azaz amikor az illető objektum él, felhasználható. Az elérhetőség és az élettartam sokszor összefügg, de nem mindig esik egybe.

Nagyobb projektekben, ahol több programozó dolgozik együtt, igen fontos az információ elrejtésének az elve: az, hogy az egyes részek optimálisan kapcsolódjanak egymáshoz. Az egyik részt író programozó csak annyit kell, hogy tudjon a másik által írt (vagy írandó) részről, amennyi szükséges, és ne többet: csak a kapcsolódási felületeket kelljen ismernie. Ne legyen felesleges és esetleges függés a részek között, mert az az emberi munkát megnehezíti, felesleges bonyolultságot okoz és ebből adódóan esetlegesen hibákat. Egy nagyobb, több programozó által több részben írandó program készítésekor milyen kellemetlen lenne, ha nem használhatnánk az *i* nevet egy ciklusváltozó számára, mert valaki gyorsabb volt és egy másik modul belsejében már elhasználta! Saját programunk is meglepően idegen tud lenni egy idő múltán, később mi magunk is „másik programozó" leszünk ebben az értelemben.

Az információ elrejtésének elvét az egyes nyelvek eltérő mértékben és súllyal képviselik, különböző finomsággal támogatják.

A programokban „objektumokat" használunk, ahol „objektum" alatt most a memóriában elhelyezkedő valamit értünk. Lehetséges, hogy ez saját névvel bír, esetleg egyszerre többel is, amelyet közvetlenül felhasználhatunk az objektum elérésére, az is lehetséges, hogy csak valamely nyelvi elem segítségével közvetve érjük el azt, vagy akár mind a kétféle módon. Csak közvetve, ilyen célra szolgáló nyelvi elemek segítségével elérhető objektumokra példák a dinamikusan (futási időben) lefoglalt objektumok, vagy a tömbök egyes elemei. A dinamikusan lefoglalt területeket általában mutatókon keresztül, a tömbelemeket a tömbből és az indexből álló összetett kifejezés segítségével érjük el. A nevek általában alfanumerikusak lehetnek: a pontos szabályok az egyes nyelvektől, környezetektől, vagy akár fordítási kapcsolóktól függnek.

Amikor egy objektumot használunk egy programban, fontos tudnunk, hogy mi az a része a programnak, amelyikből az (az adott néven) elérhető, azaz hogy mi az adott objektum hatóköre, láthatósági köre (scope), illetve, hogy mi az objektum élettartama (life, lifespan): mikor (és hogyan) keletkezik, meddig él, hogyan szűnik meg.

4.1. A memóriaterülek típusai

A láthatósági körök és a változók, objektumok élettartama szempontjából is fontos szem előtt tartanunk a memóriaterületek típusait.

4.1.1. A statikus memóriaterület

A *statikus* memóriaterületek kiosztása, méretezése fordítási időben megtörténik, csakúgy mint értékkel (alapértelmezettel vagy általunk megadottal) való ellátása is. Tulajdonképpen a program kódja is ilyen adatnak tekinthető, ha nem is kezeljük olyan módon, mint az adatterületeket. A környezet (a hardver, az operációs rendszer memóriakezelése) támogathatja ezen memóriaterületen belül a módosíthatatlan, azaz csak olvasható és a módosítható, írható-olvasható részek megkülönböztetését. Célszerű a program kódját módosíthatatlan részben tartani, hiszen ekkor a memóriába már betöltött program újabb példányát elindítva a keletkezett újabb folyamat használhatja az előző példány kódját!

A statikus memóriaterület előnye, hogy egyszerű a kezelése.

4.1.2. A dinamikus memóriaterület

A *dinamikus* területek szolgálnak tárul azon objektumok számára, amelyeket a program futása során létre akarunk hozni, több helyről elérni, majd programozottan megsemmisíteni és a hozzájuk rendelt memóriát felszabadítani. Sokszor fordításkor nem, csak futási időben lesz ismeretes, hogy milyen mennyiségű adatot kell kezelni, és általában nem jó a fordítási időben lefoglalt memória mennyisége révén a program által kezelhető adatok mennyiségére nézve önkényes korlátokat szabni. A nagyobb rugalmasságnak (mint az életben mindenhol, úgy itt is) ára van: többféle veszély is leselkedik a dinamikus memória felhasználóira. Előfordulhat, hogy

- csak lefoglaljuk a memóriát, de elmulasztjuk felszabadítani: ilyenkor beszélünk a memória elszivárgásáról (memory leakage);
- már felszabadított (és akár más célra már újra felhasznált) memóriára hivatkozunk még mint a felszabadítás előtti tartalomra;
- többször próbálunk meg felszabadítani egy adott memóriaterületet;
- mindent jól csinálunk és a memória mégis elfogy.

A dinamikus memória használatának hátránya, hogy a kezelése bonyolultabb lehet, mint a statikusé, mert a programozónak nagyobb szerepe (és mondhatnánk felelőssége) van a kezelésében és ráadásul bonyolultabb (elvontabb) nyelvi elemek használatát igényelheti.

Nem minden nyelv és környezet támogatja dinamikus memória használatát. Például a COBOL nyelvet nem így tervezték, és ha bizonyos változatokban és megvalósításokban támogatja is, ez a nyelv logikájától és mondhatnánk kultúrájától idegen.

A dinamikus memóriakezelést vagy közvetlenül nyelvi elemek (például a C++ és a Java **new** operátora, illetve a C++-ban a *delete*) vagy (szabványos) könyvtári függvények (például a C-ben a *malloc* és a *free*) támogatják.

4.1.3. Az automatikus memóriaterület

Ha egy nyelv és környezet nem támogatja a rekurziót, azaz azt, hogy egy eljárás futása során közvetlenül vagy közvetve önmagát meghívja, akkor egy eljárás egyszerre legfeljebb egy hívása lehet aktív, és így az eljáráshíváshoz szükséges adatokat lehetséges statikus, azaz a fordításkor előre lefoglalt területen tárolni. Ha cél a rekurzió támogatása, mint ahogy általában igen, akkor automatikus, azaz az eljárásba (vagy blokkba) való belépéskor lefoglalandó tárterület kell az eljáráshívások futási idejű támogatásához szükséges adatok (paraméterértékek, visszatérési cím, a regiszterek értéke a hívás pillanatában) valamint az eljárásokban deklarált adatok számára. Ezen adatok összességét *aktivációs rekordnak* hívjuk. Ezt általában egy erre a célra szolgáló verem (stack) támogatja: egy LIFO (last-in, first-out: elsőnek be, utolsónak ki) elvű, dinamikusan (azaz futás közben) változó összméretű adatszerkezet. Végrehajtási szálanként értelemszerűen külön veremre van szükség. Az implementáció akkor is használhat vermet az eljáráshívások megvalósítására, ha nem támogatja a rekurziót, hiszen verem használatával csak annyi memóriát kell az eljáráshívások számára lefoglalni, amennyi a tényleges aktív hívásokhoz kell, és általában korántsem aktív egyszerre az összes eljárás, sőt a program szerkezetéből adódóan nem is lehet az.

4.1.4. Egy egyszerű példa

Nézzük egy egyszerű (bár nem túl életszerű) példát, amelyik a Fibonacci-sorozat harmadik tagját határozza meg – igen kevéssé hatékony, de a rekurziót és különböző tárolási osztályú változókat szemléltető módon. A Fibonacci-sorozatot 1202-ben találta ki Leonardo Fibonacci. Tegyük fel, hogy egy nyúlpárból indulunk ki, és egy nyúlpár havonta egy párat ellik egyhónapos korától kezdve, és a nyulak halhatatlanok (mondjuk a vizsgált időintervallumban). A feltevés szerint tehát egy adott hónapban a nyulak az előző hónapban élt nyulakból és a nem újszülötteknek, azaz a kettővel előző generációbelieknek az utódaiból adódnak. Ha $F(n)$-nel jelöljük a nyúlpárok számát n hónap múlva, akkor tehát $F(n) = F(n-1) + F(n-2)$. Érdekes, a matematikában, az algoritmusok elemzésekor sokszor előforduló sorozat adódik így [Knu 81].

A sorozat saját magával definiálható: egy értéke az előző értékekből számítható. Kézenfekvő leképezése ennek, hogy az őt kiszámító *fib* függvény saját magát hívja. A 4.1. ábrán látható példa C++ nyelvű, de procedurális, mondhatnánk C stílusú. Azért választottuk mégis a C++ nyelvet a C-vel szemben, mert itt egyszerűbb a kimenet programozása (*std::cout*).

Egy adott környezetben ilyen eredményt ad a futtatás:

```
1 fib ++ 3 0xbffff708
2 fib ++ 2 0xbffff6d8
3 fib ++ 1 0xbffff6a8
3 fib -- 1=1
4 fib ++ 0 0xbffff6a8
4 fib -- 0=1
4 fib -- 2=2
5 fib ++ 1 0xbffff6d8
5 fib -- 1=1
5 fib -- 3=3
a végeredmény 3 5 lépésben.
```

```
#include <iostream>
int ct;
int fib (int what)
{
    int ret = 1;
    ct++;
    std::cout << ct << " fib ++ " << what << " " << (void *)&ret << std::endl;
    if (what > 1) {
        ret = fib (what - 1) + fib (what - 2);
    }
    std::cout << ct << " fib -- " << what << "=" << ret << std::endl;
    return ret;
}
int main ()
{
    int x = fib (3);
    std::cout << "a végeredmény " << x << " " << ct << " lépésben." << std::endl;
    return 0;
}
```

4.1. ábra: Fibonacci-sorozat C++-ban

Amikor *fib(3)*-at ki akarjuk számítani, szükségünk van *fib(2)* és *fib(1)* értékére, tehát több aktív, futó *fib* hívás is van, ennek megfelelően több aktív, élő hívási paraméterértékkel, lokális *ret* változóval. Azért lett kiírva a *ret* címe, hogy ezt ellenőrizzük. A *fib* függvénybe való belépések számát a *ct* nevű globális változóban tároljuk.

4.2. Hatókör

Egy név hatóköre az a része a programnak, ahonnan a jelölt programelem (változó, eljárás, típus stb.) az adott néven elérhető.

Különböző nyelvi elemek (minősítés) segítségével bizonyos esetekben be lehet látni egy másik hatókörbe is. Hatókör lehet

- az egész program
- a fordítási egység[1]
- egy alprogram
- kódblokk
- egy típus (osztály) vagy egy névtér.

[1] Fordítási egységnek nevezzük az egy forrásba tett és emiatt egyszerre fordított függvények és változók körét.

Jól láthatóan a különböző hatókörök át is fedik egymást, a programnak egy adott pontja több hatókörbe is beleesik. A hatókörök jellemzően bővülő köröket képeznek: egy függvény egy pontja beletartozik az összes őt tartalmazó blokk hatókörébe, a függvényébe, a globálisba; objektumorientált környezetben egy típus (egy osztály) hatóköre része lehet öröklődés révén más típusokénak.

```
extern int x;                    /* Van valahol egy globális x. */
#ifdef __cplusplus
namespace N {
    int x;
    void f();
}
#endif
int foo(int y)
{
    if (y == 0) return x;        /* A globális x. */
    {
        int x=2;
#ifdef __cplusplus
        if (y > 0) return ::x;   /* C++: a globális x */
        N::x++;                  /* C++: az N névtérbeli x. */
#endif
        if (y < 0) return x;     /* 2 */
        {
            int x = 3;
            return x;            /* 3 */
        }
    }
}
#ifdef __cplusplus
void N::f()
{
    x++;                         /* N::x-et jelenti. */
}
#endif
```

Egy név feloldása, azaz annak a megkeresése, hogy melyik programelemet is azonosítja a program egy adott pontján a név, a hatókörök között a legszűkebbtől a bővebbek felé haladó sorrendben történik. Emiatt egy közelebbi (belső) hatókörbeli deklaráció elfedi a bővebb (külső) hatókörbelit.

Lehetséges mindazonáltal a nevek túlterhelése (overloading), azaz hogy ugyanaz a név egyszerre többféle dolgot jelentsen. Mégpedig nyilván akkor, ha a néven kívül valami más jellemzője is van az illető nyelvi elemnek, ami alapján a szóba jöhető különböző jelöltek

megkülönböztethetőek. A típusokat fordítási időben ellenőrző nyelvek esetében lehetőség van a függvényeknél a név mellett az argumentumok számát és típusát, azaz a szignatúrát is a függvény jellemzőjének tekinteni, és így azonos nevű, de különböző szignatúrájú függvényeket kezelni:

```
void print(int i);
void print(char *s);
void print(char c);
```

Az ANSI C végez típusellenőrzést, de a fenti értelmű névtúlterhelést nem támogatja. A túlterhelt függvénynév feloldási szabályaiban az Ada nyelv figyelembe veszi a függvény visszatérési értékének típusát, a C++ és a Java azonban nem.

A C++ nyelv ismeri a hatóköroperátort: ⟨*névtér-név*⟩*::*⟨*objektum*⟩ vagy ⟨*osztálynév*⟩*::*⟨*függvénynév*⟩; láttuk az *std::cout* példát, amely az std-nek, azaz a standard könyvtári objektumok és függvények névterének *cout* objektumát jelöli, ami a szabványos kimenetet (standard output) kezeli. A *::x* pedig a globális *x* változót jelölte.

4.2.1. Globális hatókör

A globális változók a program minden pontjáról láthatók és statikus tárolási osztályúak. A különböző részek (modulok, függvények) között egyszerű kommunikációt tesznek lehetővé, de nem mindig könnyen átlátható függést okozhatnak közöttük. Ha egy eljárás (függvény) globálisokat módosít, akkor azt mondjuk, hogy ez a mellékhatása. Egy függvényt felfoghatunk úgy, mint egy tetszőlegesen bonyolult utasítássorozat „rövidítését", önálló egységként való kezelését. A függvények mellékhatása ennek az elvnek némiképp ellentmond. A C és C++ nyelvekben az összes függvényen és osztályon kívül definiált, a **static** módosítóval el nem látott változók ilyenek. A FORTRAN nyelvben lehetőség van a *COMMON* direktíva segítségével globális memóriaterületeket kijelölni, és ezeket akár modulonként különböző módon változókra bontani.

4.2.2. Fordítási egység mint hatókör

Közepes és annál nagyobb méretű programok esetén fontos, hogy ne kelljen mindent lefordítanunk, ha valahol valamit megváltoztatunk, azaz hogy önállóan fordítható részekre bonthassuk a programot[2]. Ezt a különböző nyelvek különböző egységek bevezetésével támogatják: partíciók, modulok, fordítási egységek, könyvtári egységek és alegységek. Nagy projektek esetén fontos tervezési szempont, hogy az ilyenekre való bontásból származó előnyt minél jobban ki tudjuk használni [Lak 96].

A fordítási egységet – azon nyelvek esetén, amelyek ezt egyáltalán támogatják – az egy forrásba tett és így egyszerre fordított függvények és változók alkotják[3]. A nyelvtől függ,

[2] Az Open Office program 6.0-ás verziója 800 Mhz-es processzorral és 256 MByte memóriával ellátott gépen a forrásokból kiindulva körülbelül 12 óra alatt készül el...

[3] A Java például nem támogatja ezt a fogalmat: egy publikus osztály (vagy interfész) forrása csakis a saját nevének megfelelő nevű fájlban lehet, és az egyes osztályok bájtkódja saját `.class` fájlokba kerül, még a névtelen belső osztályoké is.

hogy kívülről hogyan férhetünk hozzá a fordítási egységbe foglalt változókhoz és függvényekhez.

Amit viszont csak a fordítási egységen belül akarunk elérhetővé tenni, de kívül nem, azt a fordítási egység hatókörébe tehetjük.

Egyes nyelvekben (Pascal, Ada, C és C++) szintaktikailag is megkülönböztetjük a deklarációs és a megvalósító részeket, másokban (például Javában) nincsen ilyen megkülönböztetés. Míg a Pascal nyelvcsalád tagjaiban (Pascal, Ada) valamint a Javában egy egység fordításakor a fordítóprogram a felhasznált egységek lefordított tárgy- vagy bájtkódjából veszi a felhasználáshoz szükséges információt, és emiatt a fordítási egységek közötti szemantikus függőségi viszonyok az egységek lehetséges fordítási sorrendjét is meghatározzák, addig a C és C++ nyelvekben a felhasználandó információkat az erre a célra szolgáló forrásfájlokból (a fejlécekből, a header állományokból) veszi a fordító.

A C-ben és nyomában a C++-ban egy forrásban a külső, azaz az összes függvényen és osztályon kívül definiált **static** módosítóval ellátott változók tartoznak a fordítási egység hatókörébe.

A tárolási osztály statikus lesz, a hatókör az adott fordítási egységre korlátozódik. Alkalmazhatjuk a **static** módosítót függvényre is: ekkor a függvény csak az adott fordítási egységen belül lesz elérhető. Ez a C-s örökség a modulokra való bontás egyfajta eszköze. Annyiban nem egészen szerencsés dolog, hogy összekapcsolja az egyes részek közötti láthatósági viszonyokat, azaz tervezési szintű dolgot azzal, hogy miket rakunk egy forrásállományba, egy fordítási egységbe, ami viszont már inkább implementációs kérdés.

Az egy fordítási egységbe tartozó függvények osztoznak azokon a változókon, amelyek meghívásaik között is képesek az érték megőrzésére, a fordítási egység függvényeinek együttese, mint modul számára állapot tárolására. A változók (és a statikus függvények) neve kívülről nem látható, ezért újra felhasználható.

A C++ nyelv bevezeti a névtelen névtér fogalmát, amely a fordítási egységre lokális láthatósági kör definiálására szolgál. A névtelen névtér használata ekvivalens a külső statikus deklarációkéval, és azok helyett ajánlott.

4.2.3. Függvény és kódblokk mint hatókör

Mind statikus, mint automatikus változó hatóköre lehet egy függvény (alprogram, metódus) vagy blokkstruktúrájú nyelvnél egy kódblokk. (Végül is egy függvény törzse felfogható kódblokknak). Ezeket a változókat az illető függvényre vagy kódblokkra nézve lokálisnak nevezzük. Az egy blokkban deklarált változók láthatóak a blokkon belül levő további blokkokban is. Egy blokkból nézve láthatóak természetesen a saját és a mindenhonnan látható (globális) változók, ezen kívül pedig az adott kódblokkot tartalmazó blokkokban, függvényben, fordítási egységben deklarált változók is, amelyeket a kódblokk szempontjából tágabb értelemben szintén hívhatunk globálisaknak.

A függvényen belül deklarált statikus változók szolgálnak olyan tárterületül, amelyik értéke a függvényhívások között megmarad, és amely név érvényessége csak az illető függvényre illetve blokkra terjed ki. Hasonló szerepet tölt be az ALGOL 60 és SIMULA 67 nyelv **own** kulcsszóval jelzett lokális változója.

Ha másként nem intézkedünk, azaz nem deklaráljuk **static**-ként, akkor a függvényen vagy kódblokkon belül deklarált változó automatikus lesz, azaz a belépéskor automatikusan lefoglalt és kilépéskor automatikusan felszabaduló, bármikor más célra felhasználható területre kerül. A függvény hatókörébe tartoznak még a paraméterei is.

4.2.4. Egy típus mint hatókör

A típusok, az osztályok egyben hatókörök is. Ebben a körben a statikus (**static**) szó jelentése újabb árnyalattal bővül: az osztályhoz és nem az egyes példányokhoz tartozó adattagokat és függvényeket jelenti, ezek minősítője. A statikus adattagok a tárolási osztály szempontjából is statikusak. A példányhoz kötött (azaz nem statikus) függvények mind a példányhoz kötött, mind a statikus adattagokat és függvényeket elérik egyszerűen azok nevét használva, a statikusak értelemszerűen csak a statikusakat.

Mind a C++, mind a Java nyelvben minősíthetjük mind az adattagokat, mind a metódusokat, hogy milyen a láthatóságuk: privát, védett és nyilvános (publikus).

- A privát (**private**) tagok és metódusok csak az adott osztályban láthatók, elérhetők.
- a védett (**protected**) tagok és metódusok csak az adott típustól öröklő típusokban;
- a nyilvános (**public**) tagok és metódusok pedig mindenhonnan.

A Javában van még az alapértelmezett félnyilvános hozzáférés, amelyhez nem tartozik kulcsszó: ez az adott típus csomagjából (*package*) való elérhetőséget jelenti.

A C++-ban fenti képet árnyalja a **friend** deklaráció lehetősége: egy típus privát tagjaihoz, függvényeihez a típus „barátai" is hozzáférhetnek. A Javában nincs barátság: itt a privát tagokhoz való hozzáférés a belső osztályok kiváltsága.

Ha egy típus örököl egy másiktól, akkor ez a hatókörre vonatkozólag is következménnyel jár: egy származtatott osztály tagfüggvényei ugyanúgy elérik a bázisosztály tagjait és függvényeit, mintha azok a származtatott osztályban lettek volna bevezetve, kivéve a privát tagokat és függvényeket.

4.3. Élettartam

A programozási nyelvekben a változók (tárterületek) egyik alapvető tulajdonsága az *élettartam*. A változó élettartama a program végrehajtási idejének az a része, ameddig a tárterületet arra használhatjuk, hogy az adott objektum értékét hordozza. Az élettartam egyaránt fontos fogalom mind a hagyományos (procedurális, strukturált) mind az objektumorientált nyelvekben.

Ha pontosabban megfigyeljük egy új objektum létrejöttének a folyamatát, akkor azt látjuk, hogy először valahol memóriát kell lefoglalni az objektum számára, majd pedig az így kapott memóriát inicializálni kell, a használatra előkészíteni; be kell állítani az invariánsokat, azaz az objektumot konzisztens belső állapotba kell hozni.

Az objektum-alapú nyelvekben az utóbbi célra külön függvény szolgál: az illető típushoz tartozó úgynevezett konstruktor, aminek a neve többnyire azonos a típuséval. A túlterhelés lehetősége miatt több konstruktor is lehetséges különböző paraméterlistával. Természetesen

inicializáló-függvényt használhatunk procedurális nyelvekben is: az API-ban (azaz a programozási felületre nézve) előírhatjuk például, hogy egy struktúrát használat előtt inicializálni kell az inicializáló-függvénnyel, de ha erre nincsen nyelvi garancia, akkor a fordítóprogram nem tudja ezt támogatni, és így lehetséges hibás használat. Az objektum-alapú nyelvek azonban garantálják, hogy egy objektum csak ellenőrzötten (azaz lényegében csak a megfelelő konstruktor közreműködésével) jöhet létre.

A memória lefoglalása a futási idejű környezet dolga, a megfelelő konstruktor megkeresése és meghívása a fordítóprogramé, a konstruktor (és a destruktor) helyes megírása pedig a programozóé.

A konstruktor a fentiek miatt tárolásiosztály-semleges: nem tud róla, és nem törődik vele, hogy hol van az előkészítendő memóriaterület. A konstruktort arra használjuk, hogy egy adott (pontosan ismert) típusú objektumot hozzunk létre. Objektumorientált (azaz öröklődést támogató) környezetben előfordul, hogy a típusnak valamely őse az érdekes, amely valamely interfészt nyújt, és nem érdekes a kívánt objektum pontos típusa. Valamely objektumtól (a factorytól, azaz az „objektumgyártól") el tudunk kérni egy adott típusú objektumot, de könnyen lehet, hogy ténylegesen egy abból származtatott típusút kapunk. Az objektumgyár, a factory megfelelő metódusa azonban nem járhat el tárolásiosztály-semleges módon: valahonnan vennie kell, valahogyan elő kell állítania az objektumot, amit (vagy amire utató referenciát, mutatót stb.) visszaad.

Tipikus példa egy ilyen helyzetre a Java JDBC API-ban a *java.sql* csomagban a *DriverManager* osztály

public static *Connection getConnection(String url)* **throws** *SQLException*

metódusa, amely egy a *Connection* interfészt implementáló objektumot ad vissza, ha a dinamikusan bejegyzett JDBC-meghajtó osztályok közül talált olyat, amely hajlandó kezelni a megadott kapcsolódási String-et és sikerült is létrehoznia az adatbázis-kapcsolatot ábrázoló objektumot. Ha nem statikus metódust használunk, akkor lehetőségünk van többalakú (polimorf) metódus használatára is, ami összetettebb tervezési minták alkalmazását teszi lehetővé.

Éppen amiatt, hogy a konstruktorok összefüggnek a memóriafoglalással, külön kategóriát képeznek az illető típus metódusai között. Ezért nincsen a C++-ban, ahol pedig általában lehetséges egy osztály metódusára mutató pointer, konstruktorra hivatkozó pointer.

Egy objektum megsemmisítésekor először a rendrakási tevékenység következik, amire szükség van, ha az objektum például valamilyen külső erőforrást – fájlt, zárat, dinamikus memóriát – használt, akkor ezt fel kell szabadítani. Ezt követi az objektum által használt memória felszabadítása. Az utóbbi feladat értelemszerűen a futási idejű környezet dolga, de az előbbi az illető típusra jellemző, erre a célra szolgáló metódusé, az úgy nevezett destruktoré. A C++ nyelvben ennek neve a típus neve a komplemensképzést jelző tilde (~) mögött, jelezve, hogy ez a művelet a konstruktor kiegészítője.

A Java nyelvet eleve szemétgyűjtő algoritmus jelenlétét feltételezve tervezték. Itt nem destruktor-mechanizmus, hanem finalize-mechanizmus van. A Java virtuális gép magukat az objektumokat (tehát nem primitív típusú változót, mint **byte**, **char** vagy **int**, és nem objektumra hivatkozó referenciát) egy automatikus szemétgyűjtő mechanizmus (*garbage collector*) által felügyelt véletlen elérésű tárterületen (*garbage collected heap*) tárolja, még akkor

is, ha maga a referenciaváltozó statikus, azaz nem objektumpéldányhoz, hanem a típushoz tartozik. A használaton kívüli memóriát a *szemétgyűjtő mechanizmus* deríti fel és szabadítja fel. A szemétgyűjtő mechanizmus a virtuális gép egy önálló szálában (kis prioritással, hogy ne zavarja a többi szálat, köztük például a grafikus felhasználói felületet kezelőt) futva felderíti az *elérhetetlen* objektumokat, lefuttatja rájuk a finalize() metódust, majd felszabadítja az általuk felhasznált memóriát. A *finalize()* metódus arra a célra szolgál, hogy lehetőséget adjon bizonyos szükséges erőforrás-felszabadításokra, mielőtt az objektum végképp elvész és az általa használt memória felszabadul. A Java nyelv nem definiálja, hogy a virtuális gép mikor is fogja észrevenni, hogy egy objektum már nem érhető el, és hogy ha van *finalize()* metódus, akkor az mikor fog meghívódni, ellentétben a C++ destruktoraival, amelyek azonnal végrehajtódnak, amint az objektum élettartama véget ér.

A *statikus* élettartamú változók általában már a program kezdetén létrejönnek és a program végéig élnek. Az egy modulban (függvényben, fordítási egységben) levő objektumok létrejöttének sorrendje természetesen adódik a deklarációk sorrendjéből, de például a C++-ban a különböző fordítási egységekben levő változók kezdőértékkel való ellátásának sorrendje nem kötött, hanem implementációfüggő.

A Java nyelvben a primitív típusú adatok vagy objektumreferenciák lehetnek statikusak, de maga az objektum csak a dinamikus tárterületen jöhet létre. Ezért a Javában az alábbi osztály *statikusObjektum* nevű változója az osztály betöltésekor lefutó statikus osztályinicializáló blokkban kap értéket:

```
public class OsztályStatikusObjektummal {
    static int statikusObjektum[ ] = { 123, 456 };
}
```

Az *automatikus* változók a veremben kapnak helyet, mégpedig a függvénybe való belépéskor az eljáráshívások futási idejű támogatásához szükséges adatokkal együtt automatikusan (innen a név), és a kilépéskor a memóriaterület automatikusan felszabadul. Általában veszélyes, ha egy memóriaterületre mutató pointernek mint változónak, nagyobb az élettartama, mint a mutatott területnek. Az alábbi példában a hívott függvény olyan területre való mutatót ad vissza, amely a függvény visszatérése pillanatában felszabadul, és csak arra vár, hogy felülíródjék:

```
/**** HIBÁS! ****/
char *gettime()
{
    char wtime[24];
    /* Kitöltjük valahogyan. */
    return wtime;
}
```

A C++ nyelv számontartja az automatikus változókat, és a hatókörből való kilépés előtt (ha tehát a vezérlés bármi okból – csukó zárójelre való „rácsorgás", visszatérés (**return**), kivétel keletkezése – elhagyni készül az illető függvényt vagy blokkot) meghívja a destruktorokat – mégpedig a keletkezéssel ellenkező sorrendben.

Vannak nyelvek, ahol a programozó maga is vezérelheti a tárterületek lefoglalását. A *dinamikus* területek élettartama vagy a tárterület felszabadításáig tart, amelyet a program struktúrájától függetlenül a programozó jelöl ki, vagy akkor ér véget, amikor a területre

történő hivatkozásaink elfogynak: a terület többé nem elérhető. Ez utóbbi lehetőség támogatásához külön erőfeszítés kell a futtató környezet részéről: szemétgyűjtés. Bizonyos nyelveket és környezeteket eleve úgy terveztek, hogy szemétgyűjtő algoritmus lesz jelen (SIMULA 67, Ada, Java, C#), más nyelvekben nincsen szemétgyűjtő algoritmus vagy pedig a léte opcionális: nem része a nyelv specifikációjának. Nemcsak hogy futási időbeli erőfeszítést igényel a szemétgyűjtő algoritmus támogatása, de ha a nyelv lehetőséget ad a memória szabadabb kezelésére, akkor elvi problémák is felmerülhetnek. Ha például van mutatónk, pointerünk, amit manipulálhatunk, akkor egy dinamikusan allokált memóriaterületre mutató pointer értékét elrejthetjük. Az alábbi kódrészlet lefutása után:

```
char *pt = new char();
char mutat0[sizeof(pt)];
char mutat1[sizeof(pt)];
*(char **)mutat0 = pt;   // Bitenként átmásoljuk pt-t.
*(char **)mutat1 = pt;   // Bitenként átmásoljuk pt-t.
mutat0[0] = 0xab;
mutat1[1] = 0xcd;
pt = 0;
```

az első sorban újonnan allokált területnek eredetileg a *pt* mutatóban tárolt címe már nincsen sehol sem jelen programunkban, ezért a szemétgyűjtő úgy gondolhatná, hogy a *pt* által megcímzett memóriaterület már elérhetetlen „szemét" - de világos, hogy a *mutat0* és *mutat1* változók értéke alapján újból össze tudjuk rakni az eredeti mutatóértéket.

4.4. Példák

Statikus puffer használata • Tegyük fel, hogy egy operációs rendszer a fájlrendszerekben a fájlokhoz tartozó adminisztratív információk közül az időpont jellegűeket (például az utolsó elérés és az utolsó módosítás) valamely gép-, de nem felhasználóbarát mennyiség formájában tárolja, mint például egy adott időpont óta eltelt másodpercek száma valamely definiált egész (aritmetikai) típusú mezőben. Legyen ezen típus neve *time_t*[4]. Nyilván szükségünk lesz egy olyan függvényre, ami egy adott *time_t* értéket emberi fogyasztásra alkalmas szövegre konvertál. Hívjuk ezt a függvényt mondjuk *ctime*-nak. Felmerül a függvény használatakor, hogy ki allokálja a szöveg helyét. Mivel a hívó a priori módon nem tudhatja, hogy mennyi hely kell neki, a legegyszerűbb megoldás, hogy a függvény gondoskodik a helyről, mégpedig statikus területről, amit nem kell felszabadítani. A C szabványos könyvtárbeli ctime függvény pontosan ezt teszi, és a statikus pufferére mutató pointert ad visszatérési értékként. Deklarációja (a *time.h* fejlécből):

```
char * ctime(const time_t *tp);
```

[4] Példa erre a UNIX, amelyiknél a kiindulási időpont az úgynevezett Epoch, azaz 1970. január elseje. Nyilván emlékeznek az olvasók a 2000-es évvel kapcsolatos „Y2K bug"-ra és az azzal kapcsolatos hűhóra. Ismeretes és szerencsés módon az ez okból megjósolt világvége nem következett be! A jelenleg széles körben használatos 32 bites architektúrákhoz illeszkedő 4 bájtos előjeles egész maximális értéke a fent vázolt UNIX-os dátumábrázolási sémában 2038. január 29-ára eső maximális időpontot tud ábrázolni. Remélhetőleg sok olvasó meg fogja érni az "EPOCH 0x7fffffff-bug"-ot...

Ez a legegyszerűbb megoldás, de gondok vannak vele. Ha például egy hosszú ideig futó kódrész előtt és után is ki akarjuk iratni az időt, akkor az alábbi „megoldás" nem jó erre a célra, ahol tudnivaló, hogy *time(NULL)* a pillanatnyi időpontot jelző *time_t* értéket adja vissza:

```
char *before;   char *after;   time_t bt, at;
bt = time(NULL);
before = ctime(&bt);      /* 1 */
long_running_function();
at = time(NULL);
after = ctime(&at);       /* 2 */
printf("%s %s\n",before,after);
```

hiszen a második *ctime* hívás elhasználja a puffert, amelyben így az első híváskor belehelyezett érték elvész: felülíródik. Ezen úgy segíthetünk, ha a puffer első híváskor kitöltött értékét átmásoljuk valahova.

Ha viszont programunk többszálú, akkor nyilvánvalóan ezen felül is problematikus a statikus puffer használata, mert lehetséges, hogy több szál párhuzamosan hívja a függvényt!

A statikus memóriaterület használatát tehát jobb lenne elkerülni. Egy megoldás lenne erre, ha a hívó biztosítja a szükséges puffert, amivel átveszi az allokáció és a felszabadítás felelősségét, de ehhez persze a hívónak előre tudnia kell a pufferméretet:

```
void ctime1(char *buffer, const time_t *tp);
```

vagy ha ellenőrzésképpen közöljük a pufferméretet is:

```
void ctime2(char *buffer, int bufflen, const time_t *tp);
```

A legkényelmesebb nyilván valamilyen objektumalapú megoldás lenne, természetesen feltéve, hogy ezt támogatja az adott környezet.

Erőforrások kezelése objektumok által • A C++ nyelv konstruktor-destruktor mechanizmusa jól támogatja az erőforrások objektum-alapú kezelését: erőforrás létrehozásának és felszabadításának ez a idiomatikus (a nyelvhez legjobban illeszkedő) módja.

Tekintsük az alábbi C++-példát:

```
void bar();
void foo()
{
    char *pt = new char[1024];
    bar();
    delete[] pt;
}
```

Látszólag minden rendben van benne: a függvény elején dinamikusan foglaltunk le egy karaktertömböt, amit a visszatérés előtt gondosan fel is szabadítunk. (A **delete** operátor mellett álló szögletes zárójelpár azt jelzi, hogy nem egyetlen objektumot, hanem tömböt szabadítunk fel.) Mi történik azonban, ha a közben meghívott függvény futása közben kivétel lép fel? A vezérlés anélkül lép ki a *foo* függvényből, hogy a dinamikusan lefoglalt karaktertömb felszámolása megtörtént volna! Mivel *pt* az egyetlen mutatónk, ami erre a memóriaterületre

mutat, többé nem fogjuk tudni azt felszabadítani. A kivétel váratlan fellépése megzavarja programunk működését: ezért nem kivételbiztos (expection-safe).

Nekiállhatunk a lyukat befoltozni a kivétel elkapásával:

```
void bar();
void foo()
{
    char *pt = new char[1024];
    try {
        bar();
        delete[] pt;
    } catch (...) {
        delete[] pt;
        throw;
    }
}
```

A hasonló példa Javában elegánsabb a try-finally szerkezettel:

```
void foo() {
    try {
        get_resource();          // Az erőforrás allokálása
        bar();                   // kivételt dobhat!
    } finally {
        release_resource();      // Az erőforrás felszabadítása.
    }
}
```

Még jobb megoldás azonban az erőforrás (a C++-példabeli esetben a karaktertömb) önálló objektumban való kezelése, kihasználva az automatikus objektumoknak a hatókörből való távozáskori, destruktor általi megsemmisítését:

```
class mypuffer {
    char *pt;
public:
    mypuffer(int size) { pt = new char[size]; }
    ~mypuffer() { delete [] pt; }
};
void bar();
void foo()
{
   mypuffer a(1024);
   bar();
}
```

Szerencsére nem kell feltalálnunk a spanyolviaszt! A C++-ban a standard könyvtár *auto_ptr* sablonja (template-je) szolgál az automatikus mutatók kivételbiztos kezelésére. Az *auto_ptr* objektum egy mutatót tartalmaz, és maga is a mutatókhoz hasonlóan használható. Megsemmisülésekor a mutató által megcímzett objektumot is megsemmisíti.

Feladatok

1. A fejezetben említett nyelvek fordított (kompilált) nyelvek voltak. Vizsgáljunk meg interpretált (script) nyelveket, például az AWK, a Perl, a JavaScript, vagy a UNIX-shellek (sh, ksh, bash, csh, zsh és így tovább), hogy azokra mennyire vonatkoznak a fejezetben leírtak, így például vannak-e egyáltalán nem globális változók.

2. A 113. oldalon található példához (a mutató értékének elrejtése) adjuk meg, hogyan lehet a mutató értékét visszanyerni. Írjuk át a példát úgy, hogy nem használunk típuskényszerítést.

3. A 113. oldal *ctime* függvényes példájához (4.3.) írjuk meg a *ctime1*, *ctime2* függvényeket, illetve adjunk helyettük objektumalapú megoldást.

4. Írjuk át a 114. oldalon található példát (erőforrás lefoglalása, esetleg kivételt dobó művelet, erőforrás felszabadítása) úgy, hogy nem egy, hanem három erőforrás-objektum van, mégpedig mind C++-ban, mind Javában.

5. Keressük meg a statikus változókat inicializáló blokkot a 112. oldalon található példában (*OsztályStatikusObjektummal*)! Fordítsuk le az osztályt, és fejtsük vissza a kódját például az alábbi parancsokkal:

```
javac  OsztályStatikusObjektummal.java
javap -private -c OsztályStatikusObjektummal
```

6. Próbáljuk ki különböző számú paraméterrel futtatva az alábbi C nyelvű programot, és magyarázzuk meg a működését:

```
#include <stdio.h>
int foo (int i)
{
    if (i <= 0) {
        /* Szándékosan nem adtunk kezdőértéket. */
        int j;
        return j;
    }
    return foo(i - 1);
}

int main (int argc, char *argv[ ])
{
    printf ("%3d %8x\n", argc, foo (argc));
    return 0;
}
```

5 Típusok

A modern programozási módszertanok központi fogalma a típus. Ahhoz, hogy programjainkban bitek sorozatánál bonyolultabb fogalmakkal dolgozhassunk, a programozási nyelvek beépített típusok sokaságával járulnak hozzá. Ebben a részben áttekintjük ezeket a beépített típusokat, a típusok osztályozását, illetve megvizsgáljuk a kifejezésalkotás és -kiértékelés szabályait.

Minden feladat, amit számítógép segítségével megoldani kívánunk, kapcsolódik a minket körülvevő „valódi" világhoz. Ugyanakkor a programok, amelyeket e feladatok megoldására használunk, a számítógépek belső világában működnek. Ezért a feladatok megoldásának egyik legfontosabb eleme, hogy a való világnak a feladat szempontjából jelentőséggel bíró objektumait a számítógép belső világában modellezzük, hogy aztán a modellen megoldva a feladatot, a modellünkben kapott eredményt visszavetíthessük a való világra, értelmezhessük a kapott megoldást.

Amikor a való világ objektumait modellezzük, azoknak csupán a feladat szempontjából fontos jellemzőit vesszük be a modellbe, ezáltal a feladat szempontjából hasonlóan viselkedő csoportokat képezünk belőlük. Ennek a hasonlóságnak a kifejező eszköze a programozási nyelvekben a *típus*.

A programozási nyelvek beépítetten is tartalmaznak típusokat, amelyek a való világ bizonyos elvont fogalmait, mint a számok, betűk, szövegek stb. modellezik, mivel ezek igen gyakran megjelennek a különböző programozási feladatokban. A legtöbb nyelv ezen felül lehetőséget biztosít további típusok létrehozására is.

Ebben a fejezetben a beépített típusokkal fogunk foglalkozni.

5.1. Mi a típus?

Mielőtt nekilátnánk a típusok osztályozásának és elemzésének, nézzük meg, mit is értünk típus alatt. Amint az a bevezetőből kiderül, a típus nagyon összetett fogalom. Az ott megadott leírás nem alkalmas a behatóbb vizsgálatra, ezért több különböző definíciót fogunk felsorakoztatni, amelyek együttesen talán segítenek megérteni a lényegét.

5.1.1. Típus a fordítóprogram szemszögéből

Típus: absztrakció, amely összefoglalja a programban előforduló bizonyos objektumok bizonyos közös tulajdonságait. Ezek a tulajdonságok a kódolás (*encoding*), a méret (*size*), a szerkezet (*structure*), valamint magasabb szinten a jelentés (*semantics*). A típus jelentését a típusértékhalmaz és a típusműveletek definiálásával adhatjuk meg.

Ez a definíció, ahogy azt a fejezet címében is ígértük, a típust a programozási nyelvek szempontjából vizsgálja. Egy programozási nyelv számára a típus csakugyan ezt a négy dolgot jelenti. A nyelvnek (fordítóprogramnak) tudnia kell, hogy a típusértékeket hogyan kell a memóriában ábrázolnia (kódolás), ezzel kapcsolatban tudni kell, hogy mennyi helyet foglal el (méret). Amennyiben összetett típusról van szó, tudni kell, hogy milyen a belső szerkezete, hogy hivatkozni lehessen az összetevőire. Végül pedig ismerni kell a viselkedését, jelentését (szemantika), legalább valamilyen alacsony szinten – például az egész számok típusán általában már a nyelv biztosítja az alapműveleteket.

Természetesen ez a négy dolog megjelenik a programozó oldalán is, ám a hangsúlyok eltolódnak.

5.1.2. Típus a programozó szemszögéből

Most nézzünk egy sokkal formálisabb megközelítést, amely azonban már a programozó szemszögéből definiálja a típust, szétválasztva külső viselkedésének leírását (specifikáció) a tényleges, fizikai megvalósításától. A fejezetben használt modell részletes ismertetéséről lásd [Fót 83]-t.

Típusspecifikáció

A $T_S = (T, I_S, \mathbb{F})$ hármast *típusspecifikációnak* nevezzük, ahol

- T tetszőleges alaphalmaz,
- $I_S : T \to \mathbb{L}$[1] a specifikációs invariáns, $T_S = [I_S]$ (I_S igazsághalmaza, azaz azon elemek, amelyekhez I_S az *igaz* értéket rendeli) a *típusértékhalmaz*,
- $\mathbb{F} = \{F_1, F_2, \ldots, F_n\}$ olyan műveletek specifikációi, amelyek állapotterében szerepel T_S.

A típusspecifikáció a „mit" leírása. Csak azt határozza meg, hogy mit jelent a típus, mit csinálnak a műveletei. Ennek a leírásnak a birtokában a típus felhasználható a programozó számára.

Ez a definíció a korábban említett négy tulajdonság közül lényegében csak a jelentést adja meg. A specifikáció három eleme közül a T alaphalmaz a valós világ minket érdeklő objektumainak halmaza. A specifikációs invariáns (I_S) segítségével ezt a halmazt tovább szűkítjük, mivel elképzelhető, hogy a teljes alaphalmaz használata nem lehetséges a számítógép természetéből adódó korlátok miatt. Ha például a T halmaz a természetes számokat jelöli, akkor a specifikációs invariáns ezt valószínűleg valamilyen intervallumra szorítja meg (például $0 \ldots 2^{32} - 1$), mivel a számítógépek memóriája véges, így csak korlátos értékekkel képesek dolgozni. Ha a valós számokra gondolunk, akkor a korlátozás még erősebb, mivel a megszorítás eredménye még csak nem is intervallum lesz. A specifikációs invariáns eredményeként kapott halmaz, melyet *típusértékhalmaznak* nevezünk, tehát véges, esetleg megszámlálható számosságú. Ez utóbbi előfordulhat ha például egy olyan típusra gondolunk, amelynek alaphalmaza a fájlok halmaza, és ahol a fájlok méretét a típusspecifikáció nem korlátozza. Belátható, hogy ezt a megszorítást a típus specifikációjában kell elhelyezni,

[1] $\mathbb{L}$ a logikai értékek halmaza, azaz $\mathbb{L} = \{igaz, hamis\}$

mivel ez olyasmi, ami a típus szemantikáját befolyásolja, így erre az információra a típus használatakor szükség lehet.

A hármas harmadik tagja a típus számunkra jelentőséggel bíró viselkedéseit írja le, a típus műveleteit. Fontos, hogy ez a leírás is deklaratív („mit" jellegű), nem határozza meg a hogyant.

Figyeljük meg, hogy a típus a lehetséges értékek és a megengedett műveletek együttese. A programozási nyelvek beépített típusainál a szétválasztás nem is lehetséges, de az általunk létrehozott típusoknál is ügyelnünk kell ennek az egységnek a megmaradására.

Típusmegvalósítás

A $\mathcal{T} = (\rho, I, \mathbb{S})$ hármast *típusnak* nevezzük, ha

- $\rho \subseteq E^* \times T_S$ ábrázoló (reprezentációs) függvény,
- $I : E^* \to \mathbb{L}$ a típusinvariáns tulajdonság,
- $\mathbb{S} = \{S_1, S_2, \ldots, S_n\}$ olyan programok, amelyek állapotterében szerepel E^*.

Ez a leírás a típus megvalósításának leírása, a „hogyan"-ok megadása. Az E halmaz, az úgynevezett elemi értékek, E^* pedig az E elemeiből képzett véges sorozatok halmaza. E olyan halmaz, amelynek elemei már adottak a rendszerünkben. Lehet egy már korábban általunk megvalósított típus, vagy valamely, a használt programozási nyelvben rendelkezésre álló, egyszerű vagy összetett típus. Ha akarjuk, tekinthetjük a bitek $\{0, 1\}$ halmazát is E-nek, hiszen valahol legalul valóban bitsorozatokra képződik le minden adattípus, kényelmi okokból azonban nem szoktunk ennyire a részletekbe menni, hanem felhasználjuk a rendelkezésre álló más típusokat. Ez a lehetőség sokkal egyszerűbbé és érthetőbbé teszi a típus összetevőinek megadását.

Ábrázoló függvény • Az első összetevő, az ábrázoló (reprezentációs) függvény, valójában reláció, mivel a típusinvariánst ki nem elégítő elemekhez bármit rendelhet, de a gyakorlatban igen gyakran függvény vagy parciális függvény. Ez mondja meg, hogy bizonyos elemiérték-sorozatok milyen típusértékeket határoznak meg. A definícióból látható, hogy nem minden elemiérték-sorozathoz rendelődik típusérték, illetve egyes típusértékeknek lehet többféle ábrázolása. Képzeljük el például, hogy a racionális számokat egy egész számokból álló számpárral reprezentáljuk, ahol a páros első eleme a számlálót, a második a nevezőt jelenti. Ekkor nyilván nem rendelődik egyetlen racionális szám sem az olyan számpárokhoz, amelyek második tagja nulla, ugyanakkor a (kp, kq) alakú párok $(k, q \neq 0)$ mindegyike a p/q (p és q relatív prímek) racionális számot fogja ábrázolni.

Típusinvariáns • A hármas második tagja a típusinvariáns. Ez az invariáns tulajdonság azt írja le, hogy mely elemiérték-sorozatok határoznak meg valóban érvényes típusértékeket. Gyakran előfordul ugyanis, hogy az egyes értékeket ábrázoló sorozatok nem lehetnek tetszőlegesek, hanem valamilyen belső összefüggésnek kell eleget tenniük. Például a fenti példában csak azokat a számpárokat fogadjuk el helyesnek, amelyekben a nevező – azaz a második tag – nem nulla. További megkötés lehet, hogy a számláló és a nevező relatív prímek legyenek, illetve, hogy a nevező mindig legyen pozitív.

Típusműveletek • A típus harmadik összetevője a típusműveletek halmaza. Ezek a műveletek olyan programok, amelyeknek állapotterében szerepelnek az elemiérték-sorozatok. Ezen programok lesznek a típusműveletek megvalósításai.

Amennyiben egy ilyen programnak kimenő paramétere egy – típusértéket ábrázoló – elemiérték-sorozat, úgy a program utófeltételében szerepelnie kell, hogy a típusinvariáns teljesül az adott paraméterre. Ugyanakkor, ha egy ilyen sorozat bemenő paramétere a műveletnek, akkor a megfelelő program előfeltételében kikötheti a típusinvariáns teljesülését. Ebből látszik, hogy a program egyaránt függ a választott ábrázolástól és a típusinvariánstól, sőt mondhatjuk azt is, hogy a ábrázoló függvény és a típusinvariáns együttesét nevezzük a típus ábrázolásának.

A specifikáció és a típus kapcsolata

Az alábbiakban azt fogjuk meghatározni, hogy egy típus mikor felel meg egy adott típusspecifikációnak. A formális definíció a következő:

A $\mathcal{T} = (\rho, I, \mathbb{S})$ típus megfelel a $\mathcal{T}_S = (T, I_S, \mathbb{F})$ típusspecifikációnak, ha:

- $\rho([I]) = T_S$, és
- $\forall F \in \mathbb{F} : \exists S \in \mathbb{S} : S$ a ρ-n keresztül megoldja F-et.

Az első feltétel azt jelenti, hogy minden, a típusinvariánst kielégítő elemiérték-sorozat egy értelmes – azaz a specifikációs invariánsnak megfelelő – típusértéket ábrázol, továbbá a specifikációban meghatározott teljes típusérték-halmaz ábrázolható, a típusinvariánst kielégítő elemiérték-sorozatokkal.

A második feltétel pedig azt határozza meg, hogy minden előírt típusművelethez létezik megvalósító program a típusban. A ρ-n keresztüli megoldás azt jelenti, hogy a feladat állapotterében adott bemenő adatokat a reprezentációs függvény inverzrelációjának segítségével áttranszformáljuk az adott műveletet megvalósító program állapotterébe – azaz a típusértékeket az őket helyettesítő elemiérték-sorozatokkal helyettesítjük –, ott végrehajtjuk a programot és a kapott eredményt a ρ segítségével visszatranszformáljuk a feladat állapotterébe. Ha ekkor a művelet specifikációját kielégítő eredményt kapunk, akkor az adott program ρ-n keresztül megoldja a feladatot, vagyis tényleg az előírt műveletet valósítja meg, az adott ábrázolás mellett. Más szavakkal a program akkor oldja meg ρ-n keresztül a feladatot, ha úgy változtatja meg a típusértékeket ábrázoló elemiérték-sorozatokat, hogy az eredmény interpretációja megfelel a feladatban megfogalmazottnak.

Vegyük észre, hogy egy típusspecifikációnak több típus is megfelelhet, amelyek eltérhetnek az ábrázolásban, illetve a típusinvariánsban – mint például a korábban említett racionális számok –, vagy a típusműveletek megvalósításában.

5.1.3. A programozási nyelvek típusossága

Láthatjuk tehát, hogy az objektumok, amelyekkel programjaink dolgoznak, típussal bírnak, és ez a típus meghatározza az adott objektum viselkedését, a rajta végezhető műveleteket. A típus lényegében nem más, mint szemantikai szabályok, amelyek meghatározzák, milyen

műveletek *megengedettek* az objektumot végső soron ábrázoló bitsorozaton. Ha egy objektumon olyan műveletet próbálunk végezni, amely nem értelmezett az adott objektumra, akkor ez nyilvánvalóan egy szemantikai hiba a programunkban. A programozási nyelvek használatának egyik célja, hogy segítsék a programozót minél kevesebb hibával elkészíteni a programját, egészen pontosan minél több hibát felfedezni *fordítási időben*. Ennek érdekében a programozási nyelvek gyakran törekszenek arra, hogy minél több szemantikai elemet is képesek legyenek ellenőrizni az elkészített programban. Az egyik eszköz ezen cél megvalósítására a *típusosság* [Car 85].

A típusosság azt jelenti, hogy a programozási nyelvek típust rendelnek a programjainkban szereplő entitásokhoz. Típusa van minden konstansnak, operátornak, változónak és függvény szimbólumnak, bizonyos nyelvekben egyéb konstrukcióknak is. A kifejezések típusának meghatározásához egy *típuskikövetkeztető* rendszer használható. Bizonyos nyelvekben – ilyenek például az Ada vagy a Pascal – a szimbólumok típusa deklarációk formájában rögzített és a fordítóprogram ellenőrzi a definíció és a használat konzisztens voltát. Más nyelvekben – például az ML-ben – az explicit deklarációk helyett, ahol lehet a kifejezésből próbálja a fordítóprogram kikövetkeztetni a típust mindaddig, amíg a konzisztencia fenntartható.

Azokat a programozási nyelveket, amelyekben minden kifejezés típusa meghatározható fordítási időben *statikusan típusosnak nevezzük*. A statikus típusosság hasznos tulajdonság, de meglehetősen szigorú megkötéseket támaszt a programozási nyelvvel szemben. Ezek a követelmények szerencsére enyhíthetők, a konzisztencia megtartása mellett. Egy programozási nyelvet *erősen típusosnak* nevezünk, ha garantált, hogy minden kifejezés típusa konzisztens, még ha a típus meghatározása nem is lehetséges. Nyilvánvalóan minden statikusan típusos nyelv egyben erősen típusos is, de a fordítottja nem igaz. Az öröklődést és a belőle származó altípus-polimorfizmust támogató nyelvek például nem sorolhatók a statikusan típusos nyelvek közé, de számtalan közülük erősen típusos – például a C++, Java vagy Eiffel. A polimorfizmussal és különféle változataival bővebben foglalkozunk még a 11.1. részben és a 12. fejezetben.

Más nyelvekben a kifejezések konzisztenciája csak futási időben dönthető el, és így előfordulhat, hogy valamely művelet meghívása futási idejű hibát okoz. Ilyen nyelvek például a Smalltalk vagy a dBase, melyeket *gyengén típusosnak* nevezünk.

5.1.4. Típuskonverzió

Egy programozási feladat megoldása során szükséges lehet egy objektum típusát megváltoztatni. A típus megváltoztatását *típuskonverziónak* nevezzük. A típuskonverzió történhet automatikusan, és lehet explicit is. Ez utóbbi esetén a konverziót végző műveletet *konverziós operátornak* nevezzük.

Ábrázolás megváltoztatása

A típuskonverzió többféle célt szolgálhat. Az egyik lehetséges cél a reprezentáció megváltoztatása. Egy objektumot számtalan módon ábrázolhatunk a memóriában. Ezen lehetséges ábrázolások közül választjuk ki a céljainknak legjobban megfelelőt, amikor egy konkrét

feladat megoldásán dolgozunk. Bizonyos objektumoknak azonban egyszerre több ábrázolása is szerepelhet programunkban. Ilyen objektumok például a számok. Az egész számok például egyaránt szerepelnek az egész típusok és a valós típusok típusértékhalmazában is, így egyszerre fordulhatnak elő például lebegőpontos és kettes komplemens alakban is. A kétféle ábrázolás ugyanazt az objektumot takarja, de az objektumon végezhető műveletek megvalósítása kötődik az egyes reprezentációkhoz, így egy-egy művelet elvégzéséhez az objektumot az annak megfelelő alakra kell konvertálni. Egyes programozási nyelvekben – ezek közé tartozik az Ada nyelv is – ezeket a konverziókat mindig explicit jelölni kell. Más nyelvekben a konverziók egy része automatikusan végrehajtódik. A legtöbb nyelvben például a 2.2 + 1 kifejezés kiértékelésekor az 1 egész érték automatikusan lebegőpontos típusra konvertálódik, és a lebegőpontos számok közötti összeadás művelet hajtódik végre. Ezt a fajta típuskonverziót *bővítőnek* nevezzük, mivel a szűkebb típusértékhalmazról (egészek) térünk át a bővebbre (valósak). Ez a konverzió biztonságosan – adatvesztés nélkül – elvégezhető, ezért automatikus végrehajtása megengedhető. Ugyanez az oka annak, hogy a két lehetséges konverzió közül ezt hajtjuk végre, és nem a lebegőpontos értéket alakítjuk át egésszé (*szűkítő konverzió*).

Az alábbi példában egy rövid C, C++ vagy Java nyelvű programrészletet láthatunk.

```
double x;
int i;
x = 2.2 + 1;
i = 2.2 + 1;
```

A példában az *x* érték kiszámításakor egy bővítő típuskonverzió hajtódik végre, amely mindhárom nyelvben automatikus. Az *i* értékének meghatározásakor azonban két típuskonverzió történik. Az első, az összeadás elvégzése előtt a korábban látott bővítő konverzió, a második pedig az értékadás elvégzésekor egy szűkítő konverzió, amely az összeadás lebegőpontos eredményét az *i* változó egész típusára alakítja. Ez az átalakítás már nem tekinthető biztonságosnak, a konverzió során adatot veszítünk, az érték 3.2-ről 3-ra módosul. A C nyelv ennek ellenére automatikusan elvégzi az átalakítást itt is. A C++ szintén megengedi az átalakítást, de figyelmeztet a lehetséges adatvesztésre. Ezzel szemben a Java nyelvben ez a fajta konverzió automatikusan nem jön létre, a második értékadás a példában egy hibaüzenetet eredményez.

Karakterlánc-konverzió

Látszólag a reprezentációt megváltoztató típuskonverziók közé tartoznak az objektumot karakterlánccá konvertáló, illetve karakterláncból visszaállító műveletek is. Ezek az átalakítások azonban, noha a megfelelő műveleteket a konverziós operátorok közé soroljuk, nem típuskonverziók, ugyanis hatásukra megváltozik maga az ábrázolt objektum is. Ha a 42 számot karakterlánccá alakítjuk, az eredmény – a legtöbb programozási nyelvben legalábbis – nem a 42 szám egy másfajta ábrázolása lesz, hanem egy karakterlánc, amely a 4 és 2 karakterekből áll. Az ábrázolás megváltozása mellett, megváltozott a műveletek halmaza is, sőt lehetséges, hogy a karakterláncból nem is állítható vissza az eredeti objektum. Ilyen karakterlánc-konverzióra szolgál az Ada nyelv *Image* attribútuma, vagy a Java objektu-

mok *toString()* metódusa. Az Adában konverziót minden esetben explicit jelölni kell, míg a Java megengedi az automatikus konverziót mindaddig, amíg az az egyértelműség rovására nem megy.

Értelmezés megváltoztatása

Az ábrázolást megváltoztató típuskonverzió ugyanazon objektumhoz egy másik ábrázoló bitsorozatot állított elő, egy másfajta reprezentációt. A típuskonverziók másik változata az, amely az ábrázoló bitsorozatot változatlanul hagyja, de megváltoztatja annak értelmezését, azaz az ábrázolt objektumot. Az ilyen típusú konverzió a lefordított program kódjában nem jelenik meg, a program működésére hatása nincsen, csupán a fordítóprogram típusellenőrző rendszerének „megnyugtatására" vagy „kijátszására" szolgál.

Érdekes átmenetet képeznek az objektumorientált programozási nyelvekben az ősosztály és származtatott osztály közötti konverziók. Ha az adott nyelv az objektumokat referencia szerint tárolja – mint például a Java –, akkor a konverzió mindkét irányban elvégezhető – ha a hivatkozott objektum dinamikus típusa (lásd a 12. fejezetet) ezt megengedi – és lényegében az értelmezés megváltoztatását jelenti, hiszen az ábrázoló bitsorozat, mind a referencia, mind az objektum állapotát leíró memóriaterület változatlan marad. Ha azonban az objektumok érték szerint tárolódnak – például C++, vagy Eiffel kiterjesztett objektumok – akkor a konverzió csak a leszármazott osztálytól az ősosztály felé megengedett, amely az ábrázolás megváltoztatását is magával vonja, mivel a származtatott osztály által esetlegesen bevezetett további adattagok ilyenkor elvesznek. Ezen nyelveknél a bővítő és szűkítő konverziók hatása is módosul, hiszen a bővítő konverzió (a leszármazottról az ősre) adatvesztést eredményez, ezért a szűkítő konverzió később el sem végezhető.

Típuskonverziók a nyelvekben

C nyelv • A C nyelvben a *típuskonverzió (cast)* operátor egyszerre szolgál az ábrázolást megváltoztató típuskonverzióra a különféle numerikus típusok között, bár ennek jelölése nem szükséges, illetve az értelmezést megváltoztató konverzióra a különféle mutató és egész típusok között. Egy tetszőleges mutató típus konvertálható bármely más mutató típussá, illetve bármely egész típussá, mivel ez csupán az ábrázoló bitsorozat értelmezésének megváltoztatását jelenti. Nem konvertálható viszont lebegőpontos típusúvá, mivel ehhez a bitsorozatot is meg kellene változtatni. Megengedett ezen kívül még azonos alaptípus konstans és nemkonstans változata közötti konverzió, amely szintén változatlanul hagyja a reprezentáló bitsorozatot.

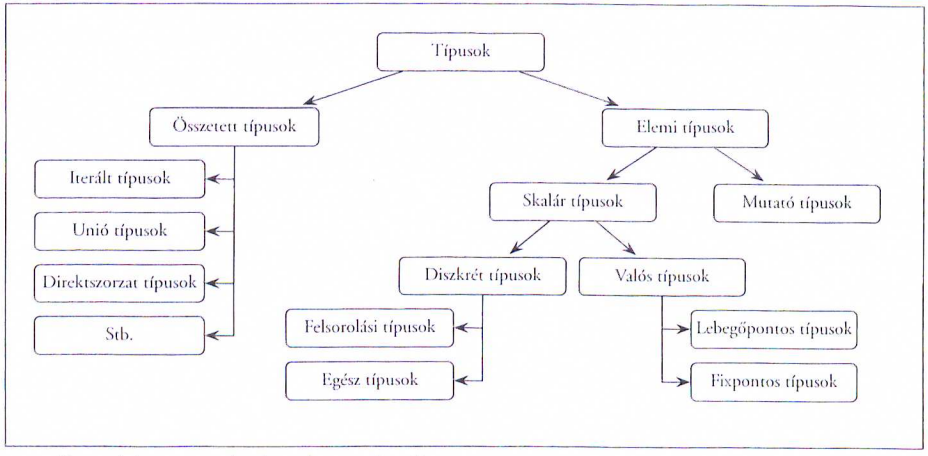
C++ • A C++-ban, a kompatibilitás megtartása végett, használható ugyan a C típuskonverziós operátora, de használata nem javasolt. A típuskonverzió eredendően veszélyes művelet, mivel kiiktatja a nyelvbe épített biztonsági mechanizmusokat, így csak alapos odafigyelés mellet szabad használni. Az egyetlen, mindenféle konverziós formát összemosó operátor helyett, a C++ a nyelv több, a típuskonverzió különféle alkalmazásait gondosan szétválasztó változatot vezetett be. Ez a szétválasztás lehetővé teszi, hogy minden helyzetben a feladatnak megfelelő operátor használatával minimalizáljuk a típuskonverzióból eredő kockázatokat.

- A **static_cast** operátor egymással kapcsolatban levő típusok között végez konverziót, például az ugyanazon osztályhierarchiában lévő mutatótípusok között, a numerikus típusok között vagy az egész és felsorolási típusok között.
- A **dynamic_cast** operátor az egyazon osztályhierarchiába tartozó mutatók közötti konverzió eszköze, amely azonban futási időben végez típusellenőrzést, így például az ősosztályról a leszármazott osztályra történő konverziókor ellenőrzi, hogy a hivatkozott objektum dinamikus típusa szerint csakugyan elvégezhető-e az értékadás (lásd a 12. fejezetet).
- A **const_cast** operátor segítségével egy konstans típusú objektum konvertálható a neki megfelelő nemkonstans típusúvá.
- A **reinterpret_cast** segítségével olyan típusok között konvertálhatunk objektumokat, amelyek között az egyetlen kapcsolat, hogy az ábrázoló bitsorozatok azonos hosszúságúak. Ez lényegében a C típuskonverziós operátorának felel meg. Míg a másik három konverziós operátor hordozható, addig a **reinterpret_cast** használata platform- vagy megvalósításfüggő kódot eredményezhet.

Java • A Java nyelvben a skalár típusok között a típuskonverzió a reprezentáció megváltoztatását jelenti, amely bővítő konverzió esetén automatikus, szűkítő konverzió esetén explicit jelölni kell. A referencia típusok esetén a konverzió csak korlátozottan használható. Egy leszármazott osztályra vonatkozó referencia automatikusan konvertálódik ősosztályra vonatkozó referenciává. Szintén automatikus a konverzió egy osztályra vonatkozó referenciáról az osztály által megvalósított interfészekre vonatkozó referenciákra. Az ellentétes irányú konverziók is lehetségesek, de ezek futási idejű típusellenőrzéssel járnak, amelynek során a virtuális gép megvizsgálja, hogy a hivatkozott objektum dinamikus típusa szerint elvégezhető-e az értékadás. Ha igen, az értékadás megtörténik, ha nem, egy kivétel váltódik ki.

Eiffel • Eiffelben a változók kétféle módon tárolhatják az objektumokat: hivatkozás és érték szerint. A hivatkozás szerinti tárolásnál az objektum egy referenciája tárolódik a változóban és maga az objektum a dinamikus tárolási területen helyezkedik el. Az érték szerinti tárolási módnál a változó magát az objektumot tartalmazza. Az ilyen objektumokat *kiterjesztettnek (expanded)* nevezük. Az típuskonverzió szabályai lényegében megfelelnek a bevezetőben leírtaknak, de hatásuk függ az objektum tárolásának módjától. A származtatott osztályról az ősosztályra a konverzió automatikus, de ha az objektum kiterjesztett, akkor a konverzió adatvesztést eredményezhet, mivel a származtatás során bevezetett új adattagok elvesznek. Ilyen objektumoknál az ellentétes irányú konverzió nem megengedett. A referencia szerint tárolódó osztályok esetén van mód az ősosztályról a származtatott osztályra irányuló konverzióra is. Ennek eszköze az úgynevezett *fordított értékadási kísérlet*. A fordított értékadási kísérlet során az ősosztályra vonatkozó referenciát megpróbáljuk értékül adni a származtatott osztályra vonatkozó referencia típusú változónak. Ha a dinamikus típusok alapján az értékadás nem végezhető el, akkor a változó értéke *void* lesz.

CLU • A CLU egy egészen különleges fajtáját vezette be az értelmezést megváltoztató típuskonverziónak. Egy absztrakt adattípus megvalósításakor, a CLU típusrendszere külön-



5.1. ábra: Típusosztályok és kapcsolataik

bözőnek tekinti az absztrakt adattípust és az ábrázoláshoz használt konkrét adattípust, ezért közöttük kötelező az explicit típuskonverzió (lásd a 10.5.3. részt).

Ada • Végezetül érdemes még megemlíteni az Ada nyelvben a konverziós lehetőséget a származtatott típus és az őstípus között. Ez is csupán az értelmezést megváltoztató konverzió, hiszen a származtatott típus örökli az őstípus reprezentációját, ám segítségével kapcsolatot teremthetünk az egyazon őstípusból származtatott különféle típusok között (lásd az 5.8. részt).

5.2. Típusok osztályozása

Az egyszerűbb áttekinthetőség végett, megpróbáljuk osztályozni a programozás nyelvekben fellelhető típusokat. Az osztályozás alapja a típusok szerkezete lesz.

A legmagasabb szinten kétféle típust különíthetünk el. Az *elemi típusok* azok, amelyek logikailag felbonthatatlanok, míg az *összetett típusok* más, meglevő típusokból tevődnek össze. Ebben a fejezetben az elemi típusokat vizsgáljuk meg, az összetett típusokkal részletesen a típuskonstrukciókról szóló (6.) fejezetben foglalkozunk.

Az elemi típusok két további nagy csoportra bonthatók. Az első a *mutató típusok*, amelyekről az 5.6. részben lesz szó. A másik csoport a *skalár típusok*. A skalár típusok egyetlen elemi mennyiséget vagy értéket reprezentáló típusok.

A skalár típusok tovább bonthatók *valós-* és *diszkrét* típusokra. Az előbbiek a valós számok különféle reprezentáció szerinti megvalósításait jelentik, a két leggyakoribb a *lebegőpontos*, illetve a *fixpontos* típusok. A diszkrét típusok még tovább bonthatók *felsorolási-* és *egész* típusokra. Az 5.1. ábra összefoglalja a fenti típusosztályokat és a közöttük levő relációkat.

5.2.1. Típusosztályok

A típusosztályoknak nem csak a típusok áttekinthetőbb kezelésében van szerepe, hanem gyakran megjelennek a programozási nyelvekben is és a típusosztályhoz való tartozás alapvetően meghatározza a típus használhatóságát. Például igen gyakori az a megkötés, hogy a **for** ciklus ciklusváltozójának diszkrét típusúnak kell lennie, vagy hogy tömböket indexelni csak diszkrét, sőt, esetenként csak egész típusokkal lehet. Szintén meghatározó lehet a típusosztály az adott típusra engedélyezett műveletek halmazát tekintve. Igen gyakori, hogy ha létrehozunk egy típust egy típusosztályon belül, akkor az adott típushoz automatikusan létrejönnek a típusosztályra jellemző műveletek.

5.2.2. Attribútumok az Ada nyelvben

Az Ada programozási nyelv erősen épít a típusosztályokra. Az egyes típusosztályokhoz meghatározott műveletek tartoznak, amelyek automatikusan létrejönnek minden, az osztályhoz tartozó típusra. Ezek a műveletek egyrészt lehetnek különféle operátorok, például az Ada nyelvben az összes skalár típus rendezett, és a relációs operátorok (<, <=, = stb.) minden skalár típuson értelmezettek. A másik fajtája a műveleteknek az úgynevezett *attribútumok*. Az attribútumok speciális, a típusosztályokra jellemző műveletek. Három fajtájuk van:

- Vannak olyan attribútumok, amelyek a típus különböző jellemzőit adják meg. Ilyenek például egy tetszőleges *S* skalár típuson értelmezett *S'First*, illetve *S'Last* attribútumok, amelyek a típus legkisebb, illetve legnagyobb elemét adják meg.
- Vannak olyan attribútumok, amelyek a típus műveletei közé tartoznak. Jellemző példák a különféle konverziós függvények, például egy tetszőleges *T* diszkrét típushoz definiáltak a *T'Pos* és *T'Val* attribútumok, amelyek függvények. Az első a *T* típusról képez az egész számok típusára, és megadja, hogy a paraméterül kapott elem hányadik a felsorolási típus elemei között. A másik függvény az ellentétes irányú átalakítást végzi.
- Az attribútumok harmadik csoportja érdekes keveréke az előző kettőnek. Keveréke, mivel a második típushoz hasonlóan az objektumokon végez műveletet és nem a típuson, viszont segítségével az adott objektum típusára vonatkozó információkhoz juthatunk. Például a *'First* attribútum értelmezett a tömb típusokon is, ilyenkor a tömb indextartományának alsó határát adja meg. Az Ada nyelvben azonban megengedett olyan tömb típust definiálni, amelynél az indexintervallumot csak a konkrét tömb objektum megadásakor kell meghatározni (lásd a 6.4.3. részt). Az attribútum az ilyen módon létrehozott objektumok esetében is megadja az indextartomány alsó határát, de ekkor az attribútumot nem a típus nevével minősítve használjuk, hanem az objektumot tartalmazó változóéval.

Vizsgáljuk meg az attribútumokat az alábbi példában, amelyben a *HétTörpe* felsorolási típust definiáljuk.

type *HétTörpe* **is** *(Tudor, Vidor, Szende, Szundi, Hapci, Morgó, Kuka);*

Pusztán abból a tényből kifolyólag, hogy a *HétTörpe* típust felsorolási típusként definiáltuk, következik, hogy számtalan művelet értelmezett rajta.

Diszkrét típus volta miatt egész típusúvá, illetve vissza is alakítható, rendre a *HétTörpe'Pos*, illetve *HétTörpe'Val* konverziós attribútumokkal.

Mivel a diszkrét típusok egyben skalár típusok is, a *HétTörpe* típus rendezett lesz, értelmezettek rá a rendezési relációs operátorok, képezhető belőle intervallum (**range**) és vizsgálható egy elemnek egy intervallumhoz való tartozása (**in**, **not in**). Ezen felül értelmezett rá még további tizennégy attribútum, amelyek a skalár típusok mindegyikére adottak, és többek között lehetővé teszik a legkisebb, illetve legnagyobb elemek lekérdezését, a karakterlánccá történő konverziót vagy éppen egy elemnek a rendezés szerinti rákövetkezőjének megállapítását.

5.3. Skalár típusosztály

A skalár típusok a programozási nyelvek legegyszerűbb típusai. A típusértek-halmaz elemei egyszerű, belső szerkezettel nem rendelkező elemek. Számtalan típus tartozik ehhez a típusosztályhoz, kezdve az egész számoktól egészen a felsorolási típusokig, így a típusok műveleteinek halmaza is széles spektrumban változik. Ennek ellenére van néhány közös jellemzője a skalár típusoknak, amelyeket ezen a szinten is megvizsgálhatunk.

5.3.1. Ábrázolás

A skalár típusosztály szinte minden programozási nyelvben megjelenik valamilyen formában. Még az objektumorientált nyelvek is megkülönböztetik a skalár típusokat az egyéb osztályoktól.

Ennek oka az, hogy a skalár típusosztály elemei olyan típusok, amelyeket általában nem is a programozási nyelv valósít meg, hanem támogatásuk, legalább részben, a számítógépek processzoraiban is megjelenik. Minden ma használatos processzor megvalósítja az egész számok típusát, kettes komplemens kóddal ábrázolva a számokat, jellemzően 16, 32, esetleg 64 biten. Ezen kívül a legtöbb általános célú processzor megvalósítja a valós számok típusát is, lebegőpontos számábrázolást használva. Ezért a programozási nyelvekben, ezen típusok megvalósításának leghatékonyabb módja az, ha felhasználjuk a processzor adta támogatást. Másrészt ezek a típusok a leggyakrabban használtak az egyes programozási feladatok megvalósításánál nagy számban előfordulnak, így a tárolás és a műveletek hatékonysága egyaránt létfontosságú a gyakorlati alkalmazhatóság szempontjából.

A programozási nyelvek tehát általában hasonlóan kezelik a skalár típusokat mind az ábrázolás, mind a művelethalmaz szempontjából. Még azokban a programozási nyelvekben is, ahol az összetett típusok vagy osztályok valójában referenciák a megfelelő objektumokra – ilyen nyelvek például az Eiffel, a Java vagy a CLU –, a skalár típusok érték szerint tárolódnak, azaz a változó magát az objektumot, és nem annak referenciáját tárolja. Az Eiffel az ilyen típusokat *kiterjesztett típusoknak* nevezi. Vannak azért kivételek is, például a Smalltalk, amelyben, mivel a változók nem rendelkeznek típussal, minden objektum, így a skalár típusok is, referencia szerint tárolódnak.

5.3.2. Műveletek

A skalár típusok legfőbb közös jellemzője, hogy rendezettek, ezért a közös műveletek a rendezettséghez kapcsolódnak:

- Relációs operátorok ($<, \leq, >, \geq, =, \neq$) általában értelmezettek rajtuk.
- Gyakori művelet a rákövetkező, illetve megelőző elem meghatározása.
- Szintén gyakori művelet a típus legkisebb illetve legnagyobb elemének meghatározása.

5.3.3. Az Ada skalár típusai

Az Ada 95 programozási nyelvben a skalár típusosztályhoz tartozó típusok a következő műveletekkel rendelkeznek:

- A relációs operátorok: <, <=, >, >=. =, /=
- Intervallumképzés: **range** ⟨*alsóhatár*⟩..⟨*felsőhatár*⟩.
- Intervallumhoz tartozás lekérdezése: **in**, **not in**

Ezen felül egy adott *S* skalár altípusra (lásd az 5.8. részt) a következő attribútumok használhatók:

- *S'First* és *S'Last* rendre a típus legkisebb, illetve legnagyobb eleme.
- *S'Range* a típusértékhalmaz (**range** *S'First* .. *S'Last*).
- *S'Base* az *S* altípus alaptípusa.
- *S'Min* és *S'Max* kétparaméterű függvények, amelyek két *S*-beli érték közül visszaadják rendre a kisebbet, illetve a nagyobbat.
- *S'Succ* és *S'Pred* egyargumentumos függvények, amelyek a paraméterül kapott *S*-beli értékhez megadják rendre annak rákövetkezőjét, illetve megelőzőjét. A műveletek nem ciklikusak; ha a típus legnagyobb, illetve legkisebb elemére hívjuk meg őket, akkor *CONSTRAINT_ERROR* kivételt kapunk. Érdemes megjegyezni, hogy a rákövetkező, illetve megelőző függvényeknek a skalár típusosztályhoz rendelése meglehetősen szokatlan, hiszen ha belegondolunk, ide tartoznak a valós számok is, azokon pedig ilyenfajta rákövetkezés – matematikai értelemben – nincs. A valós számoknak csak egy véges részhalmazát tudjuk reprezentálni – mind lebegőpontos, mind fixpontos számábrázolás esetén –, így persze meghatározható minden ábrázolható valós számhoz az őt követő valós szám, ám ez a rákövetkezés reprezentációfüggő, a matematikából megismert valós számokra nehezen leképezhető.
- *S'Image* és *S'Wide_Image* egyargumentumos függvények, amelyek a megadott *S* típusú értéket karakterlánccá alakítják. Az *S'Image* függvény 8 bites (ASCII) karaktereket tartalmazó karkaterláncot ad vissza, a másik függvény 16 bites (ISO 10646) karakterláncot.
- *S'Value S'Wide_Value* egyparaméterű függvények egy karakterláncot alakítanak át a neki megfelelő *S*-beli értékké. A két attribútum gyakorlatilag inverze a nekik megfelelő *Image* attribútumoknak.

- Az *S'Width* és *S'Wide_Width* attribútumokkal lekérdezhetjük, hogy az *S* elemeiből milyen hosszúságú karakterlánc képződik a megfelelő *Image* attribútumok használatakor.

5.4. Diszkrét típusosztály

A diszkrét típusok alapvetően kétfélék. Ebbe a típusosztályba tartoznak a felsorolási és az egész típusok. Szinte minden programozási nyelv támogat néhány típust a diszkrét típusosztályból. Mivel nem minden nyelv támogatja a felsorolási típusokat, előfordul, hogy ez a típusosztály egybeesik az egész típusokkal.

A típusosztály új műveletet általában nem vezet be, jelentőssége inkább abban áll, hogy igen gyakran ez a legbővebb típusosztály, amelyet tömbök indexelésére, illetve **for** ciklusokban a ciklusváltozó típusaként használhatunk.

Kivételként meg kell említeni itt az Ada nyelvet, amelyben a diszkrét típusokhoz két attribútum is tartozik. Ha *S* egy diszkrét típus tetszőleges altípusa, akkor

- Az *S'Pos* attribútum egy függvény, amely az *S* típusról képez az egész számokra, és megadja, hogy a paraméterként kapott *S*-beli érték hányadik eleme az *S* típusnak. Egész típusok esetén ez lényegében maga a megadott érték lesz, míg a felsorolási típusok esetén az elemek a felsorolás sorrendjében, 0-tól kezdődően számozódnak.
- Az *S'Val* attribútum az ellenkező irányú konverzióra szolgál.

Az attribútumok jelentősége például a tömbök kezelésénél válik nyilvánvalóvá. Az Ada tetszőleges diszkrét típussal megengedi a tömbök indexelését. A fenti műveletek segítségével olyan programrészletek készíthetők, amelyek függetlenek az indextípustól, hiszen általuk tetszőleges tömb indexei egész számokká alakíthatók, amelyek már rendelkeznek az egészek minden műveletével.

5.4.1. Felsorolási típusok

A felsorolási típusok típusértékhalmazát egyszerűen az elemek felsorolásával adhatjuk meg. Ábrázolásukat tekintve az elemek általában a megadásuk sorrendjében egész számokra képeződnek le. Egyetlen műveletpárral bővül a felsorolási típusok eszköztára, amelyek segítségével konvertálhatunk egy felsorolási típus és egy egész típus között. Fontos megjegyezni, hogy a felsorolási típus fogalma egyszerre takar egy típusosztályt és egy típuskonstrukciós eszközt. Azért nem került mégsem a típuskonstrukciók közé, mivel egyrészt nem már meglévő típusokból készít újat, hanem egy teljesen új típust hoz létre, másrészt mert egyes nyelvek beépítetten is tartalmaznak felsorolási típusokat.

A Pascal típusú nyelvekben általában jelen van a felsorolási típus mint típuskonstrukciós eszköz. Az alábbi példa a korábban már Ada nyelven megismert *HétTörpe* típus Pascal nyelvű változata.

type *HetTorpe* = *(Tudor, Vidor, Szende, Szundi, Hapci, Morgo, Kuka);*

Sok programozási nyelvből kimaradt a felsorolási típus, hiszen elég könnyen kiváltható valamilyen egész típussal, és a felsorolandó elemekhez rendelt, ilyen típusú konstans értékekkel. A C és C++ nyelvek még némi szintaktikus segítséget is adnak ehhez a megoldáshoz. Az **enum** szerkezet nem más, mint egész konstansok felsorolása, amelyek automatikusan egyesével növekvő értéket kapnak, hacsak nem adunk meg más értéket nekik. Az alábbi példában a *Napok* típus a hét napjait sorolja fel – számértékük 0-tól 6-ig terjed – kiegészítve egy különleges elemmel (*NincsIlyenNap*), amely a 999 értéket veszi fel.

```
typedef enum {
    Hetfo, Kedd, Szerda, Csutortok, Pentek, Szombat,
    Vasarnap, NincsIlyenNap = 999
} Napok;
```

Sajnos ennek a megoldásnak több hátránya is van. Ha a program több felsorolási típust is használ – mondjuk van egy *Honapok* típusunk is –, semmi nem akadályoz meg abban, hogy a hét egy napját értékül adjam egy *Honapok* típusú változónak, mivel valójában mindkettő egész szám. Sőt, akár össze is adhatom őket, hiszen egész számokon értelmezett az összeadás művelet is.

Hasonló hatás érhető el az Eiffelben a **unique** deklarációk segítségével.

```
Piros, Feher, Zold: INTEGER is unique;
```

Sokkal jobb megoldás az, amit a CLU kínál a felsorolási típus kiváltására. A nyelv egyik uniójellegű típuskonstrukciós eszköze, a `oneof` (részletesebben lásd a 6.2.3. részt), melynek segítségével olyan típust hozhatunk létre, amely a felsorolási típushoz hasonlóan viselkedik. Mivel ebben a megoldásban igazából a címkézett unió konstrukciónak a címke része hordozza az információt, ezért az unióban résztvevő komponenstípusoknak célszerű a nyelv egyik legegyszerűbb típusát, a `null` típust választani, amelynek egyetlen lehetséges értéke a `nil`. Az eredmény kicsit nehézkesen használható, de legalább a típusosság megmarad, a különböző felsorolási típusok értékei nem adhatók egymásnak értékül.

```
Szinek = oneof [ piros, sarga, zold : null ]
lampa : Szinek := Szinek$make_zold(nil)
...
tagcase lampa
  tag piros :
    lampa := Szinek$make_zold(nil)
  tag sarga :
    lampa := Szinek$make_piros(nil)
  tag zold :
    lampa := Szinek$make_sarga(nil)
end
...
if Szinek$is_piros(lampa) then
  Auto$megallas(en_autom)
end
```

Ennek a megoldásnak egy továbbfejlesztett változatát használhatjuk a Java nyelvben, amelyből szintén kimaradt a felsorolási típus típuskonstrukció. Felsorolási típusként itt egy speciális osztályt használunk, amelynek nincs nyilvános konstruktora, ellenben az adott osztályhoz tartozó osztályszintű konstansként tartalmazza a felsorolási típus minden elemét. Az osztályból így újabb példányok nem hozhatók létre, egy ilyen típusú változó értéke vagy **null**, vagy az előre létrehozott objektumok valamelyike. A megoldás erősen típusos tehát, nincs értékadás a különböző felsorolási típusok között, az osztályon belül megvalósíthatjuk az összes kívánt műveletet a típushoz – rákövetkezési függvény, rendezés, karakterlánccá alakítás, konverzió egészekre és vissza stb. –, sőt ha az osztályt nem **final**-ként definiáljuk, akkor az öröklődés segítségével újabb elemekkel bővíthetjük is a felsorolási típust. A megoldásról részletes leírás található [Blo 01]-ben. Nézzük meg a korábbi színes példát:

```
public final class Színek {
    static private int számláló = 0;

    private final String név;
    private final int sorszám;

    private Szinek(String név) {
        this.név = név;
        this.sorszám = Színek.számláló++;
    }
    static public final Színek PIROS = new Színek("Piros");
    static public final Színek SÁRGA = new Színek("Sárga");
    static public final Színek ZÖLD = new Színek("Zöld");

    static private final Színek[ ] elemek =
        {PIROS, SÁRGA, ZÖLD}

    public String toString() {
        return név;
    }
    public Színek következő() {
        return elemek[(sorszám + 1) % elemek.length];
    }
    . . .
}
```

5.4.2. Egész típusok

Az egész típusok szinte minden programozási nyelvben megtalálhatók, valamint ábrázolásuk és műveleti halmazuk is nagyban hasonló, mivel gyakorlatilag a processzorba épített egész típus megvalósítást használják. Az egész típusok támogatásánál a programozási nyelvek általában a lehető legnagyobb hatékonyságra törekszenek, mivel ezek a típusok talán a

leggyakrabban használtak a programozás során. Ez a hatékonyságra törekvés néhány érdekes kompromisszumhoz vezetett az egész típusok megvalósításában.

Típusértékhalmaz

Általában többféle egész típus is rendelkezésre áll a nyelvben, amelyek eltérhetnek egymásból az ábrázoláshoz használt bitek számában – azaz, hogy az egész számoknak egy milyen hosszúságú intervallumát tudjuk ábrázolni –, illetve abban, hogy előjelesek vagy előjel nélküliek az értékek – azaz, hogy az intervallum szimmetrikus-e a nullára[2], vagy csak nemnegatív értékeket tartalmaz.

Ezektől az alapesetektől csak ritkán térnek el a programozási nyelvek. A számábrázoláshoz használt bitek száma is általában nagyon hasonló, jellemző értékek a 8, 16, 32, esetleg 64. A modernebb programozási nyelvek specifikációja gyakran nem köti meg pontosan a számábrázoláshoz használandó bitek számát, csupán egy alsó korlátot ad rájuk. Ezek a korlátok azonban igen gyakran egybeesnek a fenti értékekkel, és a tényleges megvalósítások is a fenti értékeket használják, hatékonysági okokból.

Műveletek

Az egész típusok műveletei, az esetleg a típusosztályokból adódó speciális műveleteken – konverziós műveletek, típusértékhalmazt lekérdező műveletek stb. – túl általában a matematikából megszokott műveletek – összeadás, kivonás, szorzás, osztás, esetleg hatványozás pozitív kitevővel – és igen gyakran bitműveletek – léptetés jobbra, balra, bitenkénti és, vagy, kizáró vagy és negáció. A bitműveletek ismét a hatékonyságra való törekvés jelei. A matematikában ilyen jellegű műveleteket szinte soha nem használunk az egész számokon, jelentésük igazán érthetően csak az ábrázolás ismeretében definiálható, ami ellentmond a típusspecifikáció és típusmegvalósítás szokásos szétválasztásának, hiszen abban az ábrázolás a megvalósítás része. Mindazonáltal igen gyakori, hogy bizonyos feladatok megoldásánál, a megfelelő hatékonyság elérése érdekében, kénytelenek vagyunk kihasználni az egész számok ábrázolásának specialitásait, illetve felhasználni ezeket a „furcsa" műveleteket. Mindezen felül, a számítógépek processzorai ezen műveleteket szintén beépítetten tartalmazzák, sőt igen gyakran ezen műveletek végrehajtása a legolcsóbb a processzorban.

A műveletek közül kettő érdemel különleges figyelmet, mivel mindkettőnél fennáll a veszélye, hogy a művelet kivezet a típusértékhalmazból. Az egyik az osztás, a másik a negatív kitevőjű hatványozás, hiszen mindkét esetben az eredmény könnyen lehet, hogy nem egész szám lesz. Az osztásnál a legtöbb nyelv az úgynevezett *egészosztást* valósítja meg, azaz a/b osztás eredménye a hányados egészrésze lesz. Azon nyelvek pedig, amelyekben van hatványozás operátor, általában nem engedélyezik a negatív kitevők használatát.

A többi művelet is kivezethet bizonyos esetekben a típusértékhalmazból, ám ezeknél az eredmény továbbra is egész szám, csupán meghaladja a számábrázolás korlátait. Bizonyos nyelvek (például az Ada) az ilyen eseteket futási időben ellenőrzik, és valamilyen futási idejű hibát – a kivételkezelést támogató nyelvekben kivételt – váltanak ki. A legtöbb nyelv

[2] A kettes komplemens számábrázolásban az intervallum nem lesz teljesen szimmetrikus, egy k bites egész szám megvalósítás típusértékhalmaza a $[-2^{k-1} \ldots 2^{k-1} - 1]$ intervallum.

azonban ilyenkor a túlcsorduló bitekkel nem törődik, azaz az eredménynek csak a kisebb helyiértékű bitjei közül tart meg annyit, amennyi belefér az adott típus számábrázolási korlátaiba. Ha k bites előjel nélküli számokkal számolunk, akkor ez a viselkedés megfelel a matematikából megismert moduló 2^k maradékosztályokon végzett műveleteknek, de előjeles típusok esetén igen furcsa eredményeket szolgáltathat. Ha például van két előjeles 8 bites értékem – legyen például a 96 és a 32, binárisan 00110000, illetve 00010000 –, ezeknek az összege – 128, binárisan 10000000 – már meghaladja az ábrázolható legnagyobb pozitív számot, amely a 127. Ha azonban a nyelv nem támogatja a futási idejű intervallum ellenőrzést, akkor a két pozitív szám összege −128 lesz, mivel történetesen ennek a kódja az 10000000 a kettes komplemens kódolás szerint.

Egész típusok a Pascalban

A Pascal nyelv típuskészlete megvalósításfüggő. A Turbo Pascal ötféle egész típust támogat. Ezek a 8 bites, előjeles *ShortInt*, a 8 bites előjel nélküli *Byte*, a 16 bites, előjeles *Integer*, a 16 bites, előjel nélküli *Word* és a 32 bites, előjeles *LongInt*. A nyelv választhatóan támogatja a futási idejű intervallum-ellenőrzést is, azaz az ilyen működés fordítási lehetőségként bekapcsolható. Ha be van kapcsolva, a program a műveletek elvégzésekor ellenőrzi, hogy történt-e túlcsordulás és futási idejű hibát vált ki.

A műveletek halmaza nagyjából a szokásos:

- Az összeadás (+), kivonás (−) és szorzás (*) a szokásos.
- Az osztásnak kétféle változata van. A /-rel jelölt osztás művelet meglepő módon (lásd az 5.7.2. részt) kivezet az egész számok köréből, eredménye valós szám lesz. Ha szigorúan egészek közötti osztásra van szükségünk, akkor a **div** operátort kell használni. Az osztási maradék meghatározására a **mod** operátor használható.
- Támogatja az egyszerű bitaritmetikai műveleteket is: a balra és jobbra léptetés (**shl**, **shr**), valamit a bitenkénti negáció (**not**), és (**and**), vagy (**or**) és kizáró vagy (**xor**) műveleteket.
- Az összetettebb matematikai műveletek közül támogatja az abszolútérték (*Abs*) és négyzetre emelés (*Sqr*) függvényeket.

A CLU `int` típusa

A CLU-ban az egész típusosztályt az `int` típus képviseli. A típusértékhalmaz megvalósításfüggő. A műveletek halmaza nagyjából a megszokott, ellenben jelölésük eltér a megszokottól. A CLU az alap matematikai műveleteket is prefix módon, angol nevük használatával jelöli. Hogy a kifejezések írását mégis megkönnyítse, a legtöbb művelethez van megfelelő szintaktikus cukor[3]. A műveletek a következők:

- A matematikai alapműveletek: összeadás (`add`, +), kivonás (`sub`, -) szorzás (`mul`, *), osztás (`div`, /), maradékképzés (`mod`).

[3] A CLU nyelvben bizonyos műveleteknek a nyelvben alapértelmezett prefix jelölésmódján felül létezik egy infix operátor alakja is. Ezt nevezzük *szintaktikus cukornak (syntactic sugar)*, mivel csupán egy egyszűsített jelölés, amit a fordítóprogram automatikusan behelyettesít a prefix formával (lásd az 5.7.2. részt).

- Néhány összetettebb matematikai művelet: nemnegatív egész kitevős hatványozás (`power`), abszolútérték (`abs`), illetve két elem maximuma (`max`) és minimuma (`min`).
- A rendezési relációk: kisebb (`lt`, <), nagyobb (`gt`, >), kisebb vagy egyenlő (`le`, <=), nagyobb vagy egyenlő (`ge`, >=), egyenlő (`equals`, =).
- Konverziós operátorok, amelyek karakterláncból egész számot (`parse`) és viszont (`unparse`) csinálnak.
- Érdekességet jelentenek az iterátorműveletek, amelyek végiglépkednek egy egész intervallum elemein, esetleg a megadott lépésközzel. Az egész típus két alapértelmezett iterátora a `from_to` és a `from_to_by`.

Egész típusosztály az Adában

Az Ada az egész típusosztályt két további alosztályra bontja, az *előjeles egészekre* és a *moduló típusokra*. Az előbbieknél az egész számok valamely intervalluma a típusértékhalmaz, míg az utóbbinál a moduló m maradékosztályok, a típusdeklarációban megadott m értékre. Az eltérés a kétféle típus között annyi, hogy a moduló típus értékkészlete a $0 \ldots m-1$ intervallum lesz és az értékkészlet ciklikus, azaz például $m-1$-hez 1-et adva 0-át kapunk. Az előjeles típusok típusértékhalmaza tartalmazhat negatív értéket is, de nem ciklikus; ha egy művelet eredménye nem esik a megadott intervallumba, *CONSTRAINT_ERROR* kivétel váltódik ki.

```
-- Egy bájtnyi (0..255) érték:
type Bájt is mod 256;

-- Tengerszint feletti magasság (méterben):
Type Magasság is range -15000..9000;
```

Noha az Adában így látszólag tetszőleges számú egész típust hozhatunk létre, valójában ezek a típusok mind a beépített névtelen[4] *root_integer* típusból, típusszármaztatással jönnek létre. Ezáltal a programjaink jobban hordozhatók lesznek, hiszen az általunk használt egész típusok értékkészlete pontosan megfelel a program igényeinek, nem fordulhat elő, hogy egy adott megvalósítás észrevétlenül megváltoztassa az ábrázolható számtartományt, nehezen felderíthető, futási idejű hibákat okozva. A *root_integer* típus ugyan megvalósításfüggő, de a saját egész típusaink deklarációjának kiértékelésekor, fordítási időben kiderül, ha az általunk igényelt számtartomány meghaladja az adott megvalósítás képességeit.

A nyelv beépítetten is tartalmaz néhány egész típust. Minden megvalósítás kötelezően tartalmazza az *Integer* típust, amelynek legalább a $(-2^{15}+1) \ldots (2^{15}-1)$ intervallumot tartalmaznia kell. Ezen felül a nyelv specifikációja megenged további beépített típusokat (*Short_Short_Integer*, *Short_Integer*, *Long_Integer*, *Long_Long_Integer*). Ezeknek sem határozza meg a pontos értékkészletét, csupán annyit ír elő, hogy a típusértékhalmazok egy-

[4] A típus névtelensége itt azt jelenti, hogy ezzel a típussal változó nem hozható létre. A *root_integer* nem egy tényleges típus a nyelvben.

máshoz való viszonyának tükröznie kell a típusok angol nevének jelentését[5], valamint a *Long_Integer* típusnak legalább a $(-2^{31}+1)\ldots(2^{31}-1)$ intervallumot tartalmaznia kell.

A műveletek halmaza mindkét típusosztályban azonos, eltekintve a fentebb említett ciklikusságot. A tartalmazó típusosztályoktól örökölt műveleteken kívül a következők állnak rendelkezésre:

- A szokásos matematikai alapműveletek: összeadás (+), kivonás (−), szorzás (*) és osztás (/).
- Maradékképzésre kétféle operátor is létezik. A **mod**, amelyre $a = b \cdot n + (a \textbf{ mod } b)$ és $(a \textbf{ mod } b)$ előjele megegyezik b előjelével, valamint a **rem**, amelyre $a = (a/b) \cdot b + (a \textbf{ rem } b)$ és $(a \textbf{ rem } b)$ előjele megegyezik a előjelével.
- Az abszolútérték (**abs**), valamint a nemnegatív egész kitevős hatványozás (**).

Egyéb nyelvek röviden

C és C++ • A C és C++ nyelvekben előjeles egész típusok a 8 bites **char**, a 16 bites **short**, a 32 bites **long** és az **int**, amelynek mérete megvalósításfüggő, de nem rövidebb, mint a **short** és nem hosszabb, mint a **long**. A 64 bites számítógéparchitektúrák megjelenésével ez a sor még kiegészült a 64 bites egész típussal is, amelynek nincs szabványos neve, de szokták például **long long**-nak nevezni. Ezen felül az **unsigned** módosítóval mindegyik típusból képezhető a megfelelő méretű előjel nélküli típus. A műveletek a szokásosak, a túlcsordulást nem ellenőrzik.

Java • Mivel a Java programok egy szabványban rögzített virtuális gépen futnak, így a Java nyelv egészen pontosan meghatározza az egyes egész típusok számábrázolási pontosságát. A **byte**, **short**, **int**, **long** típusok mellett, amelyek rendre 8, 16, 32 és 64 bitesek, a nyelv rendelkezik még a **char**, előjel nélküli 16 bites egész típussal is. A műveletek a szokásosak, túlcsordulás-ellenőrzés itt sincs.

Eiffel • Érdemes megemlíteni még az Eiffel programozási nyelvet, amely meglehetősen ritka módon az elemi típusokat, így az egészeket is beillesztette a nyelv osztályhierarchiájába. Az egész szám típust megvalósító *INTEGER* osztály egy kiterjesztett osztály, azaz az objektumait nem referenciával, hanem érték szerint tárolják az ilyen típusú változók, így teremtve meg a hatékonysághoz szükséges kompromisszumot. Az *INTEGER* osztály egyébként a *COMPARABLE* és a *NUMERIC* osztályok leszármazottja, az előbbitől örökli a rendezéshez szükséges összehasonlító műveleteit, az utóbbitól pedig a szokásos aritmetikai műveleteket. A bitműveletek helyett az egész számokon konverziós műveletek értelmezettek, amelyek segítségével egy számból bitsorozatot lehet készíteni és viszont. Ez a megoldás részben feloldja a bevezetőben említett, a hatékonyság és az ábrázolás elrejtése között fennálló ellentétet.

[5] A nevek jelentése rendre: nagyon rövid egész, rövid egész, hosszú egész, nagyon hosszú egész.

Python • A Python nyelv kétféle egész típust tartalmaz beépítetten. A `plain integer` típus a gépi szó méretben, de legalább 32 biten ábrázolható előjeles egész szám típus, a szokásos műveletekkel. Érdekesebb a nyelvben megvalósított másik egész típus, a `long integer`, amely viszont látszólag végtelen, az ábrázolható számnak csak a memória mérete szab határt. Ilyen fajta nagy számok más nyelvekben legfeljebb szabványos könyvtári megvalósításban – például a Java *BigNumber* osztálya – szokott előfordulni, a Python ezt a nyelvbe beépített típusként támogatja.

PL/I • A PL/I nyelvben szintén kétféle egész típust használhatunk, ezek azonban nem az ábrázolható számtartományban térnek el, hanem a számábrázolás módjában. A `BINARY` típusok binárisan, kettes komplemens reprezentációban tárolódnak. A `DECIMAL` típusok viszont BCD – *Binary Coded Decimal* azaz *Binárisan Kódolt Decimális* – számábrázolást használnak. A BCD számábrázolásnál az ábrázolt számod tízes számrendszerbeli számjegyeit tizenhatos számrendszerbeli számjegyekként tároljuk (például a 12345 BCD alakja $0x012345$). Ez a számábrázolás gazdaságtalanabb, mint a bináris, hiszen 4 bitet (16 lehetséges értéket) használ egyetlen decimális számjegy (azaz 10 lehetséges érték) ábrázolására. Előnye viszont, hogy alkalmazható valós számok ábrázolására is és mentes a kettes és tizes számrendszerek közötti átváltásból adódó pontatlanságoktól.

5.4.3. Csellengők

Van két típus, amelyek kicsit kilógnak a fenti osztályozásból. Mindketten egyértelműen a diszkrét típusosztály elemei, de hogy felsorolási típusként vagy egész típusként tekintünk-e rájuk, az programozási nyelvről programozási nyelvre változik. Ez a két csellengő a *logikai* és a *karakter* típusok. Vizsgáljuk meg őket egy kicsit részletesebben!

Karakter típusok

Karakter típus nem is biztos, hogy csak egy van a nyelvben. A modern programozási nyelvek legtöbbje már beépítetten támogat valamilyen 16 bites karakterkódolást – például a Unicode-ot (lásd a 2.1.4. fejezetet) –, ugyanakkor, a korábbi rendszerekkel való kompatibilitás végett támogatják a tradicionális 8 bites karakterkódolást is. Ezek a nyelvek tehát kétféle karakter típust is tartalmaznak.

A C és C++ nyelvekben a karakter típust megvalósító **char** az egész típusok közé tartozik, 8 bites, előjeles. A típus műveletei a nyelvben az egész típusokon szokásosak. A karakterek kódolását a nyelv nem határozza meg, ez megvalósításfüggő.

A Java nyelv még mindig egész típusként tekint a karakterekre, de a Java **char** típusa 16 bites előjel nélküli, és a nyelv meghatározza, hogy a típusértékek a Unicode kódolás szerint képeződnek le karakterekre. A **char** típus műveletei ugyanazok, mint a többi egész típusnál, de a nyelv szabványos osztálykönyvtárai még számtalan hasznos műveletet tartalmaznak a karakterek kezelésére.

A Pascal és Ada nyelvekben a karakter típus felsorolási típus. Műveletei megegyeznek a felsorolási típusoknál ismertetettekkel. Az Ada kétféle karakter típust is támogat. A 256 elemű *Character* típus értékei az ISO 10646 Basic Multilingual Plane (BMP) Row 00

256 kódpozíciójának (Latin-1), a *Wide_Character* típus értékei pedig az ISO 10646 Basic Multilingual Plane (BMP) 65536 kódpozíciójának felelnek meg.

Logikai típusok

A másik csellengő a logikai típus. Sok nyelvben nem is létezik logikai típus – ilyen például a C –, hanem valamilyen más típussal, leggyakrabban valamilyen egész típussal fejezik ki a logikai értékeket. Ilyen nyelvekben általában a 0 szám jelenti a hamis értéket, a 0-tól különböző tetszőleges érték pedig az igazat.

A Delphi nyelvben például több logikai típus is létezik, ám ezek mindegyike valójában egy egész típus. A *Boolean* típus egybájtos, megengedett értékei a 0 és az 1, amelyek rendre a hamis és igaz értékek megfelelői. Ezen felül a nyelv definiálja még a *Bytebool*, *Wordbool*, *Longbool* típusokat, amelyek rendre 8, 16 és 32 bites egész típusok, amelyeknél a nulla jelenti a hamis, a nem nulla pedig az igaz értéket.

Más nyelvekben a logikai típus felsorolási típus. Elemei a hamis (*false*) és az igaz (*true*). Ha a típus rendezett, akkor általában az igaz érték a nagyobb. Ilyen nyelvek a Pascal, az Ada, a CLU vagy az Eiffel.

Akár felsorolási, akár egész típus a logikai, műveleteinek halmaza bővebb, mint a típusosztály egyéb tagjaié. A logikai típuson minden nyelvben értelmezettek bizonyos logikai műveletek. A jellemző műveletek a *logikai és* ($\wedge$), a *logikai vagy* ($\vee$), a *logikai kizáró vagy* ($\oplus$) és a *negáció* ($\neg$). Ezek az operátorok nem tévesztendők össze a megfelelő bitaritmetikai műveletekkel, különösen azokban a nyelvekben nem, ahol a logikai típus az egész típusok közé tartozik, netán nincs is külön logikai típus. Ekkor ugyanis egyazon típusú objektumokra alkalmazható mindkét változat, de a két művelet eredménye gyakorta nem esik egybe.

A logikai kifejezések kiértékelése igen gyakran nem igényli mindkét operandus kiértékelését. Az $A \wedge B$ kifejezés értéke például B értékétől függetlenül hamis lesz, ha az A értéke hamis. A legtöbb programozási nyelv, hatékonysági okokból, ilyen esetben nem értékeli ki B-t. Ez az esetek jelentős részében megfelel az igényeinknek, sőt gyakran ki is használjuk a nyelvek ezen tulajdonságát, de ha A és B kiértékelése valamilyen mellékhatással jár – például naplózzuk, hogy kiértékeltük B-t –, akkor kívánatos lehet, hogy A értékétől függetlenül a kiértékelés megtörténjen. Ezért néhány programozási nyelvben a logikai és, illetve a logikai vagy operátorok két változatban léteznek. Az egyik változat csak akkor értékeli ki a második operandusát, ha az első alapján a kifejezés értéke nem határozható meg – ezeket nevezzük *lusta* operátoroknak. A másik változat mindenképen kiértékeli mindkét operandust – ezek a *mohó* operátorok. A lusta operátorokat az Adában és az Eiffelben **and then**-nek, illetve **or else**-nek, míg a CLU-ban `cand`-nek valamint `cor`-nak nevezik. A teljes kiértékelést végző változatok nevei Adában és Eiffelben **and** és **or**, a CLU-ban pedig & és |.

Az Eiffel ezen felül még egy logikai operátorral rendelkezik, ez az **implies**, amely a logikai implikáció ($\rightarrow$) operátora. Az $A \rightarrow B$ kifejezés akkor és csak akkor hamis, ha A igaz és B hamis.

5.5. Valós típusosztály

A valós típusosztály a valós számok különböző megvalósításait tartalmazza. Mivel a valós számok számossága meghaladja az egészekét, nem lehet olyan egyszerű megvalósítást találni a valós szám típusnak, mint azt az egészek esetében láthattuk, mivel a típusértékhalmaz véges, vagy legfeljebb megszámlálható számosságú lehet. A valós típusosztályba többféle valós típus is beletartozik, ezek azonban gyakorlatilag csak a számábrázolásban térnek el egymástól, így a valós típusokat egyben tárgyaljuk és csak a különbségekre mutatunk rá.

5.5.1. Típusértékhalmaz

Az idők folyamán többféle számábrázolási módot is kidolgoztak a valós számok számítógépen történő ábrázolására, amelyek közül kettő terjedt el szélesebb körben. A két ábrázolási módnak megvannak a maga előnyei és hátrányai, így egyik megoldás sem tekinthető kizárólagosnak.

Fixpontos számábrázolás

Az egyik legegyszerűbb leképezés a valós számokról az egészekre a kerekítés. Ez szülte a *fixpontos* számábrázolást, amikor a számokat rögzített hosszúságon ábrázoltuk, amelynek egy meghatározott része az egészrészt, a maradék pedig a törtrészt tartalmazta. Ennek a számábrázolási módnak az egyszerűségén túl van még egy nagy előnye, jól meghatározott a pontossága. Ha például pénzügyi rendszereket készítünk, ahol mondjuk a törvényi előírások két tizedesjegy pontosságú könyvelést írnak elő, akkor ez azonnal leképezhető a fixpontos számábrázolásra.

Van azonban egy hátránya is ennek a megoldásnak. Nem tudunk egyszerre ábrázolni nagyon nagy és nagyon kicsi számokat. Ha ugyanis sok helyet hagyok a számok törtrészének, akkor kevés jut az egészrésznek és fordítva. Másrészt a számítások eredményének egy jól meghatározott, a számábrázolás pontosságától függő méretű intervallumba kell esnie. Márpedig ezt az intervallumot igen gyakran nem tudjuk előre meghatározni. Képzeljük el például, hogy az $x \rightarrow 1/x$ leképezést próbáljuk kiszámolni. Ha a számábrázolásunk elég nagy pontosságú, akkor nem tudunk nagyobb x értékeket ábrázolni. Ha tudunk nagy számokat ábrázolni, akkor az $1/x$ értékeket nem tudjuk kellő pontossággal tárolni, így azok értéke igen hamar 0 lesz.

Ezek a megkötések a tudományos és mérnöki alkalmazást gyakorlatilag lehetetlenné teszik, így az ábrázolás használhatósága eléggé beszűkül. Ezen felül a típus megvalósítása elég egyszerű, szükség esetén a programozó is könnyen elkészítheti, ezért az általános célú programozási nyelvek többsége beépítetten nem támogatja.

Lebegőpontos számábrázolás

A fixpontos ábrázolás problémáinak feloldására dolgozták ki a *lebegőpontos* számábrázolást. Ebben az ábrázolásmódban a számokat *normál alakban*, azaz $m \cdot 10^k$ alakban tároljuk, ahol $0 \leq |m| < 10$ és k egész szám. Az m értéket a szám *mantisszájának*, a k-t pedig a szám *karakterisztikájának* nevezzük. A számítógépek bináris világában ez a számábrázolás $m \cdot 2^k$ alakra módosul, ahol $0 \leq |m| < 2$.

A legtöbb programozási nyelv a lebegőpontos számábrázolást támogatja. A ma használatos legtöbb processzor – legalábbis az általános célú processzorok nagy része – beépítetten tartalmazza a lebegőpontos ábrázolású valós szám típus megvalósítását. 1985-ben fogadták el az IEEE 754-es szabványát (*IEEE Standard for Binary Floating-point Arithmetic* – IEEE szabvány a bináris lebegőpontos aritmetikáról), amely a korábbi különböző lebegőpontos típusmegvalósításokat egységesítette. Ez a szabvány nem csupán a számábrázolás normalizált alakját határozza meg, de szabályozza a műveletek, átalakítások hatásait is. Kétféle számábrázolást definiál, az *egyszeres* és a *dupla* pontosságút.

Egyszeres pontosság • Az egyszeres pontosságú lebegőpontos számok 32 biten ábrázolódnak. Az első bit az előjel (s), utána 8 bit a karakterisztika (k) és 23 bit a mantissza (m) ábrázolására szolgál. Néhány bitsorozathoz a szabvány egyedi jelentést rendel.

- Ha $k = 255$ és $m \neq 0$, akkor az érték *nem szám* (NaN - not a number).
  Például: 0 11111111 01000100010001000100010 $\rightarrow NaN$.
- Ha $s = 1$, $k = 255$ és $m = 0$, akkor az érték *negatív végtelen*.
  Azaz: 1 11111111 00000000000000000000000 $\rightarrow -\infty$.
- Ha $s = 0$, $k = 255$ és $m = 0$, akkor az érték *pozitív végtelen*.
  Azaz: 0 11111111 00000000000000000000000 $\rightarrow +\infty$.
- Ha $0 < k < 255$, akkor a szám értéke $(-1)^s \cdot 2^{k-127} \cdot (1.m)$, ahol $1.m$ azt a kettes számrendszerbeli számot jelenti, amelynek az egészrésze 1, a törtrésze pedig m.
  Például: 0 00000000 10000000000000000000000 $\rightarrow 2$
  vagy 1 10000001 10100000000000000000000 $\rightarrow -6.5$.
- Ha $k = 0$ és $m \neq 0$, akkor az ábrázolt érték a $(-1)^s \cdot s^{-126} \cdot (0.m)$. Ezek a *nem normalizált* értékek. Például: 0 00000000 10000000000000000000000 $\rightarrow 2^{-127}$
  vagy 0 00000000 00000000000000000000001 $\rightarrow 2^{-149}$, ami a legkisebb ábrázolható pozitív érték.
- Ha $k = 0$ és $m = 0$, akkor az ábrázolt szám – s értékétől függően – $+0$ vagy -0.
  Azaz 0 00000000 00000000000000000000000 $\rightarrow +0$
  és 1 00000000 00000000000000000000000 $\rightarrow -0$.

Dupla pontosság • A dupla pontosságú lebegőpontos számok képzési szabálya hasonló. Itt a számábrázolás 64 biten történik. Ebből 1 bit az előjel, 11 bit a karakterisztika és a fennmaradó 52 bit a mantissza.

5.5.2. Műveletek

A valós típusoknál a műveletek halmaza sokkal konzervatívabb, mint az egészeknél. A négy matematikai alapművelet ($+, -, *, /$) mellett a hatványozás, négyzetgyök-vonás és különböző konverziós műveletek szoktak előfordulni. Ez a művelethalmaz általában az, amit a processzorok a beléjük épített valós szám típusokhoz megvalósítanak, így ezek leképezése hatékony.

Ezen felül szabványos könyvtári elemként a legtöbb programozási nyelv megvalósítja a természetes alapú logaritmus függvényt, a természetes alapú exponenciális függvényt, a trigonometrikus függvényeket, és esetleg egy egyszerű véletlenszám-generátort.

5.5.3. Programozási nyelvek

Vizsgáljunk meg néhány programozási nyelvet, milyen valós típusokat valósítanak meg.

Pascal

A Pascal nyelv lebegőpontos számábrázolást használ. Az alap valós típus a *Real*, amely 48 bites. Ebből 1 bit az előjel, 8 bit a karakterisztika és 39 bit a mantissza. Ezt, megvalósítástól függően, további valós típusok egészíthetik ki. A Turbo Pascal például még további három valós típust tartalmaz. A *Single* és a *Double* a szabványos IEEE egyszeres és dupla pontosságú számábrázolásnak felelnek meg, míg az *Extended* típus 80 bites, 15 bites karakterisztikával és 64 bites mantisszával.

A Pascal megengedi, hogy a valós típusokon értelmezett műveletek egyes operandusai egészek legyenek, de a műveletek eredménye valós lesz. A valós típusok műveletei:

- Matematikai alapműveletek: összeadás (+), kivonás (−), szorzás (*), osztás (/).
- Abszolútérték (*Abs*) és egészrész (*Int*) – az eredmény valós típusú.
- Négyzetre emelés (*Sqr*) és négyzetgyökvonás (*Sqrt*).
- Természetes alapú exponenciális függvény (*Exp*) és logaritmus (*Ln*).
- Trigonometrikus függvények: szinusz (*Sin*), koszinusz (*Cos*) stb.
- Véletlenszám-generátor (*Random*).
- Konverziós operátorok: csonkolás (*Trunc*), és kerekítés (*Round*).

Az Ada valós típusai

Az Ada támogatja mind a lebegő-, mind a fixpontos számábrázolást. Az egészekhez hasonlóan tetszőleges számú valós típust hozhatunk létre. A típusdeklarációban meg kell adnunk a típus ábrázolási pontosságát. A nyelvben minden valós típust úgy tekintenek, mint egy előre definiált *root_real* típusból származtatott típust. A valós literálok ennek az osztályába tartoznak, így *universal_real* típusúak. Ez teszi lehetővé, hogy ugyanazokat a literálokat használjuk mind a lebegő-, mind a fixpontos típusoknál.

Fixpontos típus • A fixpontos típusoknál a típus abszolút pontosságát kell megadni. Ha k tizedesjegy pontossággal akarunk számokat ábrázolni, akkor a 10^{-k} számot a típus *deltájának* nevezzük. Mivel a számábrázolás kettes számrendszerben történik, a típus értékei nem $n \cdot 10^{-k}$, hanem $m \cdot 2^{-p}$ alakúak lesznek, ahol p a legkisebb olyan pozitív egész szám, amelyre $2^{-p} \leq 10^{-k}$. A 2^{-p}-t a típus *small* értékének nevezzük, és a valóságban ez a legkisebb ábrázolható érték. Ez az alábbi példában problémát is okoz, hiszen a kerekítési pontatlanságok miatt Y értéke eltér a várttól.

```
type Fixpontos is delta 0.01 range 0.00 .. 1.00;

X: Fixpontos := 0.05;        -- X valójában 0.046
Y: Fixpontos := 5.0 * X;     -- Y = 0.23
```

Lebegőpontos típus • A lebegőpontos típusoknál a típus relatív pontosságát, azaz a mantissza ábrázolásához szükséges *decimális jegyek* számát kell meghatároznunk. Egy d decimális jegyig pontos lebegőpontos típushoz legalább $b = \lceil d \cdot \ln(10)/\ln(2) + 1 \rceil$ bit hosszú mantissza szükséges a bináris ábrázolásban. Ilyenkor a karakterisztika a $-4 * b \ldots 4 * b$ intervallumból kerül ki. A nyelvbe beépített *Float* típus ábrázolási pontossága implementációfüggő, de ha az adott megvalósítás képes rá, akkor legalább 6 decimális jegyet támogatnia kell. Az egész típushoz hasonlóan az egyes megvalósítások támogathatnak további beépített lebegőpontos típusokat (*Short_Short_Float*, *Short_Float*, *Long_Float*, *Long_Long_Float*) is, ám ezek ábrázolási pontosságára csak annyi a megkötés, hogy a típusnevek angol nyelvű jelentésével összhangban kell lennie.

A műveletek halmaza nagyrészt megegyezik a lebegőpontos és fixpontos típusoknál, az eltérés csupán néhány attribútum, amelyekkel a típusok számábrázolási pontosságát lehet lekérdezni. Ezeken túlmenően a megengedett műveletek a négy alapművelet (+, −, *, /), az egész kitevős hatványozás (**) és az abszolútérték függvény (**abs**).

A lebegőpontos típus új attribútuma az *S'Digits*, amely megadja, hány decimális jegyig pontos az adott típus.

A fixpontos típus négy új attribútummal rendelkezik. Az *S'Delta* és az *S'Small* rendre a típus delta és small értékei, az *S'Aft* az ábrázolható tizedesjegyek száma, az *S'Fore* pedig a tizedesvessző előtti karakterek száma kiíratáskor, beleértve az esetleges előjelet is.

5.6. Mutató típusok

A mutató, illetve referencia típusok a memóriacímek absztrakciói. A két fogalom – mutató és referencia – között nincs egyértelmű különbség meghatározva. Általánosságban azt lehet mondani, hogy a referenciák a mutatók biztonságosabb változatai. A biztonságosság általában azt jelenti, hogy a referencia mindig létező objektumra mutat, vagy ha ez nem biztosítható, akkor legalább van egy kitüntetett, sehova sem mutató érték, és a referencia vagy egy létező objektumra mutat, vagy ezt a kitüntetett értéket tartalmazza. Ezen túlmenően a mutatók és referenciák legtöbb tulajdonságukban nagyon hasonlóan viselkednek, így együtt fogjuk tárgyalni őket, és csak a különbségekre mutatunk rá.

A mutatók kialakulásának okát a gépi nyelveknél megjelenő közvetett (indirekt) címzésben kereshetjük. Segítségükkel az objektumokat egy magasabb *hivatkozási szinten* használhatjuk. A 42-es szám hivatkozási szintje 0, a 42-t tartalmazó változó hivatkozási szintje 1, a változóra hivatkozó mutató hivatkozási szintje pedig 2 stb. Ha egy kifejezésben egy változót használunk, akkor a fordító (compiler) automatikusan csökkenti a hivatkozási szintet, így például az *a* := *b* + *1* azt jelenti, hogy *a* értéke 1-gyel nagyobb, mint a *b*-ben tárolt érték.

A mutatók használata a programozásban igen sokrétű. Gyakran akkor is mutatókkal dolgozunk, ha nem is tudunk róla. Például a nagyméretű, összetett adatszerkezetek paraméterként történő átadásakor gyakran előfordul, hogy a fordítóprogram nem az egész adatszerkezet lemásolása mellett dönt, csupán egy arra vonatkozó referenciát ad át. Hasonlóan az objektumorientált programozási nyelvekben a többalakúság (polimorfizmus) és a dinamikus összekapcsolás (lásd a 12. fejezet) megvalósításához az objektumokat referenciákon

keresztül kell használnunk. És természetesen igen gyakran tudatosan használunk mutatókat, dinamikus adatszerkezetek építéséhez, vagy egyszerűen csak azért, hogy egy objektumra egynél több helyről hivatkozhassunk.

5.6.1. Memóriagazdálkodás

Mielőtt a mutató típusok részletes tárgyalásának nekikezdenénk, érdemes néhány szót ejtenünk a programok memóriagazdálkodásáról. Egy program változóit az élettartamuk szerint három csoportba oszthatjuk:

- A *statikus változók* a változódeklaráció első kiértékelődésekor létrejönnek és utána a program futásának befejeződéséig megmaradnak. Nem minden programozási nyelv teszi lehetővé statikus változók létrehozását. Statikus változók jellemzően a program globális változói, de némely nyelvekben – ilyen például a C vagy a C++ – az alprogramoknak is lehet statikus változójuk, amelyek megőrzik értéküket a hívások között.
- Az *automatikus változók* a program futása során automatikusan létrejönnek és megszűnnek. Ezek a blokkstrukturált nyelvekben a blokkokban deklarált változók. Valahányszor a programvégrehajtás belép egy blokkba, a blokk deklarációs része kiértékelődik és az abban szereplő automatikus változók létrejönnek. Amint a blokk végrehajtása befejeződött, ezek a változók megszűnnek, így a blokkba történő két belépés között az értéküket nem őrzik meg. A legtöbb modern programozási nyelv megengedi, hogy egy alprogram meghívja önmagát (rekurzió), így elképzelhető, hogy egy automatikus változónak több, egymástól független példánya is létezik egyidőben.
- A *dinamikus változók* élettartamát a programozó maga kezeli – legalább részben – a program futása alatt. A dinamikus változókat egy mutató vagy referencia segítségével érhetjük el. Míg az előző két típusnál a változók számára a memória lefoglalásáról a fordítóprogram gondoskodik, addig a dinamikus változókat a programozó foglalja le valamilyen *allokátor* segítségével. Az ilyen változók megszüntetése, legalább részben, szintén a programozó feladata, erről a későbbiekben még részletesebben szólunk.

Ennek az éltettartambeli hármas tagozódásnak megfelelően a program által felhasznált memória is három részre oszlik:

- A *statikus memóriaterület* tartalmazza a statikus változókat. Ennek a mérete általában fordítási időben meghatározható, az itt helyet foglaló változók címe gyakorlatilag nem változik a program végrehajtása során.
- A *végrehajtási verem* az automatikus változók tárolásának helye. Amikor a programvégrehajtás belép egy blokkba, a végrehajtási verem tetején foglalódik le hely a blokk automatikus változóinak. Ez a megoldás teszi lehetővé, hogy rekurzió esetén egyazon változó több példányban létezzen. A blokk végrehajtásának befejezésekor a blokk automatikus változói törlődnek a verem tetejéről, és a végrehajtás a befoglaló blokkban folytatódik tovább.

- A harmadik memóriaterület, amelyet a dinamikus változók foglalnak el, a *dinamikus tárterület* (angolul *heap*). A program a végrehajtása során a itt tárolja a szabad memóriablokkokat, egy úgynevezett *szabad listában*. Amikor a programozó egy új változót próbál meg lefoglalni, az allokátor kiválaszt a szabad listából egy olyan memóriablokkot, amelynek mérete elegendő a létrehozandó változó tárolásához, és visszaadja a blokk memóriabeli kezdőcímét. Amikor az adott változó megszűnik, az általa elfoglalt memóriablokk visszakerül a szabad listába.

Amint azt említettük, a dinamikus változók élettartamát a programozó szabályozza. A változó létrehozásához egy allokátor meghívására van szükség, amely lefoglalja a szükséges memóriát, sőt bizonyos nyelvekben elvégzi az adott memóriaterület inicializációját is. A dinamikus változók megszüntetéséhez azonban nem mindig állnak rendelkezésre ennyire explicit eszközök. A dinamikus változók megszüntetése szempontjából a programozási nyelvek két csoportra oszthatók.

Explicit deallokátort használó nyelvek • Az első csoportba tartoznak azok a nyelvek, amelyek a változók megszüntetését is teljes egészében a programozóra bízzák. Ezekben a nyelvekben egy dinamikus változó megszüntetéséhez a programozónak egy *deallokátor* műveletet kell meghívnia, amely felszabadítja a változó által lefoglalt memóriablokkot, visszahelyezve azt a szabad listába. Bizonyos nyelvekben a deallokátor a memória felszabadítását megelőzően még felszámolja az ott tárolt változót is, például C++-ban végrehatja az adott osztály destruktorát.

Ezekben a nyelvekben a programozó közelről képes szabályozni a program memóriahasználatát, ám ez a szabadság nagy felelősséggel is jár. A programozónak gondoskodnia kell a lefoglalt memóriablokkok felszabadításáról, mivel ellenkező esetben a program feleméstheti a rendelkezésre álló memóriát. Másrészt a programozónak gondoskodnia kell arról is, hogy egy felszabadított memóriaterületre ne legyen több hivatkozás, mivel azok futási idejű hibákat eredményeznek.

Szemétgyűjtést támogató nyelvek • Más nyelvekben a memóriablokkok felszabadítása nem a programozó, hanem a program végrehajtási környezetének feladata. Az ilyen programozási nyelvekben a lefoglalt memóriablokkokhoz tartozik egy úgynevezett *hivatkozásszámláló*. Ez a számláló nyilvántartja, hogy egy adott időpillanatban hány referencia mutat az adott memóriaterületre. Amennyiben ez az érték nullára esik, az adott memóriablokk felszabadíthatóvá válik, hiszen több hivatkozás már nem lehetséges rá. Az ilyen blokkok felszabadítását a végrehajtási környezet *szemétgyűjtő mechanizmus (garbage collector)* végzi. A szemétgyűjtő mechanizmus általában meghatározott időközönként fut le, bár bizonyos programozási nyelvekben lehetőség van az explicit elindítására is.

A szemétgyűjtést használó nyelvekben a mutatók – referenciák – használata biztonságosabb, hiszen nem fordulhat elő, hogy a programozó elfelejt felszabadítani egy memóriaterületet, vagy netán felszabadított területre hivatkozik, viszont a programozónak sokkal kevesebb lehetősége van szabályozni a program memóriahasználatát. A másik hátránya ezen nyelveknek – noha az újabbnál újabb szemétgyűjtő algoritmusok egyre kifinomultabban működnek –, hogy a szemétgyűjtés periodikus terhelésnövekedést jelent a számítógépen, ami kellemetlen hatással lehet a programok valósidejű teljesítményére. A szemétgyűjtés igen

költséges feladat, az algoritmusnak ugyanis nem elegendő azokat a blokkokat felszabadítani, amelyek hivatkozásszámlálója 0, hiszen a blokkok között lehetnek körkörös hivatkozások, amelyek „kívülről", azaz a programból már nem érhetők el.

Amint az látható, a memóriagazdálkodás alapvetően befolyásolja a nyelvben használható mutató típusok tulajdonságait, megengedett műveleteit. A memóriakezelésről további részletek a 4. fejezetben találhatók.

5.6.2. Típusértékhalmaz

A mutató olyan objektum, amely egy másik objektum memóriabeli címét tartalmazza. A tényleges ábrázolás megvalósításfüggő, ám igen gyakran lényegében egy előjel nélküli egész szám, amely a hivatkozott objektum első bájtjának memóriabeli sorszámát (*címét*) tartalmazza. Referenciák használata esetén az ábrázolás ettől eltérhet, mivel a referenciák használatának egyik oka lehet a gazdaságosabb memóriahasználat, melynek során a végrehajtási környezet esetleg megváltoztathatja a hivatkozott objektumok tényleges memóriabeli címét, mindazonáltal a referencia ekkor is egy memóriabeli helyet jelöl.

Típus nélküli mutató

A mutató típusok az esetek nagy részében *típusosak*, azaz meghatározott, hogy milyen típusú objektumok címét tartalmazhatják. Ez a típusosság az ábrázolást általában már nem érinti, de lehetővé teszi a fordítóprogram számára, hogy fordítási időben ellenőrizze, hogy a hivatkozott objektumokat típushelyesen használjuk-e. Bizonyos programozási nyelvek lehetővé teszik a típus nélküli memóriacímek használatát is.

A típus nélküli mutatók használata erősen korlátozott. Mivel nem meghatározott, hogy milyen objektumra mutatnak, a hivatkozás-feloldás (dereferencia) művelete nem engedélyezett rájuk, viszont általában tetszőleges mutató típus automatikusan konvertálódik a típus nélküli mutató típusra és a fordított irányú konverzió is lehetséges, legalábbis típuskényszerítés használata esetén.

A típus nélküli mutatókat általában olyan esetben használjuk, amikor egy tetszőleges típusú objektumot szeretnénk átmenetileg tárolni. Mivel a típus nélküli mutatón keresztül az objektummal semmilyen műveletet nem végezhetünk, tényleg csak az objektumra vonatkozó referencia tárolása jön szóba. Az alábbi C nyelvű példa egy olyan lista típus megvalósítása, amelyben tetszőleges objektumokat lehet elhelyezni.

```
. . .
struct __listaelem_struct;
typedef struct __listaelem_struct* Lista;

struct __listaelem_struct {
    Lista  kovetkezo;
    void*  elem;
};
```

```
Lista beszur(Lista l, void* elem)
{
    Lista p = malloc(sizeof(struct __listaelem_struct));
    if (p != NULL) {
        p->kovetkezo    = l;
        p->elem         = elem;
    }
    return p;
}
...
Lista l = NULL;
int   i = 5;
char  text[] = "Helló világ!";

l = beszur(l, &i);
l = beszur(l, &text);
```

Amint látható, a listába tetszőleges objektumok helyezhetők, ám ahhoz, hogy az elemeket újra fel tudjuk használni, tudnunk kell, melyik objektum milyen típusú. Mivel a C-ben a **void*** mutató tetszőleges típusra hivatkozó mutatóvá konvertálódik, a programozónak magának kell gondoskodnia – valamilyen nyelven kívüli eszközzel – arról, hogy a listából kivett elemeket a tényleges típusuknak megfelelően használja.

Azokban az objektumorientált nyelvekben, ahol a nyelv definiál egy közös ősosztályt – például a Java vagy az Eiffel nyelvben –, a közös ősosztályra mutató referencia típus hasonló módon használható, mint a típus nélküli mutató. Mivel azonban az objektumok hordoznak információt a tényleges típusukról, van eszköz a nyelvben arra, hogy meggyőződjünk a végrehajtani kívánt konverzió helyességéről. A Java nyelv szabványos osztálykönyvtáraiban megvalósított tároló adatszerkezetek éppen ezen az elven működnek.

Sehova sem hivatkozó mutató

Minden programozási nyelvben van a típusértékhalmaznak egy kitüntetett eleme, a *sehova sem hivatkozó mutató*. Ez a mutatót az egyes nyelvek különböző neveken nevezik. A C-ben *NULL*, a C++-ban *0*, az Adában és Javában **null**, a CLU-ban `nil`, az Eiffelben *void* stb. Közös tulajdonságuk, hogy automatikusan konvertálódik minden mutató típusra, így bármely mutatónak értékül adhatók. Ez az érték szinte minden esetben az azonosan nulla bitsorozattal ábrázolódik, ami tehát nem egy érvényes memóriacím, egyetlen rendszerben sem. Természetesen a sehova sem hivatkozó mutatón keresztüli hivatkozás futási idejű hibát eredményez.

A programozási nyelvek jelentős részében, különösen amelyek referencia típust használnak – mint például az Ada, a Java, az Eiffel, a CLU stb. – a mutatók kezdőértéke ez a sehova sem hivatkozó mutató. Ennek az az előnye, hogy így az inicializálatlan mutatók jól megkülönböztethetők az inicializáltaktól. A C vagy C++ nyelvekben erről a programozónak kell gondoskodnia.

Azokban a nyelvekben, ahol szemétgyűjtéses memóriagazdálkodás van, és a mutatók kezdőértéke a sehova sem hivatkozó mutató, teljesül az a tulajdonság, hogy egy mutató vagy sehova sem mutat, vagy egy érvényes memóriaterületre hivatkozik, ami nagyban megkönnyíti a biztonságos programok írását.

Előre deklarálás

Azt a tényt, hogy a típusos mutató ábrázolása nem függ a hivatkozott objektumok típusától, igen gyakran kihasználjuk, például a láncolt adatszerkezetekben a „tyúk vagy a tojás" probléma feloldására. Az alábbi Ada nyelvű példában egy egész számokból álló egyirányú láncolt lista ábrázolását adjuk meg. A *ListaElem* típus egy egész számot, és egy ilyen típusra hivatkozó mutatót tartalmaz, amely a lista következő elemére mutat. A típus megadásakor szembesülünk azzal a problémával, hogy az Ada nyelvben mutató típus csak egy létező típushoz definiálható és egy rekordnak csak létező típusok lehetnek összetevői. Az ellentmondás feloldására a *ListaElem* típus előre deklarálása használható. Az előre deklaráláskor tudatjuk, a fordítóval, hogy létezik egy *ListaElem* nevű típus. Mivel a mutató típus ábrázolása nem függ a mutatott típustól, a létezés ténye elegendő ahhoz, hogy megadhassuk a *ListaElemMutató* típust, amely immár egy teljesen meghatározott típus, így felhasználható a *ListaElem* típus definíciójában komponensként.

```
-- Előre deklaráljuk a ListaElem típust.
type ListaElem;

-- Így már definiálható a rá hivatkozó mutató típus.
type ListaElemMutató is access all ListaElem;

-- És így már definiálhatjuk a ListaElem típust is.
type ListaElem is record
    Tartalom    : Integer;
    Következő   : ListaElemMutató;
end record;
```

5.6.3. Műveletek

Vizsgáljuk meg most, hogy milyen műveletek állnak rendelkezésre a mutató típusok kapcsán. A műveletek egy része – például az értékadás, vagy a memóriafoglalás – minden nyelvben megjelenik, de vannak nagyon egyedi, az adott nyelvre jellemző műveletek is, mint például a C és C++ nyelvek mutatóaritmetikája.

Értékadás

Az értékadás a mutatók között legtöbbször csak a hivatkozott memóriacím lemásolását jelenti. Ha azonban szemétgyűjtéses memóriagazdálkodást és referenciákat használunk – mint például az Ada vagy Java nyelvekben –, akkor az értékadás egyben az értéket kapó refe-

rencia által hivatkozott memóriaterület hivatkozásszámlálójának csökkentését és az újonnan hivatkozott terület számlálójának növelését is jelenti.

Egy mutató négyféle módon kaphat értéket:

- Értékül kaphatja a sehová sem hivatkozó mutatót. A nyelvek egy jelentős részében ez a mutatók kezdeti értéke.
- Értékül kaphatja egy másik mutató értékét.
- Értékül kaphatja egy allokátor által lefoglalt memóriablokk címét.
- Értékül kaphatja egy statikus vagy automatikus változó címét – amennyiben ezt a műveletet az adott nyelv támogatja.

Memóriafoglalás

A memóriafoglaló művelet (*allokátor*) minden dinamikus változókat támogató nyelvben megjelenik. Ez a művelet választja ki a megfelelő memóriablokkot a szabad listában tárolt szabad blokkok közül, de sok nyelvben – különösen az objektumorientált programozást támogató nyelvekben, mint például a Java vagy a C++ – az allokátor egyben inicializálja is a lefoglalt memóriaterületet, az ott tárolandó objektum típusának megfelelően.

Memória-felszabadítás

Memóriablokkok felszabadítására szolgáló (*deallokátor*) művelet nem minden nyelvben van. A szemétgyűjtést támogató nyelvek általában nem tartalmaznak explicit deallokátort, mivel használata a korábban említett veszélyekkel járhat. A hatékonyság ugyanakkor néha megköveteli, hogy a programozó a maga kezébe vegye a memória kezelését – például mert a szemétgyűjtő algoritmus lefutási gyakorisága túl kicsi és így két futás között a program memóriafoglalása túl nagyra nőhet. Ezt a problémát a programozási nyelvek többféle módon próbálják áthidalni.

Vannak nyelvek, amelyek megengedik, hogy a programozó explicit elindítsa a szemétgyűjtő mechanizmust. Ilyen például a Java, amelyben a *System.gc()* hívással kezdeményezhető a szemétgyűjtés.

Más nyelvek a szemétgyűjtés mellett lehetővé teszik a programozó általi deallokációt is, ilyenkor természetesen a programozó felelőssége a memóriakezelés konzisztenciájának fenntartása. Erre a megoldásra példa az Ada 95 nyelv, amelyben az *Ada.Unchecked_Deallocation* sablon megfelelő mutatótípusra történő példányosításával az adott típushoz létrehozható a deallokátor művelet.

Referenciaképzés

A referenciaképzés művelete egy automatikus vagy statikus változóra hivatkozó mutató előállítása. Vannak olyan nyelvek – ilyen például az Ada 83, az Eiffel, vagy a Java nyelvek –, amelyben ez a művelet nem megengedett. Ezekben a nyelvekben a mutatók csak dinamikus változókra mutathatnak. A nyelvek egy jelentős része azonban lehetővé teszi, hogy ezeket a változókat a dinamikusakhoz hasonlóan kezeljük.

A C és C++ nyelvekben a referenciaképzés operátora az &. Mivel ezek a nyelvek egyszerű mutatókat használnak, ez az operátor korlátozások nélkül használható.

Sokkal érdekesebb a helyzet az Ada 95 nyelvben, amely – enyhítve a 83-as változat szigorán – lehetővé tette a referenciaképzést az automatikus és statikus változók esetében. Mivel a nyelv alapvetően szemétgyűjtéses memóriagazdálkodást folytat, a biztonságos használat érdekében az így képzett referenciákat is be kellett illeszteni ebbe a rendszerbe.

A nehézséget az jelenti, hogy az automatikus változók élettartama szigorúan kötődik a blokkszerkezethez és a programvégrehajtás menetéhez. Amikor a végrehajtás belép egy blokkba, annak automatikus változói létrejönnek, amikor elhagyja, megszűnnek. Tegyük fel, hogy programunk két egymásba ágyazott blokkból áll. Az *A* blokk tartalmazza a *B* blokkot. Az *A* blokkban deklarálunk egy *IP* mutatót, a *B* blokkban pedig egy *I* automatikus változót. A *B* blokkon belül az *IP* mutatót beállítjuk az *I* változó címére. A *B* blokkot elhagyva az *I* változó megszűnik, ám az *IP* mutató, a korábbi biztonságos tulajdonságot megszegve, olyan értéket tartalmaz, amely nem **null**, de nem is érvényes memóriacím.

Ezt az ellentmondást az Ada 95 tervezői úgy oldották fel, hogy a programozónak meg kell jelölnie az **aliased** kulcsszóval azokat a változókat, amelyekre a referenciaképzés műveletét használni kívánja. Ezeket a változókat, még ha automatikusak is, a nyelv a dinamikus tárolási területen helyezi el és a blokkon belül egy referenciával helyettesíti. Más szavakkal az ilyen változókból észrevétlenül dinamikusat csinál, amelyre alapértelmezésben egyetlen referencia létezik, a tartalmazó blokk egy automatikus változóján keresztül. Ha a referenciaképzés műveletét nem használjuk, ez az egy hivatkozás a blokkot elhagyva megszűnik és a változó által lefoglalt terület felszabadul. Ha viszont képeztünk további hivatkozásokat a változóra, akkor a blokkot elhagyva a változó biztonságosan tovább élhet, hiszen a dinamikus tárterületen helyezkedik el. Az alábbi programrészlet erre mutat példát.

```
type EgészMutató is access all Integer;
IP : EgészMutató;

procedure B is
I : aliased Integer;
begin
   IP := I'Access;
end B;
```

Hivatkozás-feloldás

A mutató típusok egyik legfontosabb művelete a *hivatkozás-feloldás*, azaz a dereferencia. A hivatkozás-feloldás a mutató hivatkozási szintjének csökkentését jelenti, azaz egy objektum címéből az objektumot „állítja elő".

A programozási nyelvek többségében a hivatkozás-feloldást kifejezetten jelölni kell. Például a C-ben és a C++-ban a dereferencia operátor a prefix *. Ha *p* egy mutató, akkor **p* jelöli a mutatott objektumot. Ha *p* egy struktúrára, vagy C++ esetén egy osztályra mutat, amelynek *f* egy komponense, akkor a *(*p).f* mezőhivatkozás helyettesíthető a *p->f* alakkal, amely így szintén tartalmaz egy hivatkozás-feloldást.

Azokban a nyelvekben ahol az objektumokat referenciákon keresztül érjük el – például a Java nyelvben, vagy részben az Eiffel, illetve a CLU objektumainál – a dereferenciát nem kell jelölni, az egyes műveletekhez a nyelv meghatározza, hogy azok a referenciára, vagy a hivatkozott objektumra vonatkoznak-e. Ha például *a* és *b* két Java objektumreferencia, akkor az *a = b* értékadás, vagy az *a == b* egyenlőségvizsgálat a referenciákon értelmezett művelet. Ugyanakkor a metódushívás mindig a hivatkozott objektum művelete, így az *a.equals(b)* művelet már az objektumok összehasonlítását fogja végezni.

Az Ada nyelv itt a két véglet között áll: akkor kell jelölni a hivatkozás-feloldást, ha az nem egyértelmű. A dereferencia jelölésére a **.all** minősítőt használhatjuk, szükség esetén. Ha *I* egy egész típusú változó, *IP* és *JP* pedig két egészekre vonatkozó mutató, akkor az *I := IP*, *IP := I* értékadások, vagy éppen az *I = IP* egyenlőségvizsgálatok esetén a dereferencia jelölés nem szükséges, mivel itt egyértelmű, hogy a hivatkozott objektumról van szó, hiszen máshogy az adott műveletek nem értelmezettek, a dereferencia jelölése nem változtatná meg azok jelentését.

Ha viszont *I* helyett *JP*-t használunk, akkor a műveleteknek értelme lesz mindkét hivatkozási szinten – mind a referenciák, mind a hivatkozott objektumok szintjén – így az utóbbinál a dereferenciát jelölni kell. Ha tehát *IP := JP*-t írunk, akkor az a mutatók közötti értékadást jelenti, míg az *IP*.**all** := *JP*.**all** a mutatott objektumok közötti értékadás.

Egyenlőségvizsgálat

Szinte minden nyelvben van lehetőség a mutatók egyenlőségének vizsgálatára. Ez az egyenlőség mindig a mutató szintjén dől el, nincs köze a hivatkozott típuson esetleg értelmezett egyenlőség relációhoz. Két mutató akkor és csak akkor egyenlő, ha pontosan ugyanarra az objektumra mutatnak.

5.6.4. Alprogramra hivatkozó mutató

A mutatótípusok különleges változata az *alprogramra hivatkozó mutató*. A típusértékhalmaz ebben az esetben nem objektumok címeit, hanem alprogramok belépési pontjait tartalmazza, természetesen a sehova sem hivatkozó mutatót itt is beleértve.

Az alprogramra hivatkozó mutatók mindig típusosak, a mutató típusa lényegében a mutatott alprogram formális paramétereinek listája és függvények esetén a visszatérési érték típusa. A formális paraméterlistából csak a paraméterek típusa bír jelentőséggel.

Az alprogramok mutatóinak műveletei jóval korlátozottabbak, mint az objektumoké, mivel alprogramot dinamikusan létrehozni nem lehet. Az értékadás és egyenlőségvizsgálat az objektum mutatókkal megegyező módon történik. Ezen felül engedélyezett a referenciaképzés az alprogramokra, valamit a hivatkozás-feloldás. Ez utóbbi lényegében az alprogram meghívását jelenti.

A C – és így a C++ is – támogatja az alprogram mutatókat. Az alábbi példában egy határozott integrált számoló függvényt valósítottunk meg, amely paraméterként kapja az integrálandó függvényt. Figyeljük meg, hogy sem a referenciaképzést, sem a dereferenciát nem kellett kifejezetten jelölni, noha megengedett!

```
typedef double (*ValosFuggveny)(double);

double Integral(ValosFuggveny fv, double a, double b, int n)
{
    double x     = a;
    double delta = (b - a) / (double)(n);
    double s     = 0.0;

    if (b <= a) return 0.0;
    while ((x + delta) <= b) {
        s += delta * (fv(x) + fv(x + delta)) / 2.0;
        x += delta;
    }
    return s;
}
...
double sininteg = Integral(sin, 0.0, 1.0, 1000);
```

Az objektumorientált nyelvekben az alprogramra hivatkozó mutató típusok jelentősége erősen lecsökkent, hiszen a mutató típus kiváltható egy interfésszel, vagy absztrakt osztállyal, amelynek egyetlen metódusa az átadni kívánt függvény, a mutató helyett pedig átadható egy objektum, amely megvalósítja ezt az interfészt, vagy amely az adott absztrakt osztály leszármazottja. Az ilyen, igen gyakran adattag nélküli objektumokat szokás *függvényobjektumnak* is nevezni (lásd még a 12. fejezetet).

5.6.5. Nyelvi specialitások

Nézzünk meg most néhány egyedi nyelvi sajátságot a mutatókkal kapcsolatban!

C mutatóaritmetika

A C nyelv egyik legérdekesebb és talán legtöbbet használt jellemzője az egyedülálló mutatóaritmetikája. A mutatóaritmetika segítségével nagyon kényelmesen kezelhetjük mutatóinkat és általa a C közeli kapcsolatot teremt a tömbök és a mutatók között. A C-ben a mutatók lényegében előjel nélküli egész számokkal ábrázolódnak, értékük a hivatkozott objektum első bájtjának memóriabeli címe.

A C támogatja a típusos mutatókat, egy T típusra hivatkozó mutató típusa T*, de van benne típus nélküli mutató is, melynek neve **void***. A **void*** típus sok szempontból úgy viselkedik, mintha egy bájt típusra (**unsigned char**) hivatkozó mutató lenne, de például a dereferencia nem megengedett művelet rá.

A C mutatóaritmetikája a következő műveletekkel egészíti ki a mutató típusok szokásos eszköztárát:

- A **sizeof** operátor egy tetszőleges T típushoz megmondja, hogy az adott típus hány bájton tárolódik a memóriában.

- Minden mutató típus automatikusan konverálódik **void*** típusra.
- A **void*** mutatók automatikusan konvertálódnak tetszőleges típusra hivatkozó mutatóvá.
- A mutatók típuskényszerítéssel egész számmá alakíthatók.
- Az egész számok típuskényszerítéssel tetszőleges típusra hivatkozó mutatóvá alakíthatók.
- Egész számok és mutatók összeadása. Ha van egy *T** *p* mutatónk és egy *i* egész számunk, akkor a *p+i* kifejezés értéke is egy *T** típusú mutató lesz. Az összeadás a mutató számszerű értékét *i***sizeof**(*T*)-vel módosítja, azaz mutató értékét a hozzáadott egész szám a hivatkozott típus méretének megfelelő egységekben változtatja meg. Ezt a tulajdonságot kihasználva könnyen végiglépkedhetünk például egy tömb elemein, ha kezdetben a tömb első elemére mutatunk egy mutatóval, majd a mutatót mindig eggyel növeljük, amely így mindig a következő tömbelemre fog mutatni. Ha a hozzáadott érték negatív, a mutató természetesen az ellenkező irányba fog lépni.
- Azonos típusra hivatkozó mutatók kivonása egymásból. Az eredmény az az egész szám lesz, amelyet az első mutatóhoz adva a második mutatót kapjuk eredményül.
- Tömbszerű indexelés. A C nyelvben a tömb típusok és a mutatók közeli rokonságban állnak egymással. Egy tömb nagyon sok tekintetben úgy viselkedik, mint a tömb első elemére hivatkozó mutató. Ha *t* egy tömb, akkor a tömb *i*-dik elemére hivatkozás (*t*[*i*]) definíció szerint egyenértékű a **t*+*i*) kifejezéssel. Mivel ez utóbbi kifejezés tetszőleges *p* típusos mutató esetén értelmes, a C megengedi a *p*[*i*] alak használatát is, tetszőleges típusos mutatóra.

Ezen eszközök segítségével a C nyelv mutatótípusai igen hatékonyan használhatók, ami nem kis mértékben járult hozzá a nyelv népszerűségéhez.

Referencia típus a C++-ban

A C++ a mutatókon túl bevezette a referenciatípusokat is. Egy tetszőleges *T* típus esetén a *T*-re vonatkozó referencia típusa *T*&. A referencia típus olyan, mint egy konstans mutató. A deklaráció helyén inicializálni kell, és utána az inicializációhoz használt tárterület alternatív megnevezéseként használható. A referencia típusnál nem kell a referenciaképzés műveletet használni, mivel minden *T* típus automatikusan konvertálódik a neki megfelelő *T*& referencia típusra.

A referencia típusoknál nem kell jelölni a dereferenciát sem, mivel az inicializálást követően a referenciával már semmilyen művelet nem végezhető, így minden művelet értelemszerűen a hivatkozott objektumra vonatkozik. A referencia típusok használatának tényleges haszna alprogramok formális paraméterlistájában van. A referencia típusok segítségével megvalósítható a cím szerinti paraméterátadás a C++-ban. Ez a lehetőség a C-ből hiányzott. Helyette az objektum mutatóját kellett átadni érték szerint paraméterként. Ez azonban veszélyes volt, mivel a mutatók inicializálatlanok is lehetnek. A referencia típus mindig inicializált, így használata biztonságos.

Konstansra hivatkozó mutatók

Ha van egy p egészekre hivatkozó mutatónk, amely egy egész objektumra mutat, akkor p mutatóra megengedett a hivatkozás-feloldás művelete, és a mutatott objektum értéke megváltoztatható rajta keresztül, például hozzáadhatunk egyet. Tegyük fel, hogy van egy c konstansunk, amely egész típusú. Ha p-nek értékül adnánk egy c-re vonatkozó referenciát, akkor c értéke p-n keresztül megváltoztatható lenne, mivel p-n keresztül hivatkozva az objektumra nincs módunk megállapítani, hogy az konstans. Ahhoz tehát, hogy az ilyen jellegű szemantikai hibákat kivédhessük, azt kell megakadályozni, hogy p értékül kaphassa c referenciáját.

A C és a C++ a problémát úgy oldotta meg, hogy bevezette a **const** T típust, amely a T típusú konstansok típusa. Mivel T és **const** T két külön típus, a rájuk vonatkozó mutatók is külön típust alkotnak. Ha a mutató típusa **const** T*, a fordítóprogram hibát jelez, ha azon keresztül meg akarjuk változtatni a hivatkozott objektumot. A nyelv engedélyezi a T típus automatikus konverzióját **const** T-re, illetve a megfelelő mutató és referencia típusok között is automatikus konverzió lehetséges, így nem csak konstansokra állítható **const** T* referencia. Az így kapott mutató az adott nem konstans T típusú objektum *konstans nézete*. Alprogramok formális paramétereként, ha az alprogram nem kívánja megváltoztatni az argumentumként kapott mutatón keresztül a mutatott objektumot, célszerű a formális paramétert konstansként deklarálni. Ezáltal egyrészt biztonságosabb lesz az alprogram, mivel véletlenül sem módosíthatja a kapott paramétert, másrészt szélesebb körben használhatóvá válik, hiszen konstansra hivatkozó mutatóval is meghívható lesz. Az alábbi példa az *strcpy* szabványos könyvtári függvényt mutatja be.

```
char* strcpy(char* dest, const char* src)
{
    char *d = dest;
    while (*src != '\0') *d++ = *src++;
    *d = *src;
    return dest;
}
...
char puffer1[100], puffer2[100];

strcpy(puffer1, "Hello világ!");
strcpy(puffer2, puffer1);
```

Az Ada 95 is megkülönbözteti a mutatókat aszerint, hogy engedélyezett-e rajtuk keresztül a hivatkozott objektum megváltoztatása, de nem vezet be külön típust a konstansoknak, illetve a konstansra hivatkozó mutatóknak, hanem a mutatótípus deklarációjakor meg kell határozni, hogy milyen műveletek végezhetők a hivatkozott objektumokon. Konstansokra vonatkozó referencia csak olyan mutatónak adható értékül, amely csak olvasni engedi a hivatkozott objektumokat.

```
type EgészMutató is access all Integer;
type KonstansEgészMutató is access constant Integer;

IP : EgészMutató;
KP : KonstansEgészMutató;
I  : aliased Integer := 21;
K  : aliased constant Integer := 42;
begin
   IP := I'Access;     -- Helyes.
   IP := IP + 1;       -- Helyes.
   IP := K'Access;     -- Fordítási hiba!
   KP := K'Access;     -- Helyes.
   KP := KP + 1;       -- Fordítási hiba!
   KP := I'Access;     -- Helyes.
   KP := KP + 1;       -- Fordítási hiba! (konstans nézet)
end;
```

Vegyük észre, hogy a *KP* mutató értékét a program megváltoztathatja, mivel nem a mutató konstans, csupán a rajta keresztül hivatkozott érték.

5.7. Kifejezések

A programozási nyelvekben a kifejezések általában függvényhívások, változó-, illetve konstans-kiértékelések, valamint ezek kombinációi. A kifejezések alapvető fontosságúak a számítások leírásánál. Némely programozási nyelvekben – mint például a C vagy a C++ – a kifejezések a legelemibb utasítások. Az ALGOL 68-ban pedig a *unit*-ok – kifejezések – közé tartozik minden utasítás, beleértve a különböző vezérlési szerkezeteket is.

5.7.1. Kifejezések szerkezete

A függvények egyik fontos jellemzője az *aritásuk*, azaz az argumentumaik száma. Az egy aritású függvényeket *unáris*, a kettő aritásúakat *bináris* és általában a k aritásúakat *k-áris* függvényeknek nevezzük. Ha egy függvény aritását jelölni akarjuk a továbbiakban, akkor azt egy felső indexszel, f^k alakban tesszük. Ha a nullát is megengedjük mint lehetséges aritást, akkor a konstans értékeket is tekinthetjük függvényeknek, sőt bizonyos mértékig még a változókat is. Az ALGOL 68 vezérlési szerkezetei, blokkutasításai és egyéb unitjai is tekinthetők speciális függvényeknek, így a kifejezések lényegében egymásba ágyazott függvényhívásoknak tekinthetők.

A függvények speciális változatai az *operátorok*. Az operátorok halmaza a legtöbb programozási nyelvben rögzített. Az azonosítók túlterhelését (lásd a 7.6. részt) támogató nyelvek közül sem mindegyik engedi meg az operátor-túlterhelést – ilyen például a Java – és csak

egészen ritkán megengedett, hogy a programozó újabb operátorokat hozhasson létre. Ezen ritka kivételek egyike az Eiffel.

Az operátorok többségét nem nevekkel, hanem szimbólumokkal – például +, -, /, *, ++, < stb. – azonosítjuk, noha bizonyos nyelvekben vannak névvel jelölt operátorok is – pédául **new** vagy **delete**. Egy másik eltérés a közönséges függvényektől, hogy gyakran az operátorok meghívásának formája is különbözik. A közönséges függvényeknél általában a *prefix* jelölésmód a szokásos, és a függvény argumentumait zárójelek között kell felsorolni. Ennek oka, hogy a függvények aritása tetszőleges lehet és a zárójelek használata teszi lehetővé a fordítóprogram számára, hogy meghatározza az adott függvény aktuális paramétereinek listáját – $f(arg_1, \ldots, arg_k)$. Ezzel szemben az operátorok aritása rögzített – még az operátor túlterhelést megengedő nyelvek sem teszik lehetővé az aritás megváltoztatását – így a zárójelek elhagyhatók, illetve gyakran használatosak az *infix* és *postfix* jelölésmódok is.

Az operátorok, használatukat tekintve tehát négyfélék lehetnek, prefix, infix, postfix és mixfix operátorok.

- *Prefix* operátorok használatakor a kifejezésben először az operátor szerepel, majd utána az operandusok – k-áris operátor esetén $op^k\ arg_1 \ldots arg_k$. Mivel az operátorok aritása előre ismert és rögzített, az ilyen kifejezések balról jobbra olvasva egyszerűen kiértékelhetők – például a `-*bb**4ac` a $b^2 - 4ac$ kifejezés, prefix alakban.
- A *postfix* operátorok a kifejezésben operandusaik után állnak. Egy tetszőleges k-áris operátor esetén $arg_1 \ldots arg_k\ op^k$. A kifejezések ilyen felírását *lengyel formának* is nevezik. Az operátorok aritásának ismeretében az ilyen kifejezések egy verem segítségével könnyen kiértékelhetők: balról jobbra feldolgozva a kifejezést a változókat és konstansokat a verembe kell helyezni, ha pedig egy operátort olvasunk, a verem tetejéről az operátor aritásának megfelelő számú elemet levéve végrehajtjuk rajtuk a műveletet, és az eredményét tesszük a verem tetejére. A fenti példa postfix alakban `bb*4a*c*-`.
- *Infix* jelölésmód használata esetén a bináris operátorok a két argumentumuk között állnak, $arg_1\ op^2\ arg_2$ alakban. Ez a jelölés használatos például a aritmetikai műveleteknél, mivel a matematikában ezt az alakot szoktuk meg. Hátránya ennek a jelölésnek, hogy nem egyértelmű a kiértékelése. A `b*b-4*a*c` kifejezés feldolgozásakor csak a kifejezés alakja alapján nem dönthető el, hogy az egyes függvényhívások milyen sorrendben végezendők el. Az egyértelműség biztosításához be kell vezetni a *precedencia* és az *asszociativitás* fogalmát (lásd az 5.7.2. részt).
- A *mixfix* operátorok csoportját alkotják azok, amelyek nem sorolhatók be a fenti csoportok egyikébe sem. Ilyen operátor például a C-típusú nyelvekben a feltételes kifejezés operátor, amelynek alakja: ⟨*feltétel*⟩ ? ⟨*kifejezés-1*⟩ : ⟨*kifejezés-2*⟩. Kiértékelés szempontjából ebbe a csoportba tartoznak az ALGOL 68 vezérlési szerkezetei is.

A programozási nyelvekben általában vegyesen jelentkeznek a prefix, infix és postfix operátorok. Az unáris operátorok – lévén csak egy paraméterük van – gyakran prefix jelölésűek, de például a ++ és -- műveletek a C-típusú nyelvekben egyaránt használhatók pre- és postfix módon is. Sőt, a C++-ban, amely megengedi az operátorok túlterhelését, a két alakhoz társítható különböző viselkedés is.

A kétténvezős matematikai műveleteket – aritmetikai műveletek, relációs operátorok – általában infix operátorokkal valósítják meg a programozási nyelvek, de akadnak kivételek is. A CLU például nem használ ilyen kitüntetett operátorokat, minden függvény prefix alakban, paramétereit zárójelek között megadva használandó. Hogy a matematikai kifejezések írását megkönnyítse és olvashatóságukat javítsa, a CLU bevezetett egy *szintaktikus cukornak* nevezett eszközt. A szintaktikus cukor nem más, mint egy alternatív jelölésmód bizonyos függvényhívásokhoz. A bináris `add(a, b)` függvényhívás például írható `a + b` alakban is. Fontos azonban, hogy ez csak egy egyszerűsített jelölésmód, amelyet a fordítóprogram automatikusan az eredeti prefix függvényhívással helyettesít, még mielőtt egyáltalán értelmezni próbálná. Ennek köszönhetően ez a helyettesítés tetszőleges típuson működik, akár a programozó által definiáltakon is, feltéve, hogy az adott típus rendelkezik a megfelelő nevű és aritású művelettel.

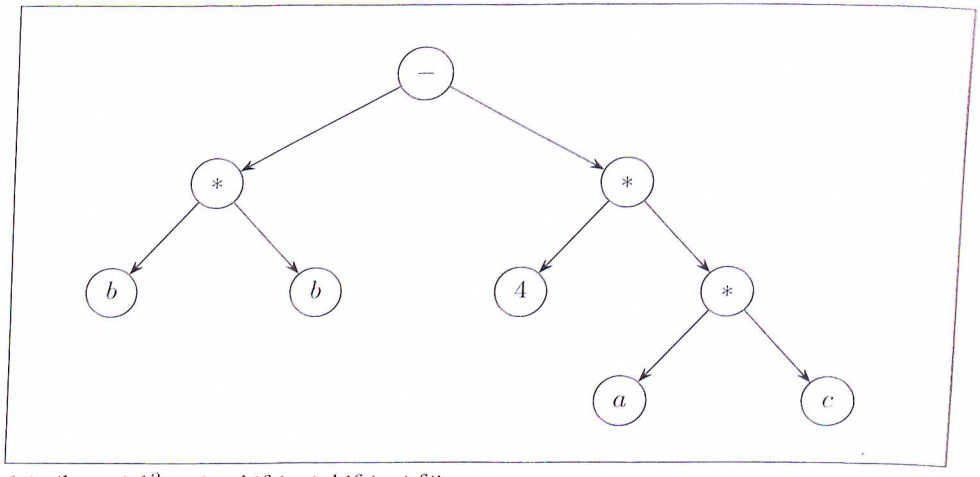
5.7.2. Kifejezések kiértékelése

Amint azt az előző részben láthattuk, a kifejezések lényegében egymásba ágyazott függvényhívások. A kifejezések kiértékelését megkönnyítendő vezessük be a *kifejezésfa* fogalmát:

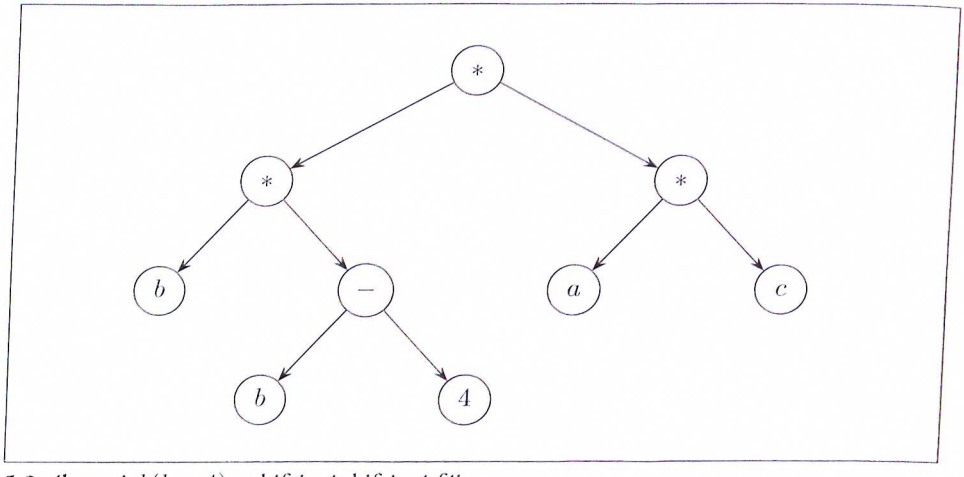
- Egy nulláris függvény – változó vagy konstans – kifejezésfája egyetlen csúcsból áll, amelynek címkéje maga a függvény.
- Egy k-áris függvény kifejezésfájának gyökere a függvénnyel címkézett, és élek indulnak belőle az egyes argumentumokhoz tartozó kifejezésfák gyökerébe.
- Egy vezérlési szerkezet kifejezésfájának gyökere a vezérlési szerkezettel címkézett és élek mutatnak belőle a benne szereplő kifejezések kifejezésfáinak gyökerébe.

Egy kifejezés kiértékelése nagyon egyszerű, ha ismerjük a hozzá tartozó kifejezésfát. A kifejezésfát a levelektől kezdve csúcsonként értékeljük ki.

- Egy levélcsúcs értéke a címkéjében található változó vagy konstans értéke. Kiértékelése bármikor elvégezhető.
- Egy k-áris függvényhez tartozó csúcs akkor értékelhető ki, ha a közvetlen leszármazottainak kiértékelése már befejeződött. A csúcs értéke ekkor a címkéjében szereplő függvény eredménye a közvetlen leszármazottakon mint argumentumokon végrehajtva. Azaz ha f^k a csúcs címkéje és a leszármazottak értéke rendre $v_1, \ldots, v_k$, akkor a csúcs értéke $f^k(v_1, \ldots, v_k)$ lesz.
- Egy vezérlési szerkezettel címkézett csúcs kiértékeléséhez végre kell hajtani a címkéjében szereplő utasítást. Ennek során a leszármazott csúcsok többször is kiértékelésre kerülhetnek. A csúcs értéke az utoljára kiértékelt kifejezés értékével fog megegyezni.



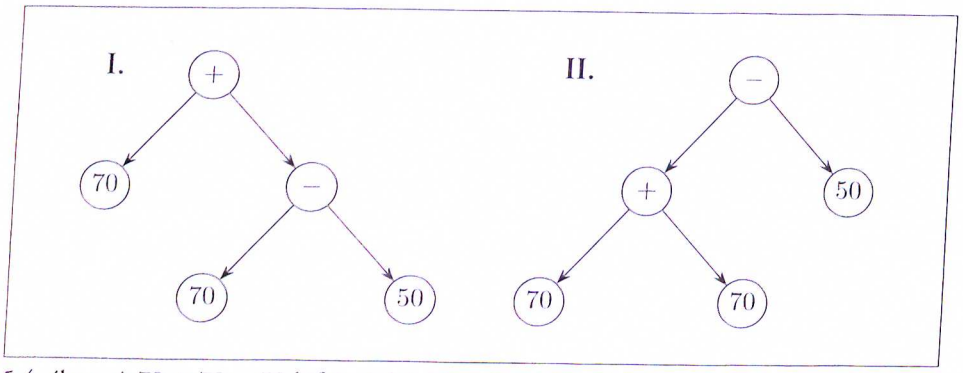
5.2. ábra: A $b^2 - 4ac$ kifejezés kifejezésfája



5.3. ábra: A $b(b - 4)ac$ kifejezés kifejezésfája

Az 5.2. ábrán látható a $b^2 - 4ac$ kifejezés kifejezésfája. Ugyanezt a fát kapjuk a prefix `-*bb**4ac` kifejezésre, és az postfix `bb*4a*c*-` változatra. Mindkét felírás egyértelműen meghatározza a kifejezésfát és ezáltal a kifejezés értékét is, azaz a szintaxis eszközeivel garantálni tudjuk, hogy a kifejezés szemantikája egyértelmű legyen.

Korántsem ennyire egyértelmű a helyzet az infix felírással. A `b*b-4*a*c` kifejezésre illeszkedik az 5.2. ábrán látható kifejezésfa, de formailag ugyanígy illeszthető rá az 5.3. ábrán látható változat is.



5.4. ábra: A 70 + 70 − 50 kifejezés két lehetséges kifejezésfája

Infix jelölésmód esetén a szintaxis eszközei nem elégségesek a kifejezésfa egyértelmű meghatározásához és így a kifejezés szemantikája nem egyértelmű. Ez a probléma nem a programozási nyelvek sajátja, hiszen az infix jelölésmódot a matematikából vették át. Az egyértelműség eléréséhez a matematikában is egy másodlagos eszközre, a *precedencia* fogalmára támaszkodunk. A precedencia meghatározza, hogy egy infix formájú kifejezésben melyik operátor „köt jobban" azaz ha a sorrend nem egyértelmű, melyiket kell előbb kiértékelni. A precedencia az, ami meghatározza, hogy a $b^2 - 4ac$ kifejezésben a szorzásokat kell előbb elvégezni és csak utána a kivonást, és így a helyes értelmezés az 5.2. ábrán látható. A precedenciaszabályok felülbírálására a zárójelek használata ad lehetőséget.

A precendecia bevezetésével az operátorokat osztályokba soroljuk, a nem egyértelmű kifejezésfa problémája azonban ezzel még nem oldódik meg teljesen, ugyanis vannak olyan operátorok, amelyek egyazon osztályba kerülnek. Ennek leggyakrabban az az oka, hogy az operátorok matematikai megfelelői is egyazon precedencia szinten vannak. Például az összeadás és a kivonás precedenciája a matematikában azonos, és ott ez nem is okoz különösebb problémát, ha egy kifejezés csak összeadásokat és kivonásokat tartalmaz, a műveletek tetszőleges sorrendben elvégezhető és az eredmény ugyanaz lesz. A számítógépek világában azonban más a helyzet. Tegyük fel, hogy Ada nyelven programozunk és az éppen használt egész típusunk típusértékhalmaza a *−100..100* intervallum. Tegyük fel továbbá, hogy az *70+70−50* kifejezés értékét szeretnénk kiszámolni. Az 5.4. ábrán láthatjuk a lehetséges kifejezésfákat. Matematikai szempontból a két kifejezésfát kiértékelve ugyanazt az eredményt kapjuk, de a konkrét megvalósítást figyelembe véve az eltérés igen jelentős. Ha ugyanis az I. változatot értékeljük ki, akkor az eredmény *90* lesz, a várakozásainknak megfelelően. A II. változatot kiértékelve azonban az összeadás eredményeként *CONSTRAINT_ERROR* kivételt kapunk, a részeredmény ugyanis meghaladja az ábrázolható legnagyobb számot.

Hasonló problémával találkozhatunk, ha a *10*10/100* kifejezést próbáljuk kiértékelni. A kifejezés matematikai értéke 1, függetlenül attól, hogy először a 10/100 osztást vagy a $10 * 10$ szorzást végezzük el. A legtöbb programozási nyelvben azonban az eredmény nem ennyire egyértelmű. A matematikában ugyanis $10/100 = 0.1$, míg a legtöbb programozási

Prec.	Operátorok	Asszociativitás
1.	Zárójel: (és)	nincs
2.	Függvényhívás: `()`, tagkiválasztás: `->` és `.`	balról jobbra
3.	Unáris operátorok: előjel operátorok: `+` és `-`, logikai negáció: `!`, aritmetikai negáció: `~`, címképzés: `&`, dereferencia: `*`, pre- és postfix növelés: `++`, -csökkentés: `--`, típuskényszerítés: (⟨*típus*⟩), méretlekérdezés: `sizeof`	jobbról balra
4.	Multiplikatív operátorok: szorzás: `*`, osztás: `/`, maradékképzés: `%`	balról jobbra
5.	Additív operátorok: összeadás: `+`, kivonás: `-`	balról jobbra
6.	Bitenkénti léptetés: balra: `<<`, jobbra: `>>`	balról jobbra
7.	Relációs operátorok: `<`, `<=`, `>` és `>=`	balról jobbra
8.	Egyenlőség: `==`, nemegyenlőség: `!=`	balról jobbra
9.	Aritmetikai „és": `&`	balról jobbra
10.	Aritmetikai „kizáró vagy": `^`	balról jobbra
11.	Aritmetikai „vagy": `|`	balról jobbra
12.	Logikai „és": `&&`	balról jobbra
13.	Logikai „kizáró vagy": `^^`	balról jobbra
14.	Logikai „vagy": `||`	balról jobbra
15.	Értékadás: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, és `>>=`	jobbról balra
16.	Vessző operátor: `,`	balról jobbra

5.1. táblázat: Precedenciaosztályok és asszociativitásuk a C nyelvben.

nyelvben a *10/100* kifejezés értéke *0*. A különbség abból fakad, hogy a matematikában egyetlen osztás művelet van, az eredmény „típusa" – egész vagy nem egész szám volta – az osztás elvégzése után derül ki. Ezzel szemben a legtöbb programozási nyelvben *egy kifejezés típusa csak az operandusok típusától függ és fordítási időben meghatározott*, azaz ha két egész számot osztunk el egymással, akkor az eredmény is egész lesz.

Ezeknek a problémáknak a kezelésére, vagyis az azonos precedenciaosztályba eső operátorok kiértékelési sorrendjének meghatározására szolgál az *asszociativitás*. Ha a $\odot$ operátor *balasszociatív*, akkor a $v_1 \odot v_2 \odot \cdots \odot v_n$ kifejezés kiértékelése a $(\ldots(v_1 \odot v_2) \odot \ldots) \odot v_k)$ zárójelezésnek megfelelően – másszóval balról jobbra – kell, hogy történjen. A *jobbasszociatív* operátorok kiértékelése szimmetrikusan, jobbról ballra történik. Az asszociativitás iránya egy precedenciaosztályon belül azonos, így az adott osztályba tartozó operátorok vegyesen is szerepelhetnek egy kifejezésben, a kiértékelésük sorrendje jól meghatározott marad. Természetesen zárójelezéssel ez is felülbírálható.

A legtöbb programozási nyelvben az operátorok túlnyomó többsége balasszociatív. A C és C++ nyelvekben azonban vegyesen szerepelnek bal- és jobbasszociatív műveletek is. Az 5.1. táblázatban a C nyelv operátorainak összefoglalása látható. A táblázatban felülről lefelé csökkenő sorrendben szerepelnek a precedenciaosztályok.

A műveletek precedenciája és asszociativitásuk iránya, hasonlóan az operátorok aritásához, a legtöbb nyelvben nem változtatható meg. Érdekes kivétel az ALGOL 68, ahol a programozó a nem unáris operátorok precedenciáját szabadon változtathatja 1 és 10 között. Az alábbi példában a kiíratás eredménye 45 lesz.

```
begin
    prio +=3, *=2;
    print(6 + 3 * 5);
end
```

5.8. Egyéb nyelvi specialitások

Ebben a részben áttekintjük azokat az egyéb nyelvi specialitásokat, amelyek a típusokhoz kapcsolódnak, de nem fértek bele a fenti, típusosztályozás-vezérelte áttekintésbe.

Ada: típusszármaztatás és altípusok

Az Ada programozási nyelvben a típusokat létrehozásuk módja szerint két csoportba sorolhatjuk.

Az *altípusokat* egy meglévő típus típusértékhalmazának megszorításával kapjuk. Az altípusképzés valójában nem hoz létre új típust, csupán kijelöli a típusértékhalmaz egy részét, ezért az altípus objektumai és az alaptípus objektumai szabadon értékül adhatók egymásnak, az értékadás szintaktikailag helyes, de végrehajtásakor – futási időben – egy értékhatár-ellenőrzés is történik, és sikertelensége esetén *CONSTRAINT_ERROR* kivétel váltódik ki. Az alábbi példában két, a nyelv specifikációjában is szereplő altípust láthatunk:

```
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

Ezzel szemben a *származtatott típusokat* úgy kapjuk, hogy egy meglévő típust „lemásolunk", azaz a származtatott típus örökli az őstípus ábrázolását, műveleteit, de azoknak egy másik *példánya* lesz. A származtatott típus új típus, ezért értékadás például nem lehetséges az őstípus és közte – bár konverziós operátor azért használható.

Miért hasznos a típusszármaztatás módszere? Tegyük fel, hogy olyan programot készítünk, amely a megtett útból (s) és az eltelt időből (t) a jól ismert fizikai összefüggés alapján ($v = s/t$) akarja a sebességet meghatározni. Az út, az idő és a sebesség is valós mennyiségek, így kézenfekvőnek tűnik az alábbi megoldás:

```
S : Float := 42.0;
T : Float := 21.0;
V : Float := S * T;
```

Illetve mégsem! A program tartalmaz egy „apró" elírást, így az eredmény hamis lesz, de erről csak program futtatása során és esetleg csak fáradságos hibakeresés árán szerzünk tudomást.

Az alábbi, javított változat épít a típusszármaztatás eszközére:

```
type Út       is new Float range 0.0 .. Float'Last;
type Idő      is new Float range 0.0 .. Float'Last;
type Sebesség is new Float range 0.0 .. Float'Last;

function "/"(S: Út; T: Idő) return Sebesség is
begin
   return Sebesség(Float(S)/Float(T));
end "/";

S : Út       := 42.0;
T : Idő      := 21.0;
V : Sebesség := S / T;
```

Ebben az esetben a három fizikai mennyiséghez három különböző típust hozunk létre. Ezen felül szükségünk van az osztás művelet megvalósítására, amely egyszerűen felhasználja, hogy az ősosztály és a származtatott osztály között lehetséges a konverzió. Mivel a típusok között az értékadás nem megengedett és csak az általunk definiált műveletek használhatók, ha bármi mást írnánk a sebesség kiszámolásakor mint osztást, amelynek számlálójában egy *Út*, a nevezőjében pedig egy *Idő* típusú kifejezés van, akkor a fordítóprogramtól – tehát fordítási időben – hibaüzenetet kapnánk. Felhasználva tehát az Ada nyelv erős típusosságát, szemantikai összefüggéseket fejezhetünk ki, amelyek teljesülését a fordítóprogram ellenőrizni képes!

5.9. Összefoglalás

Foglaljuk akkor össze még egyszer, hogy milyen kérdéseket kell megválaszolnunk, ha egy programozási nyelv típusait vizsgáljuk.

1. *Erősen vagy gyengén típusos a nyelv?*
   Mennyire végez a fordítóprogram erős típusellenőrzéseket? Ismert-e minden esetben az objektumok, változók, függvények, alprogram formális paraméterek stb. típusa már fordítási időben? Ezek a kérdések alapvetően meghatározzák az adott programozási nyelv használatát és használhatóságát. Erősen típusos nyelvekre példa az Ada, CLU, Java vagy C++. Gyengén típusos nyelvek a dBase, Smalltalk, Perl stb.
2. *Épít-e a nyelv a típusosztályokra?*
   A programozási nyelvek típusai többé-kevésbé mindig besorolhatók az általunk definiált vagy ahhoz nagyon hasonló típusosztályokba. Bizonyos nyelveknél azonban a típusosztályok szerves részét képezik a nyelv specifikációjának és a nyelv egyéb konstrukciói erre épülnek. Ilyen nyelv például az Ada, ahol a típusosztályok meghatározzák a típusok műveleteit, és azok használhatóságát, de megjelennek például a sablonoknál is, mint a formális paraméterek „típusa". Más nyelvekben az osztályok szerepe kisebb, de azért kimutatható. Például a Pascal nyelvben a tömböket diszkrét típusokkal lehet indexelni, és vannak közös műveletek, amelyek minden

diszkrét típusra használhatók. Végezetül vannak nyelvek, ahol a típusok osztályozása csupán mesterséges, a nyelven kívül álló konstrukció, amely leginkább csak az egységes tárgyalást könnyíti meg. Ilyen nyelvek a C vagy a FORTRAN.

3. *Milyen beépített típusokat tartalmaz a nyelv?*
   A beépített típusok köre nagyjából hasonló a különböző programozási nyelvekben, bár van némi eltérés, például a numerikus típusoknál az ábrázolási pontosság tekintetében. Vannak olyan típusok is, amelyek csak ritkábban fordulnak elő, ilyen például a komplex számok típusa, amely csak néhány nyelvben támogatott beépítetten (például FORTRAN, ALGOL 68).
4. *Van-e a nyelvben felsorolási típus?*
   A felsorolási típus a legegyszerűbb típuskonstrukciós eszköz, a programozási nyelvek mégis meglehetősen megosztottak a támogatásában. Még a modernnek mondható nyelvek közül is sok – például a Java vagy az Eiffel – egyáltalán nem támogatja, vagy – a C-hez és a C++-hoz hasonlóan – csak egyszerű „szintaktikus cukorként" valósítja meg.
5. *Ha nincs felsorolási típus, milyen más megoldást kínál a nyelv a kiváltására?*
   A felsorolási típus pótlására számtalan variációt láttunk, az Eiffel **unique** deklarációitól egészen a Java felsorolási típus osztályaiig. Ezeknek a megoldásoknak a vizsgálatakor mindig azt kell szem előtt tartani, hogy mennyire támogatják a típushelyes használatot.
6. *Milyen egész típusok vannak a nyelvben?*
   A programozási nyelvek mindegyike tartalmaz valamilyen beépített egész típust, de az ábrázolható számtartomány tekintetében a kép széles spektrumban változik. A legtöbb nyelv a processzorba épített egészaritmetikára alapozza az egész szám megvalósítását, hatékonysági okokból, de az elérhető típusok változnak. Szokásosak a 8, 16, 32, 64 bites, előjeles vagy előjel nélküli egész típusok, de a Python esetében láthattunk példát olyan egész megvalósításra, amely teljesen független a processzorba épített egész típustól, az ábrázolható számoknak csak a memória mérete szab korlátot.
7. *Van-e a nyelvben logikai típus?*
   A nyelvek jelentős részében van logikai típus, de vannak nyelvek – ilyen például a C –, amelyben szerepét teljes egészében átvették az egész típusok.
8. *Ha van, hová tartozik a logikai típus?*
   Azokban a nyelvekben is, amelyek tartalmaznak beépített logikai típust, változik annak megítélése. A Pascalban vagy az Adában a felsorolási típusok közé tartozik, annak tulajdonságait viseli. Más nyelvekben, mint például a CLU vagy a Delphi, inkább az egész típusok közé sorolható.
9. *Van-e a logikai operátoroknak lusta és mohó kiértékelésű változata is?*
   Amint láthattuk, az Ada, CLU vagy Eiffel nyelvekben a programozó kétféle bináris operátor közül választhat. Az egyik a hatékonyságot tartja szem előtt, és csak akkor értékeli ki a második operandusát, ha az első alapján a kifejezés értéke még nem meghatározott, míg a másik változat, felkészülve az operandusok kiértékelésének esetleges mellékhatásaira, mindig kiértékeli mindkét operandust. A nyelvek többsége csak az első, lusta kiértékelésű operátorokkal rendelkezik.

10. *Hová tartozik a karakter típus?*
A karakter típus, a logikaihoz hasonlóan, átmenetet képez a felsorolási és egész típusok között. Az Ada vagy Pascal nyelvekben a felsorolási típusok közé tartozik, más nyelvekben – C, C++, Java – az egészek közé.
11. *Definiálja-e a nyelv a karakterkódolást?*
A C esetében azt láthattuk, hogy nyelv jórészt független a használt karakterkódolástól, míg a Java vagy az Ada pontosan meghatározzák a használandó jelkészleteket.
12. *Milyen valós típusokat támogat a nyelv?*
A valós típusokhoz a legtöbb nyelv lebegőpontos számábrázolást használ, de az ábrázolási pontosság széles skálán változhat, a Pascal 80 bites *Extended*-jétől a 32 bites egyszeres pontosságú számokig. Ezen felül bizonyos nyelvek – például Ada – támogatják a fixpontos számábrázolást is.
13. *Milyen memóriagazdálkodást támogat a nyelv?*
A C vagy C++ esetében láthattuk, hogy a memóriagazdálkodás teljesen a programozó kezében van. A legtöbb objektumorientált nyelv esetén viszont a nem használt memória felszabadításáról egy szemétgyűjtő mechanizmus gondoskodik.
14. *Ha a nyelv a szemétgyűjtést támogatja, van-e lehetőség mégis az explicit deallokációra?*
A hatékony programozás néha megköveteli, hogy a programozó maga vegye kezébe a memóriagazdálkodást. A szemétgyűjtéses nyelvek közül némelyik – például az Ada – lehetővé teszi ezt a programozó számára. A Java ezzel szemben az explicit memóriafelszabadítást nem engedélyezi, de programozó kezdeményezheti a szemétgyűjtő algoritmus végrehajtását, ezáltal siettetve a már használaton kívüli memóriablokkok felszámolását.
15. *Van-e a nyelvben típus nélküli mutató?*
A C és a Pascal nyelvben lehetőség van olyan mutató típus használatára, amely semmilyen típusra nem mutat. Ezen mutatótípus előnye, hogy bármely típusos mutatóval kompatibilis. Más nyelvek – például az Ada – nem támogatják az ilyen mutatókat, mivel használatuk veszélyes. Azokban a modern objektumorientált nyelvekben, ahol van közös ősosztálya a nyelv osztályainak, a közös ősre vonatkozó referencia kiválthatja a típus nélküli mutató feladatait, jelentősen csökkentve a használatával járó kockázatokat. Ilyen például a Java nyelvben az *Object* típusú referencia.
16. *Van-e a nyelvben alprogramra hivatkozó mutató?*
A C, C++ és Ada 95 nyelvekben mutathat mutató alprogramokra is, amelyek így meghívhatók a mutatón keresztül. Más nyelvek, például az Ada 83 vagy a Java, erre nem adnak lehetőséget, de az objektumorientált programozási nyelvekben függvényobjektumok használatával hasonló hatás érhető el.
17. *Állítható-e mutató statikus vagy automatikus változókra?*
Az Ada 95, a C és C++ lehetővé teszi, hogy az automatikus, illetve statikus változókhoz referenciát képezzünk és ezeket mutatóknak értékül adjuk. Más nyelvek, mint például a Java vagy az Ada 83 a mutatókat a dinamikus változókra korlátozzák.
18. *Milyen konstans mutatók vannak a nyelvben?*
Lehet-e egy mutató konstans? Mutathat-e egy mutató konstansra? Megváltoztatható-e a konstans értéke a mutatón keresztül?
19. *Milyen egyéb specialitással rendelkezik a nyelv típusrendszere?*

Feladatok

1. Mutasson példát erősen típusos programozási nyelvre, amely nem statikusan típusos!

2. Mi a típusa a Java nyelvben az alábbi literáloknak?

```
5465435      '\u2343'
123          "\u2343"
'a'          "\\u2343"
3.14
```

3. Hasonlítsa össze a lebegőpontos és fixpontos számábrázolást!

4. Milyen veszélyekkel jár ha egész konstansokat használunk felsorolási típusok helyett?

5. Készítsen Javában felsorolási típust a hét napjainak ábrázolására!

6. Mi a különbség a következő két C nyelvű kifejezés között: *A&B*, *A&&B*?

7. Mi a `3 / 2` kifejezés értéke a C, az Ada és a Pascal nyelvben? Magyarázza meg a különbséget!

8. Mi a különbség a lusta és mohó logikai operátorok között? Mutasson példát mindkettő alkalmazására!

9. Mi a hiba az alábbi C nyelvű programrészletben?

```
long* keres(long *elso, long *utolso, long ertek)
{
    long *p = elso;
    while (p <= utolso) {
        if (*p == ertek) {
            return p;
        }
        p += sizeof(long);
    }
    return NULL;
}
```

10. Miért kell az Ada 95 nyelvben az automatikus változókat **aliased** minősítéssel ellátni, ha a referenciaképzés műveletét akarjuk rajtuk alkalmazni?

11. Készítsen programot lengyel formában adott kifejezések kiértékelésére!

12. Készítsen programot, amely egy infix jelöléssel adott kifejezést lengyel formára hoz!

6 Típuskonstrukciók

A programozási nyelvek beépített típusainak készlete, legyen az bármilyen kiterjedt, nem alkalmas önmagában minden feladat megoldására. A különböző feladatok megoldása szükségessé teszi, hogy meglévő típusainkat kombinálva új típusokat hozhassunk létre. Ebben a fejezetben áttekintjük, milyen típuskonstrukciós eszközöket kínálnak a programozási nyelvek, újabb típusok létrehozására.

Az előző fejezetben áttekintettük a programozási nyelvekben megszokott beépített típusokat. Ezek egy alap eszköztárat nyújtanak számunkra, amely azonban messze nem elég ahhoz, hogy feladatainkat megoldhassuk, ezért a legtöbb programozási nyelv[1] nyújt valamilyen lehetőséget arra, hogy a meglévő típusok készletét kiegészítsük, illetve belőlük különböző összetett szerkezeteket hozzunk létre.

A típuskonstrukciós eszközöknek három alapvető formáját határozzuk meg. Nem mindig áll rendelkezésünkre mindegyik egy-egy programozási nyelvben és a megjelenési formájuk is sokféle lehet, mindazonáltal bizonyos alapvető tulajdonságaikat megőrzik, így van értelme kategorizálásuknak.

- *Direktszorzat típus*
- *Unió típus*
- *Iterált típus*

Ezek az elnevezések a típuskonstrukciós megoldások absztrakt nevei. A különböző programozási nyelvek ezeket az eszközöket egymástól némileg eltérően valósítják meg és gyakorta más néven illetik őket. Az egyes eszközöket először általánosan mutatjuk be, majd kitérünk az egyes programozási nyelvek specialitásaira. A különbséget hangsúlyozandó, ha az adott konstrukcióról általánosságban beszélünk, akkor a fenti nevek valamelyikével hivatkozunk majd rá, ellenben ha egy nyelv specialitásáról van szó, akkor az adott nyelvbeli szóhasználatot követjük.

A fejezetben, a típusokkal kapcsolatos eddigi gyakorlatnak megfelelően, a típuskonstrukciós eszközök hatását a létrejövő összetett típus típusértékhalmazának, illetve műveleteinek bemutatásával adjuk meg.

Megjegyzés: a fejezetben a fenti típuskonstrukciós eszközökre használt definíció némileg eltér a programozási módszertanban használttól. Ennek oka, hogy a programozási módszertanban a típuskonstrukciót csak mint absztrakt adattípusok létrehozásának eszközét

[1] Vannak nyelvek, amelyekben nincsenek típuskonstrukciós eszközök. Ilyenek az assembly nyelvek vagy a speciális célnyelvek, mint például a SuperNova, amelyet adatbázistáblák kezelésére terveztek. Mindazonáltal az assembly nyelveket leszámítva az általános célú programozási nyelvek mindegyike tartalmaz valamilyen típuskonstrukciós lehetőséget.

használjuk, míg a programozási nyelvekben a típuskonstrukciók és az adatabsztrakció támogatása egymásra épülő, ám egymástól függetlenül is létező fogalmak. Erre a kérdésre az adatabsztrakcióval foglalkozó fejezetben még részletesebben is visszatérünk (lásd a 10.1. fejezet).

Típusekvivalencia

A típusok ekvivalenciája nagyon fontos kérdés a programozási nyelvekben. Ha a nyelv típusos, a típusekvivalencia határozza meg, hogy mely változók között engedélyezett az értékadás, egy adott formális paraméternek milyen aktuális paraméterek feleltethetők meg stb. A típuskonstrukciók kapcsán újra át kell gondolnunk, hogy mikor nevezünk két típust ekvivalensnek. Nézzük az alábbi Ada nyelvű példát:

R: **array***(1..10)* **of** *Integer;*
S, T: **array***(1..10)* **of** *Integer;*

A legtöbb programozási nyelv lehetőséget ad a fentihez hasonló deklarációkra. Kérdés, hogy *R*, *S*, illetve *T* típusai milyen viszonyban állnak egymással?

Az Ada a legtöbb erősen típusos programozási nyelvhez hasonlóan különbözőnek tekinti nemcsak *R* és *S*, hanem *S* és *T* típusát is, mivel az egy sorba írást csupán a külön sorba írás rövidített változatának tekinti. Egy, a fentiekhez hasonló változódeklaráció minden alkalommal egyben tartalmazza egy – külön nem nevesített – típus deklarációját is. Ebből az következik, hogy a három deklaráció három külön típus létrejöttét eredményezi és mivel a típust nem nevesítettük, a későbbiekben hivatkozni sem tudunk rájuk. Ugyancsak ebből következik, hogy – az erős típusosság miatt – ezen változók nem adhatók értékül egymásnak, mivel típusuk különbözik. Az ilyen nyelvekben a típusok ekvivalenciáját *névekvivalenciának* nevezzük, mivel csak az azonos nevű típusok számítanak ekvivalensnek, és így a változók típusainak ekvivalenciája egy nevesített közös típus létrehozásával oldható meg.

Ennél a megközelítésnél enyhébbet képvisel a C nyelv. Az alábbi példában az első értékadás típushelyes, míg a második nem. Ez egyben azt is jelenti, hogy a C nyelv lényegileg különbözőnek tekinti az egy sorba és a külön sorba írt deklarációkat.

```
struct {
    int x;
} a;

struct {
    int x;
} b, c;
...
b = c;      /* Érvényes értékadás. */
a = b;      /* Hibás értékadás. */
```

A másik végletet valósítja meg a Modula-3 által használt *strukturális típusekvivalencia*, amely két típust nem akkor tekint azonosnak, ha a nevük, vagy a deklarációjuk egyezik, hanem amikor a szerkezetük azonos. Ez a megoldás azonban nagyban gyengíti a típusosság nyújtotta biztonságot, hiszen így szemantikailag teljesen különböző típusokat, tisztán a reprezentációjuk azonossága miatt tekintünk ekvivalensnek.

Mutable és immutable típusok

A típusok osztályozásának egy másik szempontja annak vizsgálata, hogy megváltoztathatók-e a típus objektumai. Azokat a típusokat, amelyek objektumait a típus műveletei megváltoztathatják, *mutable*, míg azon típusokat, amelynek műveletei nem változtatják meg a típus objektumait, *immutable* típusoknak nevezzük.

Az immutable tulajdonság azonban nem azonos a konstansséggal. Egyrészt mivel a konstansság mint tulajdonság egy objektum jellemzője, míg az immutable tulajdonság egy típusé, másrészt egy immutable típusnak lehetnek megváltoztató jellegű műveletei. Az alábbi Java nyelvű példa egy immutable komplex szám osztály megvalósításának részlete.

```
public final class Complex {
    private final double re;
    private final double im;

    Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public Complex add(Complex z) {
        return new Complex(re + z.re, im + z.im)
    }
    ...
}
```

Az objektum két mezője, amelyek a komplex szám kanonikus alakjának valós, illetve képzetes részét tárolják, a **final** módosítóval vannak ellátva, ami azt jelenti, hogy ha egyszer értéket kaptak, értékük többet nem változtatható meg, azaz a típus valóban immutable. Ennek ellenére a típusnak van összeadás művelete, amely oly módon működik, hogy létrehoz egy új objektumot, amely az eredetitől pontosan a művelet hatásában tér el, azaz jelen esetben már az összeget tartalmazza.

Látszólag ez a viselkedés nagyon gazdaságtalan, hiszen egy bonyolultabb kifejezés kiértékelése során számtalan objektum keletkezhet és szűnhet meg, amíg a végeredmény megszületik, és csakugyan, ha az objektumok létrehozása és megszüntetése „drága" művelet, akkor nem célszerű az immutable típusok használata. Ugyanakkor az immutable típusoknak számtalan előnyük is van. A legfontosabb talán, hogy többszálú végrehajtás során az immutable típusok objektumain nem szükséges a szinkronizáció, nem kell a kölcsönös kizárás, hiszen az objektum értéke nem változhat meg, így nem fordulhat elő, hogy valamelyik olvasó szál inkonzisztens állapotban találja az objektumot. Egy másik előny lehet, hogy ha több másik objektum hivatkozik az immutable objektumra, akkor sincs szükség külön másolatok létrehozására, hiszen az első megváltoztató művelet ezt automatikusan megteszi. Ez a „lusta" másolásos megoldás néha igen jelentős teljesítménynövekedést eredményezhet.

Összefoglalva azt mondhatjuk, hogy – a feladat természetétől függően – az immutable típusok használata biztonságosabbá és hatékonyabbá teheti programunkat, ezért célszerű megvizsgálni, hogy a programozási nyelvek milyen támogatást nyújtanak az immutable típusok létrehozására. A fenti példában láthattuk, hogy például a Java nyelvben van lehetősé-

günk az immutable viselkedés elérésére a nyelv típuskonstrukciós eszközeinek felhasználásával. A Java nyelv ki is használja ezt, hiszen számtalan beépített típusa – *String*, *BigNumber*, *Integer* és a többi primitív típus csomagoló osztályai stb. – maga is immutable, mindazonáltal a Javában nincs explicit eszközünk az immutabilitás kifejezésére. Ezzel ellentétben a CLU nyelvben például, amint azt a későbbiekben látni fogjuk, minden típuskonstrukciós eszköznek megvan a mutable és immutable változata, így a nyelv maga gondoskodik a létrehozott típus helyes viselkedéséről.

Meg kell jegyeznünk, hogy az immutable típusok használhatóságához, ha nem is elengedhetetlen, de fontos, hogy a programozási nyelvben a változók ne magukat az objektumokat, hanem azok referenciáit tartalmazzák, hiszen enélkül az adott típus változói csakugyan konstanssá válnának. Ezen kívül fontos, hogy a nyelv támogassa a szemétgyűjtéses memóriakezelést, mivel ellenkező esetben a kifejezések kiértékelésekor létrejövő és megszűnő objektumok kezelése nagy terhet ró a programozóra. Figyeljük meg, hogy mind a CLU, mind a Java nyelvek bírnak mindkét tulajdonsággal és építenek is az immutable típusokra.

6.1. Direktszorzat típus

Tegyük fel, hogy éppen egy vállalat dolgozóinak nyilvántartását készítjük el. Egy-egy dolgozóról számtalan adatot kell nyilvántartanunk. A nevét, a lakcímét, a bankszámlája számát, melyre a fizetését utaljuk, magát a fizetését stb. Minden egyes dolgozóhoz ezen adatok mindegyikével rendelkeznünk kell. Egy ilyen adatcsoportot pedig összefog az a tény, hogy egyazon dolgozóra vonatkoznak. Ennek az összetartozásnak a kifejezésére használhatjuk a *direktszorzat* típuskonstrukciót.

A direktszorzat típuskonstrukció meglehetősen gyakori a különböző programozási nyelvekben, bár akadnak olyanok, amelyekből hiányzik (ilyen például a BASIC vagy a FORTRAN). A programozási nyelvekben a direktszorzat típuskonstrukciót leggyakrabban *rekordnak* nevezik.

6.1.1. Típusértékhalmaz

Egy direktszorzat megadása általában nem áll másból, mint a komponens típusok felsorolásából és a komponensekhez tartozó *szelektorok* (lásd a 6.1.2. részt) megnevezéséből. Például a fenti feladatot az Ada 95 programozási nyelvet használva a következő típusdefinícióval oldhatnánk meg:

```
type Dolgozó_Típus is record
   Név       : Név_Típus;
   Cím       : Cím_Típus;
   Számlaszám : Számlaszám_Típus;
   Fizetés   : Fizetés_Típus;
end record;
```

A *_Típus* végű azonosítók a komponens típusok nevei, míg a kettőspont előtt megadott azonosítók a megfelelő komponenshez tartozó szelektort jelölik.

Direktszorzatokban megengedett, hogy a komponens típusok – $\mathcal{T}_i$ – között legyenek azonosak, de a szelektoroknak mind különbözőnek kell lenniük. Szintén jelentőséggel bír a komponensek megadásának sorrendje. Egyrészt befolyásolhatja a direktszorzat memóriafoglalását (lásd a 6.1.3. részt), másrészt a strukturális típusekvivalenciát használó nyelveknél két típus megegyezőségéhez a komponens típusok egyezésén felül megadásuk sorrendjének is azonosnak kell lennie.

Mindezeket figyelembe véve, az alábbi formális definíciót kapjuk a direktszorzat típuskonstrukcióra: legyenek $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ típusok és $T_1, T_2 \ldots T_n$ a hozzájuk tartozó típusértékhalmazok. A $\mathcal{T}$ típus a $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ típusok *direktszorzata*, ha típusértékhalmaza $T = T_1 \times T_2 \times \cdots \times T_n$.

6.1.2. Műveletek

Tekintsük most át, milyen műveleteket végezhetünk egy direktszorzatként létrehozott típus elemein.

Szelektorok

A direktszorzat típusok legfontosabb művelete a *szelekció*, azaz ha adott egy $t \in T$, ahol $t = (t_1, t_2, \ldots t_n) \wedge \forall i \in [1 \ldots n] : t_i \in T_i$, akkor szeretnénk hozzáférni a t_i komponensekhez.

Erre használhatók a *szelektorok*. Az $s_i : T \rightarrow T_i$ függvényeket szelektoroknak nevezzük, ha $\forall i \in [1 \ldots n] : s_i(t) = t_i$. A szelektorok értéke igen gyakran nem is T_i hanem T_i-re vonatkozó referencia[2] típusú, mivel a szelektoron keresztül általában megváltoztatható az adott komponens értéke, azaz a szelektor *balérték*. Igaz ez még azokban a programozási nyelvekben is, amelyek egyébként nem támogatják a referencia típusokat.

A programozási nyelvekben a szelektor használatát gyakran nem $s_i(t)$, hanem $t.s_i$ alakban jelölik. A szelektorokat a direktszorzat létrehozásakor szokás kijelölni.

A szelektorok segítségével tehát elérhetjük a direktszorzat komponenseit (*mezőit*) így azokra alkalmazhatóvá válnak az ő típusának műveletei. Szelektorok minden, a direktszorzat típuskonstrukciós eszközt támogató nyelvben vannak, hiszen nélkülük a létrehozott típusok teljesen használhatatlanok lennének.

Értékadás

Szelektorokra minden nyelvben szükség van. Sokkal kevésbé egységes a kép az *értékadás* területén. Az értékadás természetes definíciója a direktszorzat típusoknál a komponensenkénti értékadás, amelyet a legtöbb programozási nyelv meg is enged. Ez a megoldás azonban problémákhoz vezethet, például ha valamelyik komponensre nem végezhető el az értékadás[3].

[2] A referencia típusokkal részletesebben az 5.6. fejezet foglalkozik.

[3] Ha például a kérdéses komponens egy fájl, akkor korántsem egyértelmű, hogy az értékadás megengedhető-e az adott komponensen.

A programozási nyelvek eltérő válaszokat adnak erre a kérdésre. A régebbi nyelvek általában mindenféle ellenőrzés nélkül, „vakon" végzik el az értékadást. Jó példa erre a C programozási nyelv, amelyben struktúrák értékadásakor az egyik struktúrához tartozó memóriaterület egyszerűen átmásolódik – bájtról bájtra – a másik struktúra területére. Ilyenkor a programozó felelőssége, hogy ne használjon értékadást, ha ez a viselkedés nem felel meg.

Sokkal biztonságosabb megoldást kínál az Ada programozási nyelv, amelyben a programozó kifejezetten megtilthatja az értékadás használatát a létrehozott típuson a **limited** kulcsszó használatával. Így a fordítóprogram hibát tud jelezni, ha szándéka ellenére mégis értékadást próbál végrehajtani az adott típuson. Ezután ha a direktszorzat típus valamelyik komponensére tiltott az értékadás, akkor az természetesen az egész direktszorzatra is az lesz.

Ennél általánosabb és jobban használható megoldást az kínálna, ha mi magunk definiálhatnánk az értékadás műveletét. Erre nyújt lehetőséget a C++ programozási nyelv. Itt a megoldás annak felismerésén múlik, hogy az értékadás nem feltétlenül kell, hogy a nyelvben utasítás legyen, hiszen az valójában az értékadásban részt vevő típusok egy művelete. Ez a megoldás először a C nyelvben jelent meg, de igazi előnyeit a C++ mutatta meg. Mivel a C++ megengedi a beépített operátorok túlterhelését (lásd részletesebben a 7.6.1. fejezetet), természetesen adódik a megoldás az értékadás problémájára, hiszen ha az alapértelmezett szemantika nem felel meg igényeinknek, azt bátran felüldefiniálhatjuk. Sőt az értékadás meg is tiltható egy típuson, ha az értékadó operátor definícióját olyan láthatósági területre helyezzük, amelyhez senki nem fér hozzá (lásd még a 4. fejezetet). Ennek a megoldásnak egy következménye, hogy már nem elégséges a C-ben használt memóriamásolásos megoldás az alapértelmezett értékadás esetén sem, hanem direktszorzat esetén az értékadást komponensenként kell elvégezni, hogy tényleg a helyes eredményt kapjuk.

Konstansok

Az értékadáshoz kapcsolódva meg kell vizsgálnunk még egy kérdést, hogy lehet e az adott nyelvben direktszorzat típusú konstansokat létrehozni. A konstans objektumokat a deklaráció helyén inicializálni kell, így használatukhoz szükséges, hogy egyrészt az értékadás elvégezhető legyen, másrészt meg tudjuk adni a kezdő értéket. Ennek az értéknek természetesen a direktszorzathoz kell tartoznia. Lehet tehát ez egy másik változó, ekkor azonban az a kérdés, hogy az hogyan inicializálódott, vagy szükségünk van egy olyan eszközre, amelynek segítségével létrehozhatunk a típushoz tartozó objektumokat.

Az objetumorientált nyelvekben ilyen eszközt jelentenek a konstruktorok, amelyek a kiinduló paraméterekből az adott típus egy objektumát hozzák létre. Ha a használt nyelv nem támogatja a konstruktorokat, de támogatja a dinamikus kiértékelést, akkor a konstruktorokhoz hasonló függvény alprogramokat is készíthetünk, amelyek visszatérési értéke a megfelelő objektum. A harmadik lehetőség, amely néhány programozási nyelvben adott, az aggregátumok használata. Az aggregátumok direktszorzat típusú literálok (lásd a 2.4. részt.), használatukra példát a következő oldalon látható Ada nyelvű programrészletben találhatunk.

```
type Komplex is record
    RE : Float := 0.0;
    IM : Float := 0.0;
end record;

type Pont is record
    X : Float := 0.0;
    Y : Float := 0.0;
end record;

I : constant Komplex:=(0.0,1.0);
Z : Komplex := I;
P : Pont;
begin
   Z := (42.0,42.0); P := (42.0,42.0); Z := Komplex'(1.0,1.0);
   P := (Y=>1.0, X=>1.0);
end;
```

Figyeljük meg, hogy az aggregátumok típusosak. Az *I* konstans inicializációjában, és a *Z* változóra vonatkozó első értékadásnál a programkörnyezet határozza meg a típusát, így bár a *P*-nek és *Z*-nek értékül adott aggregátumok formailag azonosak, típusuk különböző. Mindazonáltal helyesebb, és gyakran sokkal olvashatóbb, ha a *Z*-re vonatkozó második értékadásban látható módon a típust explicit kijelöljük. Az aggregátum elemei és a rekord mezői közötti megfeleltetés történhet pozíció szerint, amint az az első három értékadásban megfigyelhető, vagy név szerint, az utolsó értékadás mintájára.

Az aggregátumok használatát más nyelvek is támogatják, például a C és C++ nyelvekben kapcsos zárójelek – { és } – között adhatók meg a mezők értékei, de nincs lehetőségünk az aggregátum típusát meghatározni, és nem támogatott a név szerinti megfeleltetés sem.

Egyenlőségvizsgálat

Még az értékadásnál is összetettebb kérdés az egyenlőség vizsgálata két direktszorzathoz tartozó objektum esetén. Az alapértelmezett szemantika itt is a komponensenkénti összehasonlítás lenne, ám ennek elvégzése itt még az értékadásénál is bonyolultabb. Természetesen azok a problémák, amelyeket az értékadásnál említettünk, itt is felmerülnek, de azon felül jelentkezik még egy probléma, amelynek részleteivel a 6.1.3. részben foglalkozunk majd. Előljáróban itt csak annyit, hogy vannak olyan direktszorzatok, amelyeknél az egyszerű memóriamásolás megfelelne értékadásként, ám az összehasonlítást már nem lehetne elvégezni egyszerűen csak a két objektum memóriaterületeinek összehasonlításával. Ennek eredményeként a programozási nyelvek még megszorítóbban kezelik a direktszorzatok összehasonlítását. A C nyelv például az értékadást megengedi, de az egyenlőség vizsgálatát már nem. Azon nyelvekben pedig, ahol ez engedélyezett, az alapértelmezett viselkedés a komponensenkénti összehasonlítás, nem pedig a memóriaterületek összevetése.

Kicsomagolás

Különösen a Pascal-szerű[4] programozási nyelvekben gyakori még egy művelet a rekordokra, ez pedig a kicsomagolás. A művelet segítségével a bonyolult, minősített hivatkozások egyszerűsíthetők. Az alábbi, Modula-2-ből vett példa ezt illusztrálja:

```
TYPE DatumTipus = RECORD
   ev: EvTipus;
   ho: HoTipus;
   nap: NapTipus;
END;
TYPE EmberTipus = RECORD
   nev: NevTipus;
   szuletett: DatumTipus;
END;
VAR
   kedves: EmberTipus;
BEGIN
   WITH kedves.szuletett DO
      ev := 1974; ho := januar; nap := 22;
   END;
END;
```

A WITH utasítás segítségével a paraméterként megadott rekord mezőire minősítés nélkül hivatkozhatunk, ami nagyban megkönnyíti a program írását, de az olvashatóságot is javítja, különösen ha többszörösen egymásba ágyazott rekordokról van szó.

6.1.3. A direktszorzat ábrázolása

Vizsgáljuk meg, hogyan is tárolódik egy direktszorzat típus a memóriában. Bár a pontos részletek gyakorta nyelv-, sőt megvalósításfüggők, mégis megfigyelhetők bizonyos közös tulajdonságok.

A direktszorzat fizikai reprezentációja általában a komponensek megadásának sorrendjében tartalmazza a komponenseket, mégpedig általában az adott komponens típusának megfelelő fizikai formában. Ez utóbbi kitétel talán meglepő lehet, hiszen arra számítanánk, hogy a komponens reprezentációja nem változik meg attól, hogy egy összetettebb struktúrába építjük. Ez általában igaz is, hiszen a komponens típus műveleteit különben nem lehetne elkészíteni, az függne attól, hogy milyen direktszorzatba fogjuk a típust komponensként beépíteni.

Ennek ellenére például az Ada programozási nyelv lehetővé teszi, hogy bizonyos mezőknél meghatározzuk azok fizikai reprezentációját a rekordon belül. Hogyan lehetséges ez? A választ a szelektorfüggvények rejtik. Ezek a függvények képesek a tényleges ábrázolást a típus műveletei elől elrejteni, a művelet meghívásakor a szelektor függvény elvégzi a konverziót a direktszorzat típuson belül használt ábrázolásról a típus szokásos – a művelet által

[4] Igen népszerű és sikeres volt az elmúlt évtizedekben a Pascal programozási nyelv, így sok más nyelv vette át – kisebb vagy nagyobb részben – jelölésrendszerét, szintaktikáját. Ezeket a nyelveket szokás összefoglalva *Pascal-szerű* nyelveknek nevezni.

elvárt – reprezentációjára, majd a művelet végrehajtása után az eredményre a konverziót visszafelé is elvégzi.

Az alábbi példában (lásd [Ada 95]) egy program állapotleíró rekordjának típusát definiáljuk. A futtató rendszerben a program állapotát két gépi szó írja le, amelynek egyes bitjeihez vagy bitcsoportjaihoz különböző funkció/jelentés tartozik. A példában megadott rekorddefiníció ennek az állapotleírónak az absztrakt, erősen típusos megfelelője. Ehhez a rekordtípushoz egy ábrázolási záradékot adtunk, amelynek segítségével a rekord egyes mezőit pontosan az állapotleíró megfelelő bitjeire képeztük le. Ez a záradék így lehetővé teszi, hogy az állapotleírókat közvetlenül használjuk a programból anélkül, hogy a nyelv erős típusosságát vagy a meglévő típusok használatát fel kellene adnunk. Így a programunk nagyon hatékony lesz, ugyanakkor a fordítóprogram által adható összes segítséget igénybe vehetjük.

```
Word : constant := 4; -- Tárolási egység a bájt, egy gépi szó 4 bájt.

type State is (A,M,W,P);
type Mode is (Fix, Dec, Exp, Signif);
type Byte_Mask is array (0..7) of Boolean;
type State_Mask is array (State) of Boolean;
type Mode_Mask is array (Mode) of Boolean;

type Program_Status_Word is
   record
      System_Mask    : Byte_Mask;
      Protection_Key : Integer range 0 .. 3;
      Machine_State  : State_Mask;
      Interrupt_Cause : Interruption_Code;
      Ilc            : Integer range 0 .. 3;
      Cc             : Integer range 0 .. 3;
      Program_Mask   : Mode_Mask;
      Inst_Address   : Address;
   end record;

for Program_Status_Word use
   record
      System_Mask    at 0*Word range 0 .. 7;
                                   -- 8. és 9. bit nem használt
      Protection_Key at 0*Word range 10 .. 11;
      Machine_State  at 0*Word range 12 .. 15;
      Interrupt_Cause at 0*Word range 16 .. 31;
                                   -- második gépi szó
      Ilc            at 1*Word range 0 .. 1;
      Cc             at 1*Word range 2 .. 3;
      Program_Mask   at 1*Word range 4 .. 7;
      Inst_Address   at 1*Word range 8 .. 31;
   end record;
```

Nézzük meg például a *System_Mask* mezőt. A típusa *Byte_Mask*, ami logikai értékek tömbje. Bár a logikai értékek ábrázolása függ a konkrét megvalósítástól (lásd az 5.4.3. részt), általában azért legalább egy bájt szokott lenni, mert így célszerű. A megadott ábrázolási záradék viszont a nyolc elemet tartalmazó tömb számára összesen nyolc bitnyi helyet engedélyezett, így a fordító kénytelen egy biten ábrázolni az egyes elemeket[5]. Így az állapotleíró megfelelő bitjeit, amelyek valamiféle állapotjelzőként funkcionálnak, logikai értékekként érhetjük el, ami sokkal inkább megfelel a jelentésüknek. A rájuk való hivatkozás nem aritmetikai eszközökkel történik, hanem a tömb indexelésével, és megváltoztatásukra is a logikai operátorok használhatók.

A fenti eszköz, mint látható, meglehetősen erős, és igen mély lehetőségeket rejt, ha például valamilyen rendszerközeli programot (mondjuk egy eszközmeghajtót vagy kommunikációs protokollt) szeretnénk megvalósítani. Általában azonban nincs szükségünk arra, hogy bitről bitre megmondjuk a fordítónak, hogyan ábrázolja rekordjainkat. Ilyen esetekben a fordító saját hatáskörében dönt az ábrázolás megválasztásáról. Amint arról már szó esett, a komponensek fizikai sorrendje általában megegyezik megadásuk sorrendjével, de a komponensek között a fordító hagyhat ki helyet. Ezeket a hézagokat (angolul *padding*) arra használja, hogy az egyes komponensek megfelelő (néggyel, nyolccal, tizenhattal osztható) címekre kerüljenek. Ennek az az oka, hogy a legtöbb számítógép-architektúrán az ezektől eltérő címek elérése néha sokkal nehezebb, így az előállítandó kód sokkal bonyolultabb és lassabb lenne. Mint látható, a hézagok használata és mérete függhet a számítógép-architektúrától, a nyelvtől, sőt még a használt megvalósítástól is.

Most már érthető, miért nem lehet két direktszorzat egyenlőségét egyszerűen az általuk elfoglalt memóriaterületek összehasonlításával eldönteni. Ha ugyanis ezt tennénk, akkor abba beleszámítanának a komponensek közötti hézagok is, amelyeket viszont a fordító, hatékonysági okokból, nem inicializál, ott a memória korábbi tartalma található.

6.1.4. Egyedi nyelvi elemek

Az eddigiek, bár néha megemlítettünk konkrét nyelveket, általánosan vonatkoztak a direktszorzat típuskonstrukcióra. Több programozási nyelv vezet be azonban olyan specialitásokat, amelyek segítségével a direktszorzat konstrukció adott nyelvbeli megfelelője jobban, rugalmasabban vagy éppen biztonságosabban használhatóvá válik. Ebben a részben ezeket a specialitásokat tekintjük át.

Mezők kezdeti értéke

Az Ada programozási nyelv lehetővé teszi, hogy egy rekord típus megadásakor az egyes mezőknek kezdeti értéket adjunk. Ennek eredményeként minden ilyen típusú objektum megfelelő mezője létrehozáskor a megadott kezdeti értékre inicializálódik. Ez lehetővé teszi számunkra, hogy ha a rekordot egy absztrakt adattípus létrehozásához használjuk, akkor biztosítsuk a típusinvariáns teljesülését a létrejövő objektumokra (lásd még a 10. fejezet). A következő példában minden létrejövő dolgozó fizetése alapértelmezetten 100 000 forint.

[5] Az igazsághoz tartozik, hogy nem mindegyik fordító képes erre. A nyelv szabványa [Ada 95] megengedi, hogy egy megvalósítás korlátozza az ábrázolási záradékok használatát.

```
type Dolgozó_Típus is record
   Név        : Név_Típus;
   Cím        : Cím_Típus;
   Számlaszám : Számlaszám_Típus;
   Fizetés    : Fizetés_Típus := 100000;
end record;
```

CLU direktszorzat konstrukciók

A CLU több szempontból is speciálisan valósítja meg a direktszorzatot. Az egyik különlegesség az, hogy mint a nyelv összes típuskonstrukciós eszközéből, direktszorzatból is kettő van. Az egyik a *mutable* `record`, a másik az *immutable* `struct`. A másik különlegesség, hogy a szelektorok mint valódi eljárások, illetve függvények valósulnak meg, méghozzá a két konstrukciós eszközben, azok eltérő természetét figyelembe véve, némileg különbözően.

A CLU-ban a szelektor nem a konkrét mezőre vonatkozó referenciát visszaadó függvény, hanem egy alprogrampár, amely automatikusan létrejön minden direktszorzat típushoz. Ez az alprogrampár szétválasztja a szelektor kétféle alapvető feladatát, az adott mező kiolvasását, illetve megváltoztatását. Az alábbi példában egy CLU `record` definícióját és az egyik mezőjéhez tartozó szelektorműveleteket láthatjuk:

```
DolgozoTipus = record [ nev : NevTipus,
                        cim : CimTipus,
                        szamlaszam : SzamlaszamTipus
                        fizetes : FizetesTipus
                      ]
fonok : DolgozoTipus :=
          DolgozoTipus${ nev: "Fejes Elek",
                         cim: "Petrezselyem u. 12",
                         szamlaszam: "12345678-87654321",
                         fizetes: 800000
                       }
fizetes: FizetesTipus := DolgozoTipus$get_fizetes(fonok)
DolgozoTipus$set_fizetes(fonok,fizetes+100000)
```

A `get_fizetes` függvény végzi a fizetést tartalmazó mező kiolvasását, megváltoztatására pedig a `set_fizetes` eljárást használhatjuk. Ugyanígy a többi mezőhöz is létrejön a `get_`⟨*mezőnév*⟩ és `set_`⟨*mezőnév*⟩ pár. Mivel a műveleteket nem minősítjük a kérdéses rekord nevével, azt explicit át kell adnunk paraméterként, hogy a fordító el tudja dönteni, melyik objektumról van szó.

Struktúrák (`struct`) használata esetén a lekérdezés művelete nem változik, ellenben a mezők megváltoztatására a `replace_`⟨*mezőnév*⟩ eljárást használhatjuk. Az eltérő név azt a viselkedésbeli különbséget hivatott hangsúlyozni, hogy amíg a `set_` műveletek a meglévő objektum megfelelő mezőjének megváltoztatását végzik, addig a `replace_` műveletek egy új objektumot hoznak létre, amelynek minden mezője megegyezik a régi objektuméval, kivéve a megváltoztatott mezőt, mert az az új értéket tartalmazza.

Megjegyzés: a fenti eszközök mellett azért a nyelv támogatja a szelektorok „hagyományos" pontozásos jelölését és használatát, hiszen sok év megszokását nehéz lenne ilyen könnyen semmissé tenni. Ez a jelölésmód azonban csak úgynevezett „szintaktikus cukor", azaz csupán jelölésbeli könnyítés, amelyet a fordítóprogram automatikusan a megfelelő szelektorműveletekre cserél.

6.2. Unió típus

Tegyük fel, hogy most egy népességnyilvántartó rendszert készítünk. A férfiakról nyilván kell tartanunk a nevüket, címüket, katonai szolgálati idejüket. Erre a feladatra, mint láttuk, jó megoldás egy direktszorzat. A nőkről nyilván tartjuk a nevük és lakcímük mellett a leánykori nevüket is. Ezt is megtehetjük, egy másik direktszorzattal. Igen ám, de ekkor egy kis problémával kell szembenéznünk, mégpedig, hogy a lakosság férfiakból *és* nőkből áll. Hogyan lehet kifejezni, hogy egy lakos típusa vagy férfi vagy nő? A feladat megoldására használhatjuk az *unió* típuskonstrukciós eszközt.

Az unió típuskonstrukció sokkal ritkábban használt és ezért ritkábban is támogatott eszköz, nagyon sok nyelvben egyáltalán nincs nyelvi eszköz a kifejezésére (például BETA, Oberon, Python vagy a dBase nyelvcsalád). A modern, objektumorientált programozási nyelvekben feladatát gyakorlatilag kiválthatjuk az öröklődés és a többalakúság segítségével (lásd a 12. fejezet), így például a Java, az Eiffel vagy a Smalltalk nyelvek sem támogatják.

Mégis érdemes foglalkozni velük, egyrészt mivel a nem objektumorientált nyelvekben is szembesülhetünk ilyen jellegű feladatokkal, másrészt az egyes programozási nyelvek számos, sok tekintetben eltérő megoldást választottak megvalósítására, és ezen megoldások némelyike más jellegű problémák megoldásához is adhat érdekes és az öröklődésnél esetenként sokkal hatékonyabb eszközt. További előnye az öröklődés használatával szemben, hogy itt a típusok uniója képezhető utólag is, a típusok megváltoztatása nélkül. Az öröklődésen alapuló megoldásnál az unióban részt vevő típusoknak közös ősosztállyal kell rendelkezniük, így már a típusok elkészítésekor fel kell készülni erre az eshetőségre.

6.2.1. Típusértékhalmaz

Az unió típus megadása is az egyes komponens típusok, illetve a hozzájuk tartozó szelektorok megadásával történik, de ezek módja és szemantikája oly mértékben eltér az egyes konkrét nyelvekben, hogy azt inkább majd a nyelvi specialitások ismertetésekor tárgyaljuk. A típusértékhalmaz ebben az esetben a komponens típusok típusértékhalmazának uniója, azaz a formális definíció a következőképen alakul:

Legyenek $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ típusok és $T_1, T_2 \ldots T_n$ a hozzájuk tartozó típusértékhalmazok. A $\mathcal{T}$ típus a $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ típusok *uniója*, ha típusértékhalmaza $T = T_1 \cup T_2 \cup \cdots \cup T_n$.

6.2.2. Műveletek

A műveletekről is nehéz általánosságban beszélni, mivel itt is nagyon előtérben állnak a nyelvi sajátosságok. Az unió típuson is értelmezhető szelektorfüggvény, ennek szerepe azonban más, mint a direktszorzatnál.

Formálisan, az $s_i : \mathcal{T} \to \mathbb{L}$[6] függvény a $\mathcal{T}_i$ komponens típushoz tartozó szelektorfüggvény, ha $\forall t \in \mathcal{T} : (s_i(t) = \text{igaz} \iff t \in \mathcal{T}_i)$. Ebben a formában azonban nagyon kevés programozási nyelv valósítja meg a szelektorokat. A tényleges megoldások változatossága miatt érdemesebb tehát „unió jellegű" típuskonstrukciókról beszélni és ezeket egyenként megvizsgálni.

Kötött és szabad unió

Az unió típusok biztonságos használatával kapcsolatban felmerül az erős típusosság kérdése. Az erősen típusos programozási nyelvekben a fordítóprogram feladata azt ellenőrizni, hogy az egyes kifejezésekben az operandusok típusa megfelelő-e. Ezeket az ellenőrzéseket a fordítóprogram fordítási időben végzi, kihasználva, hogy a változók típusa a deklarációjukkor eldől és nem változik a program futása során, továbbá, hogy a változók csak a típusuknak megfelelő objektumokat tartalmazhatnak. Ez utóbbi feltételezés viszont nem állja meg a helyét, ha a nyelv támogatja az unió típuskonstrukciót.

Ha bármilyen típusból, korlátozás nélkül készíthető unió, akkor a fordítóprogram egy változóhivatkozásnál nem tudja eldönteni, hogy a hivatkozás érvényes-e, mivel az a változóban tárolt objektum tényleges típusától függ, ami egy futási idejű tulajdonság. Ez a jelenség hasonló az objektumorientált nyelvekben támogatott többalakúsághoz, ám annál sokkal kevésbé ellenőrizhető, mivel itt nincs semmilyen előírás a komponens típusokra nézve – nincs közös ősosztály – így azok felhasználhatóságával kapcsolatban semmilyen információval nem rendelkezünk.

A probléma kezelésére többféle megoldást használnak a különböző nyelvek. Konkrét megjelenési formájuk sokféle lehet, de alapvetően két csoportba sorolhatjuk ezeket a megoldásokat:

- Az egyik lehetőség a futási idejű típusellenőrzés. Amennyiben a nyelv futási időben ellenőrzi, hogy a változóban tárolt objektum *dinamikus típusa* megfelel-e a használat körülményeinek, akkor azt mondjuk, hogy a nyelvben *kötött unió* konstrukció van. Ennek ára azonban a futási időben végzett típusellenőrzés okozta sebességcsökkenés.
- A másik megoldás, hogy a nyelv nem csinál semmit, a felhasználás helyességét a programozóra bízza. Ez a megoldás a futási idejű hatékonyságot tartja inkább szem előtt. Az olyan unió konstrukciót, amely semmilyen típusellenőrzést nem végez, *szabad uniónak* nevezzük.

[6] $\mathbb{L}$ a logikai értékek halmaza, azaz $\mathbb{L} = \{\text{igaz}, \text{hamis}\}$

6.2.3. Unió jellegű típuskonstrukciók

Az ALGOL 68 union típus

Az ALGOL 68 nyelv talán az egyetlen, amely a formális definíciónak teljesen megfelelően valósította meg az unió típuskonstrukciót. A *NumberType* típust az alábbi példában mint az egész, illetve valós számok unióját definiáljuk.

```
mode union (int, real) NumberType;
NumberType number;
...
number := 3;
number := 3.14;
...
case number in
    (int i): print(("integer", i)),
    (real r): print(("real", r))
esac
```

A példában szereplő mindkét értékadás helyes. Kiolvasáskor a **case** utasítás egy speciális változatával határozhatjuk meg az éppen aktuális érték típusát. Ez a megoldás szemantikailag megfelel a formális definícióban adott szelektor-meghatározásnak.

A fenti kiolvasási mód és az a tény, hogy az unió egy önálló típus, garantálja, hogy az unióra mindig típushelyesen hivatkozunk, továbbá ezt a típushelyességet a fordítóprogram képes ellenőrizni. Ha ugyanis például egész számként próbáljuk felhasználni a *number* változót, akkor ez a **case** szerkezet egész számokhoz tartozó ágában tehető meg. Ide azonban csak akkor kerülhet a vezérlés, ha a *number* változó valóban egy egész értéket tartalmaz. Ha pedig közvetlenül használjuk a *number* változót, akkor annak típusa a fenti unió és az ALGOL 68 erős típusossága miatt nem állhat a komponens típusok helyén.

A C union típusa

Az C nyelv **union** típuskonstrukciója sokat merített az ALGOL 68-ból, mégis számottevő különbségek akadnak. Megadása a következő formában történik:

```
typedef union {
    FerfiTipus    ferfi;
    NoTipus       no;
} EmberTipus;
```

A példában az *EmberTipus* típust úgy adjuk meg, mint a *FerfiTipus* és a *NoTipus* típusok unióját. Egy *EmberTipus* típusú objektum a memóriában annyi helyet fog lefoglalni, amennyit a *FerfiTipus* és *NoTipus* típusok közül a nagyobbik foglalna el, és az egyes komponensek egymást átlapolva helyezkednek el ezen a területen.

A *ferfi* és *no* szelektorok szerepe nem az, hogy segítségükkel meghatározhassuk, hogy egy adott *EmberTipus* típusú objektum vajon férfi avagy nő-e. A C nyelvben a **union** típusok szelektorai arra valók, hogy az ilyen típusú objektum által lefoglalt memóriaterületet a

szelektor típusának megfelelően értelmezve olvassuk ki. A nyelv nem ad semmilyen támogatást annak eldöntésére, hogy az adott területen található objektum tényleg az adott típushoz tartozik-e. Ez a hiányosság nagyon veszélyessé teszi a **union**-ok használatát. Megfelelő körültekintés nélküli használatuk nehezen felderíthető, futási idejű hibák kialakulásához vezethet. Az alábbiakban erre láthatunk példát:

```
#include <stdio.h>

typedef char NevTipus[100];
typedef struct {
    NevTipus nev;
} FerfiTipus;
typedef struct {
    NevTipus nev;
    NevTipus leanykori_nev;
} NoTipus;
typedef union {
    FerfiTipus ferfi;
    NoTipus no;
} EmberTipus;

int main()
{
    FerfiTipus f = {"Gipsz Jakab"};
    EmberTipus e;
    e.ferfi = f;
    printf("nev: %s\nleanykori: %s\n",e.no.nev, e.no.leanykori_nev);
    return 0;
}
```

A fenti program hiba vagy figyelmeztetés nélkül lefordítható, ám lefuttatva jobb esetben a leánykori név helyén valamilyen olvashatatlan dolgot látunk, de az is előfordulhat, hogy a program hibaüzenet kíséretében leáll.

A hiba kivédésére sem a fordító, sem a nyelv nem kínál közvetlen eszközt, így a **union**-ok biztonságos használatáról a programozónak kell gondoskodnia. Az alábbi példa erre mutat egy lehetséges megoldást:

```
typedef enum { FERFI, NO } NemTipus;
typedef struct {
    NemTipus nem;
    union {
        FerfiTipus ferfi;
        NoTipus no;
    } adat;
} EmberTipus;
```

Ebben a példában mi magunk valósítjuk meg a szelektorokat (azok eredeti értelmezésében) az emberek nemének megadásával. Ehhez azonban uniónkat egy struktúrába kell ágyaznunk, tartalmát csak a struktúra *adat* szelektorával érhetjük el. Így a korábbi *e.no.nev* jelölés helyett az *e.adat.no.nev*-et kell használnunk. Vegyük észre továbbá, hogy még mindig a programozó felelőssége, hogy a kiolvasás előtt megvizsgálja az illető ember nemét, a nyelv még futási idejű védelmet sem nyújt a hibás kiolvasások észlelésére. Megengedett továbbá egy ember nemének megváltoztatása, anélkül, hogy a struktúra többi részét is inicializálnánk, így még a nem megvizsgálása sem jelent teljes védelmet a futási idejű hibák ellen.

Mivel a **union** komponensei lehetnek struktúrák is, amelyeken az egyenlőségvizsgálat nem megengedett művelet, így az nem megengedett magán a **union**-on sem. Ugyanakkor az értékadás még mindig elvégezhető a memóriaterület lemásolásával.

A C++ union típusa

A C++ nyelv **union** típusa nagyon sok mindenben megegyezik a C nyelvével, ami nem is meglepő, mivel a C++ elkészítésekor fontos szempont volt a C-vel való minél teljesebb kompatibilitás. Ez a terhes örökség azonban néhány érdekes kompromisszum megkötését tette szükségessé.

A C++ nyelvben a struktúrák (**struct**) az osztályok (**class** – lásd a 12. fejezetet) változatai, gyakorlatilag egyenértékűek. Az osztályoknak a C++ nyelvben írhatunk konstruktor metódusokat, amelyek meghívásáról a fordító gondoskodik minden egyes objektum létrejöttekor. A kostruktorok ilyen erős támogatása azonban problémát okoz, ha a struktúrákból uniót szeretnénk létrehozni. Ekkor ugyanis egyazon memóriaterületen több, egymást átfedő objektumot kellene létrehozni, így az egyik konstruktorának lefutása felülírná a másik eredményét. Még ha lenne is egy kitüntetett komponens, amelyiknek a konstruktora lefutna ilyenkor, mivel a **union** nem hordozza az éppen benne tárolt objektum típusára vonatkozó információt, bármelyik szelektoron keresztül kiolvashatjuk a memóriaterület tartalmát.

Ezt az ellentmondást nem könnyű feloldani, és bizonyos kompromisszumok nélkül nem is lehetséges. A C++ nyelvben azt a megszorítást tették, hogy egy olyan osztály, amelynek nemtriviális konstruktora van, nem lehet egy **union** komponense. Egy osztálynak akkor van nemtriviális konstruktora, ha egy olyan osztályból származtattuk, amelynek már van ilyen, vagy mi magunk írtunk neki konstruktort. Miért jelent megoldást ez a megszorítás? Az alapértelmezett konstruktor nem csinál semmit az objektum memóriaterületével, így az ilyen konstruktorral rendelkező típusoknak annak veszélye nélkül képezhetjük az unióját, hogy a konstruktorok felülírnák egymás eredményeit. Mivel a C nyelvben még nem volt nyelvi támogatás a konstruktorokhoz, a fenti viselkedés pontosan megfelel annak, ahogy a C nyelvben viselkednek a **union**-ok, a megoldás tehát kielégíti a visszafelé kompatibilitás követelményét is.

A fenti megoldás igen erős megszorításokat tesz. Szerencsére a C++ nyelvben már sokkal jobb eszközök adottak az unió típuskonstrukció megvalósítására. Az egyik ilyen eszköz nem más, mint az *öröklődés* (részleteket lásd a 12. fejezetben). Az öröklődés segítségével könnyen képezhetjük típusok unióját. A többalakúság használata lehetővé teszi az objektu-

mok tényleges típusának meghatározását, az *instanceof* operátor segítségével pedig még az unió típuskonstrukciónál bevezetett szelektor is megvalósítható.

A CLU címkézett uniója

A CLU nyelvben az ALGOL 68-hoz hasonló tiszta formában valósul meg az unió típuskonstrukció. A CLU filozófiájához igazodva két változata is létezik, a *mutable* `variant` és az *immutable* `oneof`. Mindkettő úgynevezett *címkézett unió*, ami azt jelenti, hogy minden komponens típushoz tartozik egy címke. Az objektum értékével együtt tárolódik a típusához rendelt címke is, így megvalósítható a formális definícióban megadott szelektor művelet. Az alábbi példa a szokásos `EmberTipus` megvalósítása CLU-ban, a komponens típusok részletei nélkül:

```
EmberTipus = variant [ ferfi: FerfiTipus, no: NoTipus]
n  : EmberTipus := EmberTipus$make_no(egy_no)
f  : EmberTipus := EmberTipus$make_ferfi(egy_ferfi)
e  : EmberTipus := n
ff : FerfiTipus
nn : NoTipus

tagcase e
   tag ferfi :
      ff:=EmberTipus$value_ferfi(e)
      Ember$Tipus$change_no(e, egy_no)
   tag no (no_tmp : NoTipus) :
      nn := no_tmp
end

if EmberTipus$is_ferfi(e) then
   ff := EmberTipus$value_ferfi(e)
   EmberTipus$change_no(e,egy_no)
else
   nn := EmberTipus$value_no(e)
end
```

A deklarációk után található `tagcase` szerkezet szemantikailag teljesen megegyezik az utána levő `if`-es struktúrával. Mint látható, a `tagcase` utasítás hasonlóan működik, mint az ALGOL 68-ban, ám ahhoz képest enyhítéseket tartalmaz. Nem kötelező például a női ágon látható *ágváltozó* megadása, használhatók helyette a `value_`⟨*címkenév*⟩ műveletek, amelyek automatikusan definiálódnak az összes megadott címkéhez. Azt, hogy milyen típusú a `variant`-ban tárolt érték, megtudhatjuk az `is_`⟨*címkenév*⟩ szelektor műveletek segítségével, amelyek szintén implicit definiáltak. Van még egy harmadik műveletcsoport is, amely minden `variant` típuskonstrukcióhoz létrejön, mégpedig a `change_`⟨*címkenév*⟩ műveletek, melyek segítségével megváltoztathatjuk az objektum tartalmát. Ezek a műveletek természetesen a címkét is megfelelően módosítják.

Az ALGOL 68-cal összehasonlítva azt mondhatjuk, hogy a kötöttségek lazultak, ami futási idejű hibákat eredményezhet, mivel például az alábbi hívás szintaktikailag helyes, és csak futási időben deríthető ki, hogy az objektum nem egy férfi adatait tartalmazza.

```
ff := EmberTipus$value_ferfi(n)
```

Ezzel szemben a címkék bevezetésével a komponensek jelentése elvonatkoztatható a típusuktól és elvileg lehetséges ugyanazon típust többször is szerepeltetni az unióban.

Az unió típuskonstrukció immutable változata, a `oneof`, mindössze annyiban tér el a `variant`-tól, hogy nem rendelkezik a `change_`⟨*címkenév*⟩ megváltoztató műveletekkel.

Variáns rekord

A Pascal típusú nyelvek megoldása az unió jellegű típuskonstrukcióra a *variáns rekord*. Az elnevezés már sugallja, hogy itt a direktszorzatot megvalósító rekord konstrukció kiterjesztéséről van szó. Logikáját tekintve hasonlít ahhoz a megoldáshoz, amelyet a C **union**-ok esetében a típusinformáció megőrzésére javasoltunk. Az alábbi, Modula-2-ből vett példa az embereket reprezentáló típus egy lehetséges megvalósítása:

```
TYPE NemTipus = (ferfi, no);
TYPE FerfiTipus = RECORD
   nev: NevTipus;
   voltkatona: BOOLEAN;
END;
TYPE NoTipus = RECORD
   nev: NevTipus;
   leanykori_nev: NevTipus;
END;
TYPE EmberTipus = RECORD
   CASE nem: NemTipus OF
   | ferfi:    f: FerfiTipus;
   | no:       n: NoTipus;
   END;
END;
```

A megoldás lényege, hogy a rekord egyes mezőit kitüntetjük és a rekord szerkezete ezen mezők értékének függvényében változhat. A fenti megoldás a C-s példa közvetlen átírása, ám már ennek is van egy nagy előnye a C-beli megoldással szemben, mégpedig az, hogy a rendszer – futási időben – ellenőrzi, hogy a *nem* mező értéke férfi-e, ha férfiként hivatkozunk rá és nő-e, ha nőként. Ha ez a feltétel nem teljesül, futási idejű hibát kapunk.

A hiba itt is futási időben jelentkezik, de ezen sajnos javítani nem lehet, hiszen a variáns rekord tényleges típusa függhet például a felhasználó inputjától is, így azt fordítási időben ellenőrizni nem lehet. Ennek a megoldásnak mégis előnye, hogy a hiba fellépte azonnali és garantált míg a C-beli megoldásnál a hiba elsikkadhat, esetleg jóval később derülhet ki, így megkeresése is sokkal nehezebb.

A fentinél egyszerűbb megoldást is kaphatunk, ha nincs szükségünk külön férfi és nő típusokra:

```
TYPE NemTipus = (ferfi, no);
TYPE EmberTipus = RECORD
   nev: NevTipus;
   CASE nem: NemTipus OF
   | ferfi:     voltkatona: BOOLEAN;
   | no:        leanykori_nev: NevTipus;
   END;
END;
```

Itt a közös részeket összevontuk és csak a különbségeket tettük a változó részbe. Ez a megoldás azonban már nem a klasszikus értelemben vett unió, hiszen az unióban részt vevő komponensek csak rejtetten adottak, nincsenek nevesítve önálló típusként.

A variáns részt vezérlő mező a használat szempontjából ugyanolyan mezője a rekordnak, mint bármelyik másik, ám értékének megváltozása a rekord szerkezetének megváltozását vonhatja maga után. Mivel az egyes variánsok fizikai ábrázolása itt is átfedi egymást, a vezérlő mező megváltoztatásakor könnyen lehet, hogy a mezők tartalma inkonzisztens lesz. Sajnos az ilyen hibák kivédéséhez a nyelv nem nyújt támogatást.

Egyetlen megszorítás vonatkozik a vezérlő mezőre, hogy típusa valamilyen diszkrét típus kell, hogy legyen.

Az Ada diszkriminánsos rekordjai

A variáns rekordok továbbfejlesztett változatait találjuk meg az Ada programozási nyelvben. Itt a rekordok paraméterezhetők, úgynevezett *diszkriminánssal* láthatók el, és a rekord szerkezete, ábrázolása függhet a diszkrimináns(ok) értékétől. Az alábbi példa a korábbi férfi–nő probléma megoldása:

```
type NemTípus is (férfi, nő);
type EmberTípus(nem: NemTípus) is record
   név: NévTípus;
   case nem is
      when férfi   => volt_katona: Boolean;
      when nő      => leánykori_név: NévTípus;
   end case;
end record;
```

Ez minimális formai különbségektől eltekintve nem különbözik a Modula-2 megoldástól, a viselkedése azonban eltér. Ha ebben a formában definiáljuk az *EmberTípus*-t, akkor minden egyes objektum létrehozásakor meg kell mondanunk, hogy milyen nemű emberre gondoltunk, és ezt a későbbiek során megváltoztatni nem lehet. Nem fordulhat tehát elő, hogy a rekord bizonyos részei inicializálatlanok lesznek. Így a típus csak részben működik unióként, mivel a típusértékhalmaza ugyan unió jellegű, de minden objektum létrehozása-

kor megszorítjuk ezt a típusértékhalmazt valamely komponens típusra, az uniónak implicit módon egy altípusát hozzuk létre.

Mire jó akkor mégis egy ilyen konstrukció? Alprogramok formális paraméterének lehet ilyen diszkriminánsos rekord paramétere, és itt a diszkrimináns értékét nem kell megadni. Írhatunk tehát olyan alprogramot, amely egyaránt működik férfiakra és nőkre is, például a név kiíratása lehet egy ilyen művelet. Természetesen egy ilyen művelet törzsében a variáns rész mezőit csak nagy körültekintéssel szabad használni, hiszen a fordító semmilyen segítséget nem tud adni az esetleges hibák kiszűréséhez, azok futási időben jelentkeznek.

Előfordulhat azonban, hogy valóban unióra van szükségünk, azaz az objektumban szeretnénk a diszkrimináns értékét megváltoztatni. Erre ad lehetőséget az alapértelmezett diszkrimináns. Az alábbiakban úgy változtatjuk meg az *EmberTípus*-t, hogy lehessen a nem megadása nélkül is ember objektumot létrehozni:

```
type NemTípus is (férfi, nő);
type EmberTípus(nem: NemTípus:=férfi) is record
   név      : NévTípus;
   case nem is
      when férfi => volt_katona  : Boolean;
      when nő    => leánykori_név : NévTípus;
   end case;
end record;

F: EmberTípus(férfi);
N: EmberTípus(nő);
E: EmberTípus;
begin
   E := F;
   E := N;
   N := F; -- Itt a fordító figyelmeztet, hogy a jobb oldali kifejezésben a
           -- diszkrimináns értéke nem megfelelő és futási időben Constraint_Error
           -- kivétel fog kiváltódni.
end;
```

Mint láthatjuk háromféle módon hozhatunk létre *EmberTípus* típusú objektumot. *F* létrehozásakor a lefoglalt memória mennyisége akkora, amelybe egy férfi adatai pontosan belefférnek. *N* létrehozásakor egy nő adatainak tárolására elegendő memória foglalódik le. Ezekben az esetekben már csak ezért sem lehetséges a diszkrimináns megváltoztatása, mivel akkor esetleg utólag meg kellene növelni az objektum helyfoglalását.

Az *E* létrehozásakor is egy férfit leíró objektum keletkezik a fordító azonban annyi memóriát foglal le, amennyi a diszkrimináns bármely értéke mellett elegendő az objektum tárolására, ezért a diszkrimináns megváltoztatható. Hogy elkerüljük az objektum inkonzisztens állapotba hozását, a diszkrimináns értéke csak az egész objektumra vonatkozó értékadással változtatható meg.

A diszkriminánsos rekord segítségével más módon is szabályozhatjuk rekordunk szerkezetét. Nem csak a mezők összetételét változtathatjuk meg, hanem a tömb típusú mezők

méretét is. Az Ada nyelv beépítetten tartalmazza a *String* típust, ám annak hossza rögzített. Az alábbi példában egy változó hosszúságú szöveg típust valósítunk meg. A szöveg tartalmát egy *String* tárolja, ám annak tartalma csak az első *hossz* karakteren érdekes. A *Szöveg* típus diszkriminánsa a szöveg lehetséges legnagyobb hossza:

```
type Szöveg(maxhossz: Natural) is record
   hossz    : Natural := 0;
   tartalom : String(1..maxhossz) := (others => ' ');
end record;
```

Mint látható, itt is egy uniót hoztunk létre, az *1*, *2*, ..., *Natural'LAST*[7] hosszúságú szövegek típusainak unióját. Fontos megjegyezni, hogy itt is adhatunk alapértelmezett értékét a diszkriminánsnak, ám arra továbbra is vonatkozik az a szabály, hogy ha az alapértelmezett diszkrimináns használatával hozunk létre egy objektumot, akkor annak a fordító akkora memóriaterületet foglal le, hogy abban a diszkrimináns tetszőleges értéke mellett elférjen az objektum. Ez pedig a fenti példában könnyen jelentheti azt, hogy fizikailag nem lehetséges az alapértelmezett diszkrimináns kihasználása, mivel az objektum nem férne el a memóriában.

6.3. Iterált típus

Tegyük fel, hogy egy iskolai osztály névsorát szeretnénk kezelni. Az előző részekben láttuk, hogyan írhatunk le egyetlen embert – tanulót – a feladat természetének megfelelően. Most az a feladat, hogy az egész osztályt írjuk le, mint az oda járó tanulók összességét. Vagyis olyan típuskonstrukciós megoldásra van szükségünk, amelynek segítségével egy meglévő típus elemeiből sorozatokat vagy halmazokat képezhetünk. Az ilyen típuskonstrukciós eszközöket *iteráltaknak* nevezzük, és mint azt látni fogjuk, számtalan változatuk létezik.

Valóban, talán a legszélesebb spektrumot felölelő típuskonstrukciós megoldás az iterált. Az eddigiektől eltérően ez a megoldás homogén, azaz nem több meglévő típusból képezünk egy újat, hanem csupán egyből. Eltérés még az is, hogy az egyes programozási nyelvek nem ritkán egynél több megoldást is kínálnak, azaz az iterált típuskonstrukciós eszközök több változatát is támogatják.

Számtalan előfordulása közül néhány:

- Vektor
- Többdimenziós tömb
- Lista
- Hasítótábla
- Sor
- Fájl
- Halmaz

[7] A *Natural'LAST* attribútum jelöli az Ada programozási nyelvben a *Natural* típus legnagyobb elemét. Ez az érték függhet a megvalósítástól, de általában legalább $2^{31} - 1$.

6.4. Vektor

Tegyük fel, hogy az osztály hallgatóit sorszámoztuk és leggyakrabban e sorszám segítségével hivatkozunk rájuk. Olyan megoldásra van szükség, amelyben a sorszám alapján történő keresés hatékonyan elvégezhető. Ez az eszköz a *vektor*. Talán az összes iterált típus közül ez a leggyakoribb és legegyszerűbb. A programozási nyelvek túlnyomó többsége támogatja.

6.4.1. Típusértékhalmaz

Egy vektor típus definiálásakor meg kell adnunk az elemtípust és az indexhalmazt is. Míg az első megadása lényegében a megfelelő típus megnevezését jelenti, így minimális szintaktikai eltérésektől eltekintve a különböző nyelvekben nagyjából azonos, addig az indexhalmaz megadása már nagyobb eltéréseket mutat. A BASIC nyelv bizonyos változataiban például akár el is hagyható, és ebben az esetben implicit a $0 \ldots 10$ intervallum lesz az indexhalmaz. Ha ettől eltérő méretű tömbökre van szükségünk, azt a tömb legnagyobb indexének megadásával jelezhetjük. A tömb elemei továbbra is nullától kezdve indexelődnek.

A C jellegű[8] nyelvekben elemszámuk megadásával definiálhatjuk a vektorokat. Egy n elemű vektor indexhalmaza a $0 \ldots (n-1)$ egészintervallum lesz. Az alábbi példában a létrehozott *OsztalyTipus* 42 rekordból áll amelyek az *EmberTipus*-hoz tartoznak:

typedef *EmberTipus OsztalyTipus[42];*

A Pascal típusú programozási nyelvekben általában egy tetszőleges diszkrét típus egy tetszőleges, nem üres intervallumát választhatjuk indexhalmaznak. Az alábbi példák az Adában készültek, de a többi nyelvben sem mutatkozik lényegi eltérés a vektorok megadásában:

type *OsztályTípus* **is array** *(1..42)* **of** *EmberTípus;*

type *TörpökTípus* **is** *(Szende, Szundi, Hapci, Kuka, Morgó, Tudor, Vidor);*

type *HéttörpeTípus* **is array** *(TörpökTípus)* **of** *EmberTípus;*

type *ÖttörpeTípus* **is array** *(TörpökTípus* **range** *Szundi..Tudor)* **of** *EmberTípus;*

Vannak programozási nyelvek a két véglet között is. Például az ALGOL 68 nyelvben az indexhalmaz kötelezően az egész típusnak valamely részintervalluma, de nincs megkötve, hogy az intervallum alsó határa a 0 kell, hogy legyen. A CLU vektor konstrukciói[9] még érdekesebbek, ezek ugyanis menet közben változtathatják az indexhalmazukat. Annyi megkötés van, hogy az indexhalmaz egész számok egy intervalluma, ám a vektor indexhalmaza futás közben kiterjeszthető.

[8] A Pascalhoz hasonlóan a C nyelvet is több más programozási nyelv vette alapul a nyelv szintaxisához. Ilyen nyelvek például a Java vagy a C++.

[9] Amint azt már sejthetjük, a CLU-ban a vektornak is van mutable és immutable változata. Az előbbit `array`-nek, az utóbbit pedig `sequence`-nek nevezi.

A fentieket összefoglalva a vektor formális definíciója a következő: legyen $\mathcal{H}$ egy rendezett véges halmaz (az *indexhalmaz* vagy *indextípus*) és $\mathcal{T}_0$ egy tetszőleges típus (az *elemtípus*). A $\mathcal{V} = \text{vect}(\mathcal{H}, \mathcal{T}_0)$ vektor típus típusértékhalmaza: $\{v : \mathcal{H} \mapsto \mathcal{T}_0\}$, azaz a $v \in \mathcal{V}$ vektor egy leképezés az indexhalmazról az elemtípusra.

Figyeljük meg, hogy az indextartomány meghatározása háromféle módon történik a különböző nyelvekben. A C/C++ vagy a Pascal esetében a megadás teljesen statikus, a típus definíciójában rögzített. Az Ada megengedi, hogy csak az objektum létrehozásakor határozzuk meg az indexhatárokat (határozatlan indexhatárú tömb), míg a CLU vagy az Eiffel megengedi az indexhatárok teljesen dinamikus változtatását a program futása során.

6.4.2. Műveletek

Tekintsük most át a vektor konstrukció műveleteit. Ezek a műveletek meglehetősen hasonlóak a különféle nyelvekben, bár egy-két különlegességgel azért találkozhatunk.

Szelekció

A legfontosabb művelet ebben az esetben is a *szelekció*. Azt gondolnánk, hogy ez most egy tetszőleges $v \in \mathcal{V}$ és $i \in \mathcal{H}$ esetén a $v(i)$ helyettesítést jelenti. Ez azonban túlzott leegyszerűsítése a szelekció műveletének, a $v(i)$-vel jelölt objektum ugyanis balérték, azaz szerepelhet értékadás bal oldalán is, rajta keresztül megváltoztatható a v vektor. Valójában azt mondhatjuk, hogy a vektor nem is $\mathcal{T}_0$ típusú elemekre képező függvény, hanem $\mathcal{T}_0$-ra vonatkozó *referencia* típusúakra (a referencia típusokról lásd még az 5.6. fejezetet). Ezt a tényt azonban a legtöbb programozási nyelv formális megadásakor nem szokták következetesen végigvezetni, mivel az nagyon bonyolulttá tenné a nyelvtan leírását. Az egyik kivétel az ALGOL 68 programozási nyelv definíciója, amelyben a változók valamilyen típusra vonatkozó referencia típusúak, míg az objektumok, amelyek értékül adhatók nekik, a hivatkozott típushoz kell, hogy tartozzanak (bővebben lásd [Tan 76]).

A különféle programozási nyelvek a vektorok esetében a szelekciót meglehetősen hasonlóan jelölik. Általában a vektor neve után, szögletes zárójelek között adják meg az indexkifejezést (`v[i]`). Eltér ettől az Ada programozási nyelv, amely a kerek zárójeleket használja itt is.

Értékadás

A direktszorzathoz hasonlóan jóval ellentmondásosabb művelet az értékadás. Sok programozási nyelv nem is foglalkozik vele, mivel egy egyszerű ciklussal át lehet másolni az egyik vektort egy másikba. A Pascal típusú nyelvek viszont általában megengedik az értékadást a vektortípusokon is.

6.4.3. Nyelvi specialitások

A vektor típusokat a különböző programozási nyelvek számtalan érdekes lehetőséggel ruházták fel. Az alábbiakban ezek közül próbáljuk meg a legfontosabbakat összegyűjteni.

C és C++ vektorok

A C programozási nyelvben a mutatók és a tömbök igen közeli rokonságban állnak egymással (egy tömb reprezentációja nem más, mint az első elemére hivatkozó mutató) és a viselkedésük is nagyon hasonló, bár azért akadnak apróbb eltérések. A szelekció művelete például úgy valósul meg, hogy a *v[i]* kifejezés a **(v+i)* kifejezéssel helyettesítődik, ahol a + az egészek és típusos mutatók közötti összeadást jelenti[10], a * pedig a hivatkozás-feloldás operátora. Ennek két érdekes következménye is van. Az egyik, hogy a szelekció művelete használható a mutató típusokra is, hiszen a fenti átiratban a kifejezés tetszőleges mutató típushoz tartozó *v*-vel érvényes. A másik érdekesség, hogy a C nyelvben az indexelés mint művelet „kommutatív", azaz a *v[i]*, illetve *i[v]* kifejezések ugyanazt jelentik, mivel a mutatóaritmetikában az összeadás kommutatív.

A mutatók és tömbök ezen rokonságának következtében alprogramjainknak tetszőleges hosszúságú tömböket átadhatunk paraméterként, hiszen valójában mindig az első elemre hivatkozó mutató adódik át. Így az Ada határozatlan indexhatárú tömbjeihez hasonlóan általános célú alprogramokat írhatunk, ám fontos eltérés, hogy a vektor nem ismeri saját méretét, így a hosszát szintén át kell adni paraméterként.

Az, hogy a C és így a C++ nyelv is operátornak tekinti az indexelést, ez utóbbiban még egy érdekességet eredményezett. Mivel a C++ nyelv támogatja az operátor-túlterhelést, lehetővé teszi, hogy saját típusainkra az indexelés (*[ ]*) operátort is definiáljuk.

Java vektorok

A Java nyelv vektorai formájukat tekintve nagyon hasonlítanak a C, illetve C++ nyelvek vektoraira, ám van néhány lényeges eltérés. A Javában nem deklarálható nevesített vektor típus, de minden *T* típushoz tartozik egy *T[ ]*-vel jelölt vektor típus. Ez a vektor típus lényegében határozatlan indexhatárú. Mivel az indextípus itt is kötelezően nem negatív egész és az indextartomány alsó határa mindig 0, ez valójában azt jelenti, hogy a vektor típus hossza nem kötött, hanem azt a tényleges tömb objektum létrehozásakor kell megadni, hasonlóan az Ada határozatlan indexhatárú tömbjeihez. Hasonlóan az Adához és eltérően a C és C++ nyelvektől, a Java vektor ismeri a hosszát és az lekérdezhető tőle.

Vektorok az Ada nyelvben

Az Ada programozási nyelvben a tömbtípusok megadásakor eltekinthetünk az indexhatárok rögzítésétől, elegendő az index típust megadnunk. Az így létrehozott – úgynevezett

[10] A C nyelvben igen erőteljes mutatóaritmetikai eszköztár áll a programozó rendelkezésére, melyről részletesebben az 5.6. fejezetben található ismertetés.

határozatlan indexhatárú – tömbtípushoz tartozó tömbök indexhatárait deklarációjuk helyén kell megadnunk. Az ilyen típus lényegében az uniója az összes lehetséges indexhatár-kombinációval létrehozható tömbtípusoknak. Amikor egy változót deklarálunk ezzel a típussal, megadva a kívánt indexhatárokat, a típusnak egy *implicit altípusát* hozzuk létre.

Bár változót nem deklarálhatunk ilyen típussal az indexhatárok megadása nélkül, alprogramoknak formális paramétert igen. Ez lehetővé teszi, hogy alprogramjainkat minél általánosabban készíthessük el. Az ilyen típusokhoz definiálható mutató típus is.

```
type ValósTömb is array (Integer range <>) of Float;

function MaximumKeresés(T: in ValósTömb) return Integer is
MAXHELY: Integer := T'First;
begin
    for I in T'Range loop
        if T(I) > T(MAXHELY) then
            MAXHELY := I;
        end if;
    end loop;
    return MAXHELY;
end MaximumKeresés;

VT: ValósTömb(1..4) := (5.0, 6.7, 2.11, 1.0123);
MAX: Integer := MaximumKeresés(VT);
```

A fenti példában megfigyelhetünk néhány egyéb érdekességet is. Ilyen például a *MAXHELY* változó inicializációjában látható *T'First*. Ez a kifejezés az Ada tömbtípusának egyik speciális művelete (*attribútuma*), amely megadja a tömb indextartományának alsó határát. A *Last* attribútum a felső határt adja meg, a későbbiekben, a ciklus fejében látható *Range* attribútum pedig magát az indextartományt, azaz a *T'First..T'Last* intervallumot jelöli. Ezen felül használható még a *Length* attribútum is, amely megadja a tömb elemeinek számát, azaz az indexintervallum hosszát. Ezen attribútumok lehetővé teszik, hogy programjainkat minél általánosabban fogalmazhassuk meg. Segítségükkel a változásokra kevésbé érzékeny és általánosan használható kódot írhatunk.

Az Ada programozási nyelv nemcsak megengedi a tömbök közötti értékadást, de meglehetősen liberális módon kezeli az indexeket is. Egy határozatlan indexhatárú tömb típus különböző altípusai között megengedett az értékadás, amennyiben az adott altípusok elemszáma megegyezik.

Tömbjeinknek kezdőértéket is adhatunk. A fenti és a következő kódrészletekben egyaránt láthatunk erre példát. Az *aggregát jelölés* egyaránt megengedi a fenti *pozíció szerinti* megfeleltetést, ahol az elemek az aggregátumon belüli sorrendjükben másolódnak a tömbbe, illetve az alább látható *név szerinti* megfeleltetést, ahol minden elemhez megadjuk, hogy melyik tömbelembe kerüljön. Az **others** jelölés a név szerinti megfeleltetés speciális esete, amely a külön fel nem sorolt indexeket jelöli. Az Ada 95 szabvány már megengedi a név, illetve pozíció szerinti megfeleltetés együttes használatát is (lásd a *B* tömb inicializációját).

Két további művelet is rendelkezésünkre áll a tömbökön. Az egyik a *résztömbképzés* vagy *szeletelés* (angolul *slicing*), amelynek segítségével egy tömbnek valamilyen indexhatárok közé eső résztömbjét képezhetjük, a másik a *konkatenáció*, amelynek segítségével két azonos határozatlan indexhatárú tömbtípushoz tartozó tömböt kapcsolhatunk össze.

```
A: EgészTömb(11..31)   := (11..20 => 5, others => 42);
B: EgészTömb(101..142) := (1, 2, 3, 4,  others => 0);
C: EgészTömb(1..42)    := (1 => 7, 5|6|11|41 => 6,
                           20..30 => 9, others =>0);
...
C := A & B(111..131);
```

CLU vektortípusok

A CLU-ban a többi konstrukcióhoz hasonlóan a vektortípusnak is létezik mutable és immutable változata. A mutable változat neve `array` (tömb), az immutable-t pedig `sequence`-nek (sorozat) nevezik.

A tömb érdekessége, hogy indexhatárai változtathatók. Létrehozhatjuk üres tömbként – ekkor az indextartomány alsó határa 1, felső határa 0 lesz – vagy elemeinek felsorolásával, egyfajta aggregát alakot használva. Ez utóbbi esetben megadható, hogy az elemek indexelése honnan kezdődjön. Többdimenziós tömb nincs, de tömbökből is képezhetünk tömböket.

```
EgeszTomb    = array[ int ]

it1 : EgeszTomb := EgeszTomb$new()
it2 : EgeszTomb := EgeszTomb$[3 : 42, 21, 14, 7, 6, 3, 2, 1]
i   : int := EgeszTomb$high(it2)
while i >= EgeszTomb$low(it2) do
   e : int := EgeszTomb$fetch(it2, i)
   EgeszTomb$addl(it1, e)
   i := i - 1
end
```

A fenti programrészlet létrehoz egy üres (`it1`) és egy, a 42 pozitív egész osztóit csökkenő sorrendben tartalmazó (`it2`) tömböt, majd ez utóbbi tartalmát fordított sorrendben átmásolja az előbbibe. Amint látható, a CLU tömbök ismerik a méretüket, indexhatáraikat és azokat megfelelő függvényekkel (rendre `size`, `low` és `high`) le is lehet kérdezni. Ezen függvények visszatérési értéke mindig `int` típusú, és mivel a CLU-ban a tömbök csak egész számokkal indexelhetők, a típusdeklarációban is csak az elemek típusát kell megadni. Figyeljük meg az `it2` inicializációjához használt aggregát jelölést. Ez tulajdonképpen egy konstruktorhívás, a kettőspont előtt az indextartomány alsó határát adjuk meg, majd utána az elemek felsorolása következik a tömbbeli sorrendjüknek megfelelően. Ilyen módon üres tömböt létrehozni nem lehet, arra a `new` konstruktor használható.

A szelekció műveletére a példában nem a rövidebb `it2[i]` jelölést, hanem a tömb típus `fetch` műveletét használtuk. A két jelölés szemantikailag nem teljesen azonos, mivel az `it2[i]` balérték, azaz a szelektoroktól megszokott módon a megfelelő elemre vonatkozó referenciával tér vissza, míg a `fetch` műveletet csak jobbértékként használhatjuk, az magát az elemet adja meg. A `fetch` művelet ellentéte a `store`, amelynek három paramétere rendre egy tömb, egy index és az adott indexen tárolandó elem. Valójában az indexelés operátora (`[ ]`), a CLU többi operátorához hasonlóan, csak szintaktikus cukor, amelyet a fordító aszerint helyettesít a `store` és `fetch` műveletek valamelyikére, hogy az bal- vagy jobbérték helyén áll-e.

A példa másik érdekessége az `addl` művelet, amely az alsó határán kiterjeszti az indextartományt és a megadott elemet az új alsó indexen helyezi el. Természetesen létezik `addh` művelet is, amely a felső végén bővíti a tömböt, és léteznek az ellentétes műveletek (rendre `reml` és `remh`), amelyekkel kiolvashatjuk és egyben eltávolíthatjuk a tömbből a legkisebb, illetve a legnagyobb indexű elemeket.

A sorozat típus külső viselkedése nem sokban tér el a tömbétől. A műveletei nagyjából ugyanazok, természetesen azzal az eltéréssel, hogy a módosító műveletek nem az eredeti objektumot változtatják meg, hanem mindegyikük egy új, a módosításnak megfelelő objektummal tér vissza. Eltérés még, hogy a `store` művelet neve `replace`-re változott – természetesen ez is egy új objektummal tér vissza – valamint hogy az inicializáló konstruktornál nem kell megadni az indextartomány alsó határát, mivel az kötelezően 1. Ezen felül a műveletek halmaza bővült a konkatenáció (`concat` vagy `||`) műveletével, amelynek segítségével két sorozat összekapcsolható.

6.4.4. Általánosítás több dimenzióra – tömbök

Tegyük fel, hogy hallgatói nyilvántartásunkat szeretnénk kiterjeszteni az egész iskolára. Egy diákot így két adat, az osztályának azonosítója és azon belül az osztálynévsorbeli sorszáma azonosít. A hallgatók tárolására a legmegfelelőbb eszköz ilyenkor egy *tömb*.

A *tömb* a vektor típuskonstrukció többdimenziós általánosítása az alábbi értelemben: legyenek $\mathcal{H}_1, \ldots \mathcal{H}_d$ rendezett véges halmazok (*indextípusok*) és $\mathcal{T}_0$ egy tetszőleges típus (az *elemtípus*). Az $\mathcal{A} = \text{array}(\mathcal{H}_1, \ldots, \mathcal{H}_d, \mathcal{T}_0)$ tömb típus típusértékhalmaza: $\{a : \mathcal{H}_1 \times \ldots \times \mathcal{H}_d \mapsto \mathcal{T}_0\}$, azaz az $a \in \mathcal{A}$ tömb egy leképezés az indexhalmazok direktszorzatáról az elemtípusra.

Több programozási nyelv nem támogatja közvetlenül a tömböket, hiszen kifejezhető vektorokból álló vektorok segítségével, a kívánt dimenziószámnak megfelelő mélységben egymásba ágyazva azokat. Természetesen ez a konstrukció csak akkor működik, ha megengedett a vektor típus, mint elemtípus. Például a C nyelvben az alábbi módon definiálhatunk többdimenziós tömböt:

```
int t[42][42][42][42]
int **p;
int l = 0;
p = t[21][21];
l = (p[21][21] == t[21][21][21][21]);
```

Amint a példából látszik, megengedett, hogy szelekciónál elhagyjunk indexeket. Ilyenkor a kapott elem egy megfelelően kisebb dimenziószámú tömb, azaz valójában a megvalósított konstrukció tényleg vektorokból álló vektor, megfelelő mélységben egymásba ágyazva, és az indexelés csak a beágyazás szintjét csökkenti.

Vannak olyan programozási nyelvek, amelyek támogatják ugyan a valódi többdimenziós tömböket, de korlátot szabnak a maximális dimenziószámra. A FORTRAN 77 nyelv például megengedi, hogy egy megvalósítás korlátozza a maximális dimenziószámot, csupán azt követeli meg, hogy a korlát legalább három legyen. Még a FORTRAN 90-es változatában is megmaradt a korlátozás lehetősége, de a minimumot felemelték hétre.

Azon nyelvekben, amelyek valóban támogatják a többdimenziós tömböket, a vektorokból álló vektor és a többdimenziós tömb lényegileg különbözik. Az alábbi, Adából vett példa ezt szemlélteti.

```
type Vektor is array(1..10) of Float;
type Mátrix is array(1..10, 1..10) of Float;
type NemMátrix is array(1..10) of Vektor;

V  : Vektor;
M  : Mátrix;
NM : NemMátrix;
...
M(3,4) := NM(3)(4);
NM(2) := V;
```

Figyeljük meg a *Mátrix* és *NemMátrix* típusok közötti eltéréseket. A *Mátrix* típusnál az indexeket egyazon zárójelpáron belülre írjuk és nem hagyhatunk el közülük egyet sem. A *NemMátrix* típus esetén viszont külön zárójelekbe kerülnek az indexek és jobbról balra elhagyhatók.

Tömbök ábrázolása

A tömbök többdimenziós típuskonstrukciók, míg a számítógépek általában egydimenziós memóriacímzést használnak, ezért a tömbök elemeit sorba kell rendezni az ábrázoláshoz. Elvileg a sorbarendezés tetszőleges lehet, de célszerű olyan megoldást választani, amelyben egy tömbelem memóriabeli címe a lehető legkevesebb számolás árán meghatározható. Azt a függvényt, amely egy adott ábrázolás mellett a tömb viszonyítási címéből és a tömbelem indexeiből meghatározza a tömbelem memóriabeli címét, *címfüggvénynek* nevezzük.

A leggyakrabban használt ábrázolási mód a *sorfolytonos* ábrázolás. Legyen T egy k dimenziós, $n_1 \times n_2 \times \cdots \times n_k$ elemű tömb, amelyben a megfelelő indextartományok a $[0 \ldots n_i - 1]$ intervallumok ($i \in [1 \ldots k]$). Legyen továbbá $p_k = 1$ és $j \in [1..k-1]$-re

$$p_j = \prod_{i=j+1}^{k} n_{i-1}$$

Sorfolytonos ábrázolásnál a T tömb viszonyítási címe a $T(0,0,\ldots,0)$ elem memória beli címe lesz, jelölje ezt t. A $T(i_1, i_2, \ldots, i_k)$ elem címe ekkor

$$t + \Big(\sum_{j=1}^{k} i_j p_j\Big) s,$$

ahol s a T tömb egy elemének mérete.

6.5. Halmaz

Tegyük fel, hogy a korábban tárgyalt iskolában a hallgatók mindenféle különórákat vehetnek fel. Azt kívánjuk eldönteni egy tanulóval és egy konkrét különórával kapcsolatban, hogy vajon az adott tanuló jár-e a kérdéses órára. Ennek a feladatnak a megoldásához minden tanulóhoz fel kell jegyeznünk, hogy mely órákra jár[11], azaz minden tanulóhoz hozzá kell rendelnünk azon órák *halmazát*, amelyeket látogat.

6.5.1. Típusértékhalmaz

Legyen $\mathcal{T}_0$ egy tetszőleges típus (az *elemtípus*) és T_0 a hozzá tartozó típusértékhalmaz. Az $S = \text{set}(\mathcal{T}_0)$ *halmaz* típus típusértékhalmaza 2^{T_0} hatványhalmaz, azaz az $s \in \mathcal{S}$ halmaz az elemtípus típusértékhalmazának egy részhalmaza ($s \subseteq T_0$).

A halmaz típus sokkal ritkábban támogatott konstrukció, mint mondjuk a vektor. Ha támogatja is egy programozási nyelv ezt a konstrukciót, általában szigorú megkötésekkel él az elemtípussal szemben. Általában csak diszkrét típusokra lehet halmaz típust definiálni és általában még a maximális elemszám is kötött ezen típusok esetében. Ennek oka a típus szokásos ábrázolásában keresendő.

A halmaz típus leggyakoribb ábrázolása egy bitvektor, amely annyi bitet tartalmaz, amennyi az elemtípus típusértékhalmazának számossága. Minden bit megfelel a típusértékhalmaz egy-egy elemének és a bit 1 értéke azt jelenti, hogy a megfelelő elem benne van a halmazban, míg a 0 azt, hogy nincs. Ennek a reprezentációnak nagy előnye, hogy a szokásos halmazműveletek (unió, metszet, komplementer stb.) egyszerű bitműveletekkel (aritmetikai és, vagy, negáció) valósíthatók meg, amelyeket a számítógép-architektúrák általában processzorszinten támogatnak. Hátránya viszont, hogy ha az elemtípus elemszáma nagy, akkor egy-egy halmaz tárolása nagyon sok helyet igényel, még akkor is, ha az adott halmaz üres.

Szokásos megszorítások a legfeljebb 256 elemű felsorolási típus (például Pascal) vagy a legfeljebb egy gépi szóban ábrázolható méretű diszkrét típus (mint például a Modula-2-ben).

Meglehetősen gyakori viszont, különösen a modernebb programozási nyelvekben, hogy bár nyelvi szinten nem szerepel a halmaz, mint típuskonstrukció, a nyelv által definiált valamely szabványos komponenskönyvtárnak már része egy halmaz megvalósítás. Ezzel a megoldással él például a C++ nyelv is, ahol a szabványos sablonkönyvtár (STL – *Standard*

[11] Esetleg fordítva: minden órához feljegyezni, hogy mely tanulók járnak oda.

Template Library) tartalmaz egy halmaz sablont. A szabványos könyvtárakról bővebben lásd a 15. részt, a könyvtári komponensként megvalósított halmaz típusokkal ebben a fejezetben nem foglalkozunk.

6.5.2. Műveletek

Értékadás és egyenlőségvizsgálat

Mivel a típuskonstrukció alkalmazhatósága éppen az egyszerű ábrázolás érdekében meglehetősen korlátozott, az értékadás és egyenlőségvizsgálat műveletek könnyen elvégezhetők és így általában megengedettek.

Halmazműveletek

A szokásos halmazműveletek – unió, metszet, különbség stb. – általában támogatottak, de a pontos műveleti készlet az egyes nyelvekben különböző lehet. Különleges helyzetben van a komplementerképzés művelete, amelyet viszont sokkal ritkábban valósítanak meg.

Halmazhoz tartozás

Az egyik alapvető halmazművelet a halmazhoz tartozás operátora, amely eldönti, hogy egy megadott elem szerepel-e egy megadott halmazban. Érdekessége, hogy ez a művelet még olyan nyelvekben is előfordul, amelyek nem támogatják a halmazt mint típuskonstrukciót, ám támogatják például az intervallumok használatát. Ilyen nyelv például az Ada, amelyben az **in** operátor segítségével eldönthetjük, hogy egy elem a megadott intervallumba esik-e, ám ennél bonyolultabb halmazokat vagy halmaz típusú változókat nem hozhatunk létre.

6.6. Egyéb iterált konstrukciók

Ebben a részben olyan speciális iterált konstrukciókat gyűjtöttünk össze, amelyek csupán egy-egy programozási nyelv eszköztárában bukkannak fel, de érdekes alternatívát kínálnak a szokásos konstrukciók mellé, vagy jobban szolgálják az adott nyelv eredeti célkitűzéseit.

Perl hasítótáblák

Tegyük fel, hogy hallgatói nyilvántartásunkat úgy szeretnénk átalakítani, hogy a hallgatókat ne sorszám, hanem név alapján lehessen azonosítani. A szükséges típuskonstrukció látszólag nem változott, hiszen továbbra is egy leképezésről van szó, amely a nevekhez hozzárendeli az adott hallgató személyi adatait. A különbség mégis igen jelentős, hiszen korábban a kulcs ismeretében egy elemet előkeresni egy szorzásba került, ha azonban a kulcs típusa tetszőleges típus lehet, a szorzás többé nem végezhető el. A *hasítótáblák* (angolul *hashtable*) a vektorok általánosításai, asszociatív adatszerkezetek, amelyekben tetszőleges típusból származó kulcshoz rendelhetünk tetszőleges értéket.

A hasítótáblák műveletei hasonlók a vektorokéhoz, de vannak lényeges eltérések is. Az egyik fontos különbség a választható indexhalmaz és így a szelekció művelete. Ezenkívül a legtöbb hasítótábla-megvalósítás lehetőséget kínál a táblában szereplő elemek kulcsainak felsorolására.

A programozási nyelvek beépítetten ritkán tartalmazzák a hasítótábla konstrukciót, sokkal inkább valamilyen szabványos könyvtári megvalósítást szoktak kínálni – ilyen például a Java *Hashtable*.

A Perl nyelv azon ritka kivételek közé tartozik, ahol a nyelv maga beépítetten tartalmazza a hasítótábla típuskonstrukciós lehetőséget. A hasítótábla skalár értékhez rendel skalárt, de a Perl nyelvben a karakterlánc skalár típusnak számít. Az alábbi példában egy egyszerű Perl programot láthatunk, amely egy hasítótábla segítségével megszámolja az egyes szavak előfordulási gyakoriságát egy szövegfájlban (a példa egy [Til 96]-beli példaprogram átirata).

```
#!/usr/bin/perl

while ( $inputline = <STDIN> ) {
   while ( $inputline =~ /\b[A-Za-z]\S+/g ) {
      $szo = $&;
      $szo =~ s/[;.,:-]$//;
      $szavak{$szo} += 1;
   }
}
print("A szavak és előfordulási gyakoriságuk:");
foreach $szo (keys(%szavak)) {
   print("$szo: $szavak{$szo}\n");
}
```

A szelekció jelölésére a kapcsos zárójeleket – { és } – használhatjuk. Figyeljük meg, hogy használatakor a tábla nevét a $ jel vezeti be. A Perlben ez azt jelenti, hogy az adott kifejezés skalár típusú. Ezzel szemben ha az egész táblára szeretnénk hivatkozni, a tábla nevét a % karakterrel kell kezdeni. A hasítótáblákban azon kulcsokhoz, amelyekhez nem rendeltünk semmit, egy speciális üres érték tartozik. Egy kulcshoz egy egyszerű értékadással rendelhetünk értéket, a szelekciós művelet segítségével. Egy már meglévő hozzárendelést a `delete` művelet segítségével törölhetünk.

A hasítótáblák között megengedett az értékadás, amely a tábla tartalmának teljes lemásolását jelenti. Fontos művelet még a `keys`, amely a táblában szereplő elemek kulcsainak listáját adja vissza. Segítségével végigjárhatjuk a tábla elemeit, amint az a fenti példa kiíratási részében megfigyelhető.

6.7. Összefoglalás

Ebben a részben azokat a kérdéseket gyűjtöttük össze, amelyek megválaszolásával hatékonyan feltérképezhetjük egy új programozási nyelv típuskonstrukciós eszköztárát. Emlékeztető gyanánt felidézünk néhány lehetséges választ, illetve megoldást a megadott kérdésekre. Ne feledjük azonban, hogy a kérdésekre a válasz nem mindig egyszerű, nem biztos, hogy az eddig látott megoldások bármelyike ráhúzható az éppen vizsgált nyelvre, még csak az sem igaz, hogy a kérdések listája teljes lenne, csupán megpróbálják rendszerezni a vizsgálat szempontjából legfontosabb aspektusokat.

1. *Milyen típusekvivalencia fogalmat használ a nyelv?*
   Itt láthattunk példát az Ada (és általában az erősen típusos nyelvek) igen szigorú értelmezésére, ahol egy típus lényegében csak önmagával ekvivalens (névekvivalencia), illetve a Modula-3 liberális strukturális ekvivalenciáját, amely csak a szerkezeti egyezést követeli meg.
2. *Támogatja-e a nyelv az immutable típusok megvalósítását?*
   A Java nyelvben a nyelv eszközeivel elérhetjük, hogy a típus objektumait ne lehessen megváltoztatni, noha nincs eszköz az immutable tulajdonság jelölésére. A CLU-ban viszont minden típuskonstrukciónak megvan a mutable és immutable változata is. Láthattuk, hogy az immutable típusok használatához fontos a szemétgyűjtéses memóriakezelés és az, hogy a változók objektumreferenciákat és ne magukat az objektumokat tartalmazzák.
3. *Van-e a nyelvben direktszorzat típuskonstrukció?*
   Amint azt láthattuk, nem minden nyelv támogatja a direktszorzat típuskonstrukciót. Ilyen nyelv volt például a FORTRAN. Más nyelvek (például a CLU) többféle megoldást is adnak a problémára. Szintén itt érdemes megvizsgálni, hogy ha az adott nyelvben nincs direktszorzat, kiváltható-e valamilyen másféle megoldással. Például a Perl nyelvben ugyan nincs direktszorzat, de listák vagy hasítótáblák segítségével készíthetünk inhomogén típuskonstrukciókat, amelyekkel utánozható a direktszorzat működése.
4. *Milyenek a direktszorzat szelektorai?*
   A szokásos szelektormegoldás a $t.s_i$ alakú jelölés, ahol a szelektor a megfelelő mezőre vonatkozó referenciát adja vissza, így balértékként is használható. Láthattuk azonban, hogy a CLU nyelvben a szelektorokat például egy lekérdező–beállító műveletpár valósítja meg.
5. *Megengedett-e a direktszorzatok közötti értékadás?*
6. *Szabályozható-e az értékadás működése?*
   Erre a kérdésre is láttunk többféle megoldást. A C nyelvben például a válasz egyértelmű nem volt. Az Ada már lehetővé tette, hogy megtiltsuk az értékadás használatát a **limited** konstrukcióval, de a legkifinomultabb megoldást a C++ adta, amelyben nemcsak megtiltani vagy engedélyezni lehetett, de az értékadás operátor felüldefiniálásával a számunkra kívánatos szemantikát érhettük el.

7. *Van-e egyenlőségvizsgálat a direktszorzat típusokra?*
   A C nyelvben a válasz erre a kérdésre is nem volt, de a legtöbb modern nyelv már megengedi az összehasonlítást.
8. *Szabályozható-e az összehasonlítás szemantikája?*
   Mivel az összehasonlítást sokkal egyöntetűbben sorolják az operátorok közé a programozási nyelvek, mint az értékadást, itt a válaszok is egyszerűbbek. Az operátor-túlterhelést támogató nyelvekben az egyenlőség operátor megfelelő felüldefiniálásával a kérdés kezelhető, míg az operátor-túlterhelést nem támogató nyelvekben általában erre sincs lehetőség.
9. *Van-e kicsomagolás művelet a direktszorzatra?*
   A Pascal típusú nyelvekben figyelhettük meg a kicsomagolás műveletet, amellyel nagyban egyszerűsíthettük a bonyolult minősített hivatkozásokat. Ilyen megoldás volt például a Modula-2 **WITH** utasítása.
10. *Milyen eszközökkel szabályozhatjuk a direktszorzat fizikai ábrázolását?*
    Van, ahol semmilyen hatással nem lehetünk a direktszorzat fizikai szerkezetére. Más nyelvekben fordítási direktívákkal, pragmákkal részleges befolyást gyakorolhatunk például a komponensek igazítására vagy a köztük levő hézagokra. A másik végletet az Ada ábrázolási záradékai képviselik, ahol bitekre lebontva meghatározhatjuk a rekordok fizikai reprezentációját.
11. *Adható-e alapértelmezett kezdőérték a direktszorzat komponenseinek?*
    Ezzel a jelenséggel az Ada nyelvben találkoztunk, ahol a rekordok mezőinek alapértelmezett értéke meghatározható a típus deklarációjában.
12. *Van-e direktszorzat literál?*
    Több nyelvben találkoztunk valamilyen aggregált jelölésformával, amely lehetővé tette az direktszorzatok kezdeti inicializációját. Természetesen ez a megoldás feltételezi, hogy megengedett az értékadás a direktszorzatok között.
13. *Van-e a nyelvben unió típuskonstrukció?*
    Mint láthattuk, az unió típuskonstrukció sokkal ritkább, mint a direktszorzat. Ha nem is szerepel a vizsgált nyelv eszköztárában, érdemes megvizsgálni, milyen más eszközökkel valósítható meg a nyelvben az unióhoz hasonló működés. Erre láthatunk példát objektumorientált nyelvekben egy közös ősosztály bevezetésével, az öröklődés és a polimorfizmus mechanizmusának felhasználásával.
14. *Kötött vagy szabad unió?*
    Láthattuk, hogy például a C-ben az unió típusnak nincs a szokásos értelemben vett szelektora, nem tudja, hogy milyen típusú értéket tárolnak benne és a beleírt érték bármelyik komponens típusként értelmezve kiolvasható belőle. Ezzel szemben az ALGOL 68 **union**-ja vagy az Ada variánsrekordja hordoz magában típusinformációt és a tárolt érték csak a tényleges típusnak megfelelő módon használható fel.
15. *Van-e vektor típuskonstrukció a nyelvben?*
    A legtöbb programozási nyelvben van lehetőség vektorok használatára bár láthattuk, hogy némely speciális nyelvből ez is hiányozhat (ilyen például a SuperNova).
16. *Mi lehet a vektor elemtípusa?*
    Itt is elképzelhetők szélsőségek, hiszen például a Perl nyelvnél láthattuk, hogy csak

skalár típusok – a Perl esetében ez lényegében a karakterláncokat és számokat jelenti – lehetnek tömbelemek. Hasonló megkötéssel találkozhatunk a FORTRAN esetében is. Az Adában pedig nem lehet tömböt létrehozni megszorítás nélküli típusokból.

17. *Mi lehet az indextípus?*
A C-ben a vektorok egész számokkal indexeltek és az indexelés 0-tól kezdődik. A CLU tetszőleges indextartományt megenged, de az indextípus még mindig kötelezően egész. Ugyanakkor a Pascal típusú nyelvekben általában tetszőleges diszkrét típussal lehet tömböt indexelni.

18. *Kötött-e az indextartomány?*
A legtöbb nyelvben a vektor típus deklarációjakor meg kell adni az indextartományt, amely utána nem változtatható meg. A C-szerű nyelvekben csak az indextartomány hosszát lehet megadni, mivel annak alsó határa mindig 0. Az Ada viszont lehetővé teszi határozatlan indexhatárú tömb típusok deklarációját is, amelyeknél a tömb objektum létrehozásakor kell meghatározni az indextartományt. Még ennél is rugalmasabbak a CLU vagy az Eiffel vektorkonstrukciói. Itt a tömb és sorozat objektumok dinamikusan, futás közben képesek megváltoztatni indextartományukat.

19. *Ismeri-e a vektor az indextartományát?*
A C vagy C++ tömbök nem tárolnak információt saját méretükről. Ha erre szükség van, a programozó feladata gondoskodni róla. A Pascal típusú nyelvekben a tömbök általában ismerik méretüket és indextartományukat és – a fordítási beállításoktól függően – ellenőrizhetik, hogy a hivatkozott index beleesik-e az indextartományba. Sok nyelvben – például az Adában, a Javában vagy éppen a CLU-ban – az indextartományt le is lehet kérdezni.

20. *Van-e értékadás művelet a vektorok között?*
Sok nyelvben a válasz nem. A C-ben az értékadás ugyan megengedett, de jelentése más, mivel nem jelenti a tömb lemásolását, csupán a tömb mint mutató másolását. A Pascal típusú nyelvek általában megengedik az értékadást, de az erős típusosság miatt csak a nevesített tömbtípusokra. Az Ada annyit lazít ezen a szigorúságon, hogy az azonos típushoz tartozó tömbök résztömbjei között is megengedi az értékadást, és csupán az elemszámok egyezését várja el.

21. *Lehet-e határozatlan indexhatárú vektor egy formális paraméter típusa?*
Azokban a nyelvekben amelyekben lehetőségünk van határozatlan indexhatárú vektor típusok létrehozására, a vektor objektumok létrehozásakor kell meghatározni az adott objektum indextartományát. Az Adában ez például a tömb változó deklarációjakor történik, mivel ott történik a szükséges memória lefoglalása. Ha tehát tömb típusú változót deklarálunk, meg kell adnunk az indextartományt. Ugyanakkor az Ada lehetővé teszi, hogy az alprogramok formális paraméterlistájában ettől eltekintsünk, így általánosabban használható alprogramokat készíthetünk. A Javában ugyanakkor a változók csupán referenciák a vektor objektumokra, így még azok deklarációjakor sem kell az indextartományt meghatározni. Így természetesen átadhatók paraméterként is. Elérhetünk hasonló hatást más nyelvekben is, például a C vagy C++ nyelvekben, kihasználva a mutatók és tömbök közötti rokonságot.

22. *Van-e lehetőség többdimenziós tömbök létrehozására?*
Sok nyelvben csupán egydimenziós tömbök (vektorok) hozhatók létre. Általában ezen nyelveken is megvalósítható a kívánt hatás, például vektorokból álló vektorok alkalmazásával. Más nyelvek viszont lehetővé teszik a többdimenziós tömbök használatát, sőt gyakran meg is különböztetik a vektorok vektora megoldástól.
23. *Van-e halmaz típus a nyelvben?*
Láthattuk, hogy a halmaz típuskonstrukció sokkal ritkább, mint a vektorok és tömbök különféle változatai. Mégis vannak nyelvek, például a Pascal vagy a Modula-2, amelyek támogatják. Más nyelvek, ilyen például a Java, illetve a C++, beépítetten ugyan nem támogatják ezt a konstrukciót, de szabványos könyvtáraik tartalmaznak valamilyen halmaz megvalósítást.
24. *Milyen elemtípusú halmazok engedélyezettek?*
Legtöbbször a nyelv, még ha támogatja is a halmaz konstrukciót, megkötéssel él az elemtípusra. Általában csak felsorolási típusok engedélyezettek és még az elemszámukra is szokott felső korlát lenni – a Pascal nyelvben ez például 256.
25. *Milyen más iterált konstrukciók vannak a nyelvben?*
Egyes nyelvek további speciális iterált konstrukciókat is támogathatnak, a nyelvbe beépítve. Ilyen például a lista vagy a hasítótábla a Perl nyelvben.

Feladatok

1. Sorolja fel az immutable típusok használatának előnyeit és hátrányait!

2. Mit jelent a strukturális ekvivalencia?

3. Milyen előnyökkel jár a határozatlan indexhatárú tömb típusok használata?

4. Készítsen címfüggvényt négyzetes mátrixok oszlopfolytonos, illetve csigavonalas ábrázolásához!

5. Miért különösen bonyolult a komplementerképzés művelete halmaz típusok esetén?

6. Készítsen négyzetes mátrix típust Ada nyelven. A mátrix mérete legyen a típus paramétere. Hogyan lehet garantálni a négyzetességet?

7 Alprogramok

Ebben a fejezetben a vezérlési absztrakció megvalósításának programnyelvi eszközéről, az alprogramról olvashatunk. Megtudjuk, miért hasznosak az alprogramok. Megtanuljuk a különbséget az eljárások és a függvények között. Megismerjük az alprogramok specifikálásával, definiálásával és hívásával kapcsolatos fogalmakat. A fejezet ismerteti, milyen paramétereket lehet átadni egy alprogramnak, és hogy ezeket a paramétereket milyen paraméterátadási technikák segítségével adhatjuk át. Kiderül az is, hogy mik a rekurzív alprogramok, és hogy miben térnek el az alprogramok a makróktól, a korutinoktól vagy az iterátoroktól. Megtudjuk, hogy az alprogramok hogyan illeszkednek be abba a környezetbe, amit a program szerkezete jelent, hogyan lehet egymásba ágyazni őket, és hogy milyen hatóköri szabályok vonatkoznak rájuk. A fejezetben végül bemutatatjuk az alprogram, mint nyelvi elem megvalósításának legfontosabb kellékeit.

Alprogramok már azelőtt léteztek, mielőtt az első programozási nyelvek megjelentek volna, írja Ravi Sethi az alprogramokról szóló fejezetének bevezetőjében [Set 96], majd magyarázatképpen hozzáteszi: voltak, akik már 1944-től kezdve azzal próbálkoztak, hogy a programok írását könnyebbé és gyorsabbá tegyék, mégpedig úgy, hogy kódrészleteket másolgattak ki egymás jegyzetfüzetéből, hogy beépítsék azokat saját programjaikba. Sebesta még korábbi példát említ: az első programozható számítógép, a Charles Babbage által az 1840-es években épített Analytical Engine is képes volt már arra, hogy egy köteg vezérlőkártyát programja több pontján újra és újra felhasználjon [Seb 02].

Az alprogram olyan nyelvi szerkezet, amelynek segítségével nevet rendelhetünk egy kódrészlethez, hogy azt később, amikor csak akarjuk, egyszerűen végrehajthassuk. A kódrészlet végrehajtásának kezdeményezése, azaz az *alprogram meghívása* a kódrészlethez rendelt név (és esetleg paraméterek) megadásával történik. Paraméterek használata esetén egy alprogram segítségével paraméteres számítást is kényelmesen megfogalmazhatunk. Ilyenkor a névvel ellátott kódrészlet, az alprogram, a paraméterként megadott adatokkal dolgozhat. Az alprogram fogalma tehát nagyon hasonlít arra a fogalomra, amit a matematikusok függvénynek hívnak immár évszázadok óta.

Az alábbi, Ada nyelven megadott programrészlet egy olyan függvény-alprogramot mutat be, amely kiszámolja két egész szám maximumát. Programunkban bárhol meghívható, ahol ilyen feladatot kell elvégezni. Ez az alprogram tehát lehetőséget ad arra, hogy egy utasítássorozatot kényelmesen újra és újra felhasználjunk, és arra is, hogy a felhasználáskor az utasítássorozatot testreszabjuk, azaz felparaméterezzük.

```
function Max (A, B: Integer) return Integer is
begin
    if A > B then return A; else return B; end if;
end Max;
```

Az alprogram meghívása történhet az alábbi módon. Feltételezzük, hogy *K*, *L* és *M* egész típusú változók:

```
K := 3; L := 2; M := 5;
K := Max(K,Max(L,M));
```

7.1. Az alprogramok hatása a szoftverminőségre

Az alprogram a *vezérlési absztrakció* megvalósításának eszköze. Azon kívül, hogy programutasítások egy sorozatának többszöri felhasználására és felparaméterezésére használható, azt is lehetővé teszi, hogy a tényleges programutasítás-sorozattól elvonatkoztatva, egy egységként kezelhessünk egy logikailag önálló kódrészletet, egy külön is megfogalmazható, jól körülhatárolt részszámítást. Emiatt az alprogram nem csak az *újrafelhasználhatóság*, hanem egyéb szoftverminőségi szempontok, nevezetesen az *olvashatóság*, *módosíthatóság* és *karbantarthatóság* miatt is kiemelten fontos nyelvi elem. Ezen utóbbi szoftverminőségi szempontok súlyát átérezhetjük, ha arra gondolunk, hogy egy szoftvertermék életciklusának általában csak kisebb részét teszi ki maga a termék elkészítése.

Újrafelhasználhatóság • Az alprogramok szerepe ebből a szempontból nyilvánvaló: ugyanazt a kódrészletet többször is fel lehet használni, különböző paraméterekkel végre lehet hajtani, anélkül, hogy többször is költenünk kellene az előállítására (megtervezésére, megírására, lefordítására, helyességének bizonyítására és ellenőrzésére, tesztelésére). Ez gazdaságosabbá teszi a programfejlesztést. Ezen kívül az elkészült teljes programtermék mérete is általában kisebb lesz.

Olvashatóság • Egy jól megválasztott név a programszöveget olvasó számára sokkal kifejezőbb lehet, mint az alprogramot megvalósító kód, hiszen az alprogram általában amúgy is egy jól megnevezhető, logikailag önálló részfeladatot old meg. Amikor az alprogram hívása többször is szerepel a programszövegben, nem kell újra és újra megérteni a működését, elég egyszer, azon a ponton, ahol megadtuk az alprogramot. (Sőt, elképzelhető, hogy pusztán az alprogram nevéből, esetleg dokumentációjából az olvasó megérti, mit is csinál az alprogram, és nem is szükséges a megvalósítását elolvasnia.)

A programszöveg alprogramokkal történő tagolása akkor is kedvezően befolyásolja az olvashatóságot, amikor az alprogramot igazán csak egyszer akarjuk meghívni. Ily módon az alprogram nem csak a programszöveg logikai, hanem *fizikai tagolásának* is eszköze[1].

Mindezek miatt elmondható, hogy az alprogramok használatával a programszöveg rövidebbé és olvashatóbbá tehető, és *a kód bonyolultsága* (komplexitása) jelentős mértékben *csökkenthető*.

Módosíthatóság • Az alprogramok egy-egy jól meghatározott feladatot oldanak meg. Az alprogramokat használó, hívó programegységek számára közömbös, hogy az alprogramok ezeket a feladatokat hogyan oldják meg. Az alprogramokat alkotó utasítássorozatokat anélkül változtathatjuk meg, hogy ennek a hívó programegységekre bármi hatása lenne. (Ezt az állítást később, a 7.3.2. szakaszban pontosítani fogjuk.) Például, ha egy alprogram feladata az, hogy egy számsorozatot rendezzen, akkor a tényleges rendezési algoritmus ismerete nélkül tudjuk használni az alprogramot arra, hogy számsorozatokat rendezzünk vele. Egy jól

[1] A programszöveg tagolásának szempontjából fontos lehet, hogyan választjuk meg alprogramjaink méretét. Az általánosan elfogadott elv az, hogy egy alprogram legfeljebb egy képernyőoldal legyen, azaz körülbelül 20-25 sor. Az ennél hosszabb, de még mindig önálló részszámítást meghatározó alprogramokat célszerű feldarabolni, néhány részletet kiemelni külön – akár csak egyszer használt – alprogramokba.

megtervezett alprogramnál az alprogram megvalósítását, azaz az alprogramot alkotó utasítássorozatot lecserélhetjük (mondjuk buborékrendezésről egy hatékonyabbra, például egy gyorsrendezésre), anélkül, hogy az alprogramot hívó programegységeken bármit is változtatnunk kellene.

Karbantarthatóság • Az alprogramok a programszövegben feleslegesen ismételt kód mennyiségét (a programszöveg redundanciáját) csökkenthetik, és ennek – a könnyebb olvashatóság és módosíthatóság mellett – szintén pozitív hatása van a program karbantarthatóságára. Ha valamit változtatni szeretnénk egy bizonyos, gyakran használt részfeladat megoldásán, akkor azt a változtatást csak egy helyen, a részfeladatot megoldó alprogramban kell elvégezni.

Noha az alprogramok elsősorban a vezérlési absztrakció megvalósításához nyújtanak nyelvi eszközöket, fontos szerepet játszanak az adatabsztrakció megvalósításában is. Az adatabsztrakció lényege, hogy programegységeink elkészítésénél elvonatkoztatunk a használt adatok ábrázolási módjától (reprezentációjától): az adatokat néhány előre megadott művelet segítségével kezeljük. Ezek a műveletek pedig alprogramok, azaz vezérlési absztrakciók: tényleges megvalósításuktól függetlenül, attól elvonatkoztatva használjuk őket adataink kezelésére. Az adatabsztrakcióról részletesebben a 10. fejezet szól.

Az alprogramok szerepét a programtervezés szempontjából is vizsgálhatjuk. Egy bonyolult feladat megoldását általában úgy szoktuk előállítani, hogy a bonyolult feladatot felbontjuk egyszerűbbekre, és azokat külön-külön próbáljuk megoldani. A fentről lefelé haladó, úgynevezett vízesés tervezési módszer egy bonyolult feladatból egyszerűbbeket, azokból még egyszerűbbeket állít elő, egészen addig, amíg elég egyszerű feladatokhoz nem jutunk, amelyeket már könnyű megoldani. A lentről felfelé haladó, úgynevezett szökőkút tervezési módszer azt vizsgálja, hogy egyszerű feladatokra adott megoldásokból milyen bonyolultabb, majd még bonyolultabb feladatokhoz tudunk megoldást előállítani. Mind a két tervezési módszer azon az elven alapszik, hogy egy bonyolultabb feladatot egyszerűbbek segítségével fogalmazunk meg. A feladat megoldása, azaz a programtermék előállítása során ennek megfelelően egyszerűbb programokból bonyolultabb programokat építünk fel, ahol az egyszerűbb programokat olyan alprogramok segítségével fogalmazzuk meg, amelyeket a belőlük épített bonyolultabb programok hívnak majd meg.

7.2. Eljárások és függvények

A programozáselméletben kétféle alprogramot szokás megkülönböztetni: az eljárásokat és a függvényeket. *Az eljárások* a változók által kifeszített téren vagy a program környezetén elvégzendő *transzformációkat adnak meg. A függvények* ezzel szemben valamilyen *értéket számítanak ki*, de transzformációt nem végeznek, sem a program változóinak értékére, sem a program környezetére nincsenek semmilyen hatással; úgy is szokták mondani, hogy a függvényeknek *nincs mellékhatásuk*.

Az eljárások meghívását az utasítások, a függvények meghívását a kifejezések közé lehet sorolni. Úgy is vehetjük, hogy az eljárásokkal a nyelv által biztosított utasítások halmazát

bővíthetjük, míg a függvények segítségével a kifejezések felépítésében használt operátorok halmazát. A fejezet elején szereplő *Max* alprogram egy függvény. Mellékhatása nincs, két értékből egy harmadikat számol ki. Meghívása egy kifejezés felépítésében szerepel. Az alábbi, szintén Ada nyelven megadott alprogram egy eljárás, ami a paramétereként kapott két változó értékét felcseréli:

```
procedure Swap (A, B: in out Integer) is
  Temp: Integer := A;
begin
  A := B; B:= Temp;
end Swap;
```

A *Swap* eljárás a változókon transzformációt hajt végre, meghívásának hatására két változó értéke is megváltozik. (Tegyük fel ismét, hogy *K* és *L* egész típusú változók.)

```
K := 42; L := 24;
Swap(K,L);
```

Mint látjuk, az eljárás meghívása önálló utasítás.

Az eljárásokat és a függvényeket az alprogram nevének kitalálásakor is érdemes megkülönböztetni. Az eljárásokat utasításként használjuk, és valamilyen tevékenységet írnak le, ezért célszerű ige szófajú nevet választani. Ezzel szemben a függvényeket kifejezésekben használjuk, értékek megfogalmazására – így egy főnév, egy melléknév vagy egy melléknévi igenév sokkal kifejezőbb. Egy számsorozatot rendező alprogramot *Rendez*-nek hívhatunk, ha eljárásként, és *Rendezett*-nek, ha függvényként valósítjuk meg.

Az alprogramok a bennük megfogalmazott számítást paraméterként átvett értékeken hajtják végre. Ezeket szoktuk *bemenő paramétereknek* nevezni. A függvények az általuk kiszámított értéket az úgynevezett *visszatérési értéken* keresztül juttatják el a hívónak. Az eljárások ezzel szemben *kimenő paramétereket* használnak erre a célra. Például a *Swap* eljárás két paramétere egyszerre bemenő és kimenő paraméter. A függvényeknek nem írunk kimenő paramétereket, csak bemenőket.

Érdemes megvizsgálni, hogy a programozási nyelvekben hogyan valósítják meg az eljárás és függvény alprogramokat.

7.2.1. Nyelvek, ahol nincs különbség az eljárások és függvények között

Sok programozási nyelv, például a C, a C++, a Java, a C# vagy a CLU, nem igazán különbözteti meg az eljárásokat és a függvényeket. Ezekben a nyelvekben az utasítások és a kifejezések között sem olyan éles a határvonal. Egy kifejezés állhat önálló utasításként, mely esetben az utasítás végrehajtása a kifejezés kiértékelését jelenti, ami ilyenkor gyakran valamilyen mellékhatás fellépésével jár. Például a *c++* kifejezés értéke megegyezik a *c* változó értékével, de mellékhatásként a *c* változót eggyel növeli. Ezekben a nyelvekben minden alprogram „függvény", noha vannak olyanok közöttük, amelyek visszatérési értékének típusa az „üres típus". (Angolul gyakran szokták ezt a típust void-nak nevezni.) Az ilyen

függvények felelnek meg lényegében az eljárásoknak. Nézzük meg, hogyan írnánk meg a maximumszámító és a változókat felcserélő alprogramokat C-ben:

```
int max (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
    /* Vagy egyszerűbben: return (a > b) ? a : b; */
}

void swap (int *a, int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}
```

A void visszatérési értékű függvényeket ritkán használjuk kifejezésekben[2], viszont a nem **void** visszatérési értékű függvényeket annál gyakrabban használjuk utasításként, úgy, hogy a visszatérési értéket „eldobjuk", és csak a mellékhatás kedvéért értékeljük ki a függvényhívást. Azokban a nyelvekben, ahol nincs külön nyelvi eszköz a kivételek kezelésére (lásd a 8. fejezetet), az ilyen eldobható visszatérési értékeket szokás használni az alprogram abnormális befejeződésének jelzésére. Jó példák erre a *printf* és *fprintf* szabványos könyvtárbeli C függvények, amelyek formázott szöveg kiíratására valók. (Az első a szabványos kimenetre, a második egy paraméterként megadott fájlba ír.) Ezek a függvények végrehajtáskor visszaadnak egy egész értéket: ha ez az érték nem negatív, az azt jelenti, hogy a végrehajtás sikeres volt, ha viszont negatív, akkor valamilyen hiba történt kiíratás közben. Nyilvánvaló, hogy nem a visszatérési érték az, ami igazán fontos a *printf* és az *fprintf* hívójának, hanem a mellékhatás, amit a műveletek a program környezetén (a szabványos kimeneten, illetve a fájlon) kifejtenek. Ha a programozó akarja, akkor persze megvizsgálhatja a visszatérési értéket, de nem kötelező ezt tennie. Az alábbi két hívás közül az elsőben vizsgáljuk a visszatérési értéket, a másodikban viszont eldobjuk:

```
FILE *f = fopen ("alma.txt", "w");

if (fprintf (f,"jonatan\n") < 0) {
    printf ("Nem sikerult a fajlba irni!\n");
}
```

[2] A C-ben például a vessző operátor segítségével felépíthető olyan kifejezés, amiben az egyik részkifejezés void. A `swap(&k,&l)`, `max(k,l)` kifejezés értéke megegyezik a `max(k,l)` kifejezés értékével, de mielőtt ezen utóbbi kifejezés kiértékelődne, végrehajtódik a *swap* is.

7.2.2. Nyelvek, ahol van különbség az eljárások és függvények között

Vannak olyan programozási nyelvek, ahol élesebb a határvonal eljárások és függvények között. Az Adában, mint láttuk, két különböző kulcsszóval vezetünk be eljárásokat és függvényeket, és egész más szövegkörnyezetben használhatjuk őket: az eljáráshívások csak utasításként, a függvényhívások csak kifejezésként szerepelhetnek. Más nyelvekben, mint például a Modula-2, az ALGOL 60 vagy a PL/I, noha létezik ugyanez a különbség, mindkét fajta alprogramot ugyanazzal a kulcsszóval, például a *procedure* kulcsszóval vezetjük be. A Modula-2-ben például szintaktikusan az különbözteti meg a függvényeket az eljárásoktól, hogy a paraméterlista után kettősponttal elválasztva megadjuk a visszatérési érték típusát:

```
PROCEDURE Max (VAR a, b: INTEGER) : INTEGER;
BEGIN
  IF a > b THEN RETURN a ELSE RETURN b END
END Max;
```

A Modula-3 nyelv is megkülönbözteti az eljárásokat és a függvényeket, de ebben a nyelvben van lehetőség arra (az **EVAL** utasítással), hogy egy függvényt csak a mellékhatásai kedvéért értékeljünk ki, a visszatérési értékét pedig figyelmen kívül hagyjuk.

A programozási nyelvek többsége nem kényszeríti rá a programozót arra, hogy a függvény alprogramokat mellékhatás nélkül, azaz a szemantikus modellnek, nevezetesen a matematikai értelemben vett függvényeknek megfelelően készítse el. Azokban a nyelvekben, amelyekben egyáltalán különbséget tesznek eljárások és függvények között, ennek megvalósíthatósági okai is vannak[3]. Így a függvények mellékhatásmentessége leginkább programozási stílus kérdése marad. A programozónak kell arra odafigyelnie, hogy egy függvényben például ne változtasson meg egy globális változót, mert az nem túl szép.

Az Ada feltételezi a programozótól a fenti stílus követését. Ennek a feltételezésnek közvetlen haszna van a párhuzamos programok írása esetén. A védett egységek (lásd a 14.2.18. szakaszt) olyan objektumok, amelyek eljárásait csak kölcsönös kizárás mellett lehet végrehajtani, függvényeit azonban lehet egyidőben (konkurensen) is – mondván, hogy úgy sincs mellékhatásuk, tehát nem interferálhatnak egymással. Így a stílus követéséért cserébe hatékonyabb párhuzamos programokat ad a nyelv.

Végül említsük meg azokat a nyelveket, amelyekben egyáltalán nem lehet mellékhatásos függvényeket írni, sőt, még eljárásokat sem. Ilyenek az úgynevezett tisztán funkcionális nyelvek (lásd a 16. fejezetetet), mint például a Haskell. Ezekben a nyelvekben az alprogramok egy az egyben megfeleltethetők a matematikai függvényeknek. Azzal, hogy a mellékhatásokat teljesen kiküszöbölik az alprogramokból, a programok bonyolultságát igyekeznek csökkenteni. Kétségtelenül megmutatkoznak ennek előnyei például akkor, amikor programjaink helyességét matematikai eszközökkel akarjuk bizonyítani.

A mellékhatások elkerülése a program olvashatóságára, egyértelműségére és hordozhatóságára is kedvezően hat. Ha mellékhatásos függvényeket hívunk meg egy kifejezésben,

[3] Nem szerencsés például megtiltani, hogy egy függvény meghívhasson egy eljárást, hiszen nyomkövetéshez, naplózáshoz hasznos lehet, ha egy függvény írni tud egy fájlba. Ez már egyfajta mellékhatás. Ha a függvény meghívhat egy eljárást, akkor abban az eljárásban, és így közvetve a függvényben is, már játszva okozhatunk további mellékhatásokat.

akkor a kifejezés értéke (és a mellékhatást kiváltó utasítások sorrendje) is függ a kifejezésben található operátorok kiértékelési sorrendjétől (precedenciájától). Ez gyakran még akkor is nehézzé teszi a kód megértését, ha a részkifejezések kiértékelési sorrendje egy kifejezésben pontosan meghatározott. Ha mégsem, akkor nagyon könnyen írhat az ember olyan kifejezéseket, amelyek értéke a nyelv által nem meghatározott, és erre a hibára a fordítóprogram nem is hívja fel a figyelmet. Elképzelhető, hogy ezután a programunk „véletlenül jól működik" egy bizonyos architektúrán, de egy másik rendszerre, vagy csak egy másik fordítóprogramra való áttéréskor hirtelen minden elromlik. A `++i+t[i]` kifejezés értéke a C-ben például függ a használt C implementációtól: vagy a `++i`, vagy a `t[i]` értékelődik ki először. Ugyanezt mondhatjuk el egy, a fentivel megegyező jelentésű Ada programról is. Javában a kifejezés értéke egyértelmű (először kiértékelődik az összeadás bal oldala, majd a jobboldala), mégsem szerencsés az ehhez hasonló kifejezések használata: túlságosan is nehézzé teszi ugyanis a programszöveg olvasását.

7.3. Az alprogramok és hívások szerkezete

Az *alprogram programegységek* definíciója két részből szokott állni: egy *specifikációból* és egy *törzsből*. A specifikáció arról ad információt, hogy az alprogramot hogyan kell használni, hogyan kell meghívni. A törzs tartalmazza azt az utasítássorozatot, amelyet végre kell hajtani az alprogram meghívásakor.

Az alprogram meghívása az alprogram nevének megadásával és a számításhoz szükséges paraméterek felsorolásával szokott történni. Azt szokás mondani, hogy az alprogram a hívása után *aktívvá* válik.

Fontos elnevezések még a *formális paraméter* és *aktuális paraméter* kifejezések is. Formális paraméternek nevezzük azokat a paramétereket, amelyek az alprogram definíciójában szerepelnek, azaz amelyek segítségével (amelyek függvényében) definiáljuk az alprogram hívásakor elvégzendő számítást. Aktuális paraméternek nevezzük azokat a paramétereket, amelyeket az alprogram hívója szolgáltat az alprogramnak, azaz azokat, amelyekkel az adott hívás során el kell végezni az alprogramban definiált számítást. Az alprogram hívásakor ezek szerint *meg kell feleltetni* az aktuális paramétereket a formális paramétereknek. Az alprogram formális vagy aktuális paramétereinek sorozatát *paraméterlistának* szokás nevezni.

7.3.1. Mi lehet paraméter vagy visszatérési érték?

Érdekes kérdés, hogy milyen egyedek (entitások) lehetnek paraméterei egy alprogramnak, illetve, hogy milyen egyedeket adhat vissza egy függvény. Egy programozási nyelvben az egyedek általában értékek, típusok, alprogramok és modulok szoktak lenni.[4] Azt, hogy érték egyedekkel paraméterezzünk egy alprogramot, majdnem minden programozási nyelv megengedi. (Kivételt jelentenek egyebek között a BASIC egyes korai változatai, melyek egyáltalán nem tették lehetővé alprogramok paraméterezését.) Érték egyed paraméterre mutatott példát az előzőekben több nyelven is elkészített maximumszámító függvényünk és

[4] Az Adában az előzőektől különböző egyedeknek vehetjük a folyamatokat (lásd a 14. fejezetet), vagy például a Clean-ben a típuskonstruktorokat (lásd a 16. fejezetet).

paramétereket felcserélő eljárásunk. Nézzük most meg, hogy milyen paraméterek kapcsán van nagyobb eltérés a különböző nyelvek között.

Meghatározatlan méretű tömb

Sok nyelv megengedi, hogy a formális paraméter típusa egy nem teljesen meghatározott típus legyen. Ennek segítségével lehet olyan alprogramot írni, amely különböző hosszúságú tömbökkel is tud dolgozni.[5] A tömb típusú változók mérete különböző nyelvekben máskor és máskor kerül meghatározásra. Egyes nyelvekben (például ASA FORTRAN, BASIC, Pascal) fordítási időben, más nyelvekben a változódeklaráció kiértékelésekor (Ada, FORTRAN 90 stb.), és vannak olyan nyelvek, amikor teljesen dinamikusan (APL, Perl, Common LISP). A legtöbb nyelvben azonban a formális paraméterekre lazább szabályok vonatkoznak, mint a változókra. Még a Standard Pascalban is tudunk olyan rendező eljárást írni, amely képes tetszőleges hosszúságú tömböket rendezni.

```
procedure rendez(var V: array [low, high: integer] : integer);
```

A PL/I-ben, a Pascalban, a Modula-2-ben, az Adában stb. az aktuális paraméter fogja a formális paraméter típusát teljesen meghatározni. A fenti példában a *low* és a *high* segítségével hivatkozhatunk az aktuális paraméter alsó, illetve felső indexhatárára. Az Adában, ha a *V* tömb egy alprogram formális paramétere, akkor a *V*-nek megfeleltetett aktuális paraméter tömb alsó indexe a *V'First* kifejezéssel, felső indexe a *V'Last* kifejezéssel, az indextartomány pedig a *V'Range* kifejezéssel kapható meg. Tehát az alprogrambeli számítást lehet ezen kifejezések függvényében megadni. (Az Adában az alprogramok formális paraméterének típusát a típusok név szerinti egyenértékűsége (ekvivalenciája) miatt nem lehet az alprogram specifikációjában felépíteni, ezért definiáltuk külön a *Vektor* típust.)

```
type Vektor is array (Integer range <>) of Integer;
procedure Rendez (V: in out Vektor) is
  J: Integer;
  Temp: Integer;
begin
  for I in V'Range loop
    J := V'First;
    while J < I and then V(J) < V(I) loop
      J := J + 1;
    end loop;
    Temp := V(I);
    V(J+1..I) := V(J..I-1);
    V(J) := Temp;
  end loop;
end Rendez;
```

[5] Az Adában nem csak a megszorítás nélküli indexhatárral rendelkező, azaz különböző hosszú tömböket megengedő típusok lehetnek nem meghatározottak, hanem a megszorítatlan diszkriminánsú rekord típusok is. Ezek a nem meghatározott típusok szintén szerepelhetnek formális paraméterként.

A C-ben változó hosszúságú tömböket mutató segítségével lehet paraméterként átadni. Az aktuális paraméter méretéhez azonban nem lehet a formális segítségével hozzáférni, úgy-hogy ilyen esetben egy külön paraméterben át szokás adni az aktuális paramétertömb méretét. Ez a megoldás természetesen bonyolultabb, nehezebben olvasható, és több hibalehetőséget hordoz magában: minden egyes hívásnál oda kell figyelni, hogy a *meret* formális paraméternek megfelelő értéket feleltessünk meg.

```
void rendez (int[ ] v, int meret)
. . .
```

A Javában a tömbök is objektumok, amelyek „tudják magukról", hogy mekkorák. Ez az információ a *length* attribútumuk segítségével kérdezhető le. Ebben a nyelvben a tömbök mérete nem is része a tömb típusnak: egy tömb típusú változó vagy formális paraméter deklarációja nem is tartalmazhat méretinformációt. A következő szakaszban látni fogunk arra egy példát, hogy hogyan kell a Javában tömböt paraméterként használni.

Többdimenziós tömb

Még izgalmasabb a meghatározatlan méretű tömbök kérdése, ha a tömb többdimenziós. Azokban a nyelvekben, amelyek lehetővé teszik a tömbindexhatárok futás közbeni lekérdezését, persze nincs sok gondunk. A Javában a többdimenziós tömbök olyan tömbök, amelyek tömböket tartalmaznak. A tartalmazott tömbök mérete persze különbözhet, de ez nem okoz problémát: a tartalmazó tömb csak objektumhivatkozásokat (referenciákat) tárol. Egy kétdimenziós, egészeket tartalmazó tömb elemeit például úgy kapja meg a fordító, hogy megkeresi benne az első dimenzióbeli indexnek megfelelő elemet, ami egy hivatkozás egy tömbre, mely hivatkozást feloldja, és megkeresi az így kapott tömbben a második dimenzióbeli indexnek megfelelő elemet. Nézzük meg, hogyan adhatnánk meg egy kétdimenziós mátrix elemeinek összegét:

```
double sum(int[ ][ ] mat) {
  int sum = 0;
  for (int i = 0; i < mat.length; ++i) {
    for (int j = 0; j < mat[i].length; ++j) {
      sum += mat[i][j];
    }
  }
  return sum;
}
```

Az Adában a többdimenziós tömbök valóban többdimenziós tömbök. Ezért aztán egy mátrix minden sora ugyanannyi elemet tartalmaz.[6] A fordító az elemeket a Javánál hatékonyabban, valószínűleg sorfolytonosan vagy oszlopfolytonosan fogja tárolni. Az aktuális

[6] Az Adában lehet tömbök tömbjét is definiálni, de ilyenkor arra kell vigyázni, hogy egy tömb elemtípusa már meghatározott típus kell, hogy legyen. (Ez a meghatározott elemtípus persze lehet egy olyan mutató típus is, amelynek bázistípusa nem meghatározott típus. Ez felel meg leginkább a Java-beli tömbábrázolásnak.)

paraméter méreteit ilyenkor a *Range*, *First* stb. attribútumok dimenziószámmal indexelt változataival kérdezhetjük le.

A FORTRAN-ban, a C-ben, a C++-ban stb. a többdimenziós tömböket szintén sor- vagy oszlopfolytonosan tárolják, viszont a fordító nem fér hozzá egy ilyen formálisnak megfeleltetett aktuális méretéhez. Ez azért baj, mert például a sorfolytonos ábrázolásnál mindenféleképpen ismerni kell az oszlopok számát ahhoz, hogy egy tömbelemet megkereshessünk. A FORTRAN-ban a formális paraméter deklarációja tartalmazhatja a szükséges információkat. Az alábbi egy FORTRAN 77-ben írt változata a fenti, elemeket összegző függvénynek:

```
      INTEGER FUNCTION MATSUM(MATRIX, ROWS, COLS)
        INTEGER ROWS, COLS, MATRIX(ROWS, COLS), I, J
        MATSUM = 0
        DO 20 I = 1, ROWS
          DO 10 J = 1, COLS
            MATSUM = MATSUM + MATRIX(I,J)
10        CONTINUE
20      CONTINUE
        END
```

A FORTRAN-ban írt függvény használata sokkal több odafigyelést igényel, mint a Java vagy egy Ada-változat használata: ez határozottan negatívuma a FORTRAN-nak. Miért? Azért, mert amikor a `MATSUM` függvényt meghívjuk, a `ROWS` és `COLS` paramétereknek a programozó fogja megfeleltetni az aktuális paraméterként átadott mátrix méreteit. Ha véletlenül rosszul tölti ki ezt a két paramétert, a függvény hibás adatokkal fog dolgozni, a mátrix elemei helyett mindenféle szemetet fog összeolvasni a memóriából.

A C-ben és a C++-ban még rosszabb a helyzet. Egy formális paraméter típusát (vagy legalábbis méretét) nem fejezhetjük ki egy másik formális paraméter segítségével, mint ahogy azt a FORTRAN-ban tettük. Persze, egy konkrét szám már használható erre a célra. A többdimenziós tömbök sorfolytonos ábrázolása miatt az oszlopok számát meg kell adnunk ahhoz, hogy a fordító elkészíthesse a megfelelő indexfüggvényt a tömbelemek visszakereséséhez. A sorok számát nem kell megadni, vagy legalábbis nem kell a formális paraméter típusában:

```
int sum (int matrix[ ][10], int rows)
{
  int i, j, s = 0;
  for (i = 0; i < rows; ++i)
    for (j = 0; j < 10; ++j)
      s += matrix[i][j];
  return s;
}
```

Egy ilyen alprogramot paraméterezhetünk a tömb sorainak számával (itt is vigyázni kell, hogy a *rows* formálisnak megfelelő aktuálist feleltessünk meg), de az oszlopok száma „bele van égetve" az alprogram definíciójába: csak 10 oszlopot tartalmazó mátrixra hívható

meg a *sum* függvény. Ha általánosabban akarjuk ezt a függvényt elkészíteni, más megoldás után kell néznünk. Használjunk mutató-aritmetikát! Ebben az esetben külön formális paraméterben adható át a sorok és az oszlopok száma. Ez a megoldás már kellőképpen általános, de azon kívül, hogy rendelkezik a FORTRAN hibájával, nevezetesen azzal, hogy könnyű a függvényhívást elrontani, van még egy hibája is: az indexelést nekünk magunknak kell leprogramoznunk, mert a fordító, információ hiányában, ezt nem tudja elvégezni helyettünk.

```
int sum (int *matrix, int rows, int cols)
{
  int i, j, s = 0;
  for (i = 0; i < rows; ++i)
    for (j = 0; j < cols; ++j)
      s += *(matrix + i * cols + j);
  return s;
}
```

A hívás helyszínén létrehozhatunk egy kétdimenziós tömböt, és megfelelő explicit típuskonverzió után átadhatjuk aktuális paraméterként *sum*-nak:

```
int m[2][2] = {{1, 2}, {3, 4}};
printf ("%d\n", sum ((int*)m, 2, 2));
```

C-ben is lehet tömböket tartalmazó tömböket csinálni, pontosabban olyan tömböket, amelyeknek elemtípusa egy mutató típus. Ez az előzőektől lényegesen különböző megoldáshoz vezet.

```
int sum (int **matrix, int rows, int cols)
{
  int i, j, s = 0;
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
      s += matrix[rows][cols];
  return s;
}
```

Ezt a függvényt már nem hívhatjuk meg kétdimenziós tömbbel. Az aktuális paramétert egész másképp – sokkal bonyolultabban – kell létrehozni, mint az előző példákban. Írhatjuk mondjuk a következőt:

```
int s0[2] = {1, 2}, s1[2] = {3, 4};
int *m[2] = {s0, s1};
printf ("%d\n", sum(m,2,2));
```

Mivel az *m* tömb elemei mutatók, egy hibás indexkifejezés könnyen vezethet ahhoz, hogy a program „kihivatkozzon" a neki rendelt memóriaterületből, ami egy futási idejű hibaként (például „Segmentation fault"-ként) jelentkezhet.

Alprogram

Egy alprogramot egy másik alprogrammal paraméterezni – ez igen célszerű lehet időnként. Képzeljünk el például egy olyan alprogramot, amelynek az a feladata, hogy függvények határozott integrálját becsülje meg. Egy függvénygörbe alatti terület nagyságára becslést adhatunk úgy, hogy meghatározzuk a függvény értékét néhány alapponton, ahogy ebben a Standard Pascal nyelven írt alprogramban tesszük:

```
function integral (function f (n: real) : real;
                   low, high, step: real) : real;
var x, sum: real;
begin
  sum := 0.0;
  for x := low to high by step
    sum := sum + f(x) * step;
  integral := sum;
end;
```

Ahhoz, hogy ne kelljen minden integrálandó függvényhez újra és újra lekódolni az integráló függvényt, alprogramot kell tudnunk átadni paraméterként. Ezt néhány korai imperatív nyelv (ASA FORTRAN, COBOL stb.) egyáltalán nem engedte meg. A funkcionális nyelvekben (beleértve már az 50-es évek végén megszületett LISP-et is), az ALGOL 60-ban, valamint számos később megalkotott nyelvben (PL/I, SIMULA 67, ALGOL 68, Pascal, FORTRAN 77, Modula-2, Modula-3 stb.) azonban már megjelent ez a lehetőség.

Jónéhány elterjedt nyelvben alprogramokat nem lehet paraméterként átadni, viszont alprogramokra hivatkozó mutatókat igen. Ilyen nyelv például a C, a C++ vagy az Ada 95. Az előbbi integráló függvényt így kódolnánk C-ben:

```
double integral (double (*f)(double), double low, double high, double step)
{
  double sum = 0.0;
  while (low <= high) {
    sum += f(low) * step;
    low += step;
  }
  return sum;
}
```

Vegyük észre, hogy az *integral* függvény törzsében az *f* által hivatkozott függvényt explicit hivatkozásfeloldás nélkül is meghívhattuk. Természetesen megengedett a *(*f)(low)* alakú hívás is. Ugyanez a kettősség megvan az Ada 95-ben is.

Egyes modern nyelvekben az alprogramok alprogrammal való paraméterezése nem annyira kézenfekvő módon történik, mint az eddig felsorolt nyelvekben. Az Ada 83-ban például csak fordítási időben kiértékelt aktuális paraméterek használhatók, mégpedig az úgynevezett „sablon" szerkezet segítségével. A Javában **interface**-ek használatával válthatók

ki a paraméter alprogramok. Erről a két utóbbi lehetőségről a 11. fejezetben tájékozódhatunk.

A paraméterként átadott alprogramok egyébként sok izgalmas kérdést vetnek fel, ezért visszatérünk még erre a témakörre a 7.7. szakaszban is.

Címke

Az ALGOL 60-ban, a PL/I-ben, a SIMULA 67-ben, az ALGOL 68-ban stb. címkéket is át lehet adni egy alprogramnak paraméterként. Tekintsük a következő ALGOL 60 függvényrészletet:

```
real procedure log(x, hiba);
  value x; real x;
  label hiba;
begin
  if x > 0
     then ...
     else goto hiba;
```

Ha a **goto** *hiba* utasításra kerül a vezérlés, az alprogram végrehajtása megszakad, sőt, minden hívó alprogram végrehajtása is, egészen addig, amíg abba a hívási környezetbe nem érünk, amelyben a címke található.

A címkeparamétert kivételkezelésre használhatjuk (lásd a 8. fejezetet). Ilyenkor a címke egy olyan kódrészletet jelöl, ami a kivételes eseményre (esetünkben arra, hogy negatív paraméterrel hívták a logaritmus függvényt) reagálni tud. Modernebb nyelvekben a címkeparamétert elegánsabb, strukturáltabb kivételkezelési mechanizmussal helyettesítik.

Típus és modul

Egy alprogram általánossága, újrafelhasználhatósága jelentős mértékben megnövelhető azzal, ha megengedjük a típussal való paraméterezhetőségét. Erről a témakörről a 11. fejezet szól.

Mi lehet visszatérési érték?

A visszatérési érték típusára a nyelvek gyakran több megszorítást tesznek, mint a paraméterek típusára. A FORTRAN-ban és az ALGOL 60-ban például a függvények csak skalár értéket adhatnak vissza. A Pascalban és a Modula-2 korai változataiban a skalárok mellett a mutatók is megengedettek. Sok imperatív nyelv azért picit engedékenyebb. Az Adában, a C-ben vagy az ALGOL 68-ban például összetett típusú értékeket is vissza lehet adni. A Modula-3-ban a visszatérési érték lehet egy alprogram is. (Az Ada 95-ben, a C-ben és a C++-ban csak egy alprogramra hivatkozó mutató.) Funkcionális nyelvekben (Haskell, LISP, ML stb.) függvényt visszaadni egy függvényből nem számít túlságosan különlegesnek. [Sco 00] Ezekben a nyelvekben, vagy például a CLU-ban, a BETA-ban, a Satherben

stb. egy függvény „több eredményt" is visszaadhat. Az alábbi CLU alprogram két egész számot vár paraméterként, és visszaadja a két számot növekvő sorrendben:

```
sort = proc (a, b: int) returns (int, int)
  if a < b then return (a, b)
  else return (b, a)
  end
end sort
```

Ezt az alprogramot így hívhatjuk meg:

```
x, y: int := sort(4, 2)
```

Végül megjegyezzük, hogy típusokkal nem csak paraméterezni lehet (lásd a 11. fejezetet). „A típusok is értékek" elvet követő nyelvekben nincs semmi különös abban, ha egy függvény visszatérési értéke egy típus. Egy ilyen függvényre később, a 219. oldalon látunk majd példát.

7.3.2. Az alprogramok specifikációja

Az alprogramot hívó programegységek számára az alprogram specifikációja az, ami elérhető. Úgy is szoktunk fogalmazni, hogy *az alprogram interfésze* a külvilág felé a specifikáció. A hívók a törzsből semmit sem látnak. Így valósul meg a vezérlési absztrakció: a hívók elvonatkoztatnak a tényleges utasítássorozattól. A specifikáció tartalmazni szokta az alprogram nevét, azt, hogy függvény-e, és ha igen, akkor milyen típusú értéket ad vissza, valamint azt, hogy az alprogramot milyen paraméterekkel kell meghívni. Szokás a specifikációt az alprogram *fejlécének* is nevezni.

function *Max(A, B: Integer)* **return** *Integer*

Egyes nyelvekben (például FORTRAN, ALGOL, Pascal stb.) az alprogramok fejléce tartalmaz egy kulcsszót, ami azt jelzi, hogy az adott szintaktikai egység egy alprogram. Más nyelvekben (például a C és leszármazottjai) nem használunk külön kulcsszót. A fenti sor helyett C-ben ezt írnánk specifikációnak:

int *max* (**int** *a*, **int** *b*)

Sőt, a visszatérési érték típusát el is hagyhatjuk a fenti specifikációból, hiszen a C-ben az alprogramok alapértelmezés szerint **int**-et visszaadó függvények[7]:

max (**int** *a*, **int** *b*)

A nyelvek között nagy eltérések mutatkoznak abban, hogy az alprogramok specifikációja mennyire információgazdag. Minél pontosabban írja le a specifikáció azt, hogy hogyan kell az alprogramot használni, annál könnyebb helyes programot írni az adott nyelven. A specifikáció segítségével dönti el ugyanis a fordítóprogram azt, hogy az alprogramot a hívó helyesen hívja-e meg, például, hogy megfelelő számú paramétert ad-e át a híváskor. Nézzük

[7] Egy másik ilyen furcsa alapértelmezett visszatérési érték típussal találkozhatunk az Objective-C nyelvben, lásd a 232. oldalt.

meg picit részletesebben, hogy milyen információkat tartalmazhat egy alprogram specifikációja a különböző nyelvekben.

A paraméterek típusa

Az egyik legfontosabb kérdés az, hogy ellenőrzi-e a nyelv fordítóprogramja, hogy az alprogram hívásakor megfelelő típusú aktuális paramétereket feleltettünk-e meg a formális paramétereknek. A FORTRAN 77 ilyen jellegű ellenőrzést még nem tartalmaz, míg a FORTRAN 90, az Ada, a Pascal, a Java és általában a legtöbb modern nyelv már igen. A C nyelv érdekes átmenetet képez a két lehetőség között. Az eredeti C nyelvben sem az aktuális paraméterek számát, sem azok típusát nem ellenőrizte a fordítóprogram. Ebben a nyelvben még a szintaxis is más volt, jobban hasonlított az ALGOL 60-ra, a PL/I-re stb[8].

```
int max (a, b)
  int a, b;
{
  return (a > b) ? a : b;
}
```

Az ANSI C viszont már támogatja az úgynevezett *prototípusok*, azaz olyan specifikációk használatát, amelyek a paraméterek típusát is tartalmazzák[9]. Erre láttunk példát a 206. oldalon. Ha az ANSI C-ben a hagyományos módon adjuk meg a formális paramétereket, akkor a fordító nem végez szám- és típusellenőrzést az aktuális paramétereken, ha azonban az új, prototípusos megoldást választjuk, akkor igen[10]. A C++-ban már csak a prototípusos változat megengedett.

Az is érdekes kérdés, hogy a paraméterek típusellenőrzése mennyire részletekbe menő. Például az ALGOL 60-ban, a korai Pascal-változatokban vagy a FORTRAN 77-ben, ha egy alprogramot akartunk átadni paraméterként, akkor a formális paraméter alprogram típusát nem tudtuk specifikálni. Az ALGOL 68-ban, a Standard Pascalban vagy a FORTRAN 90-ben viszont már igen. A 214. oldalon bemutattunk egy Standard Pascal példaprogramot, mely így nézett ki:

```
function integral (function f (n: real) : real;
                   low, high, step: real) : real;
var x, sum: real;
begin
  sum := 0.0;
  for x := low to high by step
    sum := sum + f(x) * step;
  integral := sum;
end;
```

[8] Fontos, hogy csak a szintaxisban nyilvánult meg ez a hasonlóság, hiszen az ALGOL 60-ban és a PL/I-ben az aktuális és formális paramétereknek számban és típusban meg kellett egyezniük.

[9] Ez a lehetőség valójában a C++ nyelvben jelent meg először, az ANSI C is onnan vette át.

[10] Egy paraméter nélküli alprogram esetén a prototípust az különbözteti meg a hagyományos alprogramspecifikációtól, hogy a paraméterlistában kifejezetten jelezzük, hogy nincs paraméter: **int** *f* **(void)**.

Amikor meghívjuk az *integral* alprogramot, az *f* formális paraméternek csak olyan típusú aktuálist lehet megfeleltetni, ami egy *real*-ből *real*-t előállító függvény. A fordítóprogram tehát ellenőrizni tudja, hogy az *f* függvényt, illetve a neki megfeleltetett aktuális paramétert típushelyesen használjuk-e az *integral* függvényen belül. Ezen ellenőrzés hiányában az *integral* meghívható lenne olyan paraméterrel, ami nem *real*-ből *real*-t csinál, ezért egy helyes (a fordítóprogram által elfogadott) hívás lehetne értelmetlen, típushibás is.

Az aktuális paraméterek típushelyességét befolyásolhatja az is, hogy van-e a nyelvben altípusosság, vagy általában polimorfizmus (lásd részletesebben a 11. fejezetben). A C-ben például megfeleltethetünk egy *int* típusú aktuális paramétert egy *double* típusú formálisnak: a fordító az aktuális paramétert automatikusan a megfelelő típusra alakítja át (konvertálja). Természetesen egy tömb típusú aktuális paramétert már nem feleltethetünk meg egy *double* formálisnak, mert a tömböt „nem lehet átalakítani" *double* típusra.

Az objektumorientált nyelvekben (lásd a 12. fejezetet) a típusok tartalmazhatják egymást. Ez a fajta altípusosság megengedi, hogy egy bővebb típusú (például *Síkidom*) formális paraméternek egy szűkebb típusba (például *Téglalap*) tartozó aktuális paramétert feleltessünk meg.

Az Ada nyelvben az altípusok nem típusok, és a fordítóprogram figyelmen kívül hagyja az altípusinformációkat a típushelyesség ellenőrzésekor. Ezért például egy *Integer* formális paraméternek megfeleltethető egy *Positive* altípusba tartozó aktuális, és viszont; hiszen az *Integer* és a *Positive* ugyanazt a típust jelentik.

A paraméterek száma

Vannak olyan nyelvek, melyekben az aktuális és formális paraméterek számát sem veti össze a fordítóprogram. Jellemzően ilyen nyelvek a szkriptnyelvek. Az alábbi JavaScript alprogram egy olyan függvény, amely akárhány, paraméterként megadott szövegobjektumot össze tud fűzni, elhelyezve a szövegobjektumok között az első paraméterében megadott elválasztó jelet. A függvényben az aktuális paraméterekhez az *arguments* nevű előre definiált tömb objektumon keresztül is hozzá lehet férni, a hosszát, azaz az aktuális paraméterek számát pedig a tömb *length* attribútuma adja meg. (A tömb nullától hossz mínusz egyig van indexelve.) Vegyük észre azt is, hogy az alprogram specifikációja a formális paraméterek típusát és a visszatérési érték típusát sem tartalmazza.

```
function cat (separator) {
    result = ""
    for (var i = 1; i < arguments.length; ++i) {
        result += arguments[i]
        if (i != arguments.length) result += separator
    }
    return result
}
```

Mint már említettük, az eredeti a fordítóprogram nem ellenőrizte, hogy az alprogramokat megfelelő számú (és típusú) aktuális paraméterrel hívták-e meg. Ez azonban korántsem

biztosította azt a fajta rugalmasságot, mint amit a JavaScriptben íródott példában tapasztalunk, hiszen a C nyelvű alprogramban nem rendelkeztünk az aktuális paraméterekről semmiféle információval. A C-ben és a C++-ban azonban van egy másik lehetőség is. Ha a formális paraméterek listájának végén . . .-ot írunk, akkor ezzel tetszőleges számú formális paramétert helyettesíthetünk. Az ilyen jelet tartalmazó alprogramoknak legalább egy rendes formális paraméterének lennie kell. Általában az ennek (vagy ezeknek) megfeleltetett aktuális paraméter (vagy paraméterek) adnak információt arról, hogy milyen egyéb aktuális paraméterek vannak. Jó példa erre a módszerre a már bemutatott szabványos könyvtárbeli *printf* függvény, melynek specifikációja a következő:

int *printf* (**char** **format*, . . .)

A *format* karaktertömbben vezérlőjelek találhatók, amelyek megadják, hogy milyen aktuális paraméterekre kell számítania az alprogramnak. Ezek a vezérlőjelek a százalék karakterrel kezdődnek. Például a `"%d"` egy egész típusú aktuális paramétert jelez, amelyet decimálisan kell kiírnia a *printf*-nek, míg a `"%s"` egy karaktertömb aktuális paramétert jelez. Néhány helyes meghívása ennek a függvénynek:

```
printf ("10*10=%d", 10*10);
printf ("%s=%d","10*10", 10*10);
printf ("nekem nem kell parameter");
```

Abban az esetben, amikor a *format* karaktertömbben található vezérlőjelek és a megadott további aktuális paraméterek nincsenek összhangban, a *printf* alprogram működése definiálatlan – sajnos, a fordítóprogram nem képes felderíteni az ilyen jellegű hibákat. Vannak azonban olyan nyelvek is, amelyekben a típusrendszer és a fordítóprogram már elég bonyolult ahhoz, hogy kifejezzünk olyan dolgokat, hogy egy alprogram paraméterének típusa függ egy másik paraméter értékétől. Ilyen nyelv például a Cayenne, amelynek típusrendszere a „függő típus" (dependent type) fogalmán alapul. Az alábbi példában [Aug 99] a `printf` függvény első (`fmt` nevű) paraméterének típusa `String`. A többi paraméter típusa és a visszatérési érték típusa függ ezen paraméter értékétől: `PrintfType fmt`. Például, ha az `fmt` karakterlánc a `%d` karakterekkel kezdődik, akkor a `printf` egy `Int` típusú második paramétert vár, és a további paraméterek, illetve a visszatérési érték típusát az `fmt` további része határozza meg. Ha például az `fmt` további része az üres karakterlánc (azaz a `Nil`), akkor nincs több formális paraméter, és a visszatérési érték típusa a `String` lesz.

```
PrintfType :: String -> #
PrintfType (Nil)          = String
PrintfType ('%':('d':cs)) = Int    -> PrintfType cs
PrintfType ('%':('s':cs)) = String -> PrintfType cs
PrintfType ('%':( _ :cs)) = PrintfType cs
PrintfType (_ :cs)        = PrintfType cs

printf :: (fmt::String) -> PrintfType fmt
...
```

Egyes nyelvekben alprogram formális paramétereknek alapértelmezett érték adható. Ezzel (egy kis) részben elérhető az a rugalmasság, amit az előző példákban láttunk. A 228. oldalon visszatérünk még erre a kérdésre.

Más nyelvekben egy tömb átadásával lehet utánozni a változó hosszúságú paraméterlistát. Ez a lehetőség erősen típusos nyelvekben csak azonos típusú paraméterekhez használható, de gyengén típusos nyelvekben, például a Smalltalkban egy tömb segítségével többb, akármilyen típusú paramétert átadhatunk. Az erősen típusos C#-ban szintén tömbök segítségével utánozhatjuk a változó hosszúságú paraméterlistát. A **params** kulcsszó segítségével elérhetjük azt, hogy a hívás helyén ne kelljen tömbnek szerepelnie:

```
void ShowNumbers (params int[ ] numbers) {
    foreach (int x in numbers) {
        Console.Write (x + " ");
    }
    Console.WriteLine();
}
...
int[ ] x = {1, 2, 3};
ShowNumbers (x);
ShowNumbers (4, 5);
```

A C és C++ nyelvekben a főprogramok rendelkeznek még egy érdekes tulajdonsággal, már ami a formális és aktuális paraméterek számát jelenti. Ezekben a nyelvekben a főprogramot egy *main* nevű függvény jelenti, amelynek paraméterlistája kétféle lehet. Ha nem akarunk a programból hozzáférni a parancssori argumentumokhoz, akkor a *main* specifikációja a következőképpen nézhet ki:

```
int main()
```

Ha viszont szeretnénk használni a parancssori argumentumokat, akkor a *main*-t így kell specifikálnunk:

```
int main (int argc, char* argv[ ])
```

Itt a parancssori argumentumok (mint karaktertömbök) számát (*argc*) és tömbjét (*argv*) kapjuk meg az operációs rendszertől aktuális paraméterként. A Javában az első megoldás – a parancssori argumentumok figyelmen kívül hagyása a formális paraméterlistában – nem működik. Egy főprogramot a Javában így írnánk:

```
class Argumentumok {
  public static void main(String[ ] args) {
    for (int i=0; i < args.length; ++i)
      System.out.println(args[i]);
  }
}
```

Más nyelvekben (például Ada, Pascal, Modula-2) a főprogram nem vehet át paramétereket. Ilyenkor általában könyvtári egységek segítik a programozót abban, hogy a parancssori argumentumokat kezeljük. A fenti Java alprogram így írható át Adára:

```
with Ada.Command_Line, Ada.Text_IO;
procedure Argumentumok is
begin
   for I in 1..Ada.Comand_Line.Argument_Count loop
      Ada.Text_IO.Put_Line(Ada.Command_Line.Argument(i));
   end loop;
end Argumentumok;
```

A paraméterek neve (név szerinti megfeleltetés)

A legtöbb nyelvben a formális paraméterek neve, bár része az alprogram specifikációjának, nem a hívó programegységeknek szól: a hívás érvényességének megállapításához a fordítóprogram csak a paraméterek számát és típusát használja fel. (Ezt szokás egyébként az alprogram *profiljának* nevezni. Az alprogram profilját, és – ha függvényről van szó – a visszatérési érték típusát szokták az alprogram *protokolljának* hívni.) A formális paraméterek neve ezekben a nyelvekben csak informális információt hordoz, ami a programozó számára megkönnyíti a formális paraméterek szerepének megjegyzését.

Ezzel szemben például az Adában, a PL/SQL-ben vagy a Modula-3-ban az alprogram specifikációja az alprogram formális paramétereinek nevét is felhasználhatóvá teszi a hívó programegységekben. Ugyanis ezekben a nyelvekben egy alprogram hívásakor az alprogram formális paramétereinek nem csak *pozicionálisan*, hanem *névvel jelölt formában* is meg lehet feleltetni az aktuálisokat. A pozicionális megfeleltetés azt jelenti, hogy az első formális paraméternek az első aktuálist feleltetjük meg, a másodiknak a második aktuálist, és így tovább. A névvel jelölt forma esetében nem kell ragaszkodni a sorrendhez. Legyen adott a következő Ada nyelvű alprogram-specifikáció a szabványos könyvtárból, mely egy karakterláncba egy megadott pozíció előtt befűz egy másik karakterláncot:

```
function Insert(Source   : in String;
                Before   : in Positive;
                New_Item : in String) return String
```

A következő függvényhívások mind helyesek:

```
Insert("Kacsa", 3, "lo") -- pozicionális megfeleltetés
Insert(Source => "Kacsa", Before => 3, New_Item => "lo")
Insert(Before => 3, Source => "Kacsa", New_Item => "lo")
Insert(New_Item => "lo", Before => 3, Source => "Kacsa")
```

A programozónak nem kell emlékeznie a paraméterek pontos sorrendjére; ha tudja, hogy milyen nevű formális paraméterek vannak, tetszőleges sorrendben megadhatja az ak-

tuálisokat. Sőt, a pozicionális és a névvel jelölt formát keverheti is. Ilyenkor az első néhány paramétert pozicionálisan, a többit névvel jelölt formában feleltetheti meg:

Insert(`"Kacsa"`*, New_Item =>* `"lo"`*, Before => 3)*

A név szerinti paramétermegfeleltetés különösen hasznos az alapértelmezett értékkel rendelkező formális paraméterekkel együttesen használva. (Lásd még a 228. oldalt.)

Kiváltható kivételek

Egy alprogram „speciális visszatérési értékeiként" foghatók fel azok a kivételek, amelyeket az alprogram kiválthat. A hívónak esetleg fel kell készülnie a fellépett kivételek kezelésére. Teljesen indokolt tehát, hogy az alprogram által kiváltható kivételek is megjelenjenek az alprogram specifikációjában. Az Adában nem lehet, a C++-ban nem kötelező, de például a Javában kötelező[11] megadni a kiváltható kivételeket az alprogram fejlécében:

int *beolvas(InputStream in)* **throws** *IOException*

A *beolvas* művelet egy *InputStream*-et vár paraméterként, és általában egy **int**-et ad vissza. Kivételes esetben azonban a hívás egy *IOException* kivétel fellépésével ér véget.

A kivételekről részletesebben a 8. fejezet szól.

Az objektumorientált nyelvek különlegességei

Egyes nyelvekben az alprogramok specifikációja egyéb információkat is tartalmazhat. Az objektumorientált nyelvekben (C++, Java, C#) például az alprogram (ami általában az osztálydefiníció egy eleme, úgynevezett tagja) láthatóságára tehetünk megszorítást. Gyakran a **public**, **protected** és **private** kulcsszavakat szokás használni a láthatóság jelzésére. A **public** a legnagyobb, a **private** a legkisebb láthatósági körrel ruházza fel az alprogramot. Például a Javában:

public void *start()*

Az objektumorientált nyelvekben a **static** kulcsszó is megjelenhet az alprogram specifikációjában. Ezzel szokás azt jelölni, hogy az alprogram nem egy objektumon, hanem egy osztályon hajtható végre.

public static int *exit()*

Az objektumorientált nyelvekről a 12. fejezetből tudhatunk meg többet.

7.3.3. Alprogramok törzse

Az alprogramok törzse tartalmazza azokat az utasításokat, amelyeket az alprogram minden egyes hívásakor végre kell hajtani. Általában lehetőségünk van az alprogramban úgynevezett *lokális változókat* definiálni, melyeket az alprogram által végzett számításokhoz használhatunk segédváltozókként. A változók deklarációja és az utasítások, amelyek használják ezeket

[11] Egész pontosan csak az úgynevezett „ellenőrzött" kivételeket kötelező megadni az alprogram specifikációjában.

a változókat egyes nyelvekben teljesen elkülönülnek egymástól (például Pascal, Ada), más nyelvekben kevésbé (például C), végül egyes nyelvekben (például Java, C++) egyáltalán nem. Egy Pascal alprogram törzse például két részből áll: egy deklarációs részből, ahol – többek között – a lokális változókat definiáljuk, és az utasítássorozat részből. A két részt a **begin** kulcsszó választja el egymástól.

```
procedure swap(var a, b: Integer);
var temp: Integer;
begin
  temp := a;
  a := b;
  b := temp;
end;
```

A C-ben nem írunk semmit a változódeklarációk és az utasítások közé. Egyszerűen csak meg kell, hogy előzzék a változódeklarációk az utasításokat.

```
void swap (int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

A Javában például ez a megkötés sem létezik. A deklarációs utasításokat szabadon lehet keverni az egyéb utasításokkal.

Az Adában (vagy például a PL/SQL-ben) a kivételkezelés (lásd a 8. fejezetet) nem utasítás (mint például a Modula-3-ban, a C++-ban vagy a Javában), hanem egy blokkhoz rendelhető függelékként. Ennek megfelelően egy alprogram törzse az Adában három részből is állhat: deklarációs részből, utasítás részből és egy opcionális kivételkezelő részből. Az alábbi alprogram egy verem művelete. A vermet egy rekorddal ábrázoljuk, amelynek *Data* mezője egy tömb: ebbe a tömbbe tesszük bele a tömb elemeit. A rekord *Top* mezője a legutóbb betett elem tömbbeli indexe. A tömb méretét a verem típus definíciójában, vagy esetleg a verem objektum létrehozásakor rögzítjük. Ha túl sok elemet próbálunk meg betenni a tömbbe, akkor a tömböt túlindexeljük, amit a *Constraint_Error* kivétel jelez. Ilyenkor ki kell váltanunk a *Stack_Overflow* nevű kivételt.

```
procedure Push (S: in out Stack; E: in Element) is
begin
  S.Top := S.Top + 1;
  S.Data(S.Top) := E;
exception
  when Constraint_Error => S.Top := S.Top-1; raise Stack_Overflow;
end Push;
```

Ebben az alprogram-definícióban a deklarációs rész üres, az utasítás rész két darab utasítást tartalmaz, és van kivételkezelő rész is.

Jónéhány nyelvben (például Eiffel, Turing, Blue) az alprogramok törzse helyességbizonyításhoz használható logikai állításokat is tartalmazhat. Erre a kérdésre a 13. fejezetben fogunk részletesebben kitérni.

Visszatérés az alprogramból

Egy függvényből való visszatéréskor vissza kell adni a hívónak egy megfelelő típusú értéket. Ennek formája igen változatos szokott lenni a különböző nyelvekben. A leggyakoribb talán a **return** utasítás használata. Ez szerepel például az Adában (lásd a 203. oldalt), a C-ben (lásd a 206. oldalt), a JavaScriptben (lásd a 218. oldalt), és a legtöbb egyéb modern imperatív nyelvben. A **return** utasítás hatására az alprogram azonnal véget ér[12] és visszaadja a **return** után írt kifejezés értékét.

A korai imperatív nyelvekben, például a FORTRAN-ban, az ALGOL 60-ban és a Pascalban a **return** utasítás helyett a függvénynévre vonatkozó értékadást kellett írni (lásd például a 214. oldalt). Ez azért nem volt szerencsés, mert a függvény nevét a függvény törzsében nem volt szabad elfedni, és ez egy picit elbonyolította a deklarációkra vonatkozó elfedési szabályokat[13].

De még a **return** utasításnak is megvan a maga problémája. Ha egy függvény a számításának egy pontján már tudja, hogy mit szeretne visszaadni, de a számítást valamilyen oknál fogva tovább kell folytatnia (például egy fájlt be kell zárnia, vagy egy erőforrást el kell engednie), akkor kénytelenek vagyunk egy lokális változót felvenni, ami ideiglenesen tárolja a visszaadásra szánt értéket. Hasonló probléma lép fel a következő Ada függvényben is:

```
function Max_Hely (V: Vektor) return Index is
  Rtn: Index := V'First;
begin
  for I in V'Range loop
    if V(I) > V(Rtn) then Rtn := I; end if;
  end loop;
  return Rtn;
end Max_Hely;
```

[12] Kivételt képez ez alól a szabály alól az, ha a **return** utasítás egy olyan kivételkezelt utasítássorozatban található, például a Javában egy **try** blokkban, amelyhez tartozik **finally** rész. Ekkor ugyanis a **finally** rész még végrehajtódik az alprogramból való kilépés előtt. Hasonlóképpen, C++-ban az alprogramban létrehozott lokális objektumok destruktora is lefut a kilépés előtt.

[13] A FORTRAN-ban van `RETURN` utasítás is, melynek segítségével az alprogram vissza tud térni. Ezen utasítást azonban nem használhatjuk a visszatérési érték megadására, mert azt a függvénynévre vonatkozó értékadással állíthatjuk be.

Sokkal hatékonyabb megoldást biztosít az SR nyelv[14] [Sco 00].

```
procedure Max_Hely (ref V[1:*]: int) returns rtn: int
  rtn := 1
  fa i := 1 to ub(V) ->
     if V[i] > V(rtn) -> rtn := 1 fi
  af
end
```

Itt változóként kezelhettük azt a memóriaterületet, amin keresztül a függvény eljuttatja a hívónak a visszatérési értékét. Az Eiffelben a függvény visszatérési értékét szintén lokális változóként kezelhetjük, de a változó nevét nem határozhatja meg a programozó: a név mindig a *Result* lesz.

```
max_hely (v: ARRAY[INTEGER]) : INTEGER is
  local
    i: INTEGER
  do
    Result := v.lower;
    from i := v.lower until i>v.upper loop
      if v@Result < v@i then
        Result := i
      end
    end
  end
```

Az Euclid nyelv érdekesen ötvözi a fenti lehetőséget a **return** utasítás használatával. Ha az alprogram fejlécében megadunk egy nevet a visszatérési értékhez, akkor arra a névre vonatkozó értékadást kell írnunk az alprogram törzsébe, ha viszont nem adunk meg nevet, akkor a **return** utasítás segítségével állítható be visszatérési érték.

A funkcionális nyelvekben és az ALGOL 68-ban (ami nem tesz különbséget utasítás és kifejezés között) a függvények visszatérési értéke a függvénytörzs, mint kifejezés értékével egyezik meg. A két szám maximumát meghatározó függvény így néz ki ALGOL 68 nyelven:

```
proc max = (int a, b)int:
  if a > b then a else b fi
```

Kanyarodjunk vissza még egy percre a **return** utasításhoz. Gyakran nem csak függvényekben, hanem eljárásokban is használhatjuk ezt az utasítást, de természetesen ilyenkor önmagában, visszaadott kifejezés nélkül. Például, ha egy alprogramból a kód kellős köze-

[14] A példaprogramban a *fa* jelentése „for all", azaz „mindegyikre", az *ub* jelentése pedig „upper bound", azaz „felső korlát". A nyíl felel meg annak, amit más nyelvben például **do**-nak és **then**-nek nevezünk. Az összetett utasítások lezárására a nyelv az utasítást kezdő kulcsszó fordítottját használja.

pén szeretnénk azonnal „kiugrani", visszatérni, írjunk valami olyasmit, mint ez az Ada 95 eljárás:

```
procedure Nullat_Keres (M: in Matrix; I, J: out Index) is
begin
  I := M'First(1);
  while I <= M'Last(1) loop
    J := M'First(2);
    while J <= M'Last(2) loop
      if M(I,J) = 0.0 then
          return;
      end if;
      J := J + 1;
    end loop;
    I := I + 1;
  end loop;
end;
```

Egy eljárásban nem kötelező szerepelnie **return** utasításnak, de egy függvényben igen. A Javában például fordítási hibát okoz, ha a fordító szerint egy függvény véget érhet úgy, hogy nem hajtódott végre **return** utasítás.

```
int nemFordulokLe() {
   int i = 42, j = 42 * i - 42 * (i - 1);
   if (i == j)
       return 1;
}
```

Ez a függvény jó, de nem helyes – abban az értelemben, hogy a fordító nem fogadja el, noha mi, programozók tudjuk, hogy a vezérlés mindenféleképpen eljut a **return** utasításhoz. Az Adában nem ilyen szigorú a fordító. Egy hasonló függvénydefinícióra legfeljebb egy figyelmeztetést kapnánk fordításkor. Ha azonban egy függvényről futási időben kiderül, hogy elkerülte a **return** utasítást, akkor az Ada futtató rendszere egy *Program_Error* kivételt vált ki.

Más nyelven írt törzs

Néha egy alprogram törzsét más nyelven szeretnénk elkészíteni, mint amilyen nyelvű programegységből hívni szeretnénk. Ennek lehetnek hatékonysági okai (például egy bizonyos kódrészletnek nagyon hatékonynak kell lennie, ezért assembly nyelven akarjuk megírni), vagy újrafelhasználási okai is (már meg van írva egy nyelven az alprogram, és szeretnénk használni újraírás nélkül egy másik nyelvből).

Az Adában például erre a célra való fordítóvezérlő direktívákat és könyvtári egységeket használhatunk arra, hogy egy más nyelven megírt alprogramot meghívhassunk az Ada

programunkból, vagy fordítva. Ha az *Sqrt* függvény mondjuk FORTRAN-ban íródott, az *Import* fordítóvezérlő direktíva segítségével tehetjük láthatóvá Ada programunkban.

```
function Sqrt (X: Float) return Float;
pragma Import(FORTRAN, Sqrt);
```

Nézzünk egy másik példát: A Javában egy alprogram specifikációja tartalmazhatja a *native* kulcsszót, ami azt jelenti, hogy az alprogram törzse nem Javában íródott:

```
public native float sqrt(float x);
```

A Java virtuális gépes szemlélete azt is lehetővé teszi, hogy más nyelven (például Adában) írt programjainkat minden különösebb erőfeszítés nélkül építsük be Java programjainkba. Csak arra van szükség, hogy ahhoz a másik nyelvhez is létezzen olyan fordítóprogram, ami a Java Virtuális Gép gépi kódjára képes fordítani (ilyen például a jgnat az Adához). Ugyanez a működési elve a .NET architektúrának is: mindegy, hogy egy bizonyos kódrészletet milyen programozási nyelven szeretnénk megírni vagy látni, a jelentése egy „közös nyelv", a Common Language Runtime segítségével lesz megfogalmazva.

7.3.4. Az alprogramok meghívása

Az alprogram elindítása (aktiválása) általában az alprogram nevének és aktuális paramétereinek megadásával történik. Néhány korai imperatív nyelvben, például a FORTRAN-ban és a PL/I-ben az eljárások meghívásához a `CALL` utasításra is szükség van:

```
CALL SWAP (A, B)
```

A paraméterlistát a legtöbb nyelvben (és) jelek közé szokták zárni[15]. Általában az üres paraméterlistát is jelölni szokták, üres zárójelpárral. Például az alábbi C „függvény" beolvas egy számot, és kiírja a négyzetét:

```
#include <stdio.h>
void negyzetre_emel() {
   int n;
   scanf("%d",&n);
   printf("%d",n*n);
}
```

Ezt az alprogramot így hívnánk meg:

```
negyzetre_emel();
```

[15] Ez alól kivételt jelentenek többek között a funkcionális nyelvek. A LISP-ben például nem az aktuális paraméterlistát zárójelezzük, hanem az egész hívást: `(+ 4 2)`. A modern funkcionális nyelvekben, például a Haskellben a függvénynév és a paraméterek közé csak szóközt teszünk, mint az `exp 5` kifejezésnél. Ez utóbbi forma azért nagyon furcsa, mert a függvényhívás a legmagasabb precedenciájú művelet ezekben a nyelvekben. Ennek ellenére azt a két dolgot (nevezetesen a hívott függvény nevét és a paramétereket), amelyeket a legszorosabb kapcsolat fűz össze, egy jellemzően elválasztásra, elkülönítésre használt jellel, a szóközzel kapcsoljuk össze.

Néhány nyelvben az üres paraméterlistát jelző zárójelpárt nem kell (PL/I) vagy nem szabad (Ada) kiírni. Az Adában például a négyzetre emelő alprogramot így készíthetnénk el:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Negyzetre_Emel is
  N: Integer;
begin
  Get(N);
  Put(N*N);
end Negyzetre_Emel;
```

Meghívni pedig így kell ezt az alprogramot:

```
Negyzetre_Emel;
```

A formális paraméter alapértelmezett értéke

Vannak olyan nyelvek, amelyekben a bemenő formális paraméterekhez alapértelmezett értéket lehet rendelni. Ilyenkor nem kötelező az összes formális paraméternek aktuálist megfeleltetni: az aktuális paraméter nélkül maradt formálisok fel tudják venni az alapértelmezett értéket – már ha van hozzájuk olyan rendelve. Ezekben a nyelvekben persze nem annyira rugalmas az aktuális és formális paraméterek számának (és típusának) a kezelése, mint a C/C++-ban a . . ., vagy a JavaScriptben az *arguments* tömb használatával. Ennek az az oka, hogy egy ilyen alprogramnak mindig ugyanannyi paramétere van, csak éppen nem kell mindet explicit módon megadni. Nézzük az alábbi C++ példát:

```
void draw_rectangle (int width, int height,
                     int x = 0, int y = 0, int color = 0)
```

Egy ilyen alprogram meghívásakor elég a *width* és a *height* formális paramétereknek aktuálist megfeleltetni – a többinek is lehet, de nem kötelező. Az alábbiak az utolsó kivételével mind helyes hívások:

```
draw_rectangle (10, 10);
draw_rectangle (10, 10, 20)
draw_rectangle (10, 10, 20, 15)
draw_rectangle (10, 10, 20, 15, 3)
draw_rectangle (10) // Ez hibás, height nem kapott aktuális értéket.
```

Akkor nagyon hasznos ez a lehetőség, ha vannak olyan alprogramjaink, amelyeknek nagyon sok formális paraméterük van, de ezek a paraméterek a hívások többségében mindig ugyanazt az értéket szokták felvenni. Azaz vannak olyan paraméterek, amelyek megadása fontos (ilyen volt a példánkban a *width* és a *height*), és vannak olyanok, amelyek kevésbé fontosak (ilyen volt a többi).

A C++-ban az a szabály, hogy ha egy formális paraméter alapértelmezett értékét használjuk egy hívásban, akkor az összes utána következő formális paraméterhez is az alapértelme-

zett értéket kell használni. Ezek szerint a paraméterlista végén kell elhelyezni az alapértelmezett értékkel rendelkező formális paramétereket, és célszerű csökkenő fontossági sorrendet tartani.

Az Adában, a Modula-3-ban stb. a név szerinti paraméter-megfeleltetéssel kombinálhatjuk az alapértelmezett paraméterértékeket. Így nagyon rugalmasan lehet hívni alprogramokat. A C++-ban például nem írhatnánk ezzel az Ada eljáráshívással egyenértékű hívást:

```
Draw_Rectangle(10, 10, color => 3);
```

Azokban a nyelvekben, ahol nem lehet a formális paraméterekhez alapértelmezett értéket rendelni, az alprogramnevek túlterhelése jelenthet – egy kevésbé elegáns és rugalmas – alternatívát. Erre a kérdésre még visszatérünk a 7.6. szakaszban.

Belépési pontok

Néhány nyelvben, például a PL/I-ben és a FORTRAN 77-ben egy alprogramhoz több belépési pontot is lehet definiálni. Ez azt jelenti, hogy az alprogramot nem csak „az elején lehet elkezdeni", hanem a „közepén" is. A belépési pontokat az alprogram törzsén belül bárhol elhelyezhetjük, rendelkezhetnek paraméterlistával, és ugyanúgy meghívhatók, mint maga az alprogram. A belépési ponthoz tartozó törzs a belépési ponttól kezdődik, és a befoglaló alprogram végéig tart. A befogadó alprogram lokális változói és utasításai a belépési pont törzse számára is elérhetők.

A belépési pontok segítségével például utánozhatjuk az alapértelmezett értékkel rendelkező formális paramétereket. Például az alábbi FORTRAN 77 alprogram megnövel egy változót egy adott értékkel, illetve eggyel, ha az alapértelmezett növekményt használjuk.

```
      SUBROUTINE INCWITH (K, L)
        INTEGER K, L, I
        I = L
        GOTO 10
      ENTRY INC (K)
        I = 1
10      K = K + I
      END
```

Az alapértelmezett értékkel rendelkező formális paramétereket tartalmazó alprogramokkal szemben a FORTRAN 77 eljárásunkat két különböző néven kell meghívni a két különböző esetben.

```
      CALL INCWITH (I, 3)
      CALL INC (I)
```

Az eljárás törzsének „lelke", mely mindkét hívásnál végrehajtódik, a 10-es címkétől kezdődik, és a mi esetünkben most csak egy utasítást tartalmaz.

A PL/I több belépési ponttal ellátott alprogramjai már a későbbi nyelvek modulfogalmához közelítettek. Segítségükkel absztrakt objektumokat is elegánsan meg lehetett valósítani [Koz 92a]:

```
STACK: PROCEDURE(SIZE);
  DECLARE SIZE FIXED BIN;
  DECLARE S(*) CHAR(1) CONTROLLED,
        N FIXED BIN STATIC INIT (0);
  ALLOCATE S(SIZE);
  RETURN;
PUSH: ENTRY(X);
  DECLARE X CHAR(1);
  N = N+1;
  S(N) = X;
  RETURN;
POP: ENTRY RETURNS (CHAR(1));
  N = N-1;
  RETURN (S(N+1));
END;
```

Ez az alprogram egy veremobjektumot határoz meg, elrejtett megvalósítással. Az objektum létrehozása a `STACK` belépési ponton keresztül történik. Meg kell adni a verem maximális méretét. A `CONTROLLED` változók létrehozása futási időben történik az `ALLOCATE` utasítással, melynek megadható futási időben a létrehozandó változó mérete. A `STATIC` változók az alprogram többszöri végrehajtása között is megőrzik értéküket. A fenti alprogramot így használhatnánk:

```
DECLARE C CHAR(1);
CALL STACK(10);
CALL PUSH('A');
CALL PUSH('B');
C = POP;
```

A modernebb nyelvek a belépési pontokat strukturáltabb, elegánsabb szerkezetekkel váltották fel, például alapértelmezett értékkel rendelkező formális paramétert tartalmazó alprogramokkal, és adatabsztrakciót támogató modulokkal, osztályokkal (lásd a 10. fejezetet).

Curry-módszer

A funkcionális nyelvekben egy alprogramra történő hivatkozáskor nem feltétlenül szükséges az összes aktuális paramétert megadni. Annak a kifejezésnek az értéke, amely egy félig kitöltött paraméterlistájú függvényhívást tartalmaz, egy újabb, immár kevesebb paramétert váró függvény lesz. Erről a Curry-módszernek (currying) nevezett lehetőségről a 16. fejezetben olvashatunk még. Hasonlót egyébként a Sather nyelvben is csinálhatunk. A `bind` segítségével egy művelethez (például `plus`) egy paramétert (`3`) megadhatunk, míg egy má-

sik paramétert szabadon hagyhatunk. Ez utóbbi paramétert az aláhúzásjel helyettesíti. Az a függvény már csak egy paramétert vár, ami `INT` típusú, és ugyanilyen típusú értéket ad vissza:

```
a:ROUT{INT}:INT := bind(3.plus(_));
```

Az így definiált alprogramot később a `call` segítségével meghívhatjuk, úgy, hogy a szabadon hagyott paraméter(ek)nek megadjuk az értékét:

```
x: INT := a.call(4); – x értéke 7 lesz
```

Fontos megkötés, hogy a kimenő paramétereket (beleértve a be- és kimenő paramétereket is) muszáj szabadon hagyni a `bind` használatakor, mert ezekben a paraméterekben az alprogram tényleges végrehajtásakor fogunk valamilyen értéket visszakapni.

Az alprogramok meghívása az objektumorientált nyelvekben

Az objektumorientált nyelvekben (lásd a 12. fejezetet) az alprogramok gyakran egy objektumhoz kötődnek, egy „objektumon hívjuk meg őket", és nem alprogramoknak, hanem metódusoknak nevezzük őket. Ezért az első paramétere ezeknek a műveleteknek nem a paraméterlistába, hanem az alprogram neve elé kerül. Például egy *Pont* típusú *p* objektumon az *eltol* műveletet így hívnánk meg a Javában:

```
p.eltol(4.0, 2.0);
```

Ugyanez a hívás Ada nyelven így nézne ki:

```
eltol(p, 4.0, 2.0);
```

Az objektumorientált nyelvekben egy művelet törzsén belül arra az objektumra, amelyikre meghívták a műveletet, egy erre a célra való kulcsszóval (például *this*, *self*, *Current* vagy *me*) lehet általában hivatkozni. Az a formai megkülönböztetés, ami a művelet elé írt objektumra, illetve a többi paraméterre vonatkozik, gyakran jelentésbeli különbséget is takar. Az objektumorientált nyelvekben ugyanis van lehetőség arra, hogy egy alprogram meghívásakor a futtató rendszer a művelethez tartozó több (alternatív) törzs közül válassza ki azt, amelyiket végre kell hajtani (lásd: késői vagy dinamikus összekapcsolás). Ez a választás pedig azon objektumnak, amire a művelet vonatkozik, dinamikus (azaz csak futási időben megismert) típusán alapul.

Néhány C-származék objektumorientált nyelvben nagy divat az, hogy **void** visszatérési értékű műveletet szinte sosem írunk. Ha egy objektum egy művelete egyébként nem adna vissza semmilyen értéket, akkor is úgy írjuk meg, hogy visszaadjon valamit: az objektumot magát. Ezzel a módszerrel kényelmesen lehet „halmozni a transzformációkat" az objektumon:

```
téglalap.eltol(4.0,2.0).tükröz(0.0,0.0).nagyít(2.0,0.5);
```

Ehhez a *Téglalap* osztályban így készítenénk el a műveleteket – Java szintaxist használva:

```
public Téglalap eltol (double dx, double dy) {
  xpos += dx;  ypos += dy;   return this;
}
```

Végül megjegyezzük, hogy egyes nyelvek egészen egzotikus szintaxist biztosítanak az alprogramok nevének és formális paraméterlistájának megadására. Például az Objective-C nyelvben lehet ilyen alprogram-specifikációt írni:

```
- putObject: element atX: (int) x Y: (int) y
```

Ez egy példányszintű metódus (objektumhoz rendelt alprogram, lásd a 12. fejezetet). A példányszintű metódusok a - jellel, az osztályszintű metódusok a + jellel kezdődnek. A metódus neve a következő:

```
putObject:atX:Y:
```

A metódusnak három paramétere van, amelyből az utolsó kettő (*x*, és *y*) egész, míg az első paraméter (*element*) és a visszatérési érték objektumhivatkozás típusú (*id*). Amennyiben egy metódus visszatérési értékének típusát nem adjuk meg, mindig az általános *id* típust feltételezi a nyelv. Ebben a nyelvben például szokásos, hogy a metódusok, ha nincs mit visszaadniuk, visszaadják azt az objektumot, amire meghívták őket. Egyébként a metódusok meghívása is érdekesen történik az Objective-C nyelvben. Például, ha a fenti metódus mátrix típusú objektumokon hívható meg, és *m* ilyen típusú, valamint *e* egy objektum, akkor az alábbi utasítást írhatnánk.

```
[m putObject: e atX: 5 Y: 3];
```

7.3.5. Rekurzív alprogramok

Azokat az alprogramokat, amelyek közvetlenül vagy közvetve önmagukat hívják, rekurzív alprogramoknak nevezzük. (Lásd a 3.11. szakaszt.) Egy rekurzív alprogram több példányban is aktív lehet a programvégrehajtás egy adott pontján, ami bonyolultabbá és költségesebbé teszi a megvalósításukat, mint a rekurziót nem használó alprogramokét. Rekurzív alprogramok segítségével bonyolult számításokat egyszerűen, könnyen megérthetően és matematikai eszközökkel (például indukcióval) könnyen kezelhetően programozhatunk le. A rekurzió alternatívát jelent a ciklushoz képest. Az alábbi Ada függvény rekurzió segítségével számolja ki egy tömb elemeinek összegét:

```
type Tömb is array (Integer range <>) of Integer;
function Sum (V: Tömb) return Integer is
begin
  if V'First >= V'Last
    then return V(V'First);
    else return V(V'First) + Sum(V(V'First+1..V'Last));
  end if;
end Sum;
```

Az első néhány programozási nyelv tervezői nem ismerték fel a rekurzióban rejlő lehetőséget [Set 96], vagy csak nagyon költségesnek ítélték a megvalósítását. A FORTRAN-ban (egész a FORTRAN 90-ig) nem lehetett rekurzív alprogramokat írni. A FORTRAN 90-ben és a PL/I-ben lehet, de külön jelezni kell az alprogram specifikációjában, hogy rekurzív. Ezzel a nyelv lehetőséget ad a fordítóprogramnak, hogy a nem rekurzív alprogramokat sokkal hatékonyabb kódra fordítsa. Az alábbi PL/I függvény faktoriálist számol:

```
FACT : PROCEDURE (N) RECURSIVE RETURNS (FIXED BIN);
  DECLARE N FIXED BIN;
  IF N=0 THEN RETURN(1);
         ELSE RETURN(N * FACT(N-1));
END;
```

7.3.6. Az alprogramok deklarációja

Egyes nyelvekben létezik egy olyan megkötés, hogy minden egyedről, ami a programunkban előfordul, használat előtt meg kell mondanunk, hogy micsoda. Ezt szaknyelven úgy szokás mondani, hogy az egyedeket deklarálnunk kell használat előtt. Ezekben a nyelvekben igaz ez az alprogramokra is. Vannak azonban olyan helyzetek, amelyekben az alprogram definíciójának megadásával nem tudjuk deklarálni az alprogramot. Például közvetett rekurzió esetén, azaz amikor mondjuk egy A alprogram hív egy B alprogramot, ami hívja az A alprogramot, nem lehet a két alprogram definícióját úgy megadni, hogy a használat előtt az alprogramok már definiáltak legyenek[16]. Ilyenkor segíthet egy *előrevetett deklaráció* (forward declaration), mely lényegében az alprogram specifikációjából áll, azaz csak arról ad információt, hogy az alprogramot hogyan kell használni. Illusztrálja ezt az elvet az alábbi, nem túl ötletes, de kellőképpen egyszerű Ada kódrészlet:

```
procedure Print_Tree (T: in Tree);
procedure Print_Children (T: in Tree) is
begin
  Print_Tree(Left(T));
  Print_Tree(Right(T));
end Print_Children;
procedure Print_Tree (T: in Tree) is
begin
  if not Empty(Tree) then
    Print_Node(Node(T));
    Print_Children(T);
  end if;
end Print_Tree;
```

Egy másik helyzet, amikor egy alprogramot nem a definíciójával szeretnénk deklarálni, az implementáció-elrejtés megvalósítása. Az Adában, a Modula-2-ben és egyéb, a modul

[16] Hacsak nem ágyazzuk őket egymásba. Annak viszont az a következménye, hogy a beágyazott alprogramhoz a befoglalón kívülről nem lehet hozzáférni, nem lehet a befoglalón kívülről meghívni.

fogalmát támogató nyelvekben a modul gyakran nem csak az egységbe zárásnak (encapsulation), hanem az megvalósítás elrejtésének eszköze is (lásd a 9. fejezetet). A modul specifikációs részében a modul használati utasítását adjuk meg (például a kívülről elérhető (exportált) alprogramokat, típusokat, változókat stb. deklarálhatjuk itt), míg a modul törzse tartalmazhatja az exportált egyedek implementációját. Tehát egy modul specifikációs részébe csak az alprogramok specifikációját írjuk, ami persze elégséges annak eldöntéséhez, hogy a modult használó programegységekben a megfelelő módon hívjuk-e meg ezeket az alprogramokat. Az alprogram teljes definícióját a modul törzsében helyezhetjük el. Nézzünk egy példát Modula-2 nyelven:

```
DEFINITION MODULE Stack;
EXPORT Push, Pop, IsEmpty;
TYPE ELEM = INTEGER;
PROCEDURE Push (i: ELEM);
PROCEDURE Pop() : ELEM;
PROCEDURE IsEmpty() : BOOLEAN;
END Stack.
```

A *Stack* modult használó programegységek számára deklaráltuk a *Push*, *Pop* és *IsEmpty* alprogramokat a modul specifikációs részében, amit a Modula-2-ben definíciós modulnak neveznek. Az alprogramok megvalósítása viszont az úgynevezett implementációs modulban kap helyet.

Az objektumorientált nyelvekben a modul fogalma általában egybeesik az osztállyal. Az Objective-C-ben külön adjuk meg az osztály specifikációját (**interface**) és implementációját (**implementation**): a specifikációban deklarálunk alprogramokat, típusműveleteket, de ezek definícióját az implementáció tartalmazza. Vannak olyan objektumorientált nyelvek is, például a Java, amelyek nem választják szét az osztályok specifikációját és implementációját (például két külön fordítási egységbe). Ráadásul az osztály műveletei nem sorozatot, hanem halmazt alkotnak, tehát közvetett rekurziónál sincs szükség definíciótól szétválasztott deklarációra. (A funkcionális nyelvek is hasonlóak ebből a szempontból: az egy fordítási egységen belül definiált függvények sorrendje tetszőleges, nem kell, hogy a definíció megelőzze a használatot.) A C++-ban két lehetőség is van a metódusok megadására. Az egyik a Java-szerű megadási forma, amikor az osztály kódján belül adjuk meg az alprogramok törzsét, a másik módszer szerint viszont a metódust csak specifikáljuk az osztály kódján belül (a C/C++ szóhasználatával élve csak a prototípusát írjuk le), és a törzset később adjuk meg.

```
class Stack {
  void push (int element); // csak specifikáció
  int pop () { . . . }     // teljes definíció
};
void Stack::push (int element) { . . . } // definíció
```

A külön fordíthatóság kapcsán a 7.5.1. szakaszban még egyszer vissza fogunk térni az alprogram-deklarációkra.

7.3.7. Makrók és inline alprogramok

A makró az alprogram ősének tekinthető. Az assembly nyelvekben találkozhattunk vele először, de visszaköszön a RATFOR-ból (mely a FORTRAN-hoz készült előfordító), a LISP-ből, a BLISS-ből, vagy – talán ez a legismertebb – a C előfordítójából is. A C makró például ugyanúgy használható a programszöveg tagolására, mint az alprogram. Hatékony, de nem túl biztonságos nyelvi eszköz. Paraméterezhető, de az aktuális paraméterek típusát nem veti össze a formális paraméterek típusával. A makrók hatékonyabbak az alprogramoknál, abból a szempontból, hogy az alprogramhívásból és a visszatérésből származó többletmunkát megspórolják. A makró törzse behelyettesítődik a programszövegbe minden egyes hívás helyére, így nem kell elvégezni a hívással kapcsolatos adminisztrációs munkákat (regiszterek, visszatérési cím mentése, stb.). A program futási idejének szempontjából tehát gazdaságosabb a makrók használata, viszont az elkészült program mérete makrók esetén nagyobb lesz, mint alprogramok esetén.

A fő problémát a makrókkal az jelenti, hogy kifejtésükre még a tényleges fordítás és programhelyesség-vizsgálat előtt sor kerül, és a fordítóprogram nem ellenőrzi helyességüket. A makró törzsében vétett hibák a makró első használatakor derülnek ki, ráadásul viszonylag nehéz rájönni, hogy a makróhelyettesítés utáni kódban kapott fordítási hibának okát hol kell keresni az eredeti forrásszövegben. Ezen kívül a makró helyes használatát, (megfelelő típusú aktuális paraméterek) csak a makró szövegének elolvasásával ismerheti meg a programozó. A makró törzsének megváltoztatása után az összes makróhívást újra kell gondolni, és ha szükséges, szintén meg kell változtatni. Mindezek az implementáció elrejtésének elvét sértik. Íme egy C makró:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

Vegyük észre, hogy mennyi, elsőre feleslegesnek tűnő zárójelet tartalmaz a makró törzse. Ezekre azért van mégis szükség, mert itt, a makró törzsében kell felkészülni arra a környezetre, amelybe a makró a kifejtés után kerül. Például a makró paramétereit azért zárójelezzük be minden használatkor, mert ha az aktuális paraméter egy kifejezés, akkor azt előbb kell kiértékelni, mint a makró törzsét[17].

A C makrói kapcsán még egy negatívumot tudunk felhozni: egy makróban nem használható olyan nyelvi szerkezet, ami a makróhívás helyszínén nem megengedett. Például ha makrónk egy értéket ad vissza, azaz egy kifejezésben hívjuk meg, akkor a törzse változódeklarációt vagy összetett utasítást (elágazást, ciklust) nem tartalmazhat. Ez megint sérti az implementáció-elrejtés elvét, hiszen nem csak a makró specifikációja, hanem megvalósítása is függ attól, hogy milyen szövegkörnyezetben fogjuk a makrót használni, és viszont: a lehetséges hívások nem csak a makró specifikációjától, hanem annak törzsétől is függnek.

A makrók és az alprogramok előnyeit igyekszik egyesíteni az a jónéhány nyelv (Ada, C++, Euclid, LISP stb.), amelyik támogatja az inline alprogramok használatát. Ezekben a nyelvekben a programozó általában tippeket adhat a fordítóprogramnak, hogy mely alprogramokat érdemes „behelyettesítésre" fordítani. Az Adában például az *Inline* fordítóvezérlő

[17] A makrók hatékonyságát nagymértékben tudja csökkenteni az, ha az aktuális paramétere egy bonyolultabb, időigényes számítást definiáló kifejezés. Ebben az esetben ugyanis a kifejezés mindannyiszor kiértékelődik, ahányszor a makró törzsében szerepel.

direktívát használhatjuk erre a célra[18]. Nézzünk egy példát arra, hogy egy C++ programozó hogyan tanácsolhatja a fordítónak, hogy alprogramját behelyettesítéssel fordítsa:

```
inline int max (int a, int b) { return a > b ? a : b; }
```

Azon kívül, hogy az inline alprogramok a makrók hatékonysága mellett az alprogramok specifikációja által nyújtott biztonsággal is rendelkeznek, szemantikusan is tisztább kód írását teszik lehetővé. Az inline alprogramok ugyanis csak egy implementációs technikát jelentenek, nincsenek hatással a program jelentésére. Ezt az állítást akkor értjük meg igazán, ha egy olyan példát veszünk, amelyben egy aktuális paraméternek mellékhatása is van. A *MAX(++x, y)* makróhívás során az *x* változó értéke kettővel fog megnőni, ami értelmetlennek tűnik, egész addig, amíg meg nem vizsgáljuk a makró törzsét. Hasonló probléma nem lép fel akkor, amikor az inline alprogramot hívjuk meg: *max(++x, y)*. Néhány, makrókkal kapcsolatos további problémára visszatérünk még a 242. oldalon.

7.3.8. Alprogram típusok

Egyes programozási nyelvekben (például az ALGOL 68-ban, a PL/I-ben, a Modula-2-ben vagy a Modula-3-ban) alprogram típusú változók nem csak alprogram formális paramétereként szerepelhetnek. A Modula-2-ben például, szemben mondjuk a Standard Pascallal, definiálhatunk olyan típust, amelynek típusértékei alprogramok:

```
TYPE Rendezes = PROCEDURE (VAR ARRAY OF INTEGER);
PROCEDURE QuickSort (VAR V: ARRAY OF INTEGER); ...
PROCEDURE HeapSort (VAR V: ARRAY OF INTEGER); ...
VAR Rendez: Rendezes;
VAR V: ARRAY (1..10) OF INTEGER;
...
Rendez := QuickSort;
...
Rendez(V);
```

A C-ben, a C++-ban, az Ada 95-ben stb. csak alprogramra mutató típust definiálhatunk. Az Ada 95-ben például az előző kód így nézne ki:

```
type Vektor is array (Integer range <>) of Integer;
type Rendezes is access procedure(V: in out Vektor);
procedure QuickSort (V: in out Vektor) is ...
procedure HeapSort (V: in out Vektor) is ...
Rendez: Rendezes;
V: Vektor(1..10);
...
Rendez := QuickSort'Access;
...
Rendez(V);
```

[18] Az Adában más fordítóvezérlő direktívával (pragmával) is befolyásolhatjuk a fordítóprogram által végzett optimalizációt. Például az *Optimize* pragmával megadható, hogy egy alprogramot futási időre, vagy felhasznált tárterületre szeretnénk optimalizáltatni.

Néhány nyelvben még alprogram típusú literált is lehet írni. A funkcionális nyelvekben például (lásd a 16. fejezetet) a λ-kifejezés segítségével adható értékül egy függvénydefiníció egy változónak. Az ALGOL 68-ban hasonlót így írhatunk:

```
proc(int)int f;
f := (int i)int:i+1;
print(f(10));
```

Itt *f* egy függvényváltozó, melynek a második sorban értékül adunk egy függvényliterált, majd a harmadik sorban az *f*-en keresztül meghívjuk azt.

7.4. Paraméterátadás

Az alprogram-paramétereket az információáramlás iránya alapján három csoportba oszthatjuk. Az első csoportba tartoznak a *bemenő paraméterek*, azaz azok a paraméterek, amelyeken keresztül a hívóból áramlik információ a hívott alprogramba. A második csoportba tartoznak a *kimenő paraméterek*, amelyeken keresztül a hívott alprogram juttat vissza információt a hívóba. Végül vannak a *be- és kimenő paraméterek*, amelyeken keresztül kétirányú információáramlás valósul meg.

A programozási nyelvekben más és más módon valósítják meg a bemenő, a kimenő, valamint a be- és kimenő paramétereket. Sőt, a különböző nyelvekben különbözőképpen valósítják meg még az egy csoportba tartozó paramétereket is. Ezt úgy szoktuk mondani, hogy a különböző nyelvekben különböző módokon adhatjuk át a formális paramétereknek az aktuálisokat, azaz különböző *paraméterátadási módok* vannak. Ebben a szakaszban áttekintjük a nevezetes paraméterátadási módokat.

7.4.1. Paraméterátadási módok

A paraméterek információáramlás szerinti osztályozása szemantika alapú. Ebben a szakaszban most a megvalósítás alapú osztályozását adjuk a paramétereknek. A különböző paraméterátadási módok kidolgozásakor nem csak szemantikai kérdések merültek fel, hanem hatékonysági kérdések is. Az ilyen jellegű összevetésekre a 7.4.2. szakaszban kerítünk sort.

Érték szerinti paraméterátadás

Ez a paraméterátadási mód bemenő paraméterek megvalósítását teszi lehetővé. Azt jelenti, hogy az aktuális paraméter híváskori értékét használhatjuk az alprogramon belül. A formális paraméter az alprogram egy lokális változójának tekintendő, amely az alprogram hívásakor megkapja kezdőértéknek az aktuális paraméter értékét. A formális paraméter változása nincs semmilyen hatással az aktuális paraméterre. Sőt, az aktuális paraméternek nem is kell változónak lennie, egy összetett kifejezés is megfeleltethető a formális paraméternek. Az alprogram hívásakor kiértékelődik az aktuális paraméterként szereplő kifejezés, és az így kapott érték bemásolódik a formális paraméterbe.

A Pascalban ez az alapértelmezett paraméterátadási mód, a C-ben pedig lényegében az egyetlen. A tömbök jelentenek kivételt ez utóbbi kijelentés alól. A tömbök és a mutatók fogalmát a C egy picit összemossa. Így, amikor egy tömböt adunk át paraméterként, akkor valójában a címét adjuk át. Ha az alprogramban megváltoztatunk egy tömbelemet, akkor az

aktuális paraméter által hivatkozott tömbben hajtjuk végre a változtatást. Tehát a tömbök valójában cím szerint – lásd a következő szakaszt – adódnak át a C-ben.

Nézzünk egy példát érték szerinti paraméterátadásra. Vegyük az alábbi C alprogramot, mely kiszámítja két pozitív egész szám legnagyobb közös osztóját:

```
int lnko (int a, int b)
{
    while (a != b)
      if (a > b)
          a -= b;
      else
          b -= a;
    return a;
}
```

Hogyan hívjuk meg ezt a függvényt?

```
int x = 10, y = 5;
y = lnko(x, x + y + 1);
```

A második sor végrehajtása után az *x* változó értéke marad a régi, azaz *10*, míg az *y* változó megkapja *10* és *16* legnagyobb közös osztójának az értékét, így új értéke *2* lesz.

Fontos, hogy az aktuális paraméter csak egyszer, a hívás pillanatában értékelődik ki, és íródik bele a formális paraméterbe. Ennek az a következménye, hogy nem csak az aktuális paraméter lesz közömbös a formális paraméter változásaival szemben (mint az előző példánkban *x* az *a* változásaival szemben), hanem fordítva, az aktuális paraméter változásai sem lesznek hatással a formális paraméterre. Ha például az aktuális paraméter egy, az alprogram számára nem-lokális változó, akkor annak a nem-lokális hivatkozáson keresztül történő megváltoztatása nem lesz hatással a formális paraméter értékére. Vizsgáljuk meg ezt is egy C példán keresztül:

```
int x = 1;
int f (int a)
{
    ++x;
    return a + x;
}
int main ()
{
    int y = f(x);
}
```

Az *f* meghívásának pillanatában az *a* formális paraméter felveszi az *x* globális változó értékét, azaz az egyet. Ezután az *x*-et megnöveli eggyel – ez nem változtatja meg *a*-t –, és visszaadja az *1+2*, azaz *3* értéket.

Cím szerinti paraméterátadás

Ezt a paraméterátadási módot leggyakrabban be- és kimenő paraméterek megvalósítására használják. Azt jelenti, hogy a formális paraméter megkapja az aktuális paraméter címét vagy egy hivatkozást (referenciát) rá. A formális paraméter tehát egy újabb név az aktuális paraméterhez, az alprogram végrehajtása során minden pillanatban mindketten ugyanazt az egyedet (ugyanazt a memóriaterületet) jelentik. Az aktuális paraméter változása a formális paraméter változását, a formális paraméter változása az aktuális paraméter változását vonja maga után. Ily módon valósul meg a kimenő információáramlás: amit az alprogram kiszámít a formális paraméterben, az az érték az alprogram befejeződése után elérhető az aktuális paraméteren keresztül. Természetesen az aktuális paraméternek balértéknek kell lennie, azaz olyan kifejezésnek, aminek értéket lehet adni, ami szerepelhet értékadás bal oldalán. Az aktuális paraméter például lehet egy változó, egy tömbelem, egy rekordmező stb. Az aktuális paraméter kiértékelése a balérték memóriabeli címének meghatározását jelenti: ez a cím adódik át a formális paraméternek. Az aktuális paraméter kiértékelésére egyszer, az alprogram végrehajtásának megkezdése előtt kerül sor.

A kimenő, valamint a be- és kimenő paramétereket támogató paraméterátadási módok használata esetén a fordítóprogramok ellenőrzik, hogy az aktuális paraméter balérték-e. Ellenkező esetben ugyanis az alprogramban kiszámolt eredményt „sehova sem lehetne tenni", semminek sem lehetne értékül adni. A FORTRAN kivételt jelent ez alól a szabály alól, mint ahogy ezt majd a 247. oldalon látni fogjuk.

A Pascalban egy formális paraméter deklarációjában megadható, hogy a neki megfeleltetett aktuális paraméter cím szerint adódjon át. Ehhez a *var* kulcsszót kell használnunk.

```
procedure swap (var a, b: Integer);
var temp: Integer;
begin
  temp := a;
  a := b;
  b := temp;
end;
```

Az, hogy az aktuális és a formális paraméter ugyanaz az egyed, furcsa anomáliákat okozhat. Ennek oka az „aliasok" kialakulása. Ez alatt azt értjük, hogy a program egy pontján két különböző változó ugyanazt az egyedet jelenti. Képzeljük el, hogy a *p* eljárásnak két cím szerinti formális paramétere van, *a* és *b*. Tegyük fel, hogy az eljárás feladata az, hogy kiírja és lenullázza két paramétere értékét. Ez nem túl életszerű példa, de az anomália illusztrálásához megteszi. Tegyük fel, hogy van egy *s* segédeljárásunk, ami egy paraméterrel már képes a kiírást és a lenullázást elvégezni. Így a *p* alábbi megvalósítása egész ésszerűnek tűnik:

```
procedure p (var a, b: integer);
begin
  s(a); s(b);
end;
```

Ha ezt az eljárást úgy hívjuk meg, hogy mindkét formálisnak ugyanazt az aktuálist feleltetjük meg, azaz *p(x,x)* formában hívjuk *p*-t, akkor a *p*-n belül az *a* és a *b* ugyanazt az egyedet fogják jelenteni. Amit az egyikkel csinálunk, az egyből a másikkal is megtörténik. Mire oda jutunk, hogy végrehajtsuk az *s(b)* eljáráshívást, a *b*-ben már nulla van. Tehát a *p(x,x)* nem írja ki kétszer az *x* híváskori értékét, mint ahogy azt elsőre talán gondolnánk: a második kiírás nulla lesz.

Aliasok kevésbé triviális módokon is kialakulhatnak. Például, ha az előző *p* eljárást két tömbelemmel, *t[i]*-vel és *t[j]*-vel hívjuk meg, akkor első ránézésre nincs alias – de mégis van, ha az *i* épp egyenlő a *j*-vel. Úgy is kialakulhat alias, hogy az alprogram egy összetett és egy skalár típusú változót (például egy tömböt és egy tömbelemet) vár paraméterként, és második aktuálisként megfeleltetjük az első aktuális egy komponensét (például *q(t,t[i])*). Végül aliast okoz az is, ha egy alprogramra nézve globális változót adunk át paraméterként, mint a következő példában.

```
var globalis: Integer;
procedure r (var lokalis: integer);
begin
  globalis := globalis + 1;
  lokalis := lokalis + globalis;
end r;
...
globalis := 1;
r(globalis);
```

Az *r* meghívása után a *globalis* változó értéke négy lesz.

Az aliasok kialakulása nagyon káros. Könnyen vezethet programozási hibákhoz, negatívan befolyásolja nem csak a program megbízhatóságát, hanem érthetőségét is, és nagyon nehézzé teszi a program helyességének formális bizonyítását. A szintén cím szerinti paraméterátadást használó Euclid nyelvnek például az egyik tervezési alapelve volt az aliasok kialakulásának következetes megakadályozása.

Eredmény szerinti paraméterátadás

Ez a paraméterátadási mód a kimenő paraméterek megvalósításához jó. Az alprogram formális paraméterében kiszámított eredményt helyezi vissza az aktuális paraméterbe. A formális paraméter, csakúgy, mint az érték szerinti paraméterátadás esetén, az alprogram lokális változója, melynek értéke az alprogram befejeződésekor belemásolódik az aktuális paraméterbe. Az aktuális paraméter ezek szerint balérték kell, hogy legyen. A formális paraméter nem kapja meg az aktuális paraméter értékét az alprogram hívásakor, ezért az információáramlás egyirányú.

Ez a paraméterátadási mód elég ritka – sok forrás nem is különbözteti meg a következő pontban tárgyalt érték/eredmény szerinti paraméterátadástól. Mindenesetre az Adában bizonyos paraméterek eredmény szerint adódnak át. Például a karaktereket beolvasó művelet a szabványos *Ada.Text_IO* könyvtárból így néz ki:

```
procedure Get (Item: out Character);
```

Az eredmény szerinti paraméterátadás problémáit együtt tárgyaljuk majd az érték/eredmény szerinti paraméterátadás problémáival.

Érték/eredmény szerinti paraméterátadás

Ezt a paraméterátadási módot a be- és kimenő paraméterek megvalósítására szokás használni. Lényegében az érték és az eredmény szerinti paraméterátadási módok együttes alkalmazásának tekinthető. A formális paraméter az alprogram egy lokális változója, mely az alprogram hívásakor megkapja az aktuális paraméter értékét, majd az alprogram befejeződésekor a formális paraméter értéke visszaíródik az aktuális paraméterbe. Mint az előző két, kimenő szemantikát (is) megvalósító paraméterátadási mód esetén, itt is csak balérték lehet az aktuális paraméter, hogy fel tudja fogni az alprogramban kiszámolt eredményt.

Az első nyelv, amely ezt a paraméterátadási formát támogatta, az ALGOL W volt. Mi most megint az Ada nyelvet használjuk egy példa bemutatására.

```
procedure Swap (A, B: in out Integer) is
  Temp: Integer := A;
begin
  A := B;
  B := Temp;
end Swap;
```

Ránézésre nincs különbség a *Swap* eljárás Pascal-beli (cím szerinti paraméterátadást használó) és az Ada-beli (érték/eredmény szerinti paraméterátadást használó) megvalósításai között. Mindkettő ugyanolyan jól működik. A lényeges eltérés a két paraméterátadási mód között akkor mutatkozik meg, amikor az aliasok kialakulását vizsgáljuk. Például érték/eredmény szerinti paraméterátadás esetén a *p(x,x)* eljáráshívás végrehajtása közben az *a* és *b* formális paraméterek egymástól független egyedeket jelentenek. Az *s(a)* hívás nincs hatással a *b*-re, így az *x* híváskori értéke kétszer íródik ki.

Másképp működne a globális változót használó példaprogramunk is, ha nem cím szerinti, hanem érték/eredmény szerinti paraméterátadást használnánk. Az Adában ennek formája a következő:

```
procedure R (Lokalis: in out Integer) is
```

Ebben az esetben a kódrészlet végrehajtása után a *Globalis* változó értéke három lenne.

Sajnos, az eredmény és az érték/eredmény szerinti paraméterátadásokkal is vannak gondok. Az egyik ilyen gond az, hogy az aktuális paramétert, ami, mint tudjuk, nem feltétlenül változó, hanem egy tetszőleges balérték is lehet, kétféleképpen is ki lehet értékelni. Az első lehetőség az, hogy a hívás pillanatában határozzuk meg azt a memóriacímet, amelyre az alprogramnak vissza kell másolnia a formális paraméter értékét, míg a másik lehetőség az, hogy a hívásból való visszatéréskor határozzuk meg azt. Például, ha az előbbi *r* eljárást *t(globalis)* aktuális paraméterrel hívjuk meg, ahol *t* egy tömb, akkor a két lehetőség két különböző eredményt fog jelenteni. Ha az alprogram végrehajtása előtt értékeljük ki a *t(globalis)* balértéket, úgy ahogy az Ada csinálja, akkor az alprogram *t(1)*-et fogja megváltoztatni *t(1)+2*-re. Ha viszont az alprogram befejeződésekor értékeljük ki a *t(globalis)*

balértéket, mint ahogy az ALGOL W csinálja, akkor az alprogram *t(2)*-t fogja megváltoztatni *t(1)+2*-re. (Vegyük észre, hogy a második esetben más egyed szolgáltatta a formális paraméter kezdőértékét, mint amibe visszamásoltuk a formális paraméter végső értékét.) Ugyanez a két lehetőség jelentkezne a sima eredmény szerinti paraméterátadás esetén is, ha az alprogramon belül a *lokalis := lokalis + globalis* értékadást mondjuk a *lokalis := globalis* értékadásra cserélnénk. A cím szerinti paraméterátadás ugyanilyen problémát nem vet fel: ott a *lokalis* az alprogram végrehajtása alatt egyfolytában egy alias *t(1)*-re, tehát az *r* a *t(1)*-et változtatja meg.

Egyes források az érték/eredmény szerinti paraméterátadásnak az Adában használt megfelelőjét logikailag különbözőnek tekintik az ALGOL W-beli változattól, és *másolat szerinti paraméterátadásnak* nevezik. Ebben a fejezetben mi, a források többségéhez hasonlóan, logikailag azonosnak vettük a másolat szerinti és az érték/eredmény szerinti paraméterátadást, és a különbségüket csak megvalósításbeli anomáliának tekintettük.

Van azonban még egy gond az eredmény és az érték/eredmény szerinti paraméterátadással. Változtassuk meg a *p* törzsét így:

```
procedure p (a, b: out integer) is
begin
  a := 1;
  b := 2;
end if;
```

Most vizsgáljuk meg, hogy mi a hatása egy *p(x,x)* hívásnak. Az biztos, hogy ebben a példában nem sok értelme van, hogy az *x*-et egyre is és kettőre is szeretnénk beállítani, de egy bonyolultabb példában esetleg elképzelhető lehet egy hasonló hívás. Cím szerinti paraméterátadás esetén a jelentés teljesen világos: a *p* végrehajtása alatt *x*, *a* és *b* ugyanazt az egyedet jelölik, és *x* értékét az eljárás befejeződése után az erre az egyedre vonatkozó utolsó értékadás határozza meg. Tehát *x* kettő lesz. Eredmény szerinti paraméterátadásnál azonban felmerül a kérdés, hogy először *a*-t, vagy először *b*-t kell-e bemásolni *x*-be. Egy adott programozási nyelv definíciójának mindenféleképpen állást kell foglalnia ebben a kérdésben, mert ha ez a kérdés megvalósításfüggő marad, akkor az adott nyelven írt programok hordozhatósága sérülni fog.

Szövegszerű helyettesítés

A szövegszerű helyettesítés a makrók „paraméterátadási módja". Ez fizikailag nem is nevezhető paraméterátadásnak, hiszen a paramétert nem kell átadni semminek: a makró törzse behelyettesítődik a hívás helyére, úgy, hogy a makró formális paraméterei szövegszerűen lecserélődnek az aktuális paraméterekre.

Ez a paraméterátadási technika ismét érdekes anomáliákat okoz. Már beszéltünk arról (lásd a 235. oldalt), hogy a makró törzsét, valamint a makró törzsében a formális paramétereket zárójelezni szoktuk, hogy ezzel a megfelelő kifejezés-kiértékelési sorrendet biztosítsuk. Arra is láttunk példát, hogy egy makró aktuális paraméterei egész váratlanul többször is kiértékelődhetnek (ha a makró megvalósításában többször is előfordul a megfelelő formális

paraméter), ami nem csak hatékonyságbeli problémákat vet fel, de a program jelentésére is hatással lehet, ha az aktuális paraméterek mellékhatásos kifejezések (lásd a 236. oldalt).

Megjegyezzük továbbá, hogy aliasok ugyanúgy kialakulhatnak, mint a cím szerinti paraméterátadás esetén. Az ott látott példák mind átvihetők a szövegszerű helyettesítésre is. Csak elrettentésképpen képzeljük el például az alábbi makrót:

```
#define M(a) (((a) <= ++n) ? (a + n) : n)
```

Ez a makró a hívás helyszínén található *n* változót használja. Ha a makrót *M(n)* alakban hívjuk meg, az *n* értéke eggyel nő, és a kifejezés értéke nem *n+(n+1)* lesz, hanem *(n+1)+(n+1)*.

Az aliasok kialakulásánál sokkal nagyobb problémát jelenthet az aktuális paraméterek és a makróban definiált lokális változók összekavarodása.

```
#define KIIR(x,n) { int i; for (i = 0; i < (n); ++i) printf("%d ",(x)); }
```

Ez egy utasításként használható makró. Mi történik, ha a segítségével egy programegység az *i* változójának az értékét szeretné tízszer kiíratni? A *KIIR(i,10)* hívás hatására nullától kilencig íródnának ki a számok, nem pedig az *i* íródna ki tízszer. Ez azért van, mert a makró törzsében az aktuális paraméter, az *i* váltaná fel az *x*-et, de az *i* névhez a deklarációk elfedési szabályainak megfelelően a makró lokális *i* változójának definíciója kapcsolódna.

Végül hadd említsük meg, hogy bizonyos alprogramok helyett egyáltalán nem készíthető makró. Például képzeljük el a két paraméterét megcserélő *swap* eljárást makróként.

```
#define SWAP(a,b) { int c = a; a = b; b = c; }
```

Tegyük fel, hogy *t* egy tömb és *i* ennek a tömbnek egy megengedett indexe. Ha a fenti makrót *SWAP(i,t[i])* alakban hívjuk meg, a két paraméterét nem fogja megcserélni. Mind cím szerinti, mint érték/eredmény szerinti paraméterátadásnál jól működik a szokásos *swap* ezekre az aktuális paraméterekre. Szövegszerű helyettesítés esetén azonban, ha mondjuk *t* csupa egyeseket tartalmaz, *i* pedig nulla, a következő történik. A *c=i* értékadás hatására *c* felveszi a nulla értéket. Ezután az *a=b* értékadásból lett *i=t[i]* hajtódik végre, melynek eredményeképpen *i* felveszi *t[0]* értékét, ami egy. Végül a *b=c* értékadásból lett *t[i]=c* hajtódik végre, melynek hatására *t[1]* felveszi a nulla értéket. A feladat az lett volna, hogy az *i* és a *t[i]* (azaz a *t[0]*) értékét cseréljük fel, *(0,1)*-ről *(1,0)*-ra. Helyette az történt, hogy *i*-be bekerült ugyan az egy, de a nulla érték nem *t[0]*-ba, hanem *t[1]*-be íródott.

Név szerinti paraméterátadás

Ez a paraméterátadási mód ismét a be- és kimenő paraméterek átadására használható. Talán ez a legkülönösebb a paraméterátadási módok közül. Működése sokkal bonyolultabb, mint az eddigiek voltak.

Ez a paraméterátadás az ALGOL 60-nal terjedt el, a programozási nyelvek kialakulásának kezdetekor. Akkoriban nagyon sokan programoztak assembly nyelveken, amelyek a makrók használatát támogatták. Így ésszerűnek tűnt egy olyan paraméterátadási mód bevezetése, amely a makróhelyettesítést utánozta. A név szerinti paraméterátadás sokban hason-

lít a szövegszerű helyettesítésre, de azért lényeges kérdésekben különbségek is mutatkoznak közöttük.

A név szerinti paraméterátadás egyik furcsasága az, hogy nem egyféleképpen működik, hanem többféle, homlokegyenest különböző technikát (például érték és cím szerinti átadást) ötvöz. Ráadásul a különböző megvalósítási módok közül aszerint választ, hogy az aktuális paraméter milyen alakú. Tehát nem csak az alprogram szövegétől és környezetétől függ, hogy mit fog csinálni az alprogram, hogy egyáltalán hogyan veszi át a paramétereit, hanem függeni fog attól is, hogy milyen aktuális paraméterrel hívták meg. Az ilyen paraméterátadást használó alprogram működésének megértése még az aliasokat tartalmazó/bevezető alprogramokénál is sokkal nehezebb.

Ha egy név szerinti formális paraméternek egy literált vagy egy konstans kifejezést (például *4+2*) feleltetünk meg, akkor az aktuális érték szerint adódik át. Ha viszont egy skalár változót feleltetünk meg neki, akkor az aktuális cím szerint adódik át. Most jön a következő furcsaság: ha valamilyen kifejezés az aktuális paraméter, a kifejezés újra és újra kiértékelődik, ahányszor az alprogram szövegében a formális paraméterre hivatkozás történik. Ha tehát az aktuális paraméter egy olyan kifejezés, amelynek egyik-másik komponense az alprogram végrehajtása során megváltozik, a formális paraméter a változásoknak megfelelően más és más értékkel bír majd.

Ha például az aktuális paraméter egy *t[i]* kifejezés, és az *i* értéke az alprogram végrehajtása során egy ponton megváltozik, akkor a formális paraméter onnantól kezdve egy másik tömbelemet fog már jelenteni, az *i* új értékének megfelelőt. Vagy, ha az aktuális paraméter egy *a+b* alakú kifejezés, és a *b*-t az alprogram megváltoztatja, akkor ezzel együtt a formális paraméter értéke is megváltozik, idomulva *b* új értékéhez.

Nézzünk egy híres példát [Sco 00]. Az alábbi, „Jensen's device" néven elhíresült ALGOL 60 alprogram egy szumma kifejezést tud kiszámítani.

```
real procedure sum (expr, i, low, high);
  value low, high;
  real expr;
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    rtn := rtn + expr;
  sum := rtn;
end sum
```

A függvény első két paramétere (a valós típusú *expr* és az egész típusú *i*) név szerint adódik át (ez az alapértelmezett paraméterátadási mód az ALGOL 60-ban), míg a másik két (egész típusú) paraméter érték szerint. Most számítsuk ki az alábbi kifejezés értékét:

$$y = \sum_{x=1}^{10} 3x^2 - 5x + 2$$

Ehhez csak meg kell hívni az előző függvényt azokkal a paraméterekkel, amelyeket a képletben látunk:

*y := sum (3*x*x − 5*x + 2, x, 1, 10);*

Az *x* egy skalár változó, tehát lényegében cím szerint adódik át az *i* formálisnak. Az alprogramban *i* és *x* ugyanazt az egyedet jelölik. Amikor a ciklust futtatjuk egy és tíz között (ezek az értékek adódnak át *low*-nak és *high*-nak), az *x* változó értéke végigfut ezen az intervallumon. Az összegzendő kifejezés (az aktuális paraméter *expr*-hez) a ciklusmagban újra és újra kiértékelődik, mindig a megfelelő *x* érték alapján.

Vegyük észre a hasonlóságot a név szerinti paraméterátadás és a szövegszerű behelyettesítés között. Bármilyen alakú is az aktuális paraméter, a formális mindig úgy viselkedik, mintha lecseréltük volna a programszövegben az aktuálisra. Ha az aktuális egy konstans kifejezés, akkor a formális paraméter értéke mindig ugyanaz marad: a konstans kifejezés értéke. Ha az aktuális egy skalár változó, a formális mindig ugyanazt az egyedet fogja jelölni; ez olyan, mintha az aktuális paramétert használnám az alprogram szövegében. Végül, ha az aktuális paraméter változókat tartalmazó kifejezés, akkor minden hivatkozás a formális paraméterre a kifejezés újbóli kiértékelését eredményezi, mintha az aktuális paraméter szerepelne a formális helyett. A 7.5.3. szakaszban megmutatjuk majd, hogy a név szerinti paraméterátadás nem egyszerűen egy kiokosított változata a szövegszerű helyettesítésnek, hanem attól fontos elvekben különbözik.

A név szerinti paraméterátadás esetén a szövegszerű helyettesítésnél tapasztalt problémák jelentős része nem lép fel. Bizonyos problémákkal (például az aliasok megjelenésével) azonban még mindig számolni kell. Az, hogy néhány nagyon egyszerű számítás név szerinti paraméterátadás segítségével nem fogalmazható meg, továbbra is gondot okoz. A már sokszor emlegetett *swap* eljárás név szerinti paraméterekkel sem valósítható meg.

7.4.2. A paraméterátadási módok összehasonlítása

Az érték szerinti paraméterátadás a fent bemutatottak közül szemantikusan a legtisztább. Ha az ember csak mellékhatásmentes függvények és érték szerinti paraméterátadás használatával programozik, az elkészült programtermék világos, jól olvasható és karbantartható, továbbá matematikai eszközökkel (például helyességbizonyítás, programtranszformációk) könnyen kezelhető lesz. Nem véletlen, hogy a funkcionális programozási nyelveket épp ezek miatt szokták dicsérni.

Sok programozó szerint az imperatív nyelvek eljárás fogalma segítségével sok programot frappánsabban lehet megfogalmazni, mint pusztán függvények segítségével. Az eljárások kimenő paramétereken juttatják el az általuk kiszámított eredményeket a hívónak. Ezt érték szerinti paraméterátadással nem tudjuk modellezni: az csak bemenő paraméterekre használható. (Érték szerinti paraméterekre például a szokásos *swap* eljárás természetesen nem működik.) Kimenő, illetve be- és kimenő paraméterek átadására a cím szerinti, az eredmény szerinti, az érték/eredmény szerinti és a név szerinti módok használhatók[19].

[19] A szövegszerű helyettesítést itt most nem is említjük: ez a technika annyi szemantikus problémával jár, hogy alprogramok paraméterátadásának megvalósításához nem is használják, csak makrók esetén jut szerephez.

A név szerinti paraméterátadás nagyon rugalmas technika. Elve a késleltetett összekapcsolás. A formális paraméterhez nem az alprogram meghívásakor kapcsolódik hozzá az aktuális paraméter, hanem később, az alprogram végrehajtása közben. Sőt, a törzsben újra és újra elvégződik ez az összekapcsolás, amikor csak hivatkozunk a formális paraméterre. Ez nagymértékű rugalmasságot, dinamikus, futási idejű alkalmazkodni tudást eredményez. A késleltetett összekapcsolás elve a programozási nyelvek más területein is hasonló hatással bír. Gondoljunk az objektumorientált nyelvekre, ahol a polimorfizmus megvalósítása lényegében azon alapszik, hogy a műveletek (metódusok, üzenetek) késleltetve, futási időben kapcsolódnak az objektumokhoz (12. fejezet). A lusta kiértékelésű funkcionális nyelvek (16. fejezet) vagy az imperatív nyelvekben használatos „rövidre zárt" logikai operátorok szintén ezt az elvet követik.

Nagyfokú rugalmassága ellenére a név szerinti paraméterátadás nem igazán tudott elterjedni. Ennek elsődleges oka bonyolultságából ered. Ezen technika használata mellett a programok gyakran nehezen érthetővé válnak. Szemantikus problémái (gondoljunk például a *swap* megvalósíthatatlanságára) szintén ellene szólnak. Ezen paraméterátadás megvalósítása sem könnyű: az ALGOL 60-ban például a név szerinti aktuális paraméterekből egy-egy paraméter nélküli eljárást (thunk) fordítanak, melynek dolga az aktuális paraméter kiértékelése. Ez az eljárás adódik át valójában az alprogramnak, mely a formális paraméterre történő hivatkozáskor újra és újra meghívja azt. (A paraméterként átadott alprogramok megvalósítási nehézségeiről a 7.7.1. szakaszban részletesebben beszélünk majd.) Már ebből a pár mondatból is kiolvasható, hogy a név szerinti paraméterátadás a hatékonyságával sem tündöklöl. Bizony, az összes paraméterátadási mód közül ez a legkevésbé hatékony. Végül megjegyezzük, hogy viszonylag kevés olyan gyakorlati probléma van, ahol a név szerinti paraméterátadás segítségével olyan briliáns megoldást lehetne adni, mint amit az összegző függvényünk kapcsán láttunk; ha mégis találkozunk ilyennel, azt általában hasonlóképpen elegánsan meg lehet írni úgy is, hogy egy alprogramot adunk át paraméterként.

A kimenő paraméterek modellezéséhez ezek szerint a cím szerinti és az eredmény, illetve az érték/eredmény szerinti paraméterátadási módokat szokták a programozási nyelvek tervezői választani. A két utóbbi, elsősorban az aliasok elkerülése miatt, szemantikusan tisztább, könnyebben kezelhető, mint a cím szerinti paraméterátadás. De, mint már említettük, nem ez az egyetlen szempont, amit figyelembe kell venni egy nyelv tervezésekor. A hatékonyság is fontos kérdés.

Az érték szerinti, az eredmény szerinti, illetve az érték/eredmény szerinti átadásokat összefoglaló néven szokás *adatmozgató* (data transfer) paraméterátadásnak is nevezni. Ez a név arra utal, hogy az aktuális paraméterről egy lokális másolat keletkezik a hívott alprogramban, melybe, és/vagy melyből adatokat kell másolnunk: az aktuális paraméter értékét, illetve a kiszámított eredményt. Ez az adatmozgatás igen költséges is lehet, mind tárban, mind végrehajtási időben. Nagyméretű paraméterek esetén, például tömbök, mátrixok esetén a cím szerinti paraméterátadás általában sokkal hatékonyabb: az aktuális paraméter címét, ami mondjuk egy gépi szó, kell csak a formális paraméterben eltárolnunk az alprogram meghívásakor.

A helyzetet tovább bonyolítja, hogy a hatékonyságba nem csak az számít bele, hogy az alprogram hívásakor, illetve a végrehajtás befejezésekor mennyi adminisztrációra, adatmozgatásra van szükség. Az is fontos szempont, hogy az alprogramon belül milyen hatékonyan

tudunk hozzáférni a használt adatokhoz. Ebben viszont az adatmozgató paraméterátadások jobbak: meg kell csak keresni azt a memóriaterületet, ahol a másolat, a formális paraméter található. A cím szerinti paraméterátadásnál eggyel több indirekció van: a formális paraméter megkeresése után annak tartalmát címként használva kell az aktuális paraméterrel dolgozni. Ha az alprogramon belül sok hivatkozás történik a formális paraméterre, és az aktuális paraméter nem túlságosan nagy, akkor az adatmozgató paraméterátadások sokkal hatékonyabbak, mint a cím szerinti.

Nagyméretű paraméterek esetén a cím szerinti paraméterátadás annyival hatékonyabb, mint az adatmozgatók, hogy még az érték szerinti paraméterátadásnak, azaz a tisztán bemenő paramétert modellező módnak is a vetélytársa lehet. A Pascal programozókat gyakran arra tanítják, hogy a **var** szócskát akkor kell használni a formális paraméterek előtt, ha meg szeretnénk változtatni az aktuálisat, *vagy* ha a paraméter nagyon nagy, például egy tömb [Sco 00]. Persze ez a fajta szemlélet könnyen hibás programok írását eredményezi: a hatékonyságot a szemantika elé helyezni mindig veszélyes. Cím szerinti paraméterátadásnál, ha – bár nem szeretnénk, de – véletlenül mégis megváltoztatjuk a bemenő jelentésű aktuális paramétert, a fordító nem tud minket figyelmeztetni a hibára. Hasonlóképpen, ha „csak kimenő" paramétert valósítunk meg cím szerinti paraméterátadással, a fordító nem fog szólni, hogy vigyázz, felhasználtad az aktuális paraméter értékét a számításban!

A következő szakaszban néhány érdekesebb programozási nyelvet kiválasztva megvizsgáljuk, hogy az imént felvetett problémákra milyen megoldásokat szokás adni.

7.4.3. A lehetőségek néhány programozási nyelvben

A FORTRAN paraméterei mind be- és kimenő paraméterek. A nyelv nem tesz megkötést arra vonatkozóan, hogy milyen paraméterátadási módot kell használnia a fordítóprogramnak ennek megvalósításához, cím szerintit, vagy érték/eredmény szerintit. A FORTRAN 77 előtti megvalósítások majd mindegyike kizárólag cím szerinti paraméterátadást használ, az újabb megvalósítások azonban gyakran érték/eredményt szerintit használnak skalár paraméterekhez. Ha egy FORTRAN programozó bemenő paramétert akar használni, akkor az alprogramban neki magának kell vigyáznia arra, hogy a formális paramétert (és rajta keresztül az aktuálist) meg ne változtassa. Például felvehet egy lokális változót az alprogramban, melybe bemásolja a formális paraméter értékét, és az alprogramon belül ezt a lokális változót már gond nélkül módosíthatja. Más nyelvektől eltérően a FORTRAN-fordítók nem tekintik hibának, ha egy be- és kimenő aktuális paraméter egy jobbérték (például egy számliterál, vagy egy nem balérték összetett kifejezés). Ha ugyanis hibának tekintenék, akkor nehézkes lenne bemenő paramétereket használni a nyelvben. Igen ám, de egy jobbérték esetén mi fog mondjuk cím szerint átadódni? Ha összetett kifejezésről van szó, a fordító egy ideiglenes (temporális) változót fog létrehozni, amelyben eltárolja a kifejezés értékét. Ez az ideiglenes változó fog cím szerint átadódni. Ha az alprogram a tervezett bemenő szemantika ellenére megváltoztatja a formális paraméter értékét, az ideiglenes változó fog vele együtt megváltozni. A programozó persze nem fér hozzá ehhez az ideiglenes változóhoz, számára az alprogram által a formális paraméterbe kiszámított eredmény elérhetetlen lesz. Mi történik a literálokkal, ha cím, vagy érték/eredmény szerint átadjuk őket? A fordító a literálokat ugyanúgy a programban tárolja, mint a változókat. Cím szerinti paraméterát-

adásnál például az a memóriacím fog átadódni, ahol a literál tárolódik. Ha, ismét megszegve a feltételezett bemenő szemantikát, az alprogram megváltoztatja a formális paraméterét, a fordítási idejű konstans megváltozik, ami végzetes hatással lehet a program további működésére. Sok FORTRAN IV megvalósítás nem volt felkészülve erre a problémára. Az ezekkel lefordított programokban megtörténhetett, hogy egy `SWAP(0,1)` hívás hatására a `0` és az `1` konstansok felcserélődtek. Ha ezután a program egy későbbi pontján egy `X = 1` utasítás szerepelt, az `X` változó nem az egyet, hanem a nullát kapta meg értékül.

A következő nagy hatású nyelv, az ALGOL 60, a név szerinti paraméterátadást használja alapértelmezés szerint, és az érték szerintit akkor, ha a formális paramétert a *value* kulcsszó segítségével vezetjük be. Nem sok olyan nyelv van, amely a név szerinti paraméterátadást átvette az ALGOL 60-ból. Az egyik, amely megtette, egy meglehetősen széles körben elterjedt nyelv, a SIMULA 67. Ebben a nyelvben azonban már a név szerinti a választható, és az érték szerinti az alapértelmezett paraméterátadási mód.

A név szerinti paraméterátadás problémái miatt az ALGOL W nyelvben érték/eredmény szerint adjuk át a be- és kimenő paramétereket. Az Ada nyelv bizonyos esetekben szintén ezt a paraméterátadást használja. Erre a későbbiekben még visszatérünk.

Az ALGOL 68-ban és a Pascalban a bemenő paraméterekhez az érték szerinti, be- és kimenő paraméterekhez pedig a cím szerinti paraméterátadást használjuk. (Az érték szerinti az alapértelmezett, a cím szerintit külön jelölni kell.) A COBOL beágyazott programjainál is az érték és a cím szerinti paraméterátadások közül választhat a programozó.

A C-ben alapvetően csak egyféle paraméterátadás van: érték szerinti. A be- és kimenő paraméterek megvalósításához általában a programozónak explicit módon kell címet, azaz mutatót – érték szerint – átadnia:

```
void swap (int *a, int *b)
{
   int c = *a;
   *a = *b;
   *b = c;
}
```

Ezt az alprogramot a következőképpen használhatjuk két egész típusú változó felcserélésére:

```
int a = 2, b = 4;
swap (&a,&b);
```

A szabály alól a kivételt a tömbök jelentik, melyek a C-ben egybemosódnak a mutatókkal. A tömbök cím szerint adódnak át:

```
void sort (int t[ ]) { . . . }
. . .
int t[ ] = { 3, 4, 1, 2 };
sort (t);
```

A C++ egyik újdonsága a C-hez képest, hogy bevezetett egy speciális mutató típust, az úgynevezett *referencia típust*. Ezen típus megkönnyíti a be- és kimenő paraméterek haszná-

latát is. A cím szerinti paraméterátadás a referencia típusokon keresztül történik. A mutatók kifejezett jelölésére nincs szükség sem az alprogramokon belül, sem a híváskor:

```
void swap (int &a, int &b)
{
   int c = a;
   a = b;
   b = c;
}
...
int a = 2, b = 4;
swap (a, b);
```

Nézzünk most egy másik példát [Sco 00] alapján. A referencia típusok visszatérési érték megadásához is használhatók. Ez különösen hasznos, ha egy függvénynek olyan értéket kellene visszaadni, amelyre az értékadásnak nincs értelme. Ilyen lehet például egy fájl. Persze mutatót bármire, akár egy fájlra is vissza lehet adni, de a mutató használatakor a hivatkozást explicit fel kell oldani, és ez bizony kényelmetlen lehet. A C++-ban például a << operátor használható arra, hogy egy kimenetre (fájlba, szabványos kimenetre stb.) írjunk. Ez az operátor visszaad egy hivatkozást az első argumentumára. Ily módon a << hívásokat „sorozni" lehet egy kimenetre.

```
cout << a << b << c;
```

Ha nem lennének referencia típusok, a << operátort mutatók segítségével kellene leprogramozni. Ebben az esetben viszont a hívás így nézne ki:

```
*(*(cout << a) << b) << c;
```

A C++ és az ANSI C tartalmaz egy nagyon érdekes nyelvi elemet: formális paraméterek definíciójában szerepelhet a **const** kulcsszó. Az alábbi példában ezzel tudjuk előírni, hogy a formális paraméter egy konstansra hivatkozó mutató legyen[20]. Ez azt fogja jelenteni, hogy a formális paraméter által hivatkozott egyedet az alprogram nem tudja megváltoztatni. Az aktuális paraméternek nem kell konstansra hivatkozó mutatónak lennie: a fordító automatikusan át tudja alakítani a formális paraméternek megfelelő típusra. Legyen T egy típus, és tekintsük a következő függvénydefiníciót és függvényhívást.

```
int f (const T *p) { ... }
...
T x = ...;
int i = f(&x);
```

Noha az x változót a cím szerinti paraméterátadást utánozva adtuk át az f függvénynek, az az x értékét nem tudja megváltoztatni. Szemantikusan az x egy bemenő paramétere lett az f-nek. Ha a T típusú értékek nagyok, ezzel a trükkel megspórolható az adatmozgatás, ami

[20] Ha azt akarjuk kifejezni egy deklarációban, hogy az r változó konstans mutató, akkor azt a következőképpen írnánk: T * **const** r.

egy érték szerinti paraméterátadással járna: helyette az általában hatékonyabb cím szerinti átadás költségei terhelnék csak az alprogramhívást. Még jobban látszik ennek a technikának az előnye, ha a C++ referencia típusával kombináljuk. Ebben az esetben ugyanis tisztán cím szerinti paraméterátadással valósítunk meg egy bemenő paramétert.

```
int f (const T &p) { . . . }
. . .
T x = . . .;
int i = f (x);
```

Azért azt ne gondoljuk, hogy ez a technika „olcsón" biztosítani tudja azt a szemantikus biztonságot, amit az érték szerinti paraméterátadás nyújt. A cím szerinti átadás miatt aliasok továbbra is kialakulhatnak. Noha az alprogram a formális paraméterén, *p*-n keresztül nem tudja megváltoztatni az aktuális paramétert, az alprogram törzsén belül nem tudhatjuk, hogy egy másik hivatkozáson keresztül nem tettük-e meg ugyanezt. Az érték szerinti paraméterátadásnál ellenben garantált, hogy a formális paraméter értéke az alprogram végrehajtása alatt nem változik, ha az alprogram kifejezetten meg nem változtatja. Nézzünk erre egy példát: itt a hívás eredményeképpen nem a *42* fog kiíródni kétszer, hanem először a *42*, azután pedig a *43*.

```
void g (const int &p, int &q) {
  cout << p;
  ++q;
  cout << p;
}
. . .
int x = 42;
g (x, x);
```

Vegyük észre azt is, hogy az érték szerinti paraméterátadásnál a formális paramétert teljes jogú lokális változóként használhatjuk, akár értéket is adhatunk neki (anélkül persze, hogy ennek az aktuális paraméterre bármilyen hatása lenne). Ugyanez nem tehető meg a **const** segítségével megvalósított, cím szerint átadott bemenő paraméterek esetén.

Más nyelvekben is van a **const**-hoz hasonló eszköz. A Modula-3-ban például egy formális paramétert deklarálhatunk **READONLY** módúként. A fordítóprogram meg fogja akadályozni, hogy az alprogramon belül a formális paraméter egy értékadás bal oldalán szerepeljen, vagy hogy egy másik alprogramnak be- és kimenő szemantikával adódjon át aktuálisként. A **READONLY** segítségével deklarált bemenő paraméter általában érték szerint adódik át, ha kis méretű, és cím szerint, ha nagy méretű. Ha egy jobbérték-kifejezést adunk át aktuálisként egy cím szerinti **READONLY** paraméternek, akkor a fordító – a FORTRAN-fordítókhoz hasonlóan – egy ideiglenes változót hoz létre, amelyben a jobbérték-kifejezés értéke tárolásra kerül, és amelynek a címe átadódik a formális paraméternek.

Az Adában (és más, hasonló nyelvekben) szemantikusan világosabb megközelítést alkalmaznak. A programozó nem paraméterátadási módokat, hanem szemantikus módokat ír elő a formális paraméterekhez. Az **in** módúként specifikált paraméterek bemenő, az out

módú paraméterek kimenő, és az **in out** módú paraméterek be- és kimenő paraméterek lesznek. Azzal, hogy a paraméterek ténylegesen milyen paraméterátadási technikával kerülnek átadásra, a programozó nem fog foglalkozni: az ilyen implementációs részletkérdéseket a fordítóra bízza. Az Ada 95 szabályai értelmében egy **in** módú paramétert az alprogram nem változtathat meg, sem értékadással, sem úgy, hogy **out** vagy **in out** módú paraméternek felelteti meg aktuálisként egy alprogramhívásban. Hasonlóképpen, egy **out** módú paraméter értékét az alprogram nem olvashatja ki (az Ada 83-ban egyáltalán nem, az Ada 95-ben pedig addig, amíg értéket nem adott neki). Az **in out** módú paraméterekre természetesen minden megengedett. A skalár és mutató típusú értékekre egyébként az Ada a megfelelő adatmozgató paraméterátadást használja, míg összetett típusú értékeknél a nyelv definíciója nem mindig határozza meg, hogy adatmozgató vagy cím szerinti paraméterátadási módot kell használni; a fordítóprogramok ebben eltérhetnek egymástól. (Például jelölt típusoknál kötelező cím szerintit használni, de rekordoknál nem.) Ez azért nem okoz elméletileg hordozhatósági problémát, mert az Ada helytelennek (erroneous) tekint minden olyan programot, amelynek eredménye függ attól, hogy a fordító milyen paraméterátadási módot használ. Ezt a fajta „helytelenséget" a fordítóprogramoknak nem kell tudniuk észrevenni és mondjuk fordítási hibával honorálni.

Érdekes eltérések mutatkoznak a különböző nyelvekben az altípussal deklarált formális paraméterek esetén [Sco 00]. A Pascalban például, ha egy intervallum típusba tartozó paramétert cím szerint akarunk átadni, az aktuális paraméter típusa nem lehet szűkebb, mint a formális paraméter típusa. Ellenkező esetben ugyanis elképzelhető lenne, hogy az alprogramban olyan műveletet hajtunk végre a formális paraméterrel, aminek az eredményét nem lehetne az aktuális paraméterben eltárolni.

```
type egytolszazig = 1..100;
var a: 1..10;
    b: 1..1000;
procedure p (var n: egytolszazig);
begin
  n := 100;
end;
...
a := 5; b := 5;
p(b);  (* Ez rendben van. *)
p(a);  (* Ez fordítási hibát okoz. *)
```

Az Adában egy ilyen paraméter adatmozgató paraméterátadással adódna át. Mindkét hívás helyes lenne, hiszen az alprogramon belül nem okozna gondot (a lokális változóként kezelt) formális paraméterre vonatkozó értékadás. Ellenben, amikor az alprogram véget ér, a kimenő paraméterértéknek bele kell esnie az aktuális paraméter altípusába. Ha ez nem teljesül, futási idejű hibát (dinamikus szemantikai hibát) kapunk, mely egy előre definiált kivétel kiváltódásában nyilvánul meg.

A Javában érték szerinti paraméterátadás van. Mivel az objektumok implicit hivatkozásokon keresztül érhetők el, és egy objektum paraméterként történő átadása a hivatkozás érték szerinti átadását jelenti, ezért egy alprogram megváltoztathatja az aktuális paraméter-

ként kapott objektumot, pontosabban azt az objektumot, amire az aktuális paraméterként kapott hivatkozás hivatkozik. (Szokás ezt *megosztás szerinti paraméterátadásnak* (call-by-sharing) is nevezni. Ez az elnevezés egyébként a CLU-ból származik.) Természetesen az aktuális paraméter értékét, azaz magát a hivatkozást nem tudja megváltoztatni az alprogram. Az alábbi példában az *f (a,sb)* hívásnak az lenne a hatása, hogy az *sb* által hivatkozott objektumhoz hozzáfűződne a `"szia"` karakterlánc.

```
int a = 5;
StringBuffer sb = new StringBuffer();
void f (int x, StringBuffer s) {
  ++x;
  s.append("szia!");
  s = new StringBuffer();
  s.append("viszlát!");
}
```

A C# már sokkal bonyolultabb paraméterátadási modellt használ. Alapértelmezés szerint ugyanúgy működik, mint a Java: az úgynevezett hivatkozás típusok – osztályok, interfészek, delegate típusok és tömb típusok – átadásakor az implicit hivatkozások adódnak át érték szerint, míg az úgynevezett érték típusok – primitív típusok, felsorolási típusok és **struct** (struktúra) típusok – átadásakor maguk az értékek. Az alapértelmezett mechanizmustól eltérni a cím szerinti és a kimenő szemantikájú paraméterekkel lehet. A cím szerinti paraméterátadást a **ref** kulcsszó használatával kell jelezni, mind a formális, mind az aktuális paraméter előtt. (Ez formáját tekintve arra hasonlít, mint amikor a C-ben utánoztuk a cím szerinti paraméterátadást.) A **ref** használata esetén mind a hivatkozás, mind az érték típusokba tartozó aktuális paramétereket meg tudja változtatni az alprogram. (Hivatkozás típusú paraméter esetén tehát nem csak a hivatkozott objektumot, de magát a hivatkozást is!) Végül, az **out** kulcsszó használatával (melyet szintén el kell helyezni mind a formális, mind az aktuális paraméter előtt) olyan cím szerinti paraméterátadást érhetünk el, amelynek során a fordító a kimenő szemantikát fogja kikényszeríteni. (Nevezetesen az aktuális paraméternek nem kell inicializáltnak lennie, mert a formális sem inicializálódik automatikusan. A formális értékét addig nem lehet kiolvasni, amíg értéket nem adunk neki. Az alprogram nem térhet vissza anélkül, hogy a kimenő paraméter értéket ne kapott volna. Az alprogram befejeződése után az aktuális paraméter inicializáltnak tekintendő.) Vigyázzunk, hogy egy hivatkozás típusú paraméter érték szerinti átadása nem ugyanolyan, mint egy érték szerinti paraméter cím szerinti átadása. Legyen például *C* egy osztály, melynek egy adattagja van, az egész típusú *i*. Legyen továbbá *S* egy ugyanilyen szerkezetű **struct** típus.

```
void g (C c, ref S s) {
  c = new C(); c.i = 1;
  b = new S(); b.i = 1;
}
C a = new C(); a.i = 0;
S b = new S(); b.i = 0;
```

A *g(a,* **ref** *b)* hívás hatására az *a* hivatkozással nem történik semmi, a *b* változó értéke ellenben megváltozik, az *i* mezőjének az értéke egy lesz.

Az Eiffelben az alprogram a formális paraméterét nem változtathatja meg, ugyanúgy, mint ahogy az Adában az **in** módú paraméterek nem változhattak meg. Az Eiffelben – mint a Javában vagy a C#-ban – kétféle típust különböztetünk meg: hivatkozás típusokat, és úgynevezett kifejtett típusokat. Minden típusnak van kifejtett és hivatkozásos változata is. Ha a formális paraméter valamilyen hivatkozás típusú, akkor a formális paraméterhez új értéket (új hivatkozást) nem rendelhetünk, de az általa hivatkozott objektumot (annak mezőit) megváltoztathatjuk az alprogramon belül. Ha mind az aktuális, mind a formális paraméter valamilyen hivatkozás típusba tartozik, cím szerinti paraméterátadás valósul meg: a formális paraméter ugyanarra az objektumra hivatkozik, mint az aktuális. Ha mind az aktuális, mind a formális paraméter kifejtett típusú, akkor érték szerinti paraméterátadás valósul meg: az aktuális paraméterről másolat készül a formális paraméterbe. Ha az aktuális paraméter hivatkozás típusú, a formális pedig kifejtett, akkor az aktuális paraméterről (illetve az általa hivatkozott objektumról) másolat készül, és ez kerül a formális paraméterbe. Ezt gyakran hivatkozásfeloldásnak (dereferencing) nevezik. Ebben az esetben, ha az aktuális paraméter nem hivatkozik objektumra, azaz az értéke **void**, futási idejű kivétel lép fel. Az utolsó eset az, amikor az aktuális paraméter kifejtett típusú, a formális paraméter viszont hivatkozás típusú. Ebben az esetben az aktuális paraméterről másolat készül, és ez a másolat íródik bele a formális paraméter által hivatkozott objektumba.

A CLU-ban ismét nagyon hasonló a helyzet. A típusokat a mutable és az immutable kategóriákba soroljuk. A mutable típusú értékek objektumhivatkozásokat, az immutable típusú értékek tényleges objektumokat tartalmaznak. A primitív típusok immutable típusok, de a típuskonstrukcióknak megvannak a mutable és az immutable változatai is. A mutable objektumok megosztás szerint, az immutable objektumok pedig érték szerint adódnak át paraméterként.

Végezetül a lusta kiértékelésű funkcionális nyelvek paraméterátadási módjáról kell még szót ejtenünk. Ezekben a nyelvekben a paraméterek átadása hatását tekintve úgy történik, mint az érték szerinti paraméterátadásnál. A különbség az, hogy az aktuális paraméter kiértékelése és a formális paraméter értékének meghatározása nem történik meg a függvény meghívásakor, hanem csak később, akkor, amikor először szükség van a paraméter értékére. Ezt a fajta paraméterátadást *igény szerintinek* szokták nevezni (call-by-need).

Az igény szerinti paraméterátadás hasonlít egy picit a név szerintire, a késleltetett összekapcsolás elve miatt. A név szerinti paraméterátadással szemben azonban csak egyszer fog kiértékelődni az aktuális paraméter.

7.5. Az alprogramok környezete

Ebben a szakaszban azt vizsgáljuk meg, hogy az alprogram hogyan illeszkedik be környezetébe, a teljes programba. Megemlítjük a legfontosabb idevágó hatóköri és láthatósági szabályokat, és áttekintjük az egymásba ágyazás és a külön fordíthatóság kérdéskörét. Részletesebb információkért lásd a 4. és a 9. fejezeteket.

7.5.1. Külön fordíthatóság

Csak azok a nyelvek alkalmasak arra, hogy nagyméretű, hosszú élettartamú szoftverrendszereket készítsenek bennük, amelyekben a program különböző részeit le tudjuk fordítani anélkül, hogy az egész programot le kellene fordítani [Seb 02]. Ez a lehetőség ugyanis igen fontos a csapatmunkában történő fejlesztés támogatása, a kódírás hatékonysága, valamint a szoftver újrafelhasználhatósága és kevésbé költséges módosíthatósága szempontjából. A programrészek külön fordíthatóságának segítségével ugyanis lehetőség nyílik arra, hogy egy rendszernek csak azokat a részeit kelljen újrafordítani, amelyek megváltoztak a szoftverfejlesztés vagy -karbantartás legutóbbi lépésében.

Az alprogramok természetes tagolási szintet kínálnak azoknak a nyelveknek, amelyek egyáltalán lehetővé teszik *fordítási egységek* készítését, azaz programrészek különálló fordítását. Az ASA FORTRAN-ban például a fordítási egység és az alprogram lényegében egybeeső fogalmak[21]. A C-ben egy alprogram lehet önálló fordítási egység, de egy fordítási egység állhat több alprogramból (valamint változó- és típusdeklarációból) is. Az Adában szintén: egy alprogram szerepelhet fordítási egységként, és vannak más programegységek is, amelyekből fordítási egységeket képezhetünk. Akadnak ellenben olyan nyelvek is, amelyekben az alprogram nem lehet önálló fordítási egység: a Modula-2-ben, a Cleanben, a Javában stb. csak magasabb tagolási szintet, modult, osztályt használhatunk fordítási egységek kialakítására.

Néhány nyelvben, például a FORTRAN II-ben vagy a Pascal korai változataiban a *fordítási egységek* csak egész programok lehetnek. Különálló programrészeket (alprogramokat vagy más programegységeket) nem lehet önmagukban lefordítani. Ezek a nyelvek lényegében használhatatlanok komoly alkalmazások fejlesztésére. A FORTRAN és a Pascal későbbi változataiban meg is próbálták megszüntetni ezt a súlyos hiányosságot.

Az ASA FORTRAN-ban (és például a C-ben) már van lehetőség *független fordításra* (independent compilation). Ez a kifejezés azt jelenti, hogy az alkalmazást több fordítási egységből lehet összeállítani, de ezek a fordítási egységek egymástól teljesen függetlenek, nem rendelkeznek információkkal egymásról. Ennek a technikának az a hátránya, hogy a fordítási egységek egymásra hivatkozásakor nem történik típusellenőrzés. Az ASA FORTRAN-ban például az alprogramok (eljárások és függvények) külön fordítási egységek (a FORTRAN szóhasználatában szegmensek). Nincs olyan nyelvi eszköz, amellyel két szegmens közötti kapcsolatot lehetne megfogalmazni. Ezért, ha egy alprogram hívásakor az aktuális paraméterek száma vagy típusa nem egyezik meg a formális paraméterek számával és típusával, a fordító nem képes hibát jelezni.

A PL/I-ben az egy fordítási egységben található alprogramok hívásakor a fordító összeveti az aktuális és formális paramétereket, de a különböző fordítási egységekben levő (úgynevezett külső) alprogramok hívásakor nem képes erre. Azért, hogy a programok biztonságosságát növelje, a PL/I lehetővé teszi azt, hogy egy külső alprogramot deklaráljunk, azaz megadjuk a specifikációját a hívó fordítási egységben. A deklarációkor megadott specifikációt a fordító össze tudja vetni az összes hívással. Sajnos, az alprogram definíciójában és a más fordítási egységekben előforduló deklarációkban szereplő specifikációkat a fordítóprogram továbbra sem képes egyeztetni.

[21] Ha eltekintünk a főszegmenstől és az adatszegmenstől, melyek szintén fordítási egységek.

A C-ben azokat a más fordítási egységekben definiált függvényeket, melyek visszatérési értékének típusa nem **int**, kötelező deklarálni. A deklarációt ilyenkor a fordítóprogram – a PL/I-hez hasonlóan – nem képes összevetni az alprogram definíciójával. A nem deklarált külső alprogramokra a C az **int** visszatérési értéket feltételezi, és ezt követeli meg az alprogram használatakor is. Azzal, hogy az alprogram definíciójában milyen visszatérési típus szerepel, a C fordító nem foglalkozik. Tovább bonyolítja a helyzetet az, hogy az eredeti C-ben a paraméterek típusát nem tartalmazza az alprogram specifikációja. Az ANSI C-ben, ha nem használunk prototípust, azaz nem adjuk meg a paraméterek típusát, szintén nem kerül ellenőrzésre az aktuális paraméterek száma és típusa. Ha tehát olyan deklarációt írunk, amely nem tartalmaz paramétertípust, az alprogram használata (hívása vagy például címének paraméterként való átadása) nem lesz megfelelőképpen típusellenőrizve. A – majdnem tökéletes – megoldás a fejállományok (header file) használata. Az alprogram deklarációját egy fejállományban helyezzük el, amelyet mind a definiáló, mind a használó fordítási egységekbe betöltünk egy-egy **#include** utasítással. Ezzel a módszerrel nagyon sok hibalehetőséget kizárhatunk, csak arra kell odafigyelni, hogy ugyanazt a fejállományt töltsük be a definiáló, és az összes használó fordítási egységbe, valamint, hogy minden, más fordítási egységből használható függvényt deklaráljunk a fejállományban.

Az újabb nyelvekben, például a FORTRAN 90-ben, a Modulában stb. általában lehetőség van arra, hogy *külön fordítható* fordítási egységeket definiáljunk (separate compilation). Ezekben a nyelvekben – a független fordítással ellentétben – meg kell fogalmaznunk a fordítási egységek közötti kapcsolatokat is. Az Adában a fordítási egységek specifikálják a kívülről elérhető egyedeket (objektumokat, alprogramokat, formális paramétereket, típusokat stb.), azaz információkat közölnek róluk, melyek alapján ezeket az egyedeket más fordítási egységekből típushelyesen lehet majd használni. Például az Adában egy alprogram lehet egy önálló fordítási egység. Ha egy másik fordítási egységben meg akarjuk hívni ezt az alprogramot, akkor a hívó fordítási egység számára elérhetővé kell tennünk. Erre a **with** utasítás használható, melyet a hívó fordítási egység legelején kell elhelyeznünk.

```
with Swap;
procedure Sort (A, B: in out Integer) is
begin
  if A > B then Swap(A,B); end if;
end Sort;
```

A **with** utasítással jelezzük a fordítónak, hogy a *Sort* eljárásban szereplő *Swap* hivatkozást a külön fordítási egységként elkészített *Swap* specifikációjával kell összevetni.

7.5.2. Beágyazhatóság

Az ALGOL 60-ban ugyan nem lehetett egy programot több fordítási egységre bontani, viszont megjelent benne egy másik nagyon fontos fogalom: a blokkszerkezetesség (más szóval blokkstrukturáltság). Ez azt jelenti, hogy programegységeket egymásba lehet ágyazni. A különböző nyelvek különböző mértékben támogatják a blokkszerkezetességet. Az ASA FORTRAN-ban például semmiféle lehetőség nincs arra, hogy mondjuk két alprogramot egymásba ágyazzunk. A C-t sem nevezik blokkszerkezetes nyelvnek, bár az egymásba

ágyazható blokk utasításai már mutatnak közös vonásokat az egymásba ágyazott alprogramokkal. Az alábbi C példában az *f* alprogram törzsébe beágyazott blokkban egy olyan *j* változót is deklarálunk, amely elfedi a befoglalóban deklarált ugyanolyan nevű változót.

```
void f (int p)
{
   int i = 1, j = 1;
   {
      int j = 2, k = 2;
      printf ("Beágyazott blokk. %d %d %d", i, j, k); /* 1 2 2 */
   }
   printf ("Befoglaló blokk. %d %d", i, j);      /* 1 1 */
}
```

A fejlettebb nyelvek már nem csak az alprogramokat kínálják a programszöveg tagolására, hanem modulokat, osztályokat is. Az alprogramok fontos építőkövei ezeknek a magasabb szintű tagolást lehetővé tevő nyelvi szerkezeteknek. A Javában az osztályokat egymásba ágyazhatjuk, sőt, osztályokat beágyazhatunk alprogramokba is, de alprogramokat alprogramokba nem. A Cleanben épp ellenkezőleg: modulokat nem ágyazhatunk be sem modulokba, sem alprogramokba, azonban alprogramokat beágyazhatunk alprogramokba. Végül vannak olyan nyelvek is, amelyekben „mindent mindenbe" be lehet ágyazni. Ilyenek például a Modula-2 vagy az Ada.

```
procedure Sort (A, B: in out Integer) is
  procedure Swap (X, Y: in out Integer) is
    Z: Integer := X;
  begin
    X := Y;
    Y := Z;
  end Swap;
begin
  if A > B then Swap(A,B); end if;
end Sort;
```

A programegységek egymásba ágyazása az alapja a hatóköri és láthatósági szabályoknak. Például, ha egy alprogramot beágyazunk egy másik alprogramba, akkor a beágyazott alprogramot csak a befoglaló alprogramon belül használhatjuk. Ezzel a módszerrel a programegységek közötti kapcsolatok számát, és így a program bonyolultságát csökkenthetjük.

A beágyazott alprogramok nem csak a bennük definiált, úgynevezett *lokális* változókhoz (illetve egyéb egyedekhez, például alprogramokhoz, típusokhoz) férnek hozzá. Használhatják a befoglalóban definiált egyedeket, valamint a befoglalót befoglalóban definiált egyedeket is, és így tovább, rekurzívan. Végül használhatják azokat az egyedeket (modulokat, osztályokat, alprogramokat), amelyek *globálisak*, azaz nincsenek semmi másba beágyazva. Szokás azokat az egyedeket, amelyek egy alprogramból használhatók, de nem ott, „lokálisan" lettek definiálva, az alprogramra nézve *nem-lokális* egyedeknek nevezni. Az előző Ada

példában a *Swap* eljárás használhatná az *A* és *B* változókat, mert azok nem-lokális változói a *Swap* eljárásnak.

A közös nem-lokális változókon keresztül az alprogramok információt tudnak cserélni egymás között. Ez egy általában nagyon hatékony, de a kód bonyolultságát nagymértékben növelő lehetőség. A modularizáltságot, és így a program olvashatóságát, karbantarthatóságát elősegíti, ha az alprogramok egymással a paramétereiken keresztül kommunikálnak. A nem-lokális változókon keresztül történő információcserével viszont a program hatékonyságát növelhetjük. A paraméterátadás költséges dolog, főleg az adatmozgató paraméterátadási módok esetében. Az alprogramok egymásba ágyazásának és a nem-lokális változókon keresztül történő kommunikációnak ésszerű alkalmazásával programjaink hatékonyságát a bonyolultságuk növelése, illetve a tervezési és modularitási elvek feladása nélkül növelhetjük. Így írhatnánk át az előző Ada eljárásunkat úgy, hogy a *Sort* és a *Swap* közös változókon keresztül kommunikáljanak paraméterek helyett:

```
procedure Sort (A, B: in out Integer) is
  procedure Swap is
    Z: Integer := A;
  begin
    A := B;
    B := Z;
  end Swap;
begin
  if A > B then Swap; end if;
end Sort;
```

7.5.3. Statikus és dinamikus hatókör

A programozási nyelvek döntő többsége statikus (más néven lexikális) hatóköri szabályokat használ. Ez azt jelenti, hogy a hatókör a programegységek egymásba ágyazásán alapul. Egy deklaráció pontosan az őt közvetlenül befoglaló programegység végéig van érvényben: azon kívül nem, de azon belül még az őt követő beágyazott programegységekben is érvényben van. Ezért mondhatjuk azt, hogy például egy beágyazott alprogram hozzáférhet a befoglaló alprogram változóihoz.

Van azonban néhány olyan nyelv (például sok LISP-változat, a SNOBOL, az APL vagy a Perl), amely dinamikus hatóköri szabályokat használ[22]. Ez azt jelenti, hogy egy deklaráció hatóköre nem a csak tartalmazott, hanem a hívott programegységekre is ki fog terjedni. Ha egy *A* alprogramban deklarálunk egy változót, majd meghívunk egy *B* alprogramot, a *B* alprogram használhatja a változót, még akkor is, ha *B* nincs az *A* alprogramba beágyazva. A következő ALGOL-szerű formában megadott példaprogramban az egész típusú *I* változó

[22] A LISP újabb változatai legtöbbször már statikus hatókört használnak. Eredetileg a Perlben csak dinamikus hatókör volt, de a jelenlegi nyelv választást ad a programozónak, hogy dinamikus vagy statikus hatóköri szabályokat akar-e.

nő eggyel, ha statikus, és a valós típusú *I* változó, ha dinamikus hatóköri szabályok vannak érvényben [Koz 92a]:

```
begin
  integer I := 1;
  procedure P is
  begin
    I := I+1;
  end P;
  begin
    real I := 1.0;
    P;
  end;
end;
```

Nem véletlen, hogy a nyelvek többsége a statikus hatóköri szabályokat használja. Sokkal könnyebben megérthető egy program statikus, mint dinamikus hatóköri szabályok mellett. Továbbá a statikus típusellenőrzés is megkívánja a statikus hatóköri szabályokat. Vannak azonban olyan feladatok, amelyekhez a dinamikus hatókör segítségével nagyon frappáns, illetve hatékony megoldást készíthetünk.

Ami az alprogramokat illeti, a statikus és dinamikus hatókör fogalma világíthatja meg a különbséget a név szerinti paraméterátadás és a szövegszerű helyettesítés között. Egy statikus hatóköri szabályokat alkalmazó nyelvben ugyanis a név szerinti paraméterátadást használó alprogramok engedelmeskednek a statikus hatóköri szabályoknak, a szövegszerű helyettesítést használó makrók azonban nem. Az alprogramban ugyanis a formális paraméternek átadott aktuális a hívó környezetében fog újra és újra kiértékelődni akkor, amikor a formális paraméterre hivatkozás történik, míg a makrónál az aktuális paraméter a makrótörzs környezetben fog kiértékelődni. (Erre a furcsaságra láttuk a *KIIR* példát a 243. oldalon.) Továbbá, az alprogram törzse az alprogram definíciója helyén hajtódik végre, abban a környezetben, amit a programszöveg jelöl ki lexikálisan (statikusan) az alprogram számára. Ezzel szemben a makró törzse a hívás helyén érvényes környezetben, tehát dinamikus hatókörnek engedelmeskedve fog kiértékelődni.

Mielőtt a fenti gondolatot egy példán keresztül részletesebben bemutatnánk, ismertetjük az ALGOL 60 név szerinti paraméterátadásának egy alternatív definícióját:

- Az aktuális paramétereket szövegszerűen behelyettesítjük a formális paraméterek helyére, de úgy, hogy az aktuális paraméterekben előforduló változók és az alprogramban definiált lokális változók ne ütközzenek. Ha egy aktuális paraméterben van egy ugyanolyan nevű változó, mint amilyen lokális változó van az alprogramban, akkor az alprogram lokális változóját, valamint minden rá vonatkozó hivatkozást át kell nevezni.
- Az így kapott alprogramtörzset szövegszerűen be kell helyettesíteni a hívás helyszínén. Ha az alprogramban szerepelnek olyan nem-lokális változók, amelyekkel megegyező nevű lokális változók vannak a hívóban, akkor a hívóban szereplő lokális változókat (és minden rájuk vonatkozó hivatkozást) át kell nevezni.

Tekintsük az alábbi példát:

```
begin
  integer n;
  procedure P(x);
  begin
    integer i;
    i := n; print i; print x;
  end;
  n := 1;
  begin
    integer n; integer i;
    n := 2;
    i := 3;
    P(i + 1);
  end;
end;
```

Az első lépés során az *x* helyére beírjuk az *i+1* értéket, de észrevesszük, hogy az aktuálisban szereplő *i* és a *P*-ben deklarált *i* összeütközésbe kerültek egymással. A megoldás az, hogy átnevezzük a lokális változót, mondjuk *j*-re. Ezután a *P* törzse a következő lesz.

```
begin
  integer j;
  j := n; print j; print (i + 1);
end;
```

A második lépés során ezt a törzset behelyettesítjük a hívás helyszínére. Ismét névütközéssel állunk szemben, mert a *P* törzsében szerepel egy *n*, de az *n* változó a hívás helyén mást jelent. Tehát a hívás helyén átnevezést kell végrehajtani, az *n*-et *m*-re. Ezután ez lesz a programból:

```
begin
  integer n;
  n := 1;
  begin
    integer m; integer i;
    m := 2; i := 3;
    begin
      integer j;
      j := n; print j; print (i + 1);
    end;
  end;
end;
```

A program azt írja ki, hogy 1 4.

Mi történik akkor, ha egy ezzel „egyenértékű" C programot írunk, de a fenti program név szerinti paraméterátadású alprogramja helyett makrót használunk? A szokásoknak megfelelően a makró formális paraméterét gondosan zárójelezzük.

```
int n = 1;
#define P(x) { int i = n; printf("%d %d", i, (x)); }
int main ()
{
  int n = 2;
  int i = 3;
  P(i + 1);
  return 0;
}
```

Kifejtés után az alábbi kódot kapjuk:

```
int n = 1;
int main ()
{
  int n = 2;
  int i = 3;
  { int i = n; printf("%d %d", i, (i+1)); };
  return 0;
}
```

Ez a program azt írja ki, hogy `2 3`.

7.5.4. A változók élettartama

Az alprogramok lokális változói és formális paraméterei általában addig élnek, amíg az alprogram véget nem ér. A lokális változók és a formális paraméterek az alprogram minden egyes elindításakor létrejönnek, majd, amikor az alprogram befejeződik, ezek a változók felszámolódnak. Az alábbi C kódrészlet egy jellemző hibát szemléltet. A fordító szerint ez helyes függvény, de ha meghívjuk, és utána használni próbáljuk azt az egész számot, amire a visszatérési értékként kapott mutató mutat, azt tapasztaljuk, hogy a szám már nem is létezik: felszámolódott, amikor kiléptünk a függvényből.

```
int * ne_hivj_meg ()
{
  int i = 1;
  return &i;
}
```

Néhány nyelvben van kivétel a fenti szabály alól. Az ASA FORTRAN-ban például az alprogram lokális változói statikusak, azaz élettartamuk a program teljes működésére kiterjed. Más nyelvekben, például a PL/I-ben, a CLU-ban, a C-ben vagy a C++-ban egy erre a célra való kulcsszó segítségével lehet statikus élettartamú lokális változókat deklarálni.

```
int next ()
{
  static int i = 0;
  return ++i;
}
```

A *next* függvényben deklarált *i* változó statikus élettartamú. A függvényből való kilépéskor nem számolódik fel, sőt az értékét is megőrzi a következő függvényhívásig. Ezért lehetséges az, hogy a *next* függvény minden egyes meghívása egy újabb egész érték visszaadását fogja eredményezni. Az alábbi program szintén ugyanígy működik:

```
int j = 0;
int next ()
{
  return ++j;
}
```

A különbség a kettő között az, hogy az első megoldás modularizáltabb. Az ott definiált *i* változó hatóköre nem terjed ki az egész programra, mint a második példában deklarált *j* változóé, hanem csak a *next* függvényre. A függvényen kívül az *i* változó semmilyen módon nem érhető el.

A Javában – beágyazott osztályok használata esetén – nem csak egy lokális változó, hanem egy aktuális paraméter is „túlélheti" az alprogram végrehajtását.

```
Object nevetAd (final String keresztnév) {
  final String vezetéknév = "Kovács ";
  return new Object() {
          public String toString() { return vezetéknév + keresztnév; }
        };
}
```

Ha ezt a metódust meghívjuk egy aktuális paraméterrel, akkor visszakapunk egy olyan objektumot, amely emlékezni fog erre az aktuális paraméterre, még sokkal azután is, hogy a metódus végrehajtása befejeződött. Amikor kiírjuk *pityu*-t, meghívódik a *toString()* művelete, mely előkeresi az aktuális paraméterét és lokális változóját annak a hajdan volt metódushívásnak, amely *pityu*-t létrehozta, és ezekből kiszámítja a visszaadandó *String* objektumot.

```
Object pityu = nevetAd("István");
. . .
System.out.println(pityu);
```

7.6. Az alprogramnevek túlterhelése

Néhány nyelv megengedi, hogy egy program egy adott pontján ugyanahhoz az alprogramnévhez több alprogram is tartozzon. Ha ezzel a névvel meghívunk egy alprogramot, a

fordító a hívás körülményeiből deríti ki, melyik alprogramot szeretnénk végrehajtani. Ha a hívásból nem derül ki egyértelműen a fordító számára, hogy melyiket szeretnénk elindítani az azonos nevű (azaz „túlterhelt", angolul overloaded) alprogramok közül, fordítási hibát kapunk. Ha pedig egy olyan túlterhelt változat sincs, amely megfelelne a hívásnak, megint csak fordítási hiba lesz az eredmény.

Ezek szerint a túlterhelt alprogramok közötti választás fordítási időben történik egy statikus típusellenőrzést használó nyelvben. A fordítónak az alprogram hívásához szükséges információk alapján kell a döntését meghoznia. Melyek lehetnek ezek az információk? Ez már nyelvtől függő. Például a fordító az aktuális paraméterek típusa alapján választhatja egyik vagy másik alprogramot. Az biztos, hogy az információk, amely alapján a döntést meg kell hozni, az alprogram specifikációjában találhatók, hiszen a hívó programegység csak a hívott specifikációját „ismeri". Azt szokás mondani, hogy a különböző túlterhelt változatoknak a „szignatúrájuk" kell, hogy különböző legyen. A szignatúra azonban más és más szokott lenni a nyelvekben.

Az Adában minősített névvel is hívhatók alprogramok. A minősítés annak a csomagnak a nevével történik, amelyik tartalmazza az alprogram definícióját. Ha két különböző csomagban definiálunk két ugyanolyan alprogramot, akkor a minősített név segít annak az eldöntésében, hogy mikor melyiket akarjuk meghívni. Ha nem minősített névvel szeretnénk meghívni az alprogramot, vagy ha az alprogramok minősített neve is ugyanaz (azaz ugyanabban a csomagban vannak definiálva), akkor más különbözőséget kell felmutatni a hívás helyszínén. Ez lehet például az aktuális paraméterek száma, illetve az aktuális paraméterek típusa. Ezenkívül használhatjuk megkülönböztetésre azt, hogy van-e visszatérési érték (azaz függvény-e az alprogram), és ha igen, akkor mi a visszatérési érték típusa. A formális paraméterek neve és módja, noha részeit képezik az alprogram specifikációjának, nem szolgálhatnak a túlterhelt alprogramok megkülönböztetésére.

```
function A return Integer;
function A return Boolean;
procedure A;
procedure A (S: String);
procedure A (I: in out Integer);
procedure A (I: in Integer; J: in Boolean := True);
procedure A (V: Natural := 42); -- ezzel baj van!
```

Az első hat alprogram megfelelő túlterhelései az *A* névnek. A hetedik alprogramot azonban már nem vehetjük hozzá az első hathoz, mert nem különbözik eléggé az ötödiktől. Ha az első hat közül bármelyiket meghívjuk, a hívás körülményeiből majdnem mindig eldönthető, hogy melyik változatot kell végrehajtani. Elképzelhetők azonban olyan hívások is, amikor ez nem igaz. Ha *K* egész típusú változó, akkor az *A(K)* hívás nem egyértelmű: lehet az ötödik, vagy a hatodik alprogramra történő hivatkozás is. Sőt, az Ada fordító még az *A(2)* hívást sem tekinti egyértelműnek, noha ez a hívás egyébként nem lenne helyes az ötödik alprogramra nézve.

A Javában és a C++-ban a visszatérési értékkel rendelkező alprogramok meghívhatók úgy, hogy a visszatérési értéket nem használjuk semmire, ezért ezekben a nyelvekben a szi-

gnatúrába nem vették be a visszatérési érték típusát: ha két alprogram csak ebben tér el egymástól, akkor azok nem tekinthetők túlterhelt változatnak. Az objektumorientált nyelvekben definiált altípus reláció tovább bonyolítja a túlterhelés szabályait. Az altípusosságnak köszönhetően egy aktuális paraméter több típusba is tartozhat egyidejűleg. Tekintsük az alábbi Java kódvázlatot:

```
void m(Object o) { . . . }
void m(String s) { . . . }
```

Az `m("Szia")` hívás mindkét túlterhelt változatnak megfelel, a másodiknak azonban „jobban": tehát a fordító egyértelműnek tekinti a helyzetet, a második alprogramot fogja meghívni[23].

A Clean nyelvben egész másképp használható a túlterhelés: ott a túlterhelt változatok csak nagyon picit térhetnek el egymástól, és a túlterhelés tényét külön jelezni kell. Az alprogramok típussal való paraméterezésének eszközét jelenti a Clean túlterhelés fogalma, ily módon az Ada vagy a C++ sablon fogalmára hasonlít. (Lásd a 11. fejezetet.)

Miért adjuk ugyanazt a nevet két alprogramnak? Elsősorban akkor jó ez a lehetőség, ha az alprogramok tényleg majdnem ugyanazt csinálják. Például egy egészekből álló tömböt rendező alprogramnak is a *sort* nevet adhatjuk, meg egy valósakból álló tömböt rendezőnek is.

Ezen gondolatnak megfelelően sok nyelv előre definiált túlterhelt alprogramokat kínál a programozók számára. Igaz ez az Adára, a C++-ra, a Javára is, sőt, olyan nyelvekre is (ALGOL 60, FORTRAN stb.), amelyek nem engedik meg a programozónak a túlterhelt alprogramok készítését. Például az Ada szabványos könyvtáraiban találhatók olyan *Put* nevű eljárások, amelyekkel karakterláncokat, karaktereket, illetve egész számokat lehet kinyomtatni a szabványos kimenetre vagy egy fájlba.

A túlterhelés egy másik alkalmazási területe a formális paraméterek alapértelmezett értékének helyettesítése. A Javában például így készítenénk el a 228. oldalon látotthoz hasonlóan használható kódot:

```
void draw_rectangle (int width, int height) {
  draw_rectangle(width, height, 0, 0, 0);
}
void draw_rectangle (int width, int height, int x, int y, int color) {
  . . .
}
```

A túlterhelés fegyelmezett használata általában pozitív hatással van a program olvashatóságára, érthetőségére, mert az absztrakció és a polimorfizmus (lásd a 11. fejezetet) eszköze. Átgondolatlan, ügyetlen használata viszont a kód bonyolultságát könnyen megnövelheti. Ráadásul az altípusokkal, az alapértelmezett paraméterekkel és az automatikus típuskonverzióval való interferenciája sokszor gondot okoz.

[23] Ha mégis az első alprogramot szeretnénk meghívni a `"Szia"` paraméterrel, használhatjuk a következő hívást: `m((Object)"Szia")`. Ezzel a típuskényszerítéssel (lásd az 5.1.4. fejezetet) az aktuális paraméter statikus típusát *Object*-re változtattuk, így a második alprogramnak a hívás már nem fog megfelelni.

7.6.1. Operátorok túlterhelése

Az operátorok olyan alprogramok, amelyek speciális szintaxissal hívhatók. Például az összeadásra a legtöbb nyelvben a + operátor használható, melyet az alprogramoktól eltérően nem prefix, hanem infix alakban szokás meghívni. Sok nyelv az előre definiált típusaihoz előre definiált (gyakran túlterhelt, mint az egészekhez is és a valósakhoz is használható +) operátorokat biztosít, de nem engedi meg, hogy a programozó az általa készített típusokhoz hasonló szemantikájú operátorokat definiáljon. Ilyen nyelv a Java is, ami egyébként megengedi a sima alprogramok túlterhelését. Ezzel szemben például a C++-ban vagy az Adában, ha készítünk mondjuk egy mátrix típust, írhatunk hozzá infix alakban is hívható + műveletet:

function `"+"` *(A, B: Matrix)* **return** *Matrix* **is** ...

Ha X és Y mátrixok, akkor mind az $X+Y$, mind a `"+"`(X, Y) kifejezések érvényesek. Az így definiált operátor örökli a a nyelv által definiált operátor precedenciáját.

Az Adában fontos megkötés, hogy a nyelv által definiált operátorokat túl lehet terhelni, de újabb operátorokat nem lehet definiálni. Az ML-ben, a Cleanben stb. azonban ez is megtehető.

```
infixr 8 tothe;
fun x tothe 0 = 1.0
  | x tothe n = x * (x tothe (n-1));
```

Ez a `tothe` nevű ML függvény [Sco 00] egy infix alakban használható operátor lesz, mely jobbról asszociatív (erre utal az `infixr` szócskában az r betű), és a precedenciaszintje 8 lesz. A FORTRAN 90 is megengedi új infix operátorok definiálását, de megköveteli azt, hogy az operátort pontok közé zárjuk (például `A .cross. B`), és minden ilyen operátorhoz ugyanazt a precedenciaszintet rendeli.

Végül említsük meg, hogy egyes nyelvek (például a LISP vagy a Smalltalk) teljesen egységes jelölésrendszert biztosítanak a vezérlési szerkezetekhez, operátorokhoz és alprogramokhoz.

```
if a > b then max := a else max := b;          (* Pascal *)
(if (> a b) (setf max a) (setf max b))        ; LISP
(a > b) ifTrue: [max <- a] ifFalse: [max <- b].   "Smalltalk"
```

7.7. Az alprogramok megvalósítása

Ahhoz, hogy jobban megértsük, melyik nyelvben miért azok a lehetőségek vannak az alprogramok kapcsán, amik vannak, célszerű legalább vázlatosan megismerni az alprogramok megvalósításának módját. Ez a szakasz ezt a célt szolgálja.

A program végrehajtása során az alprogramok egymást hívják. Egy időben tehát több alprogram is lehet aktív, sőt, rekurzió esetén ugyanaz az alprogram többször is („több *példányban*"). Az alprogramok hívási szemantikája a verem adattípusra emlékeztet: mindig az az alprogram ér véget először, amelyik a legutóbb hívódott meg. Nem csoda, hogy az alprogramok hívását egy verem, az úgynevezett *végrehajtási verem* (runtime stack) segítségével

valósítják meg. Amikor egy alprogramot aktiválnak, információk kerülnek be róla a végrehajtási verem tetejére. Ha az alprogram újabb alprogramot aktivál, a hívott a hívó fölé kerül a veremben. Amíg a hívott alprogram aktív, addig a hívó is az marad. Amikor a hívott véget ér, az adatai kikerülnek a végrehajtási veremből, és a vezérlés visszakerül a hívóhoz. Ezután a hívónak lehetősége van újabb alprogramot hívni, ami megint a hívó fölé fog kerülni a végrehajtási veremben. Ha a hívó befejezte az összes utasítását, akkor adatai kikerülnek a veremből, és a vezérlés visszatér az őt hívó programegységhez.

Ezek szerint a program végrehajtásának egy tetszőleges pillanatában meg tudjuk mondani, hogy mely alprogramok aktívak, ha végignézzük a végrehajtási vermet. A verem tetején található az az alprogram, ahol épp a vezérlés van, azaz amelyben az éppen végrehajtott utasítás van. A végrehajtási veremben található bejegyzések azt mutatják meg, hogy melyik alprogram melyik másik alprogramot aktiválta. Pontosabban, ha a rekurziót is figyelembe vesszük: hogy melyik alprogram végrehajtási példány melyik másik alprogram végrehajtási példányt aktiválta. Minden példány (kivéve a legfelsőt a veremből) pontosan egy másikat aktivált, azt, amelyik fölötte van a veremben; hasonlóan, minden példányt (kivéve a főprogramot) pontosan egy másik példány aktivált, az, amelyik alatta van a veremben. Az egymást aktiváló alprogram végrehajtási példányokat *hívási láncnak* is nevezzük.

Pontosan milyen információkat tartalmaz a végrehajtási verem egy alprogram-példányról? Mindenféleképpen tárolnia kell azt a memóriacímet, amelyhez vissza kell térni akkor, amikor az alprogram végrehajtása befejeződött. Ezen kívül tárolnia kell azt, hogy mi volt a regiszterek értéke akkor, amikor meghívták. Ezek az információk azért szükségesek, mert amikor a hívott alprogram visszatér, a hívónak pontosan ugyanabban az állapotban kell tovább futnia, mint amilyen állapotban a híváskor volt. Azokat az adatokat, amelyeket a végrehajtási veremben egy alprogram-példányról tárolunk, a példány *aktivációs rekordjának* nevezzük.

A legtöbb programozási nyelvben az aktivációs rekord tartalmazni szokta az alprogram-példány lokális változóit és paramétereit is. Amíg az alprogram végrehajtás alatt áll, addig a végrehajtási veremben tartozik hozzá egy aktivációs rekord, amiben a lokális változói és paraméterei vannak. Amikor az alprogram véget ér, az aktivációs rekordja törlődik a végrehajtási veremből, és ezzel megszűnnek a lokális változói, illetve paraméterei is. Ez alól kivételt jelentenek azok a lokális változók, amelyeket statikus élettartammal deklaráltunk. Ezek a változók ugyanis nem a végrehajtási veremben tárolódnak, hanem egy fordítási időben meghatározott méretű és kiosztású tárterületen, az úgynevezett *statikus memóriaterületen.*

Ne felejtsük el azt sem, hogy az alprogramok rekurzívak is lehetnek: több példányuk is lehet egy időpillanatban aktív. Ilyenkor minden példánynak saját aktivációs rekordja van a végrehajtási veremben, és ebben a rekordban saját példányai vannak a lokális változókból. Épp azért tároljuk a lokális változókat az aktivációs rekordban, hogy a rekurzív alprogramok minden hívott példánya saját lokális változókkal rendelkezhessen. A FORTRAN-ban (egész a FORTRAN 90-ig) nem lehet rekurzív alprogramokat írni, ezért (vagy mert?) az alprogramok lokális változói statikus élettartamúak. A statikus lokális változók sokkal hatékonyabbá teszik a programokat, mert nem kell az alprogram hívásakor a megfelelő mennyiségű tárterületet lefoglalni, majd kilépéskor felszabadítani, és mert a lokális változók címét programbetöltéskor meg lehet határozni (egyébként indirekt címzéssel lehetne csak elérni a lokális változókat) – cserébe viszont elveszítjük a rekurzív alprogramok írásának lehe-

tőségét. A FORTRAN 90-ben és a PL/I-ben lehet alkut kötni: ha egy alprogramot úgy deklarálunk, hogy lehessen rekurzívan hívni, akkor a lokális változói a végrehajtási veremben fognak tárolódni, a többi alprogramra viszont a lokális változók statikus élettartamúak lesznek. Az igazsághoz hozzá tartozik, hogy a statikus lokális változók használata csak a futási időt tekintve hatékonyabb, az elfoglalt tárterületet tekintve nem. Ugyanis az összes használt alprogram összes statikus lokális változója az egész program végrehajtásának ideje alatt létezik, és foglalja a memóriát.

Az alprogramok aktivációs rekordja különböző méretű, hiszen ez a méret függ a lokális változók, valamint a formális paraméterek számától és típusától. Sőt, sok nyelvben ugyanannak az alprogramnak a különböző példányai is különböző méretű aktivációs rekordokkal rendelkeznek: gondoljunk arra, hogy például az Adában egy formális paraméter vagy egy lokális változó mérete csak futási időben dől el, az aktuális paraméterek függvényében. Ezért a hívások kezelése nem olyan egyszerű, mint elsőre gondolnánk. Az aktivációs rekordban azt is nyilván kell tartanunk, hogy hol ér véget a rekord, azaz, hogy hol kezdődik a hívó programegység aktivációs rekordja. Ezeket a mutatókat *dinamikus láncnak* (dynamic link) szokták nevezni.

A következő problémát a nem-lokális változók okozzák. A globális változók statikus élettartamúak, tehát a statikus memóriaterületen tárolódnak. Ezért használatuk nagyon egyszerű. Ezzel szemben azok a nem globális változók, amelyek egy alprogramra nézve nem-lokálisak, sokkal több vesződséggel járnak. A nem blokkszerkezetes nyelvekben, mint például a C, ahol minden nem-lokális változó statikus, sokkal egyszerűbb dolga van az alprogramoknak, mint a blokkszerkezetes nyelvekben. Hogyan lehet megtalálni egy nem-lokális, de nem is globális változót a végrehajtási veremben? Ha dinamikus hatóköri szabályok vannak érvényben, akkor a nem-lokális változó valamelyik hívó alprogramban található. El kell tehát indulni a dinamikus lánc mentén, és meg kell keresni az első olyan aktivációs rekordot, ahol szerepel az adott változó. Ha viszont a nyelv statikus hatókört használ, akkor nem a hívó, hanem a befoglaló alprogramokban találjuk meg a megfelelő változót. Szerencsére a statikus hatóköri szabályok értelmében egy beágyazott alprogram csak akkor lehet aktív, ha a befoglaló alprogram is aktív: hiszen a befoglaló alprogramon kívülről a beágyazott alprogram nem hívható meg. Tehát a végrehajtási veremben van valahol egy olyan aktivációs rekord, ahol a beágyazott alprogram befoglaló alprogramjának adatai vannak. Persze lehet, hogy a nem-lokális változó nem a közvetlen befogadóban, hanem egy közvetett befogadóban található. Ahhoz, hogy a befogadó alprogramok aktivációs rekordjait könnyebben megtaláljuk, a fordító minden alprogram aktivációs rekordjában elmenti a befoglaló alprogram aktivációs rekordjának kezdőcímét is. Ezt szokták *statikus láncnak* (static link) nevezni. Tehát egy nem-lokális változót a statikus hatóköri szabályok esetén a statikus lánc mentén haladó bejárással lehet megtalálni[24].

A fordító az alprogramok fordításakor nagyon sok optimalizációt végezhet el. Szó volt már korábban arról, hogy például dönthet úgy a fordító, hogy az alprogram szövegét makrószerűen kifejti a hívás helyén. Sőt, egyes nyelvekben a programozó még tanácsolhatja is ezt a fordítónak. Egy másik fontos, jól ismert optimalizációs módszer a rekurzív alprogra-

[24] Van egy másik elterjedt módszer a nem-lokális változók nyilvántartására és felkutatására, mely az úgynevezett *display* használatán alapul. Ezen módszer ismertetésére itt most nem térünk ki.

mok esetében alkalmazható. A rekurzió ugyanis, noha elegáns, könnyen olvasható kódot eredményező programozási technika, nagyon költséges mind a futási idő, mind a felhasznált tár tekintetében. Egy ciklus használata sokkal hatékonyabb. Vannak olyan rekurzív alprogramok, amelyeket a fordító könnyen át tud alakítani ciklusra: ezek az úgynevezett végrekurzív (tail-recursive) alprogramok. Ezekben a rekurzív hívás az alprogram utolsó utasításaként helyezkedik el. Ezt az optimalizációs technikát egy példa segítségével mutatjuk be. Tekintsük az alábbi bináris keresést megvalósító C függvényt [Set 96]:

```
#define yes 1
#define no 0
int X[ ] = { 0, 11, 22, 33, 44, 55, 66, 77 };
int T;
int search (int lo, int hi)
{
    int k;
    if (lo > hi) return no;
    k = (lo + hi) / 2;
    if (T == X[k]) return yes;
    else if (T < X[k]) return search(lo, k - 1);
    else return search(k + 1, hi);
}
```

A *search* függvény átalakítható erre az alakra:

```
int search (int lo, int hi) {
    int k;
L:  if (lo > hi) return no;
    k = (lo + hi) / 2;
    if (T == X[k]) return yes;
    else if (T < X[k]) hi = k - 1;
    else lo = k + 1;
    goto L;
}
```

A rekurzív hívás az alprogram legvégén szerepel, tehát amikor visszatér belőle a vezérlés, a hívó is egyből véget ér. Ez azt jelenti, hogy a hívó lokális változóinak (beleértve a formális paramétereket is) értékére már nincs többé szükség. Felesleges új példányt létrehozni belőlük a rekurzív hívás számára, használjuk inkább a meglévő változókat. Csak arra kell odafigyelni, hogy a rekurzív hívás aktuális paramétereinek megfelelően töltsük ki a formális paramétereket megvalósító változókat.

7.7.1. Paraméterként átadott alprogramok megvalósítása

A nem-lokális változók bonyolulttá tehetik a paraméterként átadott alprogramok megvalósítását is. A paraméterként átadott alprogram ugyanis hivatkozhat olyan változókra, amelyek nem benne lettek deklarálva, tehát nem részei az alprogram definíciójának. Az alprog-

ram végrehajtásához ezért nem elégséges magának az alprogramnak az ismerete, szükség van az alprogram környezetére – ami viszont lehet, hogy más, mint annak az alprogramnak a környezete, amelyiknek az alprogramot átadták paraméterként.

A nem blokkszerkezetes nyelvekben az alprogramok környezete nagyon egyszerű: az összes nem-lokális változó globális, azaz statikus élettartamú. Minden nem-lokális változóra történő hivatkozás feloldható programbetöltéskor, ezért ezekben a nyelvekben az alprogramok paraméterként történő átadása nem ütközik nagy nehézségbe: egyszerűen át kell adni az alprogram kódjának kezdőcímét. A C-ben és a C++-ban például így, alprogramokra mutató mutatókkal történik a paraméterként átadott alprogramok megvalósítása.

A blokkszerkezetes Modula-2-ben az a szabály, hogy csak legkülső szintű alprogramok adhatók át paraméterként, beágyazottak nem. Ez azt jelenti, hogy ebben a nyelvben is megoldható a paraméter-alprogramok átadása egy memóriacím segítségével: nem globális nem-lokális változóra nincs hivatkozás a paraméterként átadott alprogramból.

A Modula-3-ban (és még jópár más nyelvben) ellenben beágyazott alprogramokat is át lehet adni paraméterként. Ilyenkor a paraméterként átadott alprogram „visz magával" egy környezetet, amelyben végre kell hajtani. Ezt az együttest *lezártnak* (closure) szokás nevezni: a hordozott környezet lezárja az alprogramot, minden kifelé mutató hivatkozás az alprogramból feloldható ebben a környezetben. A lezárt átadásának megvalósítása meglehetősen bonyolult. A hatóköri szabályok szempontjából három megközelítés képzelhető el.

Sekély kötés (shallow binding) • A lezártban levő környezet megegyezik a paraméterként átadott alprogram hívójának környezetével. Ez a triviális megoldás: dinamikus hatókört feltételezünk, a paraméterként átadott alprogram úgy kerül végrehajtásra, hogy annak a környezetét látja, akinek paraméterként átadták. (Ezt a megközelítést alkalmazza például a SNOBOL nyelv.)

Mély kötés (deep binding) • A lezártban lévő környezet a statikus hatókörnek megfelelő, azaz a paraméterként átadott alprogram a végrehajtásakor azt látja, amit a definíciója helyén lát. Ez az a megközelítés, amit a nyelvek többsége követni szokott.

Ötletszerű kötés (ad-hoc binding) • A lezártban lévő környezet annak az utasításnak a környezete lesz, amely az alprogramot paraméterként átadta.

A függvény által a visszatérési értékben visszaadott alprogramok még bonyolultabb problémákat vetnek fel. Ezért például a Modula-3 megengedi ugyan, hogy egy függvény alprogramot adjon vissza, de kiköti, hogy csak legkülső szintű alprogramot szabad visszaadni.

A név szerinti paraméterátadást szintén lezárt átadásával szokás megvalósítani. Ezt teszi az ALGOL 60 is. A név szerinti paraméter helyett egy alprogram adódik át a megfelelő (mély kötéses) hivatkozási környezettel. Ez a (paraméter nélküli) alprogram hívódik meg, amikor a név szerinti formális paraméterre hivatkozás történik. Az alprogram a vele átadott hivatkozási környezetben lefut, és kiértékeli az aktuális paramétert. Látható, hogy a név szerinti paraméterátadás nagyon rossz hatékonyságú. A formális paraméterre történő minden hivatkozás egy alprogram végrehajtását vonja maga után.

7.8. Iterátorok

A 3.10.7. és 3.10.8. szakaszokban már megismerkedtünk az iterátor (vagy bejáró) fogalmával. Vessük most össze az iterátorokat az alprogramokkal.

Gyakran írunk ciklust olyankor, amikor valamilyen tevékenységet egy értéksorozat minden elemére szeretnénk elvégezni. Például, ha egy n szám faktoriálisát akarjuk kiszámolni, akkor az alábbi ciklust írhatjuk:

```
long faktoriális = 1;
for (int i = 1; i <= n; ++i)
  faktoriális *= i;
```

Azaz az *1*, *2*, ... *n* értékek mindegyikére szeretnénk elvégezni azt a tevékenységet, hogy az értéket hozzászorozzuk a *faktoriális* változóhoz. Egy másik példa lehet az, hogy egy adatszerkezet (például egy lista vagy egy bináris fa) minden elemét ki szeretnénk írni a képernyőre. Az értékek, amikre a tevékenységet el akarjuk végezni, az adatszerkezet elemei, míg a tevékenység az érték kiírása. Ha előre leprogramozzuk azt, hogy az elemeket ki lehessen egymás után nyerni az adatszerkezetből, akkor a feladat egy ciklus segítségével ismét könnyen megoldható:

```
while ⟨van még feldolgozatlan elem⟩
  ⟨vegyük a következőt⟩
  ⟨írjuk ki⟩
```

Az iterátorok ezt a fajta vezérlési szerkezetet valósítják meg egy nyelvi szerkezetként. Ahelyett, hogy a ciklus megírásának részleteivel bajlódnánk, fogalmazzuk csak meg magát a tevékenységet; a ciklusváltozó kezelését, a ciklus befejeződését ellenőrző, illetve a ciklus haladását biztosító utasításokat bízzuk az iterátorra. Minden programozó legalább ezerszer írt olyan ciklust, ami végigmegy egy tömb elemein és csinál velük valamit. Az összes ilyen ciklus lényegében ugyanúgy néz ki. Miért kellene minden egyes alkalommal megírni ezt a ciklust? Rejtsük el a részleteket egy újra felhasználható programegységbe, egy vezérlési absztrakcióba! Amikor szükség van egy ilyen ciklusra, akkor csak azt programozzuk le, ami egyedi benne: a tevékenységet, amit végre kell hajtani az elemek sorozatán. Paraméterezzük fel a megfelelő iterátort, és használjuk azt!

A CLU nyelvben például van egy olyan előre definiált iterátor, a `from_to`, amely az egész számok felsorolására szolgál egy adott `x. .y` intervallumban. Ennek az iterátornak a megvalósítását így képzelhetjük el:

```
from_to = iter (x, y : int) yields (int)
  i: int := x
  while i <= y do
    yield i
    i := i + 1
  end
end
```

A `yield` utasítás hatására az iterátor végrehajtása felfüggesztődik, és visszaadja az `i` értékét a hívó programegységnek. Az iterátor következő hívásakor a végrehajtás nem az iterátor elejétől indul, hanem folytatódik onnan, ahol a legutóbb abbahagyta: a felfüggesztéshez használt `yield` utasítást követő utasítástól. Az iterátor akkor ér véglegesen véget, amikor elfogytak az utasításai, azaz ebben az esetben akkor, amikor a `while` ciklus véget ért. Az iterátort a következőképpen használhatjuk a faktoriális-számításhoz:

```
faktorialis: int := 1
for i in from_to(1, n) do
  faktorialis = faktorialis * i
end
```

A `for in` szerkezet addig hívogatja az iterátort, amíg az tud valamilyen értéket visszaadni. Ha az iterátor véget ér, akkor befejeződik a `for in` végrehajtása is.

Egy általunk írt adatszerkezethez mi is elkészíthetünk egy iterátor műveletet, amellyel az adatszerkezetben tárolt elemeket lehet felsorolni. Ezután az adatszerkezet elemeinek feldolgozását, bármit is jelentsen ez a feldolgozás, ezen iterátor segítségével kényelmesen végezhetjük el.

Több olyan nyelv van, amely támogatja az iterátorok készítését. Ilyen nyelvek például a Sather, a Trellis vagy az Alphard. Ez utóbbi volt talán az első nyelv, amely bevezette az iterátorokat. Megjegyezzük, hogy az Alphard szóhasználata szerint az iterátorokat generátoroknak nevezik. Ez az elnevezés azért is találó, mert az iterátorokat egyfajta elem-generálóknak is tekinthetjük. Fontos követelmény, hogy a generálás során az iterátor nem épít az elemek reprezentációjára vagy sorrendjére. Az Alphardban az iterátorok nem mint atomi nyelvi konstrukciók, hanem mint speciális osztályok jelennek meg. Ezek az osztályok hasonló karakterisztikával és viselkedéssel bírnak, pontosabban minden generátor osztály rendelkezik az alábbi (vagy ahhoz hasonló) interfésszel:

- Legelső elem generálása (`&start`)
- Következő elem generálása (`&next`)
- Generálás befejezése (`&finish`)
- Aktuális elem kiolvasása (`&value`)
- Véget ért-e a generálás (`&done`)

Magától értetődik, hogy a fenti metódusok közül az első három eljárás, a negyedik lekérdezés, míg az utolsó predikátum. A nyelvben, illetve a nyelvhez az alábbi iterátorszerkezetek állnak rendelkezésre:

- `upto` (egészekkel megadott zárt intervallumon való iterálás)
- `stepup` (hasonló mint az upto, csak lépésköz is megadható)
- `first` (megadott feltételeknek megfelelő első elem megkeresése)
- `invec` (vektor elemeinek sorra vétele)

Az alábbi Alphard nyelvű példa a generátorok kombinálását és magas absztrakciós erejét mutatja be.

```
first i from upto(1,n) suchthat A[i]>max then max:=A[i]; maxhely:=i fi
```

A fenti forrás az angolul tudó olvasó számára szinte azonnal megérthető a magas kifejezőerőnek köszönhetően. A kódrészlet egy tömb megadott intervallumába eső elemei közül az első olyan elemet keresi meg, mely nagyobb a `max` változóban megadott elem értékénél. Ennek az elemnek az indexe kerül a `maxhely` változóba.

További érdekesség, hogy a nyelv tervezői – kielemezve az iterátorok viselkedését és kihasználva azt a tényt, hogy az iterátorok maguk is osztályok – az iterátorokhoz verifikációs szabályokat is definiáltak, és az iterátorokat verifikálták. Ennek jelentősége abban rejlik, hogy feltételezve azt, hogy az iteráció során alkalmazott predikátum (például `suchthat` kulcsszó után kerülő kifejezés) és az elemeken végzett műveletek terminálnak, az iterátorokkal készült konstrukciók maguk is terminálnak és megfelelnek az elvárt működésnek.

A Java is bevezeti az iterátor fogalmát a *java.util.Enumeration* és a *java.util.Iterator* interfészek segítségével. A Java az iterátor megvalósításához szintén adatabsztrakciót (objektumot) használ. Az iterátor belső állapotát nem egy megszakadt, felfüggesztett „alprogram" belső állapota jelenti, hanem egy objektum belső állapota, amelynek műveleteivel az iterátort léptethetjük. Egy *v* vektor elemeit így írnánk ki a Javában:

```
Enumeration e = v.elements();
while (e.hasMoreElements())
  System.out.println (e.nextElement());
```

A C# még elegánsabb nyelvi szerkezetet biztosít egy iterátor objektummal történő bejárásra.Tegyük fel, hogy *numbers* egy egészeket tartalmazó adatszerkezet, amit fel lehet sorolni (például egy tömb).

```
foreach (int x in numbers) {
    Console.WriteLine(x);
}
```

A C++ iterátorait [Jos 99] a Standard Template Library adatszerkezeteinek bejárására használhatjuk. Az adatabsztrakció mögé rejtett iterátor-viselkedést ötvözik az operátor-túlterhelés lehetőségeivel, így használatuk formailag megegyezik a mutatók használatával. Az adatszerkezeteket megvalósító osztályok szoktak tartalmazni egy *iterator* nevű tagot, ami egy típus (melynek tényleges reprezentációja az adatszerkezet függvénye), valamint az adatszerkezet elejét és végét jelző iterátorokat (pozíciókat) visszaadó *begin()* és *end()* tagfüggvényeket. Tekintsük például a *list* adatszerkezetet. Egy egészeket tartalmazó *adatok* lista elemeit a következőképpen írhatjuk ki a szabványos kimenetre:

```
list<char>::iterator pos;
for (pos = adatok.begin(); pos != adatok.end(); ++pos) {
  cout << *pos << ' ';
}
```

A C++ iterátorok a rajtuk keresztül elvégezhető műveletek alapján két csoportba sorolhatók: azok az iterátorok, amelyeken keresztül csak kiolvasni lehet az adatszerkezet elemeit (ezek a konstans iterátorok, melyek típusát általában a *const_iterator* tag tartalmazza), illetve azok, amelyeken keresztül módosítani is lehet az elemeket. Egy másik fajta csoportosítása

az iterátoroknak azon alapszik, hogy az adatszerkezet elemeit milyen sorrendben tudják visszaadni (előrefelé, előre- és visszafelé, illetve közvetlen elérésű (random access) módon), vagy, hogy az adatszerkezet végigolvasására (input iterator) vagy feltöltésére (output iterator) használhatók.

7.9. Korutinok

A korutinok az alprogramokhoz nagyon hasonló programegységek. Végrehajtásuk a szimmetrikus vezérlési modell szerint történik, szemben az alprogramok aszimmetrikus (hívó–hívott) vezérlési modelljével. A korutin nem egyben hajtódik végre, mint egy alprogram, hanem szakaszosan. Az alprogramok végrehajtásánál a hívó alprogram-példány nem kapja vissza a vezérlést addig, amíg a hívott alprogram be nem fejeződik. A korutinok esetében más a helyzet. Egy korutin előbb is átadhatja a vezérlést egy másik korutinnak, mint hogy befejeződne. Amikor újra meghívják, ott folytatja a működését, ahol előzőleg abbahagyta. A korutinok tehát úgy adogathatják egymás között a vezérlést, ahogy nekik tetszik: nincsenek a végrehajtási verem szigorú LIFO szabályaihoz kötve, mint az alprogramok. A korutinok hívását ezért gyakran nem is hívásnak (call) nevezzük, hanem *folytatásnak* (resume).

Milyen feladatokhoz használhatjuk a korutinokat? Az egyik legfontosabb felhasználási területük a párhuzamosság látszatának keltése. Egy korutint megfeleltethetünk egy „folyamatnak", egy logikailag összefüggő utasítássorozatnak. A korutinok egymással egy időben hajtódnak végre, csakúgy, mint a folyamatok egy konkurens rendszerben. Persze egy időben valójában csak egy korutin aktív, de ha gyakran adogatják át egymásnak a vezérlést, a párhuzamosság látszatát tudják kelteni. Egyes nyelvek a párhuzamosság megvalósítására ügyesebb nyelvi elemeket kínálnak, például végrehajtási szálakat, mint az ALGOL 68, a Modula-3, az Ada, az Occam vagy a Java. Erről a 14. fejezetben részletesebben is beszélünk.

Az első magas szintű programozási nyelv, amely a korutinokat támogatta, a SIMULA 67 volt. Ezt a nyelvet, amint a neve is sugallja, szimulációk készítéséhez találták ki. A szimulációkhoz gyakran van szükség önálló folyamatok modellezésére, így kerültek a nyelvbe a korutinok. Más nyelvek is átvették ezt a nyelvi eszközt, például a BLISS, az Interlisp vagy a Modula-2.

Az iterátorok és a korutinok rokon fogalmak. Ha egy nyelvben vannak korutinok, akkor abban a nyelvben az iterátorokat sem lenne nehéz megvalósítani [Sco 00]. Tekintsük azt a CLU programrészletet, amely a `from_to` iterátor segítségével faktoriálist számolt:

```
for i in from_to(elso,utolso) do
    ...
end
```

Ezt a fordítónak a következő oldalon látható kódra kellene alakítania (a megfogalmazáshoz ALGOL-szerű szintaxist használunk).

```
begin
    i: int;
    vege: bool;
    it: coroutine := new from_to(elso, utolso, i, vege, current_corutine);
    while not vege do
        ...
        transfer it
    end
    destroy it
end
```

A ciklus elején létrehozunk egy korutint, mely az *i* és *vege* változókat fogja arra használni, hogy visszaadja a következő számot, és hogy jelezze, ha elfogytak a számok. A **transfer** segítségével újra és újra átadjuk a vezérlést a korutinnak, amíg az a *vege* változót igazra nem állítja. A ciklus után megszüntetjük a korutint.

Az iterátor megvalósítása a következő lehet:

```
coroutine from_to (from, to: int; ref i: int; ref done: bool; caller: coroutine);
    i := from
    done := from <= to
    detach
    loop
        i := i+1
        done := i <= to
        transfer caller
    end
end
```

A korutin meghívásakor máris beállítjuk a cím szerint átvett *i* és *done* paraméterekbe az első visszaadandó adatot, majd a **detach** segítségével leválasztjuk az új korutint a régiről. A korutin következő hívása bevisz a ciklus törzsébe, ami – amíg meg nem szüntetik a korutint – újra és újra ki fog értékelődni a korutin meghívásakor. A ciklusmag utolsó utasításával, amikor az *i* és *done* változókban már elhelyeztük a következő kiszámított adatot, visszaadjuk a vezérlést a hívó korutinnak.

7.10. Összefoglalás

Ebben a fejezetben a vezérlési absztrakcióról, és a megvalósítására használható nyelvi elemről, az alprogramról volt szó. Megtudtuk, hogy az alprogramok használata a kód bonyolultságát csökkenti, az elkészített szoftvertermék újrafelhasználhatóságát, olvashatóságát, módosíthatóságát és karbantarthatóságát pedig növeli.

Kétféle alprogramot különböztettünk meg a hívás módja szerint. Az utasításként hívható alprogramokat eljárásnak, a kifejezésben hívhatókat függvényeknek neveztük. Megtanultuk, hogy a függvények készítésekor a mellékhatásokat kerülni kell.

Az alprogramok bemenő paramétereken keresztül kapják meg a bennük megfogalmazott számításhoz szükséges információkat, és kimenő paramétereken, illetve függvények esetén a visszatérési értéken keresztül juttatják el a hívónak a számítás eredményét. Az alprogramok a számukra nem-lokális változók értékét is használhatják, megváltoztathatják. Az alprogramok specifikációja tartalmazza azokat az információkat, amelyek szükségesek az alprogram meghívásához. A fordítóprogram ellenőrizheti, hogy a hívások megfelelnek-e ennek a specifikációnak.

A nyelvek a bemenő, a kimenő, valamint a be- és kimenő alprogram-paramétereket különböző paraméterátadási módok (érték, cím, eredmény, érték/eredmény és név szerinti stb.) segítségével modellezhetik. Egy alprogram paraméterei lehetnek újabb alprogramok is. A paraméterként átadott alprogramok megvalósítása egy blokkszerkezetes, statikus hatóköri szabályokat követő nyelvben a lezárt fogalmára épít.

Az alprogramok rugalmas használatát a különböző nyelvek a változó hosszú paraméterlistával, a formális paraméterek alapértelmezett értékével, a túlterheléssel, a csak részben meghatározott típusú paraméterekkel és a programozó által definiált operátorokkal támogathatják. A rekurzív alprogramok, az alprogram típusok, valamint a Curry-módszer tovább növelhetik a nyelvek használhatóságát.

Feladatok

1. Válaszoljon röviden a következő kérdésekre!
   a) Mi a különbség az eljárások és a függvények között?
   b) Mit jelent az, hogy egy függvénynek van vagy nincs mellékhatása?
   c) Milyen célt szolgál az alprogram specifikációja, és milyen elemekből áll?
   d) Mi a formális és mi az aktuális paraméter?
   e) Mi lehet paramétere egy alprogramnak? Adjon példákat különböző nyelvekből!
   f) Miért jó, ha a fordító ellenőrzi az aktuális paraméterek típusát?
   g) Mit jelent a paraméterek név szerinti megfeleltetése?
   h) Miért jó, ha a formális paraméternek lehet alapértelmezett értéke?
   i) Mikor van szükség arra, hogy alprogramokat előre deklaráljunk?

j) Milyen esetben célszerű inline alprogramokat használni? Miben különböznek a makróktól?

k) Az információáramlás irányát tekintve milyen paramétereket különböztethetünk meg?

l) Hogyan működik az érték, a cím, az eredmény és az érték/eredmény szerinti paraméterátadás?

m) Milyen szemantikus és hatékonyságbeli problémákat vet fel az érték, a cím, az eredmény és az érték/eredmény szerinti paraméterátadás?

n) Mi a különbség a név szerinti paraméterátadás és a szövegszerű helyettesítés között?

o) Milyen nyelvekben, és hogyan lehet előírni azt, hogy az alprogram ne változtathassa meg egy formális paraméter értékét? Miért jó ez?

p) Mit jelent a másolat, a megosztás és az igény szerinti paraméterátadás? Mely nyelvek kapcsán jöttek szóba?

q) Mit jelent az alprogramnevek túlterhelése?

r) Milyen nyelvekben lehet operátorokat túlterhelni, illetve definiálni?

s) Mire használható a végrehajtási verem?

t) Mi az aktivációs rekord?

u) Mire használják a dinamikus és a statikus láncokat?

v) Mi a lezárt?

w) Mire használjuk az iterátorokat?

x) Hogyan hajtódnak végre a korutinok?

2. Írjon olyan függvényt, ami egy tömbben tárolt természetes számokról eldönti, hogy relatív prímek-e!

3. Készítsen „quicksort" algoritmus alapján rendező eljárást!

4. Készítsen mátrixot transzponáló alprogramot! Írja meg függvényként és eljárásként is. Melyiket mikor célszerű használni?

5. Készítsen olyan eljárást, amely kiírja a képernyőre, hogy a paramétere érték, cím, érték/eredmény vagy név szerint, esetleg szövegszerű helyettesítéssel lett-e átadva!

6. Írjon olyan alprogramot, mely egy paraméterként megadott intervallumban megkeresi a szintén paraméterként megadott függvény legelső zérushelyét! Oldja meg ezt a feladatot minél több nyelven!

7. Készítsen C és Ada nyelven olyan függvényt, mely két mátrixot összeszoroz! Hasonlítsa össze a felhasznált nyelvi elemeket! (Operátor-túlterhelés, paraméterek száma, a paraméterek típusának meghatározottsága.)

8. Írjon olyan alprogramot, amely kiszámolja az n. Fibonacci-számot! Oldja meg a feladatot ciklussal és rekurzióval is. Milyen hatékonyságbeli különbségek figyelhetők meg? Készítsen olyan rekurzív alprogramokat erre a feladatra, amelyek hatékonysága nem lesz sokkal rosszabb, mint a ciklust használó megoldás!

8 Kivételkezelés

A szoftverfejlesztés során alapvető fontosságú, hogy az előállított termék helyesen és megbízhatóan működjön, azt a feladatot végezze el, amit a vásárló kitűzött és semmilyen körülmények között ne okozzon nem várt mellékhatást. Ezen célok elérésének segítésére vezettek be több programozási nyelvben kivételkezelő és helyességbizonyító eszközöket. Ebben a fejezetben a futási időben fellépő hibák kezelésére szolgáló módszereket és eszközöket vizsgáljuk meg.

A kivételkezelés feladata olyan események kezelése, amelyek nehézzé vagy egyenesen lehetetlenné teszik a program futásának normális folytatását. Ilyen esemény lehet például nullával osztás, nullpointer[1] által mutatott értékre hivatkozás, sikertelen típuskonverzió, hiba az operációs rendszerben vagy hardverben, vagy kommunikációs probléma (megszakad a kapcsolat, vagy a „túloldal" nem tartja be a protokollt). A kivételkezelés az ilyen problémák kezelésére nyújt többé-kevésbé elegáns eszközöket.

8.1. Bevezetés

Már a legelső magas szintű programozási nyelv, a FORTRAN is nyújtott eszközöket programfutás közben fellépő hibák kezelésére, habár ez csak a be- és kimenet (*input/output*) hibáira korlátozódott. Az első programozási nyelv, amely kifejezetten általános kivételkezelésre szolgáló eszközöket nyújtott, a PL/I volt. Ezt követte később többek közt a CLU, az Ada, a C++, az Eiffel, a Delphi, a Java és a C#. Más nyelvekben is vannak olyan nyelvi vagy könyvtári eszközök, amelyekkel kivételkezelést lehet megvalósítani (például a C nyelvben a *goto* utasítás, illetve a szabványos könyvtár *setjmp* és *longjmp* függvényei), ezeket mégsem lehet kivételkezelő eszközöknek hívni, mert nem kifejezetten erre a célra találták ki vagy éppen nagyon nehézkes a használatuk.

8.1.1. A kivételkezelés alapfogalmai

Ugyan a legtöbb nyelv kivételkezelő mechanizmusa kicsit különbözik a többitől, vannak közös alapfogalmaik (bár néha másképpen hívják ezeket).

A kivétel olyan futás közben fellépő hiba, ami megszakítja az utasítások normális végrehajtási sorrendjét. A kivétel lehet hardver (illetve operációs rendszer) által észlelt kivétel vagy szoftver (maga a szoftvertermék, illetve ennek keretrendszere) által észlelt kivétel. Amikor a hardver észlel kivételes helyzetet, azt általában megszakítással (angol néven *interrupt*) jelzi az operációs rendszernek, az pedig az alkalmazásnak. Az első kivételcsoportba tartozik többek között a nullával osztás vagy egy fájlból a fájl végén túli olvasás. Az utóbbi csoportba

[1] C++-ban a *NULL*, a Javában **null**, az Eiffelben *Void*

tartozik többek között egy tömb melléindexelése. A kivételkezelés feladata az ilyen futási hibák kijavítása, ha lehetséges, vagy a program körülményekhez képest legkisebb kárt okozó leállítása[2] (például a már feldolgozott adatok mentése leállás előtt), ha a hiba nem javítható.

A legtöbb nyelvben lehet saját kivételeket (*exception*) definiálni. Erre vagy már meglévő nyelvi elemeket (osztályokat, objektumokat stb.) használnak vagy külön kivétel-definiáló utasításokat vezettek be. Egyes nyelvekben magába a nyelvbe építettek be kivételeket – erre példa az Ada 95 *CONSTRAINT_ERROR*-ja –, míg más nyelvekben csak a nyelvhez tartozó szabványos könyvtárak nyújtanak előredefiniált kivételeket – ilyen például a Java *NullPointerException*-je. Néhány nyelvben kivétel dobásakor paramétert is lehet átadni a kivételeknek (ha a kivétel egy objektum, akkor például az objektum adataival (*attribute*-jaival) lehet paraméterezni a kivételt), míg más nyelvekben erre nincs lehetőség.

A kivételek kiváltását kivételdobásnak (*throw*) nevezzük (egyes nyelvek a *raise*, a *signal* vagy a *trigger* szavakat használják erre a fogalomra). A legtöbb nyelv esetében „maguktól" is keletkezhetnek kivételek (jellemzően a hardver által észlelt kivételek ilyenek), de általában a programozó is dobhat kivételt. A kivételeket kivételkezelő nyelvi elemekkel (*exception handler*) kapják el és kezelik. Az adott kivételhez tartozó kivételkezelő megkeresésére is többféle megoldást használnak az egyes nyelvek.

A program adott logikai szintjén nem kezelt kivételek továbbdobódhatnak (*propagation*), azaz az alprogramhívási láncban (*call chain*) feljebb szereplő alprogramokban újra kiváltódhatnak. Ez nagyon fontos eszköz arra, hogy a program a megfelelő logikai szinten tudja kezelni a hibát. Tegyük fel például, hogy programunk adatok rekordjait olvassa ki egy fájlból. A program úgy működik, hogy a főprogram megnyitja a fájlt, majd a fájlleírót (*file descriptor*) átadja a rekordolvasó alprogramnak, ami aztán kiolvassa a rekordokat. Ekkor a rekordolvasó alprogramban keletkezik kivétel, ha érvénytelen adatok vannak a fájlban, de ennek az alprogramnak nem áll rendelkezésére minden adat a hiba kezelésére, ami ebben az esetben a fájl nevének és egy hibaüzenetnek a kiírása. Ilyenkor célszerű továbbdobni a kivételt. Egyes nyelvekben az alprogramok, illetve metódusok specifikációjához hozzátartozik azoknak a kivételeknek a felsorolása is, amelyeket az adott alprogram dobhat. Azt, hogy az alprogram ténylegesen nem dob más kivételt, a fordító vagy a futásidejű keretrendszer ellenőrzi.

Az újabb nyelvekben általában van lehetőség arra, hogy egy blokk után mindenképpen lefusson egy kódrészlet (*finally* blokk), akár keletkezett kivétel, akár nem, akár kezelték a kivételt, akár nem. Ez elősegíti, hogy a lefoglalt erőforrásokat kivételes helyzetben is felszabadítsuk.

Ebben a fejezetben nem foglalkozunk sem a logikai, sem a funkcionális nyelvek kivételkezelésével, hanem a megfelelő programozási megközelítést bemutató fejezetekben (a 17., illetve a 16. fejezetek) tárgyaljuk ezeket. Ennek az az oka, hogy az ilyen nyelvekben előforduló kivételkezelés megértéséhez szükséges fogalmakat csak azokban a fejezetekben vezetjük be.

[2] Ebben a fejezetben a program leállásán azt értjük, hogy valamilyen hiba következtében fejezi be működését a program.

8.1.2. Miért jó a kivételkezelés?

Ha nincs elegáns kivételkezelés, akkor a programozónak minden „veszélyes" utasítás után meg kellene vizsgálnia, hogy nem történt-e valami végzetes hiba, és ha igen, akkor ezt kezelnie kell. Nézzük például a következő C nyelvű [Ker 89] programrészletet, amely soronként lemásolja a `copy.c` fájlt `copy.c.bak` néven:

```
const int buflen = 512;

void fv(void)
{
    FILE* input = NULL;
    FILE* output = NULL;
    char* line = NULL;

    line = (char*)calloc(1, buflen + 1); /* Memória lefoglalása. */
    input = fopen("copy.c", "r");        /* Fájlok megnyitása. */
    output = fopen("copy.c.bak", "w");

    fgets(line, buflen, input);          /* Egy sor beolvasása. */
    while (!feof(input)) {               /* Amíg nem értünk a fájl végére. */
        fputs(line, output);             /* A sor kiírása. */
        fgets(line, buflen, input);
    }
    fclose(input);                       /* Fájlok lezárása. */
    fclose(output);
    free(line);                          /* Memória felszabadítása. */
}
```

Ez a programrészlet ugyan megoldja a feladatot, de mi történik, ha például nem sikerül megnyitni a `copy.c` fájlt? Ekkor az *input* mutató értéke *NULL* lesz, és az első ráhivatkozáskor „Segmentation fault"-tal[3] (vagy ennek megfelelő hibaüzenettel) leáll a program. Ezt semmiképpen nem nevezhetjük megbízható programnak.

C nyelven többféle helyes megoldást adhatunk erre a feladatra. Az első megoldás az, hogy minden „veszélyes" utasítás után megnézzük, történt-e hiba, és ha történt, akkor ott helyben kezeljük. Egy másik megoldás, hogy nem helyben kezeljük, hanem a **goto** utasítással egy közös hibakezelő-részre ugrunk. Ezt nem csak a **goto** utasítással, hanem a *setjmp* és *longjmp* függvényekkel is megvalósíthatjuk, de ez utóbbiak használata nagyban rontja a programkód olvashatóságát és érthetőségét.

[3] A Segmentation fault a UNIX vagy UNIX-szerű operációs rendszerek jellegzetes hibaüzenete ilyen esetben. Windows operációs rendszerben a „General Protection Failure" vagy „Általános védelmi hiba" hibaüzenet felel meg ennek.

Az első megoldás a következőképpen néz ki (az *fv* függvény hibakódot ad vissza, ha nem sikerült az algoritmus valamelyik lépését végrehajtani, továbbá az *errno* globális változó is felveszi a hibára jellemző értéket):

```
const int buflen = 512;

int fv(void)
{
    FILE* input = NULL; FILE* output = NULL;
    char* line = NULL; char* read;
    int written;

    line = (char*)calloc(1, buflen + 1);
    if (NULL == line)
        return NO_MEMORY;
    input = fopen("copy.c", "r");
    if (NULL == input) {
        free(line);
        return CANNOT_OPEN_INPUT;
    }
    output = fopen("copy.c.bak", "w");
    if (NULL == output) {
        free(line);
        fclose(input);
        return CANNOT_OPEN_OUTPUT;
    }
    read = fgets(line, buflen, input);
    if (NULL == read) {
        free(line);
        fclose(input);
        fclose(output);
        return READ_ERROR;
    }
    while (!feof(input)) {
        written = fputs(line, output);
        if (written < 0) {
            free(line);
            fclose(input);
            fclose(output);
            return WRITE_ERROR;
        }
        read = fgets(line, buflen, input);
        if (NULL == read) {
            free(line);
            fclose(input);
            fclose(output);
            return READ_ERROR;
        }
    }
```

```
    fclose(input); fclose(output);
    free(line); return OK;
}
```

A fenti programrészletben a hibakezeléstől szinte nem is lehet látni azt, hogy tulajdonképpen mit csinál a program. Ráadásul mindegyik hibakezelő programrészlet majdnem ugyanazt csinálja: felszabadítja a lefoglalt erőforrásokat és hibakódot ad vissza. Sokkal célszerűbb lenne az ugyanolyan jellegű hibák kezelését egy helyen elvégezni.

A **goto** utasítás segítségével a C-ben a következő módon lehet egy helyre tenni a hibakezelést:

```
const int buflen=512;

int fv(void)
{
    FILE* input = NULL; FILE* output = NULL;
    char* line = NULL; char* read;
    int written; int hiba;

    line = (char*)calloc(1, buflen + 1);
    if (NULL == line) {
        hiba = NO_MEMORY;
        goto err;
    }
    input = fopen("copy.c", "r");
    if (NULL == input) {
        hiba=CANNOT_OPEN_INPUT;
        goto err;
    }
...
    while (!feof(input)) {
        written = fputs(line, output);
        if (written < 0) {
            hiba = WRITE_ERROR;
            goto err;
        }
        read = fgets(line, buflen, input);
        if (NULL == read) {
            hiba = READ_ERROR;
            goto err;
        }
    }
    hiba = OK;
err:
    if (NULL != input) fclose(input);
    if (NULL != output) fclose(output);
    if (NULL != line) free(line);
    return hiba;
}
```

Az előző megoldáshoz képest csak annyit nyertünk, hogy a függvény lefutása után elvárt állapotot helyreállító utasítások egy helyre kerültek a programban (ezek nem feltétlenül állnak egy-két utasításból), de továbbra is minden egyes veszélyes függvényhívás után meg kell vizsgálnunk a visszatérési értékét, és ez olvashatatlanná teszi a kódot.

A fenti megoldásoknak az a fő problémájuk, hogy a hibakezelést „kézzel" kell megírni minden „veszélyes" utasítás után, ugyanis a nyelv nem ad lehetőséget arra, hogy ne helyben, közvetlenül a hibát okozó utasítás után, hanem egy külön hibakezelő részben kezeljük a hibát. További probléma, hogy a hibát különbözőképpen jelzik a különböző függvények. Egyes függvények *NULL*-t adnak vissza hiba esetén, mások *-1*-et vagy bármilyen negatív számot. Ugyan a C könyvtári függvények többé-kevésbe egységesen ezeket a visszatérési értékeket használják hibajelzésre, de néhány esetben ez nem járható út (mint azt később látni fogjuk).

Nézzük, hogy egy képzeletbeli kivételkezeléssel kiegészített C nyelven hogy nézne ki a fenti függvény, ha a használt könyvtári függvények kivételeket dobnának. Ez a képzeletbeli kivételkezelés úgy működik, hogy a **try** blokkban keletkezett kivételeket a blokkot követő **catch** blokkokban lehet kezelni. A **catch** blokkokat követő **finally** blokk tartalma mindig lefut, akár keletkezett kivétel, akár nem (ezt a kivételkezelést a Java nyelv kivételkezelésé-ről mintáztuk, lásd a 8.3.6. pontot). Feltételezzük, hogy a használt könyvtári függvények *MemoryException*, illetve *IOException* kivételeket dobnak futási idejű hiba esetén:

```
const int buflen = 512;

int fv(void)
{
  FILE* input = NULL;
  FILE* output = NULL;
  char* line = NULL;

  try {
    line = (char*)calloc(1, buflen + 1);
    input = fopen("copy.c", "r");
    output = fopen("copy.c.bak", "w");
    fgets(line, buflen, input);
    while (!feof(input)) {
      fputs(line, output);
      fgets(line, buflen, input);
    }
  }
  catch (MemoryException e) { ret = NO_MEMORY; }
  catch (IOException e) { ret = e.error_code; }
  finally {
    if (NULL != line) free(line);
    if (NULL != input) fclose(input);
    if (NULL != output) fclose(output);
    return ret;
  }
}
```

Láthatjuk, hogy a **try** blokk belsejében szereplő utasítások megegyeznek az első példánkban szereplő hibakezelés nélküli változat utasításaival, azaz ez a kód ugyanolyan olvasható lett, mint az eredeti, de kezeljük a hibákat. Ha egyáltalán nem szeretnénk a hibákkal foglalkozni ebben a függvényben, akkor kivételkezelés esetén ezeket egyszerűen tovább lehet adni (pontosabban nem kell kezelni, csak specifikálni a kivételeket), míg kivételkezeléssel nem rendelkező nyelvekben ezt csak nehézkesebben lehetne megoldani, esetleg három-négy függvényhívás „mélységből" kellene hibakódokon keresztül visszaadni a hiba okát. További előnye ennek a megoldásnak, hogy minden esetben felszabadítjuk a lefoglalt erőforrásokat, nem fordulhat elő, hogy a kód esetleges kibővítése után egy új hiba kezelése során nem tesszük meg ezt, hiszen a **finally** blokk tartalma minden esetben lefut.

A kivételkezelés egy további előnye, hogy elkülönül az alprogram (függvény) visszatérési értéke a hibát jelző kódtól. Nézzük a következő példát: az alábbi C nyelven írt függvény a paraméterként megadott szöveget számként próbálja értelmezni és az ennél eggyel nagyobb számot adja vissza. Ezt a feladatot a program a C nyelv szabványos könyvtárában található *atoi* függvény segítségével oldja meg:

```
int fv(char* s)
{
  return atoi(s) + 1;
  /* semmi hibakezelés! */
}
```

A függvény jól működik egészen addig, amíg a paramétere tényleg szám. Azonban ha a bemenet nem értelmezhető számként, akkor a függvény nem tud hibát jelezni, hiszen az *atoi* nem használhatja az egész értéket visszaadó C könyvtári függvények esetében szokásos hibajelzést (*-1*-es visszatérési érték), mert ez a (megengedett) `"-1"` bemenethez tartozó eredmény. Sőt, az összes egész érték lehet eredmény, így az *atoi* függvény nem tud hibát jelezni.

Többféle módon lehet megoldani ezt a problémát. Egy lehetséges megoldás az, hogy nem egész típusú értéket ad vissza a függvény, hanem egy rekordot (struktúrát), aminek egyik mezője azt tartalmazza, hogy sikeres volt-e a művelet, a másik mezője pedig magát az értéket. Ez azonban egy kicsit nehézkesen használható, hiszen nem lehet a függvényt olyan kifejezésbe írni, ami egész értéket vár.

```
struct atoi_eredmeny {
  int egesz;
  int hiba;
};
struct atoi_eredmeny atoi2(char* s);

int fv(char* s)
{
  struct atoi_eredmeny eredmeny;
  eredmeny = atoi2(s);
  if (!eredmeny.hiba) return eredmeny.egesz + 1;
  /* hibakezelés */
}
```

Egy másik megoldás az, hogy az egész értéket egy további paraméterként adjuk át (cím szerint), a visszatérési érték pedig a hibát jelzi. Ez is kissé kényelmetlen, mert mutatót kell csinálni az egész típusú változóból, továbbá az ilyen „mellékhatások" nehezítik a program megértését. Persze lehetne fordítva is, hogy a hibakódot adjuk át paraméterként, de ettől ugyanolyan nehézkes maradna a megoldás.

```
int atoi3(char* s, int* i);

int fv(char* s)
{
  int i;

  if (-1 != atoi3(s, &i)) return i + 1;
  /* hibakezelés */
}
```

A szintén szövegből számot készítő *strtol* könyvtári függvény hasonló módszert használ hiba jelzésére: egy mutatót kell cím szerint megadni paraméterként, amely sikertelen végrehajtás esetén a szöveg elejére mutat a függvény visszatérése után.

Egy harmadik megoldás a problémára, hogy egy globális változó tartalmazza, hogy a legutóbbi függvényhívás sikeres volt-e, vagy sem. Ez azonban párhuzamos környezetben vet fel problémákat, amikor ugyanazt a globális változót egyszerre többen akarják írni.

```
extern int atoi_hiba;

int atoi4(char* s);

int fv(char* s) {
  int i;

  i=atoi4(s);
  if (!atoi_hiba) return i+1;
  /* hibakezelés */
}
```

Több C könyvtári függvény is a közös *errno* változót használja részletesebb hibajelzésre. Ennek a megoldásnak párhuzamos környezetben fellépő problémákon túl is van egy további hátránya: az egymás után következő függvényhívások felülírhatják a saját hibakódukkal a közös változót és így elveszhet a hiba eredeti oka, ha nem figyel a programozó.

Ha az *atoi* függvény kivételt dobna, akkor ezek a problémák nem merülnének fel, jól elkülönülne a hiba a visszatérési értéktől.

Az objektumorientált nyelvekben a konstruktorok kicsit hasonlóan működnek a fenti példában szereplő függvényhez, hiszen csak az elkészült objektumot (illetve hiba esetén nullpointert vagy ennek megfelelő objektumot) adhatnak vissza, és ez gyakran nem elég a megfelelő hibakezeléshez, mert egy bonyolultabb konstruktorban sokféle hiba történhet. A kivételkezelés ilyenkor is jól jön.

8.1.3. A kivételkezelések összehasonlításának szempontjai

A következő pontokban a különböző nyelvek kivételkezelését fogjuk összehasonlítani. Ehhez szükséges meghatározni, hogy milyen szempontok alapján vizsgáljuk meg a nyelvek kivételkezelését.

- Milyen nyelvi elem a kivétel? Lehet csoportosítani a kivételeket? Lehet hierarchiába szervezni a kivételeket? A kivételek csoportosítása azért fontos, hogy egyszerűen el lehessen különíteni az „érdekes" (a program adott logikai szintjén kezelendő vagy kezelhető) és az „érdektelen" (a program egy más szintjén kezelendő) kivételeket.
- Milyen nyelvi elemhez kapcsolódik a kivételkezelés (utasítás, blokk, alprogram, egyéb)?
- A kivétel kezelése után honnan folytatódik a program (a kivétel keletkezésének helye után, a kivételkezelő után vagy újraindul a kivételt eredményező blokk)?
- Tovább dobódhat-e a kivétel? Mi történik a kezeletlen kivétellel? A kivételkezelő dobhat kivételt?
- Paraméterezhető-e a kivétel? A kivétel típusa nem mindig nyújthat elégséges információt a hiba kezeléséhez.
- Van olyan nyelvi elem, ami mind kivételes, mind normális végrehajtás esetén *mindenképpen* végrehajtódik, azaz van *finally*? Ennek csak akkor van értelme, ha a kivételkezelés egy-egy utasításnál nagyobb egységhez kapcsolódik.
- Lehet-e és ha igen, kell-e specifikálni a kivétel dobását? Nagyon fontos, hogy a nyelv rákényszeríti-e a programozót a kivételek kezelésére még fordítási időben, vagy csak futási időben, tesztelés közben derül ki, hogy bizonyos kivételeket nem kezel a program.
- Hogyan viselkednek a kivételek párhuzamos környezetben? Ennek csak akkor van értelme, ha a nyelv támogat párhuzamos végrehajtást.

8.2. A kivételkezelés kezdetei

8.2.1. Egy utasítás kivételkezelése: FORTRAN

Az 1950-es években kifejlesztett FORTRAN nyelvet [Lőcs 77] tekintjük az első magasszintű programozási nyelvnek. A nyelvet elsősorban tudományos számítások céljára tervezték. Már ez a nyelv is nyújt eszközöket a futási időben fellépő hibák kezelésére, bár ezek csak a be- és kimeneti (input-output) műveletek közben fellépő hibákra vonatkoznak. Az ilyen utasítások ugyanis kaphatnak egy `err` paramétert, ami a hibakezelő utasítás címkéjét (*label*) tartalmazza:

```
      open(file='proba.file', err=100, unit=2)
c     Folytatódik a program
c     ...
100   write(*,110)
110   format('Hiba!')
```

Mint látható, ez még nem igazi kivételkezelés, egyszerűen csak a nyelv tervezői felkészítettek bizonyos utasításokat egy elegáns hibakezelésre.

8.2.2. Több utasítás kivételkezelése: COBOL

A COBOL nyelvet [Bak 74] szintén az 1950-es években fejlesztették ki, nem sokkal a FORTRAN után. A FORTRAN-nal ellentétben a COBOL-t elsősorban üzleti szoftverek készítésére szánták. A hibakezelés terén a COBOL előrelépett a FORTRAN-hoz képest. A nyelv kétféle eszközt is tartalmaz a hibák kezelésére: jellemzően aritmetikai utasítások után meg lehet adni, hogy milyen címkén folytatódjon a program futása hiba esetén, illetve a be- és kimeneti hibák kezelésére is ad egy általánosabb eszközt a nyelv.

Az első hibakezelésre a következő példát mutatjuk:

```
DIVIDE Num1 BY Num2 GIVING Num3 REMAINDER Num4
  ON SIZE ERROR DISPLAY "Hiba".
```

A fenti utasítás elosztja `Num1`-et `Num2`-vel, a hányadost a `Num3` változóba teszi, a maradékot pedig `Num4`-be. Ha hiba történik (például a `Num2` értéke 0 vagy a hányados, illetve a maradék nem fér bele `Num3`-ba, illetve `Num4`-be), akkor a „Hiba" üzenetet írja ki a program. Ez ugyanolyan eseti jellegű hibakezelés, mint a FORTRAN-os. A be- és kimenettel kapcsolatos hibák kezelésére azonban egy általánosabb megoldást nyújt a nyelv. Az eljárásokat tartalmazó főrészben (`PROCEDURE DIVISION`) a `DECLARATIVES` és *`END`* `DECLARATIVES` kulcsszavak között minden, a programban használt fájlhoz rendelhetünk egy hibakezelőt:

```
PROCEDURE DIVISION.
DECLARATIVES.
Hiba SECTION.
    USE AFTER EXCEPTION PROCEDURE ON SajatFile.
Hibakezeles2.
    DISPLAY "Hiba".
END DECLARATIVES.
```

Ez már lényegesen fejlettebb megoldás, mint amit a FORTRAN-nál láthattunk, hiszen nem kell minden be- és kimeneti utasításnál egyenként megadni a hibakezelőt, elég egy helyen megadni (ez azzal az előnnyel is jár, hogy a programozó nem fogja elfelejteni vagy elrontani a hibakezelő megadását).

Még ez sem nevezhető azonban általános kivételkezelésnek, hiszen csak a be- és kimeneti műveletek során fellépő hibákat kezeli, nincs lehetőség saját kivételek használatára.

8.2.3. Dinamikus kivételkezelés: PL/I

A PL/I [Rák 74] volt az első programozási nyelv, ahol megjelent az általános kivételkezelés (már a hatvanas években), ennek megfelelően eléggé kezdetleges mechanizmus áll a programozók rendelkezésére.

A PL/I-ben a kivételeknek nevük van (mint egy címkének), így nem lehet se csoportosítani, se paraméterezni ezeket. A nyelvbe beépítettek néhány kivételt, saját kivételeket

pedig a CONDITION kulcsszó segítségével készíthetünk. Az ON utasítással lehet megadni, milyen utasítás hajtódjon végre, ha kivétel keletkezik az ON utasítás lefutása után. Kivételeket a SIGNAL utasítással lehet dobni:

```
ON CONDITION(SAJATKIVETEL) PUT LIST('A');
SIGNAL CONDITION(SAJATKIVETEL);
```

Amikor egy utasítás kivételt dob, akkor az éppen érvényben lévő kivételkezelőt hívja meg, így a kivételkezelés utasításhoz kötődik. A nyelv érdekes tulajdonsága, hogy egy-egy utasításra le lehet tiltani vagy engedélyezni lehet egyes kivételeket. A kivételkezelőt felül lehet bírálni, a REVERT utasítással pedig vissza lehet állni az előző kivételkezelőre:

```
P: PROC;
  /* ... */
  ON ZERODIVIDE PUT LIST('A');
  N=0;
  X=X/N;              /* 'A'-t ír ki. */
  BEGIN
    ON ZERODIVIDE PUT LIST('B');
    X=X/N;            /* 'B'-t ír ki. */
    REVERT ZERODIVIDE;
    X=X/N;            /* 'A'-t ír ki. */
  END;
  ON ZERODIVIDE PUT LIST('C');
  X=X/N;              /* 'C'-t ír ki. */
  (NOZERODIVIDE):
  X=X/N;              /* Nem keletkezik kivétel. */
END;
```

A kivételkezelő lefutása után visszakerül a vezérlés a kivételt dobó utasítás utáni utasításra, hacsak nem volt egy GO TO a kivételkezelőben, ami kiugrott a kivételkezelőből. A kivétel keletkezése és kezelése után a program a kivételt dobó utasítás után folytatódik, ezért nincs értelme a kivétel terjedéséről beszélni és éppen ezért a kivételek dobását sem lehet specifikálni.

Mint említettük, kivétel keletkezése esetén mindig az éppen érvényben lévő kivételkezelő fut le. Egy kivétel keletkezésekor az a kivételkezelő van érvényben, amely kivételkezelőt létrehozó ON utasítás legutoljára futott le:

```
  ON ZERODIVIDE PUT LIST('A');
  N=0;
  IF X = 10 THEN
    GOTO LABEL1;
  ON ZERODIVIDE PUT LIST('B');
LABEL1:
  X=X/N;
```

A fenti programrészlet lefutása során ha X értéke 10 volt, akkor a program 'A'-t fog kiírni, nem pedig 'B'-t. A kivételkezelőnek ez a dinamikus kiválasztása meglehetősen nehézzé teszi a program olvasását.

8.3. Fejlett kivételkezelés

8.3.1. Statikus kivételkezelés: CLU

A CLU nyelvet [Lis 81] jóval a PL/I elkészülte után fejlesztették ki, így többek között a kivételkezelése is jóval fejlettebb, bár az újabb nyelvekhez képest kissé szokatlan.

A CLU-ban a kivételek nem objektumok, hanem kifejezések, így nem csoportosíthatjuk a kivételeket. A kivételek elkapására szolgáló `except` ágakban azokat a konkrét kifejezéseket kell megadni, amelyeket el szeretnénk kapni, vagy pedig az `others` kulcsszóval jelölt ágban lehet elkapni az összes kivételt. A nyelvben kétféle kivétel van, amelyeket a `signal`, illetve az `exit` utasításokkal lehet kiváltani. Az első kivételfajta alprogramokhoz kapcsolódik, míg a második utasításokhoz. A `signal` utasítással eldobott kivétel befejezi az alprogram működését és visszatér a hívóhoz. Az `exit` utasítással eldobott kivételt ezzel szemben még abban az alprogramban kell kezelni, ahol eldobtuk (kell lennie egy olyan kivételkezelőnek az alprogramban valahol a kivételt kiváltó utasítás után, ami elkapja ezt a kivételt). Ebben az esetben a program futása a kivételkezelő után folytatódik. Mint láthatjuk, a PL/I-gyel ellentétben a CLU-ban fordítási időben lehet eldönteni, hogy a program adott pontját keletkező kivételt melyik kivételkezelő kapja el.

Ha egy alprogramban egy olyan kivétel lép fel, amelyet az alprogram nem kezel, akkor `failure` kivétel keletkezik, tehát az újabb nyelvekkel ellentétben a CLU-ban *nem* terjednek tovább a kivételek. Ennek az az oka, hogy így nem kell ismerni az alprogram megvalósítását (azt, hogy milyen további alprogramokat hív meg és ezek milyen kivételeket dobhatnak) ahhoz, hogy tudjuk, milyen kivételek keletkezhetnek még benne (lásd [Lis 92]). Persze azt megtehetjük, hogy a továbbadni kívánt kivételt elkapjuk a kivételkezelőben, majd újra eldobjuk.

A `signal` utasítással kiváltott kivételeket fel kell sorolni kell az alprogram fejlécében a `signals` kulcsszó után, kivéve a beépített `failure` kivételt.

Mivel a kivétel egy kifejezés, paramétert is átadhatunk egy kivételnek, például így:

```
P1=proc() signals(Kivetel(string))
  signal Kivetel("Baj van!")
end P1

P2=proc()
  P1()
  except
    when Kivetel(s:string)
    % Itt s értéke "Baj van!".
  end
end P2
```

A CLU nem ad olyan nyelvi szerkezetet, amely mind kivételes, mind kivétel nélküli esetben lefut (más nyelvekben ezt *finally* blokknak hívják). Mivel `signal` utasítással kiváltott kivétel befejezi az alprogram futását és csak a hívóban lehet kezelni, nem is tudunk ilyen szerkezetet csinálni.

8.3.2. Kivételek terjedése: Ada 95

Az Ada nyelv [Don 80, Ada 83] első változatát 1983-ban szabványosították, majd ezt a szabványt 1995-re felülvizsgálták, és több új elemet is bevezettek a nyelvbe. Kivételkezelés már az Ada 83-ban is volt, de a felülvizsgálat fontos újdonságokat vezetett be ezen a téren is.

Az Ada 95-ben a kivételeket az **exception** nevű nyelvi elemmel ábrázoljuk. Négy beépített kivétel van, a *CONSTRAINT_ERROR* (akkor keletkezik, ha egy megszorítás, például egy típus értékkészlete sérül), a *PROGRAM_ERROR* (ha valami megsérti a program vezérlési szerkezeteit, például egy függvény visszaadott érték nélkül akar visszatérni), a *STORAGE_ERROR* (ha elfogy a memória) és a *TASKING_ERROR* (taszkokkal kapcsolatos hibák). Ebből következően az Ada 95 nem ad lehetőséget a kivételek csoportosítására, nem lehet hierarchiába szervezni ezeket. Kivételeket a **raise** ⟨*kivételnév*⟩ utasítással vagy pedig a **Exception** csomag *Raise_Exception* alprogramjával lehet dobni.

Az Ada 95-ben a kivételkezelés blokkhoz kapcsolódik. Bármely blokkban lehet kivételt kiváltani, ezeket a blokk vége után írt **exception** kulcsszó utáni **when** ágakban lehet elkapni. Kivételkezelőt külön blokk definiálása nélkül egy alprogram törzse, egy csomag (*package*) törzse, egy taszk törzse és egy **accept** blokk után tehetünk. A kivételkezelő **when** ágaiban meg kell adni, hogy milyen típusú kivétel kezelésére képes (lehet többféle is, az **others** kulcsszóval rendelkező ág pedig a CLU-hoz hasonlóan az összes addig el nem kapott kivételt elkapja), valamint egy azonosítót is meg lehet adni. Ez utóbbi egy *Exception_Occurence* típusú változó, ami felveszi a dobott kivétel értékét. Ez utóbbi típus műveleteivel információt lehet szerezni a kivételről (például meg lehet tudni a paramétereit). A kivétel kezelése után a vezérlés nem kerül vissza abba a blokkba, ahol a kivétel keletkezett, a nyelv nem ad külön eszközt arra, hogy újra próbálkozzunk a kivételt dobó blokk végrehajtásával.

Az Ada 95 jellegzetessége, hogy az alprogramok, blokkok, csomagok és taszkok deklarációs része részben futási időben értékelődik ki, így ekkor is keletkezhet kivétel. Az első három esetben a kivétel ugyanúgy terjed felfelé a hívási láncon, mintha az alprogram, a blokk vagy a csomag kódja váltotta volna ki. Ha egy taszk deklarációs részében keletkezik kivétel, akkor a taszk komplett állapotba (részletekért lásd a 14.2.4. pontot) kerül, a taszk aktiválási pontján pedig a beépített *TASKING_ERROR* lép fel.

Ha blokkban keletkezik kivétel, és azt nem kezeli a blokkhoz tartozó kivételkezelő, akkor a kivétel változatlanul továbbdobódik, azaz egy szinttel feljebb újra jelentkezik ugyanaz a kivétel (ez újdonság a CLU-hoz képest). Ha a kivétel eljut a főprogramig, és az sem kezeli, a program futása leáll. Ha egy taszkban keletkezik a kivétel, és nem kezeli semmi, akkor a taszk komplett állapotba kerül (befejeződik a futása), de a kivétel nem terjed át a taszkot indító másik taszkra vagy főprogramra.

A kivételkezelő is dobhat kivételt. Ez hasznos olyankor, ha például tudjuk, hogy az elkapott *CONSTRAINT_ERROR* indexelési hibát jelez, ekkor a **when** *CONSTRAINT_*

ERROR ágban lehet dobni egy saját, indexelési hibát jelző kivételt. Az elkapott kivételt újra el lehet dobni a paraméter nélküli **raise** utasítással. Ennek a **when others** ágban lehet értelme, ahová akkor jut a vezérlés, ha egyáltalán nem várt kivétel jelentkezik. Ekkor ez az ág menti, ami menthető, majd újra kiváltja az elkapott kivételt.

Az Ada 83-ban még nem lehetett paraméterezni a kivételeket, az Ada 95 azonban bevezette az *Exception_Occurence* típust és az ehhez kapcsolódó alprogramokat. Ezek közül a *Raise_Exception* függvénnyel lehet a kivételeknek paramétert átadni, amit aztán a kivételkezelőben le lehet kérdezni.

Az Ada 95 nem ad külön nyelvi eszközt a *finally*-ra, de blokkok egymásba ágyazásával el lehet érni ezt (a külső blokk végére kell tenni azt, aminek mindenképpen le kell futnia). Egy példa erre:

```
procedure P is
  occ:Exception_Occurrence;
  volt_kivetel:boolean:=false;
  begin
  -- Erőforrások lefoglalása.
    begin
    -- Utasítások végrehajtása.
    exception
    when Exception1 | Exception2 =>
      -- Ezekre a kivételekre számítok, kezelem is ezeket.
    when e:others =>
      -- A többi kivételt továbbdobom. Először elmentem,
      -- majd a külső blokkban az erőforrások
      -- felszabadítása után újra eldobom.
      Save_Occurrence(occ,e);
      volt_kivetel:=true;
    end;
  -- Az erőforrások elengedése. A Javában ez lenne a finally blokk.
  if volt_kivetel then
    -- Újra eldobom az elmentett kivételt.
    Reraise_Occurrence(occ);
  end if;
end P;
```

Nem túl elegáns megoldás, hiszen a programozónak kell törődnie a kivétel elmentésével és újra eldobásával. Meg kell jegyeznünk, hogy az Ada 83-ban még nem volt lehetőség erre a trükkre sem.

Az Ada 95-ben nem lehet megadni, hogy melyik alprogram milyen kivételt dob. Ez azt jelenti, hogy ha a programozó elfelejtette valamelyik kivételét kezelni, az csak futási időben derül ki, amikor valami eldobja ezt a kivételt, és ettől leáll a program. Ez egy elég nagy hátrány, hiszen így csak a dokumentációból derülhet ki, ha egy programkönyvtár valamelyik alprogramja kivételt dob, a fordító nem tud figyelmeztetni erre.

8.3.3. Kivételosztályok megjelenése: C++

A C++ nyelvet [Str 00, Str 00] az Adához hasonlóan a nyolcvanas évek elején kezdték el fejleszteni, és ez a fejlesztés a mai napig tart. A nyelv legutolsó szabványa 1999-ben készült el. A nyelv kivételkezelése is fejlődött közben, részben az Ada és a CLU hatására.

A C++-ban bármilyen típusú változót el lehet dobni kivételként. Ez azt jelenti, hogy saját kivételeinknek külön osztályokat definiálhatunk, csoportosíthatjuk és hierarchiába szervezhetjük ezeket. A C++ szabványos programkönyvtárai például csak olyan kivételeket dobnak, amelyek típusa leszármazottja az *std::exception* osztálynak. A kivétel dobására a **throw** utasítás szolgál. Mivel a kivétel objektum, lehet paraméterezni úgy, hogy a kivételobjektum adatainak értéket adunk. Ha a kivétel egy egyszerű típus, akkor az értékét is fel lehet fogni a kivétel paramétereként.

A kivételkezelés a C++-ban is blokkokhoz kapcsolódik. A **try** blokkban dobott kivételt az utána következő **catch** blokkok kaphatják el. A kivételkezelés így néz ki:

```
try {
    // Kivétel keletkezhet.
}
catch (típus [név]) {
    // Kivétel kezelése.
}
```

Egy **try** blokkhoz több **catch** blokk is tartozhat, mindegyik csak a saját kivételtípusát kapja el (a típusegyezést a C++ szabályai szerint kell érteni). Ha a **catch** blokkban meg van adva egy név is, akkor az a változó felveszi a kivételként dobott objektum értékét. Ha a **catch** blokkban . . .-ot adunk meg típus és változó helyett, a **catch** az összes kivételt elkapja:

```
try {
    // Mindenféle kivétel keletkezhet.
}
catch (. . .) {
    // Minden kivételt elkap.
}
```

A kivétel kezelése után a vezérlés nem kerül vissza abba a blokkba, ahol a kivétel keletkezett, így nem lehet megpróbálni újra végrehajtani a kivételt dobó programrészt. Ez azzal is jár, hogy a **try** blokkban létrehozott lokális objektumok destruktorai meghívódnak még a kivételkezelő indulása előtt. Ennek következménye, hogy alaposan meg kell gondolnunk, ha egy destruktorban kivételt akarunk dobni, hiszen lehet, hogy amikor a destruktor lefut és kivétel keletkezik benne, egy kivétel már „eldobott" állapotban van. Ilyenkor ugyanis leáll a program, mert egyszerre nem lehet két kivétel „eldobva".

A kezeletlen kivétel továbbdobódik a hívó függvény felé. Ha a *main* függvény (a főprogram) sem kezeli a kivételt, akkor meghívódik a *terminate* függvény, ami alapesetben az *abort* függvényt hívja, ami leállítja a program futását. A *terminate* függvényt felül lehet definiálni a *set_terminate* függvénnyel, de a felüldefiniált függvénynek is be kell fejeznie a program futását (pontosabban: nincs definiálva a program működése, ha mégis visszatér a

függvény). POSIX szálakat (*thread*) használva az elindított szálban dobott, és nem kezelt kivétel csak a szál működését állítja le, az egész programét nem, ugyanis a kivétel nem jut vissza az indító szálhoz. Azonban vannak más, párhuzamosságot támogató programkönyvtárak is a C++-hoz (például a PVM), ezekben másképp történhet a kivételek kezelése. A keletkezett kivételt újra el lehet dobni a paraméter nélküli **throw** utasítással.

A C++ nem ad külön nyelvi eszközt a *finally*-ra. Az Ada 95-nél bemutatott blokkok egymásbaágyazása C++-nál nem elég, mert nincs „kivétel-előfordulás" típus, így nem tudjuk egyszerűen elmenteni a kivételt, hogy később újra eldobjuk. Egy megoldás erre a problémára az, hogy a mindig lefuttatni kívánt kódot egy lokális változó destruktorába tesszük. Ha a függvény valamilyen módon véget ér, akkor megszűnnek a lokális objektumai, és így meghívódnak ezek destruktorai is. Egy példa erre:

```
class File {
    FILE* f;
public:
    // Az egyszerűség kedvéért tekintsünk el a hibakezeléstől.
    File(char* filenev) { f = fopen(filenev); };
    ~File() { fclose(f); };
};
void fv()
{
    File f("valami.txt");
    try {
        // Itt keletkezhet kivétel
    }
    catch(int i) {
        // Kezelek egy fajta kivételt
    }
}   // Itt mindenképpen lezárjuk a fájlt!
```

A C++ lehetőséget ad arra, hogy megadjuk azt, hogy egy függvény kivételek mely részhalmazát dobhatja: a függvény fejlécében a **throw** kulcsszó után zárójelek között sorolhatjuk fel ezeket. Ha nem teszünk ilyet, akkor mindenféle kivételt dobhat (C-vel kompatibilitási okokból), viszont ha igen, akkor csak ezeket dobhatja. Ha olyan kivételt dob a függvény, amelyet a programozó nem sorolt fel, akkor az *unexpected* függvény hívódik meg, ami leállítja a programot. Ezt a függvényt is felül lehet definiálni (a *set_unexpected* függvénnyel), de akkor is be kell fejeznie a program futását, nem térhet vissza. Meg lehet adni üres halmazt is, ekkor a függvény semmilyen kivételt nem dobhat. Sajnos a fordító nem ellenőrzi, hogy a felsorolt kivételeket a hívó kezeli-e vagy sem.

8.3.4. Helyességbizonyítás és kivételkezelés: Eiffel

Az Eiffel nyelvet [Mey 91] a nyolcvanas évek második felében fejlesztették ki a „Design by Contract" elv köré. Ez az egyik oka, hogy a nyelv kivételkezelése jelentősen különbözik kortársaitól.

Az Eiffelben a kivételek nem objektumai a nyelvnek. Ugyan van nyelvi elem arra, hogy kivételt elkapjunk (**rescue** záradék), de kivételt dobni, illetve az elkapott kivétel tulajdonságait lekérdezni csak úgy lehet, hogy a kivételt dobó, illetve kezelő metódusok osztályai örökölnek az *EXCEPTIONS* osztálytól. Ez azért van így, mert a nyelv tervezői szerint jellemzően nincsenek olyan műveletek, amelyek egy kivétel objektum állapotát meg tudnák változtatni, így felesleges objektumokat használni kivételekként. A kivételeket a kivételkód alapján lehet megkülönböztetni, ezeket az *EXCEPTIONS* osztályból örökölt *exception* és *developer_exception_code* (programozó által definiált kivétel esetén) metódus meghívásával lehet megkapni. Ez a két kód egész szám, ez alapján lehet esetleg a saját kivételeket csoportosítani (például a 10 és 15 közötti kódú kivételek hálózati hibát jeleznek), de a nyelv előredefiniált kivételeit nem lehet így csoportosítani.

Az Eiffel kivételkezelésének filozófiája különbözik az eddig megismertektől: kivételnek azt tekinti, ha valami okból sérül a „szerződés", amit az alprogram hívója és írója kötött, ezért a kivételkezelés az alprogramhoz kapcsolódik. Eiffelben akkor keletkezik kivétel, ha sérül a „szerződés", azaz valamelyik megszorítás (előfeltétel, ciklusinvariáns, stb.) hamis lesz, vagy ha *Void* értékű változón akar olyan műveletet végezni a program, aminek nincs értelme, ha az operációs rendszer jelez valamilyen hibát, vagy ha egy meghívott alprogram sikertelenül fejeződik be vagy a *raise* metódushívással dob egy kivételt a programozó.

A kivételkezelés alprogramhoz kötődik, helye az alprogram törzse után megadott **rescue** záradék, feladata pedig az, hogy megpróbálja újra végrehajtani a feladatot (és így teljesíteni a szerződést), vagy pedig helyreállítani az osztály invariánsát és jelezni a hívó felé, hogy nem sikerült a végrehajtás (ez a szervezett pánik). Ilyenkor sikertelenül fejeződik be az alprogram végrehajtása (*failure*) és kivétel keletkezik a hívóban.

```
fv: INTEGER is
  do
    -- Itt keletkezhet kivétel.
  rescue
    -- Itt kell kezelni a kivételt.
  end;
```

Az Eiffel kivételkezelésének jellegzetessége az, hogy lehetőséget ad arra, hogy ha egy alprogramban kivétel lép fel, akkor lehessen újra próbálkozni a végrehajtással (például az alprogram egy egész számot szeretne beolvasni a felhasználótól, és ha nem számot kap, akkor próbálkozzon újra). Erre szolgál a **retry** utasítás és az, hogy a második (sokadik) próbálkozáskor nem kapják meg újra a kezdőértéküket a lokális változók. Fontos megjegyezni, hogy Eiffelben a kivétel csak akkor „tűnik" el, ha sikerült újra végrehajtani az alprogramot. Ha a **rescue** záradék végrehajtása nem **retry**-jal fejeződik be, a hívóban mindenképp kivétel keletkezik, ugyanis nem sikerült végrehajtani az alprogramot. Fontos megjegyeznünk, hogy az alprogram utófeltételének teljesítése az alprogram feladata, nem pedig a **rescue** záradéké.

Az Eiffelben a kivételek terjedése inkább a CLU-ra hasonlít, mint a kortárs nyelvekre (C++, Ada). Minden osztálynak van egy *default_rescue* metódusa. Ezt az *ANY* osztályban (minden osztály ebből származik) definiálták, és ott üres a törzse. Ez az a metódus, ami akkor hívódik meg, ha egy alprogramban kivétel keletkezik, és az alprogramhoz nem tartozik

rescue záradék, így az Eiffelben nincs kezeletlen kivétel. Az alprogramot tartalmazó osztály felüldefiniálhatja ezt a metódust (például azért, hogy ez a metódus állítsa helyre az osztályinvariánst). Ellenben ha egy **rescue** záradéknak nem sikerül második (sokadik) próbálkozásra sem végrehajtania az alprogramot, és befejeződik a futása, akkor a hívóban keletkezik kivétel, hiszen az alprogram végrehajtása sikertelen volt. Ilyenkor ugyan lehet, hogy egy más kóddal rendelkező kivétel váltódik ki, de ez megőrzi az eredeti kivétel kódját is. Ha egy kivétel feljut egészen a főprogramig, és az sem kezeli, akkor leáll az egész program, és kiírja, hogy milyen kivételek vezettek a program leállásához.

Kivételt dobni az *EXCEPTIONS* osztályból örökölt *raise* metódussal lehet. Ennek meg kell adni a kivétel kódját és egy üzenetet (ez utóbbit írja ki a rendszer, ha leáll a program egy kezeletlen kivételtől). Ezen kívül egy környezetet (*context*-et) is lehet adni a kivételnek, ezzel lehet igazából paraméterezni a kivételt.

Mivel a kivételkezelés alprogramhoz kötődik, nincs lehetőség az Ada 95-nél bemutatott „trükközésre" a *finally* szimulálásához. A C++-os megoldás azért nem használható, mert az Eiffelben automatikus szemétgyűjtés van és nem lehet destruktort írni. A *finally*-t általában arra használják, hogy helyreállítsák az osztályinvariánst: erre az Eiffelben a **rescue** záradék szolgál. Ha az osztályinvariáns például előírja, hogy egy bizonyos fájlnak minden metódus lefutása után lezártnak kell lennie, akkor ezt a fájlt mind az alprogram végén, mind a kivételkezelő részben le kell zárni. Az ilyen műveleteket célszerű a *default_rescue* metódusba tenni, és ha egy alprogramhoz tartozik **rescue** záradék, akkor onnan meg lehet hívni ezt a metódust (ha nem tartozik, akkor automatikusan a *default_rescue* metódus hívódik meg). Persze az osztályinvariánst magában az alprogramban is meg kell tartani, nem elég, ha csak a kivételes esetekben törődünk ezzel. Ez megtörténhet a *default_rescue* metódus meghívásával, és ekkor nem kell kódot ismételnünk.

Az Eiffelben a kivételek tényleg kivételes események, normális esetben nem fordulnak elő, így nem is lehet specifikálni őket. Mivel kivétel keletkezik akkor is, ha megsérül egy megszorítás (például előfeltétel), ezért jó megszorításokat, invariánsokat kell megadni (és nem kivételeket). Ezeket a megszorításokat és invariánsokat a program helyességének bizonyítása közben is fel lehet használni, részletesebben lásd a 13. fejezetben.

8.3.5. A *finally* blokk: Modula-3

A Modula-3 nyelv [Nel 91] a Pascal nyelvcsaládba tartozó Modula-2 továbbfejlesztéseként született meg a nyolcvanas évek második felében. Kivételkezelése az Ada kivételkezelésére hasonlít, de a C++-ból is vettek át elemeket.

A kivételeket a Modula-3 nyelv az Ada 95-höz hasonlóan az **EXCEPTION** nevű nyelvi elemmel ábrázolja, így nem lehet csoportosítani őket. Kivételeket a **RAISE** utasítással lehet dobni. A Modula-3 kivételkezelésének érdekessége, hogy a ciklus belsejéből kiugró **BREAK** utasítást úgy kezeli, mintha a program egy *ExitException* kivételt dobott volna, a ciklusokhoz pedig beépítve tartozik egy olyan kivételkezelő, ami elkapja ezt a kivételt. Hasonlóan az alprogramból kilépő **RETURN** egy *ReturnException*-t dob, ami paraméterként megkapja a visszaadni kívánt értéket. Ezt a kivételt az alprogramok beépített kivételkezelője kapja el.

A kivételkezelés a C++-hoz hasonlóan a **TRY** blokkhoz kapcsolódik, az ezen belül eldobott kivételeket a **TRY** blokkot követő **EXCEPT** blokkban lehet elkapni. Az **EXCEPT** blokk végén lehet egy **ELSE** ág, ami az összes, ebben az ágban még el nem kapott kivételt fogja el, mint az Adában. A kivétel kezelése után a program futása a kivételkezelő után folytatódik, nincs egyszerű lehetőség az újrapróbálkozásra. A kivételeket mindig el kell kapni, illetve meg kell adni azokat, amelyek továbbdobódhatnak. Ha olyan kivétel keletkezik, amelyet nem kezel az alprogram és nem is specifikál, akkor a program leáll, mint a C++-ban. A súlyos hibákat jelentő futási hibákat (*runtime error*) nem lehet elkapni.

A Modula-3 támogatja a többszálú programozást. Ha egy szálban olyan kivétel keletkezik, amelyet nem kezel a szál, akkor az egész program leáll, ellentétben az Ada 95-tel. A kivételeket eldobásukkor lehet paraméterezni, de legfeljebb egy paramétert kaphatnak.

A Modula-3-ban jelent meg a **TRY–FINALLY** blokk, ami úgy működik, hogy akár keletkezik kivétel a **TRY** blokkban, akár nem, mindenképpen végrehajtódnak a **FINALLY** blokk utasításai. Ez nagyon hasznos, ha például megnyitottunk egy fájlt, és mindenképpen le akarjuk zárni akkor is, ha kivétel keletkezett a fájl tartalmának feldolgozása közben.

8.3.6. Ellenőrzött kivételek: Java

A Java nyelvet [Nyé 01] a kilencvenes években fejlesztették ki. Nagyon sok elemet a C++-ból vettek át, de a kivételkezeléshez a Modula-3-ból is használtak fel elemeket, valamint komoly újításokat is vezettek be ezen a téren.

A Java nyelvben a kivételek objektumok. A nyelvben található összes kivételosztály a *java.lang.Throwable* osztály leszármazottja (csak ennek az osztálynak a leszármazottjait lehet eldobni). A *Throwable* osztálynak két közvetlen leszármazottja van, az *Error* és az *Exception* osztály. Az előbbi osztály leszármazottjai mind nem ellenőrzött kivételek, azaz nem kötelező elkapni őket (az *Error* leszármazottjait nem is „illik" elkapni, mert például az *OutOfMemoryError* kivétellel nem sok mindent tehet a program). Ezek a kivételek felelnek meg a Modula-3 futásidejű hibáinak. Az *Exception* osztály leszármazottjai közül a *RunTimeException* és ennek leszármazottjai szintén nem ellenőrzött kivételek, de ezek nem végzetes hibák, csak éppen túl sok helyen fordulhatnak elő, és nem lehet elvárni a programozótól, hogy állandóan figyeljen az *ArrayIndexOutOfBoundsException* és más hasonló kivételek fellépésére. Természetesen a nem ellenőrzött kivételeket is el lehet kapni. Ha mégse kapunk el egy nem ellenőrzött kivételt, akkor leáll a program (illetve ha a kivétel egy szálban keletkezett, akkor csak az a szál áll le, a kivétel nem terjed tovább az indító szálra). Az *Exception* többi leszármazottja viszont mind ellenőrzött kivétel, azaz ha egy metódus specifikálja, hogy dobhat egy ilyen kivételt, akkor azt el *kell* kapni, vagy pedig specifikálni kell, hogy a hívott metódus továbbdobja ezt a kivételt. Ezt még a fordító ellenőrzi, nem csak futási időben derül ki (ha kiderül egyáltalán), hogy egy kivételt elfelejtettünk elkapni. Ez nagyon erős eszköz, amely a programozót arra kényszeríti, hogy igenis foglalkozzon például egy URL objektum elkészítésénél azzal, hogy esetleg rossz értékkel hívta meg a konstruktort. A kivételeket a C++-hoz hasonlóan a **throw** utasítással lehet eldobni.

A Javában (a C++-hoz hasonlóan) blokkhoz kapcsolódik a kivételkezelés. Ugyanúgy egy **try** blokkba kell beágyazni az esetlegesen kivételt dobó utasítást, és hasonlóan **catch** blokkokkal lehet elkapni a kivételeket. Annyi különbség van, hogy a Javában kötelező megadni egy paramétert, ami kivétel esetén felveszi a kivételobjektum értékét. Javában a C++-szal ellentétben nincs lehetőség arra, hogy . . .-ot írjunk a **catch** ágba, amivel az el tudná kapni az összes kivételt. Helyette az öröklődési hierarchiát kihasználva a **catch** *(Exception e)* {} ágat lehet használni.

Hasonlóan a C++-hoz, a vezérlés a Javában sem kerül vissza a kivételt kiváltó blokkba a kivétel dobása után, nincs egyszerű lehetőség az Eiffelben megismert újrapróbálkozásra (a Javában még **goto** sincs, amivel a C++-ban még lehetne „ügyeskedni").

Javában minden **try** blokkot követhet az esetleges **catch** blokkok után egy (Modula-3-ból ismerős) **finally** blokk. A **finally** blokk mindenképpen végrehajtódik, akár történt kivétel, akár nem. Akkor is végrehajtódik, ha olyan kivétel keletkezett, amit egyik **catch** blokk sem kezelt.

8.3.7. A Delphi kivételkezelése

A Delphi nyelv [Lis 00] a kilencvenes években született meg a Turbo Pascal nyelv egyfajta objektumorientált továbbfejlesztéseként. A kivételkezelése leginkább a „rokon" Modula-3 nyelv kivételkezelésére hasonlít.

A Javához hasonlóan a kivételek Delphiben is objektumok, méghozzá Exception osztályúak (vagy az Exception egy leszármazottjából példányosodtak), így az osztályhierarchia alapján lehet csoportosítani őket. Ebben a nyelvben is blokkhoz kötődik a kivételkezelés. A **try** blokkban keletkezett kivételeket az ezt követő **except** blokkban lehet kezelni. Az **except** blokk kezeli az összes kivételt **on** ágakban. Egy ilyen **on** ágban a C++-hoz hasonlóan meg lehet adni csak típusnevet vagy azonosítót is (az utóbbi esetben a megadott azonosító felveszi a kivételobjektum értékét). Az **except** blokk végén lehet egy **else** ág is, ez akkor fut le, ha olyan kivétel keletkezett, amit az előző ágak nem kaptak el (a hatása egyenértékű az **on** *Exception* ággal). A kivétel kezelése után a program a kivételkezelő után folytatódik, nincs lehetőség az Eiffelben megismert **retry**-ra.

A nem kezelt kivétel a hívó alprogramban újra kiváltódik, így terjed felfelé a hívási láncon. A kivételkezelőben is lehet dobni kivételt, valamint az éppen kezelt kivételt is újra el lehet dobni a paraméter nélküli **raise** utasítással. Ha egy kivételt egyáltalán nem kapunk el, akkor leáll a program. Ha egy többszálú programban az egyik szálban kezeletlen kivétel keletkezik, akkor a Javához és az Ada 95-höz hasonlóan csak ez a szál áll le, az egész program megy tovább. A Modula-3-mal és a C++-szal ellentétben a Delphiben nem lehet megadni, hogy milyen kivételeket dobhat egy metódus.

A **try** blokkhoz nem csak **except**, hanem **finally** blokk is tartozhat (egyszerre csak az egyik), mint a Modula-3-ban. Ez utóbbi ugyanazt a cél szolgálja: akár keletkezik kivétel

a **try** blokkban, akár nem, mindenképpen lefut. A Java **try–catch–finally** szerkezetét két egymásba ágyazott **try**-jal lehet elérni:

```
try
try
  ... { itt jönnek az utasítások }
except
  on e: Exception do
    ... { a kivétel kezelése }
end
finally
  ... { ez mindenképpen lefut }
end
```

8.3.8. Egymásba ágyazott kivételek: C#

A C# nyelvet [Sch 02] a kilencvenes évek végén fejlesztették ki egyfajta válaszként a Java nyelvre. Ennek megfelelően a kivételkezelése nagyon hasonlít mind a C++-éra, mind a Javáéra.

A Javához hasonlóan a C#-ban is a kivételek objektumok. Minden kivételobjektum osztályának a *System.Exception* osztály leszármazottjának kell lennie. Ennek az osztálynak a jellegzetessége, hogy van egy *InnerException* nevű attribútuma, ami azt a kivételt tartalmazza, ami az aktuális kivételt váltotta ki (vagy **null**-t, ha nincs ilyen). Például ha egy statikus objektum konstruktorának futása közben keletkezik egy kivétel, és maga a konstruktor nem kapja el ezt a kivételt, akkor a konstruktor hívásának helyén egy *System.TypeInitializationException* kivétel keletkezik, és ennek az *InnerException* mezője tartalmazza az eredeti kivételt, ami a konstruktorban keletkezett. Ekképpen tetszőleges számú kivételobjektumot lehet összefűzni. Szintén nagyon hasznos a *StackTrace*, illetve a *TargetSite* attribútum. Az első azt tartalmazza szöveges formában, hogy a kivétel keletkezésekor milyen metódusok voltak meghívva, míg az utóbbi a kivételt dobó metódusra mutat.

Kivételt a **throw** utasítással lehet dobni, elkapni pedig a C++-hoz hasonlóan **try** és **catch** blokkokkal lehet. A C# nyelvben is van **finally** blokk, amely ugyanúgy működik, mint Javában. Annyiban fejlettebb a C# elődeinél, hogy a fordító hibát jelez, ha a kivételkezelő **catch** ágak megadásakor az ős-kivételosztályt kezelő ág megelőzi a leszármazott-kivételosztályt kezelő **catch** ágat.

8.3.9. Kivételkezelés függvényekkel: Common Lisp

A Lisp nyelv első változata még az 50-es években jelent meg. A nyelvet azóta is folyamatosan fejlesztik, így az 1994-ben szabványosított Common Lisp nyelv [Ste 90] már kivételkezeléssel is rendelkezik. Lispben „minden függvény", ezért természetesen a kivételek kezelését is két külön függvény segítségével lehet megoldani, nincsenek külön kivételkezelő nyelvi

elemek. Ennek ellenére érdemes megnézni ezt a megoldást is, mert filozófiája némileg különbözik az eddig megismert kivételkezelésektől.

A Lispben a kivételt egy atom ábrázolja (egy atomot lehet eldobni és elkapni), ezért a kivételeket nem lehet csoportosítani, egy kivételkezelő csak egyféle kivételt tud elkapni. A kivétel paraméterezhető, a kivételatom felveszi a paraméter értékét, ezt adja vissza aztán a kivételkezelő **catch** függvény. Lispben nem lehet megadni, hogy egy függvény milyen kivételt dobhat. Ha egy kivételt nem kapunk el, akkor továbbterjed a hívási láncon felfelé. Ha a főprogram sem kapja el a kivételt, akkor leáll a program.

A Lispben természetesen a kivételkezelést is függvényekkel valósították meg. A nyelv szintaktikájának érdekessége, hogy a kivételkezelést előbb kell megadni, mint a kivételdobást. A **catch** függvényen belül a **throw** függvénnyel lehet kivételt dobni:

```
(catch 'hibahiba
  (print "Eddig minden rendben")
  (throw 'hiba "Baj van!")
  (print "Ez mar nem fut le"))
```

A fenti függvényrészlet kiértékelése során a Lisp értelmező (*interpreter*) kiírja a „Eddig minden rendben van" feliratot, de az „Ez mar nem fut le" feliratot már nem. A `catch` függvény visszatérési értéke a „Baj van!" szöveg lesz. A kivétel keletkezése után a kivételkezelő után folytatódik a program futása, nincs egyszerű lehetőség az Eiffelben megismert `retry`-ra.

Ha egy függvény kivételt dob, és nem kezeli, akkor az a hívó függvényben jelentkezik, azaz továbbterjed. A nyelvben nincs kifejezetten kivételkezelő, a `catch` függvény utáni függvényhívásokkal lehet kezelni az esetlegesen keletkezett kivételt. Annak eldöntése, hogy egyáltalán keletkezett-e kivétel, nem is olyan egyszerű, hiszen csak abból indulhatunk ki, hogy milyen értéket adott vissza a `catch` függvény:

```
(setq kivetel_tortent
     (catch 'kivkiv
       (if (equal i 1)
          (throw 'kiv "i=1")
         nil)
       (eval "OK")))
(if (equal kivetel_tortent "OK")
   (print "Minden rendben")
  (progn
    (print "Kivétel történt, melynek oka:")
    (print kivetel_tortent)))
```

Ha nem keletkezik kivétel a `catch` függvényben, akkor a visszatérési értéke az utolsó függvény visszatérési értéke lesz (a fenti példában ez az „OK" szöveg). Kissé nehézkes megoldás, hiszen egy – a feladat megoldása szempontjából – felesleges utasítást kell beírnunk a kódba, pusztán technikai okokból.

8.3.10. Új megoldások: Perl

A Perl nyelvet [Sch 01] a nyolcvanas évek végén kezdték el fejleszteni. Ugyan kifejezetten kivételkezelő eszköz nincs a nyelvben, de az `eval` és a `die` operátorok segítségével lehet a Lisphez hasonló kivételkezelést végezni. Az `Error` programkönyvtár (a Perl szóhasználatban csomag (*package*)) is ezeket használja, hogy a programozó számára a C++-ból vagy a Javából ismerős formát nyújtson a kivételkezeléshez. Ez a megoldás nem szigorúan nyelvi eszköz (a nyelvnek nem része a kivételkezelés), de olyan egyedi ötletek fordulnak elő benne, amelyeket érdemes bemutatni. Az alábbiakban az `Error` csomag lehetőségeit mutatjuk be [Bar 01, Sha 02].

A kivételobjektumok osztályának az `Error` osztályból kell származnia. A Perl támogatja a többszörös öröklődést, ezért ez nem olyan nagy megkötés. Az `Error` osztály is az `Error` csomag része. Mivel ezek a kivételek objektumok, az osztályhierarchia alapján csoportosítani lehet őket. A kivétel dobására a `throw` függvény szolgál. A kivételeket az objektumok adatainak értéket adva paraméterezhetjük. A kivételek nem a nyelv részei, ezért nem lehet specifikálni a kivételek dobását.

A kivételkezelés blokkhoz kapcsolódik. A kivételkezelés úgy van megvalósítva, hogy az `Error` osztály osztályszintű `try` függvényét blokkokkal és záradékokkal (*clause*) paraméterezzük. A továbbiakban az egyszerűség kedvéért a `try` utáni blokkot `try` blokknak, a `catch` utáni blokkot `catch` blokknak, stb. nevezzük annak ellenére, hogy a nyelv szempontjából ezek nem kitüntetett blokkok.

```
use Error qw(:try);

@Error::Sajat::ISA = (Error);

sub kezelo {
  my $e = shift;
  my $folytasd = shift;
  print "Még egyszer elkaptam a kivételt: $e";
  $$folytasd=1;
}

try {
  print "Dobok egy kivételt!";
  throw Error::Simple("Baj van!");
}
catch Error::Simple with {
  my $e = shift;
  my $folytasd = shift;
  print "Kivételt kaptam: $e ";
  $$folytasd=1;
}
```

A program a következő oldalon folytatódik.

```
# Folytatás az előző oldalról.
except {
  return {
    Error::Simple => \&kezelo
  };
}
otherwise {
  my $e = shift;
  print "Erre nem gondoltam: $e ";
}
finally {
  print "Ez mindig lefut";
};
```

A `try` blokkokon belül dobott kivételeket tudjuk csak elkapni. A kivételeket többféle módon kezelhetjük. A `catch` blokkokban egy-egy kivételosztály példányait lehet elkapni, míg az `except` blokkban futási időben(!) – tehát például egy paraméter értékétől függően – lehet hozzárendelni egy kivételkezelő függvényt a kivételosztályhoz (egészen pontosan az `except` blokknak egy olyan hasítótábla-referenciát kell visszaadnia, ahol a kivételosztályok neve a kulcs, és az ezeket kezelő függvényre mutató referencia az érték). A `catch` blokk (illetve az `except` blokkban a kivételosztályhoz rendelt kivételkezelő függvény) első paramétereként a kivételobjektumot kapja meg (a Perlben nem a más nyelvekből megszokott formális paraméterek veszik fel az aktuális paraméterek értékeit, hanem a `@_` tömbben kapja meg az alprogram, illetve a blokk a paramétereit. A *`shift`* függvény paraméter nélkül ebből a listából veszi ki az első elemet és ezt adja vissza). Az `otherwise` blokk akkor hívódik meg, ha egyetlen `catch` és `except` sem kapja el a kivételt (a C++ `catch(...)`-jéhez hasonlóan). A Perl gyengén típusos nyelv, ezért ilyenkor is megkaphatjuk a kivételobjektumot egy skalár típusú változóban. A `finally` blokk mindenképpen lefut: ha nem keletkezett kivétel, akkor a `try` blokk lefutása után; ha kezeltük a kivételt, akkor a kezelése után; ha nem kezeltük a kivételt, akkor pedig a kivétel dobása után. Egyetlen záradékot sem kötelező megadni, de csak `catch`-ből és `except`-ből lehet több. Természetesen itt is fontos a kivételkezelők sorrendje, mert ha több kivételkezelő is el tudná kapni ugyanazt a kivételt, akkor csak az elsőként megadott hívódik meg.

A kivétel kezelése után a `try` utáni utasítástól folytatódik a program, kivéve, ha a `catch` blokkban vagy az `except` blokkban megadott kivételkezelő függvényben a második paraméterként megkapott skalárreferenciának értéket adunk (a fenti példában ez a `$folytasd` változó). Ekkor ugyanis úgy folytatódik a program futása, mintha nem kaptuk volna el a kivételt, és az ugyanehhez a `try`-hoz tartozó további kivételkezelők megpróbálhatják elkapni ezt a kivételt. Ez azt jelenti, hogy a kivételt nem egy felsőbb szintre, hanem a saját szintjére dobja tovább. Ilyen lehetőséget a korábban ismertetett programozási nyelvek nem nyújtanak.

Ha egy kivételt nem kezeltünk, akkor továbbdobódik, és külső `try` blokkokban kezelhetjük. Ha ezt nem tesszük meg, akkor leáll a program, kivéve ha egy *`eval`* operátoron belül történt a kivételdobás. Ez annak a „mellékhatása", hogy a kivételkezelés *`die`* és *`eval`* operátorokkal van megvalósítva. Természetesen a kivételkezelő is dobhat kivételt, illetve az elkapott kivételt újradobhatja a kivételobjektum `throw` metódusának meghívásával.

8.4. Összefoglaló

A kivételkezelés nagyon hasznos programozási eszköz, nem véletlen, hogy az újabb nyelvek mind adnak rá valamilyen megoldást. A legnagyobb előnye az, hogy jól elkülöníti a feladatot ténylegesen megoldó kódot a hibakezeléstől, így könnyebben átláthatóvá és módosíthatóvá teszi a programot, és nem okoz problémát, hogy a hibakezelő kód a hiba előfordulásától messze (a hívási láncon több lépéssel feljebb) van. Másik fontos előnye, hogy objektumorientált nyelveknél az objektumok konstruktorai természetüknél fogva nem tudnak hibakódot visszaadni, egy kivétel dobásával azonban egyszerűen értesíthetik a konstruktor meghívóját nemcsak arról, hogy nem sikerült létrehozni az új objektumot, hanem arról is, hogy miért nem (erre lehetne kimenő (*out*) paramétert is használni – lásd a 7.4. fejezetet –, de ez nem túl elegáns).

A kivételkezelésnek azonban vannak hátrányai is: kissé elrontja a program strukturáltságát, hiszen egyfajta **goto** utasításként működik, továbbá egyes nyelvekben (például C++) elég veszélyes lehet, ha olyankor dob kivételt a program, amikor nem szabadna, és a fordító nem figyelmeztet erre. Szintén nem szabad megfeledkeznünk arról, hogy a kivételkezelésnek ára van: bizonyos nyelvek fordítói (például C++) kénytelenek többletkódot generálni a kivételkezeléshez, amelyre nem lenne szükség, ha más módon végeznénk hibakezelést.

Érdemes megvizsgálni, hogy a tárgyalt nyelvek a programkönyvtáraikban milyen mértékben használnak kivételeket (ez jelzi, hogy a nyelv tervezői mennyire tartják kívánatosnak a kivételeket). A Javában gazdag kivételkészlet van, a kivételek szigorú hierarchiába szervezettek, így kénytelen a programozó használni őket, de ez nem olyan bonyolult. Ezzel szemben a C++ szabványos könyvtárai nem nagyon használják a kivételeket, részben azért, mert a sok használt C függvény miatt úgyis mindenhol hibakódokkal jelzik a hibákat. Az Ada 95-ben csak négy előre definiált kivétel van, míg a Delphiben egy külön **unit**-ot kell használni arra, hogy használni lehessen a kivételeket. Az Eiffelben a kivételeket elsősorban a sérült megszorítások adják, amelyeket rendszeresen használ a standard programkönyvtár (például ami a Javában *IndexOutOfBoundsException* lenne, az Eiffelben az indexelő metódus előfeltételének megsértése).

A tárgyalt nyelvek közül talán a Java és a C# nyelv kivételkezelése a legjobban kidolgozott, leginkább használható, ami nem meglepő, hiszen ez a két nyelv igen fiatal, és a tervezőik meg tudták vizsgálni a többi nyelvet ebből a szempontból is. A jövőben elkészülő nyelvek éppen ezért valószínűleg újabb megoldásokat és ötleteket fognak tartalmazni, mint azt a Perl nyelvhez tartozó kivételkezelő csomagnál láthatjuk.

8.5. Példaprogramok kivételkezelésre

8.5.1. C++

Az alábbi példaprogram egy kételemű vermet valósít meg. A veremnek két művelete van, a *push* egy elemet tesz a verem tetejére, a *pop* pedig egy elemet vesz le a verem tetejéről. Mindkét metódus kivételdobással jelzi, ha hiba történt (teli verembe szeretnénk elemet tenni vagy üres veremből szeretnénk elemet kivenni). Mindkét kivétel osztálya a *VeremException* osztályból származik, ezzel mutatunk példát a kivételek csoportosítására.

```
#include <iostream>
#include <string>

using namespace std;

class VeremException {
  protected:
    string hiba;
  public:
    VeremException(string _hiba): hiba(_hiba) {};
    string toString(void) {
      return hiba;
    };
};

class VeremTeliException: public VeremException {
  public:
    VeremTeliException(void):
        VeremException("Megtelt a verem!");
    }
};

class VeremUresException: public VeremException {
  public:
    VeremUresException(void):
        VeremException("Ures a verem!");
    }
};
```

```
class Verem {
  int v[2];
  int sp;

  public:
    Verem(void) {
      sp=0;
    };

    void push(int i) throw (VeremTeliException) {
      if (sp >= 2) throw VeremTeliException();
      v[sp] = i;
      sp++;
    };

    int pop(void) throw (VeremUresException) {
      if (sp <= 0) throw VeremUresException();
      sp--;
      return v[sp];
    };
};

int main(void)
{
  Verem v;

  try {
    v.push(1);
    v.push(2);
    v.push(3); // Itt VeremTeliException keletkezik
    v.pop();
    v.pop();
    v.pop();  // Itt VeremUresException keletkezik
  }
  catch(VeremException ve) {
    cout << "A kovetkezo problema tortent:" << ve.toString() << endl;
  };
  return 0;
}
```

8.5.2. Java

Az alábbi példaprogram ugyanazt a feladatot oldja meg, mint a 8.5.1. pontban található program: egy kételemű vermet valósít meg. A veremnek két művelete van, a *push* egy elemet tesz a verem tetejére, a *pop* pedig egy elemet vesz le a verem tetejéről. Mindkét metódus kivételdobással jelzi, ha hiba történt (teli verembe szeretnénk elemet tenni vagy üres veremből szeretnénk elemet kivenni). Mindkét kivétel osztálya a *VeremException* osztályból származik, ezzel mutatunk példát a kivételek csoportosítására. Érdemes megfigyelni, hogy míg a C++-os megoldásnál a program veszi észre a hibát (a műveletek elején megvizsgálja, hogy a verem állapota megfelelő), itt a JVM[4] veszi észre, hogy nem megfelelő a verem állapota, és ezt egy kivétellel jelzi.

```
class VeremException extends Exception {
  protected String hiba;

  public VeremException(String _hiba) {
    hiba=_hiba;
  }

  public String toString() {
    return hiba;
  }
}

class VeremTeliException extends VeremException {
  public VeremTeliException() {
    super("Megtelt a verem!");
  }
}

class VeremUresException extends VeremException {
  public VeremUresException() {
    super("Ures a verem!");
  }
}

class Verem {
  int[] v;
  int sp;

  public Verem() {
    v=new int[2];
    sp=0;
  }
```

[4] Java Virtual Machine: az a program, ami végrehajtja a Java programot

```
  public void push(int i) throws VeremTeliException {
    try {
      sp++;
      v[sp]=i;
    }
    catch (ArrayIndexOutOfBoundsException e) {
      sp--;
      throw new VeremTeliException();
    }
  }

  public int pop() throws VeremUresException {
    try {
      sp--;
      return v[sp];
    }
    catch (ArrayIndexOutOfBoundsException e) {
      sp++;
      throw new VeremUresException();
    }
  }
}

public class VeremM {

  public static void main(String args[]) {
    Verem v=new Verem();
    int i;

    try {
      v.push(1);
      v.push(2);
      v.push(3); // Itt VeremTeliException keletkezik
      i=v.pop();
      i=v.pop();
      i=v.pop(); // Itt VeremUresException keletkezik
    }
    catch (VeremException ve) {
      System.out.println("A kovetkezo problema tortent: "+ve);
    }
  }
}
```

8.5.3. Ada 95

Az alábbi példaprogram ugyanazt a feladatot oldja meg, mint a 8.5.1. pontban található program: egy kételemű vermet valósít meg. A veremnek két művelete van, a *push* egy elemet tesz a verem tetejére, a *pop* pedig egy elemet vesz le a verem tetejéről. Mindkét metódus kivételdobással jelzi, ha hiba történt (teli verembe szeretnénk elemet tenni vagy üres veremből szeretnénk elemet kivenni). Láthatjuk, hogy a 8.5.2. pontban található megoldáshoz hasonlóan itt is a futásidejű keretrendszer veszi észre, hogy kiürült, illetve teli van a verem.

```
package VEREM_E is
    type Verem is limited private;
    procedure Push(v: in out Verem; i: INTEGER);
    procedure Pop(v: in out Verem; i: out INTEGER);
    VeremTeliException: exception;
    VeremUresException: exception;
private
    type Tomb is array(0..1) of INTEGER;
    type Verem is record
        sp: NATURAL := 0;
        v: Tomb;
    end record;
end VEREM_E;

with Ada.Exceptions; use Ada.Exceptions;

package body VEREM_E is
    procedure Push(v: in out Verem; i: INTEGER) is
    begin
        v.v(v.sp) := i;
        v.sp := v.sp + 1;
    exception
        when CONSTRAINT_ERROR =>
            Raise_Exception (VeremTeliException'Identity, "Megtelt a verem!");
    end Push;

    procedure Pop(v: in out Verem; i: out INTEGER) is
    begin
        v.sp := v.sp - 1;
        i := v.v(v.sp);
    exception
        when CONSTRAINT_ERROR =>
            Raise_Exception (VeremUresException'Identity, "Ures a verem!");
    end Pop;
end VEREM_E;
```

```
with VEREM_E, TEXT_IO, Ada.Exceptions;
use VEREM_E, TEXT_IO, Ada.Exceptions;

procedure Kivetelek is
    v: Verem;
    i: INTEGER;
begin
    Push(v, 1);
    Push(v, 2);
    Push(v, 3);          -- Itt VeremTeliException keletkezik.
    Pop(v, i);
    Pop(v, i);
    Pop(v, i);           -- Itt VeremUresException keletkezik.
exception
    when e: VeremTeliException | VeremUresException =>
        Put("A kovetkezo problema tortent:");
        Put_Line(Exception_Message(e));
end Kivetelek;
```

8.5.4. Eiffel

Az alábbi példaprogram ugyanazt a feladatot oldja meg, mint a 8.5.1. pontban található program: egy kételemű vermet valósít meg. A veremnek két művelete van, a *push* egy elemet tesz a verem tetejére, a *pop* pedig egy elemet vesz le a verem tetejéről. Mindkét metódusnak az előfeltétele ellenőrzi, hogy nem teli verembe szeretnénk-e elemet tenni vagy üres veremből szeretnénk-e elemet kivenni.

```
class
  VEREM
creation
  make
feature
  v: ARRAY[INTEGER];
  sp: INTEGER;

  make is
    do
      !!v.make(0, 1);
      sp:=0;
    end;

  push(i: integer) is
    require
      VeremNincsTeli: sp < 2
    do
      v.put(i, sp);
      sp := sp + 1;
    end;

  pop: INTEGER is
    require
      VeremNemUres: sp > 0
    do
      sp := sp - 1;
      Result := v.item(sp);
    end;
end
```

```
class
  KIVETEL
creation
  make
feature
  make is
    local
      v: VEREM;
      i: INTEGER;
      volt_kivetel:BOOLEAN;
    do
      !!v.make;
      if not volt_kivetel then
        v.push(1);
        v.push(2);
        v.push(3);  -- Itt sérül a VeremNincsTeli előfeltétel.
      else
        i := v.pop;
        i := v.pop;
        i := v.pop;    -- Itt sérül a VeremNemUres előfeltétel.
      end;
    rescue
      if not volt_kivetel then
        volt_kivetel := true;
        retry;
      end;
    end;
end
```

Feladatok

1. Soroljon fel néhány programozási nyelvet, ahol a kivételek nem objektumok!

2. Miért jó, ha az alprogramok specifikációja tartalmazza, hogy milyen kivételeket dobhat az alprogram?

3. Mire használható a *finally* szerkezet a Java nyelvben?

4. Egészítse ki az alábbi C++ nyelvű programot:

```
#include <cstdlib>
#include <iostream>
#include <string>

class NemSzam {
    std::string hibasBemenet;
    friend std::ostream& operator<<(std::ostream& os, const NemSzam& nsz);
public:
    NemSzam(const std::string& s) { hibasBemenet = s; }
};

std::ostream& operator<<(std::ostream& os, const NemSzam& nsz) {
    return os << "Hibás bemenet: " << nsz.hibasBemenet;
}

long szam(const std::string& s) throw (NemSzam) {
    ⟨hiányzó kódrészlet⟩
}

int main(void) {
    try {
        cout << szam("12") << endl;
        cout << szam("-34") << endl;
        cout << szam("szoveg") << endl;
    }
    catch (NemSzam ns) { cout << "Hiba történt: " << ns << endl; }
    return 0;
}
```

A kiegészítendő *szam* függvény feladata az, hogy a paraméterként megadott szövegből számot készítsen, ha számot tartalmaz a szöveg. Ha nem értelmezhető 10-es számrendszerbeli számként a paraméter, akkor *NemSzam* típusú kivételt dobjon a függvény. Használja fel a szabványos függvénykönyvtárban található *strtol* függvényt! Helyes megoldás esetén a következő kimenetet adja a program:

```
12
-34
Hiba történt: Hibás bemenet: szoveg
```

5. Egészítse ki az alábbi Java nyelvű programot:

```
import java.lang.*;

class REKivetel extends Exception {
    String hibasBemenet;

    public REKivetel(String s) {
        hibasBemenet = s;
    }

    public String toString() {
        return "Hibás bemenet: " + hibasBemenet;
    }
}

public class RepuloEkezet {
    String bemenet;

    public RepuloEkezet(String s) {
        bemenet = s;
    }

    String konvertal() throws REKivetel {
        ⟨hiányzó kódrészlet⟩
    }

    public static void main(String[] args) {
        RepuloEkezet r;
        try {
            r = new RepuloEkezet
                    ("a\'rvi\'ztu\"ro\" tu:ko:rfu\'ro\'ge\'p");
            System.out.println(r.konvertal());
            r = new RepuloEkezet("a: nincs magyarban");
            System.out.println(r.konvertal());
        }
        catch(REKivetel e) {
            System.out.println("Hiba történt: " + e);
        }
    }
}
```

A kiegészítendő *konvertal* metódus feladata az, hogy az osztály konstruktorának paraméterként megadott repülő ékezetes szöveget ékezetessé alakítsa, és ezt adja vissza. Ha a magyar nyelvben nem előforduló ékezetet talál, akkor azt kivétel dobásával jelezze a metódus. Helyes megoldás esetén a következő kimenetet adja a program:

```
árvíztűrő tükörfúrógép
Hiba történt: Hibás bemenet: a: nincs magyarban
```

6. Egészítse ki az alábbi Eiffel nyelvű programot:

```
class
    ORA

creation
    make

feature
    delelott(idopont: STRING): BOOLEAN is
      require
          idopont_korrekt: forma_rendben(idopont);
      ⟨hiányzó kódrészlet⟩
    end;

feature
    make is
      local
          masodik: BOOLEAN;
      do
          io.put_boolean(delelott("11:23"));
          io.put_string("%N");
          io.put_boolean(delelott("23:11"));
          io.put_string("%N");
          if not masodik then
              io.put_boolean(delelott("szoveg"));
              io.put_string("%N");
          end;
      rescue
          masodik := true;
          retry;
    end;
end
```

A kiegészítendő *delelott* metódus feladata az, hogy eldöntse a paraméterként megadott szövegről, hogy délelőtti időpontot takar, vagy sem. A metódus előfeltétele az, hogy a paraméter tényleg egy időpont. Ezt a *forma_rendben* metódus dönti el, amelynek megírása szintén az Olvasó feladata.

9 Modulok

Ebben a fejezetben a moduláris szoftvertervezés főbb elméleti kritériumai és gyakorlati szempontjai kerülnek ismertetésre. Ezt követik a csomagokkal kapcsolatos alapismeretek, és az alkalmazásukra vonatkozó tanácsok. A fejezetet néhány imperatív és funkcionális programozási nyelven íródott példaprogram teszi teljessé.

Összetettebb feladatok esetén biztosan elérjük azt a pontot, amikor megoldó programunk kezelhetetlen méreteket ölt. A probléma mérete növekedésének megfelelően a programrendszer fejlesztése szempontjából tehát előnyösebb, ha azt nem önmagában, hanem egymástól minél inkább független, jól definiált kapcsolatban álló programegységekként tervezzük meg. Ezen tervezési egységeket moduloknak, a tervezés ilyen módját pedig moduláris tervezésnek nevezzük.

A továbbiakban részletesen kitérünk a modulok elméleti hátterét adó elvekre, a gyakorlati felhasználás szempontjaira, majd a fejezet végén bemutatjuk ennek nyelvi támogatását és speciális megvalósítását az Assembly, C, Java, FORTRAN, Pascal, Modula, Ada és SML nyelvekben.

9.1. A moduláris tervezés

Ha jobban belegondolunk, a modularitás segítségével több, az objektumorientált programtervezésre jellemző kritériumnak is – elsősorban az újrafelhasználhatóság, kiterjeszthetőség és kompatibilitás szempontjainak – könnyebb megfelelni. Ezen alapszik az úgynevezett moduláris tervezés. (A modul itt tehát nem programnyelvi fogalom, hanem a tervezés egysége.)

9.1.1. A moduláris tervezés kritériumai

Vegyük sorra, miket tekintünk a moduláris tervezés legfőbb szempontjainak!

Moduláris dekompozíció • A moduláris dekompozíció a feladat több egyszerűbb részfeladatra bontását jelenti, amely részfeladatok megoldása már egymástól függetlenül elvégezhető. Ennek segítségével csökkentjük a feladat bonyolultságát.

A módszert általában ismételten alkalmazzuk, azaz a részfeladatokat magukat is tovább bontjuk. Ezzel lehetővé tesszük azt is, hogy a feladat megoldásán egyszerre több ember dolgozzon. A módszer egy fával ábrázolható, ahol a fa csomópontjai az egyes dekompozíciós lépéseknek felelnek meg.

Például a felülről lefelé történő programtervezés alapvetően erre épül. Itt a tervezők a probléma absztrakt leírásából kiindulva a dekompozíció ismételt alkalmazásával jutnak el a leírás egy olyan szintjére, amely már implementálható a konkrét programnyelven.

Moduláris kompozíció • A moduláris kompozíció azt jelenti, hogy programjainkat minél inkább már meglévő programegységekből, mint építőkövekből szeretnénk felépíteni. Egy programtervezési módszer akkor elégíti ki ezt a kritériumot, ha támogatja olyan szoftverelemek létrehozását, amelyek szabadon kombinálhatók egymással. Míg a moduláris dekompozíció a program kiterjeszthetőségét, a moduláris kompozíció az újrafelhasználhatóságot támogatja.

Az a célunk ezáltal, hogy a programkészítés során minél több már létező, szabványos elemet használjunk fel. Például már évtizedek óta fejlesztenek a numerikus algoritmusok témájában alprogram-könyvtárakat.

Megjegyzés: az előző két kritérium független egymástól, sőt egymás ellen is dolgozhatnak, az ismételt dekompozíciós lépések eredményeként kapott modulok ugyanis igen sok esetben nagyon is speciális részproblémákat oldanak meg, s ezáltal nem igazán használhatók fel újra.

Moduláris érthetőség • Ez a kritérium azt jelenti, hogy a modulok önállóan is egy-egy értelmes egységet alkossanak, megértésükhöz minél kevesebb „szomszédos" modulra legyen szükség. Ez a szempont a programok karbantartása során lesz jelentős.

Moduláris folytonosság • Az elnevezést a folytonos függvények területe ihlette: a specifikáció „kis" változtatása esetén a programban is csak „kis" változtatás szükségességét írja elő. Ez azt is jelenti, hogy a „kis" változások egyes modulokat érintsenek csupán, s ne az program egész szerkezetét.

Sajnos az, hogy mit nevezünk „kis" változtatásnak a specifikációban, illetve a programban, nincs precízen meghatározva, a fogalom mégis érthető.

Egy egyszerű példa: programjainkban numerikus, illetve szöveges literálok helyett használjunk szimbolikus konstansokat – hiszen ekkor változás esetén csak egy helyen, a definícióban kell javítani.

Moduláris védelem • Szintén célul kell kitűznünk a program egészének védelmét abnormális helyzetek bekövetkeztekor. Egy hiba hatása csakis egy – esetleg néhány – modulra korlátozódjon! Egy olyan módszer például, amely előírja, hogy minden adatbeolvasó modul maga felelős az adatok ellenőrzéséért, eleget tesz a moduláris védelem szempontjának.

9.1.2. A modularitás alapelvei

Az előző kritériumhalmaz teljesítéséhez bizonyos elveket feltétlenül szem előtt kell tartanunk.

A modulok nyelvi támogatása • A moduláris tervezés csak akkor lehet hatékony, ha a modulok illeszkednek a használt programozási (vagy programtervezési, specifikációs) nyelv szintaktikai egységeihez. Egy programozási nyelv esetén ez az elv azt is jelenti, hogy a nyelvi moduloknak önmagukban is lefordíthatónak kell lenniük.

Ez az elv ugyan teljesen technikai jellegű, de ha nem teljesül, komoly gondjaink lehetnek az előzőekben felsorolt kritériumok megvalósításával. Például ha egy rendszert jól elhatárolható részfeladatokra akarunk bontani (dekompozíció), akkor az a célszerű, ha ezek egyúttal különálló szintaktikai és fordítási egységeket is képeznek. Másrészt ha egy modul nem egy nyelvi egység, hanem több, vagy esetleg több egység bizonyos részeiből állítható össze, akkor a moduláris kompozíció válik lehetetlenné.

Megfelelő programnyelvi eszközök támogatása nélkül a legkiválóbb módszertani elvek is olyanok, mint „a madár szárnyak nélkül". A programnyelvi eszközöket csak részben lehet helyettesíteni a programozási stílussal és gondolkodásmóddal.

Kevés kapcsolat • Minden modul minél kevesebb másik modullal kommunikáljon!

A modulok között sokféle kommunikáció lehetséges: a modulok meghívhatják a másik modul által nyújtott szolgáltatásokat, megoszthatják adatszerkezeteiket, stb. Ez az elv ezen kapcsolatok számát akarja korlátozni.

Célul kell kitűzni, hogy egy n modulból álló rendszerben a modulkapcsolatok száma minél közelebb legyen az $n - 1$ minimális értékhez, és minél távolabb az $n * (n - 1)/2$ maximális értéktől.

Ez az elv voltaképpen a folytonosság és a védelem kritériumaiból ered: ha túl sok a kapcsolat a modulok között, akkor egy változtatás hatása vagy egy hiba terjedése is túl sok modult érint.

Gyenge kapcsolatok • Ha két modul egyáltalán kommunikál egymással, akkor olyan kevés információt cseréljenek, amennyi épp csak szükséges.

Ez az elv is a folytonosság és a védelem szempontjait veszi elsődlegesen figyelembe, hiszen minél kisebb fokú a modulok közti kapcsolat, annál kevésbé hatnak ki az egyes modulokban bekövetkező változások a többi modulra, és hasonlóképpen a hibák is annál kevésbé terjednek tovább.

Explicit interfészek • Ez az alapelv azt mondja ki, hogy ha két modul kommunikál egymással, akkor annak ki kell derülnie legalább az egyikük szövegéből.

Ez az elv több kritériumot is érint: elsősorban a dekomponálhatóságét és a komponálhatóságét, hiszen mindkettő érdekében világosan jelezni kell minden külső kapcsolatot; a folytonosság kritériumának teljesíthetőségéhez is fontos, hogy az interfészből derüljön ki egyértelműen, egy változtatás mely programegységekre lesz hatással; de talán a legjobban a moduláris érthetőség függ tőle – a modulok közti kapcsolatok nem láthatók át világosan ennek teljesítése nélkül.

Információ-elrejtés • Egy modulról semmilyen más (belső) információt nem használhatunk fel, csak azt, amit explicit módon nyilvánosként deklaráltunk.

Ennek az elvnek az alkalmazása feltételezi, hogy az összes modult a többi modul tervezője csak annak hivatalos interfészén keresztül ismerheti. Ez azt jelenti, hogy minden modul alapvetően két részből áll: egy interfészből és egy privát, más modulokból semmilyen eszközzel nem elérhető részből. Ezt a lehetőséget a programnyelvnek kell biztosítania. Természetesen elképzelhető, hogy az egész modul az interfészbe kerül, de az elv azt mondja ki, hogy ez legyen a „kivétel".

Az információ ily módon történő elrejtése a folytonosság alapvető feltétele. Ha ugyanis egy modulban csak a privát, rejtett részt változtatjuk meg, és az interfész változatlan marad, a modullal kommunikáló más modulokat ez a változás egyáltalán nem érinti.

Nyitott és zárt modulok • Az eddigi szabályok egy részét a „hagyományos" programnyelvek is támogatják. Jó modulszerkezetet, információ-elrejtést megvalósíthatunk például a Modulában vagy az Adában. Az újrafelhasználhatóság követelménye azonban további támogatást is igényelne – ennek megfogalmazásához először két definíciót vezetünk be:

- Egy modult zártnak nevezünk, ha más modulok számára egy jól definiált interfészen keresztül érhető el, és csakis változatlan formában kerülhet felhasználásra. Ez programnyelvi szempontból azt jelenti, hogy a zárt modul egy önállóan lefordított programegység, amelyet egy programkönyvtárban tárolhatunk, és szükség szerint hozzászerkeszthetünk programunkhoz.
- Egy modult nyitottnak nevezünk, ha még kiterjeszthető, ha az általa nyújtott szolgáltatások bővíthetők, vagy például hozzávehetünk további mezőket a benne levő adatszerkezetekhez, s ennek megfelelően módosíthatjuk a modul eddigi szolgáltatásait.

Jól látható, hogy a zártság és a nyitottság a hagyományos szemléletben egymásnak ellentmondó követelmények. A „hagyományos" programnyelvek általában lehetőséget adnak zárt modulok készítésére, a felhasználó által tervezett absztrakt adattípus megvalósítására (modul, csomag stb.), nyitott modulok támogatására pedig jó példa az Ada 95 hierarchikus könyvtárstruktúrája, amelyről a továbbiakban még részletesen szólunk.

Az úgynevezett objektumorientált megközelítés egy új lehetőséget kínál majd: itt a kiindulási alap az osztály – ami az absztrakt adattípus megfelelője –, s egy új eszközt, az öröklődést adja arra, hogy már meglévő, lezárt osztályokból – azokat nyitottként kezelve – hozzunk létre új osztályokat.

Az újrafelhasználhatóság • A szoftverfejlesztés során nagyon gyakran kell hasonló (de nem azonos) programelemeket készíteni. Például a programok nagy része használ rendezéseket, kereséseket, stb.

A programozók legtöbbje szívesen veszi (venné), ha ezeket nem kell(ene) újra és újra megvalósítania, hanem rendelkezésére állnának megfelelően újrafelhasználható modulok.

Az ezzel kapcsolatos problémák jelentős része abból származik, hogy bár a feladatok sokszor erősen hasonlók, mégsem azonosak. Tudnunk kell természetesen azt is, hogy egy modul csak akkor használható fel újra, ha elég általános.

Általános programrészeket készíteni rövid távon valójában költségesebb (hiszen gondosabb tervezést és megbízhatóbb megvalósítást igényel), mint egy-egy konkrét feladatot megoldani. Így csak professzionális programozóknak éri meg a befektetést, mert csak több program készítése esetén térül meg.

Milyen követelményeket fogalmazhatunk meg a modulstruktúrára vonatkozóan, ha újrafelhasználható programrészeket akarunk készíteni?

A típusok változatossága • A modulokat úgy kell elkészíteni, hogy többféle típusra is működjenek. Például a verem típus moduljának segítségével képesek kell hogy legyünk egészek vagy karaktersorozatok vermeit egyaránt létrehozni.

Ezt az elvárást a programozási nyelvek általában a sablonnak (generic, template) nevezett programnyelvi eszközzel támogatják, de az öröklődés módszerével is megvalósítható.

Adatszerkezetek és algoritmusok változatossága • Figyelembe kell venni, hogy a program esetenként sokféle adatszerkezettel dolgozik – például a fa, a lista, a tömb, a szekvenciális fájl stb. mind bejárható szekvenciálisan – bár természetesen más-más algoritmussal. Egy lineáris kereső eljárás akkor elég általános, ha ezek mindegyikére működik.

Egy típus – egy modul • Egy típus műveletei kerüljenek egy modulba, így a modul felhasználója könnyebben át tudja tekinteni, milyen műveletek megengedettek az adott típusú objektumokra.

Reprezentáció függetlenség • Egy adattípus reprezentációjának megváltoztatása ne okozzon a modulon kívül változtatásokat! Ezt a rejtett reprezentáció, illetve implementáció segítségével érhetjük el.

Ennél általánosabb igény is előfordulhat: esetleg szeretnénk, hogy programunkban egyszerre jelenjen meg egy típus több reprezentációja is. Például a komplex számok összeadása, illetve szorzása más-más reprezentáció mellett egyszerűbb. Ez az igény a hagyományos programnyelvekben nagyon nehézkesen teljesíthető. A lényeg: anélkül használhassunk egy objektumot, hogy tudnánk, mi a reprezentációja, illetve műveleteinek implementációja.

Ez a követelmény a szoftverek kiterjeszthetősége szempontjából is lényeges. A probléma általánosan úgy fogalmazható meg, hogy egy adott művelet alkalmazásakor valahol ki kell választani a típusnak megfelelő konkrét algoritmust. Ha az erre vonatkozó döntés a szolgáltató modulban van, akkor ennek ismernie kell minden elérhető implementációt – s bár ez önmagában is nagyméretű és nehezen kezelhető modulokhoz vezetne, nem oldaná meg az utólag felmerülő új típusvariációk egyszerű kezelését.

Gondoljunk például egy grafikus rendszerre, amely kezeli a pont, egyenes, háromszög, négyszög, kör alakzatok listáját. Egy lista azonos típusú elemeket kell hogy tartalmazzon; ehhez a programcsomag eredeti forrásszövegében az adattípust hagyományos módon egy variáns részt tartalmazó rekorddal szokták reprezentálni.

Az egyes alakzatok kirajzolása, transzformálása stb. különböző algoritmusok végrehajtását igényli. Ezt megint csak hagyományos módon a programszövegben egy, a variáns rekord lehetőségeinek megfelelő `case` utasítás kezeli.

Tegyük fel, hogy megvettünk egy programcsomagot, amelynek segítségével használni tudjuk alakzatainkat. Ha utólag derül ki, hogy például KRESZ táblák kirajzoltatásához egy hatszögre is szükségünk lenne, általában két lehetőségünk van: vagy megbízzuk az eredeti programcsomag tervezőjét, hogy módosítsa a programcsomagot megfelelően, vagy elkészítjük mi az új alakzatot, az összes műveletét, azokat is, amelyek egyébként változtatás nélkül azonosak az eredeti programcsomagban szereplőkkel. Ekkor programjainkban is valószínűleg egy nagy `case` utasítás írja majd le az esetszétválasztást.

Mindkét lehetőség – mind az eredeti programszöveg módosítása, mind a felhasználói program megváltoztatása – fáradságos, nehezen kezelhető és költséges.

Az itt vázolt problémára is az igazi megoldást az objektumorientált programozás eszközei jelentik.

9.2. A modulok nyelvi támogatása

Az eddigi elmélet tárgyalása után tekintsük át, hogy mely programnyelv milyen eszközöket nyújt a moduláris programozás megvalósításához.

9.2.1. Alacsonyszintű programnyelvek

Kezdetben a programozás egyetlen nyelve a gépi kódnak megfelelő assembly nyelv volt. Az ezen programnyelven megírt programot annak forrásszöveg alakjából egy speciális fordítóprogram, az assembler alakította futtatható (gépi kódú) formába. Az assembler feladata elég egyszerű volt: az assembly utasítások (úgynevezett mnemonic-ok) lefordítása és a címhivatkozások kiszámítása. Ezen címhivatkozások céljául pedig akár olyan szimbólumokra is hivatkozhattunk, melyek nem voltak részei az adott forrásfájlnak, azokat akár egy másik forrásfájlban is definiálhattuk. Ez a lehetőség biztosította az önálló fordítási egységek létrehozását, ezáltal ezt a modulképzés kezdeti megvalósításának tekinthetjük. Így azonban egy futtatható program létrehozásához még egy lépés szükséges a fordítás után: az összeszerkesztés (linkelés) alatt az önállóan lefordított kódegységek még fel nem oldott hivatkozásait számíttatjuk ki. A fordítási egység forrásszövegét ezért elég volt csak egyszer a fordítóval lefordíttatni, a lefordított kódot pedig más és más programokkal akár többször is újra összeszerkeszthetjük. A lefordított fordítási egységek kódjának tárolását megkönnyítendő, azokat összegyűjtve úgynevezett programkönyvtárakban helyezhettük el. Ezen fogalmak és alapelvek a magasabb szintű nyelvekre is érvényben maradtak, csak ott rendszerint minden teendőt maga a fordítóprogram végez el.

Az assembly nyelvhez legközelebb a kifejezetten rendszerprogramozásra tervezett C programozási nyelv [Ker 89] áll. Ennek megfelelően maga a nyelv egyszerű és „gépközeli", viszont a fordítási egységek kezelését elég hatékonyan oldja meg. Ezen megoldás az előfordítót felhasználva lehetővé teszi, hogy programunkat modulokra bontsuk. Minden fordítási egység két forrásfájlra tagolódik: a modul interfészét publikus deklarációk formájában tartalmazó úgynevezett fejléc (header) fájlra, valamint a tényleges megvalósítást tartalmazó forrásfájlra. Modulunk minden felhasználási helyén a megfelelő fejlécfájl kerül forrásszinten beágyazásra (*#include*). Mivel ugyanazon modul fejléce akár több más felhasznált modulon keresztül is beágyazható, a többszörös deklarálás hibájának elkerülése érdekében az előfordító által biztosított feltételes fordítást hívhatjuk segítségül, valahogy így:

```
#ifndef MODUL_H /* Modulazonosító szimbólum. */
#define MODUL_H
/* Ezt a részt a fordító így csak egyszer fogja feldolgozni. */
. . . /* Ide kerül a modul interfészdeklarációja. */
#endif
```

Egy fordítási egységet így akkor lehet hiba nélkül lefordítani, ha minden hivatkozott fejlécfájl elérhető a fordító számára. Ekkor azonban az összeszerkesztési lépés kihagyását egy külön kapcsolóval jelezni kell, futtatható kód előállításakor viszont elég a már lefordított fordítási egységeket felsorolni.

Fejlécfájlokat használva a deklarációk láthatósága valamilyen szinten mégis szabályozhatóvá válik, annak ellenére, hogy maga a C nyelv erre nem ad lehetőséget. A fejlécfájlok tartalma ugyanis teljesen nyilvános, hiszen minden olyan modul felhasználhatja, ahová a fejlécfájlt beágyazzuk. A megvalósító modul deklarációit azonban kívülről senki sem használ(hat)ja, ezért azokat a modulra nézve lokálisnak és privátnak tekinthetjük.

9.2.2. Procedurális programnyelvek

A magasszintű programnyelvek közül a fordítási egységeket a FORTRAN [Lőcs 77] vezette be. A FORTRAN programok szegmensekből állnak, amelyek közül az egyik a főszegmens, a többi pedig a következő háromféle valamelyike lehet: függvény (*FUNCTION*), eljárás (*SUBROUTINE*) vagy adatszegmens (*BLOCK DATA*). A szegmensekben használt azonosítók (a szegmens neve és a *COMMON* adatmezők kivételével) lokálisak. A szegmensek elején deklarációs sorok állnak, utánuk következik a végrehajtható rész. A deklarációs rész kiértékelése statikus, a változók leképezése a memóriába fordítási időben történik. Ezen szegmensek a nyelv fordítási egységei. A nyelv nem tartalmaz elemet a szegmensek közötti kapcsolatok leírására, fordítás közben pedig nincs mód a kapcsolatok helyességének ellenőrzésére.

Ezen megszorításokat kiküszöbölték a nyelv FORTRAN 90-es változatában. Egy újabb fordítási egység, a modul (*MODUL*) bevezetésével lehetőség nyílt interfész (*INTERFACE*) definiálására, melynek segítségével a modulok közti kapcsolatok ellenőrzése már fordítás közben elvégezhető.

```
module fv_def ! Ez a modul tartalmazza az interfész definícióját.
  interface fv
    ! Az interfész deklarálja a függvény két különböző szignatúrájú változatát.
    function real_fv(param1, param2) result(visszatérési_érték)
      real param1, param2, visszatérési_érték
    end function real_fv
    function double_fv(param1, param2) result(visszatérési_érték)
      double precision param1, param2, visszatérési_érték
    end function double_fv
  end interface
end module

! Egy külön fordítási egység tartalmazza a megvalósítást.
function real_fv(param1, param2) result(visszatérési_érték)
  real param1, param2, visszatérési_érték
  visszatérési_érték = param1*param2
end function real_fv
function double_fv(param1, param2) result(visszatérési_érték)
  double precision param1, param2, visszatérési_érték
  visszatérési_érték = param1*param2
end function double_fv
! A program a következő oldalon folytatódik.
```

```
! Folytatás az előző oldalról.

program main    ! A főprogram modulja.
 use fv_def     ! Hivatkozás az interfész fordítási egységére.
  real r1, r2, r3
  double precision d1, d2, d3
  data r1/2401.0/, r2/9.81245/
  data d1/2401.0d0/, d2/9.81245d0/
! Az interfészen keresztül a két különböző szignatúrájú függvényváltozat felhasználása.
  r3 = fv(r1, r2)
  d3 = fv(d1, d2)
  write(*,'(1x,1p,e16.9,t25,d16.9)') r3, d3
end program
```

A modularitást még alaposabban támogatja a Modula programozási nyelv, melynek fejlesztése már a harmadik generációnál tart. Eredetileg Niklaus Wirth fejlesztette ki a Pascal megalkotásakor szerzett tapasztalatait felhasználva. A Modula-2 egyik legsikeresebb jellemzője a modulok közötti interfészek explicit leírásának biztosítása. Az interfészek és modulok leírását gyakorlatilag egy az egyben örökölte a Modula-3 [Nel 91] is.

A modulok a Modula-3 alapvető építőkövei. A modulok deklarációk nevesített gyűjteményei, beleértve a konstans-, típus-, változó-, és alprogram-deklarációkat. Egy modul a saját deklarációit elérhetővé teheti más modulok – az ügyfelei – számára. Ezt a kívülről látható részt írják le az interfészek, amelyek mellett a moduloknak van rejtett implementációs része is.

A modulok importálják azokat az interfészeket, amelyeket felhasználnak, és exportálják azokat, melyeket megvalósítanak. Az interfészek használatával biztonságosan tudjuk moduljainkat külön fordítani. Nem kell félni attól, hogy valamely hivatkozásunk érvénytelenné válik egy már lefordított modulban. Egy-egy modul kifelé csak az interfésze(i)n keresztül látszik, ami lehetőséget nyújt arra, hogy programunkat átlássuk anélkül, hogy a részleteket ismernénk. Ez a karbantartást és a betanulást egyszerűsíti le nagy mértékben.

A Modula-3 fordítási egységei a modulok és az interfészek. A modul olyan, mint egy blokk, kivéve az azonosítók láthatóságát: az entitások láthatók egy blokkban, ha az adott, vagy valamely befoglaló blokkban deklarálták, míg egy modulban akkor, ha az adott, vagy valamely, a modul által importált vagy exportált interfészben deklarálták.

A modulok exportálnak egy interfészt, hogy törzset rendeljenek az interfészben deklarált eljárásoknak. A modulok vagy interfészek importálnak egy interfészt, hogy láthatóvá tegyék az abban deklarált entitásokat. Egy modult a következőképpen definiálhatunk:

MODULE ⟨*azonosító*⟩ EXPORTS ⟨*exportált interfészek listája*⟩;
    ⟨*import utasítások*⟩;
    ⟨*deklarációk*⟩;
⟨*modul törzs -- blokk*⟩ ⟨*azonosító*⟩;

```
(* Egyszerű példaprogram. *)
MODULE Hello EXPORTS Main; (* Main: predefinit, üres interfész. *)
    FROM SIO IMPORT Put; (* import utasítás *)
BEGIN
    Put("Hello world!"); (* törzs blokk *)
END Hello.
```

A program modulok és interfészek összessége. Tartalmaz minden olyan interfészt, melyet moduljai vagy interfészei importálnak vagy exportálnak, és amelyben minden eljárást, interfészt és modult legfeljebb egyszer definiáltak. A program végrehajtása a modulok törzsének végrehajtását jelenti a következő szabályok alapján:

- Ha egy modul (M) függ egy másik modultól (N), akkor N törzse előbb hajtódik végre, mint M-é;
- M függ N-től, ha M használ legalább egy olyan interfészt, amelyet N exportál, vagy M függ egy olyan modultól, amely függ N-től;
- M használ egy I interfészt, ha M importálja vagy exportálja I-t, vagy ha M használ egy olyan interfészt, amely (közvetlenül vagy közvetve) importálja I-t;
- az előbbiek alapján nem meghatározott esetben implementációfüggő a végrehajtás sorrendje.

A nyelvben lehetőség van sablonok írására is. A modularitás jellege miatt csak modul tehető sablonná, illetve annak interfész része. A sablonnak paraméterek adhatók, méghozzá csak interfészekkel paraméterezhetünk. Ezeket már a GENERIC INTERFACE-ben fel kell tüntetni és a GENERIC MODULE leírásában is szerepelniük kell.

9.2.3. Az Ada nyitott modulok támogatása

Az Ada nyelv szintén a Pascalból fejlődött ki rendszerprogramozási célból. Fejlesztésének legfőbb szempontja a biztonság és programhelyesség biztosítása, ezért különösen ügyeltek a moduláris megközelítésből adódó előnyök kihasználására.

Az Ada egyik legnagyobb erőssége az úgynevezett *csomagokban* (*package*) rejlik, melyekben a specifikációs és a törzs rész segítségével egyértelműen elválaszthatjuk egymástól egy absztrakt típus felhasználói felületét és implementációját. Így függetlenül lefordítható az implementáció és a csomag szolgáltatásait igénybe vevő ügyfél, feltéve, hogy a specifikáció változatlan marad. Ez kisebb méretű programok esetén használható is, de összetettebb feladatok megvalósításakor nehézkessé válhat.

Előfordulhat, hogy két, egymástól logikailag független csomagot akarunk készíteni, amelyek közösen használnak egy *átlátszatlan* (*private*) típust. Az Ada 83-ban ezt két különböző módon valósíthatjuk meg: az egyik, hogy a közösen használt típust nem átlátszatlanként deklaráljuk, így mindkét csomag számára látható lesz. Ebből következően azonban az összes többi őt használó csomag is látni fogja az adott típus reprezentációját, ez viszont az absztrakció lényeges megsértését jelentené. A másik lehetséges megoldás az Ada 83-ban az, hogy a két, logikailag független feladat megoldását közös csomagban valósítjuk meg, amely csomag átlátszatlan részében már elhelyezhetjük a közösen használt elemeket is. Így

az absztrakció ugyan nem sérül, de a csomag mérete akkora lesz, hogy ez lényegesen megnöveli a fordítási költségeket. Az sem túl szerencsés, hogy logikailag független programozási elemeket helyezünk el közös csomagban.

A másik lényeges probléma az Ada 83 csomagjaival akkor merül fel, amikor egy már meglévő absztrakt típus szolgáltatásait akarjuk kiterjeszteni újabb műveletek hozzávételével. Egy csomag specifikációs részének kiegészítésekor természetesen az egészet újra kell fordítani, de az összes olyan alkalmazást is, amely használja a módosított csomag szolgáltatásait, attól függetlenül, hogy szükségük van-e az újonnan hozzávett elemekre, vagy sem.

Az Ada 95-ben [Nyé 98] az efféle problémák elkerülése végett vezették be a hierarchikus könyvtárstruktúrát a gyermekcsomag és a gyermek-alprogram fogalmával együtt. A gyermekcsomag tulajdonságai:

- Láthatóság szempontjából lehet nyilvános (public) és átlátszatlan (private).
- Logikailag alárendeltje a szülőcsomagnak és eléri annak látható és átlátszatlan részét is.
- A szülőcsomag neve előtagja a gyermekcsomag nevének.
- Bármely csomagnak lehet sablon (generic) gyermeke, viszont sablon csomagnak csak sablon gyermeke lehet.

Először a nyilvános (public) gyermekcsomagokkal foglalkozunk. Kezdjük rögtön egy példával! A feladat egy kanonikus alakú komplex számokat kezelő csomag létrehozása:

```
-- Ada példaprogram.
package Komplex_Szám is
  type Komplex is private;
  function Konstruktor(Valós, Képzetes: Float) return Komplex;
  function "+" (arg1, arg2: Komplex) return Komplex;
  -- A "-", a "*" és a "/" hasonlóan;
  function Valós_Rész(X: Komplex) return Float;
  function Képzetes_Rész(X: Komplex) return Float;
private
...
end Komplex_Szám;
```

Később bővül a feladat, a komplex számok trigonometrikus alakjának kezelését is lehetővé kell tenni. Az Ada 83-ban a fenti csomagot kell kiegészíteni az új műveletekkel, míg az Ada 95-ben ehelyett a következő megoldás adódik:

```
package Komplex_Szám.Polar is
  function Polar_2_Komplex(R, Theta: Float) return Komplex;
  function "abs" (arg: Komplex) return Float;
  function Arg(arg: Komplex) return Float;
end Komplex_Szám.Polar;
```

A gyermekcsomagra vonatkozó láthatósági szabályok kialakításakor úgy tekintették, mintha a látható része a szülő látható része végére lenne beillesztve, az átlátszatlan (private) része pedig a szülő átlátszatlan részében lenne. Ebből adódóan a gyermekcsomag látja a szülő látható részét, az átlátszatlan (private) része, valamint a megvalósítása (package body) pedig a szülő átlátszatlan (private) részét is eléri. Mindazonáltal az itt deklarált új műveletek nem primitív műveletei a típusnak, mert nem azonos csomagban vannak vele.

Tehát a gyermekcsomag átlátszatlan részei a szülőcsomag átlátszatlan részéhez csatlakoznak. A nyilvános (public) gyermekcsomag látható részeiben viszont nem láthatók a szülőcsomag átlátszatlan részei, mivel ez átnevezésekre adna lehetőséget, és így ezen a módon a rejtett részek is láthatóvá válnának mindenki számára. Ez szintén az absztrakció lényeges megsértését jelentené. (Az átlátszatlan (private) gyermekcsomagokra más láthatósági szabályok vonatkoznak, mint azt a későbbiekben látni fogjuk.)

Az előbbi példán bemutatjuk a láthatósági szabályok alkalmazását a gyakorlatban.

```
with Komplex_Szám.Polar;
package Teszt is
...
-- Itt a műveletek a következőképpen szerepelhetnek:
-- Komplex_Szám.Valós_Rész(),
-- Komplex_Szám.Polar.Arg()

with Komplex_Szám.Polar; use Komplex_Szám;
package Teszt is
...
-- Itt a műveletek a következőképpen szerepelhetnek:
-- Valós_Rész(),
-- Polar.Arg()

with Komplex_Szám.Polar;
use Komplex_Szám; use Komplex_Szám.Polar;
...
-- Itt a műveletek a következőképpen szerepelhetnek:
-- Valós_Rész(),
-- Arg()
```

A szülő törzse (egy **with** segítségével) elérheti és így használhatja is az öröklődési fában szereplő összes leszármazottját. A gyermekcsomag automatikusan (**with** és **use** nélkül) használhatja a szülő összes entitását. Egy gyermek szintén **with** segítségével használhatja az összes testvérének a szolgáltatásait (persze csak a már lefordított testvéreiét). A gyermekcsomagok specifikációjának átlátszatlan (private) része, valamint a törzse látja a szülő átlátszatlan (private) részét. A **with** egy gyermekcsomagra magában foglal egy **with**-et a szülőjére is. Egy csomagra vonatkozó **use** után a gyermekek entitásai ⟨*szülő neve*⟩.⟨*gyermek neve*⟩ formában érhetők el.

Ezek a láthatósági szabályok mindenütt rekurzívan értendők, azaz többszintű hierarchiát építhetünk fel a szülő- és gyermekcsomagokból, és így egy gyermekcsomag átlátszatlan része és törzse az összes ősének átlátszatlan részét látja, de a szülők nem láthatják az ő átlátszatlan részeit.

Egy csomagnak természetesen egynél több gyermeke is lehet. A teljesség kedvéért megjegyezzük, hogy a példákban említett komplex szám csomagot három részben a legjobb megvalósítani: a szülőben az átlátszatlan (private) típust, valamint a négy alapműveletet, az egyik gyermekben a kanonikus alakra, a másikban pedig a trigonometrikus alakra vonatkozó műveleteket érdemes elhelyezni. Így a lehető legkisebb részt kell esetenként újrafordítani, és a hibalehetőségek száma is csökken.

Tehát a nyilvános (public) gyermekcsomag révén elérhető, hogy egy átlátszatlan típus több csomagban is látható legyen, ugyanakkor lehetőséget is ad egy csomag újrafordítás nélküli kiterjesztésére is. Így a **tagged** típus mellett még egy lehetőség van az Ada 95-ben a kiterjesztéssel való programozásra, hiszen a gyermekcsomagokban is elhelyezhetünk átlátszatlan részeket, és ezekben kiterjeszthetjük a szülőben megvalósított típust.

A névkonvenciókkal (⟨*szülő neve*⟩.⟨*gyermek neve*⟩) kapcsolatban is érdemes megjegyezni, hogy ez nem csak egy, a nyelv tervezői által bevezetett formális szabály, hanem nagyon komoly szerepe van a funkcionálisan egybe tartozó elemek azonosításában, amely más, objektum elvű programozást támogató programozási nyelvekben nem nyilvánul meg ilyen egyértelműen.

Nagyobb rendszerek fejlesztésekor előfordulhat, hogy implementációs megfontolásokból részekre akarjuk bontani a feladat megvalósítását, anélkül, hogy mindez az ügyfelek számára láthatóvá válna. Ezenkívül ide kapcsolódik az Ada 83 csomag-szerkezetének egy eddig még nem tárgyalt hiányossága. Az Ada 83-ban a csomag törzsének részekre bontását egyedül alprogramok (eljárások és függvények) segítségével lehetett elvégezni. Bár ezek az alprogramok alegység formájában egymástól függetlenül is lefordíthatók voltak (az elé jük írt **separate** kulcsszó segítségével), a törzsben található külső (alprogramokon kívüli) deklarációk bármilyen változásának hatására az összes alegységet újra kellett fordítani.

Erre a problémára is megoldást kínál az Ada 95-ben bevezetett *átlátszatlan (private) gyermekcsomag*. Az átlátszatlan gyermekcsomag csupán bizonyos láthatósági szabályokban különbözik a nyilvános gyermekcsomagtól, ezért itt csak az eltéréseket ismertetjük. Átlátszatlan gyermekcsomag a hierarchia bármely részében deklarálható, és egy csomag átlátszatlan voltát a specifikáció elején szereplő **private** kulcsszóval jelezzük. Az átlátszatlan gyermekcsomag minden része látja a szülő átlátszatlan részét, beleértve a specifikáció látható részét is. Egy átlátszatlan gyermekcsomag a hierarchiának csak azon részfájában látható, amelynek a gyökere az ő szülője; nem látható egyetlen publikus testvére specifikációjában sem, valamint azon csomagokban sem, amelyek a szülőjének nem leszármazottai. Ezek a szabályok biztosítják, hogy az átlátszatlan gyermekcsomag sem sértheti meg az absztrakciót, hiszen az átlátszatlan részek láthatóvá tétele itt nem megvalósítható, mivel maguk az átlátszatlan gyermekcsomagok sem kívülről, sem az őket esetleg elérő nyilvános (public) testvérek látható részeiben nem láthatók.

Példaként vegyük egy operációs rendszer megvalósításának következő kiragadott részletét:

```
package OS is
  type File_Descriptor is private;
...
private
  type File_Descriptor is new Integer;
end OS;

package OS.Exceptions is
  File_Descriptor_Error, File_Name_Error, Permission_Error: exception;
end OS.Exceptions;

with OS.Exceptions;
package OS.File_Manager is

  type File_Mode is (Read_Only, Write_Only, Read_Write);

  function Open(File_Name: String; Mode: File_Mode) return File_Descriptor;

  procedure Close(File: in File_Descriptor);
...
end OS.File_Manager;

procedure OS.Interpret(Command: String);

private package OS.Internals is
...
end OS.Internals;

private package OS.Internals_Debug is
...
end OS.Internals_Debug;
```

A példában a szülőcsomag tartalmazza a rendszerben használatos típusok deklarációját, a három nyilvános gyermek (két csomag és egy alprogram) pedig a rendszer funkcionális bontását biztosítja. Érdemes megfigyelni az *OS.Interpret()* eljárást, amely az *OS* csomag egy gyermek-alprogramja, és így a törzsében látható a csomag **private** része is. Ami pedig a **private** csomagok láthatóságát illeti, mivel mindkettő az *OS* csomag gyermeke, az *OS* minden gyermekének törzsében láthatók. Másrészt viszont az *OS.Internals* és az *OS.Internals_Debug* csomagok minden része látja az *OS* teljes specifikációját, a látható és a **private** részét is. Ez azonban biztonságos, mivel eme két **private** gyermekcsomag specifikációját egyetlen más csomag specifikációja sem látja, és így az *OS* **private** részének egyetlen definíciója sem válhat láthatóvá ezeken keresztül.

9.2.4. Objektumorientált programnyelvek

Az objektumorientált nyelvek a nyitott és zárt modulok kérdését az öröklődés segítségével oldják meg. Ezen kívül rendszerint még további más nyelvi eszközöket is biztosítanak a modularitás által támogatott információ-elrejtés és -beburkolás, valamint a strukturáltság megvalósítására.

Itt érdemes például a Java nyelv úgynevezett csomag lehetőségét megemlíteni. Minden csomag egy önálló *azonosítóteret* (névtér, namespace) vezet be, amivel egyedivé teszi a benne definiált típusok neveit. Így lehet védekezni a különböző csomagokban definiált egyező nevű típusok keveredése ellen.

Egy csomagnak nyelvi szinten a fordítási egységek adnak tartalmat. A csomaghoz tartozó fordítási egységekben található a csomag kódja: az osztályok és az interfészek, összefoglaló néven a típusok deklarációi.

A Java csomagszerkezete hierarchikus, tehát a csomagok között alárendelő viszony létezhet. Egy csomag tetszőleges számú alcsomagot tartalmazhat. Egy alcsomag is csomag, mindössze annyiban tér el a többitől, hogy megadja, melyik csomagnak képezi részét. Ennek megfelelően egy alcsomagnak is lehetnek alcsomagjai, vagyis a programozó által definiált csomagok faszerkezetet alkotnak.

Az alcsomag nem kapcsolódik erősebben a hierarchiában felette álló csomaghoz, mint egy tetszőleges másik csomag. A csomagok struktúrájának csak egy rendezett felépítés létrehozása a célja, hogy a programozó jobban tudja követni a fejlesztés menetét. Például a láthatósági viszonyok nem javulnak az alcsomag számára, azok is olyanok, mint egy tetszőleges csomag esetében. Viszont a csomagok bevezetésével a láthatóság kérdése finomabban szabályozható: ha nem adunk meg explicit láthatóságot a **private**, **protected** vagy **public** módosítókkal, akkor az egy új láthatósági módot, az úgynevezett csomagvédett láthatóságot jelzi. Az így megjelölt elemek ugyanazon csomagon belül nyilvánosak, míg más csomagból nézve átlátszatlanok.

Minden fordítási egység elején deklarálni lehet azt, hogy a fordítási egység melyik csomaghoz tartozik. Ezt a deklarációt elhagyva a fordítási egység egy névtelen csomaghoz fog tartozni.

A csomagdeklaráció után következnek az importdeklarációk. Egy importdeklaráció lehetővé teszi más csomagokban deklarált publikus típusok egyszerű névvel történő elérését azok kiterjesztett neve helyett. Az importdeklaráció a csomagnak csak arra a fordítási egységére vonatkozik, amelyikben szerepel, vagyis a csomag többi fordítási egységében az importált típusra csak a teljes névvel lehet hivatkozni, vagy ott is importálni kell azt.

Az importálás nem érinti az importált csomagban szereplő importdeklarációkat, vagyis az importált csomag importjaiban szereplő típusokra nem lehet egyszerű névvel hivatkozni, csak egy külön, explicit importálás után. Ez annyit jelent, hogy az importálás nem tranzitív. Az importálás nem vonatkozik az átvett csomag alcsomagjaira, tehát azokat nem lehet az importálás után egyszerű névvel elérni, csak külön importálással.

Egy csomag létrehozásához egyszerűen el kell készíteni a hozzá tartozó fordítási egységeket, és mindegyikben deklarálni kell, hogy az adott egység ehhez a csomaghoz tartozik. Ezek után az importdeklarációkat és típusdeklarációkat kell megírni minden fordítási egységben.

Például tegyük fel, hogy a *gyumolcs.alma* nevű csomag két fordítási egységből áll, a *Jonatan.java*-ból és a *Starking.java*-ból:

```
// Jonatan.java
package gyumolcs.alma; // A csomag egyik fordítási egysége.
public class Jonatan {    // A csomag egyik típusa.
   public Jonatan(){ System.out.println("Jonatan vagyok!"); }
};

// Starking.java
package gyumolcs.alma; // A csomag másik fordítási egysége.
public class Starking {    // A csomag másik típusa.
   public Starking() { System.out.println("Starking vagyok!"); }
};

// Teszt.java
// Tesztprogramunk egy névtelen csomagba kerül.
import gyumolcs.alma.*;          // Igény szerinti típusimport.
import gyumolcs.alma.Starking; // Egyszerű típusimport.

public class Teszt {
   public static void main(String[] args) {
      Object j1, j2;
      j1 = new gyumolcs.alma.Jonatan();
      j2 = new gyumolcs.alma.Starking();
   }
}
```

9.2.5. Funkcionális programnyelvek

A funkcionális programozási nyelvek tulajdonságai – miszerint nincsenek változók (csak úgynevezett *kötések*), nem szerepelhetnek a programokban értékadások, semelyik objektum értéke nem változhat meg a program futása során (amit átlátszóságnak hívnak) és a megszokott szekvencia, elágazás és ciklus helyett a függvénykompozíció, mintaillesztés és rekurzió áll a rendelkezésünkre – könnyen elemezhetővé teszik az ilyen nyelveket. A funkcionális programozási nyelveket deklaratívnak tekintik, szemben a harmadik generációs, úgynevezett imperatív programozási nyelvekkel.

A modularitást támogató eszközöket az ML nyelven keresztül vizsgáljuk meg. Itt a modulok alapvető elemei a szignatúrák és a struktúrák. A szignatúrák struktúrákat specifikálnak, szerepüket tekintve tehát más nyelvek interfészeihez, osztály típusaihoz, csomagspecifikációihoz hasonlíthatjuk őket, míg a szignatúrákat implementáló struktúrák az implementációk, osztályok vagy csomagok szerepét töltik be.

A szignatúra egy programmodul vagy -struktúra leírása, specifikációja. Egy struktúra tartalmazhat mindenféle deklarációt: típuskonstruktorokat, kivételeket, objektumok szimbolikus névhez történő kötését, valamint kifejezetten a modularizációt támogató alstruk-

túrákat és alstruktúrák megosztását. A szignatúra egy struktúra ezen deklarációinak elemenkénti specifikációja. Egy struktúra megfelel egy szignatúrának vagy implementál egy szignatúrát, ha tartalmazza a szignatúrában specifikált, vagy azoknál általánosabb típusú deklarációkat.

Nézzük a következő egyszerű példát, amelyben az a' egy típusváltozó és használata által a sor alábbi specifikációja polimorf a sor elemtípusát illetően:

```
signature QUEUE =
  sig
    type a' queue
    exception Empty
    val empty : a' queue
    val insert : a' * a' queue -> a' queue
    val remove : a' queue -> a' * a' queue
  end
```

Már meglévő szignatúrákat kétféleképpen használhatjuk újra, új szignatúrák definiálásához: beágyazással és specializációval. Nézzünk ezekre egy-egy példát:

```
signature QUEUE_WITH_TEST =
  sig
    include QUEUE
    val is_empty : a' queue -> bool
  end
signature QUEUE_AS_LISTS =
  QUEUE where type a' queue = a' list * a' list
```

A beágyazást tehát a szignatúra bővítésére használhatjuk, míg a specializáció az implementációhoz visz közelebb, az absztrakt típusok reprezentációjának megadásával. Fontos megjegyezni, hogy két szignatúra ekvivalens, ha a specifikációk páronként típusekvivalensek.

A struktúra deklarációk sorozatából álló programegység. Példaként nézzük a fenti QUEUE szignatúra következő polimorf implementációját (a kétlistás reprezentációt az indokolja, hogy az ML-ben beépítetten támogatott lista típus műveleteit tekintve inkább veremként működik):

```
structure Queue =
  struct
    type a' queue = a' list * a' list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (h,t)) = (x::h,t)
    fun remove (nil,nil) = raise Empty
      | remove (h,nil) = remove (nil, rev h)
      | remove (h, x::t) = (x, (h,t))
  end
```

10 Absztrakt adattípusok

Az adatabsztrakció *módszere és ennek következtében az absztrakt adattípusok fontos szerepet játszanak a modern programozási módszertanban (lásd [Jac 75, Dij 76, Dah 72]). Éppen ezért sarkalatos kérdés annak vizsgálata, hogy a típusabsztrakció milyen kívánalmakat támaszt a nyelvek felé, és azok hogyan felelnek meg ezen kihívásoknak. Ebben a részben azt tekintjük át, hogy milyen eszközökkel támogatják a különböző programozási nyelvek az adatabsztrakciót. Ez az eszköztár kiterjed az alprogramokra, kifejezésekre, típuskonstrukciókra és sok más olyan dologra, amelyekkel a könyv más fejezetei foglalkoznak, így ebben a fejezetben különösen sok hivatkozást fogunk találni más részekre.*

Minden feladatot meg lehet oldani FORTRAN-ban. Amit nem lehet megoldani FORTRAN-ban, azt meg lehet oldani Assemblyben. Amit nem lehet megoldani Assemblyben, azt nem lehet megoldani. – Az igazi programozó [Pos 83]

Az idézet tréfás, de mégis igaz. Miért van hát akkor szükség más programozási nyelvekre? Azért, mert a feladatok többségét sokkal könnyebb megoldani más nyelveken. Ennek két oka van. Az egyik, hogy vannak bizonyos célfeladatok megoldására készült nyelvek (például relációs adatbázisok lekérdezésére az SQL nyelv). A másik ok a programozási módszertan fejlődése. Természetesen fegyelmezett programozással szinte minden programozási paradigmát át lehet ültetni bármilyen általános célú programozási nyelvre. Mindazonáltal nyelvi támogatás nélkül ezen megoldások általában nagyon sok mechanikus tevékenységet kívánnak a programozótól. A mechanikus tevékenységek pedig elsődleges forrásai a hibáknak. *Ezért a programozási nyelveknek mindenkor egyik legfontosabb feladata a használt programozási módszertanok aktív támogatása.* Ezáltal mentesíthetik a programozót a mechanikus feladatok végzése alól és így csökkenthetik az emberi figyelmetlenségből adódó hibákat.

10.1. Típuskonstrukciók és adatabsztrakció

A legtöbb programozási nyelv különböző eszközöket nyújt a programozónak ahhoz, hogy meglévő típusokból újabbakat állítson elő. Ezeket a típuskonstrukciós eszközöket használva azonban kötött szemantikával rendelkező típusokat tudunk csak előállítani. A 6. fejezetben részletesen megvizsgáltuk, hogy melyik konstrukciós eszköz pontosan milyen tulajdonságú típust állít elő az összetevő típusokból. Ezek az „előre gyártott" szemantikák azonban ritkán felelnek meg közvetlenül az igényeinknek. A valós világ modellezni kívánt objektumai ritkán direktszorzatok, vektorok vagy éppen halmazok.

Mire használhatók akkor ezek a konstrukciók? Segítségükkel modellezhetjük a valós világ objektumait. Az ember nem direktszorzat, de ha például egy telefonkönyv-alkalmazást készítünk, akkor egy embernek *az alkalmazás szempontjából lényeges* tulajdonságai jól leírhatók egy direktszorzat segítségével. Azaz a gyakorlatban a típuskonstrukciókat nem önállóan használjuk, hanem arra, hogy segítségükkel számunkra fontos típusokat ábrázoljunk.

Az 5.1. részben formálisan is definiáltuk, hogy mi egy típus. Abban a definícióban a típus ábrázolása mindig elemi értékek sorozatával történt ami igaz is, ha elég mélyre ásunk a program szerkezetében. A gyakorlatban a típusok reprezentálásához meglévő típusokat, illetve azokból a nyelv típuskonstrukciós eszközeivel létrehozható típusokat használunk. Ez azt jelenti, hogy meglévő típusainkból lényegében két lépésben hozzuk létre a kívánt új típust. Először egy köztes típust alakítunk ki valamilyen típuskonstrukciós eszköz segítségével. Ez a típus már bírni fogja az általunk választott ábrázolás szerkezetét, de szemantikáját – műveletei – még el fog térni a megvalósítani kívánt típusétól. Így a második lépésben a típuskonstrukció és a kiindulási típusok műveleteinek segítségével meg kell valósítanunk az új típus műveleteit.

Például ha a racionális szám típust vizsgáljuk, ábrázolásnak választhatunk egy direktszorzatot, amely két egész számot tartalmaz. Ehhez fel fogjuk használni a kiválasztott nyelv direktszorzat típuskonstrukcióját, de a kapott típus még nem a racionális szám típus, csupán egy olyan típus, amelyben két egész szám szerepel egymás mellett. Műveletei sem a racionális számok szokásos műveletei, azaz mondjuk az összeadás, kivonás, szorzás és osztás, hanem az egyes komponensek lekérdezése, beállítása stb. Viszont ez a típus már alkalmas arra – amire a korábban rendelkezésre álló egész szám típus önmagában még nem volt alkalmas –, hogy műveletei és az egész számok műveletei segítségével együttesen már ki tudjuk fejezni a racionális szám típus műveleteit.

10.2. Elvárások a programozási nyelvekkel szemben

Amint azt a bevezetőben is említettük, a programozó jogos elvárása, hogy a használt programozási nyelv aktívan támogassa a használt programtervezési módszertant. Az alábbiakban felsoroljuk azokat a területeket, ahol nyelvi támogatást várunk a programozói munka hatékonyabbá és egyszerűbbé tételéhez. Ez a támogatás egyrészt az alábbi elvárásoknak eleget tevő nyelvi eszközök, illetve szintaktikus megkötések meglétét jelenti.

1. *Modularitás*, azaz az egyes típusokat önálló fordítási egységekben lehessen megvalósítani. Ez biztosítja a típusok újrafelhasználhatóságát, valamint a hatékony programfejlesztést, hiszen az egyes modulok könnyedén átvihetők más programokba, és a különböző egységeket más-más programozó is fejlesztheti, nem zavarva egymás munkáját.
2. *Enkapszuláció – egységbe zárás.* A nyelv kezelje összetartozó egységként a típusértékhalmazt és műveleteit.
3. *Az ábrázolás elrejtése.* Ezen eszköz segítségével a nyelv maga biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki. Ez a megszorítás lehetővé teszi az ábrázolás, illetve megvalósítás megváltoztatását anélkül, hogy a változások a programhierarchiában felfelé gyűrűznének.
4. *A specifikáció és az implementáció szétválasztása* külön fordítási egységbe. Ekkor az adott típust használó más modulok a típusspecifikáció birtokában elkészíthetők, függetlenül a tényleges megvalósítástól.
5. *A modulfüggőség kezelése.* A fordítóprogram maga kezelje a modulok közötti kapcsolatokat (egyik használja a másikat stb.).

6. *Konzisztens használhatóság.* A felhasználói és beépített típusok ne különbözzenek a használat szempontjából! Továbbá a típusokat minél inkább a valós világban megszokott, „természetes" módon lehessen kezelni!
7. *Általánosított programsémák támogatása.* A programozó lehetőleg minél általánosabban írhassa meg programjait. A nyelv adjon lehetőséget az ismétlések minimalizálására, nem csak a közvetlen kódismétlések elkerülésére, hanem a magasabb szintű megoldási struktúrák többszöri megírásának elkerülésére. Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.

10.3. Modulokra bontás

A professzionális használatra szánt programozási nyelvek mindegyikében megjelenik a modularizáció támogatása. A modernebb nyelvekben a modulokra bontás alapját egyre inkább a típusokra bontás jelenti, azaz egy modul egy típust valósít meg. Mint azt látni fogjuk a modularizációnak nagy szerepe van az ábrázolás elrejtésében is, mivel a legtöbb nyelvben a modulok határai és a különféle láthatósági kategóriák határai gyakran egybeesnek, a modulok természetes módon korlátozzák a bennük definiált azonosítók láthatóságát. A modularizáció támogatásának eszközeivel részletesebben foglalkozunk a 9. fejezetben.

10.4. Egységbe zárás

Az adatabsztrakciós programtervezés lényege, hogy a típusokat nem csak mint elemek egy halmazát tekinti, hanem a rajtuk értelmezett műveletekkel együtt, egységként használja. Ennek az egységnek a kifejezését jelenti az enkapszuláció, azaz az *egységbe zárás.*

Az egységbe zárás fogalma szorosan kapcsolódik az ábrázolás elrejtéséhez is. Azok az alprogramok jogosultak egy típus belső szerkezetéhez hozzáférni, amelyek a típus műveleteit valósítják meg, azaz részesei az egységbe zárásnak. Minden más alprogram számára a típus átlátszatlan.

Az objetumorientált nyelvekben az egységbe zárásnak különösen fontos a szerepe, mivel az öröklődés során a származtatott típus örökli az őstípus *primitív* műveleteit, vagyis azokat, amelyek a típussal „egységbe vannak zárva". A Java vagy C++ nyelvekben ezek a műveletek lényegében az osztály névterében definiált műveletek, míg például az Ada 95-ben a típus deklarációját követően bevezetett, a típust szignatúrájukban tartalmazó műveletek lesznek ezek.

10.5. Az ábrázolás elrejtése

Az adatabsztrakció egyik legfontosabb eleme az ábrázolás (reprezentáció) elrejtése. Az ábrázolás elrejtése garantálja, hogy a program architektúrájának szintjei egymástól függetlenül megváltoztathatók legyenek. Egy alsóbb szinten a megvalósítás megváltoztatása nem terjed tovább felfelé a struktúrában, mivel a felsőbb szint csupán a használt típus specifikációjától függ.

Az ábrázolás elrejtésének támogatása, hogy a programozási nyelv – általában szintaktikus eszközökkel – kizárja, hogy egy típus használója hozzáférjen a típus reprezentációjához. Mivel a típus reprezentációja rejtett, *átlátszatlannak* is szokták nevezni. Az adatabsztrakciós programtervezés során persze minden típus átlátszatlan kell, hogy legyen. Azt persze tudjuk, hogy végeredményben minden típust bitek sorozata reprezentál, ha elég mélyre ásunk a program szerkezetében, ezért ezt az elnevezést általában az olyan típusokra használjuk, ahol a típus használója számára a típus szerkezete bitek sorozatánál magasabb szinten nem hozzáférhető.

10.5.1. Átlátszatlan típus a C-ben

Nem minden nyelv rendelkezik azonban eszközökkel átlátszatlan típusok megvalósítására. A C nyelvben például nincs ilyen eszköz. Az alábbi példa a komplex szám típus C nyelvű megvalósításának részlete:

```
/* complex.h: */
...
typedef struct __complex_struct {
    double re;
    double im;
} Complex;

extern Complex init(double re, double im);
...

/* complex.c */
...
public Complex init(double re, double im)
{
    Complex z;
    z.re = re;
    z.im = im;
    return z;
}
...
```

Ebben a megoldásban az ábrázolás hozzáférhető a típus használója számára. Ha deklarálunk egy `z` komplex számot, akkor szintaktikailag helyes a `z.re` hivatkozás. Ez azonban azt jelenti, hogy ha a későbbiekben a komplex számok kanonikus alakja helyett szeretnénk inkább polárkoordinátás ábrázolásra áttérni, akkor a korábban helyesen működő programjaink esetleg nem fognak működni, mivel a *Complex* típusnak a továbbiakban nem lesz *re* mezője. Mivel a nyelv itt nem nyújt támogatást, a programozó fegyelmezettségére bízzuk, hogy ne éljen vissza ezzel a lehetőséggel.

Egy másik lehetséges ábrázolás látszólag megoldja ezt a problémát:

```
/* complex.h: */
...
struct __complex_struct;
typedef struct __complex_struct* Complex;

#define COMPLEX_ERROR NULL

extern Complex init(double re, double im);
...


/* complex.c: */
...
struct __complex_struct {
    double re;
    double im;
};

Complex init(double re, double im)
{
    Complex z = malloc(sizeof(struct __complex_struct));
    if (z != NULL) {
        z->re = re;
        z->im = im;
    }
    return z;
}
...
```

Ebben a megoldásban a programozó valóban nem férhet hozzá a *re* vagy *im* komponensekhez. Mégsem rejtettük el azonban a reprezentációt, mivel az most nem a *__complex_struct* struktúra, hanem az arra hivatkozó mutató típus. Ez pedig továbbra is „látható", azaz alkalmazhatók rajta a mutatókra szokásos műveletek. Az eredmény bizonyos szempontból még rosszabb is, mint a korábbi változat, mivel egyrészt ezután a programozónak gondoskodnia kell a létrehozott komplex szám objektumok megszüntetéséről is, hiszen minden *Complex* objektum létrehozása dinamikus memóriafoglalással jár. Másrészt a mutató típusokon értelmezettek a ==, +, - operátorok, ám a jelentésük közel sem a komplex számoknál megszokott egyenlőség, összeadás, kivonás. Hasonlóképp az értékadás művelet működése is eltér az elvárttól. Ez az eltérés pedig, figyelmetlen használat esetén, olyan hibákhoz vezethet, amelyek felderítése nehéz, mivel a fordítóprogram nem tud segítséget nyújtani ehhez.

10.5.2. Ada típusok privát nézete

Az Ada nyelvben már megfigyelhető az ábrázolás teljes elrejtése. A típust definiáló csomag specifikációs részében szerepel ugyan a reprezentáció leírása, de az el van választva a nyilvános azonosítók deklarációjától a specifikációs rész privát részében (a **private** kulcsszó után). A privát részben leírtak logikailag „nem láthatók", azaz a típus használója nem hivatkozhat az ott definiált fogalmakra.

```
package Complex_Numbers is
    type Complex is private;
    function NewComplex(RE: Float; IM: Float := 0.0) return Complex;
    ...
private
    type Complex is record
        RE : Float := 0.0;
        IM : Float := 0.0;
    end record;
end Complex_Numbers;

package body Complex_Numbers is
    function NewComplex(RE: Float; IM: Float := 0.0) return Complex is
    begin
        Return Complex'(RE,IM);
    end New;
    ...
begin
end Complex;
```

10.5.3. CLU absztrakt adattípusok

A CLU nyelvben nagy hangsúlyt fektettek az adatabsztrakció támogatására, így az ábrázolás elrejtésére is nagyon jó eszközöket vezettek be. Az alábbi példa a komplex típust megvalósító modul egy részlete.

```
complex = cluster is newcomplex, re, im ...

    rep = struct[ re : real, im : real ]

    newcomplex = proc (r: real, i: real) returns ( complex )
        return ( up(rep$_(re: r, im: i)) )
    end newcomplex

    re = proc ( c : cvt ) returns ( real )
      return ( rep$get_r( c ) )
    end re
```

```
    im = proc ( c : complex ) returns ( real )
       return ( rep$get_i( down(c) ) )
    end im
...
```

A reprezentációt a `rep` típus definíciója határozza meg. A reprezentáció elrejtése, a konkrét (`rep`) és az absztrakt (`complex`) típusok szétválasztása annyira erős, hogy a nyelv megköveteli közöttük az explicit típuskonverziót. Nem arról van szó tehát, mint az Ada esetében, hogy az ábrázolás a típusnak elrejtett belső szerkezete, hanem ténylegesen egy külön típus, amely történetesen megfelel a megadott típusspecifikációnak, az 5.1. részben definiált megfelelés fogalom szerint.

A fenti példában megfigyelhetjük, hogy az absztrakt és konkrét típusok közötti konverziót kétféle módon végezhetjük el. Az egyik az automatikusan létrejövő `up` és `down` műveletek használata, amelyek rendre a konkrét típusról absztrakt típusra, illetve a fordított irányú konverziót végzik. Ezek a konverziók megfigyelhetők a `newcomplex` konstruktor, illetve a `im` művelet megvalósításában.

A legtöbb művelet működése során a kapott, absztrakt típusú paramétereit konkrét típusra konvertálja, elvégzi a megfelelő átalakításokat, majd az eredményt visszakonvertálja az absztrakt típusra, így bevezettek egy egyszerűsített jelölést. Ha a művelet szignatúrájában valahol a `cvt` kulcsszó szerepel, az azt jelenti, hogy ott a fenti konverziókat automatikusan el kell végezni. Azaz például a `re` művelet paraméterlistájában az aktuális paraméternek absztrakt típusúnak kell lennie, míg a formális paraméter konkrét típusú.

10.5.4. Láthatósági szintek

Az ábrázolás elrejtésének támogatására a nyelvek az összetett típusok komponenseinek láthatóságát gyakorta szintekre bontják, amely szintek meghatározzák, hogy pontosan kik férhetnek hozzá az adott komponenshez. Vannak olyan nyelvek, ahol egyáltalán nincs ilyen jellegű szabályozás, ilyenek például a C, a Pascal vagy Modula-2 nyelv.

Az Ada 83-ban a korábban már látott kétszintű láthatóság használatos. A típus specifikációja nyilvános, minden felhasználó modul hozzáférhet, míg az ábrázolás és a megvalósítás rejtett, csak a típust megvalósító modulban hozzáférhető.

Az objektumorientált nyelvekben leggyakrabban három láthatósági szintet szoktak megkülönböztetni, amelyek a következők:

- *nyilvános (public)* – az adott komponens mindenki számára látható
- *védett (protected)* – az adott komponens csak a leszármazottak számára látható (csak az öröklődést támogató nyelvekben)
- *privát (private)* – az ábrázolás teljesen rejtett része, csak a műveletek megvalósításában használható komponensek

Ezt a háromszintű láthatóságot használja például az Ada 95 vagy a C++ is. A Java a három szint helyett négyet használ, megkülönböztetve a csak az osztályt tartalmazó csomagban látható azonosítók szintjét.

Egyes nyelvekben (például az Eiffelben) a láthatóság még ennél is finomabban szabályozható, a típus megvalósításakor rendelkezhetünk arról, hogy mely osztály (és leszármazottai) férhessenek hozzá az adott komponenshez. A láthatósági szintekről részletesebben lásd a 12. fejezetet.

10.6. A specifikáció és implementáció szétválasztása

Azon nyelvekben, melyek támogatják az „egy modul – egy típus" elvet, gyakran lehetőség van a típusspecifikációnak a megvalósítástól külön, önálló fordítási egységben történő leírására. Ez a szétválasztás három célt szolgál:

- Segít abban, hogy az egyes modulok egymástól függetlenül elkészíthetők legyenek.
- Lehetővé teszi az ábrázolás és megvalósítás észrevétlen megváltoztatását.
- A típus használatához szükséges összes információ elkülönülten rendelkezésre áll, így a megvalósítás lefordított állapotban is szállítható.

A reprezentáció elrejtése sajnos teljes egészében nem lehetséges. Bár a használó modul számára nem szükséges – sőt nem is tanácsos – az ismerete, a fordítóprogramnak tisztában kell lennie azzal, hogy mennyi memóriát kell egy, az adott típushoz tartozó objektum számára lefoglalni. Ezért, ha a specifikációt és az implementációt két fordítási egységre bontjuk szét, gondoskodnunk kell arról, hogy csak a specifikációs rész birtokában a fordító program képes legyen a szükséges memóriafoglalást meghatározni.

C és C++ fejállományok

A C/C++ nyelvekben a specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, amely az adott típust használni akarja, így persze nem beszélhetünk a specifikáció és az implementáció igazi szétválasztásáról, legalábbis nem a nyelv szintjén.

A bemásolás megkönnyítésére a specifikáció egy külön, speciális forrásfájlban – *fejállomány* (header fájl) – leírható és azt egy külön program, az úgynevezett előfordító segítségével beemelhetjük a megfelelő fordítási egységekbe. Ha egy típus specifikációját többször is beírjuk egy fordítási egységbe, az hibát jelent, így ennek kivédéséről a fejállomány megírásakor a programozónak gondoskodnia kell.

A fejállományban el kell helyezni a műveletek specifikációját és az ábrázolást is, amely, mivel egyszerűen bemásolódik a felhasználó modul forráskódjába, teljesen hozzáférhető. A C++ nyelvben a láthatósági osztályok használata korlátozza ugyan az ábrázolás belső szerkezetéhez való hozzáférést, de mivel a fejállomány és a típust megvalósító modul semmilyen – a nyelv által definiált – kapcsolatban nincs egymással, a fejállományt módosítva anélkül érhetjük el a típus belső részleteit, hogy például az implementáció forráskódjához hozzá kellene nyúlnunk.

Leképezés mutatókra

A probléma egyik lehetséges megoldása, hogy az absztrakt adattípus mindig egy mutatóra képeződik le, amely aztán a ténylegesen reprezentáló bájtsorozatra mutat. Ilyenkor a használat helyén a helyfoglalás pontosan meghatározható, anélkül, hogy a tényleges ábrázolásról bármit is elárulnánk, ám a használatot alaposan megnehezíti a bevezetett indirekció és a dinamikus memóriakezelés szükségessége. Erre a megoldásra láthattunk példát a C nyelv kapcsán a 10.5.1. fejezetben, és ezt a megoldást figyelhetjük meg az alábbi, a Modula-2-ből vett példában is.

```
DEFINITION MODULE Complex_Numbers;
    TYPE Complex;
    PROCEDURE NewComplex(R, I: REAL): Complex;
    PROCEDURE Add(Z1, Z2: Complex);
    ...
END Complex_Numbers.

IMPLEMENTATION MODULE Complex_Numbers;
    CONST GUARD = 12345
    TYPE Complex = POINTER TO ComplexStr;
    TYPE ComplexStr = RECORD
        R: REAL;
        I: REAL;
        G: CARDINAL;
    END;

    PROCEDURE NewComplex(R, I: REAL): Complex
    VAR Z: Complex;
    BEGIN
        NEW(Z);
        IF Z#NIL THEN
            Z^.R := R; Z^.I := I; Z^.G := GUARD;
        END;
        RETURN Z;
    END;

    PROCEDURE Add(Z1, Z2: Complex);
    BEGIN
      IF (Z1#NIL) AND (Z2#NIL) AND
         (Z1^.G = GUARD) AND (Z2^.G = GUARD) THEN
          Z1^.R := Z1^.R + Z2^.R;
          Z1^.I := Z1^.I + Z2^.I;
      END;
    END Add;
    ...
BEGIN
END Complex_Numbers.
```

Ez a megoldás lényegében az objektumreferenciákkal dolgozó nyelvek – például a Java, az Eiffel vagy a CLU – működését utánozza, ám az automatikus konstruktorok és destruktorok, valamint a hivatkozásszámlálás és szemétgyűjtéses memóriakezelés hiánya miatt sokkal nehezebben használható és veszélyesebb. Előnye ugyanakkor ennek a megoldásnak, hogy lehetővé teszi a tényleges ábrázolás – azaz a mutatóval hivatkozott adatszerkezet – megváltoztatását anélkül, hogy a modult felhasználó más programrészeket újra kellene fordítani.

Láthatósági területek

Az Ada teljesen más megközelítést választott. Itt a specifikáció és az ábrázolás került egy fordítási egységbe, így persze a reprezentáció fizikailag nincs elrejtve, de a nyelv szintaktikus eszközeivel gondoskodik arról, hogy ezen információkat a típust felhasználó másik programegység írója ne tudja kihasználni. A fordítóprogram viszont pontosan meg tudja határozni az adott típus helyfoglalását. Látszólag ez a megoldás semmiben nem tér el a C és C++ nyelvek fejállományaitól, ám mivel a nyelv egy csomag specifikációs és implementációs részeit összetartozó egészként kezeli, bármelyik megváltozása a csomag újrafordítását teszi szükségessé.

Fizikai szétválasztást nem támogató nyelvek

Más nyelvekben (például az Eiffelben) a fizikai – külön forrásfájlba történő – szétválasztás nem lehetséges – így persze a helyfoglalási probléma sem jelentkezik –, ám a fejlesztőeszköz támogatja a kód többszintű „nézetét", így a felhasználás szempontjából lényegtelen részletek eltakarhatók.

A Java nyelvből ez a lehetőség hiányzik, itt a specifikáció és megvalósítás összemosódik. Ebben az esetben azonban a Java bájtkód szerkezete olyan, hogy abból meg lehet állapítani az adott osztály interfészét, kiválaszthatók a külső használatra szánt metódusok, sőt az öröklődéshez szükséges információk is kinyerhetők belőle, így mégis lehetőség nyílik a megvalósítás lefordított állapotban történő szállítására is.

10.7. A modulfüggőség kezelése

A programok legmagasabb szintű építőkövei a modulok. A program működése ezen modulok interakciója. Minden modul igényel szolgáltatásokat és segítségükkel más szolgáltatásokat valósít meg, így a modulok között egyfajta függőség alakul ki, hiszen például egy modul megváltozása esetén szükségessé válhat a tőle függő modulok újrafordítása stb.

Némelyik programozási nyelv (például a C vagy a C++) teljes egészében a programozóra bízza ezen függőségek kezelését, minden fordítási egységet önállóan kezel. Éppen ezért vannak speciális eszközök (például a make), amelyek kizárólagos feladata ennek a feladatnak a megkönnyítése. Mivel azonban a nyelv ehhez semmilyen támogatást nem nyújt, így ezen megoldások nem tökéletesek, például túl sokszor fordítanak újra valamit, vagy éppen nem veszik észre az újrafordítás szükségességét stb.

Más nyelvekben ezért az ilyesfajta függőségek kezelése a fordítóprogram feladata. Az Ada nyelvben például a **with** utasítással kell megadnunk, hogy milyen más fordítási egységektől függ a szóban forgó modul. Ez garantálja azt, hogy a modul fordítása előtt mindazon

modulok specifikációs részének lefordítására sor kerül (ha szükséges), amelyektől az adott modul függ. A részleteket lásd a 9. fejezetben.

10.8. Konzisztens használat

Talán az egyik legfontosabb elvárás a programozási nyelvvel szemben, hogy ne tegyen különbséget a beépített és a programozó által definiált típusok között a használat szempontjából. Ugyanúgy lehessen összetett típusokat (tömböket, rekordokat stb.) definiálni a segítségükkel, ugyanúgy lehessen változókat definiálni a saját típusokkal, ugyanolyan módon képezhessünk belőlük kifejezéseket, mint a beépített típusokból stb.

Az egységes használatnak egy további vonzata, hogy az új típust be lehessen illeszteni a nyelv logikájába. Ha például az adott nyelvben az a konvenció, hogy egy típust a *Read* művelet olvas be, akkor fontos, hogy a saját típusokhoz is definiálhasson a programozó egy *Read* nevű műveletet, azaz legyen lehetőség a *Read* azonosító túlterhelésére[1].

Egy azonosító *túlterhelése* (overloading) azt jelenti, hogy a programszöveg egy adott pontján az azonosítónak több definíciója is érvényben van. Ilyenkor a fordító a hivatkozási környezetből dönti el, hogy a több lehetséges értelem közül melyiknek megfelelően használja az adott azonosítót. Túlterhelés szinte minden programozási nyelvben van, hiszen például a $3 + 4$, illetve a $3.13 + 4.2$ kifejezésekben a $+$ operátor más-más műveletet jelöl. A fordítóprogram ilyen esetekben a paraméterek (operandusok) típusa alapján határozza meg, hogy az adott azonosítóhoz milyen jelentést kell társítania. Ezt a jelenséget nevezzük *statikus összekapcsolásnak* (a túlterhelésről részletesebben lásd a 7.6. fejezetet). A mi vizsgálatunk szempontjából a legfontosabb kérdés, hogy ezt előidézheti-e a programozó maga is, azaz lehetséges-e, hogy egy, a programozó által definiált azonosítónak a programszöveg valamely pontján több definíciója is érvényben legyen, melyek közül a fordítóprogram az argumentumok típusa alapján választja ki a ténylegesen végrehajtandó példányt.

Nem tekintjük túlterhelésnek a blokkstruktúrájú nyelvekben általában rendelkezésre álló lehetőséget a változónevek átlapolására olyan módon, hogy ha egy x változót definiáltunk az A blokkban, akkor definiálhatunk ezzel megegyező névvel változót az A-ban levő B blokkban, mivel az el fogja takarni az A-beli x-et, arra legfeljebb valamilyen minősített jelöléssel (például $A.x$) hivatkozhatunk. Szintén nem számít túlterhelésnek az objektumorientált nyelvekben az a lehetőség, hogy egy leszármazott osztály megváltoztathatja egy, az ősosztályban deklarált művelet definícióját. Ezt a jelenséget *felüldefiniálásnak* nevezzük és ebben az esetben a hivatkozáshoz tartozó tényleges definíció nem fordítási időben – statikusan –, hanem futási időben – dinamikusan – választódik ki. Ez utóbbi jelenséget *dinamikus összekapcsolásnak* nevezzük és részletesebb leírása a 12. fejezetben található.

A túlterhelés egy speciális esete az operátor-túlterhelés (lásd a 7.6.1. fejezetet), amikor a nyelv operátorait ($+$, $-$, $*$, $/$, $=$, $<$, $>$, *sizeof*, **new** stb.) terheljük túl. Ennek nagy jelentősége van abban, hogy a felhasználói típusokat természetes használatuknak megfelelően használhassuk. Ha például definiálunk egy mátrix típust, akkor szeretnénk az összeadást $+$-szal, a szorzást $*$-gal infix módon jelölni, továbbá az A mátrix i-edik sorának j-edik elemét jó lenne *A[i,j]* alakban elérni, mivel ezt szoktuk meg a matematikában. Kérdés ilyenkor,

[1] Nincs mindig szükség túlterhelésre, mivel az objektumorientált programozási nyelvekben az egyes osztályok gyakran önálló névteret alkotnak, így minden osztályban definiálható a *Read* művelet, túlterhelés használata nélkül.

hogy nyelv megengedi-e az operátor aritásának vagy precedenciájának megváltoztatását. Ez a legtöbb esetben nem lehetséges. Vannak nyelvek, például a Java, amelyek általában megengedik ugyan a túlterhelést, de nem engedik meg az operátorok túlterhelését.

Az operátor-túlterhelést megengedő nyelveknél nagy kérdés, hogy az értékadást mint utasítást vagy mint operátort definiálják-e. Ha ugyanis az értékadás operátor (mint például a C++-ban), akkor megengedett a túlterhelése, míg ha utasítás, akkor persze nem (lásd Ada).

További kérdés az operátorok kapcsán, hogy definiálhatunk-e újabbakat a beépítetteken felül, azaz van-e „szabad operátor". Az Eiffel például lehetővé teszi új, unáris prefix, illetve bináris infix operátorok definiálását.

10.9. Általánosított programsémák

A programozó célja, hogy minél gyorsabban és hatékonyabban készítse el és tartsa karban programjait. Ennek a hatékonyságnak a jegyében készít minél általánosabb típusokat és alprogramokat, hogy azokat mind az éppen aktuális, mind más programjaiban a lehető legszélesebb körben használhassa fel újra. Maga a programozási módszertan is olyan programozási megoldásokat tanít, amelyek minél általánosabban használhatók. Ezt a törekvést a programozási nyelvek is támogatják kisebb-nagyobb mértékben. Ebben a részben ezeket az eszközöket gyűjtjük össze. Az eszközök némelyike olyan megszokott és hétköznapi, hogy talán nem is vesszük észre, hogy ebbe a kategóriába tartoznak. Rájuk nem is térünk ki részletesen, csupán a teljesség kedvéért szerepelnek az alábbi felsorolásban.

Alprogramok

Alprogramok: a programkód egyes – többször, több helyen használt – részleteit alprogram formájában írjuk meg, csökkentve ezáltal a felesleges kódismétléseket, megkönnyítve a karbantartást. Lényegében minden programozási nyelven készíthetünk alprogramokat.

Alprogramok paraméterezése

Alprogramok paraméterezése: az alprogram legfontosabb paramétereit kigyűjtjük és rájuk formális néven hivatkozva írjuk meg az alprogramot. Ezáltal az alprogram sokkal általánosabban használható, a paraméterezés helyes megválasztása esetén sokkal hordozhatóbb lesz. Talán meglepő lehet, hogy a paraméterezhető alprogramokat külön csoportba soroltuk, de ennek oka, hogy nem minden programozási nyelv teszi lehetővé alprogramjaink paraméterezését. Az assembly nyelvek vagy éppen a BASIC lehetővé teszik, hogy alprogramokat hívjunk, ám a hívó és a hívott alprogram között kommunikációt nekünk kell megoldani, például globális változókon keresztül. Az alprogramokról és paraméterezésükről részletes szól a 7. fejezet.

Típusok paraméterezése

Típusok paraméterezése: típusaink paraméterezésével azokat általánosabbá és sokkal szélesebb körben használhatóvá tehetjük.

Ilyen paraméterezett típusok például az Adában a diszkriminánsos rekordok, vagy a határozatlan indexhatárú tömb típusok (lásd a 6.2.3. és a 6.4.3. részt). Mivel ennek megfelelő hatás dinamikus memóriakezeléssel és explicit konstrukciós műveletek használatával is elérhető, viszonylag kevés nyelv támogatja használatukat.

Alprogrammal történő paraméterezés

Alprogrammal történő paraméterezés: alprogramjainkat más alprogramokkal paraméterezhetjük, így például az általános formában kimondott programozási tételek hasonló általánossággal valósíthatók meg.

Gyakori megoldás az alprogramra vonatkozó mutató típusok (lásd az 5.6. részt) használata, bár egyes programozási nyelvek – Ada 83, BASIC – ezt nem támogatják. Az objektum orientált programozási nyelvekben a dinamikus összekapcsolás és speciális, úgynevezett függvényobjektumok segítségével tehetjük ezt meg (lásd a 12. fejezetet).

A dBase, Clipper és FoxPro nyelvekben hasonló hatást érhetünk el makrók használatával. Ezen nyelvek, kihasználva interpretált jellegüket, lehetővé teszik, hogy szövegtartalmú változókat, mint programrészleteket végrehajtsunk, így például egy alprogram nevét paraméterként átadva, a megfelelő makróutasítással meghívhatjuk azt.

Típusokkal történő paraméterezés

Típusokkal történő paraméterezés: programozási tételeink, alapvető adatszerkezeteink és adattípusaink is elvont fogalmakat használnak, felettük definiáltak, azaz csupán a számukra feltétlenül szükséges megszorításokat teszik ezekre a fogalmakra. Ennek következménye, hogy általában nem egyetlen típus elégíti ki ezen megszorításokat, hanem sok. Szeretnénk a fenti tételek, adatszerkezetek, típusok megvalósítását csak egyszer megadni és ezt a leírást általánosan használni az összes, a feltételeket kielégítő típussal.

Ezen igény kielégítésére többféle eszköz kínálkozik. A C nyelvben például nem áll rendelkezésre ilyen eszköz nyelvi szinten, ám hasonló hatások – korlátozott mértékben – elérhetők az előfeldolgozó rendszer makróinak használatával.

Az objektumorientált programozási nyelvekben a hatás elvileg elérhető lenne kizárólag az öröklődés és a többalakúság használatával, hiszen a szükséges közös tulajdonságokat össze lehetne fogni egyetlen absztrakt osztályba – vagy interfészbe –, amely aztán közös őse lehetne az összes, a feltételeknek eleget tevő osztálynak. A megoldás egyetlen hibája, hogy előre – a nyelv alapvető osztályhierarchiájának kialakításakor – ismerni kellene az összes számításba jövő feltételkombinációt.

Ezért a típussal paraméterezést támogató nyelvek valamilyen önálló konstrukciót vezetnek be. Ezekben megadhatunk formális paraméterként típusokat, amelyeket aztán más típusokhoz hasonlóan használhatunk a struktúra belsejében. Amikor a formális paramétereknek aktuális értéket adva létrehozzuk a struktúra megadott típusokhoz tartozó változatát, a struktúra példányosításáról beszélünk.

A C++ nyelvben a típussal paraméterezés lehetőségét a sablonok (template) biztosítják. A sablon alig több, mint valamiféle makróhelyettesítés, ahol a fordító a megadott aktuális

típussal helyettesíti a hozzá tartozó formális paraméter minden előfordulását. Bizonyos minimális szintaxisellenőrzést ugyan végez a fordító, de a formális paraméternél csak annyit határozhatunk meg, hogy az adott paraméter egy típust jelöl, így nem írhatjuk elő a megvalósítandó alprogram vagy típus számára fontos tulajdonságok (például bizonyos műveletek) meglétét. Ekkor persze a fordítóprogram nem is tudja a sablon megírásakor ellenőrizni, hogy a paraméterekre használt műveletek léteznek-e, illetve nem tudja kivédeni a nem megfelelő típussal történő példányosítást. Az ebből származó hibák így a példányosításkor jelentkeznek. Előnye viszont a nyelvnek, hogy a template példányosítása teljesen automatikusan történik.

Az Eiffel programozási nyelvben a sablon neve generic és mint a nyelvben mindenhol, itt is az erősen objektumorientált megközelítés érvényesül. A generic formális paraméterei osztályok lehetnek, ahol a szükséges speciális tulajdonságokat azáltal lehet meghatározni, hogy megadható, mely osztályból kell származnia az aktuális paraméternek. Ez a megoldás annyival jobb a C++-belinél, hogy a sablon belsejében használt műveletek a megadott ősosztály műveletei lehetnek, így a használat helyessége a sablon megírásakor ellenőrizhető. Hátránya a megoldásnak, hogy egyrészt a szükséges közös tulajdonságokat jóval előre tudni kell, hogy a megfelelő közös ős definiálható legyen. Ekkor viszont a problémák jórészt megoldhatók az öröklődés és a polimorfizmus eszközeivel.

Az Ada 95 nyelvben a sablonok paraméterezhetősége sokkal szélesebb körű és rugalmasabb a korábbiaknál. Itt egyrészt lehetőség van az Eiffelhez hasonló, megadott őstípusból származó típussal paraméterezni, de kiköthető, hogy az aktuális paraméter valamilyen típusosztályba (diszkrét típus, felsorolási típus, tetszőleges típus stb.) tartozzon. Megadhatók a használni kívánt további műveletek, amelyek segítségével kikerülhető a kötelező közös ős megléte. Sőt, a programozó által megadott műveletek segítségével a sablon még rugalmasabb használatára nyílik lehetőség, megtehető például, hogy egy rendezési relációt használó sablont az egész számokkal példányosítva nem a szokásos rendezési relációt adjuk meg, hanem például az oszthatóság parciális rendezési relációját. Hiányzik viszont a C++-ban megfigyelhető rugalmasság, amit az automatikus példányosítás biztosít.

Figyeljük meg, hogy míg a C++-ban a típussal való paraméterezésnél a „típus" fogalmat inkább a programozási nyelv szemszögéből használjuk (lásd az 5.1. részt), ahol igazából csak a típusértékhalmaz (vagyis a reprezentáló elemiérték-sorozatok halmaza) számít, addig az Ada és az Eiffel már inkább a programozási módszertan szemszögéből közelít, azaz a típust típusértékek és a hozzájuk tartozó műveletek egységeként kezeli. A sablonokkal részletesen a 11. fejezet foglalkozik.

Paraméterezés magasabb szintű struktúrákkal

Ne gondoljuk, hogy ezekkel a lehetőségekkel kielégítettünk minden elképzelhető igényt. Felmerülhetnek problémák, ahol összetettebb, magasabb szintű struktúrákkal kell paraméterezni. Az Ada 95-ben például sablonnak lehet paramétere másik sablon, pontosabban szólva előírható, hogy az aktuális paraméter a megadott sablon példányosításával keletkezett csomag vagy alprogram legyen.

Ezeken felül elképzelhetők még további igények, például paraméterezés típusok tömbjével, ahol a paraméterként átadandó típusok száma példányosításnál dől el, de az átadott típusok mind egyazon típusosztályhoz kell, hogy tartozzanak.

Ezekre egyelőre még várni kell, de elképzelhető, hogy a mind újabb és újabb programozási nyelvek, követve a programozók igényeit és a módszertan fejlődését, támogatni fogják némelyiket.

10.10. A programozási nyelvek értékelése

Foglaljuk akkor össze még egyszer, hogy milyen kérdéseket kell megválaszolnunk, ha egy programozási nyelv adatabsztrakciót támogató eszközeit vizsgáljuk.

1. *Támogatja-e a nyelv az ábrázolás elrejtését?*
   Láthattunk példát olyan nyelvekre, amelyek igen összetett láthatóságkezeléssel rendelkeznek, ilyenek például a Java, az Eiffel vagy a C++, míg más nyelvekben egyáltalán nincs eszköz típusaink belső szerkezetének elrejtésére. Ezek közé tartozik a C, a Pascal vagy a FORTRAN.
2. *Ha nem támogatja a nyelv az ábrázolás elrejtését, milyen eszközeink vannak a hasonló hatás elérésére?*
   A C, illetve Modula-2 nyelvekben megfigyelhettük, hogy ha egy mutató mögé rejtettük adattípusunkat, akkor legalább részben megőrizhettük az ábrázolás elrejtésének előnyeit, hiszen a típus reprezentációja megváltoztatható maradt, a felhasználó modulok újrafordítása nélkül. Vannak azonban olyan nyelvek, ahol ez a lehetőség sem adott, ilyen például a BASIC nyelv.
3. *Ha a nyelv támogatja az ábrázolás elrejtését, milyen eszközökkel teszi azt?*
   Az adattípus belső szerkezetének elrejtésére számtalan különféle variációt és eszközt figyelhettünk meg a különböző nyelvekben. Közös bennük, hogy a típus belsejében deklarált azonosítókat különböző láthatósági osztályokba sorolják, amelyek meghatározzák azon modulok körét, amelyek hivatkozhatnak az adott azonosítóra. Az Ada 83 nyelv csak két láthatósági osztályt különböztetett meg, a nyilvános, mindenki számára hozzáférhető, illetve a rejtett (*private*) osztályt, amelyet csak a típust megvalósító modul használhat. Az Ada 95, a C++ és még sok más objektumorientált programozási nyelv a klasszikusnak mondható hármas osztályozást támogatja, ahol a korábban említett két kategóriát kiegészíti egy harmadik, a védett (*protected*) láthatósági osztály, amelybe tartozó azonosítók a típust megvalósító modulon kívül a leszármazott típusokat megvalósító modulokban is láthatók. A Java négy osztályt vezet be, külön kategóriába sorolva az azonos csomagba tartozó osztályokat. A másik végletet az Eiffel képviseli, ahol minden egyes azonosítóhoz pontosan meg lehet határozni azon osztályok körét, amelyek hozzáférhetnek.
4. *Támogatja-e a nyelv a modularizációt?*
   A BASIC vagy a Pascal nyelvekben az „egy program – egy modul" elv érvényesült, minden program lényegében egy fordítási egységben helyezkedett el. Ez a megkötés lényegében lehetetlenné teszi, hogy egynél több ember fejlesszen egy programot,

ezért a legtöbb – professzionális felhasználásra szánt – programozási nyelvben van valamilyen eszköz a modularizációra. A modularizáció alapjául az adatabsztrakciós módszertan a típusokra bontást használja, így az ezt támogató nyelvek – például a CLU – általában az „egy típus – egy modul" elvet követik.

5. *Milyen eszközt ad a nyelv a specifikáció és megvalósítás szétválasztására?*
Láthattunk nyelveket, ahol a szétválasztás csak logikai, mint például a Java vagy Eiffel nyelvekben. Az utóbbiban a láthatósági osztályok rugalmassága miatt a típus külső interfésze felhasználóról felhasználóra változhat. A Modula-2 és Ada nyelvekben viszont a specifikációs és implementációs részek fizikailag is ketté vannak választva, két külön fordítási egységet képeznek. A CLU nyelvben az absztrakt típus – specifikáció – és a konkrét típus – megvalósítás – teljesen különálló típusoknak számít.

6. *Hogyan kezeli a nyelv a modulok közötti függőségeket?*
Némely nyelvek ezt egyáltalán nem kezelik. Ilyenek például a C és a C++, amelyeknél a függőségek leírására egy külön nyelvet és az ahhoz tartozó értelmezőt (`make`) használhatunk. Az Ada és Modula-2 nyelvekben itt is erős támogatást figyelhetünk meg, a függőségeket a modulok rögzítik a nyelv eszközeivel és a fordítóprogram kezeli azokat.

7. *Van-e különbség a beépített és a programozó által definiált típusok között?*
A saját típusoknak a beépítettekkel megegyező használata a legfontosabb szempont az adatabsztrakciós módszertan támogatásában. A konzisztens használat azonban sokféle eszköz együtteseként állhat elő, amelyek függenek a nyelv természetétől, a benne használt konvencióktól.

8. *Felhasználhatjuk-e saját típusainkat típuskonstrukciókban vagy újabb típusok reprezentációjában?*
A legtöbb nyelv támogatja ezt, kivételt jelentenek természetesen azon nyelvek, amelyekben saját típusok létrehozására sincs lehetőség (ilyen például a BASIC), illetve bizonyos scriptnyelvek. A Perlben például a hasítótáblák csak skalár értékeket és karakterláncokat tartalmazhatnak.

9. *Átadhatók-e paraméterként a saját típusok?*
Saját adattípusaink reprezentálására általában valamilyen típuskonstrukciót használunk. Bizonyos nyelvekben – például az ALGOL 60-ban – az összetett típusok nem adhatók át alprogramnak paraméterként, így saját típusaink sem, ha ilyen ábrázolást választunk. Ezek a nyelvek azonban elég ritkák, a nyelvek többségében a saját típusok szabadon átadhatók paraméterként.

10. *Van-e a nyelvben túlterhelés?*
A konzisztens használhatóság egyik alapvető követelménye, hogy az azonos jellegű műveleteket azonos néven nevezhessük. Az Ada nyelvben például egy típust általában a *Get* művelettel lehet beolvasni egy szöveges fájlból. A konzisztens használhatóság megköveteli, hogy saját típusainkhoz is definiálhassunk *Get* műveletet, ez azonban a *Get* azonosító túlterhelését igényli. Ugyanakkor az objektumorientált nyelvekben gyakori, hogy egy osztály önálló névteret alkosson, így a benne definiált metódusok nem terhelik túl más osztályok hasonló nevű metódusait, a konzisztens

használhatóság mégis megmarad (például Java). A C programozási nyelvben viszont nincs lehetőség az azonosítók túlterhelésére, így kénytelenek vagyunk a különböző típusok azonos műveleteit különböző néven nevezni.

11. *Megengedett-e az operátor-túlterhelés?*
A túlterhelés speciális esete az operátorok túlterhelése, ami természetesen nagyon sokat tud hozzáadni a konzisztens használhatósághoz, hiszen a matematikai képletek egy az egyben átültethetők a programszövegbe, amely így sokkal olvashatóbb, érthetőbb lesz. Ennek ellenére vannak olyan, a túlterhelést általában támogató nyelvek, amelyek nem támogatják az operátorok túlterhelését. Ilyen például a Java.

12. *Definiálható-e a nyelvben újabb operátor?*
Az Eiffel nyelv a fentieken túl még azt is lehetővé teszi, hogy újabb, prefix vagy infix operátorokat definiáljunk típusunkhoz. Ez a lehetőség azonban igen ritka.

13. *Készíthetők-e eljárás alprogramok a nyelvben?*
Az alprogramok legegyszerűbb fajtája az eljárás, szinte minden nyelvben van lehetőség ilyenek készítésére. Speciális eset a C nyelv, amelyben eredetileg minden alprogram függvény volt, de nem volt kötelező felhasználni a visszatérési értékét.

14. *Készíthetők-e függvény alprogramok a nyelvben?*
A legtöbb programozási nyelvben erre is van lehetőség, de kivételként meg lehet említeni a BASIC nyelvet, amelyben nincs függvényjellegű alprogram.

15. *Paraméterezhetők-e az alprogramok objektumokkal a nyelvben?*
A válasz legtöbbször erre a kérdésre is igen, de láthattunk kivételt erre is, például a BASIC nyelvet vagy az assembly nyelveket.

16. *Definiálható-e paraméteres típus?*
Sok programozási nyelvben erre nincs lehetőség. Az objektumorientált nyelvekben hasonló hatást érhetünk el, ha megfelelően paraméterezhető konstruktorokat és dinamikus memóriakezelést használunk. Az Ada nyelvben azonban láthattunk példát megszorítás nélküli típusokra, amelyeknél az objektum létrehozásakor határozhatjuk meg a paraméter értékét és az befolyással lehet az objektum méretére, belső szerkezetére.

17. *Paraméterezhetők-e az alprogramok alprogramokkal a nyelvben?*
A legtöbb nyelvben erre is van lehetőség, a szokásos megoldás az alprogramra hivatkozó mutató átadása paraméterként. Kivételt jelent például az Ada 83, amelyben ilyen mutatókat nem lehetett definiálni, vagy éppen a BASIC nyelv. Az objektumorientált nyelvekben úgynevezett függvényobjektumok és a dinamikus összekapcsolás segítségével érhetünk el hasonló hatást.

18. *Van-e lehetőség típussal paraméterezni?*
Igen sok nyelvben, még a modernekben is – például Java – nincs lehetőség erre. Ugyanakkor sok nyelvben találhatunk lényegesen különböző megoldásokat, például a C++, az Eiffel vagy éppen az Ada sablonjait. Érdemes megvizsgálni, hogy milyen eszközökkel specifikálhatjuk a formális paraméter típusokat, milyen előírásokat tehetünk és mennyire rugalmasak az így megfogalmazható leírások.

Feladatok

1. Valósítsa meg többféle nyelven a racionális szám típust! Ügyeljen arra, hogy a megvalósítás minél jobban illeszkedjen a nyelv típusrendszerébe! Hasonlítsa össze a kapott megoldásokat!

2. Készítsen összegző eljárást C, C++, Eiffel és Ada nyelven! Milyen mértékben tehető általánossá a megoldás az egyes nyelveken?

11 Típussal való paraméterezés

A típussal való paraméterezhetőség lehetővé teszi általánosan megírt, rugalmasan viselkedő, széles körben használható programegységek készítését. Ebben a fejezetben a típussal való paraméterezhetőség és a polimorfizmus fogalmával ismerkedünk meg. Áttekintjük a polimorfizmus aleseteinek egy lehetséges csoportosítását, és azt, hogy az egyes programozási nyelvek hogyan támogatják ezek megvalósítását.

A programfejlesztés során gyakran merül fel annak igénye, hogy általánosan megfogalmazott, széles körben alkalmazható megoldásokat adjunk. Különösen igaz ez a könyvtári egységek fejlesztésére. Megfelelő paraméterválasztással már az alprogramok is alkalmasak több hasonló feladat megoldására (lásd a 7.1. részt), de a felhasználhatóságnak korlátot szab, hogy az alprogramok paraméterei nem lehetnek típusok. Nincs tehát lehetőség arra, hogy különféle típusokon megadott, de egyébként nagyon hasonló feladatok megoldására egyetlen alprogramot írjunk. Erre a problémára kínál megoldást a *típussal való paraméterezhetőség*.

A típusparaméterrel ellátott alprogramokat *polimorfnak*, azaz *többalakúnak* nevezzük. Ennek oka az, hogy ezek az alprogramok többféle típuskörnyezetben használhatók, sőt előfordulhat az is, hogy a különböző típusokhoz különböző implementáció tartozik. A típusparaméterrel ellátott alprogramokat gyakran *sablonnak* is nevezzük.

A típussal való paraméterezhetőség már említett előnye az, hogy az egymáshoz kellőképpen hasonló, de különféle típusokon megadott feladatokhoz egyetlen megoldást adhatunk. Ez a kód nagyfokú újrafelhasználhatóságát teszi lehetővé, elősegítve ezzel tömör és alaposan tesztelt programkönyvtárak létrehozását, illetve feladatcsoportok megoldását az alprogramcsokrok írásához szükséges programozói munka jelentős csökkentésével.

Vezérlési absztrakció

Eddig csak az alprogramok típussal való paraméterezhetőségéről esett szó. Ez a nyelvi elem tulajdonképpen *vezérlési absztrakciót* tesz lehetővé, azaz konkrét típusokra specializált alprogramok helyett típusfüggetlen, általános megoldásokat adhatunk. Ilyen megoldás készíthető például akkor, ha különböző alaptípusú tömböket szeretnénk rendezni. Lehetséges természetesen az is, hogy minden elemtípushoz külön rendező alprogramot írunk, de elegánsabb, ha egyetlen rendező alprogramot készítünk és ezt paraméterezzük az elemtípussal. Pontosan ezt szemlélteti a következő Ada példa:

```
generic
   type Elem is private;
   type Index is (<>);
   type Vektor is array (Index range <>) of Elem;
   with function "<" (X, Y : Elem ) return Boolean is <>;
procedure Rendez (V : in out Vektor );
```

Azt, hogy az alprogramot típussal fogjuk paraméterezni, a **generic** kulcsszóval jelezhetjük a fordítónak. A **generic** kulcsszó után soroljuk fel az alprogram polimorffá tételéhez szükséges információkat. A tényleges típusparaméter az *Elem* és az *Index* lesz, de a *Vektor* is típusparaméter, hiszen több vektor típus is lehet a példányosítás során aktuális paraméter. Például:

```
type V1 is array ( Integer range <> ) of Integer;
type V2 is array ( Integer range <> ) of Integer;
```

Itt *V1* is és *V2* is lehet aktuális paraméter, és ezek különböző típusok az Adában használatos név szerinti típusekvivalencia miatt. Látható tehát, hogy egy polimorf alprogramnak egyszerre több *típusparamétere* is lehet. A **private** kulcsszó azt jelzi, hogy az *Elem* által képviselt típus szerkezetéről semmilyen feltételezéssel sem élünk (attól eltekintve, hogy létezik rá értékadás és egyenlőségvizsgálat.) Az *Index* típusnak diszkrét típusnak kell lennie, ezt jelzi a *(<>)*. A *Vektor* olyan tömb, aminek ez az *Elem* az alaptípusa, indexhalmaza pedig az *Index* típus egy tetszőleges intervalluma (ezen utóbbira utal a **range** <>). Az alprogram ezen az általános *Vektor* típuson fog működni. A **with function** "<" kezdetű sor azt jelzi, hogy a *Rendez* törzsében szükségünk lesz egy, az *Elem* típuson működő < (kisebb) operátorra. Ez azt jelenti, hogy a rendező alprogramunk működtetéséhez meg kell adnunk az *Elem* típus mellett egy elemeket összehasonlító műveletet is.

A alprogramsablon a *buborékrendezést* valósítja meg. Az alprogramban használt *Vektor* típus és a < művelet absztrakt, hiszen ábrázolásuk nem ismert.

```
procedure Rendez (V : in out Vektor ) is
   temp: elem;
 begin
   for J in reverse Index'Succ(V 'First)..V'Last loop
     -- A vektor utolsó elemétől visszafelé a másodikig.
     for I in V'First..Index'Pred(J) loop
       -- A vektor első elemétől a J előtti eleméig.
       if not (V(I)< V(Index'Succ(I))) then
         -- Ha a két szomszédos elem sorrendje nem megfelelő, akkor csere.
         temp:=V(I);
         V(I):=V(Index'Succ(I));
         V(Index'Succ(I)):=temp ;
       end if;
     end loop;
   end loop;
 end Rendez;
```

Hogyan kell használni ezt a sablont? Először példányosítjuk (lásd a 11.4. részt), így kapunk az alprogramsablonból egy „igazi" alprogramot. A példányosítás során a formális sablonparamétereknek aktuális paramétereket feleltetünk meg.

```
type Int_Vektor is array (Integer range <>) of Integer;
procedure Int_Rendez is new Rendez(Integer, Integer, Int_Vektor, "<");
```

Ezután az *Int_Rendez* eljárást használhatjuk *Int_Vektor* típusú változók rendezésére. Például, ha *A* ilyen típusú, akkor az alábbi alprogramhívást írhatjuk:

```
Int_Rendez(A);
```

A C++-ban sablonokat a *template* szerkezettel készíthetünk. Például egy adott függvénysablonnak lehetnek paraméterei, amelyeket hagyományos típusként vagy típusnévként használ a függvény törzse. Ezt szemlélteti a következő példa [Str 00].

```
template <class Elem>
void rendez(vector<Elem>& v)
{
    const size_t n = v.size();

    for (int gap = n/2; 0 < gap; gap /= 2) {
        for (int i = gap; i < n; i++) {
            for (int j = i-gap; 0 <= j; j -= gap) {
                if (v[j + gap] < v[j]) {
                    // A v[j] és v[j + gap] felcserélése.
                    Elem temp = v[j];
                    v[j] = v[j + gap];
                    v[j + gap] = temp;
                }
            }
        }
    }
}
```

A példa az ismert *Shell-rendezést* [Knu 87] valósítja meg tetszőleges *Elem* elemtípusú tömbre. Az *Elem* paraméterhez tartozó érték olyan típus lehet, amelynek van értékadás művelete (az *Elem temp = v[j]* miatt), másoló konstruktora (a *v[j] = v[j + gap]* miatt) és egy < összehasonlító operátora (a *v[j + gap] < v[j]* miatt). Tehát az *Elem* helyére írhatunk például **int**-et vagy *string*-et is. A rendező algoritmus megvalósításához csupán a megfelelő típusműveletek meglétére van szükségünk, a konkrét típus ismerete itt még nem fontos. Minden típusra, amely teljesíti a fenti feltételeket, használhatjuk majd a *rendez* rendezést, nem kell azt újra és újra megírni. Sőt, függvénysablonok esetén még explicit példányosításra sincs szükség. Ha egész számokból álló vektort szeretnénk rendezni, írjuk egyszerűen ezt:

```
vector<int> v;
...
rendez(v);
```

Vegyük észre, hogy az Ada nyelvben a típusparaméterekre való megszorítások már a típussal paraméterezett egység specifikációjában szerepelnek, a C++-ban azonban csak a függvény törzséből vagy a függvény dokumentációjából tudhatjuk meg, hogy az *Elem* típusnak milyennek is kell lennie.

Végül nézzünk meg egy funkcionális nyelven írt példát is, ami a beszúrásos rendezés algoritmusát valósítja meg [Pla 99]. Legyen a nyelv mondjuk a Clean:

```
isort :: [a] -> [a] | Ord a
isort [ ] = [ ]
isort [a:x] = Insert a (isort x)

Insert :: a [a] -> [a] | Ord a
Insert e [ ] = [e]
Insert e [x:xs]
   | e <= x   = [e,x:xs]
   | otherwise = [x:Insert e xs]
```

A példában tulajdonképpen két polimorf függvényt is használunk. Az `Insert` egy a típusú elemet szúr be egy a típusú elemekből álló, már rendezett listába úgy, hogy a rendezést ne rontsa el. Ehhez szükséges, hogy az a típusra létezzen rendezési reláció (<), ezt jelzi az `Ord a` a függvénydeklaráció végén. Az `isort` az `Insert` műveletet használja a beszúrásos rendezés megvalósításához. Mivel az `Insert` művelet csak olyan a típusra működik, amelyre létezik < művelet, az őt használó `isort` függvényben is meg kell követelnünk, hogy az a típus rendelkezzen ilyen művelettel.

Adatabsztrakció

A típussal való paraméterezhetőség fogalma nem merül ki az alprogramok típussal való paraméterezésében. Felmerülhet annak igénye is, hogy típusokkal paraméterezett, általános típusokat hozzunk létre. Ezek közös tulajdonsága az, hogy a típusparaméter konkrét megvalósításának ismerete nélkül fogalmazhatjuk meg a típusműveleteket. A típus tehát *absztrakt*, a hozzá tartozó *konkrét típus* csak példányosításkor vagy végrehajtáskor derül ki, azaz amikor a típusparaméternek valamilyen tényleges típust feleltetünk meg. Ekkor az absztrakt típusunk is konkrét típussá válik, például a következőkben ismertetett absztrakt *Verem* típus az *Elem* típusparaméter tényleges értékének megadása után valamilyen konkrét típuson működő veremmé válik. Erről bővebben a 11.4. részben lesz szó. Először nézzük az Ada nyelven írt változatot:

```
generic
   type Elem is private;
   Max : Integer;
package Vermek is

   type Verem is limited private;

   procedure Push (V : in out Verem; X : in Elem );
      -- Elem behelyezése a verem tetejére.
   procedure Pop ( V : in out Verem; E : out Elem );
      -- Legfelső elem kiemelése a veremből.
```

```
   function Top (V : Verem) return Elem;
      -- Legfelső elem értékének lekérdezése.

   function Is_Empty (V : Verem ) return Boolean;
      -- Üres-e a verem.

   function Is_Full (V : Verem ) return Boolean;
      -- Tele van-e a verem.

   Ures, Megtelt : exception;
      -- Kivételek annak az esetnek a kezelésére, amikor egy üres
      -- veremből próbálnak elemet kivenni vagy értéket lekérdezni,
      -- illetve egy már megtelt verembe próbálnak elemet betenni.

private
   type Tomb is array (1..Max) of Elem;

   type Verem is
      record
         Adatok    : Tomb;
         Veremteto : Natural := 0;
      end record;

end Vermek;
```

Itt is a **generic** kulcsszó jelzi, hogy egy polimorf programegység (jelen esetben csomag) specifikációja következik. A **generic** kulcsszó után soroljuk fel a programegység azon paramétereit, amelyek a polimorfizmushoz kellenek. Jelen esetben egy olyan általános vermet készítettünk, amelynek elemtípusa tetszőleges olyan típus, amire létezik értékadás és egyenlőségvizsgálat (a **private** kulcsszó ezt jelzi), és a veremben egyidejűleg tárolható elemek maximális száma, *Max* is paraméter[1]. Ezután következik a **package** kulcsszó, melyet követően a sablon csomag által exportált *Verem* típust és a rajta értelmezett műveleteket soroltuk fel. (A most megadott *Verem* típushoz *csak* az itt leírt műveletek segítségével férhetünk hozzá, mivel a típust **limited private**-ként definiáltuk, megtiltva ezzel a vermekre vonatkozó értékadást és egyenlőségvizsgálatot. Lásd még a 6.1.2. részt). Az *Ures* és a *Megtelt* kivételek (**exception**); ezek használatáról a a 8. fejezet szól. A sablon csomag specifikációjának átlátszatlan részében (a **private** kulcsszó után) adjuk meg a verem (külvilág elől elrejtett) reprezentációját: példánkban a vermet egy tömb segítségével valósítjuk meg. A verembe pakolt elemek az *Adatok* tömbbe kerülnek, a legfelső elem helyét a *Veremteto* tömbindex adja meg.

[1] Lehetett volna szebb megoldást is adni, amelyben a verem maximális mérete nem a csomagnak, hanem a verem típusnak lett volna a paramétere, és a reprezentációhoz úgynevezett diszkriminánsos rekord típust használtunk volna. Az általunk választott megoldás azonban jobban kiemeli a sablonok lehetőségeit, és ebben a fejezetben ez volt a döntő szempont.

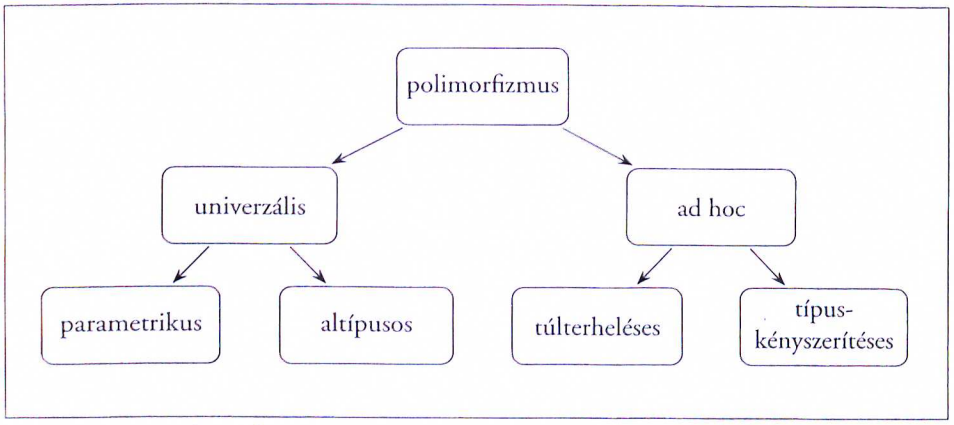
A sablon csomag törzsében kell megadnunk a típusműveletek *absztrakt* megvalósítását. A megvalósítás azért absztrakt, mert a tényleges elemtípust nem ismerjük. Mivel a vermet reprezentáló *Tomb* típust *Elem* típusú elemek tömbjeként adtuk meg, maga a *Tomb* típus is absztrakt lesz.

```
package body Vermek is
   procedure Push (V: in out Verem; X: in Elem) is
   begin
      if V.Veremteto < Max then
         V.Veremteto := V.Veremteto + 1;
         V.Adatok(V.Veremteto) := X;
      else
         raise Megtelt;
            -- Teli verembe próbáltak meg még egy elemet betenni.
      end if;
   end Push;
   procedure Pop (V: in out Verem; E: out Elem) is ...
   function Top (V : Verem) return Elem is ...
   function Is_Empty (V: Verem) return Boolean is ...
   function Is_Full (V: Verem) return Boolean is ...
end Vermek;
```

A példában csak a *Push* művelet (absztrakt) megvalósítását részleteztük: az *X* elem elhelyezéséhez először azt kell ellenőriznünk, hogy van-e még hely a veremben, azaz a *Veremteto* értéke kisebb-e a *Max*-nál. Ha igen, akkor a *Veremteto*-t eggyel nagyobb értékre állítjuk, és a tömbbe betesszük *X*-et az új *Veremteto* által mutatott helyre.

A C++-ban ugyanez a verem a következőképpen néz ki:

```
class VeremMegtelt {}; // Kivételként használt osztályok.
class VeremUres {};

template<class Elem, int max> class Verem {
  Elem adatok[max];   // Az elemek az alapértelmezett konstruktor
  int veremteto;      // szerint kapnak kezdőértéket a tömbben.
public:
  Verem() : veremteto(0) {}
  void push(const Elem& e) throw (VeremMegtelt) {
    // Másoló konstruktor teszi a tömbbe az elemet.
    if (veremteto < max) { adatok[veremteto++] = e; }
    else { throw VeremMegtelt(); }
  }
  Elem pop() throw (VeremEmpty) { ... }
  Elem top() const throw (VeremEmpty) { ... }
  bool isEmpty() const { ... }
  bool isFull() const { ... }
};
```



11.1. ábra: A polimorfizmus aleseteinek egy csoportosítása

A C++ nyelvben a paraméteres típusnak az osztálysablon felel meg. Egy **template** osztályt lényegében ugyanúgy készítünk el, mint egy hagyományos C++ osztályt: nyelvtanilag az a különbség, hogy az előbbit a **template** kulcsszó vezeti be, amit <> zárójelek között a típus paraméterei követnek vesszővel elválasztva. Ebben a példában a veremnek két paramétere van: a veremben tárolt elemek típusa (*Elem*) és a verem maximális mérete (*max*). A megadott absztrakt megvalósítás épít arra, hogy az *Elem* típusnak van alapértelmezett konstruktora és másoló konstruktora, de ez a specifikációban nem jelenik meg, mint ahogy a 355. oldal példájánál sem jelent meg az *isort* specifikációjában a < művelet.

A **template** paramétere lehet típus (osztály, struktúra), primitív típus (**int**) és másik **template** egyaránt. A **template** belsejében az *Elem* nevet úgy használhatjuk, mint bármilyen típusnevet. Például szerepelhet tömb elemtípusaként, visszatérési értékként, lehet rá vonatkozó mutatót deklarálni, műveleteit meghívni stb.

Funkcionális nyelveken is készíthetünk absztrakt adattípusokat. Erről a lehetőségről bővebben a 16.4.2. rész szól.

11.1. Polimorfizmus

A polimorfizmus nagyon általános fogalom, ezért célszerű a polimorfizmus aleseteit valamilyen módon csoportosítani és a csoportok tulajdonságait külön tárgyalni. Számos ilyen csoportosítás létezik, mi a Cardelli-Wegner-félét [Car 85] fogjuk használni. A felosztás szerint a polimorfizmusnak két nagy csoportja van, az *univerzális* és az *ad-hoc* polimorfizmus. Ezeken belül további két-két alcsoportot különböztetünk meg, mint az a 11.1. ábrán látható.

Az *univerzális polimorfizmus* általában típusok végtelen halmazán működik, az egyetlen megkötés az, hogy a típusok valamilyen közös tulajdonsággal rendelkezzenek. Ilyen például az a függvény, ami egy halmaz maximumát számítja ki. Ennek minden olyan halmaz típusra működnie kell, amely elemtípusához létezik összehasonlító operátor (<). A megfelelő összehasonlító művelet megléte lesz a felhasználható típusok közös tulajdonsága.

Az olvasóban felmerülhet az a kérdés, hogy miért végtelen azon típusok halmaza, amiken a függvény működik. A válasz az, hogy a függvény nemcsak minden meglévő, összehasonlító művelettel rendelkező típusra működik, hanem azokra is működni fog, amelyeket csak később definiálunk. Ha tehát definiálunk például egy *Ember* típust, ahol a < operátor az embereket az életkoruk alapján hasonlítja össze, akkor a típussal paraméterezett függvényünket nyugodtan használhatjuk embereket tartalmazó halmazokra is. Könnyen látható, hogy tetszőleges számú új típust megadhatunk és függvényünk mindegyikre működni fog. Ennek oka az, hogy a függvénytörzs megadásához nem kell ismernünk a típus reprezentációját, elég, ha tudjuk, hogy létezik a halmaz elemtípusához < operátor; tehát a függvényünk potenciálisan végtelen sok típusra működik.

Az univerzális polimorfizmus neve abból ered, hogy a különböző típusokhoz ugyanazt a kódot rendeli, azaz egyetlen (univerzális) *absztrakt megvalósítást* kell megadnunk, ami aztán az összes (megfelelő tulajdonságú) típusra alkalmazható lesz. Ezzel szemben az *ad-hoc* polimorfizmus csak típusok véges halmazán működik. Nézzünk például egy kiíratő függvényt! Nyilvánvaló, hogy a különböző típusokhoz tartozó értékeket más és más módon szeretnénk kiíratni. Az egész számokat például elég azonnal karaktersorozattá alakítani, egy rekord kiíratásánál azonban jobb, ha valamilyen formázást is alkalmazunk. Például:

```
nev:       Gipsz Jakab,
szul_ev:   1990,
szul_hely: Szombathely
```

Látszik tehát, hogy ebben az esetben nem alkalmazható ugyanaz a kód, a különböző típusokhoz más és más megvalósításra van szükség. Mivel csak véges számú típushoz tudjuk elkészíteni a függvény a típushoz tartozó megvalósítását, ezért itt nem igaz, hogy a függvény végtelen sok típusra működik. A felhasználható típusok halmazának elemei általában egymástól teljesen függetlenek, itt nem követeljük meg az azonos szerkezetet. Minden típushoz külön kód tartozik, ez egyben azt is jelenti, hogy a különböző típusokra való példányosítás jóval több programozói munkát jelent, mint az univerzális polimorfizmus esetében, de a különféle típusokra specializált programkódok gyakran hatékonyabbak, mint az általánosan megfogalmazott absztrakt megvalósítások.

A következőkben részletesen tárgyaljuk a polimorfizmus különböző formáit. Mielőtt erre rátérnénk, lássuk, milyen nyelvek vagy nyelvcsaládok támogatják ezeket!

A *parametrikus polimorfizmust* sok funkcionális nyelv, illetve bizonyos formáját például az Ada vagy a C++ támogatja (lásd a 11.1.1. alfejezetet). Az *altípusos polimorfizmus* leginkább az objektumorientált nyelvekre jellemző. A *túlterheléses* és *típuskényszerítéses polimorfizmus* használata nagyon elterjedt, a modern programozási nyelvek többsége támogatja ezeket.

11.1.1. Parametrikus polimorfizmus

A *parametrikus polimorfizmus* a polimorfizmus legteljesebb formája. Parametrikusan polimorf például az előbb említett halmazon való maximumkeresés. A parametrikusan polimorf

programegységek gyakorlatilag minden olyan típuson működnek, amely megfeleltethető a típusparaméternek. Alkalmazáskor a típusparaméter aktuális értéke dönti el, hogy milyen típusúak a *polimorf argumentumok* (itt polimorf argumentumnak nevezünk mindent, ami a szóban forgó programegység belsejében polimorf lehet, például függvényparaméterek, típusok stb.). A már előbb említett példában a polimorf argumentum a halmaz, aminek a maximumát keressük: tényleges típusa nyilván a típusparamétertől függ, hiszen ez fogja meghatározni, hogy miknek a halmazával dolgozunk. A típusparaméter nyelvtől függően lehet *implicit* vagy *explicit*. Implicitnek nevezzük a típusparamétert akkor, ha a programozónak a polimorf programegység használatakor nem kell külön megadnia azt, hogy a típusparaméter milyen értéket vesz fel. A típusparaméter explicit, ha minden használatkor egyértelműen meg kell adni a paraméter értékét. A funkcionális nyelvek általában implicit, a nem funkcionális nyelvek explicit típusparamétert használnak. Sajnos ahhoz, hogy egy nyelv támogatni tudja a parametrikus polimorfizmust, valamilyen *egységes típusábrázolás* szükséges (például minden típust mutatókkal ábrázolunk), ami esetenként a hatékonyság csökkenéséhez vezethet.

Nézzünk néhány konkrét példát parametrikus polimorfizmusra! Az egyszerűség kedvéért nem halmaz, hanem lista vagy tömb maximumát keressük. A legegyszerűbb talán egy funkcionális nyelven írt változat. Tekintsük ezt a Clean függvényt.

```
maxker :: [a] -> a | < a
maxker [x]  = x
maxker [x:xs]
   | x < m    = m
   | otherwise = x
   where
      m = maxker xs
```

A példa minden olyan listára működik, amelynek elemtípusára létezik a < művelet. Az a egy *típusváltozó*, helyére bármilyen típus helyettesítődhet, ha van hozzá < művelet. Maga a törzs két részre osztható: az első eset (`[x]`) az, amikor a listának csak egy eleme van. Ekkor nyilván ez lesz a maximum is. Ha a lista több elemű, akkor meg kell vizsgálni, hogy az első elem nagyobb-e, mint a lista maradék részének maximuma; ha igen, ez lesz az új maximum. Látható, hogy a függvény rekurzívan hívja önmagát addig, amíg a lista el nem fogy.

Az alábbi Ada alprogramsablon ugyanezt a funkcionalitást valósítja meg tömbökre:

```
generic
  type Elem is private;
  type Index is(<>);
  type Vektor is array (Index range <>) of Elem;
  with function "<" (Bal, Jobb: Elem) return Boolean is <>;
function Maxker(V: Vektor) return Elem;
-- A program a következő oldalon folytatódik.
```

```
-- Folytatás az előző oldalról.
function Maxker(V: Vektor) return Elem is
   Max: Elem := V(V'First);
begin
   for I in V'Range loop
      if Max < V(I) then
         Max := V(I);
      end if;
   end loop;
   return Max;
end;
```

Ennek az alprogramnak (a buborékrendezéses példához hasonlóan) az *Elem*, az *Index* és a rájuk építő *Vektor* lesz a típusparamétere. A maximum kiszámításához megint szükségünk lesz egy < műveletre. Nézzünk egy példát a függvény használatára is:

```
with Text_Io, Maxker; use Text_Io;
procedure Maxker_Demo is
   type Int_Vektor is array (Integer range <>) of Integer;
   function Int_Maxker is new Maxker(Integer, Integer, Int_Vektor);
   A: Int_Vektor(1..5) := (2, 8, 9, 23, 5);
begin
   Put("A vektor maximuma " & Integer'Image(Int_Maxker(A)));
end Maxker_demo;
```

Látszik, hogy nem közvetlenül a *Maxker* függvényt használtuk, hanem létrehoztunk belőle egy *Int_Maxker* nevű konkrét példányt (lásd a 11.4. alfejezetet), ami már egészeken működik. Mivel ezt az úgynevezett példányosítást az Adában minden konkrét esetre el kell végeznünk, az Ada polimorfizmusa lényegében egy *„szintaktikai cukormáz"* (syntactic sugar): csak egyszer kell leírnunk a függvény törzsét, de minden típushoz új példányt kell készítenünk belőle.

A GJ (Generic Java) nyelvben az Adához hasonlóan már a deklarációban megkövetelhetjük a paraméterre az összehasonlító műveletet [Bra 98]. A Javában az ilyen követelményeket **interface**-ekkel szokás megadni. Egy típus akkor implementál egy interfészt, ha az abban szereplő összes műveletet megvalósítja. Az összehasonlíthatóságot a *java.lang.Comparable* interfész implementálása (lásd a 12.7.6. részt) biztosítja.

```
public interface Comparable<A> {          // Generic Java
    public int compareTo (A that);
}
```

Itt az **interface** is kapott egy *A* típusparamétert. Ennek oka az, hogy csak arra van szükségünk, hogy egy *A* típusú objektum egy másik *A* típusú objektummal legyen összehasonlítható. A Javában a *compareTo* művelet tetszőleges objektummal való összehasonlítást valósít meg [Nyé 01], ami nem felel tökéletesen annak, amit várunk egy ilyen művelettől. A Generic Java – többek között ebben is – nagyobb kifejezőerővel rendelkezik a Javánál.

```
public interface Comparable {                    // Java
    public int compareTo (Object that);
}
```

Ezek után a *max* metódus *Elem* típusparaméterét megszoríthatjuk úgy, hogy implementálja a *Comparable<Elem>* interfészt. Metódusoknál a típusparamétert (vagy típusparamétereket) a művelet visszatérési típusa előtt kell deklarálni[2]:

```
class Max {
  public static <Elem implements Comparable<Elem>> Elem
     max (Collection<Elem> xs) {
        Iterator<Elem> xi = xs.iterator();
        Elem m = xi.next();
        while (xi.hasNext()) {              // amíg van következő elem
            Elem x = xi.next();             // a következő elem
            if (w.compareTo(x) < 0) {       // elemek összehasonlítása
                m = x;
            }
        }
        return m;
    }
}
```

A *max* művelet csak olyan gyűjteményekre működik, amelyek elemei egyetlen típushoz (amit az *Elem* jelöl) tartoznak, és erre a típusra létezik a *'compareTo()* metódus. Ezt elő is kell írni a specifikációban. Az *Iterator<Elem>* az *Elem* elemtípusú gyűjtemények elemeit sorolja fel. Ennek felhasználásával és az *Elem* paraméter *compareTo()* műveletének segítségével a függvény meg tudja keresni a legnagyobb elemet, tehát a művelet aktuális típusparaméterének meg kell valósítania a *compareTo* műveletet, a következő példában is látható módon.

```
class EgyTípus implements Comparable<EgyTípus> {
    int érték;
    public int compareTo (EgyTípus that) {
        return this.érték - that.érték;
    }
    ...
}
```

Csakúgy, mint a C++-ban, a Javában sem kötelező a *max* művelet hívásakor a típusparaméter konkrét értékét megadni (lásd a 11.4. részt):

```
LinkedList<EgyTípus> lista = new LinkedList<EgyTípus>();
// ...
EgyTípus listaMax = Max.max(lista);        // nem Max.max<EgyTípus>(lista);
```

[2] A Java nyelv parametrikus polimorfizmust támogató prototípusa egyelőre csak az `extends` kulcsszót támogatja, erre kell cserélni a példánkban szereplő `implements` kulcsszót is.

Ha olyan típussal példányosított listára hívjuk meg a *max* műveletet, amely nem implementálja a *Comparable* interfészt, fordítási hibát kapunk.

Látható, hogy a funkcionális példa jóval egyszerűbb volt a többinél. Ennek az az oka, hogy a modern funkcionális nyelveknek alapeleme a parametrikus polimorfizmus, nem olyan „szintaktikai cukormáz", mint például az Adában vagy a C++-ban.

11.1.2. Altípusos polimorfizmus

Az objektumorientált nyelvekben gyakran előfordul, hogy egy objektumot több típushoz tartozónak tekintünk. Például a harcsa, a keszeg vagy a ponty mind halak, így egy adott ponty nemcsak a pontyok, hanem a halak közé is tartozik. Mivel a halak közös tulajdonsága az, hogy tudnak úszni, ha megadunk egy úszás műveletet, annak minden halra működnie kell; akkor is, ha a konkrét hal altípusa keszeg, és akkor is, ha ponty. Úgy is tekinthetjük tehát, hogy a konkrét hal több típushoz is tartozik: egyszerre hal és például harcsa. A hal típusra működő úszás műveletet tehát nevezhetjük *altípusosan polimorfnak* (*inclusion polymorphism*), hiszen működik a harcsa, a keszeg és a ponty típusra is, amelyek mind a hal típus altípusai.

Az altípusos polimorfizmus fogalma nagyon hasonló a parametrikus polimorfizmuséhoz, de eléggé elkülönülnek egymástól mind a gyakorlatban, mind elméleti hátterükben ahhoz, hogy külön csoportként tárgyaljuk őket. A kettő közötti különbség oka, hogy míg a parametrikus polimorfizmusnál maga a *programegység volt többalakú*, itt azokat az *objektumokat tekintjük többalakúnak*, amiket a programegység használ. Hogyan lehetséges ez? Valójában az altípusos polimorfizmusnál az objektumot több osztályhoz tartozónak tekintjük. Ezek az osztályok egy közös őstípus altípusai, ezért viselkedésüknek vannak közös tulajdonságai, erre lehet alapozni az absztrakt implementációt. Ezt a polimorfizmust valósítják meg az objektumorientált nyelvek, ahol az altípus-hierarchiát az öröklési hierarchia adja – erről bővebben lásd a 12. fejezetet. (Ez az elterjedt és széles körben használt objektumorientált nyelvekre oly jellemző szemlélet egyébként súlyos problémákat okoz a nyelvek típusrendszerében. Számtalan kutató és számtalan – kutatási céllal készült – programozási nyelv mutatta meg, hogy az altípusosság és az öröklődés szétválasztása növeli a nyelvek biztonságosságát, mert csökkenti az adott nyelven írt programokban előforduló típushibák lehetőségét. Könyvünk terjedelmi korlátai nem teszik lehetővé ezen igencsak érdekes témakör részletes kifejtését. Az érdeklődő olvasónak figyelmébe ajánljuk [Bru 02]-t, mely alaposan körüljárja ezt a kérdést, és számos további hivatkozást is tartalmaz. Mindössze annyit jegyzünk meg, hogy a Generic Java nyelvet alkotói már a legújabb kutatási eredményeket is figyelembe véve készítették el, így sikerült összeegyeztetni a nagy kifejezőerőt a biztonságos típusrendszerrel.)

Az altípusos polimorfizmus (a parametrikus polimorfizmushoz hasonlóan) jól használható általános feladatok megoldására. Gyakori, hogy egy adott osztályhierarchiában a közös műveleteket, tulajdonságokat egy *absztrakt alaposztályba* (*abstract base class*) fogják össze. Ez az osztály absztrakt metódusokkal rendelkezik, és nem lehet objektumot létrehozni belőle. A leszármazottaknak meg kell valósítaniuk az ős absztrakt metódusait, hogy példányokat lehessen belőlük létrehozni. Az absztrakt ős tehát egy közös *felületet* biztosít a leszármazottak

kezeléséhez (lásd a 12.7.4. részt). Az osztály-hierarchiával kapcsolatos általános műveletek megírásakor felhasználhatjuk a közös felület által nyújtott lehetőségeket.

A C++ nyelvben a dinamikus kötést a virtuális függvények, az interfészeket a *tisztán virtuális osztályok* biztosítják:

```
#include <iostream>
#include <algorithm> // for_each
#include <vector>
#include <functional> // mem_fun

struct Shape {
    virtual void draw() = 0; // virtuális az összes leszármazottban.
};

struct Circle : public Shape {
    void draw() { cout << "circle drawn."; };
};

struct Square : public Shape {
    void draw(){ cout << "square drawn."; };
};

void drawAll(vector<Shape*> shapes) {
        for_each(shapes.begin(),shapes.end(),mem_fun(&Shape::draw));
}

int main() {
    Shape* s1 = new Circle;
    Shape* s2 = new Square;
    s1->draw();             // Circle::draw hívása
    s2->draw();             // Square::draw hívása
    vector<Shape*> shapes;
    shapes.push_back(s1);
    shapes.push_back(s2);
    drawAll(shapes);
}
```

Itt a *Shape* (alak) osztálynak a rajzoló metódusa (*draw*) tisztán virtuális, így *késői összekapcsolással* történik a feloldása, ha mutatón keresztül hívjuk meg. A *0* érték teszi tisztán virtuálissá, ami azt vonja maga után, hogy a *Shape* osztályból nem lehet példányt létrehozni, csak ilyen típusú mutatót deklarálni.Ezt a mutatót csak a *Shape* osztály leszármazottjaira állíthatjuk. A leszármazott osztályok felüldefiniálják a *draw* metódust, és mindig a megfelelő leszármazott osztály *draw* művelete fog meghívódni. Tehát a példában az *s1*-re és az *s2*-re meghívott rajzoló metódus rendre a *Circle* és a *Square* osztály *draw* művelete lesz. Ezt

használja ki a *drawAll* függvény, amely alakzatok vektorban tárolt mutatóit kapja paraméterül, hogy kirajzolja ezeket az alakzatokat. A *for_each* szabványos könyvtári sablonfüggvény [Ste 95] végighalad a vektoron, (a két iterátor a vektor elejét és "utolsó utáni" elemét jelzi) és az összes elemére meghívja a *Shape::draw()* tagfüggvényét a *mem_fun* (member function) STL sablonfüggvény segítségével. A *drawAll* törzsén is látható, hogy a sablonok és az objektumorientált módszerek milyen jól illeszthetők egymáshoz, és hogyan bontják a kódot érthető, fogalmi egységekre.

Az igazi objektumorientált nyelvekben, mint a Java, a Smalltalk, az Eiffel stb., a példánymetódusok *alapértelmezés szerint virtuálisak*, mindig a dinamikus típus szerint hívódnak meg.

Az altípusos polimorfizmus másik gyakori felhasználása a gyűjtemény típusok megvalósítása. Ha a nyelvben létezik egyetlen *közös ősosztály* (mint például a Javában az *Object* vagy az Eiffelben az *ANY*), erre az osztályra lehet építeni a gyűjtemények megvalósítását. Mivel minden objektum ennek az osztálynak a leszármazottja, tetszőleges objektumot beletehetünk a gyűjteménybe. Ennek a megközelítésnek hátránya az, hogy általában nem kezeli a primitív típusokat (vannak azonban olyan nyelvek, például a C#, ahol a fenti megközelítés ellenére ez megoldott), továbbá az, hogy nem alkalmazható, ha nincs közös ős a nyelv osztályhierarchiájában (mint például a C++ esetén).

Az Eiffelben a gyűjtemény típusok közös őse az absztrakt *CONTAINER[G]* sablon osztály. Ebben vezették be a gyűjteményekkel kapcsolatos alapvető műveleteket, mint például a *has* és *empty* lekérdezések, amelyek tartalmazási és üresség vizsgálatot biztosítanak minden gyűjteményhez.

A nyelv tervezői a gyűjtemények létrehozásánál különválasztották a hozzáférés, a tárolás és a bejárás nézőpontjait.

A hozzáférés azt határozza meg, hogy a gyűjtemény felhasználói hogyan érhetik el a gyűjtemény elemeit. Például egy verem vagy egy sor esetében egyszerre csak egy elem érhető el, és a típus felhasználóinak nincs módjuk befolyásolni, hogy melyik legyen az az egy elem. Ezzel szemben egy tömb vagy egy hasítótábla használói egy indexet, vagy általánosabban egy kulcsot adnak egy elem eléréséhez.

A tárolás határozza meg, hogy a gyűjteményeinknek hány tételt szabad tartalmazniuk és hogy vajon ezt a tulajdonságot meg szabad-e változtatni. Sok gyűjtemény véges, mások elvileg végtelenek (ilyenek például a prímszámok). A véges szerkezetek közül néhány korlátos, néhány korlát nélküli. Az elsőre példa az *ARRAY[G]*, a tömb osztály sablonja, a korlát nélkülire a *LINKED_LIST[G]* osztály, a láncolt listákat megvalósító sablon típus.

A bejárás határozza meg, hogy – ha egyáltalán lehetséges –, hogyan lehet egy gyűjtemény elemeit egyenként, mindegyiket pontosan egyszer, egy világosan meghatározott sorrendben elérni. Például egyes gyűjtemények szekvenciálisan bejárhatók, egy- vagy két irányban. A fák pre-, post-, vagy inorder bejárásra adnak lehetőséget.

A fenti szempontok mindegyikéhez az Eiffel szabványos könyvtára egy (és csak egy) osztály-hierarchiát biztosít. A hozzáférési hierarchia gyökere a *COLLECTION* osztály, a tárolásé a *BOX*, a bejárási hierarchia teteje pedig a *TRAVERSABLE* osztály. Mindhárom osztály a *CONTAINER* osztály leszármazottja.

Egy új, speciális gyűjtemény – mint például a *LINKED_LIST[G]* osztály – megvalósítása úgy történik, hogy a fenti hierarchia mindhárom ágából kiválasztunk egy megfelelő

osztályt, s azokat többszörös öröklléssel kombináljuk. Így egy olyan új osztály kapunk, amelyet teljes mértékig jellemez a hozzáférési, tárolási és bejárási tulajdonsága.[3]

```
deferred class ALAKZAT
 feature
  kiir is
    deferred
  end
end

class KOR
 inherit
   ALAKZAT
 feature
   kiir is
    do
     io.putstring ("KOR VAGYOK");
     io.new_line
   end
end

class SOKSZOG
 inherit
   ALAKZAT
 feature
   kiir is
    do
     io.putstring ("SOKSZOG VAGYOK");
     io.new_line
   end
end

class NEGYZET
 inherit
   SOKSZOG
     redefine kiir
     end
 feature
   kiir is
    do
     io.putstring ("NEGYZET VAGYOK");
     io.new_line;
   end;
end
```

[3] Ez az módszer nyilván ahhoz vezet, hogy osztályunk többszörösen örököl majd a CONTAINER osztálytól.

```
class TESZT
 creation
   make
 feature
  t:  LINKED_LIST[ALAKZAT]
  o1, o2, o3: ALAKZAT

  make is
   local i:INTEGER
   do
    !NEGYZET! o1 -- Létrehozza az objektumokat.
    !KOR! o2
    !SOKSZOG! o3
    !!t.make          -- Létrehozza a láncolt listát.
    feltolt           -- Feltölti t-t.
    from              -- Ciklussal végigmegy rajta:
      t.start
     until
      t.off
     loop
      t.item.kiir
      t.forth
    end
  end
  feltolt is
   do
    t.put_front(o1)
    t.put_front(o2)
    t.put_front(o3)
  end
end
```

A fenti példában a *LINKED_LIST[G]* osztályt a (**deferred** kulcsszó használata miatt absztrakt) *ALAKZAT* osztállyal példányosítottuk. A *KOR* és a *SOKSZOG* osztályok ennek konkrét leszármazottjai, ahol már megvalósítottuk a *kiir* jellemzőt, ezt definiálja át a *SOKSZOG* osztály leszármazottja, a *NEGYZET*. Az *ALAKZAT* típusú *o1*, *o2* és *o3* hivatkozásokhoz a megfelelő konstruktorok hívásával (*!NEGYZET! o1*, *!KOR! o2*, *!SOKSZOG! o3*) rendre hozzárendelhetünk *NEGYZET*, *KOR*, illetve *SOKSZOG* típusú objektumokat. Ezekkel feltöltve az *ALAKZAT*-ok *t* listáját, egyszerűen ciklusban végiglépkedhetünk rajta, s a *t.item.kiir* kiíratás mindig a lista aktuális elemének (*item*) dinamikus típusa szerint választódik majd ki.

11.1.3. Túlterheléses polimorfizmus

A 11.1. alfejezet bevezetőjében az ad-hoc polimorfizmusra adott példa (360. oldal) valójában a *túlterheléses polimorfizmust* (*overloaded polymorphism*) szemlélteti. A túlterheléses poli-

morfizmus tulajdonképpen csak egy hasznos *szintaktikai cukormáz*: ugyanaz a név használható több, különböző típuson működő alprogramhoz, és a környezet – azaz az alprogram aktuális paraméterei – segítségével dönthető el, hogy éppen melyiket kell használni. (Az alprogramnevek túlterheléséről már volt szó a 7.6. szakaszban.)

A túlterheléses polimorfizmus feloldása elképzelhető úgy is, hogy az előfeldolgozás során a fordítóprogram minden változatnak új nevet ad, és a hívás helyén a közös nevet mindig a megfelelő új névre cseréli. Ez történik például CLU-ban a beépített, túlterhelt operátorok használata esetén.

Az olvasó a túlterheléses polimorfizmus és az Ada **generic** vagy a C++ **template** fogalmát összevetve könnyen gondolhatná azt, hogy az Ada **generic** polimorfizmusát valójában ebbe a kategóriába kellene sorolnunk. Mi az oka annak, hogy mégsem ezt tettük? A válasz az, hogy (mint azt már az ad-hoc polimorfizmus leírásában kifejtettük) a nem univerzálisan polimorf programegységeknél a különböző típusokhoz általában különböző megvalósítások tartoznak. Az Ada **generic**-nél ezzel szemben egyetlen egységes, absztrakt megvalósítást adunk (sajnos a fordító több konkrét megvalósítást fog előállítani, hiszen az Adában és a C++-ban a típusok nincsenek egységesen ábrázolva, lásd a 11.1.1. részt). Látható tehát, hogy a túlterheléses polimorfizmus csak egy absztrakt felületet biztosít (de absztrakt megvalósítást nem), ezért a polimorfizmus egy korlátozott formájának tekinthető (a típuskényszerítéses polimorfizmussal – lásd a 11.1.4. részt – együtt).

A Java nyelvben a felhasználó csak metódusokat definiálhat felül, operátorokat nem, viszont a nyelv által biztosított túlterhelt operátorváltozatokat használhatja. Például a + operátor használható számokra és karakterláncokra is:

```
2.0 + 2.0       // double típusokra összeadás
"2.0" + "2.0" // karakterláncok összefűzése
```

Nézzünk egy másik (C++ nyelven írt) példát! A szabványos könyvtárbeli *sort()* függvény alapértelmezés szerint a túlterhelt < művelet adott típusra vonatkozó változatát használja:

```
class Ember { /*...*/ };

bool operator< (Ember e1, Ember e2)
{
    return e1.eletkor() < e2.eletkor();
}

int main()
{
    vector<Ember> baratok;
    std::sort(baratok.begin(), baratok.end()); // <(Ember,Ember) alapján
}
```

A túlterheléses polimorfizmus a funkcionális nyelvekben is létezik. Az absztrakt felületet ehhez a **class** eszköze biztosítja. Erről bővebben a 16.4. rész szól.

11.1.4. Típuskényszerítéses polimorfizmus

Tegyük fel, hogy a szorzás műveletét valós számokra definiáltuk. Ilyenkor esetleg hasznos lehet, ha a műveletet egész számokra is használhatjuk. Ennek nincs elvi akadálya, ha az egész típusú értékek minden veszteség nélkül valósakká alakíthatók.[4] Ez történik akkor, amikor a nyelv a *típuskényszerítéses polimorfizmust* (*coercion polymorphism*) alkalmazza: ha a műveletet nem a megfelelő típusú paraméterekkel hívták meg, akkor azokat a nyelv automatikusan a művelet által várt típusúra konvertálja.

A típuskényszerítéses és a túlterheléses polimorfizmus fogalma (a parametrikus és az altípusos polimorfizmushoz hasonlóan) nehezen különböztethető meg. Azonban míg a túlterheléses polimorfizmus valójában *szintaktikai rövidítés*, addig a típuskényszerítéses polimorfizmus *szemantikai művelet*: az aktuális paraméterek olyan típusúra konvertálódnak, ami megfelel a formális paramétereknek, és eközben az adatreprezentáció megváltozik. Ha ez a konverzió nem történne, akkor típushiba lépne fel. Ilyen konverzió történik például a C nyelvben [Str 00]:

```
double max(double egyik, double masik)
{
  return (egyik > masik) ? egyik : masik;
}

int main()
{
  return max(42 * 42, 40.5 * 43);
}
```

A típuskényszerítéses polimorfizmus a C++-ban például már működhet felhasználói típusokra is, ha konverziós konstruktort vagy konverziós operátort írunk.

A típuskényszerítésnek egy másik fontos példáját ismét a C++ nyelven mutatjuk be. Ebben a nyelvben teljesül ugyanis, hogy ha a *Derived* osztály a *Base* osztály leszármazottja, akkor egy *Base** típusú változó típuskényszerítés nélkül kaphat *Derived** típusú értéket. Ilyenkor persze a fordító automatikusan *Base** típusúra konvertálja a *Derived** típusú mutatót.

A Javában a típuskényszerítés a metódushívásoknál is megtörténik. Tegyük fel, hogy a *Derived* osztály a *Base* osztályból származik, és egy *C* osztálynak van egy *m(Base)* szignatúrájú metódusa. Ekkor a metódust bármilyen *Base* objektummal, vagy annak leszármazottjával meg lehet hívni. Az alábbi programrészletben a fordítóprogram *implicit típuskényszerítést* alkalmaz a *derived* referenciaváltozóra, és *Base* típusúra konvertálja azt, mert a metódus szignatúrája ezt írja elő [Rog 01]:

```
C c = new C();
Derived derived = new Derived();
c.m(derived);
```

[4] Sajnos a gyakorlatban ez nem mindig így történik. Például a Java nyelv gond nélkül `float` (lebegőpontos valós) típusúvá konvertál egy `long` (hosszú egész) típusú változót. Mivel az előbbi típust 4, míg az utóbbit 8 bájton ábrázolja a Java, ez konverzió adatvesztéssel jár.

A típuskényszerítés kétféleképpen történhet. Az egyik lehetőség a *statikus típuskényszerítés*, ilyenkor a konverziós operátor fordítási időben automatikusan bekerül a függvény és paramétere közé. A másik lehetőség az, hogy *dinamikusan*, futási időben történik a paraméterek ellenőrzése.

Mint már említettük, a típuskényszerítéses és a túlterheléses polimorfizmus fogalma bizonyos esetekben erősen összemosódik. Ezt mutatja a következő példa is [Car 85]:

3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0

Tegyük fel hogy van egy olyan + műveletünk, ami a paraméterek minden, fent megadott kombinációjára működik. Hogyan magyarázhatjuk a művelet polimorfizmusát?

1. Elképzelhető, hogy a függvénynek négy túlterhelt jelentése van, minden paraméterkombinációhoz egy.
2. Lehetséges az is, ha csak két túlterhelt változat létezik, egy az egész, egy a valós paraméterekhez. Amikor az egyik paraméter egész, a másik valós, akkor az egész paraméter valóssá konvertálódik. Ilyenkor nyilvánvalóan a valós paraméterekhez készített változat hívódik meg.
3. A harmadik lehetőség az, hogy csak valós paraméterekhez létezik definíció és minden egész paraméter valóssá kényszerítődik.

Mindhárom változatnak megvan a maga létjogosultsága. Az, hogy egy nyelv melyiket választja, a nyelvtől függ. Itt tehát általánosan nem tudjuk eldönteni, hogy milyen polimorfizmus valósul meg, a kérdés csak konkrét nyelvekre, a nyelv definíciójának ismeretében válaszolható meg.

11.1.5. A polimorfizmus megvalósítása monomorf nyelvekben

A programozási nyelveknek viszonylag kis része támogatja természeténél fogva a parametrikus polimorfizmust. Ilyenek például az ML-hez hasonló, tisztán funkcionális nyelvek [Car 85]. Az altípusos polimorfizmust általában az objektumorientált nyelvek (Eiffel, Java stb., lásd a 12. fejezetet) támogatják. Ezek tehát azok a nyelvek, amelyek az univerzális polimorfizmus valamilyen formáját valósítják meg. Ezen nyelvek típusrendszerét nevezzük *polimorfnak*. A *monomorf típusrendszerek* – a polimorf rendszerekkel ellentétben – csak egyetlen viselkedést engednek meg az objektumoknak. A parametrikus polimorfizmus esetén a polimorf programegységek többféleképpen viselkedhetnek, azaz különféle típusokra működnek. Az altípusos polimorfizmusnál maguknak az objektumoknak lehet több különböző viselkedésük, mint azt a halas példánál láttuk. A monomorf rendszerek nem teszik lehetővé ilyen egyértelműen a többféle viselkedést, de bizonyos speciális esetekben tehetnek kivételeket. Ha egy típusrendszer semmilyen módon nem enged polimorf viselkedést, a rendszert *szigorúan monomorfnak* nevezzük.

A szigorú monomorfizmus előnyei közé tartozik, hogy a típushibák fordítási időben, egyszerűen kideríthetők. Az ilyen nyelveknek azonban nagy hátránya, hogy kifejezőerejük

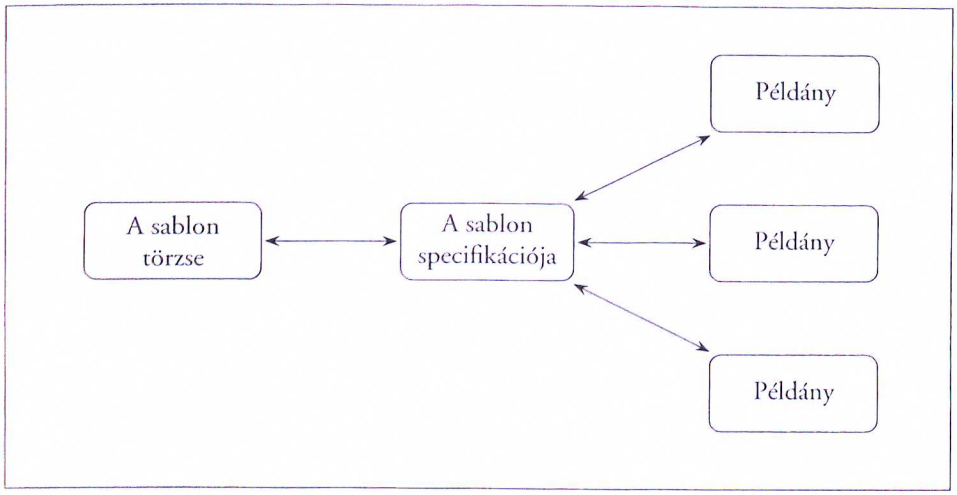
erősen korlátozott, hiszen nem engedik meg, hogy az objektumok viselkedése a környezettől függően változhasson. Ezért bizonyos esetekben a monomorf rendszerek is engedményeket tesznek a polimorfizmus irányába. Ezeket a típusrendszereket *jobbára monomorfnak* nevezzük. A jobbára monomorf kifejezés azt jelenti, hogy a nyelv alapvetően monomorf, de vannak bizonyos esetek, amikor valamilyen szintű polimorfizmust is lehetővé tesz. Ilyen nyelv például a Pascal vagy a C. Az alábbiakban felsoroljuk azokat a polimorfizmus irányában tett engedményeket, amelyeket az ilyen nyelvek támogathatnak:

- *Túlterhelés (overloading)* Ugyanazt a nevet különféle típusokon működő alprogramok is megkaphatják, például a + műveletet definiálhatjuk mátrixokra is.
- *Típuskényszerítés (coercion)* Például egész paraméter használható olyan műveleteknél, amelyek valós paramétert várnak és fordítva.
- *Altípusosság (subtyping)* Ha valamilyen típusból altípust képzünk (például a típusértékhalmaz megszorításával, lásd az 5.8. részt), az altípus elemei automatikusan az eredeti típushoz tartozónak is tekinthetők, tehát az altípus használható minden olyan helyen, ahol az eredeti típus szerepel.
- *Értékmegosztás (value sharing)* Jellegzetes példa erre a **nil** konstans, ami bármilyen típusú mutatóként használható.

Ezek egymástól teljesen független módszerek a monomorfizmus enyhítésére. Természetesen akár mind a négy lehetőség előfordulhat ugyanabban a nyelvben, de ez nem szükségszerű. Az ilyen nyelvek már nem tekinthetők szigorúan monomorfnak.

Lássuk, hogyan illeszthetők be ezek a kivételek a polimorfizmus előbb megadott definícióiba! Az első három enyhítés esetében nyilvánvaló, hogy melyik polimorfizmus-típushoz tartoznak (bár látható, hogy a megadott altípusosság nem valósítja meg teljes egészében az altípusos polimorfizmust, annak csak egy alesetét jelenti, hiszen altípust nem csak a típusértékhalmaz direkt megszorításával érhetünk el, hanem azzal is, ha egy meglévő típus specifikációját további tulajdonságokkal egészítjük ki, mint a halas példában is). A negyedik lehetőség tulajdonképpen a parametrikus polimorfizmus egy alesete. Első közelítésben azt gondolhatnánk, hogy az előbb említett **nil** mutató tulajdonképpen egy túlterhelt konstans. Ez azonban a következők miatt nem igaz: a **nil** konstans használható bármilyen típushoz tartozó mutatóra, az olyan típusokhoz tartozókhoz is, amiket még nem is definiáltunk, tehát lényegében végtelen számú típusra működik, ráadásul a különböző típusokhoz tartozó **nil** mutató ábrázolása azonos, ami túlterheléses polimorfizmus esetén általában nem igaz. Nagyon hasonló ehhez az Eiffel *NONE* osztálya, ami definíció szerint minden (létező és még nem létező) osztály leszármazottja. A *NONE* típusú *Void* szerepe a **nil**-éhez hasonló.

Az olvasóban felmerülhet a kérdés, hogy ha az Ada (és a C++) parametrikus polimorfizmust valósít meg, akkor mi az oka annak, hogy nem szerepelnek a parametrikusan polimorf típusrendszerrel rendelkező nyelvek között. Ennek egyik oka az, hogy ezekben a nyelvekben a polimorf programegység egyfajta kivétel, míg a polimorf típusrendszerekkel rendelkező nyelveknél a polimorf elemek a nyelv alapkonstrukciói közé tartoznak. A másik ok a már említett *szintaktikus polimorfizmus*: az Ada **generic**-ek és a C++ **template**-ek valójában csak szintaktikai rövidítést jelentenek a hasonló törzsű programegységek definíciójához.



11.2. **ábra:** A sablon-szerződés modell

Szemantikájuk a makróhelyettesítésnek felel meg, tehát specializált kódok készülnek minden olyan típushoz, amivel használták a polimorf elemet, ami nem jellemző a polimorf típusrendszerrel rendelkező nyelvekre.

11.2. A sablon-szerződés modell

Egy polimorf művelet vagy típus használatakor szigorú szabályoknak kell megfelelni. Meg kell bizonyosodnunk arról, hogy a példányosításkor megadott aktuális paraméterek, illetve funkcionális nyelveknél a típusrendszer által levezetett aktuális típusok megfelelnek-e a formális típusparamétereknek. Az aktuális és formális paraméterek egyeztetésének módját az úgynevezett *sablon-szerződés modell* segítségével tárgyaljuk [Nyé 98].

A polimorf konstrukciókhoz, röviden sablonokhoz (polimorf függvények, absztrakt adattípusok, csomagok) tartozó deklarációk és törzsek értelmezése és érvényességének ellenőrzése a sablon fordításakor történhet. Ez lehetővé teszi azt, hogy a sablonban előforduló hibák korán, már a sablon fordításakor kiderüljenek, így példányosításkor biztosak lehetünk abban, hogy a példányosítandó sablon önmagában helyes.

Miért nevezzük a modellt sablon-szerződés modellnek? Azért, mert a sablon specifikációja implicit módon egy *szerződést* fogalmaz meg a sablon törzse és a példányosítás között (lásd a 11.2. ábrát). Mivel a törzset már a példányosítás előtt lefordítottuk, biztosított, hogy a törzs megfelel a specifikációnak (azaz betartja a specifikáció által előírt szerződést), így ha a példányosítás is betartja a szerződést, garantált, hogy a példány is érvényes lesz. A módszer tehát lehetővé teszi a példány és a sablon egymástól független kezelését. Így a sablon törzsének megváltoztatása, amennyiben a törzs továbbra is eleget tesz a szerződésnek, nincs

befolyással a példány érvényességére. Ha a sablon-szerződés modellt nem tartjuk be, előfordulhat, hogy bizonyos hibák csak a példány fordításakor, vagy pláne futtatásakor derülnek ki. Ahhoz, hogy a típushibákat már jó időben megtaláljuk, természetesen az kell, hogy a sablon specifikációjában megjelenjen minden, a példányosító típussal szembeni követelés: a formális paraméter típusosztálya meghatározza az összes lehetséges aktuális paramétertípus esetében használható művelethalmazt. Ez azt jelenti, hogy a sablon törzsében nyugodtan használhatjuk ennek a művelethalmaznak az elemeit anélkül, hogy tartanunk kellene attól, hogy az aktuális típushoz nem létezik ez a művelet (hiszen akkor az aktuális paraméter megsértené a szerződést, így a példányról már fordítási időben kiderül, hogy nem érvényes).

Az eddigi példákból kitűnik, hogy az Ada és Eiffel nyelvek például fontosnak tartják a sablonok sablon-szerződés modell szerinti kezelését, míg a C++ nyelv nem.

A kérdés a modellel kapcsolatban az, hogy a fordító hogyan ellenőrzi a szerződés betartását. Ahhoz, hogy az aktuális típus garantáltan biztosítsa a törzs számára szükséges műveleteket, a specifikáció és a törzs egyeztetésekor a fordító a *„lehető legrosszabbat"* feltételezi a példányosításról, hiszen ha létezik olyan művelet, amit a specifikáció nem követel meg, de a sablon törzse használja, akkor létezhet olyan példányosítás, ami megfelel a specifikációnak, de nem biztosítja az említett művelet meglétét, így nem tudjuk biztosítani a sablon és a példányosítás függetlenségét. A másik oldalról – tehát a példányosítás érvényességének ellenőrzésekor – a fordító a *„lehető legjobbakat"* feltételezi a példányosításról, azaz hogy a példányosítás a specifikációban megfogalmazott összes követelésnek eleget tesz, és ezt ellenőrzi is. Ezáltal biztosított az, hogy a törzs és a példányosítás érvényessége egymástól teljesen elkülönítve vizsgálható, nem kell minden példány érvényességét újra megvizsgálni, ha a törzset módosítjuk, sőt a példányosításhoz nincs is szükség a sablon törzsének ismeretére, elég, ha a specifikációt ismerjük.

Az Eiffelben például, ha *A –> COMPARABLE* az *X* osztály sablonparamétere, akkor ez egyrészt jelzés az *X* osztály törzsének, hogy használhatja a *COMPARABLE* osztály műveleteit is a sablonparaméter típusú változókon, másrészt jelzés a sablon példányosítójának, hogy csak a *COMPARABLE* osztály leszármazottaival szabad példányosítani.

A funkcionális nyelvekben szintén megvalósul a sablon-szerződés modell: polimorf függvények definiálásakor megadható minden olyan információ, ami szükséges ahhoz, hogy csak jó típussal lehessen példányosítani. A 356. oldal Clean nyelvű példájában ez a következőképpen jelenik meg: a típusparaméterre olyan megszorítást tettünk, hogy az aktuális paraméterre léteznie kell a < összehasonlító operátornak. Ha olyan típusra próbálunk példányosítani, amire ez a művelet nem létezik, már fordítási időben hibát kapunk[5].

Természetesen vannak olyan nyelvek is, amelyeknél a példányosítás nem a sablon-szerződés modell szerint működik. Az ilyen nyelveknél sok kellemetlenség adódhat, ha a

[5] A tisztán funkcionális nyelvekhez írt fordítóprogram általában képes önállóan levezetni a függvények típusát, ezért bizonyos esetekben nem kell megadnunk a függvény specifikációját. Típuslevezetéskor a fordítóprogram azt is ellenőrzi, hogy milyen műveleteknek kell léteznie a típusparaméterre. Ha a programozó által megadott specifikáció nem felel meg a fordító által levezetett típusnak (például a programozó kifelejtett valamilyen szükséges megszorítást a típusparaméterre), akkor a fordítóprogram hibát jelez.

példányosításnál nem ellenőrizzük, hogy a sablon törzse nem használ-e valamilyen műveletet, amit az aktuális típusnak kéne biztosítania. Nézzük milyen problémák kerülhetnek elő akkor, ha egy programozási nyelv nem tartja be a sablon-szerződés modellt!

Az Ada 83 ugyan általában betartotta a sablon-szerződés modellt, de tartalmazott egy súlyos hibát, amit az Ada 95-ben már kijavítottak. Az alábbi példa, bár másra nem igazán jó, jól szemlélteti ezt a hibát.

```
generic
   type A is private;
procedure Sablon;

procedure Sablon is
   x: A;
begin
   ...
end Sablon;

procedure Példány_1 is new Sablon(Integer);  -- Ez rendben van.

procedure Példány_2 is new Sablon(String);  -- Ezzel baj van!
```

A problémát az okozza, hogy a *Sablon* sablon törzsében deklarálhattunk egy *A* típusú változót, de az Ada 83 szabályai alapján a sablon specifikációjában nem jelenik meg az a követelés a példányosító típussal szemben, hogy objektumok létrehozhatók legyenek belőle, tehát *A* aktuális értéke tetszőleges típus lehetett. Ez általában (például *Integer* típussal való példányosításkor) nem okoz gondot, de bizonyos típusok, például a megszorítás nélküli indexhatárú tömbök (ilyen a *String* is), nem alkalmasak arra, hogy az indexhatárok megadása nélkül objektumot hozzunk létre belőlük. Ezeket a típusokat *nem teljesen meghatározott típusoknak* nevezzük. Nézzük, mi történt, amikor a sablont egy nem teljesen meghatározott típussal (például a *String* típussal) próbálták példányosítani! Mivel a sablon specifikációjában nem követeljük meg, hogy az *A* típus *teljesen meghatározott* legyen, ez a példányosítás megengedett volt. Az „*x: A*" deklaráció az *A* típusparaméter aktuális értékét felhasználva az „*x: String*" deklarációnak felelt meg. Így azonban nem szabad változót deklarálni, hiszen a *String* megszorítás nélküli indexhatárú tömb, ahhoz, hogy konkrét objektumot tudjunk létrehozni belőle, meg kell adnunk az indexhatárokat (tehát például az *x: String(1..20)* deklaráció helyes lenne). Ekkor azonban a sablon törzsében az *x: A* helyett az *x: A(1..20)* deklaráció kellene, hogy szerepeljen. Összegezve tehát: a sablont példányosítani lehetett megszorítás nélküli típusokra, azonban amikor a példányt valaki használni próbálta, hibaüzenetet kapott, mivel az *x* változót nem lehetett létrehozni.

Nyilvánvaló, hogy ez sértette a sablon-szerződés modellt: a sablon specifikációjában nem szerepelt a példányosító típussal szemben támasztott összes követelés, mivel nem jeleztük, hogy definit típust várunk példányosításkor. Erre megoldást jelentett az Ada 95

nyelvben a teljesen meghatározott és a nem teljesen meghatározott típusparaméterek megkülönböztetése:

```
generic
   type A is private;          -- Teljesen meghatározott típus.
   type B (<>) is private;     -- Nem teljesen meghatározott típus.
procedure Sablon;

procedure Sablon is
   x: A;                       -- Megengedett
   -- y: B;                    -- Nem megengedett
begin
   ...
end Sablon;

procedure Példány_1 is new Sablon(Integer, String);      -- Ez helyes
-- procedure Példány_2 is new Sablon(String, Integer);   -- Ez nem
```

A sablon specifikációjában azt, hogy egy típus nem teljesen meghatározott, a *(<>)* kulcsszóval jelezhetjük. Minden olyan típusváltozó, amit a *(<>)* nélkül deklaráltak, csak teljesen meghatározott típussal példányosítható. Ezzel a változtatással sikerült megszüntetni a sablon-szerződés modellt megsértő problémát, így az Ada 95 nyelv már teljesen a sablon-szerződés modell szerint működik.

11.3. Sablonparaméterek

A sablonparaméterek négy csoportból kerülhetnek ki: lehetnek objektumok, alprogramok, típusok és típusosztályok, illetve modulok.[6]. Az alábbiakban a négy paraméterfajta által biztosított lehetőségeket, a formális paraméterek specifikációjának szintaxisát, illetve a sablon-szerződés modell megvalósítását tárgyaljuk.

11.3.1. Típus és típusosztály

A legjellemzőbb sablonparaméter talán a *típus*. Ez az oka annak, hogy a polimorfizmussal kapcsolatban a legtöbbet a *típussal való paraméterezhetőségről* beszéltünk. A legegyszerűbb eset az, amikor a típusparaméternek bármilyen típus megfeleltethető. Példaként tekintsük a *length* függvényt, amely egy lista hosszát határozza meg. Ez a függvény független a lista elemtípusától, így a *length* függvény formális típusparaméterére semmilyen megkötést nem kell tenni. Funkcionális nyelven megfogalmazva a függvény típusát:

```
length :: [a] -> Int
```

[6] Az Ada 95-ben nincs jelentősége annak, hogy alprogramok önállóan legyenek sablonparaméterek, hiszen itt már bevezették a (paraméterként átadható) alprogramra irányított mutatók lehetőségét, azonban az Ada 83-ban még a sablonparaméterként használt alprogramok nyújtották az egyetlen lehetőséget arra, hogy alprogramot paraméterként adjunk át. (Lásd a 7. fejezetet.)

Ennél bonyolultabb az az eset, amikor az aktuális típusparaméter valamilyen műveletét szeretnénk használni a sablonban. A *sablon-szerződés modell* értelmében ezt jeleznünk kell a sablon specifikációjában, azaz valamilyen megkötést kell adnunk a formális típusparaméterre. Ez a megkötés lehet egyszerűen annyi, hogy valamilyen műveletnek léteznie kell a példányosításra használt típushoz. Ezt szemlélteti a következő Ada alprogram [Nyé 98]:

```
generic
  type Elem is limited private;
  with function "<" (A, B: Elem) return Boolean;
  with procedure Beolvas (SX: in out Boolean; DX: out Elem;
                          X: in File_Type);
  with procedure Kiir (X: in File_Type; E: in Elem);
procedure Osszefesul (X, Y: in File_Type; Z: in File_Type);
```

Az *Osszefesul* függvény a két rendezett bemeneti fájlt fésüli össze úgy, hogy a kapott fájl is rendezett legyen. Ennek megvalósításához szüksége van egy, az *Elem* típuson működő < függvényre, valamint *Beolvas* és *Kiir* eljárásokra.

```
procedure Osszefesul( X, Y: in File_Type; Z: in File_Type ) is
   SX, SY: Boolean := True;
   DX, DY: Elem;
begin
   Beolvas(SX,DX,X);
   Beolvas(SY,DY,Y);

   while SX or else SY loop
      if SX and then (not SY or else DX < DY) then
         Kiir(Z,DX);
         Beolvas(SX,DX,X);
      elsif SY and then (not SX or else DY < DX) then
         Kiir(Z,DY);
         Beolvas(SY,DY,Y);
      else
         Kiir(Z,DX);
         Beolvas(SX,DX,X);
         Beolvas(SY,DY,Y);
      end if;
   end loop;
end Osszefesul;
```

Az alprogram kezdetben beolvas egy-egy elemet az *X* és az *Y* fájlból. Az *SX* és az *SY* a *Beolvas* alprogram kimenő paraméterei, értékük azt mutatja, hogy az olvasás sikeres volt-e vagy nem (azaz volt-e még a fájlban kiolvasható elem, vagy már a végére értünk). A *DX* és a *DY* szintén kimenő paraméterek, ezek tárolják a kiolvasott értéket. A *Kiir* művelet a második paraméterként kapott értéket írja az első paraméterként kapott fájl végére.

Az Adában lehetőség van arra is, hogy a típusparaméter valamilyen *típusosztályhoz* való tartozását követeljük meg (például azt, hogy az aktuális típusnak diszkrét típusnak kell lennie). Ez történik akkor is, amikor valamilyen tömb típust adunk át paraméterként. Jó példa erre a 353. oldalon található példa *Index* és *Vektor* típusa.

Az absztrakt megvalósításhoz szükséges művelet meglétét gyakran úgy garantáljuk, hogy az aktuális paraméternek valamilyen *osztály leszármazottjának* kell lennie. Ekkor az ősosztály műveleteire nyugodtan hivatkozhatunk, mert garantált, hogy a leszármazott osztályok is adnak valamilyen megvalósítást a műveletre. Ez történik az objektumorientált nyelvekben. Mint a 11.1.1. fejezetben láttuk, a GJ nyelvben tehetünk megszorításokat a paraméteres osztály, **interface** vagy metódus típusparaméterére a típusparaméter deklarációjakor, és a példányosító típusnak meg kell felelnie a megszorításoknak, különben fordítási hibát kapunk.

Hasonlóképpen, az Eiffelben azt az osztályt adhatjuk meg, amitől a sablonparaméter örököl. Ezzel egyben biztosítható a sablon törzsében használt típusműveletek megléte. Például, ha egy olyan polimorf vektor típust szeretnénk megvalósítani, amire működik a vektorok közötti összeadás, akkor szükségünk lesz az elemtípuson megadott összeadás műveletére. Ez úgy garantálható, hogy az elemtípus az *ADDABLE* osztály leszármazottja.

A harmadik lehetőség az, hogy a típust egy *műveletosztályhoz* tartozónak tekintjük. A műveletosztály gyakorlatilag egy *absztrakt felület*, amelyhez minden típusra készíthető egy implementáció. Ilyenek a tisztán funkcionális nyelvek `class`-ai (lásd a 16.4.1. részt), illetve a Java **interface**-ei. Ezek a műveletosztályok gyakorlatilag túlterheléses polimorfizmust valósítanak meg, míg a sablonok, amikben használjuk őket, parametrikusan polimorfak.

11.3.2. Alprogram

Az Ada nyelvben alprogramok nem csak akkor adhatók át sablon paraméterként, amikor egy típusosztályt segítenek meghatározni, mint a 377. oldal példájában, amikor a *<*, *Beolvas* és *Kiir* alprogramok az *Elem* típusról való feltételezéseink megfogalmazásához kellettek. Sőt, az Ada 83-ban az egyetlen lehetőség az alprogramokkal való paraméterezésre a sablonok használata. Az alábbi *Integrál* sablon függvény specifikációja arra utal, hogy a sablon paraméterként átadott *F* függvény határozott integráljának közelítését szeretnénk leprogramozni. (Vessük össze ezt a megoldást a 7.3.1. rész 214. oldalon található példájával, ahol is Pascal nyelven készítettünk el egy hasonló függvényt.)

```
generic
  with function F(X: Float) return Float;
function Integrál(Alsó_Határ, Felső_Határ, Lépésköz: Float) return Float;
```

Mint már említettük, az Ada 95 nyelvben – oly sok más nyelvhez hasonlóan – már másik lehetőségünk is van az alprogramokkal történő paraméterezésre, mégpedig „alprogramra mutató" típusú alprogramparaméterek segítségével. Erről a 7.3. részben (a 214. oldalon) olvashatunk részletesebben.

11.3.3. Objektum

Sablonparaméterként értékek és változók is átadhatók. Ennek segítségével leprogramozhatók például olyan *absztrakt típusok* vagy műveletek, amelyek mindenféle méretű összetett típusokon működnek. Erre volt példa az a verem típus, aminek egy maximális méretet is adtunk át paraméterként (lásd a 356., illetve 358. oldal példáit).

A következő példa azt mutatja meg, hogy hogyan lehet értéket, illetve változót sablonparaméterül adni az Adában:

```
with Text_IO; use Text_IO;
procedure Obj_P_Mod is
  generic
    E: in Integer;
    V: in out Integer;
  procedure G;
  procedure G is
  begin
    V := E * V;        -- E := E * V hibás, mert E nem balérték
  end G;
  X: Integer := 6;
  procedure P is new G(X, X); -- ugyanaz, mint G(6, X)
begin
  X := X+1;
  P;
  Put_Line(Integer'Image(X)); -- az X értéke 42
end;
```

Az *E* egy bemenő paraméter (ezt jelzi az **in** kulcsszó), tehát csak az értékét használhatjuk, ezért nem szerepelhet értékadás bal oldalán. Az **in out** kulcsszópár azt jelenti, hogy a hozzá tartozó *V* be- és kimenő paraméter, tehát értékét a sablon módosíthatja. (Pontosabban: a *V*-nek a példányosításkor megfeleltetett aktuális paramétert a példány módosíthatja.)

11.3.4. Modul

Gyakori, hogy egy sablon egy meglévő másik sablon szolgáltatásait egészíti ki, vagy egyszerűen csak a másik sablon által megadott absztrakcióra épít. Ez kényelmesen kifejezhető úgy, hogy a meglévő sablont (sablon paramétereivel és szolgáltatásaival egységbe zárva – lásd a 12.4. részt) egy egységként átadjuk az új sablon paramétereként. Ez bizonyos nyelveken, például Eiffelben, nagyon egyszerűen megtehető, hiszen a szolgáltatások egységbe zárásának eszköze az *osztály*, és sablon paramétere egyébként is csak osztály lehet. Erre láttunk példát a 11.3.1. rész *VEKTOR* sablonjának elkészítésekor is.

A sablon paraméterfajtákban gazdagabb Ada 95 nyelvben is van lehetőség hasonlóra. Szolgáltatások egységbe zárására a csomag szerkezet használható ebben a nyelvben, így a

kívánt hatás csomagok sablon paraméterként történő szerepeltetésével érhető el. Nézzünk erre egy példát!

```
generic
  type Elem is digits <>;
  type Index is (<>);
  type Matrix is array (Index range <>, Index range <>) of Elem;
package Matrix_Aritmetika is
  function "*" ( A, B: Matrix ) return Matrix;
  ...
end Matrix_Aritmetika;
package body Matrix_Aritmetika is ...

type Real is digits 5;
type Matrix is array (Integer range <>, Integer range <>) of Real;
package Real_MA is new Matrix_Aritmetika(Real,Integer,Matrix);
```

A *Matrix_Aritmetika* egy alapvető mátrixműveleteket (például összeadás, szorzás, skalárral való szorzás) megvalósító sablon csomag. Tegyük fel, hogy most egy olyan sablont szeretnénk írni, ami ezekre az egyszerű aritmetikai műveletekre építve *bonyolultabb műveleteket* (például mátrixinvertálást, háromszögekre bontást stb.) valósít meg. Írhatnánk például ezt:

```
generic
  type Elem is digits <>;
  type Index is (<>);
  type Matrix is array (Index range <>, Index range <>) of Elem;
  with function "*" ( A, B: Mátrix ) return Matrix is <>;
  ...
package Osszetett_Matrix_Muveletek is
  function Inverz ( M: Matrix ) return Matrix;
...
```

Látható, hogy a *Matrix_Aritmetika* sablon összes paraméterét, sőt, az összes benne megadott műveletet fel kell sorolnunk új sablonunk paraméterlistájában. A bonyolultságon kívül problémát okozhat az is, hogy a felsorolt függvények *nem képeznek szemantikai egységet*, azaz példányosításkor nem kötelező a *Matrix_Aritmetika* csomag műveleteit használnunk, hanem például a * műveletet vehetjük máshonnan. Elképzelhető az is, hogy ez a * művelet nem mátrixszorzást, hanem valamilyen egészen más mátrixok közötti műveletet valósít meg. Nyilvánvaló, hogy ennek a * műveletnek az *Osszetett_Matrix_Muveletek* csomag törzsében (a szokásos mátrixműveletekkel együtt) való alkalmazása nem a várt eredményt hozná. Ezeknek a problémáknak az orvoslására szolgál a (sablon) csomag sablonparaméterként való megadása.

```
generic
  with package Aritmetika is new Matrix_Aritmetika (<>);
package Osszetett_Matrix_Muveletek is
  function Inverz ( M: Aritmetika.Matrix ) return Aritmetika.Matrix;
...
```

Ebben a példában a formális paraméter csomag specifikációja tartalmazza egy sablon csomag nevét: az aktuális paraméter csomag ennek a sablon csomagnak kell, hogy egy példánya legyen.

A formális paraméter csomag (itt *Aritmetika*) neve egy csomagot (nem egy sablon csomagot) jelent az *Osszetett_Matrix_Muveletek* sablonban. Komponenseire is hivatkozhatunk, például az *Aritmetika."+"* műveletre, sőt, akár a formális paraméter csomag formális paramétereire is, például az *Aritmetika.Matrix* típusra. A minősítés feloldása is lehetséges, az alábbi módon:

```
generic
  with package Aritmetika is new Matrix_Aritmetika (<>);
  use Aritmetika;
package Osszetett_Matrix_Muveletek is
  function Inverz ( M: Matrix ) return Matrix;
...
```

A *(<>)* jelzés a formális paraméter csomag specifikációjában azt jelenti, hogy a formális paraméternek bármilyen olyan csomagot megfeleltethetünk aktuálisként, ami az előtte álló sablon (jelen esetben a *Matrix_Aritmetika*) egy példánya.[7] Az *Inverz* szignatúrájában a *Matrix* az *Aritmetika.Matrix* típust jelenti.

11.4. Példányosítás

Egy sablon tényleges használata előtt tudnunk kell, hogy a *formális sablonparamétereknek* milyen *aktuális sablonparaméterek* felelnek meg. Ezt a megfeleltetést nevezzük *példányosításnak*.

A tisztán funkcionális nyelveken (ML, Clean, Haskell, Miranda stb.) a parametrikusan polimorf függvények hívásakor nem kell semmilyen módon megadni az aktuális paramétereket, a környezet alapján a típuslevezető rendszer határozza meg őket. Ezért ezekben a nyelvekben a példányosítás fogalma mást jelent. Példányosításnak a *típusosztályok konkrét típusokra történő megvalósítását* nevezzük. Erről bővebben a 16.4.1. rész szól.

Bizonyos nyelveken valamilyen módon példányosítani kell a sablont, e nélkül a sablon nem használható.

Példányosításon (*instantiation*) azt a folyamatot értjük, amelynek során a fordítóprogram a sablonból a paraméterek megfeleltetésével létrehozza a megfelelő deklarációkat, illetve definíciókat, amelyek szükségesek a sablon használatához.

A fordítóprogram megkeresi a sablonparaméterekben szereplő neveket, és azokat konkrét deklarációkhoz köti. Ezt a folyamatot névkötésnek (name binding) nevezzük. A létrehozott függvényeket és osztályokat *specializációknak* nevezzük.

A példányosítást sok szempontból lehet osztályozni. Például aszerint, hogy a fordítóprogram mikor végez példányosítást: a sablon tényleges használata esetén (*igény szerinti példányosítás, on-demand instantiation*), vagy bármikor, amikor a programozó *explicit* módon utasítja rá (*explicit instantiation*).

[7] A (<>) helyett adható valamilyen megszorítás is a paramétercsomagra.

Egy másik osztályzás szerint a fordító lehet *lusta* (*lazy*) összetettebb sablonok (például osztálysablonok) esetén, ahol nem generálja le az osztály összes tagját, hanem csak azokat, amelyekre szükség van a sablont használó kód fordításához, vagy *mohó* (*eager*), ha a teljes sablont példányosítja.

A harmadik kérdés pedig az, hogy a fordítóprogram a példányosításkor hogyan feleltei meg a paramétereket. Meg kell-e adnia a programozónak a sablon paramétereit, vagy a fordítóprogram vezeti le azokat?

11.4.1. Explicit példányosítás

Az Adában a példányokat külön kell deklarálni és csak ezekre lehet hivatkozni. Ezt már láttuk a 362. oldal *Maxker_Demo* példájában is. A C++-ban is rá lehet bírni *explicit példányosításra* a fordítóprogramot, bár ezt jóformán csak a sablonok megszorításainak ellenőrzésére használják [McN 00].

11.4.2. Igény szerinti példányosítás

A C++ nyelvben a fordítóprogram csak akkor példányosít egy adott sablont, ha arra tényleg szükség van. Például:

```
Link<int> *pointer;    // Itt nincs szükség példányosításra.
Link<int> intlink;     // Most viszont már példányosítani kell a Link sablont.
```

Egy mutató (például a *pointer* változó) deklarációja esetén a mutató által hivatkozott típusra (a példánkban ez a *Link*<**int**>) nincs szükség a C++-ban. Ezért a *pointer* változó deklarációjakor a fordító még nem példányosítja a *Link* sablont az **int** típussal. Azt a helyet, ahol egy adott sablon definíciójára először szükség van, *példányosítási pontnak* (*point of instantiation*) nevezzük. A fenti példában ez a *Link*<**int**> *intlink* deklaráció.

A példányosítás hasonlóan működik az Eiffelben is.

A GJ nyelvben a *típustörlés (type erasure)* művelettel a fordítóprogram a sablon osztályokat közönséges Java osztályokká alakítja a fordítás során, így sablon példányosítás egyáltalán nem történik. Ennek az az oka, hogy ebben a nyelvben a sablonok lényegében a futási idejű típusellenőrzések fordítási időbe való áthelyezésére valók, a sablon aktuális paramétereire specializált kód nem jön létre a sablonok használatakor.

11.4.3. Lusta példányosítás

A C++ nyelvben a példányosító folyamat *lusta:* nem állítja elő a sablon egészét, ha arra nincs szükség. Ezt mutatja be a következő példa.

```
template <class T> class List {
    int size() const;
    void sort();  // használ < operátort
    ...
};
```

```
class NoComp { /* nincs < operátor hozzá */ };

void f(List<NoComp>& ln, List<string>& ls)
{
    cout << ln.size();
    cout << ls.size();
    ls.sort();
}
```

A fordító létrehozza a *List<NoComp>::size()*, a *List<string>::size()* és a *List<string>::sort()* függvényeket, mert ezekre történt hivatkozás az *f* függvényben. A *List<NoComp>::sort()* függvény ellenben nem jön létre, hiszen azt *f* nem hívja. Ez egyébként a „szerencsénk": a *List<NoComp>::sort()* művelet ugyanis értelmetlen, azért, mert a *NoComp* típuson nincs < operátor értelmezve, ami pedig szükséges a *sort* definíciója szerint. Vegyük észre, hogy itt sérül a sablon-szerződés modell! A *List* sablon *NoComp* típussal történő példányosításakor nem ellenőrzi a fordító, hogy az aktuális paraméter típus megfelel-e minden olyan követelménynek, amelyet a sablontörzs támaszt vele szemben. A követelményekkel csak akkor szembesülünk, amikor a példány műveleteit meghívjuk, azaz amikor a művelet létrejön. A C++ szemlélete tehát az, hogy nyugodtan példányosíthatunk olyan típussal egy sablont, amely nem felel meg minden követelménynek, amíg nem használunk olyan műveletet a példányból, ami a hiányzó követelményre épít.

Gyenge és erős típusosság

A C++ sablonok *gyenge típusosságot* (*weak typing*) biztosítanak egy erősen típusos nyelvben [Eck 00], mivel a sablon nem követeli meg a paraméterétől, hogy pontos típusa legyen. Ehelyett csak azt igényli, hogy a megfelelő tagfüggvények és osztálytagok, amelyeket a sablon meghív, elérhetők legyenek a sablonparaméterben. Ezáltal a sablon sokkal rugalmasabban használható. A Smalltalk és Python nyelvekben a metódushívás eredendően gyengén típusos, ott tehát nincs is szükség **template**-ekre. Ezzel szemben az Ada vagy a GJ nyelvek explicit módon megszoríthatják a sablon paramétereit, tehát itt továbbra is az *erős típusosság* (*strong typing*) valósul meg. Ennek az az előnye, hogy a sablon-szerződés modellnek megfelelően a sablon hibái nem csak a sablon használatakor, hanem már a sablon fordításakor kiderülnek.

11.4.4. Sablonparaméterek megfeleltetése

A sablonparaméterek *megfeleltetése* az alprogramok paramétereihez hasonló módon történik (lásd a 7. fejezetet). Általában a pozíció szerinti megfeleltetést használjuk, de például az Ada megengedi a név szerinti megfeleltetést is. Bizonyos nyelveken, mint például az Ada és a C++, feltételezett értékeket rendelhetünk a sablonparaméterekhez.

Az Ada nyelvben *explicit példányosítás* történik, azaz a sablon példányosításakor egy új alprogramot vagy csomagot hozunk létre a konkrét paraméterek megadásával. Ezután ezt az újonnan létrehozott programegységet kell használnunk. Ezt szemlélteti a 11.1.1. rész 362. oldalán található példa is.

Más nyelvekben ez nem szükséges, elég, ha a hívás helyén adjuk meg a típusparaméter konkrét típusát. A C++ nyelvben például csak az osztálysablonok példányosításakor kötelező megadni az alapértelmezett értékekkel nem rendelkező paramétereket, a függvénysablonok paramétereit azonban *levezeti* a fordítóprogram. Például:

```
template <class T>
void swap(T& t1, T& t2) { /* ... */ }
int main()
{
    int i = 5, j = 7;
    swap(i, j);      // swap<int> hívása
}
```

Fontos megjegyezni, hogy a függvénysablonok típusparamétereinek levezetésekor a fordító nem hajt végre konverziókat (kiterjesztést, szabványos vagy felhasználói konverziót). Ezt szemlélteti a következő példa:

```
template<class T> T max(T,T);
//...
void f()
{
    max(3.14, 1.59);    // rendben: max<double>(3.14, 1.59)
    max(3.14, 8);       // hiba: max<?>(int, double) nem létezik
}
```

Az ilyen esetekben a (hibás) típuslevezetés elkerüléséhez a programozónak kell explicit módon megadni a sablonparaméter(ek) típusát, így a függvénynév feloldásakor nem történik implicit típuskényszerítés. Tehát az előbb ismertetett példában a probléma a következőképpen oldható meg:

```
max<double>(3.14, 8); // max<double>(3.14, double(8))
```

11.4.5. Specializáció

A C++ **template**-ek előnye, hogy az igény szerinti és lusta példányosítás miatt nem használt („felesleges") kódot a példányosítás nem állít elő. Egy érdekes trükkel azonban még a kevésnél is kevesebb kód generálására is rávehetjük a nyelvet. Tekintsük a következő példát:

```
template <class T>
class Vector {};

Vector<int> v1;
Vector<bool> v2;
Vector<AClass> v3;
Vector<A*> v4;
Vector<B*> v5;
```

Ekkor 6 osztály jön létre, a megfelelő (használt) műveletekkel. A kód méretének növekedését megfelelő *változatok* (*specializációk*) készítésével azonban elkerülhetjük. A helyben kifejtett függvények nem növelik a kódot, és az öröklődés kihasználásával elérhetjük, hogy egyetlen példány kezelje az összes *Vector<T*>* típust [Str 00]. Nézzük, hogyan történik ez:

```
template<>
class Vector<void*> {
    void** p;
    void* operator[](int i);
    /* ... */
};

template<class T>
class Vector<T*> : private Vector<void*> {
    typedef Vector<void*> Base;
    Vector():Base() {}
    explicit Vector(int i): Base(i) {}
    T*& operator[](int i) {
      return reinterpret_cast<T*&>(Base::operator[](i));
    }
};
```

Ezek után a fordítóprogram minden mutatókat tartalmazó vektor típus példányosításakor ugyanezt a kódot fogja felhasználni, nem állít elő új kódot.

A specializációt ezen kívül az is hasznossá teszi, hogy segítségével bizonyos típusokhoz specializált kódot adhatunk meg. Ekkor a specializáció eltakarja a példányosítással előálló változatot, azaz azokra a típusokra, amelyekhez létezik specializált változat, a fordító nem készít példányt, hanem ezt a specializált változatot használja.

11.5. Sablonok és öröklődés

A sablonok új típusok létrehozásához nyújtanak segítséget. Ebben a részben megvizsgáljuk, hogy a nyelvek típusrendszerébe hogyan illeszkednek a sablonok által létrehozott típusok. Ha *A* altípusa *B*-nek, és egy adott *T<A>* paraméteres típus is altípusa *T<B>*-nek, akkor *T* támogatja a *kovariáns, azaz „együtt változó" altípusosságot*. (Ilyen *T* típus a tömb típus a Javában: a *String[]* altípusa *Object[]*-nek. Ellenkező esetben az altípus tulajdonság *invariáns („nem változó")*. A kovariáns altípusosságot mutató *T* típusok általában biztonsági rést okoznak a nyelv statikus típusrendszerében. Az alábbi Java kódrészlet például típushibát tartalmaz, amit a fordítási idejű típusellenőrzés nem mutat ki, és csak futási időben derül ki:

```
Object[] t = new String[1];
t[0] = new Object();
```

A GJ nyelvben a *Stack* altípusa a *Vector* típusnak, de a *LinkedList<Stack>* nem altípusa *LinkedList<Vector>*-nak. Viszont a *Stack<A>* altípusa a *Vector<A>* típusnak, a *Stack<Integer>* típus pedig a *Vector<Integer>*-nek. Tehát itt invariáns altípusosságot figyelhetünk meg.

A C++-ban sincs kovariáns altípusosság [Str 00]:

```
class Shape { /* */ };
class Circle : public Shape { /* */ };
class Triangle : public Shape { /* */ };

void f(set<Shape*>& s)
{
    s.insert(new Triangle());
}

void g(set<Circle*>& s)
{
    f(s);      // Hiba: nincs konverzió set<Circle*>&-ről set<Shape*>&-ra.
}
```

Az *f()* tehetne *Triangle**-t a *set<Circle*>*-ba, ha azt *set<Shape*>*-ként kezelné. Így viszont a *set<Circle*>* nem *Circle**-okat tartalmazna.

Tehát a GJ-ben és a C++-ban is invariáns a paraméteres típusokra (a tömbök kivételével) az altípusosság.

Az Eiffelben a polimorfizmus fogalma szorosan összekapcsolódik az öröklés fogalmával, de mivel a függvényneveket nem lehet túlterhelni, meg kell említenünk egy érdekes jelenséget. Lássuk a következő példát [Mey 91]:

```
class A[G] feature
    f(x: G) is ... end
end

class B inherit
    A[INTEGER]
end

class C inherit
    A[REAL]
end
```

Látható, hogy az *A* osztály polimorf, egyetlen művelete van, a szintén polimorf *f*. A *B* és a *C* osztályok az *A* osztály *INTEGER*-re, illetve *REAL*-re való példányosítását öröklik. Tegyük fel, hogy egy *D* osztály mind *B*-től, mind *C*-től örököl. Ekkor nyilvánvalóan két változata létezik az *f*-nek, amelyek ütköznek egymással, mivel a típusaik nem felelnek meg egymásnak. Hogyan lehet feloldani ezt az ellentmondást? Az Eiffel egy speciális nyelvi elemet, a **select** záradékot adja a probléma megoldására (lásd a 12.7.5. részt).

```
class D
 inherit
  B
     rename f as bf
  end
  C
     rename f as cf
     select cf
  end
  ...
end
```

11.6. Összegzés

Ebben a fejezetben láthattuk azt, hogy a *polimorfizmus* fogalmának bevezetése nagyon hatékony absztrakciót tesz lehetővé: megteremti az elméleti alapot a *vezérlési* és az *adatabsztrakció* eszközéhez. A polimorfizmus fogalmát tovább bontottuk, így megkaptuk az *univerzális* és az *ad-hoc* polimorfizmusok, ezeken belül a *parametrikus* és az *altípusos*, illetve a *túlterheléses* és a *típuskényszerítéses* polimorfizmus fogalmát.

A különféle programozási nyelvek különféle nyelvi eszközöket biztosítanak a polimorfizmus eléréséhez. A tisztán funkcionális és objektumorientált nyelvek teljes mértékben támogatják a parametrikus vagy az altípusos polimorfizmust. Az Ada és a C++ egyfajta *szintaktikus parametrikus polimorfizmust* valósít meg. Láttuk azt, hogy *jobbára monomorf* nyelvek is adhatnak eszközöket a polimorfizmus bármely típusának bizonyos mértékű támogatására, és kiderült az is, hogy ezekhez képest a *szigorúan monomorf* nyelvek jóval kevesebb kifejezőerővel rendelkeznek.

A *sablon-szerződés modell* biztosítja azt, hogy a sablonok törzse és a példányosítás egymástól teljesen független legyen. A szerződés modell betartása nagyban megkönnyíti a moduláris fejlesztést, illetve a csoportban való programozást.

Láthattuk, hogy a polimorfizmus használata növeli a kód megbízhatóságát és újrafelhasználhatóságát, ezért nagyon hasznos eszköz *programkönyvtárak fejlesztésére.*

A típusrendszerekkel, nyelvek tervezésével és szemantikájával kapcsolatos kutatások során sok olyan eredmény született az utóbbi 15 évben, amelyek a polimorfizmus egyéb lehetőségeit (például illeszkedés-polimorfizmus, virtuális típusok, csoportos polimorfizmus, „dependent" típusok és típusok értékként történő használata) tárták fel, és amelyek a jövő programozási nyelveire – elsősorban az objektumorientált nyelvekre – remélhetőleg nagy hatással lesznek majd. Ezeket az új eredményeket sok érdekes nyelvben (Emerald, School, Theta, Quest, LOOM, Objective ML, Cayenne, gbeta, Scala stb.) kipróbálták, megvalósították már, ám az így keletkezett nyelvek általában kutatási célokra lettek tervezve, és így nem terjedtek el szélesebb körben.

Terjedelmi okok miatt ez a fejezet nem tért ki ezekre az újabb irányzatokra, hiába izgalmasak, és hiába mutatják be olyan jól a széles körben elterjedt, ipari és üzleti alkalmazások készítésére használt nyelvek hibáit, gyengeségeit. Ebben a fejezetben a Cardelli–Wegner-féle polimorfizmus fogalom fényében tárgyaltuk azokat a nyelveket, amelyek a gyakorlatban a legfontosabbak, amelyeket nem az akadémiai szféra, hanem a programozók használnak.

Feladatok

1. Hasonlítsuk össze az univerzális, illetve az ad-hoc polimorfizmust!
2. Milyen előnyei, illetve hátrányai vannak a szigorú monomorfizmusnak?
3. Hogy valósíthatók meg a különböző polimorfizmusok monomorf nyelveken?
4. Miért van szükség példányosításra és hogyan történik a példányosítás az egyes nyelvekben?
5. Melyek a polimorfizmus fajtái és milyen nyelvekben valósulnak meg?
6. Készítsünk olyan polimorf eljárást, amely megcseréli két (azonos típusú) paraméterét!
7. Készítsünk olyan függvényt, amely igazat ad vissza, ha a két paramétere egyenlő, hamisat, ha nem!
8. Helyesen működik-e az előző függvény karakterláncokra? Ha nem, hogyan adható olyan megoldás, amely a karakterláncokat is megfelelően kezeli?
9. Milyen polimorfizmust célszerű használni egy összehasonlító operátor megadására? Mely nyelvek támogatják ezt?
10. Mikor érdemes a típuskényszerítéses polimorfizmust használni, és milyen nyelveken lehetséges ez?
11. Miért nevezzük az Ada és a C++ nyelv parametrikus polimorfizmusát szintaktikusnak?
12. Hogyan osztályozhatjuk a nyelveket a példányosítás módja szempontjából?

12 Objektumorientált programozás

Az objektumorientált programozás során az adattagokat és az adatokat kezelő műveleteket egységbe zárhatjuk. Az osztály egyes elemei különböző láthatósági viszonyokkal rendelkezhetnek.

Az objektumorientált programozás egyik legfontosabb jellemzője, hogy az osztályok az öröklődés segítségével osztályhierarchiába szervezhetők. Az öröklődéskor egy ősosztály adattagjai és metódusai a leszármazott osztálynak is részét képezik. Az öröklődés megvalósítása az egyes programozási nyelvekben sok tanulságos kérdést vet fel.

A fejezetben áttekintjük az objektumorientált programozás általános elméletét és az egyes programozási nyelveken való megvalósítását.

A számítógépes programok segítségével az ember a valós világot próbálja modellezni. Az informatika eddigi története során számos módszer alakult ki annak érdekében, hogy ezt a modellezést könnyebbé, és a valóságot jobban közelítővé tegye.

Az objektumorientált szemlélet a valóság megközelítésének és ábrázolásának egy módszere [Mey 00, Ang 97, Bud 91]. A modellezés során a valós tárgyakat objektumokkal ábrázolja, amelyeket állapotukkal (tartalmukkal, adataikkal) és metódusaikkal jellemez. Az objektumorientált megközelítés tehát egy programozási szemléletmód, melynek alapján rendszerfejlesztési módszertanokat is kidolgoztak. Ezek a módszertanok a teljes fejlesztési folyamatot átfogják, a megvalósíthatósági elemzéstől kezdve az analízisen, tervezésen és implementáláson keresztül a tesztelés és karbantartás folyamatáig. Az egyes objektum-modellek speciális jellemzőkkel rendelkeznek, amelyek lehetővé teszik, hogy a valós világ egységeihez hasonló módon viselkedjenek. Az *analízis* során a rendszert együttműködő objektumok összességeként modellezzük, a *tervezés* és az *implementáció* során ezen objektumokat alakítjuk ki. Az objektumorientált szemlélet alkalmazásával, a valós világ és a modell kapcsolatának szorosabbá tételével, nagymértékben megkönnyítjük a valóság megértését, a valósághoz közelebb álló megközelítések alkalmazásával pedig egyszerűbben áttekinthetővé és könnyebben módosíthatóvá válik a program.

Objektumorientáltnak nevezünk egy programot, mely egymással kapcsolatot tartó objektumok összessége, ahol minden objektumnak megvan a jól meghatározott feladata.

12.1. Az osztály és az objektum

Az ember a körülötte lévő tárgyakat, valós objektumokat észreveszi, leegyszerűsíti, megkülönbözteti és rendszerezi. A végső cél a bonyolult világ megismerése, működésének megértése. A felhasznált eszköz pedig a modellezés. A modellezés során az ember olyan alapvető gondolatmenetet (algoritmust) használ, amelynek segítségével *elvonatkoztat*, *megkülönböztet*, *osztályoz*, *általánosít* vagy éppen *leszűkít*, *részekre bont* és *kapcsolatokat épít fel*.

Az *absztrakció* az a szemléletmód, amelynek segítségével a valós világot leegyszerűsítjük úgy, hogy csak a lényegre, a cél elérése érdekében feltétlenül szükséges részekre összpon-

tosítunk. Ez azt jelenti, hogy elvonatkoztatunk a számunkra pillanatnyilag nem fontos, közömbös információktól és kiemeljük az elengedhetetlen fontosságú részleteket.

Mit tekintünk *objektumnak*? Az objektumorientált szemléletben az objektumok a modellezendő valós világ egy-egy önálló egységét jelölik. Az objektumokat meghatározza *belső állapotuk* és a *nekik küldhető üzenetekre* való reagálásuk. Ez a „reagálás" jelentheti azt, hogy megváltozik az objektum belső állapota, illetve az objektum elvégez valamilyen „műveletet", kijelez a külső világ számára valamilyen értéket, üzenetet küld más objektumoknak, létrehoz, töröl, másol, illetve mozgat más objektumokat stb.

Az objektumokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük, és kategóriákba, osztályokba soroljuk, oly módon, hogy a hasonló tulajdonságokkal rendelkező objektumok egy osztályba, a különböző vagy eltérő tulajdonságokkal rendelkező objektumok pedig külön osztályokba kerülnek. Az osztályozás folyamata tulajdonképpen az *általánosítás* és a *specializálás* műveleteinek segítségével valósul meg. Az objektumok között állandóan hasonlóságokat vagy különbségeket keresünk, hogy ezáltal bővebb vagy szűkebb kategóriákba, osztályokba soroljuk őket.

Az osztályozás a természetes emberi gondolkodás szerves része. Az ugyanolyan fajta jellemzőket (adatokat) tartalmazó, és az ugyanolyan viselkedés-leírással (*metódusokkal*) rendelkező objektumokat egy *osztályba* soroljuk. Az objektum-osztályok hordozzák a hozzájuk tartozó objektumok jellemzőit. Minden *objektum* valamilyen osztály *példánya* (*instance*), rendelkezik osztályának sajátosságaival, átveszi annak tulajdonságait az adatszerkezetre és a műveletekre vonatkoztatva egyaránt.

Elmondhatjuk, hogy az *objektum* információkat tárol, és kérésre feladatokat hajt végre. Ilyen értelemben az objektum nem más, mint *adatok* (attribútumok) és *metódusok* (műveletek) összessége, ez utóbbiak elvégzik az objektumra szabott feladatot vagy leírják az objektum viselkedését.

Az objektumoknak mindig van *belső állapotuk* – ezt az adatok pillanatnyi értékei jellemzik. Metódushívások után az objektumok állapota megváltozhat. Az objektumok emlékeznek állapotukra és a feladatvégzési folyamat mindig egy *kezdőállapotból* (alapértelmezett érték, kezdőérték) indul, és egy másik állapotba megy át. A következő állapotátmenetnél onnan folytatja a folyamatot, ahol előzőleg abbahagyta.

Fontos kérdés az objektumok egyértelmű *azonosítása*, azonosíthatósága is. A valós életben minden objektum azonosítható. Lehetséges természetesen, hogy egy adott osztálybeli ábrázolás esetén két objektumnak ugyanaz a belső állapota, ez azonban még nem jelenti azt, hogy azonosak.

A fent említett fogalmak az objektumok *statikus jellemzőit* alkotják. A *dinamikus modell* a rendszer időbeli viselkedését írja le, – viselkedés alatt az objektumokat érő hatásokat, eseményeket, ezek sorrendjét, a műveletek, a metódusok végrehajtásának ütemezését, az állapotokat és azok változásait értjük. A dinamikus modellnél nyilvánvaló, hogy több objektum kapcsolatát, vagy egy objektumot és az őt érő környezeti hatásokat vizsgáljuk. Az objektumok nincsenek egyedül, kapcsolatban állnak egymással.

A kapcsolatban álló objektumok kommunikálnak egymással. A kommunikáció *üzenetküldés* formájában történik. Az üzeneteket rendszerint kívülről elérhető metódushívással ábrázoljuk. Az üzenetet a megszólított objektum azonosítójával minősítjük, és lehetnek paraméterei: *Obj. Üzenet (Paraméterek)*. Ha az objektumtól valamilyen választ várunk az üze-

netre, akkor ezt a változó paramétereken keresztül vagy a metódus visszatérési értékeként kaphatjuk meg.

Az osztályok leírásához a legmegfelelőbb eszközt az *absztrakt adattípusok* jelentik (lásd a 10. fejezetet): jellemezni kell az objektumok *értékhalmazát* és rögzíteni az *elérhető műveletek halmazát*, definiálva ezek absztrakt tulajdonságait. Így egy típusspecifikációt adunk meg, amely még nem törődik a reprezentációval, csak azt határozza meg, hogy milyen szolgáltatásokat nyújt a külvilág felé. A reprezentációt és a műveletek megfelelő implementációját elrejtjük.

12.1.1. Az osztályok és objektumok az egyes nyelvekben

Nézzük meg, hogyan hozhatunk létre osztályokat és objektumokat néhány programozási nyelvben.

SIMULA 67

A SIMULA 67 alapvető újítása és központi fogalma az objektum, amelynek az osztálydeklaráció révén definiált saját adatai és tevékenységei vannak. Az osztálydeklaráció egy mintát határoz meg, és az ezen mintához illeszkedő objektumok egy osztályhoz tartoznak.

```
Class Teglalap(TeglNev, Szelesseg, Magassag);    ! Osztály 3 paraméterrel;
  Text TeglNev; Real Szelesseg, Magassag;        ! Paraméterek specifikációja;
Begin
  Real Terulet, Kerulet;          ! Attribútumok;

  Procedure Frissit;              ! Művelet;
  Begin
    Terulet := Szelesseg * Magassag;
    Kerulet := 2*(Szelesseg + Magassag)
  End of Frissit;

  Procedure Kiir;                 ! Művelet;
  Begin
    OutText(" I am a Teglalap "); OutText(TeglNev); OutImage;
    OutText(" Szelesseg: ");      OutFix(Szelesseg,2,6);
    OutText(" Magassag: ");       OutFix(Magassag,2,6);
    OutText(" Terulet: ");        OutFix(Terulet,2,7);
    OutText(" Kerulet: ");        OutFix(Kerulet,2,6); OutImage
  end of Kiir;

  Frissit;                        ! Téglalap élete;
  OutText("Teglalap letrehozva."); OutImage;
  Kiir;
End of Teglalap;
```

A *Teglalap* osztályhoz tartozó objektumot a *new Teglalap("Kicsi", 2, 3)* kifejezéssel lehet létrehozni, műveletei a *Frissit* és a *Kiir*.

Smalltalk

A Smalltalkban minden objektum, még az osztályok is. Ez első megközelítésben azt jelenti, hogy az osztályoknak is vannak adattagjaik és metódusaik. Ez a gondolat nemcsak a Smalltalk következetességét mutatja, hanem arra is lehetőséget nyújt, hogy az osztály hozza létre a saját példányait, ami az osztály (mint objektum) művelete.

A metódusok a Smalltalk szóhasználata szerint a kapott üzenetek hatására lefutó eljárások. Egy új osztályt az alábbi módon tudunk létrehozni.

```
Object subclass: #Szamla
  instanceVariableNames: 'egyenleg'
  classVariableNames: ''
  poolDictionaries: ''
  category: nil !
```

A `Szamla` osztálynak egy `egyenleg` nevű adattagja lesz. Metódusokat az alábbi módon tudunk definiálni:

```
!Szamla class methodsFor: 'instance creation'!
  new
    |r|
    r := super new.
    r init.
    ^r
  !!

!Szamla methodsFor: 'instance initialization'!
init
  egyenleg := 0
!!
```

A `new` metódus az osztályhoz tartozó metódus, a konstruktor, mely létrehoz egy új példányt. Az `init` metódus a `Szamla` objektumaihoz tartozik és az objektum adattagjainak kezdeti értéket állít be.

Az osztály egy példányát a `Szamla new` utasítással hozhatjuk létre.

Érdemes megemlíteni, hogy mivel minden objektum, így például a vezérlési szerkezetek is azok. Nézzük meg, hogyan valósítható meg egy számlálós ciklus Smalltalkban. Ezt a `Number` osztály metódusaként valósították meg a következő szintaxissal:

⟨*alsó határ*⟩ `to:` ⟨*felsőhatár*⟩ `do:` `[:`⟨*ciklusváltozó*⟩ `|` ⟨*ciklusutasítások*⟩`]`

Például:

```
a := Array new: 10.
1 to: 10 do: [ :i | a at: i put: 0 ]
```

A Smalltalk felfogása szerint itt az `1`-nek küldjük a `to: do:` üzenetet, ahol a `to:` utáni `10` a ciklus felső határa, és a `do:` utáni blokk kerül végrehajtásra.

C++

C++-ban új osztályt az alábbi módon deklarálhatunk:

```
class Teglalap{
    int x, y;
  public:
    void ertekadas(int, int);
    int terulet() { return (x * y); }
};
```

A *Teglalap* osztály két (rejtett) adattagja *x* és *y*. Az osztálynak két nyilvános metódusa van, az értékadó és a terület számító metódus. A terület számító függvényt már az osztály deklarációjában definiáltuk, az értékadást az osztálydeklaráción kívül, a következő módon adhatjuk meg:

```
void Teglalap::ertekadas(int x1, int y1)
{
   x = x1;    y = y1;
}
```

A *Teglalap* osztály egy új objektumpéldányát létrehozhatjuk statikus, illetve dinamikus helyfoglalású változóként is:

```
Teglalap haz;                  // Statikus helyfoglalású példány.
haz.ertekadas(5,3);
int thaz = haz.terulet();      // thaz = 15

Teglalap* kert = new Teglalap; // Dinamikus helyfoglalású példány.
kert->ertekadas(20,17);
int tkert = kert->terulet();   // tkert = 340
```

Object Pascal

Az Object Pascalban [Can 00] új osztályt szintén a **class** kulcsszó segítségével hozhatunk létre[1]:

```
type
 TDatum = class
   private
     Ev, Honap, Nap: Integer;
   public
     procedure Beallit(e,h,n: Integer);
     function Szokoev: Boolean;
     ...
end;
```

[1] A korábbi Turbo Pascal változatokkal való kompatibilitás megőrzése érdekében emellett meghagyták az **object** típus lehetőségét is.

Az objektum példányok létrehozása dinamikusan történik[2], a változó deklarálása után meg kell hívnunk az osztály konstruktorát (lásd a 12.3. pontot), amely lefoglalja és inicializálja a szükséges memóriaterületet (*Create*). A használat befejeztével a memóriaterület felszabadítása is a programozó feladata (*Free*).

```
var
  EgyNap: TDatum;
begin
  ...
  EgyNap := TDatum.Create(...);      // Létrehozás.
  ...
  EgyNap.Beallit(2003, 4, 6);        // Használat.
  ...
  EgyNap.Free;                       // Felszabadítás.
  ...
end;
```

Az adatmezők és a metódusok mellett az osztálytípusok *tulajdonságokat* (**property**) is tartalmazhatnak. A tulajdonság egy olyan név, amely a megadott olvasás és/vagy írás műveleteken keresztül férhet hozzá az objektum adattagjaihoz.

```
property Ho: Integer
  read Honap write Ho_Ir;
```

Ha a tulajdonságnév kifejezésben szerepel, akkor értékét a *read* direktíva után megadott adat vagy metódus határozza meg. Ha értékadás bal oldalán szerepel, akkor a megadott értéket a *write* direktíva után megadott adat vagy metódus kapja meg. A fenti példában a *TDatum* osztály objektumaira a *Ho* tulajdonsag olvasáskor a *Honap* mezőt adja vissza, értékadás esetén pedig a *Ho_Ir* eljárást hívja meg.

Eiffel

Az Eiffel [Mey 91] tulajdonképpen nem is nevezhető objektumorientáltnak, inkább az *osztályorientált* a helyes kifejezés. Önálló objektumok nem hozhatók létre osztálytól függetlenül, és az osztályok sem viselkedhetnek objektumként. Eiffelben az osztály a fordítás egysége is.

Típusként tekintve az osztály leírja objektumok és műveleteik egy halmazát, amelyek a rendszer végrehajtása során létezhetnek. Ez fordítva is igaz, minden objektum, ami a rendszer végrehajtása során létezhet, ezen rendszer valamely osztályának egy példánya.

Az Eiffelben nincs más modularizáló eszköz az osztályon kívül, így egy programrendszer felépítése itt azt jelenti, hogy analizáljuk, hogy a rendszer milyen típusú objektumokkal fog dolgozni, és mindezen típusra készítünk egy-egy osztályt. Bizonyos esetekben az osztály csak egy szempontból érdekes, mint egy modul, amely számos eljárást összegyűjt. Ekkor nincs is feltétlenül attribútuma.

[2] Ellentétben a Turbo Pascallal.

Eiffelben osztályt a következő módon hozhatunk létre:

```
class KOMPLEX
  feature
    valos, kepzetes : REAL
  feature
    ertekadas(r, i : REAL) is
      do
        valos := r;
        kepzetes := i
      end;
      .....
end -- class KOMPLEX
```

A *KOMPLEX* osztálynak két adattagja (*valos*, *kepzetes*) és egy metódusa (*ertekadas*) van. Új objektum létrehozásához deklarálnunk kell a változót – például: *z : KOMPLEX;* –, majd az objektum számára még memóriát is kell foglalnunk a *!!* operátorral: *!!z;*. A memória felszabadítása automatikus.

Java

A Java [Nyé 01] magas fokú objektumorientáltsága következtében a Java program osztályok és objektumok összefüggő halmazát jelenti. A program futtatása nem más, mint adott objektumok és/vagy osztályok metódusainak meghívása.

Egy program megírása egy vagy több új osztály definiálását jelenti, a programvezérlés pedig nem más, mint metódusok megírása, felüldefiniálása, illetve a megfelelő eseménykezelők meghatározása.

A nyelv legkisebb önálló egységei az osztályok. Az osztály logikailag azonos típusú, összetartozó egyedek modellje. Ez a modell egyetlen egészet alkot és a külvilág számára egységesnek mutatkozik. A leírás adatmező deklarációkat és metódusleírásokat tartalmaz. Működése során a program példányosítja az osztályokat, s így objektumokat hoz létre. A Java az objektumokat dinamikusan kezeli. Minden változó hivatkozás egy objektumra.

Az alábbi példa egy új osztály definiálását mutatja [Nyé 01]. Az osztálynak két adatmezője és egy metódusa van.

```
public class Alkalmazott {
  public String nev;
  public int fizetes;
  public void fizetestEmel (int novekmeny) {
    fizetes += novekmeny;
  }
}
```

A következő kódrészlet definiál egy *Alkalmazott* típusú lokális változót, létrehoz egy új objektumpéldányt, értékadással nevet és fizetést ad neki, s végül fizetésemeléssel jutalmazza.

```
Alkalmazott a;
a = new Alkalmazott();
a.nev = "Kovács János";
a.fizetes = 50000;
a.fizetestEmel(6300);
```

C#

A C# osztályaiban elsősorban a C++, a Java és a Visual Basic hatását figyelhetjük meg, kombinálva új elemek bevezetésével. Szintaxisában egy osztály a C++-hoz áll legközelebb, ahogy a [C# Ref 03] oldal alapján készített példában is láthatjuk:

```
using System;
class Szemely {
    private string neve ="N/A";
    private int kora = 0;
    public string Nev {
        get {
            return neve;
        }
        set {
            neve = value;
        }
    }
    public int Kor {
        get {
            return kora;
        }
        set {
            kora = value;
        }
    }
    public override string ToString() {
        return "Név = " + Nev + ", Kor = " + Kor;
    }
    ....
}
```

A C# osztályok az adattagokon és metódusokon kívül *tulajdonságokat* (*property*) is tartalmazhatnak. A tulajdonságoknak nincs külön kulcsszava, mint például a Visual Basicben vagy az Object Pascalban. Az előző példában egy *Szemely* típusú objektum *Nev* vagy *Kor* tulajdonságát a **get**, illetve **set** *elérőkön* (accessor) keresztül használhatjuk. Egy tulajdonság set elérőjében a speciális *value* változó tartalmazza a felhasználó által az értékadásban megadott értéket. A *ToString* metódusban is automatikusan érvényesül a **get** elérő meghívása.

```
Szemely valaki = new Szemely();
...
valaki.Nev = "Kati";
valaki.Kor = 99;
Console.WriteLine("Személy adatai", valaki);
```

A tulajdonságok elérőin keresztül sok esetben áttekinthetőbben, tisztább szintaxissal valósíthatjuk meg a rejtett reprezentáció biztonságos kezelését, mint a hagyományos lekérdező/beállító metódusokkal. Például, ha a *valaki* objektum *Kor* tulajdonságát növelni akarjuk eggyel, akkor egyszerűen írhatjuk:

```
valaki.Kor += 1;
```

Ha külön metódusokat használtunk volna az objektum *kora* adattagjának beállítására, illetve lekérdezésére, akkor a kód a következő lenne:

```
valaki.KorIr(valaki.KorOlvas() + 1);
```

Ada 95

Mivel már meglévő programozási nyelvet terjesztettek ki, és az egyik fő szempont az Ada 83-mal való kompatibilitás volt, az osztályok deklarációja mellett lehetőség van hagyományos összetett adattípusok deklarálására is. Ennek következtében az Ada 95 programozási struktúrája nem szigorúan objektumorientált (lásd [Ada 95]). Az Ada 95-ben nem egy teljesen új fogalmat vezettek be az osztályok leírására, hanem a már meglévő rekordfogalmat bővítették ki. Így az osztályok, mivel azok speciális rekordok, csak az attribútumokat tartalmazzák, az osztályok műveletei ezen struktúrán kívül, speciális módon adhatók meg. Ennek következtében az Ada 95-ben nem is beszélhetünk a hagyományos értelemben osztályokról.

Az osztály attribútumainak és metódusainak egységbezárását úgy valósíthatjuk meg, ha egy csomagba (**package**) tesszük az attribútumokat tartalmazó rekordot és a metódusokat.

```
package Ember_Tip is
   type Ember is tagged private;
   procedure Nev_Ir (P : in out Ember; Nev : String );
   procedure Cim_Ir (P : in out Ember; Cim : String );
   procedure Kor_Ir (P : in out Ember; Kor : Integer );
   function Neve (P : Ember ) return String;
   function Cime (P : Ember ) return String;
   function Kora (P : Ember ) return Integer;
   procedure Megjelenit (P : in Ember );
private
   type Ember is tagged record
      Nev : String (1 .. 30);
      Cim : String (1 .. 30);
      Kor : Integer;
   end record;
end Ember_Tip;
```

Az *Ember* típus reprezentációja rejtett, így specifikáltuk a nevet, címet, kort beállító és lekérdező műveleteket, valamint a *Megjelenit* eljárást. Ezek törzsét a csomag törzsében kell megadjuk.

12.2. Jelölési módok, diagramok

Fontos az egységes jelölésmód bevezetése. Az előzőekben megismerkedhettünk az *osztály*, az *objektum*, az adatok, a metódusok, az állapot fogalmával. Azt mondtuk, hogy az osztályt adatok és metódusok összességével írjuk le, az objektumok pedig az osztályok példányai, és egy-egy pillanatban egy objektumot mindig egy adott állapot jellemez. Ezeket a meghatározásokat próbáljuk most diagramok segítségével ábrázolni.

12.2.1. Osztálydiagram

Az osztálydiagram tartalmazza az osztály nevét, az objektumok megvalósítására használt adatokat és típusukat, valamint a metódusokat.

Az adatok az állapotok leírásának belső ábrázolásai, a metódusok pedig a műveletek vagy üzenetek. A metódusokat paraméterlista követheti.

12.2.2. Objektumdiagram

Az objektumdiagram tartalmazza az osztály nevét, amelyből példányosítottuk és az objektum pillanatnyi állapotát.

12.2.3. A példányosítás ábrázolása

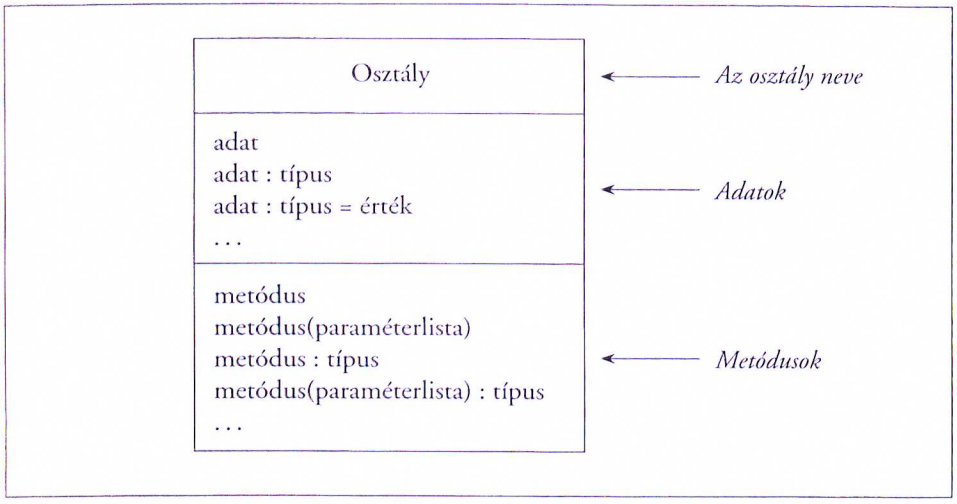
Az objektumok példányosításának ábrázolását mutatja a 12.3. ábra. Több objektum példányosítása esetén a példányszámot is külön fel kell tüntetni (12.4. ábra).

12.3. Objektumok létrehozása és megszüntetése

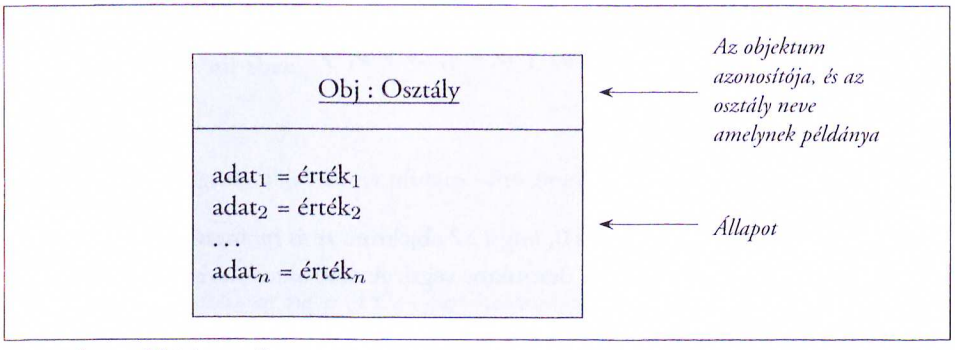
Az egyes programok objektumaira jellemző az életciklusuk: „megszületnek", „élnek", „meghalnak".

Az absztrakt adattípusok megvalósításánál fontos, hogy az objektumok létrehozásakor a rendszer állítsa be a típusinvariánst, vagyis állítsa be az objektum kezdeti állapotát a típusspecifikációban előírtnak megfelelőre. Ezt a feladatot végzi el az objektum *konstruktora*. Ekkor történik meg az objektum adattagjainak kezdőértékadása és az objektum működéséhez szükséges kezdeti tevékenységek végrehajtása[3]. A nyilvános osztályok konstruktorának

[3] A konstruktor feladata például, hogy virtuális metódust tartalmazó objektum létrehozása esetén a Virtuális Metódus Tábla (VMT) aktuális bejegyzését hozzárendelje az objektumhoz, hogy ennek felhasználásával tudja a program, hogy milyen címen található metódust kell meghívnia dinamikus kötés esetén, lásd a 12.7.2. pontot.



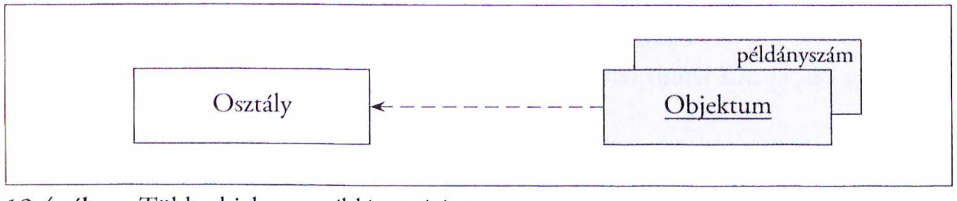
12.1. ábra: Osztálydiagram



12.2. ábra: Objektumdiagram



12.3. ábra: A példányosítás



12.4. ábra: Több objektum példányosítása

is nyilvánosnak kell lennie, hogy az objektumot használó program hozzáférhessen. Előfordulhatnak ugyanakkor olyan helyzetek is, amikor szeretnénk biztosítani, hogy egy osztálynak csak egyetlen példánya létezzen a memóriában, ekkor célszerű a külvilág elől rejtett konstruktort adni (lásd [Gam 95]).

Néhány nyelvben létezik destruktor is, melyet az objektum megszűnésekor hív meg a rendszer vagy a programozó, és a memóriabeli felszabadítást végzi el.

Az egyes programozási nyelvek különböző eszközöket adnak a konstruktorok és a destruktorok megvalósítására.

C++

A C++-ban egy osztály konstruktorának neve megegyezik az osztály nevével. Egy osztálynak több konstruktora is lehet a függvénynév túlterhelés lehetősége miatt.

```
class Komplex {
    double re, im;
  public:
    Komplex() { re = 0; im = 0; }
    Komplex(double a, double b) { re = a; re = b; }
};

Komplex z1;
Komplex z2(1, 1);
```

A *z1* objektum *re* és *im* mezője is 0, míg a *z2* objektum *re* és *im* mezője is 1 lesz.

Az objektumok felszabadítását a destruktor végzi. A destruktor neve az osztálynév elé írt ~ (tilde) jellel képezhető.

A C++-ban a destruktor statikus objektum megszűnésekor (hatókörből való kilépés), illetve dinamikus objektum helyének felszabadításakor automatikusan meghívódik. Ezért nem lehet paramétere sem, hiszen nem feltétlenül a programozó hívja meg[4].

Object Pascal

Object Pascalban egy osztályhoz több különböző konstruktort hozhatunk létre, ezek neve tetszőleges lehet és akármennyi paraméterük lehet. Deklarálásukkor a *constructor* kulcsszót kell használni. A *TObject* osztály (minden osztály őse) konstruktorának neve *Create*, általában érdemes ezt a nevet használni.

Object Pascalban a konstruktort az objektumpéldány használata előtt külön meg kell hívni.

[4] Az objektumok létrehozására és megszüntetésére vonatkozó részletes szabályok meghaladják a jelen leírás kereteit, ehhez Bjarne Stroustrup C++ könyvét ajánljuk [Str 00].

Egészítsük ki az előző dátum osztályt (lásd a 12.1.1. pontot) konstruktorral:

```
type
 TDatum = class
   private
     Ev, Honap, Nap: Integer;
   public
     constructor Create(e, h, n : Integer);
     ...
 end;
constructor TDatum.Create(e, h, n : Integer);
...
```

Ha az objektumok erőforrásokat foglalnak le, akkor, ha már nincs szükségünk az objektumra, ezeket fel is kell szabadítani. A *Destroy* virtuális metódus az alapértelmezett destruktor, célszerű ekkor ezt megfelelő módon átdefiniálni.

A *Free* metódus meghívásával elkerülhetjük, hogy üres hivatkozást próbáljunk felszabadítani, a *Free* csak valódi objektum esetén hívja a *Destroy*-t:

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

A már megsemmisített objektumra mutató változót ezután nekünk kell **nil**-re állítani.

Java

A Javában a konstruktor neve (a C++-hoz hasonlóan) megegyezik az osztály nevével, lehet(nek) paramétere(i), és nem lehet visszatérési értéke. Az objektum létrehozásakor automatikusan meghívódik. Az ősosztályok példányváltozói inicializálhatók a közvetlen ősosztály vagy az osztály konstruktorainak hívásával. Ha a kódban egyik sincsen, akkor az ősosztály paraméter nélküli konstruktora hívódik meg. Ha egy konstruktor saját vagy közvetlen ősosztálybeli konstruktort hív, akkor ez a hívás kell az első legyen a konstruktor törzsében. A konstruktor hívása előtt a példányváltozókra nem lehet hivatkozni.

Ha egy osztálynak nincs konstruktora, akkor a fordító a következőt generálja neki:

```
class AOszt extends BOszt {
  AOszt() {
    super();
  }
}
```

Ennek egy felhasználása:

```
Valt = new AOszt();
```

A Javában nincs destruktor, az allokált memória felszabadítása automatikus. A Java-értelmező tartalmaz egy belső memóriaellenőrző eljárást, amely az értelmezővel párhuzamosan fut és időről időre felszabadítja azokat a memóriahelyeket, amelyekre semmi sem hivatkozik. Destruktor helyett minden osztályban definiálható a *finalize()* metódus, s bár a meghívásának pillanata nem pontosan definiált, de biztosított, hogy még az objektum tárterületének újrafelhasználása előtt meghívódik.

A *finalize()* metódus osztályszintű megfelelője a *classFinalize()* osztálymetódus. Egy osztály *classFinalize()* metódusa azelőtt hívódik meg, hogy az osztályt törölné a szemétgyüjtő.

Eiffel

Eiffelben konstruktort a *creation* záradékkal adhatunk meg.

```
class KOMPLEX
  creation
    ertekadas
  feature
    valos, kepzetes : REAL
  feature
    ertekadas(r, i : REAL) is
      do
        valos := r;
        kepzetes := i
      end;
end -- class KOMPLEX
```

Az alábbi utasítás hatására létrejön a *z*-hez rendelt objektum, melynek kezdeti értéket is adunk.

```
z : KOMPLEX;
!!z.ertekadas(1,0);
```

Mint a Javában, nincs explicit destruktor Eiffelben sem. Amikor egy objektumra már nem hivatkozik többé a program, egy automatikus szemétgyűjtő eljárás (garbage collector) megszünteti az objektumot.

Ada 95

Ada 95-ben létezik két előre definiált jelölt (**tagged**) típus – a *Controlled* és a *Limited_Controlled* –, s ezek leszármazottaiban felüldefiniálva az *Initialize* műveletet, megadhatjuk, hogy az adott objektum létrehozásakor milyen programrész fusson le. Hasonlóképpen a megfelelő *Finalize* művelet fut le az objektumhoz tartozó memóriaterület felszabadítása előtt.

A példányosítás és a Self (this) fogalma

Az osztály és az objektum definíciójából kitűnik, hogy az objektum memóriabeli ábrázolása nem más, mint egy adatszerkezet ábrázolása. Az osztály tulajdonképpen egy típusleíró sablon. Adatokat és metódusokat ábrázolunk. A példányosítás folyamata során létrejön az objektum – összes adattagjával – és utasítások elvégzésére felszólító üzeneteket vár. A metódusokat az osztályhoz rendeljük, és nem másoljuk be minden újonnan létrehozott objektumba. Ez azt jelenti, hogy az objektum leírójában már nem szerepelnek a metódusok, csak az osztály leírójában. Az objektumok tudják, hogy melyik osztály példányai, és ennek alapján történik a metódusok kiválasztása.

Joggal vetődhet fel az a kérdés, hogy ha egy osztályból több objektumot példányosítunk, honnan tudjuk, hogy éppen melyik objektum hívta meg a megfelelő metódust, és a metódus melyik objektum adataival fog dolgozni? Ezt a programozási nyelvek az egyes metódusokban egy külön erre a célra szolgáló hivatkozással oldják meg, amely mindig a metódust meghívó aktuális objektumpéldányra mutat. Ez az úgynevezett *Self* (egyes nyelvekben *this* vagy *Current*) „paraméter". A *Self* metódushíváskor egyértelműen rámutat arra az objektumra, amellyel a metódusnak dolgoznia kell. Ez azt is jelenti, hogy ha az objektum saját magának akar üzenetet küldeni, akkor a *Self.Üzenet(Param)* formát kell, hogy használja, vagyis a metódustörzsekben az adott példányra mindig a *Self* (*this*, *Current*) segítségével hivatkozhatunk. (Ezt nem kell mindig feltüntetni, számos nyelvben ez az alapértelmezett.)

12.4. Egységbezárás

Egy-egy osztályban a megvalósítást leíró adatok és a működést megvalósító metódusok egyetlen közös, zárt egészet alkotnak. Így hatékonyabban modellezik a valóságot, melyben kölcsönösen függnek egymástól.

Az *egységbezárás* (*encapsulation*) azt jelenti, hogy az adatstruktúrákat és az adatokat kezelő metódusokat egy egységként kezeljük, és elzárjuk őket a külvilág elől. Az objektumok állapota és viselkedése, a feladatok elvégzésének *„hogyan"*-ja az objektum belügye. Az objektum belseje sérthetetlen. Az objektum védi belső adatait, nem engedi, hogy idegen objektumok hozzáférjenek, csak saját műveletei dolgozhatnak velük. A műveletek aktuális belső megvalósítása is el van rejtve a rendszer többi része elől.

Ez a megközelítés, ahogy már említettük, tulajdonképpen visszavezet az absztrakt adattípusok elvéhez. Az adatabsztrakciós megközelítés azt jelenti, hogy nem a feladatot írjuk le, hanem a rendelkezésre álló egyszerűbb fogalmak és eszközök segítségével újabb, összetettebb fogalmakat írunk le mindaddig, amíg rendelkezésünkre nem áll az eredeti feladat fogalomterének teljes számítógépes modellje. Ezután a feladatot a saját fogalomterében, a valós világban megfigyelt viselkedéssel megegyezően oldjuk meg. Ez a megközelítési mód illeszkedik az objektumorientált szemlélethez. Ahogy a 10. fejezetben láttuk, a *specifikáció* szintjén megadjuk a fogalom külső leírását, ezen belül is a minket érdeklő jellemzőket (*típusérték-halmaz*) és a számunkra fontos viselkedési formákat (*műveletek*). A második szint a számítógépen történő ábrázolást (*reprezentáció*) és a belső viselkedés (*megvalósítás, implementáció*) megadását jelenti. A fogalmak ilyetén leírása nem csupán a világ egy-egy objektumára korlátozódik, hanem azoknak a feladat szempontjából hasonló tulajdonságokkal bíró

halmazaira. Ezeket a halmazokat, viselkedésük és jellemzőik leírásával együtt *típusnak* vagyis *osztálynak* nevezzük. A viselkedési formákat, a műveleteket, metódusokat elláthatjuk elő- és utófeltételekkel, így elmondhatjuk, hogy az objektumorientált programozás az egységbezárás segítségével megvalósít minden olyan követelményt, amelyet az absztrakt adattípusok elmélete megkövetel (lásd még a 13. fejezetet is).

Számos objektumorientált nyelv az egységbezárást úgy valósítja meg, hogy a korábbi struktúra, illetve rekord (**struct**, **record**) típusát továbbfejlesztve bevezeti az *osztály típust* (**class**). Az osztály típusban is lehetnek adattagok, mint a korábbi rekordokban, de metódusok is ugyanúgy megjelenhetnek benne. Ezt az utat követték például a C++ vagy az Object Pascal tervezői.

Azokból a programozási nyelvekből, amelyeket „teljesen" objektumorientáltan terveztek – ilyenek például a Smalltalk, a Java vagy az Eiffel –, ki is hagyták a direkt szorzat és az unió típuskonstrukciók szokásos megvalósítását (lásd a 6. fejezetet), az *osztály típus* és az *öröklődés* támogatása elegendőnek bizonyult.

Az Ada 95-ben az egységbezárás csak közvetett módon valósítható meg. Az Ada 83 rekord fogalmát bővítették ki, de ezen rekordok csak az osztály attribútumait tartalmazzák, a műveleteket nem. Így az egységbezárás csak a **tagged** rekord típus és a műveletek egy modulba (**package**) helyezésével valósítható meg.

12.5. Adatrejtés, interfészek

Az egységbezárás tulajdonsága azt is jelenti, hogy az objektum elrejti belügyeit, ezáltal más programrészek nem tudják „elrontani" az objektum belsejét, és az objektum belsejében keletkezett hiba sem tud átterjedni más programrészekre. Ahhoz, hogy az objektum kommunikálni tudjon a külvilággal és zárt is maradjon, szükséges valamilyen szűrő felállítása, amely pontosan annyi információt enged a külvilág számára látni, amennyi az objektum biztonságát nem veszélyezteti, de segítségével hatékonyan fel tudjuk használni az objektumot. Ez a szűrő az *interfész*.

Az egységbezárás tehát az alábbi szabályok segítségével valósul meg:

- Az objektum csak olyan üzenetekre reagál, amelyekre megtanították.
- Az objektumot csak az interfészen keresztül lehet elérni.
- Az objektum interfésze a lehető legkisebb legyen.

Módszertani követelmény, hogy az adatok csak a metódusokon keresztül legyenek elérhetők. Bár a nyelvek egy része ad lehetőséget az adatok közvetlen elérésére, ne éljünk vele.

Az interfész *definiálása* a programozó feladata. Ezt úgy tudja elvégezni, hogy az objektum osztályában kifejezetten megadja a külvilág számára elérhető metódusokat (esetleg az adatok egy részhalmazát is). Ezek az adatok és metódusok *nyilvánosak* (*public*). A rejtett adatok és a rejtett metódusok pedig nem nyilvános, *privát* (*private*) elérhetőségűek. A privát elérhetőségű adatokat, metódusokat csak az objektum használhatja, a külvilág számára ezek láthatatlanok és elérhetetlenek. Ideális esetben az, hogy egy adat nyilvános, legfeljebb

lekérdezési lehetőséget biztosít a külső használónak, megváltoztatási, felülírási lehetőséget nem. Ez kizárólag a metódusok feladata. (Ezt valósították meg például az Eiffelben.)

Az objektumorientált programozási nyelvekben általában van lehetőség a nyilvános és a privát adatok és metódusok megkülönböztetésére – gyakran a *public*, illetve a **private** kulcsszavak segítségével. A láthatósági szabályok betartását az egyes nyelvek fordítási időben ellenőrzik, és a szabályok megsértését hibaként jelzik.

Egyes programozási nyelvek a privát és a nyilvános elérésű tagok mellett más hozzáférési szinteket is meghatároznak. Ezek részletes ismertetésére a védett adatrejtési mód tárgyalásánál térünk ki.

A Smalltalk adatrejtési lehetőségei

A Smalltalkban az objektumok belső változóit csak az adott *objektum* műveletei láthatják közvetlenül. Ezzel a Smalltalk eléri, hogy egy osztály reprezentációja rejtve marad a kliensek előtt. Egy kliens csak üzenetküldés útján manipulálhat egy objektumot, így attól, hogy megváltozik a reprezentáció, a klienseknek nem kell feltétlenül megváltoznia. Azonos osztályba tartozó objektumok sem tudják közvetlenül elérni egymás adattagjait – ezt nevezzük *objektumszintű* láthatóságnak.

Minden művelet *mindig* látható, ezért csak ajánlás, hogy a megjegyzésekben **private** szócskával megjelölt műveleteket ne hívjuk meg közvetlenül.

Az adattagok teljesen rejtettek, és ezt nem is lehet megváltoztatni. Ha egy adattagot el akarunk érni, akkor az írásához és olvasásához egy-egy metódust kell definiálnunk. (Ezek neve lehet ugyanaz, mint az adattagnak.) Például:

```
Object subclass UjOszt
instanceVariableNames: 'anInt'
classVariableNames:''
poolDicitonaries:''
...
anInt
  ^anInt.
...
anInt:a
  anInt:=a.
```

Az elérhetőség szabályozása a C++-ban

A C++-ban az osztályok definíciójában a `public:` minősítés után álló adattagok és metódusok nyilvánosak, a `private:` minősítésűek rejtettek az osztály kliensei felé. A láthatóság osztályszintű, ez azt jelenti, hogy azonos osztályba tartozó objektumok el tudják érni közvetlenül egymás rejtett adattagjait is. Osztályok (`class`) esetén a `private` az alapértelmezés[5],

[5] Rekord típusok (`struct`) használatakor az alapértelmezés az, hogy minden adattag és metódus nyilvános (`public`).

a nyilvános és rejtett részek tetszés szerinti sorrendben megadhatók. Ennek megfelelően a következő osztálydefiníciók egyenértékűek:

```
class A {
    int i;
  public:
    void set_i(int n) { i = n; };
    int get_i() { return i; };
};

class A1 {
  private:
    int i;
  public:
    void set_i(int n) { i = n; };
    int get_i() { return i; };
};

class A2 {
  public:
    void set_i(int n) { i = n; };
    int get_i() { return i; };
  private:
    int i;
};
```

Ajánlás [Str 00], hogy az adattagok ne legyenek nyilvánosak, hanem rejtettek, vagy védettek (lásd a 12.7.1. pontot).

Az Object Pascal adatrejtése

Az Object Pascalban **private** kulcsszóval az objektumok belső használatú részeit jelöljük, de ezek ténylegesen csak az osztályt deklaráló egységen (forrásfájlon) kívül nem láthatóak. A módszertani elvárásoknak megfelelő adatrejtést csak akkor tudjuk megvalósítani, ha minden osztályt önálló fájlba teszünk.

A szokásos módon itt is a **public** kulcsszó jelöli a bárki számára látható részeket.

Az alapértelmezés a **published** láthatóság, ezek a mezők vagy metódusok nem csak futásidőben érhetők el, hanem a tervezés alatt is. A Delphi komponenspalettáján lévő összes komponensnek van **published** interfésze, amit különböző Delphi eszközök használnak, például az Object Inspector. Egy osztály minden olyan mezőjénél, ahol nem adunk meg más kulcsszót, a **published** lesz érvényes.

A nyelv ugyan megengedi, hogy az adatmezőket nyilvánosként adjuk meg, célszerűbb azonban (ahogy Marco Cantù is javasolja [Can 00]) ezek helyett inkább tulajdonságokat használni (lásd a 12.1.1. pontot).

A Java hozzáférési kategóriái

A Java osztályaiban is a **public**, illetve a **private** kulcsszóval jelölhetjük a nyilvános, illetve a külvilág elől rejtett tagokat.

Az alapértelmezés – ha mást nem írunk – az úgynevezett „félnyilvános" láthatóság: ez az azonos csomagban definiált osztályok számára is enged hozzáférést [Nyé 01].

A Javában a láthatóság osztályszintű, vagyis azonos osztályba tartozó objektumok elérik egymás rejtett tagjait is.

Az Eiffel szelektív láthatósága

Az Eiffel nyelv fejlesztői más nyelvekben ritkán használt technikát, a *szelektív* láthatóságot valósították meg. Ez azt jelenti, hogy *egy osztály minden egyes jellemzőjére* (**feature**) külön megadható, hogy mely osztályok érhessék el. Eiffelben a láthatóság objektum- és nem osztályszintű, így ha azt akarjuk, hogy az adott osztály más példányai is elérhessenek egy bizonyos jellemzőt, akkor ezt is explicite meg kell adnunk a láthatósági szabályokban.

A láthatóságra vonatkozó kívánalmakat úgy adhatjuk meg, hogy a **feature** kulcsszó után kapcsos zárójelek között felsoroljuk azon osztályoknak a nevét, melyek számára az itt következő jellemzőket láthatóvá szeretnénk tenni. Két speciális szót is használhatunk itt: az egyik az alapértelmezett, ezért el is hagyható *ANY* (mindenki számára látható), a másik a *NONE* (az adott objektumon kívül nem látható).

```
class C
  feature {ANY}
    x : T; -- minden osztály láthatja
  feature {C}
    y: U; -- a C osztály példányai láthatják
  feature {NONE}
    z : V; -- csak az adott objektumon belül látható
end -- class C
```

12.5.1. Friend metódusok, osztályok

Az egységbezárás egy nagyon fontos tulajdonságot határoz meg – azt, hogy az objektummal csak az interfészen keresztül lehet kommunikálni, a belső adatokhoz nem lehet hozzáférni. Ezért rögzíti az objektumorientált megközelítés a láthatósági területet és az adatvédelmi elveket. A rejtett adattagokhoz így csak az osztály metódusai férhetnek hozzá, a külvilág pedig csak a publikus metódusokat hívhatja meg.

A C++-ban kompromisszum született arra – mert szükség volt rá, habár nem igazán illik az objektumorientált szemlélet koncepciójába –, hogy egy másik osztály metódusainak, vagy egy külső eljárásnak meg lehessen engedni azt, hogy bizonyos esetekben hozzáférjen az objektum adataihoz. Így meg lehet valósítani az ismeretségi kapcsolatokat.

Ezeket az eljárásokat, metódusokat „barát"(**friend**)-eljárásoknak, metódusoknak nevezzük. Ezek a metódusok már nem az osztály objektumainak küldött üzenetek, hanem külső függvények, így itt már nem érvényes az, hogy a *Self* segítségével látják az objektumot.

Ennek megfelelően az aktuális objektumot is meg kell adni egy további paraméterben. Az az eset is megengedett, hogy egy osztály barátnak nyilvánítson egy másik osztályt. Ekkor a barát osztály (**friend**-osztály) metódusai elérhetik az illető osztály privát adatait.

A barátság kifejezésére a következő deklarációs részek születtek az osztálydeklaráción belül:

- Eljárásra: **friend** *Eljárásnév(paraméterlista);*
  (Az összes ilyen nevű eljárásnak engedélyezi a privát adatokhoz való hozzáférést.)
- Egy másik metódusra: **friend** *Osztály.Metódus(paraméterlista);*
- Egy másik osztályra: **friend** *Osztálynév;*

C++ példa

A C++ nyelvben [Str 00] lehetőség van az operátorok átdefiniálására. Ennek segítségével létrehozhatunk például egy olyan osztályt, amely komplex számok ábrázolását valósítja meg és átdefiniálja az alapműveleteket.

Elvárásunk az osztállyal szemben, hogy két komplex számot, illetve valós és komplex számot is össze tudjon adni *bármely sorrendben.*

Ha *a* és *b* az osztály példányai, akkor *a + b* esetén meghívódik az *a* objektum + operátora a *b* paraméterrel. Az *a + 2.5* esetén is hasonló történik, de a paraméter a *2.5* lesz. Nehezen értelmezhető viszont a *2.5 + a* kifejezés, hiszen a *2.5* **double** típusú objektum + operátora nem hívható meg *komplex* paraméterrel. Egy lehetséges megoldás a **friend** minősítés használata az összeadó függvényre:

```
class komplex {
      double re, im;
    public:
      komplex(double r = 0, double i = 0) { re = r; im = i; };
      komplex operator+(komplex p);
      komplex operator+(double p);
      friend komplex operator+(double, komplex);
};

komplex komplex::operator+(komplex p) { . . . };

komplex komplex::operator+(double p)      // Az osztálymetódus
{
    komplex temp;
    temp.re = re+p;
    temp.im = im;
    return (temp);
};
```

```
komplex operator+(double p1, komplex p2) // A barát függvény
{
    komplex temp;
    temp.re = p1+p2.re;
    temp.im = p2.im;
    return(temp);
}

int main()
{
    komplex c1,c2;
    c1 = c2 + 2.5;    // Az osztály metódusa.
    c2 = 2.5 + c1;    // A barát függvény.
    return 0;
}
```

12.6. Osztályadat, osztálymetódusok

Az eddigiek során azt mondtuk, hogy az osztály leíró, deklaratív egyed, míg az objektum az osztály példánya. Mi konkrétan objektumokkal dolgozunk és nem osztályokkal. Ha szükségünk van egy adott objektumra, akkor példányosítunk és az objektum interfészén keresztül, üzenetküldés segítségével befolyásoljuk az objektum működését egy adott cél elérése érdekében.

Világos tehát, hogy az adatokat az objektum tartalmazza és az adatok pillanatnyi értékei meghatározzák az objektum egy-egy állapotát. Ezért ezeket az adatokat *példányadatoknak* nevezzük. A metódusok az objektum adatain fejtik ki a hatásukat és a metódusokat egy *Objektum.Metódus(Paraméterek)* üzenettel lehet meghívni, ezért ezeket a metódusokat *példánymetódusoknak* nevezzük.

Az osztály szerepe, hogy leírja a típust, ennek alapján hozzuk létre az egyes objektumokat, lefoglalva az adataik számára a megfelelő memóriahelyet, és a *Self* paraméter segítségével a metódusok mindig tudják majd, hogy melyik objektum adataival dolgozzanak.

Egyes programozási nyelvek – ilyen például a Smalltalk – tervezői szerint a világon minden objektum. S ha minden objektum, akkor természetesen az osztályok is objektumok, s mint ilyenek rendelkez(het)nek belső állapottal (attribútumokkal) és metódusokkal.

Az objektumok ugyanakkor osztályok példányai is, kérdés, hogy ezek az osztály-objektumok milyen osztály példányai? A Smalltalk válasza az, hogy minden osztályhoz létrehoz egy „metaosztályt", amelynek egyetlen lehetséges objektuma az adott osztály. Az előző kérdés természetesen ezekre a metaosztályokra is feltehető – hiszen ezek is objektumok – de melyik osztály objektumai? A metaosztályok egy külön erre a célra bevezetett speciális osztály, a *MetaClass* objektumai lesznek.

Az osztályok, mint objektumok belső állapotát leíró adatokat *osztályadatoknak* hívjuk. Ezek az adatok az osztályról tárolnak információkat és így nem feltétlenül változnak az egyes objektumok állapotváltozásaival együtt. Helyük az osztály helyfoglalásakor foglalódik le és

az osztály helyének felszabadulásakor szabadul fel. Egyes programozási nyelvekben (például a C++-ban, a Javában, a C#-ban) ezek az osztályváltozók úgynevezett *statikus* (**static**) adatok és az osztály objektumai is tudják használni őket. Egy adott osztályváltozóból mindig pontosan egy létezik és az osztály minden egyes példánya ezen „osztozik". Például egy osztály esetén egy osztályváltozóban számolhatjuk az ehhez az osztályhoz létrehozott objektumokat.

Létezhetnek speciális metódusok is, amelyek hatásukat nem az objektum adatain fejtik ki, hanem az osztályon, illetve az osztályadatokon. Ezek a metódusok az *osztálymetódusok*. Természetesen az osztálymetódusok nem használhatják az egyes objektumok adatait, hiszen azokat nem is ismerik. Az osztálymetódusoknak nincs a *Self* (**this**) paramétere – az osztály műveleteivel kapcsolatban nincs értelme kijelölni az aktív példányt. Az osztálymetódusok akkor is végrehajtódhatnak, amikor az adott osztálynak nincsenek példányai. Például, ha a nyugdíjkorhatárt az *Alkalmazott* osztály osztályváltozójaként definiáljuk, akkor ennek változtatása történhet olyankor is, amikor az osztálynak éppen nincs objektuma.

Az osztálymetódusokat általában az osztály nevével minősítve lehet meghívni: *Osztály.Metódus(Paraméterek)* vagy *Osztály::Metódus(Paraméterek)* üzenettel, de objektumnak is küldhetők üzenetként. Ha nem minősítünk egy osztálymetódus-hívást, akkor mindig az aktuális objektum példány osztályának megfelelő osztálymetódus hívódik meg.

Az osztályváltozók kezdőértéket csak egyszer, az osztály inicializációja alkalmával kapnak, és az inicializáció egymáshoz képest előfordulási sorrendben történik.

Smalltalk

A Smalltalkban, ahogy a 12.1.1. pont példájában is láttuk, új osztály létrehozásánál az

```
instanceVariableNames:...
```

és a

```
classVariableNames:...
```

záradékok kötelezően következnek egymás után, itt adhatjuk meg az adott osztály példányváltozóinak és osztályváltozóinak nevét.

Osztálymetódusokat és példánymetódusokat is definiálhatunk[6], a rendszer az osztálynév után írt **class** kulcsszóval különbözteti meg ezeket egymástól:

```
⟨osztálynév⟩ class methods
  ⟨metódusnév_1⟩ ⟨metódustörzs_1⟩
  ⟨metódusnév_2⟩ ⟨metódustörzs_2⟩
  ...
⟨osztálynév⟩ methods
  ⟨metódusnév_1⟩ ⟨metódustörzs_1⟩
  ⟨metódusnév_2⟩ ⟨metódustörzs_2⟩
  ...
```

Az objektumok létrehozása mindig az *osztálynak* küldött üzenettel történik.

[6] Ezeket a Smalltalk fejlesztői környezetben teljesen külön menüpont segítségével adhatjuk meg.

C++, Java, C#

A C++-ban, Javában és C#-ban osztályváltozókat és osztálymetódusokat a **static** minősítővel hozhatunk létre. Használatukra az általános szabályok érvényesek.

Az alábbi példa C++-ban [Str 00] mutatja be az osztályváltozó és az osztályművelet definiálását és használatát. A *Datum* osztálynak – és nem az egyes objektumoknak – jellemzője egy alapértelmezett dátum, és ennek beállítása. Ezt használhatjuk az egyes objektumok létrehozásakor, mint alapértelmezett értéket:

```
class Datum {
    int ev, ho, nap;
    static Datum alapert_datum;
public:
    Datum(int ee=0, int hh=0, int nn=0);
    //...
    static void beallit_alapert(int,int,int);
};

Datum::Datum(int ee, int hh, int nn)
{
    // Ha paraméter nélküli konstruktor hívás történik,
    // akkor az alapert_datum értéket kapja meg az új objektum.
    ev = ee ? ee : alapert_datum.ev;
    ho = hh ? hh : alapert_datum.ho;
    nap = nn ? nn : alapert_datum.nap;
};

void Datum::beallit_alapert(int e, int h, int n)
{
  // A statikus adattag értékének megváltoztatása.
  Datum::alapert_datum = Datum(e, h, n);
}

// A statikus adattag definiálása.
Datum Datum::alapert_datum(2003, 3, 16);
```

Object Pascal

Az Object Pascalban azt, hogy egy műveletet osztálymetódusként szeretnénk használni, az osztály definíciójában a **class** kulcsszóval jelezhetjük [Can 00]:

```
type
  A = class
    procedure B;
    class function ObjSzam :Integer;
  end;
```

Itt a *B* az osztály objektumainak metódusa, míg az *ObjSzam* az *A* osztálymetódusa. Jó tudni, hogy osztályváltozó létrehozására (egyelőre) nincs külön kulcsszó – a hatást szimulálhatjuk azzal, hogy az osztályt tartalmazó egység (**unit**) implementációs részében, rejtetten létrehozunk egy változót, amiből csak egyetlen példány jön létre, és amit az osztály összes objektuma, valamint az osztálymetódusok közösen használhatnak. Ez a változó kaphat kezdeti értéket is:

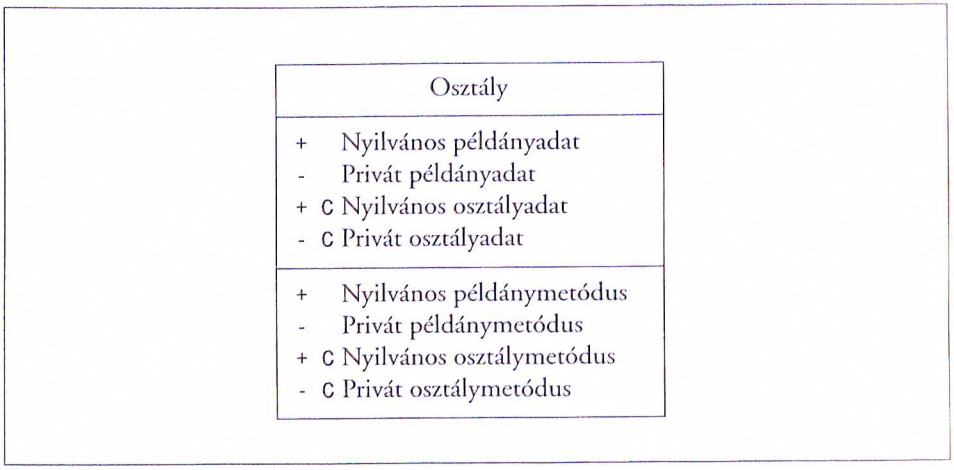
```
implementation
 var
   ObjSz : Integer = 0;

 class function A.ObjSzam :Integer;
  begin
    Result := ObjSz;
 end;
```

Az *ObjSzam* osztálymetódus törzsében használhatjuk az *ObjSz* rejtett változót.

Osztálydiagramok

Az osztálydiagramoknál is fel szoktuk tüntetni, ha egy adat osztályadat vagy ha egy metódus osztálymetódus, amit az adat- vagy metódusnév elé írt `C` betű jelez, a *class* rövidítéseként. A + jel a nyilvános, a - jel a rejtett elérést jelenti (lásd a 12.5. ábrát).

Az egyszerűség kedvéért a példányadatokat *adatoknak*, a példánymetódusokat *metódusoknak* nevezzük és csak az osztályadat, osztálymetódus fogalmakat különítjük el.



12.5. ábra: Osztályadatok, osztálymetódusok

12.7. Öröklődés

Az objektumorientált megközelítés, ahogy eddig is láttuk, a valós világ leírásakor az objektumokat bizonyos tulajdonságaik azonossága alapján *osztályokba* sorolja. Például megkülönböztetjük az élőlényeket az élettelenektől. A következő lépés, hogy az osztályokon belül alosztályokat hozhatunk létre. Így például az élőlényeket különböző alosztályokba csoportosíthatjuk – ahogy az állatrendszertani leírás is mutatja: amikor egy ismeretlen állatot szeretnénk besorolni, eldöntjük, hogy a puhatestűek, az ízeltlábúak, a gerincesek stb. törzsébe tartozik-e, majd, ha úgy találjuk, hogy ízeltlábú, akkor megvizsgáljuk, hogy a rákok, rovarok vagy pókszabásúak osztályába esik-e, s ha rovar, akkor szárnyas-e vagy szárnyatlan. A szárnyas rovaroknak is sok alosztálya van: a pillangók, legyek, bogarak stb.

Minden kategóriának számos alkategóriája lehet, s ezek alatt az alkategóriák alatt is további alkategóriák vannak. Minden szint speciálisabb, mint az őt megelőző, ugyanakkor az is igaz, hogy ha egy tulajdonságot (jellemzőt) definiáltunk, akkor minden alosztály ez alatt a szint alatt tartalmazza ezt a jellemzőt. Vagyis, ha például egyszer már azonosítottunk egy élőlényt, mint a lepkék rendjébe tartozót, azután már nem szükséges külön kiemelnünk, hogy ízeltlábú – ez következik abból, hogy a lepkék az ízeltlábúak közé tartoznak.

A fentieknek megfelelően más esetekben is általánosan leírhatjuk, hogy mi jellemzi az egyes osztályokat, milyen alosztályaik vannak [Mey 00, Ada 95, Pas 89].

Az alosztályok létrehozásánál mindig támaszkodunk annak az osztálynak a tulajdonságaira, amelyiknek az új alosztály része. Ezt úgy is mondhatjuk, hogy az *alosztály örökli az ősosztály tulajdonságait*. Természetesen az ősosztály is lehet, hogy egyúttal egy másik osztály alosztálya, vagyis tulajdonságainak egy részét szintén örökölte. Ezáltal az alosztályok leírása leegyszerűsödik, hiszen az öröklött jellemzőket nem kell mindig leírni, egyszerűen újra felhasználjuk a már adott osztályokat.

Ekkor az új alosztályt a kiindulásul vett osztály *örökösének*, illetve a kiindulásul vett osztályt az új osztály *szülőjének* nevezzük. Az öröklést tehát úgy tekinthetjük, mint osztályok között egy alosztályképzési műveletet. Az osztályok így öröklődési hierarchiába szervezhetők.

Így, ha például a *Tanuló* osztály az *Ember* osztály örököseként jött létre, akkor a *Tanuló* objektumaira igaz, hogy azok egyúttal *Ember* típusúak is. Ez azt jelenti, hogy egy *Tanuló* típusúként deklarált *b* objektumot tekinthetünk *Ember* típusúként is. Fordítva természetesen nem igaz az állítás, egy *Ember* osztálybeli objektum nem feltétlenül esik egyúttal a *Tanuló* által leírt altípusba is. Ezt a lehetőséget, nevezetesen, hogy egy változó több osztály objektumaként is tekinthető, nevezzük a változók *polimorfizmusának* (többalakúságának), erről részletesebben a 12.7.2. alfejezetben írunk.

Az előbb vázolt alosztályképzési mechanizmus voltaképpen olyan eszközt igényel, amelynek segítségével moduljaink egyszerre zártak és nyíltak. Zártak, hiszen önmagukban leírnak egy osztályt – mondjuk az ízeltlábúak, vagy a grafikus alakzatok osztályát –, a megfelelően definiált műveletekkel, de ugyanakkor nyíltak, s ezzel lehetővé teszik, hogy segítségükkel az eredeti osztály alosztályait, specializációit hozzuk létre, – például a rovarok, a sokszögek, vagy a körök osztályait – úgy, hogy a leírásnál nem kell mindent elölről kezdeni, felhasználhatjuk az eddigi osztályokat. A zárt modulok által nyújtott, az interfész-

ben leírt szolgáltatásokat más modulok egyszerűen felhasználhatják, s ha lehetőségünk van rá, hogy ugyanezen modulokat nyíltként használjuk, ez eszközt ad kezünkbe az öröklődés megvalósítására.

A specializációt gyakran úgy valósítjuk meg, hogy az egyes alosztályokba további attribútumokat vezetünk be – például, ha az *Emberek* osztályt az *Élőlények* osztályból származtatjuk, akkor természetesen kibővíthetjük az *Élőlények* osztályt új attribútumokkal, mint az *intelligencia* vagy az *anyanyelv*, de ezzel ugyanakkor leszűkítjük a lehetséges objektumok halmazát, az *Emberek* osztály típusértékhalmaza részhalmaza lesz az *Élőlények* osztály típusértékhalmazának.

Lehetőséget kell biztosítani arra is, hogy az egyes alosztályokban bővítsük a lehetséges műveletek halmazát olyan új műveletekkel, amelyeket az ősben még nincs értelme bevezetni. Egy tipikus példa lehet erre a matematikai gyűrű, amit úgy definiáltak, hogy a matematikai csoport tulajdonságait bővítették. A csoportnak van egy művelete, mondjuk a '+', néhány tulajdonsággal. A gyűrű egy csoport, tehát neki is van '+' művelete, ugyanazokkal a tulajdonságokkal, de bevezet egy új műveletet is, a '*'-t, új tulajdonságokkal [Mey 00].

Az altípusképzés során megengedjük a változtatást is, vagyis azt, hogy az új osztályokban az egyes műveletek átdefiniálhatók legyenek, implementációjuk, esetleg specifikációjuk is különbözzön az ősosztálybelitől. Például az *Alakzat*-ok osztályának lehet egy *Kirajzol* művelete, amely egy általános grafikus alakzat képernyőre való kirajzolását eredményezi, jogos elvárásunk, hogy ennek az osztálynak például a köröket, vagy a sokszögeket leíró örököseinél a *Kirajzol* művelet egy kört, illetve sokszöget rajzoljon ki. Dinamikus összekapcsolás kell lehetőséget adjon arra, hogy a változó aktuális típusa – futás közben – határozza meg a művelet végrehajtandó implementációját. A műveletek átdefiniálása a leszármazottakban természetesen nem történhet tetszőleges módon, ahogy ezt még a továbbiakban megvizsgáljuk.

Ez a megközelítés az, amit *specializáció öröklésnek* vagy *specifikáció öröklésnek* nevezznek [Sik 01].

Egy meglévő osztályt bármikor felhasználhatunk más osztályok definiálására egyszerűen azzal a céllal is, hogy a már meglévő kódot újra fel tudjuk használni. Ez akkor valósul meg, ha a leszármazott osztály implementációjában, rejtetten használjuk az ősosztályt. Ezt nevezhetjük *újrafelhasználó öröklésnek* [Sik 01], vagy *implementációs öröklődésnek* [Mey 00] Tervezési szempontból ennél valójában jobb egy rejtett belső objektum használata.

Az öröklődés lehet *egyszeres* vagy *többszörös*. Egyszeres öröklődésről akkor beszélünk, ha a leszármazott osztálynak *pontosan egy* közvetlen ősosztálya van. Többszörös öröklődés esetén a leszármazottnak legalább két közvetlen ősosztálya van. A leszármazott osztály mindig *örökli* az ősosztály adatait és metódusait. A többszörös öröklődés kezelése gondot okozhat, ha például két közvetlen ősosztályban is szerepel egy-egy ugyanolyan nevű adat vagy metódus, akkor a leszármazott osztályban ez többértelműséghez vezethet – erről bővebben a 12.7.5. részben lesz szó.

Az öröklődés megvalósítása SIMULA 67-ben

A SIMULA 67 volt az első programozási nyelv, amely eszközöket adott az öröklődés megvalósítására. Egy osztálydefiníciót (vagy blokkot) egy már meglévő osztállyal „prefixelve"

megkapjuk a prefix osztály – mai szóhasználattal: a szülő – minden attribútumát és metódusát. Ezeket tetszés szerint kibővíthetjük, illetve átdefiniálhatjuk. Ez egyszeres öröklődési lehetőséget jelent.

Például, ha van egy *Pont* osztályunk, ahol az egyes pontokat a kordinátáikkal adjuk meg, és alkalmazhatjuk rájuk a *Kiir* és *Eltol* műveleteket, akkor ennek leszármazottjaként – ezzel az osztállyal prefixelve – definiálhatunk egy *Szines_Pont* osztályt, ahol az objektumokban még a színüket is tároljuk, és ezt az információt is kiiratjuk.

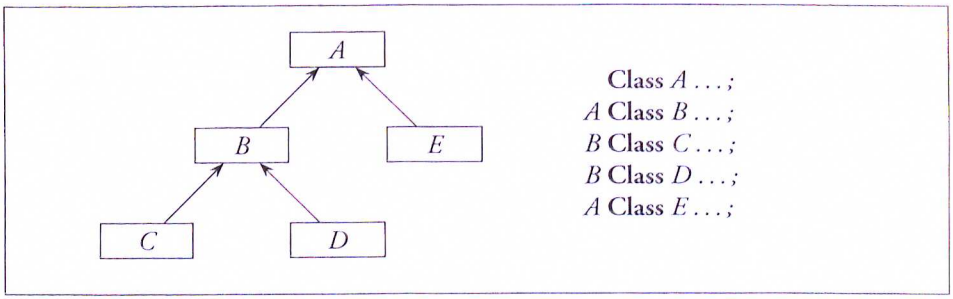
```
Class Pont(X, Y); Real X,Y;   ! koordináták;
 Begin
   Procedure Kiir;
   Begin
     OutText("Pont: ");
     OutFix(x, 1, 4); OutText(","); OutFix(y, 1, 4);
   End of Kiir;
   Procedure Eltol(Dx, Dy);   ! Dx, Dy az eltolás értékei;
   Real Dx,Dy;
   Begin
     X := X + Dx; Y := Y + Dy;
   End of Eltol;

   Kiir;                      ! A pont élete.;
   OutText(" letrehozva"); OutImage
 End of Pont;

Pont Class Szines_Pont(C); Character C;
        ! A Pont-ot használjuk a Szines_Pont deklarálására. ;
        ! Egy további attribútumot definiáltunk. Az élete megváltozik. ;
 Begin
   ! A Szines_Pont élete követi a pont életét: ;
   OutText("Szin "); OutChar(C); OutImage
 End of Szines_Pont;
```

A programunkban definiálhatunk *Pont* és *Szines_Pont* típusú referenciákat, s amikor az egyes objektumokat létrehozzuk, a törzsük lefut.

```
Ref(Szines_Pont) A;
Ref(Pont) C;
C :- New Pont(5, 6);
A :- New Szines_Pont(3, 4, 'G');
```

Az előző utasítások után a képernyőn megjelennek az egyes objektumok kiiratásai:

```
Pont:  5.0, 6.0 letrehozva
Pont:  3.0, 4.0 letrehozva
Szin G
```



12.6. ábra: Prefixlánc SIMULA 67–ben

Az öröklődés megvalósításában egy különleges lehetőség az *inner* kulcsszó használata. Ennek helye a programkódban meghatározza, hogy mikor hívjuk meg a belőle származott osztály blokkját. Bármely leszármazott osztály objektumában a végrehajtáskor először az öröklődési fájában, az általa kijelölt úton található legősibb osztály blokkja hajtódik végre, majd annak a végén, vagy az abban található *inner* kulcsszó elérésekor lépünk át a leszármazottak kódjára.

A prefixek alkalmazásával akár *prefixláncok* is felépíthetők. Ha $C_1, C_2, \ldots, C_n$ osztályok és C_1-nek nincs prefixe és a C_k osztálya prefixe a C_{k-1} osztály ($k = 2, \ldots, n$), akkor a $C_1, C_2, \ldots, C_{k-1}$ osztályok a C_k osztály prefixláncát alkotják. Kikötötték, hogy egy osztály nem szerepelhet a saját prefixláncában.

Öröklődés a Smalltalkban

Smalltalkban az öröklődést, ahogy a 12.1.1. pontban már láttuk, az ősosztálynak küldött `subclass:` üzenettel valósították meg. Az alábbi példában a `Madar` osztály az `Allat` osztály leszármazottja [Sma 91].

```
Object subclass: #Allat
  instanceVariableNames:
    'nev tudas viselkedes maxSeb szin '
  classVariableNames: ''
  poolDictionaries: '' !
...
 Allat subclass: #Madar
  instanceVariableNames: 'repulo '
  classVariableNames: ''
  poolDictionaries: '' !
...
```

C++ példa öröklődésre

A *Sikidom* osztályban vezessük be a *szelesseg* és *magassag* jellemzőket, valamint az értéküket beállító metódust. Ennek örökösei a *Teglalap* és a *Haromszog* osztályok.

```
class Sikidom {
  protected:
    int szelesseg, magassag;
  public:
    void beallit_ertekek(int a, int b) { szelesseg = a; magassag = b; }
};

class Teglalap: public Sikidom {
  public:
    int terulet() { return (szelesseg * magassag); }
};

class Haromszog: public Sikidom {
  public:
    int terulet() { return (szelesseg * magassag / 2); }
};
```

A *Teglalap* és *Haromszog* osztály is rendelkezik a *Sikidom* osztályban bevezetett adattagokkal és metódussal. A *terulet()* metódus új mindkét leszármazott osztályban, más-más módon definiálva. Ennek a metódusnak a bevezetése egy általános tervezési kérdést is felvet: úgy tűnik, talán célszerűbb lenne már az ősosztályban – jelen esetben a *Sikidom* osztályban – összegyűjteni azokat a metódusokat, amivel a leszármazottak mindegyike rendelkezni fog, még akkor is, ha ezen a szinten nem tudunk tényleges törzset rendelni hozzá. Erre a kérdésre a polimorfizmus és dinamikus kötés, valamint az absztrakt osztályok kapcsán visszatérünk majd (lásd a 12.7.2. és a 12.7.4. pontokat).

A C++ megengedi a többszörös öröklődést is, ha például az előzőeken túl bevezetjük az *Egyenlooldalu* osztályt is, akkor a *Negyzet* osztály lehet ennek és a *Teglalap* osztálynak a leszármazottja:

```
class Egyenlooldalu {
public:
    bool e;
};

class Negyzet : public Teglalap, public Egyenlooldalu {
    Negyzet() { e = true; };
};
```

A többszörös öröklődés kérdéseivel bővebben a 12.7.5. pontban foglalkozunk.

Öröklődés az Object Pascalban

Az Object Pascal az egyszeres öröklődést támogatja, interfészek (lásd a 12.7.6. pontot) segítik a többszörös öröklődést. A szintaxist az alábbi példa mutatja, a *TAllat* osztálynak leszármazottja a *TKutya*, itt vezetjük be a kutyákra jellemző *ugat* függvényt.

```
type
  TAllat = class
  public
    function eszik : string;
  private
    fajta : string;
  end;
  TKutya = class (TAllat)        { TAllat leszármazottja. }
  public
    function ugat : string;
  end;
var
  Kutya1 : TKutya;
begin
  { ... }
  writeln(Kutya1.eszik);
  writeln(Kutya1.ugat);
  { ... }
end.
```

Öröklődés az Eiffelben

Eiffelben az öröklődési kapcsolatok az osztály definíciójában az **inherit** kulcsszó után adhatók meg. A leszármazott az ősének összes adattagját és metódusát elérheti, és adott esetben módosíthatja az öröklődési reláció során. Az öröklődési záradékban az **inherit** kulcsszó után fel kell sorolni azoknak az ősosztályoknak a nevét, amelyektől örökölni szeretnénk, s az egyes ősök **redefine** záradékaiban kell megadjuk azokat a jellemzőket, amelyeket módosítani szeretnénk az öröklés során. Azok a jellemzők, amelyeket nem említünk meg a **redefine** záradékban, természetesen változatlanul öröklődnek tovább. A nyelv többszörös öröklődést tesz lehetővé, ezt a 12.7.5. részben tárgyaljuk.

```
class SOKSZOG inherit
   ALAKZAT
 feature
   kiir is
     do
      io.putstring ("SOKSZOG VAGYOK");
     end;
end
```

```
class NEGYZET inherit
   SOKSZOG
     redefine kiir
     end
 feature
   kiir is
     do
      io.putstring ("NEGYZET VAGYOK");
      io.new_line;
    end;
end
```

Öröklődés a Javában

A Javában osztályok között egyszeres öröklődést valósítottak meg. Arra, hogy ez a nyelv tervezőinek szándéka szerint alapvetően az altípus reláció megvalósításának eszköze, az *extends* kulcsszó használata figyelmeztet. Az alábbi *Fonok* osztályt a 397. oldalon bevezetett *Alkalmazott* osztályból származtatjuk.

```
public class Fonok extends Alkalmazott {
  final int MAXBEOSZT = 20;
  Alkalmazott[ ] beosztottak = new Alkalmazott[MAXBEOSZT];
  int beosztottakSzama = 0;
  public void ujBeosztott(Alkalmazott b) {
    beosztottak[beosztottakSzama++] = b;
  }
}
```

A Javában lehetőség van arra is, hogy egy osztályt úgy deklaráljunk, hogy neki már ne lehessen leszármazottja. Ehhez a **final** kulcsszó használata szükséges.

A többszörös öröklődés lehetőségét Javában is az interfészek adják, erről a 12.7.6. részben olvashatunk.

Öröklődés a C#-ban

A C#-ban az öröklődés – bár „hivatalosan", és a szintaxis szerint a C++-hoz hasonló –, ténylegesen a Java alapján működik: az osztályok között egyszeres öröklődés lehet, a többszörös öröklődéshez interfészek kellenek. Itt is meg lehet tiltani a további származtatást, csak itt a **sealed** kulcsszót használják erre.

```
class Ososztaly { . . . }
class Szarmaztatott: Ososztaly { . . . }
```

Öröklődés az Ada 95-ben

Ada 95-ben csak az úgynevezett *tagged* rekord típusoknak lehetnek leszármazottaik. A 399. oldalon definiált osztályból a következő módon származtatható új osztály.

```
with Ember_Tip; use Ember_Tip;
package Tanulo_Tip is
   type Tanulo is new Ember with
      record
         Kredit_Atlag : Float;
         Szint        : Integer;
      end record;
   procedure Megjelenit (P : in Tanulo );
end Tanulo_Tip;
```

A példában az *Tanulo* osztály két új nyilvános adattaggal lett kibővítve az *Ember* osztályhoz képest, változatlanul örökli az *Ember* típus nevet, címet, kort beállító és lekérdező műveleteit és átdefiniálja a *Megjelenit* metódusát.

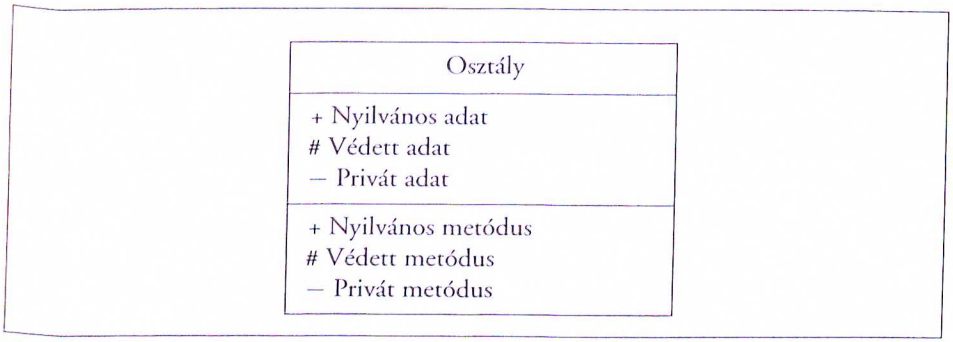
Objektumokat az alábbi módon hozhatunk létre:

```
   P1    : Ember;
   S1    : Tanulo;
   Gabor : Ember;
   Gabi  : Tanulo;
begin
   Nev_Ir(Gabor,"Gabor");
   Cim_Ir(Gabor,"Budapest" );
   Kor_Ir(Gabor,35);

   Nev_Ir(Gabi,"Gabi");
   Cim_Ir(Gabi,"Pomaz" );
   Kor_Ir(Gabi,20);
   Gabi.Kredit_Atlag:=4.5;
   Gabi.Szint:=3;

   Megjelenit(Gabor);
   Megjelenit(Gabi);

   P1 := Ember(Gabi);
   S1 := (Gabor with 3.05, 2);
end;
```

A rejtett részeknek csak a megfelelő beállító művelettel lehet értéket adni, a nyilvános adattagok közvetlenül is beállíthatók. A *Megjelenit(Gabor);* az *Ember* típus primitív műveletét hívja, míg a *Megjelenit(Gabi);* a *Tanulo* típusban átdefiniáltat. Az Adában csak

	külvilág	az objektum	leszármazott
public	elérhető	elérhető	elérhető
protected	nem elérhető	elérhető	elérhető
private	nem elérhető	elérhető	közvetlenül nem elérhető

12.1. **táblázat:** A háromféle védelmi mód hatása



12.7. **ábra:** Nyilvános, védett és privát adatok, metódusok

a konverzió jelölésével helyes a *P1 := Ember(Gabi);* értékadás és az értékadás során *Gabi* átlaga és szintje el fog veszni.

Érdekes lehetőség, hogy az *S1 := (Gabor* **with** *3.05, 2)* értékadásnál a leszármazottban definiált két új adattag értékét külön meg lehet adni, így „kiterjesztve" az objektum típusát.

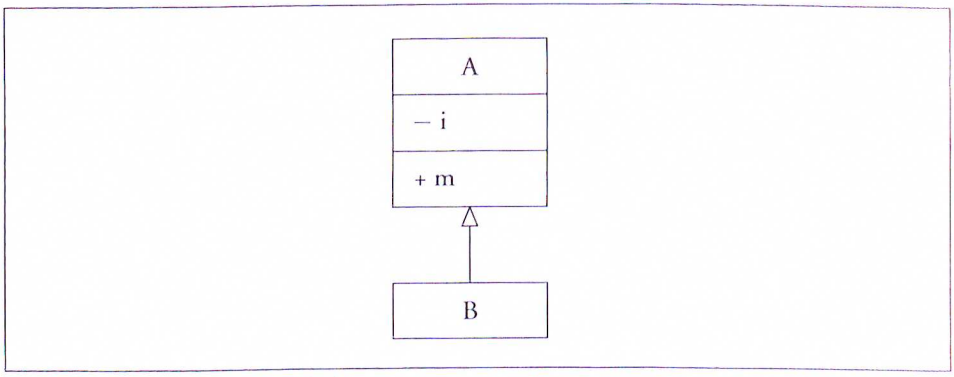
12.7.1. Adatrejtés és öröklődés

A nyilvános (**public**) és privát (**private**) adatrejtési módon kívül számos objektumorientált programozási nyelvben létezik egy harmadik is, amelynél a hozzáférés csak az öröklődésen keresztül valósul meg. Ez az adatrejtési mód a *védett* (**protected**). A védett adattagok és metódusok a külvilág számára éppolyan elérhetetlenek, mint a privátként megjelöltek, az örökös számára pedig éppúgy elérhetőek, mint a nyilvánosnak definiáltak.

A háromféle védelmi mód hatását a 12.1. táblázatban foglaltuk össze. Osztálydiagramoknál ezt a 12.7. ábrán látható módon tüntetjük fel. A # jel mutatja a védett, csak a leszármazott számára hozzáférhető adatokat és metódusokat.

Az öröklődési hierarchián belül a legtöbb programozási nyelvben megmaradnak az adatrejtési elvek, ezért a privátként deklarált adatok, metódusok a leszármazott számára nem lesznek elérhetők, és a leszármazott meghívni sem tudja az ős privát metódusait (helyesebben: természetesen a leszármazott osztály technikailag örökli az ős privát adatait, metódusait, csak közvetlenül használni nem tudja őket: *láthatatlanként* örökli azokat)[7]. A többi adat és metódus elérhető lesz.

[7] Egyes nyelvekben előfordulhat, hogy a rejtett láthatóság hozzáférési szabályai ettől eltérnek – például az Object Pascalban a fentiek csak a különböző egységekben bevezetett osztályok esetén igazak.



12.8. ábra: Adatrejtéses öröklődés

Tegyük fel, hogy az *A* osztály tartalmaz egy nyilvános *m* metódust és egy privát *i* adatot. Feltételezzük, hogy az *m* metódus használja az *i* adatot! Ha a *B* osztály az *A*-ból öröklődik, akkor természetesen örökli az *m* metódust, és ez minden további nélkül használni tudja az *i* adatot, habár az nem lesz látható a *B*-ben, így egyetlen más új *B*-beli metódus sem tudja közvetlenül használni. A rejtettként (**private**-nak) deklarált adatok és metódusok is átöröklődnek, csak ezek nem lesznek láthatók, vagyis elérhetők a leszármazott osztályban. Ezt a 12.8. ábrán látható módon szemléltethetjük.

Számos programozási nyelv megengedi azt is, hogy egyszerű újradeklarálással a leszármazott osztály megváltoztassa az adatok, metódusok adatrejtési módját. Például, ha a leszármazottban nyilvánosként vezettünk be egy, az ősben már eleve védettként bevezetett metódust, akkor ez a művelet a továbbiakban nyilvános lesz.

Öröklési módok

Az öröklődés megadásakor bizonyos programozási nyelvekben – például a C++-ban – lehetőség van arra is, hogy módosítsuk a megadott adatrejtési módokat. Így a leszármazott osztályban felülbírálhatjuk az ős adatrejtési elveit, de megtehetjük azt is, hogy bizonyos adat- és metódus-elrejtéssel megszüntetjük a többszörös öröklődés nehézségeit.

Rejtett öröklési mód megadásával implementáció-öröklés esetén elrejthetjük a külvilág és a későbbi leszármazottak elől is az öröklött jellemzőket. A nyilvános öröklődési mód – a legtöbb nyelvben ez az alapértelmezés – segíti a specifikáció-öröklés támogatását.

A C++ öröklési módjai

A C++-ban alapvetően három öröklési mód, adatrejtési módosító létezik: a **private**, a **public**, és a **protected**, melyek közül a **private** az alapértelmezett. Az öröklési módok a 12.2. táblázat szerint módosítják az adatrejtési módokat.

Megfigyelhetjük, hogy az ősosztálybeli privát adatok, metódusok semmiképp nem válhatnak elérhetővé a leszármazottak számára, s bár ezek továbbra is létezni fognak, közvetlenül elérhetetlenek maradnak. A nyilvános (**public**) öröklési mód esetén a leszármazott

Az ősosztálybeli adatrejtési mód	öröklési mód adatrejtési módosító	A leszármazott osztálybeli adatrejtési mód
`private`	`public`	közvetlenül nem elérhető
`protected`	`public`	`protected`
`public`	`public`	`public`
`private`	`private`	közvetlenül nem elérhető
`protected`	`private`	`private`
`public`	`private`	`private`
`private`	`protected`	közvetlenül nem elérhető
`protected`	`protected`	`protected`
`public`	`protected`	`protected`

12.2. **táblázat:** A C++ öröklési módjai és az adatrejtési módok

megtartja az ősökben meghatározott láthatósági szinteket. A rejtett (**private**) öröklési móddal származtatott osztályban az ősosztály minden adata, metódusa rejtetté válik, így rejtve marad egy esetleges további öröklődés során. A védett (**protected**) öröklési mód esetén az örökölt jellemzőket csak a külvilág elől rejtjük el, a leszármazott felhasználhatja ezeket saját újonnan bevezetett metódusaiban.

Az Eiffel öröklési módjai

Az Eiffelben a leszármazott úgy változtathatja az őseitől örökölt jellemzők láthatóságát, ahogy neki tetszik. Például egy csak az osztályon belül látható jellemzőt nyilvánossá tehet, és fordítva. Mindezt az öröklődési záradék *New_exports* részében tehetjük meg.

Az alapértelmezés az, hogy amit a leszármazott nem változtat meg, annak a láthatóságára az érvényes, ami a szülő osztályban volt.

```
class ROGZITETT_VEREM[T]
  inherit
    VEREM[T]
      -- ...

    TOMB[T]
      rename
        betesz as tomb_betesz,
        elem as tomb_elem
      export
        {NONE} all;
      end
    -- ...
end
```

A *ROGZITETT_VEREM* osztály a viselkedési jellemzőit a *VEREM* osztálytól örökli. Itt nincs szükség a láthatósági viszonyok módosítására. A belső reprezentációját viszont a *TOMB* osztálytól származtatja, és nem szeretnénk, hogy a leszármazott osztályt a *TOMB* jellemzőin keresztül is el tudjuk érni. Emiatt a *TOMB*-től örökölt összes jellemzőt rejtetté tesszük.

12.7.2. Polimorfizmus és dinamikus kötés

A specifikáció öröklődésnél beszélhetünk a *helyettesíthetőségről*. A helyettesíthetőség azt jelenti, hogy a származtatott osztály objektumai bármilyen körülmények között helyettesíthetik az ősosztály objektumait, vagyis a származtatott osztály objektumai felvehetik az ősosztály szerepét, viselkedését, és hasonló környezetben használva nem feltétlenül lehet megkülönböztetni őket az ősosztály valamelyik példányától. Ez a folyamat természetes, mivel a leszármazott osztályban szerepel az ősosztály minden adata és metódusa, így példányai bármikor úgy viselkedhetnek, mint az ősosztálybeliek. Az is igaz kell legyen, hogy ahol az ősosztály szerepel formális paraméterként, ott a leszármazott aktuális paraméterként előfordulhat. A helyettesíthetőség az osztályok közötti **is-a** reláción alapul. Ez kifejezi azt, hogy az őstől a leszármazott irányába *szűkítés* (*specializálás*), a leszármazottól az ős felé pedig *általánosítás* történik.

A specifikáció öröklődés, ahogy eddig is láttuk, osztályok között egy altípus-képzési művelet, így, ha például a *Kör* osztály az *Alakzat* osztály örököseként jött létre, akkor a *Kör* objektumaira igaz, hogy azok *Alakzat* típusúak is. De ha egy *Kör* típusúként deklarált *kiskör* objektumot tekinthetünk *Alakzat* típusúként is, akkor értékül is adhatjuk egy *alak* nevű, *Alakzat* típusú objektumnak, azaz megengedett az *alak := kiskör* értékadás. Ugyanakkor, mivel fordítva nem igaz az állítás, hiszen egy *Alakzat*-beli objektum nem feltétlenül esik a *Kör* által leírt altípusba, ezért a a *kiskör := alak* értékadás nem megengedett. Ez a kapcsolat már így is gondot jelent a hagyományos típusfelfogás szerint, hiszen ha a *kiskör*-t *Kör* típusként, s az *alak*-ot *Alakzat* típusúként deklaráltuk, akkor az *alak := kiskör* értékadás típuskeveredést kellene jelezzen.

A fentiek egyik megvalósítási lehetősége, hogy a programozási nyelvben a változókat típusnélkülinek tekintjük, s tetszőleges értékadást megengedünk – ezt a megoldást választotta például a Smalltalk. Ennél megbízhatóbb, s nyilván ezért is jobban elterjedt az a választás, hogy bevezetjük a *statikus* és a *dinamikus típus* fogalmát. Egy változó *statikus típusa* az az osztály, amellyel a deklaráció során összekapcsoltuk. A változó *dinamikus típusa* pedig – ami azt mutatja meg, hogy futás közben (dinamikusan) milyen osztálybeli objektumra hivatkozhat – a statikus típus, vagy annak egy tetszőleges leszármazottja lehet. Így a fenti példában az *alak* változó statikus típusa az *Alakzat* osztály, míg az *alak := kiskör* értékadás után dinamikus típusa a *Kör* osztály. Ezért van az, hogy a legtöbb tisztán objektumorientált nyelvben az objektumokat implicit referenciaként reprezentálják.

Azt a lehetőséget, hogy egy változó a program futása során több osztály objektumára is hivatkozhat, nevezzük a *változók polimorfizmusának* (többalakúságának).

A polimorfizmusnak többféle formája létezik, erről bővebben a 11. fejezetben olvashatunk. Az objektumorientált programozásban használt polimorfizmus-fogalom lényegében az altípusos polimorfizmust jelenti.

Az öröklődés során megengedjük, hogy a leszármazottaknál az egyes műveletek implementációja és/vagy specifikációja különbözzön az ősosztályban bevezetett művelettől. Pél-

dául, ahogy korábban írtuk, ha a grafikus alakzatok osztályának van egy *Kirajzol* művelete, amely egy általános alakzat képernyőre való kirajzolását eredményezi, akkor ennek az osztálynak a köröket, vagy a sokszögeket leíró örököseinél a *Kirajzol* művelet egy kör, illetve sokszög kirajzolását eredményezi. Mi történjen azonban a fenti esetben? Tegyük fel, hogy az előző példában az *Alakzat* osztályban már létező *Kirajzol* műveletet a *Kör* osztályban felülbíráltuk, azaz más implementációt rendeltünk hozzá. Ha az *alak := kiskör* értékadás után alkalmazzuk az *alak*-ra a *Kirajzol* műveletet, melyiket hajtsa végre? Az *Alakzat* osztályban vagy a *Kör* osztályban meghatározott implementációt? Természetes elvárásunk lenne, hogy a speciálisabb, az adott esetben a *Kör* osztályban meghatározott műveletet végezze el. A *dinamikus összekapcsolás* ad lehetőséget arra, hogy a változó dinamikus típusa határozza meg a művelet végrehajtandó implementációját. A dinamikus összekapcsolás futási idejű esemény. Számos programozási nyelvben azt a műveletet, amely dinamikusan kapcsolódik az adott objektumhoz, *virtuális műveletnek* nevezik.

Lényegében ugyanez a kérdés merül fel abban az esetben is, amikor az *Alakzat* osztályban megvalósítjuk a *Mozgat* metódust oly módon, hogy meghívjuk az osztály *Eltüntet*, *Eltol* és *Kirajzol* metódusait. (Az *Eltol* metódus csak a koordinátákat változtatja meg.) Tegyük fel, hogy a *Kör* osztályban átdefiniáljuk az *Eltüntet* és a *Kirajzol* metódusokat az osztálynak megfelelően. Mi történik, mi történjen, amikor a *kiskör* objektumnak hívjuk az (örökölt) *Mozgat* metódusát? Ez először is meghívja az *Eltüntet* metódust. De melyiket? Az *Alakzat* vagy a *Kör* osztályban definiáltat? A dinamikus összekapcsolás ad lehetőséget arra, hogy várakozásainknak megfelelően a *kiskör* dinamikus típusának megfelelőt, vagyis a *Kör* osztályét hívja. Ezt követően meghívja az *Alakzat* osztályban definiált *Eltol* metódust, majd (megintcsak a dinamikus összekapcsolás miatt) a *Kör* osztály *Kirajzol* metódusát.

Metódusok átdefiniálása a leszármazottban

Vizsgáljuk meg kicsit részletesebben, hogyan definiálhatjuk át a leszármazottban az ősöktől örökölt műveleteket, kell-e valamilyen megszorítást érvényesítenünk, ha az átdefiniálás mellett szeretnénk megőrizni a helyettesíthetőségből származó előnyöket is.

Fontos szempont lesz az is, hogy mindig képesek legyünk megkülönböztetni a műveletek átdefiniálásának lehetőségét a metódusnevek túlterhelésének támogatásától. A túlterhelés azt jelenti (lásd a 7.6. fejezetet), hogy egy program egy adott pontján ugyanahhoz az alprogramnévhez több alprogram is tartozhat. Általában igaz, hogy ha egy programozás nyelv megengedi a metódusnevek túlterhelését, akkor ezek a híváskor megkülönböztethetőek kell legyenek – ez egyszerű akkor, ha mondjuk más a paraméterek száma. A paraméterek típusa alapján történő megkülönböztetés bizonyos helyzetekben már nem is olyan magától értetődő, ahogy majd látni fogjuk.

Statikus és dinamikus átdefiniálás

Több programozási nyelvben – ilyenek például az Object Pascal vagy a C++ – az objektumok metódusai alapértelmezésben statikus kötésűek. Ez azt jelenti, hogy ha a programszövegben előfordul egy *obj.uzenet(...)* alakú hívás, akkor a végrehajtandó kódban ez már helyettesítődik egy, az *obj* változó statikus típusának megfelelő *uzenet* nevű metódus címére mutató hívással. Például az *alak.Kirajzol* esetén az *alak* statikus típusa alapján az *Alakzat*

osztályban, míg a *kiskör.Kirajzol* esetén a *kiskör* statikus típusa alapján a *Kör* osztályban definiált *Kirajzol* művelet kerül meghívásra.

A statikus átdefiniálás nem támogatja az altípusos polimorfizmushoz kapcsolódó dinamikus metódus kiválasztást, ami az objektumorientált programozás lényeges eleme. Dinamikus kötés esetén a meghívott metódus címe ténylegesen csak futási időben derül ki, hiszen ha valamely osztály változójának a leszármazott osztályok objektumai is értékül adhatók, akkor csak futás közben lehet eldönteni, hogy az adott metódus melyik osztálybeli megvalósítását kell meghívni. Ezért az objektumorientált programozási nyelvek valamilyen formában mindenképpen támogatják a dinamikus kötés megvalósítását. Általában elmondható, hogy azoknál a programozási nyelveknél, amelyeket kifejezetten objektumorientáltnak terveztek – mint például a Smalltalk, a Java vagy az Eiffel – ez az alapértelmezés. A programozási nyelvek másik része a dinamikus kötést úgy valósítja meg, hogy egyes metódusokat „virtuálisként" lehet deklarálni, és a virtuális metódusok hívásai lesznek dinamikus kötésűek. Egy-egy új osztály deklarálásakor virtuális metódusainak címei bekerülnek az osztály úgynevezett *Virtuális Metódus Táblájába* (Virtual Method Table, VMT). Virtuális metódus hívása esetén a program futási időben az objektum dinamikus típusa alapján a megfelelő VMT-ből tudja kiolvasni, hogy melyik metódust kell meghívnia.

Altípusok metódusai

Az öröklődés lehetővé teszi, hogy már meglévő típusokból altípusokat hozzunk létre. Például *Kör* $\subseteq$ *Alakzat*. Ez a kapcsolat azonban nem csak a típusértékhalmazokra vonatkozik, hanem általában az altípusba eső objektumok viselkedésére is, vagyis a leszármazottban biztonságosan csak olyan módon definiálhatjuk felül az ősöktől örökölt műveleteket, hogy az altípusbeli objektumokat továbbra is használhassuk minden olyan kontextusban, ahol a szülő osztálybelieket [Bru 02]. Kezelhessük például alakzatok listáját annak műveleteivel, ahol az egyes konkrét alakzatok lehetnek körök, sokszögek, téglalapok stb.

Egy metódust jellemez a szignatúrája, vagyis a paramétereinek és az eredményének a típusa, valamint az elő- és utófeltétele.

Legyenek

faa = **proc**(*Alakzat*) **returns** (*Alakzat*)
fka = **proc**(*Kör*) **returns** (*Alakzat*)
fak = **proc**(*Alakzat*) **returns** (*Kör*)
fkk = **proc**(*Kör*) **returns** (*Kör*)

tetszőleges metódusok, és vizsgáljuk meg, hogy a szignatúra alapján melyiknél lehetséges, hogy a másikat helyettesítse, milyen átdefiniálás engedhető meg a leszármazottban.

$fka \subseteq faa$? $\quad fak \subseteq faa$?
$fkk \subseteq faa$? $\quad faa \subseteq fka$?

Egy függvényt $A \rightarrow B$ alakban írunk, ha A típusú paramétere van és B típusú eredményt ad. Ha az $A' \rightarrow B'$ függvények halmaza altípusa az $A \rightarrow B$ függvényhalmaznak, azaz ha $A' \rightarrow B' \subseteq A \rightarrow B$ altípus, akkor képesek kell legyünk arra, hogy az első függvénytípus egy elemét használhassuk minden olyan kontextusban, ahol a másodikat. Tegyük fel, hogy van egy f függvényünk, ami $A \rightarrow B$ típusú. Ha egy $f' : A' \rightarrow B'$ függvényt az f helyén akarunk használni, akkor ennek is A-beli argumentumot kell fogadnia és B-beli eredményt adnia. Mivel az f' értelmezési tartománya A', így akkor alkalmazható A-beli

elemekre, ha $A \subseteq A'$, vagyis ha $a : A$ tekinthető mint A'-beli, és $f'(a)$ típus-helyes. Másrészt, ha az f' eredménye B'-beli, akkor $B' \subseteq B$ garantálja majd, hogy az eredmény B-beliként legyen kezelhető. Összegezve: $A' \to B' \subseteq A \to B$, ha $A \subseteq A'$ és $B' \subseteq B$.

Ez azt jelenti, hogy átdefiniálásnál az eredmény típusa lehet azonos (novariáns), vagy lehet speciálisabb, mint az ősben (ezt nevezik „együtt változó", kovariáns vagy monoton lehetőségnek, lásd a 11.1. alfejezet), ugyanakkor a paraméterek típusa csak azonos (novariáns), vagy kevésbé speciális lehet (ezt nevezik anti-monotonnak, vagy kontravariánsnak is). Az előző példában így igaz:

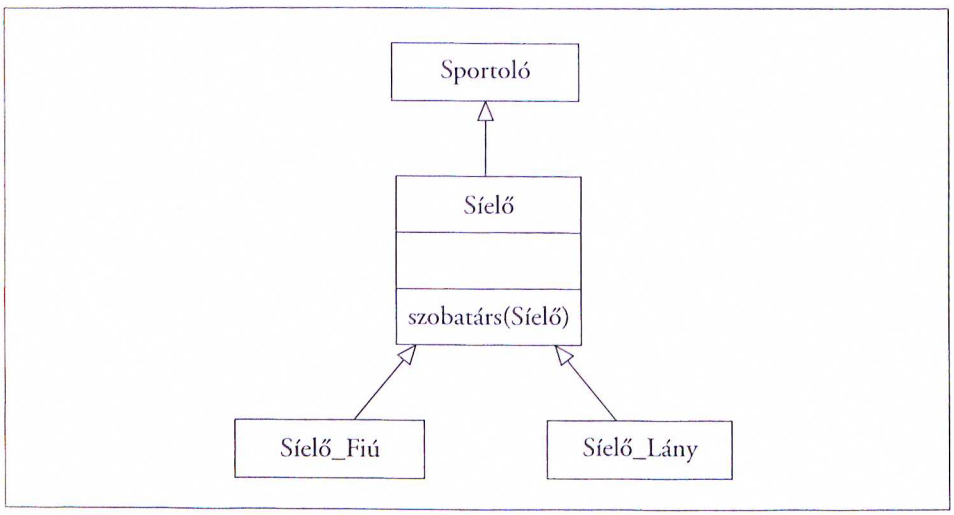
- az *fak* $\subseteq$ *faa* – az eredmény lehet speciálisabb
- az *faa* $\subseteq$ *fka* – a paraméterek lehetnek kevésbé speciálisak

A metódusok elő- és utófeltételeinek átdefiniálására hasonló szabályok érvényesek: az utófeltétel maradhat azonos (novariáns), vagy szűkíthető (kovariáns), de az előfeltétel csak azonos (novariáns) maradhat, vagy esetleg gyengíthető (kontravariáns). Erről részletesebben lásd a 13. fejezetet.

A programozási nyelvek többsége az átdefiniálásnál megköveteli a szignatúra azonosságát, így fordítási időben tudja ellenőrizni a hívást.

Előfordulhat olyan helyzet, amikor igény merül fel arra, hogy a leszármazott valamelyik metódus paraméterében kovariáns módon definiálja át az őstől örökölt jellemzőket. Például származtassuk az *Sportoló* osztálytól a *Síelő*-k osztályát, s tekintsük ennek két alosztályát, a *Síelő_Lány* és a *Síelő_Fiú* osztályokat. Tegyük fel, hogy a *Síelő* osztályban bevezettünk egy *szobatárs* metódust, melynek egy *Síelő* típusú paramétere van.

Hogyan tudjuk a *Síelő_Lány* osztályban felüldefiniálni a *szobatárs* metódust? Van-e mód kovariáns átdefiniálásra (*Síelő_Lány* típusú paraméterre), vagy csak novariáns (tetszőleges *Síelő* lehet továbbra is a paraméter típusa), esetleg kontravariáns (*Sportoló* típusú paraméter) lehetőségeket ad a programnyelv?



12.9. ábra: Síelők osztályhierarchiája

Polimorfizmus és dinamikus kötés a C++-ban

A C++-ban a fordító a **virtual** kulcsszót tartalmazó műveletek esetén ad lehetőséget a dinamikus összekapcsolásra. A leszármazott objektumai értékül adhatóak az ősosztálybeli objektumoknak, de ügyelni kell arra, hogy ha ténylegesen az objektumokat használjuk, és nem referenciákat, akkor ez csonkítást eredményez, és a virtuálisként jelölt műveletek esetében is statikus összekapcsolás történik. A dinamikus összekapcsolás csak referenciák esetén működik helyesen.

Például, vezessük be ha a *Teglalap* osztályban a virtuális *f()* függvényt, ami itt a *0* értéket adja vissza, majd származtassuk belőle a *Szines_Teglalap* osztályt, ahol egész kódokkal tároljuk a *szin*-t, és átdefiniáljuk az örökölt *f()* függvényt, hogy a *szin* értékét adja vissza. Ekkor, ha létrehozunk egy *Teglalap* és egy *Szines_Teglalap* típusú objektumot, a polimorfizmus látszólag megengedett, az ős típusúnak értékül adható a leszármazott, de valójában csonkítás történik, és az *f()* függvényt statikusan köti az objektumhoz. A valódi polimorfizmus és a dinamikus kötés csak akkor megvalósítható, ha a változók címére hivatkozunk, ahogy az alábbi példa is mutatja.

```
class Teglalap {
  protected:
    int szelesseg, magassag;
  public:
    Teglalap(int a, int b) { szelesseg = a; magassag = b; };
    virtual int f() { return (0); }
};

class Szines_Teglalap: public Teglalap {
     int szin;
  public:
     Szines_Teglalap(int a, int b, int c) : Teglalap(a,b), szin(c) {};
     virtual int f() { return (szin); }
};

void main()
{
    Teglalap a(2,3);
    Szines_Teglalap t(3,4,5);
    ...
    a = t;                  // Csonkítás
    cout << a.f();          // Teglalap::f()

    Teglalap* a1 = &a;
    Teglalap* a2 = &t;
    cout << a1->f();        // Teglalap::f()
    cout << a2->f();        // Szines_Teglalap::f()
};
```

A leszármazott az örökölt virtuális metódusokat csak „megbízható" módon definiálhatja át: a paraméterek típusa nem változhat (novariáns), az eredmény típusa lehet „együtt változó" (kovariáns) is. A paraméterek típusának változtatására a túlterhelés ad lehetőséget. A korábban látott *Sielő* osztály *szobatárs* metódusában ha a *Sielő_Lány* leszármazott szeretné a paraméter típusát *Sielő_Lány* típusúra (kovariáns módon) változtatni, megteheti, de az így bevezetett *szobatárs* metódus nem átdefiniálása, hanem túlterhelése lesz az örökölt metódusnak.

```
class sielo {
   public:
      . . .
      virtual void szobatars(sielo * s){
          cout << "sielo szobatarsa sielo\n";
   };
};

class sielo_lany : public sielo {
   public:
      . . .
      virtual void szobatars (sielo_lany *g) {    // Túlterheli.
           cout << "sielo_lany szobatarsa lany\n";
      };
};

class sielo_fiu : public sielo {
   . . .
};

void main()
{
    sielo *s;
    sielo_lany *g;
    sielo_fiu *b;
    g = new sielo_lany;
    s = g;
    s->szobatars (b);    // A sielo::szobatars-at hívja.
    g->szobatars (g);    // A túlterhelt sielo_lany::szobatars-at hívja.
    s->szobatars (g);    // A sielo::szobatars-at hívja.
}
```

Érdemes megjegyezni, hogy ha a leszármazottban nem definiáljuk át az örökölt *szobatars* metódust, akkor a *g->szobatars (b);* hívás esetén a fordítók sokszor nem ismerik fel, hogy az örökölt *sielo*-beli *szobatars* metódust kellene hívni, hanem hibajelzést adnak, míg ha a leszármazottban bevezetjük az örökölt metódus átdefiniált változatát is, akkor gond nélkül tudják értelmezni az előző hívást is.

Az Object Pascal eszközei

Az Object Pascal változói objektumok referenciái, a polimorfizmus azt jelenti, hogy az ősosztálybeli referencia aktuálisan valamelyik leszármazott osztály objektumára is hivatkozhat. A metódusokat alapértelmezésben – a C++-hoz hasonlóan – statikusnak tekintjük. Dinamikus, futási idejű metódus kiválasztáshoz a metódust a **virtual** kulcsszóval virtuálisként kell bevezetni – például: **procedure** *f;* **virtual**. Ha ezt a leszármazottban szeretnénk felüldefiniálni, akkor az **override** kulcsszóval tehetjük meg – például: **procedure** *f;* **override**.

Lehetséges a metódusnevek túlterhelése is, ehhez az **overload** kulcsszót használjuk – **procedure** *f(a:string);* **overload;**. Túlterhelésnél, ha a metódusok paraméterezése különbözik, akkor két azonos nevű metódust használhatunk, azonos paraméterek esetén az újonnan bevezetett metódus eltakarja az örököltet.

Megengedett a virtuális metódusok túlterheléssel történő eltakarása is, ekkor kell a **reintroduce** kulcsszó is – például: **procedure** *f;* **reintroduce; overload;** [Can 00].

Polimorfizmus és dinamikus kötés a Javában

A Javában a változók objektumok referenciái, így polimorfizmusuk is alapértelmezés, s nincs szükség külön kulcsszóra a dinamikus összekapcsolás támogatásához sem. Ha öröklődéskor a leszármazottban egy metódust felüldefiniálunk egy megegyező szignatúrájú metódussal, akkor mindig a dinamikus típusnak megfelelő metódust hívja.

A következő példában [Nyé 01] az *Alkalmazott* osztályban definiáljuk a *potlek* metódust. A *Fonok* osztály leszármazottja az *Alkalmazott*-nak és felüldefiniáljuk benne a *potlek* metódust.

```
public class Alkalmazott {
  . . .
  int nyelvekSzama;
  public int potlek() { return nyelvekSzama * 5000; }
  public int fizetesPotlekokkal()
    { return fizetes() + potlek(); }
}

public class Fonok extends Alkalmazott{
  . . .
  public int potlek()
    { return super.potlek() + beosztottakSzama * 1000; }
}
```

Automatikusan a változó dinamikus típusának megfelelő metódust hívja:

```
Alkalmazott a = new Fonok("Kis Pista");
int p = a.potlek();              // Fonok osztály potlek metódusa.
int fp = a.fizetesPotlekokkal(); // Alkalmazott osztály metódusa.
```

Polimorfizmus és dinamikus kötés a C#-ban

A C#-ban – a Javához hasonlóan – az osztályok objektumait referenciákon keresztül kezelhetjük, ugyanakkor a metódusok – a C++-hoz hasonlóan – lehetnek statikus és dinamikus elérésűek. A dinamikus elérésű metódusok bevezetésekor itt is a **virtual** kulcsszót használjuk:

public virtual void *f() {. . .}*

Átdefiniálásukkor – az Object Pascalhoz hasonlóan – az **override** kulcsszó szükséges:

public override void *f() {. . .}*

A túlterheléshez nem szükséges külön kulcsszó, a paraméterek megfelelése alapján keresi meg az adott híváshoz „legközelebb" álló metódust (bővebben lásd a nyelv referencia leírásában [C# Ref 03]).

A leszármazott **new** módosító segítségével elrejtheti az őstől örökölt metódust – ez felel meg az Object Pascal-beli újrabevezetés (*reintroduce*) lehetőségének. Így például a **new public virtual void** *f()* elfedi az őstől örökölt virtuális *f()* metódust.

Polimorfizmus és dinamikus kötés az Eiffelben

Az Eiffel teljesen objektumorientált szemléletben tervezett nyelv, így a változók alapértelmezés szerint referenciák, és az aktuális műveletek hívása csak dinamikusan történhet. A leszármazott az öröklődési záradékban az ős megadásakor (a *redefine* záradékban) kell jelezze, ha valamelyik műveletet át akarja definiálni.

```
class SOKSZOG inherit
    ALAKZAT
        redefine F end
...
```

A rutinok elő- és utófeltételeinek átdefiniálása a leszármazottban csak „biztonságos" módon megengedett, azaz az előfeltételt csak gyengíteni, az utófeltételt csak szűkíteni lehet (lásd a 13. fejezetet). A típusinvariáns is csak szűkíthető, ezzel is kifejezve, hogy a leszármazott típusértékhalmaza része az ősének [Mey 91]. Érdekes módon, ugyanakkor a nyelv támogatja, hogy a leszármazott átdefiniáláskor a műveletek paramétereit kovariáns módon változtassa, s ezzel váratlan, futási idejű hibák is keletkezhetnek. Például, tegyük fel, hogy a *SOKSZOG* osztályban az *F* művelet törzsében hívja az átdefiniált paraméter egy, csak ebben a leszármazott osztályban definiált műveletét, *G*-t, míg az *ALAKZAT* osztály másik leszármazottjában, a *KOR* osztályban a *G* nem szerepel.

```
class ALAKZAT
 feature
  F(A:ALAKZAT) is do
    io.putstring ("F az ALAKZATBAN");
  end
end
```

```
class SOKSZOG inherit
    ALAKZAT
        redefine F end
 feature
   F(A:SOKSZOG) is do
     io.putstring ("F a SOKSZOGBEN");
     A.G;          -- Hívja a paraméter új G műveletét.
    end
  G is do
     io.putstring("G uj SOKSZOGBEN")
   end
 end

class KOR inherit
    ALAKZAT  -- Nem definiálja át F-t.
 ...
 end
```

Ekkor, ha *ALAKZAT* típusú referenciához kapcsoljuk polimorfizmussal a *SOKSZOG* típusú objektumot, és nem a dinamikus típusnak megfelelő paraméterrel hívjuk az *F* műveletet, akkor a fordító ezt nem észleli, s csak futás közben derül ki, hogy az aktuális paraméter nem jó.

```
o1, o2: ALAKZAT;
make is
 do
    !SOKSZOG! o1   -- Hozzárendeli o1-hez a SOKSZOG objektumot,
    !KOR! o2       -- o2-höz a KOR objektumot.
     o1.F(o2);     -- Futás közbeni hiba.
```

Az ősosztály metódusainak meghívása

A megvalósítás során szükségünk lehet arra, hogy meghívjuk az ősosztály vagy ősosztályok metódusait, vagy akár arra is, hogy a gyermekosztályban egy örökölt metódust úgy bővítsünk ki, hogy felhasználjuk a már meglévő, az ősosztályban megírt metódust.

Az egyszeres öröklődést támogató nyelvekben az ősosztály eléréséhez általában valamilyen kulcsszót használhatunk.

Például az Object Pascalban az **inherited** kulcsszóval hivatkozhatunk egy metóduson, vagy akár egy konstruktoron belül is az ősosztály ugyanilyen nevű metódusára, miközben előtte, vagy utána tetszőleges tevékenységeket elvégezhetünk [Can 00]:

```
type TSzulo = class ...
  procedure f(...); virtual; ...
end;
{ A program a következő oldalon folytatódik. }
```

```
{ Folytatás az előző oldalról. }
type TGyerek = class(TSzulo) ....
  procedure f(...); override; ...
end;

procedure TGyerek.f(...);
begin
  inherited f(...);  { Itt a TSzulo f műveletét hívjuk. }
  ...
end;
```

A Javában (és a Smalltalkban) ugyanezt a célt szolgálja a **super** kulcsszó, ahogy a 432. oldal példájában is láttuk, ha a *Fonok* osztályban felüldefiniáljuk az *Alkalmazott*-tól örökölt *potlek* metódust:

```
public int potlek() {
    return super.potlek() + beosztottakSzama * 1000;
}
```

A C#-ban a **base** kulcsszóval minősítve kaphatjuk meg az ősosztály metódusát [C# Ref 03]:

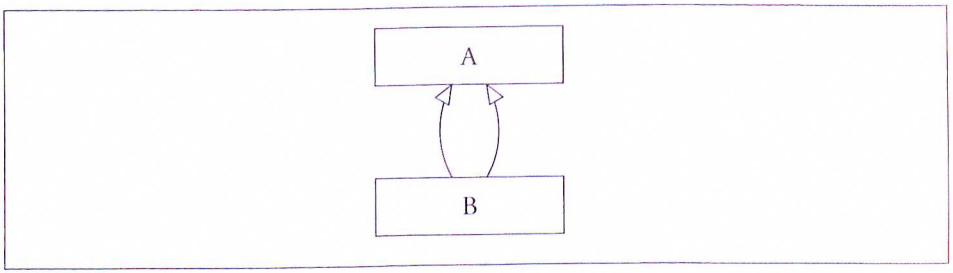
```
public class A {
      ...
      public virtual void Kiir() { ... }
}
class B: A {
      ...
      public override void Kiir() {
         // Hívja a szülő Kiir metódusát:
         base.Kiir();
         ....
      }
}
```

Vannak nyelvek, ahol a leszármazottban az ősosztály nevével minősíthetjük annak elérhető metódusait.

A C++-ban például a :: hatáskör operátorral hivatkozhatunk az ős megfelelő műveletére: az *Ősosztály::Metódus(paraméterlista)*; alakú üzenetküldéssel – ezt használhatjuk majd például a többszörös öröklődés konfliktusainak feloldására (lásd a 12.7.5. pontot).

Az Eiffelben fontos tervezési elv volt, hogy a leszármazott osztály szövegében az öröklődési záradékon kívül sehol ne szerepeljen hivatkozás a szülőre. A nyelv elsősorban ezért támogatja a többszörös öröklődés egy különleges formáját, az *ismételt öröklődést* is.

Ha az átdefiniálás során szeretnénk felhasználni az *f* jellemző szülőtől örökölt formáját, akkor kétszer kell származtatnunk az őstől: az egyik ágon átnevezzük az adott jellemzőt,



12.10. ábra: Ismételt öröklődés

mondjuk *old_f*-re, hogy tudjunk rá az osztály törzsében hivatkozni, a másik ágon pedig felüldefiniáljuk, felhasználva az átnevezett régi verziót[8]:

```
class D
  inherit D
    rename f as old_f
  end;
  inherit D
    redefine f
    select f
  end
feature
  f is
    do
      ...
      old_f;
      ...
    end
end
```

A metódusok meghívása

A leszármazottak bevezethetnek új, az ősökben esetleg még nem értelmes metódusokat is, így bővíthetik a lehetséges műveletek halmazát. Például az *Állat* osztály *Kutya* leszármazottjában bevezethetjük az *ugat* műveletet. A polimorfizmus és a dinamikus összekapcsolás lehetősége felveti azt a kérdést is, hogy miképpen történjen ezen új metódusok hívása. Tegyük fel, hogy az *Állat* osztályban bevezettük az *eszik* metódust is. Ha létrehozzuk az *Állat* osztály egy *kedvenc* nevű változóját, és a polimorfizmust felhasználva értékül adjuk neki az *Állat* osztály egy leszármazottjának, akár a *Kutya* osztálynak, akár a *Macska* osztálynak stb. egy objektumát, akkor erre nyilván megengedett lesz a *kedvenc.eszik* hívás – az objektumorientált elveknek megfelelően leginkább dinamikus összekapcsolással. De vajon

[8] A select utasítás jelentését lásd a 12.7.5. pontban.

elfogadható-e, hogy ha tudjuk, (vagy tudni véljük), hogy a *kedvenc* dinamikus típusa *Kutya*, akkor lehetséges legyen a *kedvenc.ugat* hívás is?

A programozási nyelvek nagy része – például a C++, Java, Object Pascal, Eiffel stb. – a hívható műveleteket egyértelműen a változó *statikus típusához* köti, így fordítási hibát ad a fenti esetben. Ez a döntés biztonságosabbá teszi a programot, hiszen ha megengedjük, hogy a metódusok összekapcsolása teljesen dinamikusan, futási időben történjen – ahogy például a Smalltalkban vagy a Dylanban stb. –, akkor, ha a *kedvenc* változó aktuális dinamikus típusa valamilyen hiba következtében mondjuk mégsem a *Kutya*, hanem a *Macska* osztály egy objektuma, amelyiknek nyilván nincs *ugat* metódusa, ez futási hibát jelent.

12.7.3. Absztrakt osztályok

A programtervezés során, amikor az osztályhierarchiákat létrehozzuk, a legfontosabb, hogy kialakítsuk a megfelelő típusspecifikációkat. Ez a szint még független a megvalósítástól. A típusértékhalmaz tényleges reprezentációja és a műveletek implementációja egy következő lépés. A polimorfizmus és a dinamikus kötés lehetősége eszközt ad a kezünkbe, hogy a tervezés során olyan általános, absztrakt osztályokat is létrehozzunk, és egységes módon kezeljünk, amelyeknél még nem döntünk a konkrét megvalósításról, hanem majd a leszármazott választhatja ki a számára legmegfelelőbb implementációt.

Az absztrakt osztályok tartalmazhatnak absztrakt metódusokat, vagyis olyan metódusokat, amelyeket csak specifikáltunk, és még nem valósítottunk meg, de lehet implementált műveletük is. Fordítva is igaz, ha egy osztálynak van legalább egy absztrakt metódusa, akkor azt az osztályt absztrakt osztálynak nevezzük. Egy absztrakt osztálynak általában nem lehet konkrét objektuma. Változói a polimorfizmus segítségével a különböző leszármazottainak referenciáit tartalmazhatják. Például egy általános *Alakzat* osztálynál még nem tudjuk megadni a *terulet* metódus implementációját, csak leszármazottainál, például a *Teglalap* vagy *Kor* osztályokban, de célszerű a tervezésnél már az *Alakzat* osztályban jelezni, hogy szeretnénk majd területet számoló tényleges műveletet is. Ekkor akár *Alakzat* típusú elemek listáját is kezelhetjük majd egységes módon.

Az objektumorientált programozási nyelvek többsége ad eszközöket absztrakt osztályok létrehozására. Vannak nyelvek, amelyekben osztály absztrakt metódus nélkül is lehet absztrakt – ilyen például a Java, a C#, vagy az Eiffel, és vannak, ahol az absztrakt metódus szükséges ehhez – ilyen például a C++. Azokban a nyelvekben, ahol van lehetőség a további származtatás megtiltására – például a Javában a **final**, a C#-ban a **sealed**, az Eiffelben a **frozen** kulcsszóval, ezek absztrakt osztályok esetén természetesen nem használhatók, hiszen ez megtiltaná, hogy leszármazottat hozzunk létre, és így nem lehetne az implementációt megadni.

Felépíthetjük absztrakt osztályok hierarchiáját, a leszármazottak megvalósíthatják az örökölt absztrakt műveleteket, de mindaddig biztosan absztraktak maradnak, míg marad legalább egy absztrakt, meg nem valósított műveletük.

Absztrakt osztályok a programozási nyelvekben

SIMULA 67 – Smalltalk • Érdemes megemlíteni, hogy noha már a SIMULA 67-ben, az első objektumorientált programozást támogató nyelvben leírták, hogy a virtuális műveleteknek nem kötelező, hogy implementációja is legyen, vagyis lehet absztrakt művelet is [Dah 85], a teljesen objektumorientált szemléletben tervezett Smalltalkból kimaradt ez a lehetőség.

C++ • A C++-ban az absztrakt osztályokat nem jelöli külön kulcsszó, ha van absztrakt műveletük – ezt a törzs helyére írt = *0* jelzi –, akkor az osztály is automatikusan absztrakt lesz, példánya nem hozható létre. Például az előbb említett általános *Alakzat* osztályban a tervezés szempontjából jobb, ha a *terulet* függvényt a

```
virtual int terulet() = 0;
```

specifikációval absztrakt műveletként vezetjük be, mintha az alábbihoz hasonló semmitmondó definíciót adnánk meg rá:

```
virtual int terulet() { return (0); };
```

Object Pascal • Az Object Pascalban az absztrakt műveletet a **virtual** és az **abstract** kulcsszóval kell jelölni. Például:

```
function f: string; virtual; abstract;
```

A megvalósítást ugyanúgy jelezni kell a leszármazottban az **override** kulcsszóval, mint a többi virtuális művelet felüldefiniálását. Az osztály az absztrakt műveletektől nem válik absztrakttá, létrehozhatjuk példányait, s futásidejű kivételt vált ki, ha még nem implementált műveletére hivatkozunk [Can 00].

Java • A Javában is az **abstract** módosítót kell használni az absztrakt osztályok és műveletek létrehozásakor. Itt az előbbi általános szabályok érvényesek [Nyé 01]. Például definiálhatjuk az absztrakt *Alakzat* osztályt és bevezethetjük benne az absztrakt területszámító *terulet* függvényt a specifikációjával.

A tényleges leszármazottban egyszerűen megadjuk az implementációt:

```
public abstract class Alakzat { . . .
   public abstract double terulet();
   . . .
}
public class Teglalap extends Alakzat { . . .
   public double terulet() { . . . }
   . . .
}
```

C# • A C#-ban is az **abstract** módosítót kell használni az absztrakt osztályok és műveletek létrehozásakor. A megvalósítást – a Javától eltérően, az Object Pascalhoz hasonlóan – itt is jelezni kell az **override** kulcsszóval [C# Ref 03].

```
abstract class Alakzat {. . .
   public abstract int terulet(); . . .
}
abstract class Konvex: Alakzat {. . .
   public void f() {. . .}
}
class Teglalap: Konvex {. . .
   public override int terulet() {
     . . .
   }. . .
}
```

Az *Alakzat* absztrakt osztályban bevezetjük a *terulet* absztrakt metódust. A *Konvex* osztály bevezet egy új metódust, a *f*-t, de mivel nem ad implementációt a *terulet*-re, ez is absztrakt marad. A *Teglalap* osztály adja a *terulet* megvalósítását. Mivel ebben az osztályban már nincs absztrakt metódus, így itt az osztály megadásánál az **abstract** kulcsszó nem kötelező, bár lehetséges.

Eiffel • Az Eiffelben az absztrakt osztályokat és metódusokat a **deferred** (késleltetett) kulcsszó jelzi az osztály bevezető sorában, illetve a művelet törzsének helyén.

```
deferred class ALAKZAT ...
feature ...
  kiir is
    deferred
  end ....
end
```

Az absztrakt művelet megvalósítása a leszármazottban nem jelent átdefiniálást, így ezt nem is szükséges feltüntetni az öröklődési záradékban.

```
class KOR inherit
    ALAKZAT
feature ...
    kiir is
        do
            io.putstring ("KOR VAGYOK");
        end; ...
end
```

A nyelvben lehetőség van arra is, hogy egy már megvalósított műveletet absztrakttá tegyünk: ezt az öröklődési záradékban, az **undefine** részben jelezhetjük. Ekkor gyakorlatilag annyi történik, hogy egy effektív (implementációval rendelkező) jellemzőt absztrakttá

(**deferred**) teszünk. Ez a művelet megőrzi a rutin teljes szignatúráját, a hozzákapcsolt elő- és utófeltételekkel együtt, csupán a törzset veszi le róla.

Az **undefine** záradékra a többszörös öröklődés egy különleges lehetőségénél, az összekapcsolásnál (join) lesz majd szükség (lásd a 12.7.5. pontban), ez az oka annak, hogy a nyelv szabványos könyvtárában gyakran találkozhatunk vele, például:

```
deferred class CHAIN [G]
  inherit
    CURSOR_STRUCTURE [G]
        undefine
            prune_all ....
```

Az, hogy Eiffelben az absztrakt műveleteknek is adhatunk elő- és utófeltételeket, s a késleltetett, absztrakt osztályoknak is lehet típusinvariánsa, komolyan támogatja az áttekinthető, újrafelhasználható osztályhierarchiák kialakítását. Például a *LIST* osztályban a legtöbb művelet még absztrakt, a majdani leszármazottak adják a konkrét megvalósítást, de az elő- és utófeltételek, valamint a típusinvariáns segítenek az osztály szemantikájának megfogalmazásában:

```
deferred class LIST[G]...
feature ....
  forth is
    -- A kurzort egy pozícióval előbbre viszi.
    require
      not after
  deferred
    ensure
      index = old index + 1
  end
...
invariant
non_negative_count:count >= 0
offleft_by_at_most_one:index >= 0
offright_by_at_most_one: index <= count + 1
....
end
```

12.7.4. A közös ős

Bizonyos programozási nyelvek – ilyenek például a Smalltalk, a Java, a C#, vagy az Eiffel – abból az elvből indulnak ki, hogy minden egyes osztály egyetlen, absztrakt közös ősből származik. Ebben az osztályban lehet ekkor összpontosítani az összes osztályra jellemző műveleteket, például a bitenkénti összehasonlítás, másolás stb. lehetőségeit. Más nyelvek – például a SIMULA 67, a C++ vagy az Ada 95 –, nem vezetnek be közös ősosztályt, és így lehetővé teszik új, önálló hierarchia-fák kialakítását.

A közös ős egy másik előnye a már említett helyettesíthetőségből származik. Azt mondtuk, hogy minden származtatott osztály előfordulhat ott, ahol az ősosztály szerepelt, vagyis a származtatott osztályok helyettesíthetik az ősöket. Ez előnyünkre válhat a metódusok paraméterezésénél. Nyugodtan deklarálhatunk például metódusokat olyan formális paraméterekkel, amelynek típusa az ősosztály, és amikor szükség van a metódus tényleges meghívására, a paraméterek lehetnek valamelyik leszármazott osztály példányai. A közös ős típusú formális paraméter minden aktuális paraméter típusát felveheti.

Azokban a programozási nyelvekben, amelyek bevezetik a közös ősosztályt – ezt általában *Object*-nek vagy *TObject*-nek hívják –, ha egy új osztály bevezetésekor nem adunk meg ősosztályt, automatikusan ez az osztály lesz a szülőosztály.

Érdekességként megemlítjük, hogy az Object Pascal nyelvben az előző lehetőségek egyvelegét figyelhetjük meg: itt ugyanis az **object** és a **class** kulcsszó is jelölhet osztályt és ezt a nyelv kétféleképpen kezeli. Az **object** osztályt a Turbo Pascal 5.5-nél vezették be és így került be az Object Pascalba (és a Delphibe) is. Az **object** osztályoknak nincs előre definiált közös ősük, míg az Object Pascalban bevezetett **class** osztályoknak minden esetben van közös ősük, a *TObject* osztály.

12.7.5. Többszörös öröklődés

Többszörös öröklődésről beszélünk, ha egy osztálynak egynél több közvetlen őse van. Ez a legegyszerűbb megközelítésben jelentheti azt, hogy az adott osztály két (vagy több) osztályozásban tekinthető alosztálynak. Bertrand Meyer objektumorientált programtervezésről írt könyvében [Mey 00] számos lehetséges példát sorol fel, ahol egy osztály természetes módon alosztálya több szülőosztálynak is. Például a vállalati repülőgépek osztálya (*VALLALATI_REPULOGEP*) egyrészt örökli a repülőgép osztály (*REPULOGEP*) tulajdonságait, s a pilóta ennek megfelelően használja majd, másrészt, a vállalat könyvelőjének szemében ez is egy eleme a vagyontárgyak osztályának (*VAGYONTARGY*), s ennek megfelelően kell a könyvelésben kezelni. Hasonló megfontolások alapján az Eiffel szabványos könyvtárainak tervezésénél különböző absztrakciókat vezettek be, például a numerikus tulajdonságú objektumok – az absztrakt *NUMERIC* osztály leszármazottai – rendelkeznek a szokásos +, −, *, / stb. műveletekkel, az összehasonlítható objektumok – az absztrakt *COMPARABLE* osztály leszármazottai – <, > stb. műveletekkel, s ezekből származnak, többszörös öröklődést használva az *INTEGER*, *REAL* stb. osztályok.

A többszörös öröklődés egy másik fontos lehetősége, amikor a specifikációs és az implementációs öröklődés kombinálására használjuk. Az egyik ős adja a funkcionalitást, a másik – lehetőleg rejtetten – a reprezentációt. Például a konkrét, fix méretű vermek osztálya (*FIXED_STACK*) specifikációjában leszármazottja az absztrakt verem osztálynak (*STACK*), s implementációját a konkrét tömb osztálytól (*ARRAY*) örökli. Az öröklődés ezen formáját nevezi Bertrand Meyer „érdekházasságnak" [Mey 00]. Szellemes hasonlata szerint az öröklődés ezen formája hasonlít egy olyan házassághoz, ahol a szegény arisztokrata (az absztrakt osztály leszármazottja) a vagyonos család (a megvalósítást adó osztály) gyermekével házasodik, s így mindkettő eléri célját: az implementáció segíti a funkcionalitást.

A C++-ban a többszörös öröklődést többféle módon lehet megvalósítani:

- A többszörös specifikációs örökléshez lehet több nyilvános őse egy osztálynak, például:

```
class A { //...
  public:
    virtual void f() { cout << "f az A-ban\n"; };
};
class B { //...
  public:
    virtual void g() = 0;
};
class C : public A, public B { //...
  public:
    // Felülírja az A::f() függvényt.
    void f() { cout << "f a C-ben\n"; };
    // Megvalósítja a B::g() függvényt.
    void g() { cout << "g a C-ben\n"; };
};
```

A leszármazott örökli ősei adattagjait és metódusait, átdefiniálhatja, megvalósíthatja, vagy változatlanul hagyhatja őket. A polimorfizmus és dinamikus kötés segítségével az ősök referenciáin keresztül is elérhetők az öröklött metódusok, például:

```
int main()
{
  A* a1;
  B* b1;
  C c;
  a1=&c;
  b1=&c;
  // Hívja a dinamikus típusnak megfelelőt:
  a1->f();
  b1->g();
  return 0;
}
```

Az ősökön keresztül természetesen csak a tőlük öröklött metódusok érhetők el, nem írhatjuk például, hogy: `a1->g();`, hiszen az *A* szülőnek nincs *g* metódusa.

- A specifikációs és implementációs öröklés együttes megvalósításához azt az osztályt, ahonnan a specifikációs öröklést szeretnénk, nyilvános (**public**) ősosztálynak, az implementációs öröklés ősosztályát pedig rejtett (**private**), vagy védett (**protected**) ősként adjuk meg. Érdemes megfigyelni, hogy az alapértelmezés a rejtett öröklő-

dés. Például a *C* osztály az *A*-nak nyilvános, a *B*-nek rejtett, csak implementációs leszármazottja lesz:

```
class C : public A, B { //...
  public:
    void f() { cout << "f a C-ben\n"; };
    void g() { cout << "g a C-ben\n"; };
};
```

Ekkor a polimorfizmus és dinamikus összekapcsolás csak az *A*-beli referenciákon keresztül működik, s bár a *C*-ben felüldefiniálhatjuk és nyilvánosan elérhetővé tehetjük a *B*-től öröklött *g* függvényt is, de *B*-beli referencián keresztül nincs polimorfizmus, a *C* nem altípusa a *B*-nek.

- Lehetséges a többszörös implementációs öröklés is. Ekkor a leszármazott felüldefiniálással, vagy explicit megadással nyilvánossá teheti, amit az ősöktől elérhetően örökölt (őseinek védett és nyilvános jellemzőit), a többi örökölt adattag és metódus a külső felhasználó számára nem lesz elérhető. Hasonlóképpen nem lehetséges az ősök referenciáin keresztül a polimorfizmus.

```
class C : A, B { //...
  public:
    // Felülírja az A::f() függvényt.
    void f(){cout<<"\n f az C-ben \n";};
    // Elérhetővé teszi a B-től öröklött g-t.
    B::g;
};
```

A *C*-beli *c*-re a *c.f()* hívással a felüldefiniált metódust, a *c.g();* hívással a *B*-beli *g* metódust érjük el.

Az Eiffelben az alapértelmezés a specifikációs öröklődés, de – ahogy a 12.7.1. részben láthattuk –, a szelektív láthatóság lehetősége az implementációs öröklődést is támogatja, ekkor minden öröklött jellemzőt elrejthetünk az osztályt használók elől.

A névütközés kezelése

A többszörös öröklődés általános elvi lehetősége számos kérdést is felvet. A lényegében különböző absztrakciókat leíró ősökben is előfordulhat, hogy van azonos nevű és szignatúrájú metódusuk – melyiket, hogyan örökölje a leszármazott? Az átnevezés jó eszköz lehet, ha azt szeretnénk, hogy ezek különbözőek maradjanak, de elképzelhető az is, hogy „összekapcsolással" olyan törzset adjunk meg a leszármazottban, amelyik mindkét ősnek megfelel.

Átnevezés

Az Eiffelben az öröklődési záradékban a leszármazott egyszerűen átnevezheti az ősöktől örökölt jellemzőket. Ez nem változtatja meg az örökölt jellemzőt, csak az adott osztályban bevezet egy új nevet a számára. Ezt a lehetőséget használhatjuk a névütközések feloldására is. Például:

```
class A
 feature
  f ....
end;

class B
 feature
  f ....
end;

class C
 inherit
   A
    rename f as af
    end;
   B;
 feature ....
end;
```

A *C* osztály örököl az *A*-tól és a *B*-től, s mindkét ősének van egy *f* jellemzője. Bármelyiket átnevezheti – például az *A*-tól öröklöttet *af*-re –, ezáltal megkülönböztethetővé válnak. A fenti példában így az *f* hivatkozás a *C* osztály objektumainál a *B*-től öröklött *f*-t jelenti majd.

Az Eiffelben nem megengedett a jellemzők neveinek túlterhelése sem, tehát különböző szignatúrájú, de azonos nevű jellemzők öröklése esetén is szükséges az átnevezés.

Az átnevezés ugyanakkor, ahogy Bertrand Meyer írja, jó lehetőséget ad arra is, hogy a leszármazott saját elgondolásainak megfelelő névkonvenciót vezessen be [Mey 00]. A *WINDOW* osztályt definiálhatjuk például a *TREE* és a *RECTANGLE* leszármazottjaként, de a *WINDOW* osztály alkalmas módon átnevezheti az örökölt jellemzőket.

A C++ nem ad lehetőséget átnevezésre.

Összekapcsolás

Többszörös öröklődés használatakor előfordulhat az az eset is, hogy több, azonos néven és szignatúrával örökölt jellemzőt szeretnénk a leszármazottban összevonni (join) és közösen megvalósítani.

A C++-ban, ha a leszármazott például több ágon is örököl egy **virtual void** *f()* metódust, és felüldefiniálja, akkor a polimorfizmus és dinamikus összekapcsolás révén bármelyik ősének referenciáján keresztül ezt éri el:

```
class A{
  public:
     virtual void f() { cout << "f az A-ban\n"; };
};

class B{
  public:
     virtual void f() { cout << "f a B-ben\n"; };
};

class C: public A, public B {
  public:
      virtual void f(){ cout << "f a C-ben\n"; };
};
```

A *C* osztályban átdefiniált *f* metódus helyettesíti mindkét ősétől örökölt *f*-t, nincs névütközés sem az ősök referenciáin keresztül – például *C c; B* b1 =&c;* esetén *b1->f();*, sem közvetlenül, egy *c.f();* hívásnál.

Más eredményre jutunk ugyanakkor, ha esetleg a *C*-ben szeretnénk kiválasztani valamelyiket az ősöktől örökölt implementáció közül:

```
class C: public A, public B {
  public:
     B::f();
};
```

Ez a döntés valójában ellentmond annak, hogy a többszörös nyilvános öröklődéssel a *C* osztályt az *A és* a *B* osztályok altípusaként hozzuk létre. Így nem lesz megbízható az eredmény, ugyanis a *B*-beli *f* kiválasztása érvényesül a *b1->f();* és a *c.f();* hívásban, de az *A* a1 =&c; a1->f();*-ben természetesen nem, ott csak az *A*-beli implementációt hívja/hívhatja.

Lehetséges az az eset is [Str 00], hogy a leszármazott az ősöktől ugyanazon a néven, de különböző szignatúrával örököl metódusokat, ekkor a **using** deklaráció segíthet a megfelelő metódus kiválasztásában:

```
class A{
  public:
     virtual int f(int) {...};
     ...
};
```

```
class B{
  public:
    virtual double f(double) {...};
};

class C: public A, public B {
  public:
    using A::f;
    using B::f;
    virtual void f(){...};
    ...
};
```

Ekkor a *C* osztályban az *f* túlterhelt változatai lesznek elérhetők.

Az Eiffelben lehetséges azonos nevű jellemzők összekapcsolása, de csak akkor, ha azonos a szignatúrájuk és absztraktak. Ehhez lehet szükséges az örökődési záradék **undefine** része (lásd a 12.7.3. részt), így újra absztrakttá tehetünk egy már megvalósított jellemzőt is. A szignatúrák megegyezését esetleges átdefiniálással is elérhetjük, sőt az azonos névhez is eljuthatunk átnevezéssel. Az így létrejövő metódus előfeltételében az összekapcsolt jellemzők előfeltételei *vagy* kapcsolatban, utófeltételében pedig az utófeltételek *és* kapcsolatban egyesülnek. A leszármazott a jellemzőt megtarthatja absztraktnak, vagy adhat hozzá implementációt. „Több absztrakt madarat is megölhetünk egy konkrét kővel" – mondja Bertrand Meyer [Mey 00].

```
class D
  inherit A
    rename g as f
    undefine f
  end;
  inherit B
    rename h as f
    undefine f
  end;
  feature
    f : T is
    ...
end -- class D
```

Az összekapcsolásnak már a legfelső absztrakciós szinten, a már eleve **deferred** rutinok esetében is fontos szerepe lehet a különböző szempontú absztrakciók összevonásában. Ebben az esetben természetesen nincs szükség **undefine** részekre. Például a nyelv szabványos könyvtárában a *CHAIN* osztály, ami a szekvenciális struktúrákat listáit írja le, két absztrakt

osztálytól örököl, ahol mindkettőnek van egy *item* jellemzője, ami a kurzor pozíciónál álló elemet adja vissza:

```
deferred class CHAIN[T] inherit
    BIDIRECTIONAL[T] ...
    -- Van egy item deferred rutinja a bejárási hierarchiából, ez mindig a
    -- bejárás során az aktuális elem értéke.
    ACTIVE[T] ...
    -- Van egy item deferred rutinja az elérési hierarchiából, ez a
    -- kurzorpozíciónál lévő elem.
...
end
```

A *CHAIN* osztály örökli mindkét *item* jellemzőt. Normálisan ezt a névütközést átnevezéssel kellene feloldani, de most voltaképp az a kívánatos, hogy ezt a kettőt összekapcsolja.

A „rombusz-öröklődés" kezelése

A többszörös öröklődés egy különleges lehetősége az úgynevezett ismételt öröklődés, amikor egy leszármazott valamelyik ősosztályát az öröklődési gráfban többféle úton is eléri, többször örökli ugyanazokat az attribútumokat, illetve metódusokat. Kérdés, ténylegesen hány példányban kapja meg, és hogyan éri el ezeket? Ez az úgynevezett „rombusz-öröklődés" problémája.

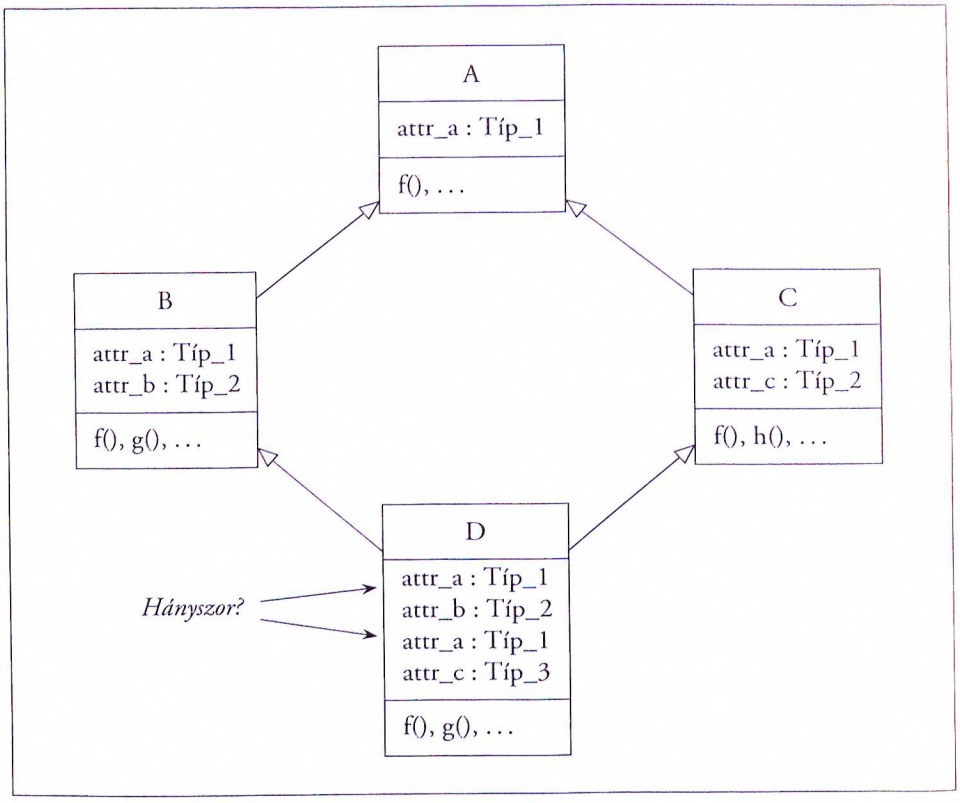
Ha ugyanis egy *A* ősosztályban létezik legalább egy adat (*attr_a*) vagy egy metódus (*f*), az *A* osztály *minden leszármazottja* örökli ezeket. Tegyük fel, hogy az *A* osztálynak van két leszármazottja, *B* és *C*, ekkor ezeknek is lesz *attr_a* attribútuma és *f* metódusa, valamint bevezethetnek új attribútumokat (*attr_b*, *attr_c*), illetve metódusokat (*g*, *h*), sőt át is definiálhatják az örökölt *f* metódust. Ha ezután létrehozunk még egy új leszármazottat, *D*-t, amely többszörös öröklődést használva a *B*-ből és *C*-ből öröklődik, akkor hány példányban jelenjen meg a *D*-ben az örökölt adat vagy metódus? Ha kétértelműség van, hogyan válasszunk?

A programozási nyelvek tervezői a fenti kérdésekre különböző válaszokat adtak. A probléma elemzésénél vizsgáljuk meg külön-külön a többször örökölt adattagok és a metódusok esetét a két jellegzetes, többszörös öröklődést támogató nyelvben, a C++-ban és az Eiffelben.

Adattagok többszörös öröklése

A C++-ban az az alapértelmezés, hogy a leszármazottban minden adattag annyiszor jelenik meg, ahányszor a közös őstől örököltük, s az osztálynévvel minősítve (például: *C::attr_a*) különböztethetjük meg az azonos nevű adattagokat.

Az ismételt öröklődésnél lehetnek olyan helyzetek is, amikor nincs értelme a közös őstől örökölt tagokat többszörözni. Például, ha a *Szemely* osztályt jellemezzük a *Neve*, *Szuletesi_hely*, stb. adatokkal, s ebből származtatjuk a *Tanar* és a *Diak* osztályokat, majd ezekből többszörös öröklődéssel a *H_Gy* (Hallgatói_Gyakorlatvezető) osztályt, aki *Tanar* és *Diak* egyszerre, akkor nincs értelme, hogy az előbbi adatokat egynél többször tároljuk.



12.11. ábra: A többszörös öröklődés nehézségei

A C++-ban ilyen esetekben a szülő osztályok (most a *Tanar* és a *Diak*) dönthetnek a „virtuális ős" lehetősége mellett, ekkor a *Szemely* osztálytól örökölt adattagok a közös leszármazottakban csak egyszer jelennek majd meg [Str 00].

Lehetséges természetesen az is, hogy a közös ősben előfordulnak egyszer és többször tárolandó adattagok is – például lehet, hogy az ösztöndíjat és az óradíjat más-más banki számlaszámra küldik. A C++-ban nincs mód ilyenfajta vegyes kezelésre.

Az Eiffelben – szemben a C++-szal – az alapértelmezés szerint a közös őstől öröklött jellemzők csak egyszer jelennek meg, s ha a leszármazott – jelen esetben a *H_Gy* osztály – átnevezi valamelyiket, ezzel dönt a többszörözésükről. Ez a megoldás lehetővé teszi az örökös számára, hogy az adattagokról a tényleges felhasználás szintjén, egyedileg döntsön [Mey 91].

Metódusok többszörös öröklése

A rombusz-öröklődés során a közös őstől változatlanul örökölt virtuális metódusokat a C++-ban is és az Eiffelben is a közös leszármazott egyszer örökli.

Más a helyzet, ha az öröklődés során valamelyik osztály átdefiniálja a közös őstől örökölt metódust. Ekkor a közös gyerek ebből a metódusból már többet örököl, s ezt C++-ban csak saját átdefiniálással, Eiffelben átdefiniálással vagy átnevezéssel oldhatja fel.

Külön kérdés ekkor a polimorfizmus és a dinamikus kötés megvalósítása. Legyen *f* az *A* osztálytól örökölt metódus (lásd a 12.11. ábrát). Tegyük fel, hogy a *B* is és a *C* is átdefiniálja, és *D* nem vezet be új megvalósítást rá. Ekkor, ha a programunkban létrehozunk egy *D* típusú objektumot, amit értékül adunk egy *A* típusú *a1* referenciának (ez polimorfizmussal elvben megvalósítható), majd hívjuk az *f* műveletet – *a1.f* – akkor a dinamikus kötéssel az *f* melyik változata legyen érvényes? A *B*-ben vagy a *C*-ben bevezetett? Nem az a kérdés itt, hogy nincs implementáció, hanem az, hogy túl sok van. Az Eiffelben ennek a kérdésnek a megoldására vezették be az öröklődési záradék *select* részét.

```
class A
  feature
    f : T is
    ...
 end -- class A
 class B
   inherit A redefine f end
  feature
    f : T is
    ...
end -- class B
class C
  inherit A redefine f end
  feature
    f : T is
    ...
end -- class C
class D
  inherit B rename f as bf select bf end;
  inherit C rename f as cf end
end -- class D
```

A hasonló többszörös öröklődésből felmerülő esetekben a **select** záradékban kell jelezni azt, hogy a leszármazott (esetünkben a *D* osztály) melyik öröklött jellemzőt választja ki arra az esetre, ha egy ősén keresztül hívódna a kérdéses rutin.

A többszörös öröklődés problémáira hivatkozva számos programozási nyelv nem is engedi meg a többszörös öröklődést, és vannak, akik azt tanácsolják, hogy *inkább teljesen kerüljük is el.* Mások úgy vélik, hogy a gondok nagy része megoldható, és a többszörös öröklődés előnyei kihasználhatók, ha a specifikációs öröklésnél alapvetően absztrakt osztályoktól öröklünk, s a reprezentációhoz szükséges adattagokat és a metódusimplementációkat legfeljebb egy őstől kapjuk. Ez a megközelítés vezetett az interfészek nyelvi elemként való bevezetéséhez.

12.7.6. Interfészek

Az interfész (*interface*) egyfajta különleges absztrakt osztályt jelent, amelyből hiányoznak a példányváltozók, csak metódus-deklarációkat (és esetleg osztályszintű állandókat) tartalmaz. Az interfész - mint nevéből is látszik - egy felületet biztosít, egy olyan felületet, amely a programban egy absztrakciós szint bevezetésének lehetőségét rejti: a feladat megvalósításának egy bizonyos szintjén el lehet tekinteni a konkrét megvalósítástól. Ez nagymértékben megkönnyíti a tervezést és növeli a módosíthatóságot.

Egy interfész tényleges használata a megvalósításán keresztül történik. Egy osztály akkor valósít meg egy interfészt, ha az összes általa deklarált metódushoz ad törzset. Ezáltal az absztrakt program konkréttá válik. Mindenütt, ahol az adott interfész szerepelt, a polimorfizmusnak megfelelően szerepelhet bármilyen, az interfészt implementáló osztály.

Az interfészek között is létezhet öröklődés, tehát az interfészeket is öröklődési hierarchiába lehet szervezni, sőt a többszörös öröklődésnek (lásd a 12.7.5. pontot) sincs semmilyen akadálya, hiszen ha hiányoznak a példányváltozók és a metódustörzsek, minden egyértelművé válik. Egy osztály tetszőleges számú interfészt implementálhat. Ha adott egy feladat, amely például két jól elkülöníthető részfeladatra bontható, és ezt egy osztálynak kell megvalósítania, akkor ez megoldható úgy, hogy a két részfeladathoz tartozó metódusok absztrakt módon bekerülnek két interfészbe, és az osztály mindkettőt implementálja.

Protokollok az Objective-C-ben

Az Objective-C nyelv a C Smalltalk-alapú objektumorientált kiterjesztése [Pin 91]. A nyelvben az egyszeres öröklődés támogatása mellett bevezetik a protokollok (**protocol**) fogalmát.

A protokoll absztrakt műveletek gyűjteménye. Egy protokollba foglalhatjuk valamilyen tulajdonság jellemzőit, amelyek több osztályban is megvalósíthatók. Ha egy osztály megvalósítja ezeket a műveleteket, akkor azt mondjuk, hogy illeszkedik a protokollhoz. A protokoll tekinthető egy adattagokat nem tartalmazó absztrakt bázisosztálynak is.

Például definiálhatunk egy olyan protokollt, amely azt fejezi ki, hogy egy osztály objektumait el lehet menteni fájlba, és onnan vissza is tölthetők:

```
@protocol Archivalas
-read: (FILE*) f;
-write: (FILE*) f;
@end
```

Azt, hogy az *Alakzat* osztály megvalósítja ezt a protokollt, az alábbi módon adhatjuk meg:

```
@interface Alakzat: Object
  <Archivalas>
```

Hasonlóan elkészíthetjük a referenciaszámlálás képességét kifejező protokollt is:

```
@protocol ReferenciaSzamlalo
- novelo;
- csokkento;
- (unsigned) referencia;
@end
```

Egy osztály egyszerre több protokollt is megvalósíthat:

```
@interface Alakzat : Object
  <Archivalas, ReferenciaSzamlalo>
```

Ezzel voltaképpen többszörös öröklődést fogalmazunk meg. Az Objective-C-ben ez csak a protokollok használatával valósítható meg.

A többszörös öröklődés protokollok között is megengedett, ez azt jelenti, hogy az így bevezetett protokollt megvalósító osztályok megfelelnek az öröklődési listában felsorolt protokolloknak is.

A Java interfészei

A Javában az interfészek egy új referencia típust jelentenek, mely absztrakt metódusok deklarációjának és konstans értékeknek a gyűjteménye.

Egy interfész mindig absztrakt, ezt fölösleges is külön jelezni, de csak a *public* minősítés használatával válik nyilvánossá, egyébként csak az adott csomagon belül használható.

Interfészeket az alábbi módon deklarálhatunk [Nyé 01]:

```
public interface Rajzolhato {
   public void kirajzol();
}

public interface Rendezheto {
   public boolean kisebbMint(Rendezheto a);
   public boolean Egyenlo(Rendezheto a);
}
```

Az interfészek között is létezik öröklődés, ezt a Javában kiterjesztésnek nevezzük. A *Rajzolhato* interfész kiterjesztése így történhet:

```
public interface Szines_Rajzolhato extends Rajzolhato {
    public int SZIN = 2;
    public int milyenSzinu();
}
```

A *Szines_Rajzolhato* interfész egy *SZIN* konstanst és csak két deklarált metódust (*kirajzol* és *milyenSzinu*) tartalmaz.

Interfészeknél megvalósítható a többszörös öröklődés is.

```
public interface Szines_Rendezheto extends Szines_Rajzolhato, Rendezheto {
  int MERET = 1;
}
```

Egy osztály egy interfészt az **implements** kulcsszóval valósíthat meg, ekkor az összes örökölt metódus törzsét meg kell adja:

```
class Alakzat implements Szines_Rajzolhato {
  public void kirajzol() { . . . }
  public int milyenSzinu() { . . . }
}
```

A Java szabványos könyvtára is számos interfészt tartalmaz, ez nagyon fontos például a grafikus felületek kezelése vagy a párhuzamosan futtatható programszálak megvalósítása esetén, de már a *java.util* csomagnak is része a Gyűjtemény Keretrendszer (Collections Framework Interface) [Nyé 01].

Object Pascal

Az Object Pascalban az interfészek absztrakt metódusok és tulajdonságok gyűjteményei. Az interfészeknek egyedi azonosítókat kell adni. Minden interfész alapértelmezésben az *IUnknown* interfész leszármazottja, de tetszés szerinti interfészhierarchiákat is definiálhatunk. Az *IUnknown* interfészben bevezetett metódusokat (*QueryInterface*, *_AddRef* és *_Release*) a *System* unitbeli *TInterfacedObject* osztályban valósították meg, célszerű az interfészeinket megvalósító osztályokat ebből származtatni. Például (lásd [Can 00]) bevezethetjük az *IRendezheto* és az *IRajzolhato* interfészeket (az interfészek nevét *I* betűvel szokás kezdeni):

```
type
  IRendezheto = interface
    [...]
    function kisebbMint(a : IRendezheto): Boolean; virtual;
    function Egyenlo(a : IRendezheto): Boolean; virtual;
  end;
  IRajzolhato = interface
    [...]
    procedure Kirajzol; virtual;
  end;
```

A *[...]* között megadjuk az azonosítójukat, és specifikáljuk a *kisebbMint*, az *Egyenlo*, illetve a *Kirajzol* műveleteket.

A *TAlakzat* osztályt létrehozhatjuk a *TInterfacedObject* osztály leszármazottjaként az *IRajzolhato* interfész megvalósításával:

```
type
  TAlakzat = class (TInterfacedObject, IRajzolhato)
   public
    procedure Kirajzol; virtual;
    ...
  end;
```

Egy osztály több interfészt is megvalósíthat[9].

```
type
  TRendezhetoAlakzat = class (TInterfacedObject, IRajzolhato, IRendezheto)
   public
    procedure Kirajzol; virtual;
    function kisebbMint(a : IRendezheto): Boolean; virtual;
    function Egyenlo(a : IRendezheto): Boolean; virtual;
    ...
  end;
```

A megvalósító osztály lehet absztrakt is, ekkor az absztrakt műveleteket a szokásos módon az **abstract** kulcsszóval kell jelölni.

C#

A C# interfész-lehetőségei hasonlóak a Javában bevezetetthez, bár konstansokat nem definiálhatunk bennük. Az interfészek tartalmazhatnak tulajdonság deklarációkat, indexelőket és eseménykezelőket is [C# Ref 03]. Interfészek között megengedett a többszörös öröklődés.

```
interface IAinterf {
   void MethA1();
   void MethA2();
}
interface IBinterf {
   void MethB1();
   int x { // Tulajdonság.
      get;
      set;
   }
}
interface IA_Binterf: IAinterf, IBinterf {
   void f();
   void g();
}
```

[9] Ha véletlenül azonos metódusnevek szerepelnek bennük, akkor az átnevezés is megoldható [Bor 99].

Az interfészeket osztályok vagy rekordok valósíthatják meg, ezt az öröklődési záradékban kell feltüntetni. Ha a megvalósító osztálynak van ősosztálya is, akkor ezt kell elsőként írni.

```
class Aoszt: Ososzt, IAinterf, IBinterf {
  private int myX;
  // Metódusok megvalósítása ...
  public int x {        // Tulajdonság megvalósítása.
    get { return myX; }
    set { myX = value; }
  }
}
```

Az interfészek a C#-ban is egy felületet specifikálnak, amely segíti a programozót hasonló viselkedésű objektumok közös szempontok alapján történő kezelésében.

12.7.7. Beágyazott osztályok, tagosztályok

Bizonyos programozási nyelvek nem csak a program legfelső szintjén engedik meg az osztályok definiálását, hanem más osztályok belsejében is. Ezeket nevezzük *beágyazott osztályoknak*, vagy *tagosztályoknak*.

Az alábbi Java példában (lásd [Nyé 01]) a *Lista* osztály *Elem* tagosztálya a külvilág elől rejtve marad, csak a *Lista* osztály műveleteiben érhetjük el:

```
public class Lista{
  private Elem elso;
  private static class Elem {
    Object adat;
    Elem elozo, kovetkezo;
    Elem (Object adat, Elem elozo, Elem kovetkezo){
      this.adat = adat;
      this.elozo = elozo;
      this.kovetkezo = kovetkezo;
    }
  }
  public void beszur (Object adat) { ... }
  public void torol (Object adat) { ... }
  private void torol (Elem elem) { ... }
  private Elem keres (Object adat) { ... }
}
```

A beágyazott osztályok lehetőségeiről bővebben például a *Java 2 útikalauz programozóknak* című könyvben [Nyé 01] olvashatunk.

12.8. Összefoglalás

A fejezetben megismerkedtünk az objektumorientált programozás jellemzőivel és megvalósításával néhány programozási nyelvben.

Az objektumorientált program egymással kapcsolatot tartó objektumok összessége, mely objektumok jól meghatározott jellemzőkkel és feladatokkal rendelkeznek.

Az objektumok a valós világ egy-egy egységét jelölik. Az objektumokat tulajdonságaik alapján osztályokba soroljuk. Áttekintettük, az osztályok és objektumok egyes nyelvekben való megvalósítását. Az osztályok és objektumok jelölésére egységes úgynevezett osztály-, illetve objektumdiagramokat is bevezettek. Ezek lehetővé teszik, hogy könnyen áttekinthető módon ábrázoljuk osztályok egymással való kapcsolatát, függetlenül a probléma megvalósításához választott programozási nyelvtől.

Az osztályokban az adattagok és a metódusok egységbezártan jelennek meg. Módszertanilag ajánlott, hogy az osztály egy objektumának adattagjait közvetlenül ne érhesse el más objektum, az objektum belső állapotát csak saját metódusai módosíthassák. Ennek megvalósítását szolgálják a különböző adatrejtési módok.

Az osztályoknak lehetnek olyan adataik és metódusaik, melyek nem egy-egy objektum példány létrehozásakor nyernek „értelmet", hanem az osztályhoz tartoznak, az osztály deklarációjakor jönnek létre. Ezeket osztályadatoknak, illetve osztálymetódusoknak nevezzük.

Az objektumorientált programozás legfontosabb tulajdonsága, hogy az öröklődés segítségével az osztályokat hierarchiákba rendezhetjük. Öröklődés során a leszármazott osztályban az ősosztály jellemzői mellett új jellemzőket hozhatunk létre, vagy a korábbi metódusok átdefiniálhatók. Egyes programozási nyelvekben az öröklődés során a láthatósági viszonyok is újraértelmezhetők.

Az öröklődés maga után vonja a helyettesíthetőség fogalmát, e szerint a származtatott osztály objektumai helyettesíthetik az ősosztály objektumait. A polimorfizmus és a dinamikus kötés lehetősége támogatja az egyes nyelvekben ennek megvalósítását.

Az absztrakt osztályokból (interfészekből) származtatott osztályokban más-más módon valósíthatjuk meg az absztrakt metódusokat. Ez segíti a programtervezést és megkönnyíti a módosíthatóságot.

Egyes nyelvekben megengedett a többszörös öröklődés, amikor egy osztálynak több közvetlen őse is lehet. A többszörös öröklődés, előnyei mellett problémákat is felvet. Interfészek használatával ezek nagyrészt kiküszöbölhetők.

Az objektumorientált programozás a mai szoftverfejlesztés legelterjedtebb módja. Ennek legfőbb oka, hogy hatékonyan támogatja az újrafelhasználható programok írását, a jól megtervezett osztályhierarchiák segítségével.

Feladatok

1. Milyen szabályok és ajánlások fogalmazhatók meg az egységbezárásnál?
2. Milyen probléma megoldásakor érdemes *friend* osztályt használni?
3. Mely mindennapi életből vett feladattal tudná jól szemléltetni az öröklődést?
4. Milyen előnyei vannak a közös ős létének?
5. Ismertesse egy példán keresztül a dinamikus kötést!
6. Hasonlítsa össze a C++, az Eiffel és a Java polimorfizmust támogató lehetőségeit!
7. Hasonlítsa össze a C++ többszörös öröklődését a Java interfészek által nyújtott támogatásával!
8. Hasonlítsa össze a különböző programozási nyelvek konstruktorait és destruktorait!
9. Hasonlítsa össze a C++ és az Eiffel többszörös öröklődését!

13 A programhelyesség-ellenőrzés nyelvi elemei

Amint már a könyv bevezetőjéből kiderült, a programok helyessége fontos minőségi jellemző. A fejezetben megvizsgáljuk, hogy milyen nyelvi eszközök állnak rendelkezésre ezen jellemző javítására. Az ismeretanyag tárgyalása egy konkrét példán keresztül történik. Rámutatunk arra, hogy a helyesség értelmezéséhez, a programozási nyelv formai és tartalmi követelményein túl, szükségünk van a program elvárt működésének megadására. Ezért először a program működését leíró, úgynevezett helyességi specifikációval foglalkozunk. Ezután fogunk csak hozzá a programhelyesség fogalmának értelmezéséhez. Megadjuk a ciklusok, a metódusok, az osztályok és a programok helyességének definícióját. Feltárjuk az összefüggéseket a helyesség, a kivételkezelés és az öröklődés között. Végezetül röviden ismertetjük a programhelyesség-ellenőrzés problémáit és korlátait.

Jelen fejezet azt mutatja be, hogy a programhelyesség-ellenőrzés területén alkalmazott módszerek hogyan és milyen mértékben támogatottak és alkalmazhatók egy modern programozási nyelv esetében.

Szándékosan használjuk az ellenőrzés szót a bizonyítás vagy a tesztelés helyett. Mint ismeretes, a programok helyességének ellenőrzésére alapvetően két alternatíva létezik:

A leggyakrabban alkalmazott módszer a tesztelés. A tesztelés során az elkészült program viselkedését előre meghatározott bemenő adathalmazon vizsgálják. Ellenőrzik, hogy a program működése megfelel-e az elvártaknak.

A másik alternatíva, amikor a programokat nem teszteljük, nem próbáljuk ki, hanem megkísérlünk matematikai precizitással érvelni arról, hogy helyesek. Ezt a programok helyességéről való tudományos alapokon nyugvó érvelést nevezzük programhelyesség-bizonyításnak.

A fejezetben közölt ismeretanyag gyakorlati jellegű. Mindkét korábban ismertetett alternatíva jegyeit magán viseli bizonyos mértékben.

Ismertetjük, hogy a programhelyesség-bizonyítás elméleti háttere hogyan alkalmazható egy választott programozási nyelvben. Az elmélet alkalmazásával, annak gyakorlati megvalósításával azonban eltávolodunk az elméleti háttértől. A fejezet középpontjában nem a programhelyesség-bizonyítás elméleti kérdései állnak. Nem célunk bemutatni a programhelyesség-bizonyítás módszereit, elméleti alapjait, mindazonáltal rövid áttekintést adunk az elmélet és gyakorlat közötti összefüggésekről.

Rámutatunk arra is, hogy a programhelyesség témakörével kapcsolatos nyelvi elemek programozási nyelvre való ültetésével lehetőség nyílik a programok helyességének futásidejű ellenőrzésére. A program futása során végrehajtott ellenőrzés rokonságot mutat a teszteléssel. A lényeges különbség, hogy ebben az esetben a program működésének ellenőrzése egy belső nyelvi mechanizmuson keresztül történik. Az ellenőrzés pedig éppen a programviselkedés formális megadásának köszönhető, mely a program megvalósításába épül bele. Ezt a program működését megadó formális leírást pedig éppen a programhelyesség-bizonyítás elméletének nyelvi szerkezetre való átültetése teszi lehetővé.

Választás

A fejezet fókuszában az Eiffel programozási nyelv programhelyesség-ellenőrzésének nyelvi eszközei és lehetőségei állnak. Az Eiffel nyelv választása mellett több érv is szólt. Az Eiffel egy modern objektumorientált nyelv, melyben a programhelyesség-ellenőrzés szinte minden nyelvi lehetősége adott. A nyelvhez korszerű fejlesztői környezet társul. Az Eiffel nyelv és fejlesztői környezet használatának erőssége három terület szoros összefonódásán alapszik.

- A programhelyesség-bizonyítás elméleti hátterére épül az Eiffel nyelv helyesség-ellenőrzést támogató eszköztára és számos nyelvi szerkezete.
- A komponensalapú fejlesztéshez a nyelv alkotói egy új programozási szemléletet, módszertant dolgoztak ki, mely Design by Contract [Mey 00] néven terjedt el. Ez a módszertan rendkívül hatásosan alkalmazható sok, de kevéssé összetett elemből álló bonyolult rendszerek tervezése és minőségbiztosítása során.
- Az Eiffel nyelv fejlesztői keretrendszere praktikus segédletet nyújt a programhelyesség-ellenőrzés futásidejű vizsgálatára és a kialakított módszertan lehető legnagyobb támogatására.

Végezetül a nyelv viszonylagos ismertsége és népszerűsége (legalábbis az oktatási és kutatási területen) ismételten megerősített minket választásunkban.

Az Eiffel nyelv mellett számos más nyelv is felmerült alternatívaként. Az Alphard nyelv egyike volt azon nyelveknek, melyben programozási nyelvi eszközként megjelentek a programhelyesség- bizonyításhoz szükséges nyelvi elemek az 1970-es évek közepén [Wul 74]. A nyelv megvalósítása és programhelyesség-ellenőrzést biztosító szerkezetei bizonyos értelemben túlmutatnak az Eiffel nyelv által biztosított lehetőségeken. Az absztrakt adattípus és konkrét megvalósítása közötti párhuzam tisztábban van jelen, a bejárókhoz megadható szabályok pedig unikumnak számítanak. Sajnos azonban a nyelvhez nem készült (hatékony és teljes körű) fordító és a nyelv egyáltalán nem ismert.

Egy másik alternatíva lehetett volna a Sather nyelv [Gom 97], mely egy modern objektumorientált programozási nyelv. A Sather szintaxisában a C/C++ irányvonalát követi, míg szemantikája egy részét – beleértve a programhelyesség-bizonyítás nyelvi eszközeit is – az Eiffeltől örökölte. A C/C++-szerű nyelvtana miatt – megítélésünk szerint – azonban a Sather nyelven írt programok a témába most betekintőknek nehezebben érthetők, mint az Eiffel nyelvi megfelelőik. Továbbá a Sather programhelyesség-ellenőrzés támogatásában sem tartalmaz újdonságokat az Eiffelhez képest, bár öröklődési- és típusrendszere talán jobban átgondolt.

A témával mélyrehatóbban foglalkozni kívánók figyelmébe ajánljuk a Turing [Hol 83, Hol 84] és Euclid [Lam 77] nyelveket is, melyek az Alphard kortársai voltak.

Gondolatébresztő

Mielőtt a fejezet témájában elmélyednénk, gondoljuk át, hogy mit is értünk helyesség alatt. A köznapi életben gyakran használjuk a következő kifejezéseket: a program jól működik,

a program helyes, vagy éppen a program hibás. Próbáljuk meg tisztázni a használt fogalmak tartalmát. Ehhez vegyük az alábbi rövid programrészt, mely egy függvényt valósít meg Eiffelben.

```
f : INTEGER is
do
   result := 1
end
```

Ha feltesszük a kérdést, hogy helyes-e a fenti program, nehezen tudunk rá érdemlegesen válaszolni.

Ahhoz, hogy a program egyáltalán értelmezhető legyen számunkra vagy a fordítóprogram számára, a programnak teljesítenie kell bizonyos formai követelményeket. Ezek a követelmények adják a programozási nyelv szintaxisát.

A szintaxis önmagában nem elegendő a program működésének megértéséhez. Szükségünk van az egyes nyelvi szerkezetek jelentésének és hatásának ismeretére is. Ezt nevezzük szemantikának.

A szintaxis és szemantika birtokában még mindig nem vagyunk képesek megválaszolni a feltett kérdést. Ez azért van így, mert a programhelyesség relatív fogalom. Egy program nem tekinthető helyesnek, vagy éppen helytelennek önmagában. A helyesség értelmezéséhez szükségünk van a programmal szemben támasztott követelményeket megadó leíráshoz, a megoldandó probléma minél pontosabb megfogalmazásához. Erre a leírásra a továbbiakban mint helyességi specifikációra, vagy ha nem okoz félreértést, akkor egyszerűen mint specifikációra fogunk hivatkozni. A fenti program nyilván nem teljesíti a következő specifikációt: „A program tetszőleges másodfokú egyenlet valós gyökeit számolja ki." Ugyanakkor a program maradéktalanul megfelel a „A program egy konstans függvényt valósít meg. A függvény visszatérési értéke 1" specifikációnak.

Az objektumorientált megközelítés sajátosságai

Az objektumorientált megközelítés előtt a strukturált programok lényeges tulajdonságai jellemezhetők voltak egy igen egyszerű modellel. A modell egy fiktív programozási nyelvre, az úgynevezett *while programozási nyelvre* épült [Loe 87] (lásd a 3.3.1. alfejezet). Egyszerűsége ellenére rendelkezett a harmadik generációs programnyelvek majdnem minden lényeges tulajdonságával és megfelelő alapot biztosított a programok helyességének formális igazolására is.

Az objektumközpontú programozás esetén ez az egyszerű modell nem alkalmazható. Az objektumorientált programok modellezése jóval bonyolultabb modellt követel meg. Megjegyezzük, hogy bár léteznek az objektumorientált megközelítést formálisan leíró elméleti modellek, úgynevezett objektum-kalkulusok [Aba 98, Gun 94], de ezek tárgyalására itt nem térünk ki.

Absztrakt adatábrázolás

Az objektumorientált számítási modell középpontjában a függvények és eljárások helyett az absztrakt adattípus áll. A nyelvi eszközök lehetővé teszik az absztrakt adattípus tulajdonságainak és műveleteinek egységbe zárását és a belső tulajdonságok, illetve megvalósítás külvilágtól való elrejtését. Az absztrakt adattípus szolgáltatásai a külvilág számára biztosított nyilvános eljárásgyűjtemény felületen érhetők el. A szolgáltatások megvalósítása a külvilág számára rejtett és nem hozzáférhető.

Típusrendszer

Az objektumközpontú programozás egyik legfontosabb alappillére a bővíthető típusrendszer. Nemcsak az alaptípusok szélesebb köre áll rendelkezésre, de a fejlesztő új típusokat is létrehozhat a már meglévő típusok kombinációjával, különböző típuskonstrukciókkal. Olyan önmagukban is érdekes és magas elvonatkoztatási szintű eszközök állnak rendelkezésre, mint például a generikusság, mely típussal paraméterezhető modulok készítését teszi lehetővé. A többalakúság és öröklődés további érdekes és hatásos eszközöket nyújtanak.

Itt említjük meg, hogy a programhelyességhez, mint fogalomhoz, természetesen hozzátartozik az is, hogy a program típusosan helyes. A típushelyesség kérdésének megválaszolása maga is érdekes és rendkívül sokoldalú feladat. Ez részben a fenti szerkezeteknek, illetve a következő bekezdésben megemlítésre kerülő dinamizmusnak tudható be. A típushelyesség problémakörére itt nem térünk ki, mert túlmutatna a könyv lehetőségein. A továbbiakban feltételezzük, hogy az általunk írt programok típusosan helyesek.

Dinamikusság

Az objektumorientált megközelítésben fokozottan jelen van a dinamikusság. Ez alatt azt értjük, hogy a program futása során objektumok keletkeznek és szűnnek meg. Az objektumokra hivatkozás típusú változókkal hivatkozhatunk. Mivel általánosságban egy objektumra több változó is hivatkozhat, ugyanaz az objektum több módon is elérhető és módosítható. Ezt nevezzük idegen szóval aliasing-nak.

Objektumorientált problémamegoldás

Egy probléma objektumorientált megoldása esetén általában osztályok nagyobb csoportja van jelen. Az osztályok különböző mértékben kapcsolódnak egymáshoz. A feladat megoldásához osztályok példányainak összehangolt működésére, kommunikációjára van szükség. Egy probléma megoldásakor, a megvalósítás helyességének ellenőrzése és igazolása során az összes, a problémamegoldás során használt osztály helyessége igazolandó.

Megközelítés

Az előző bekezdésben felsorolt sajátosságok miatt az objektumorientált programok helyességének ellenőrzése egészében rendkívül bonyolult. Továbbá az objektumorientált probléma-

megoldás jellege miatt egy program helyességének ellenőrzése a programot alkotó osztályok tucatjainak vizsgálatát jelenti.

A reménytelennek látszó helyzet megoldása egy egyszerű ötleten alapszik. Megpróbáljuk a problémát (jelen esetben a program helyességének ellenőrzését) elemi problémákra osztani. Az alapötlet az, hogy az egyes osztályok helyességének ellenőrzését külön kezeljük. Az ellenőrzött, kipróbált osztályok már mint helyes építőelemek használhatók.

A módszer azért alkalmazható, mert az objektumorientált megközelítésben használt adatabsztrakciónak köszönhetően különválasztható az osztály felülete és megvalósítása. Ha csak egy osztály megvalósítása változik és felülete nem, akkor az őt használó többi osztályra a változtatás nincs (közvetlen) hatással, ezeknek az osztályoknak a módosítása, újbóli ellenőrzése nem (feltétlenül) szükséges.

Ezt a megközelítést használja az Eiffel programnyelv és a hozzá kapcsolódó módszertan is. A megközelítés az alábbi előnyökkel jár:

- Az egyes osztályok helyessége viszonylag könnyen ellenőrizhető, kezelhető.
- A program módosítása csak a megváltozott programrészek (bizonyos osztályok) helyességének újbóli igazolásával, ellenőrzésével jár.
- A már ellenőrzött, helyes osztályok könyvtárba rendezhetők és újra felhasználhatók. Ilyen ellenőrzött, helyesen működő elemeknek tekinthetők az Eiffel programozási nyelv részét képező alaposztályok.

Áttekintés

Jelen fejezetben először részletesen a helyességi specifikációt alkotó nyelvi eszközöket ismertetetjük. A helyességi specifikáció nyelvi elemeit egy példán keresztül mutatjuk be az olvasónak, lehetőséget biztosítva arra, hogy a nyelvi elemek használatának előnyeit a gyakorlatban is megfigyelhesse.

Az Eiffel programok helyességi specifikációjának birtokában képesek leszünk a programhelyességet, mint fogalmat értelmezni. Látni fogjuk azonban, hogy a helyesség nem minden esetben fejezhető ki a specifikációs nyelvben.

Végezetül röviden vázoljuk a programhelyesség-ellenőrzés elméleti és gyakorlati problémáit.

13.1. A helyességi specifikációs nyelv

A korábban ismertetett rövid programhoz tartozó helyességi specifikációk a feladat követelményeit szövegesen adták meg. A szöveges megadással alapvetően két probléma van. Ha eszközökkel szeretnénk támogatni a programok helyességének ellenőrzését, az ilyen jellegű szöveges megfogalmazás nem megfelelő, mert az alkalmazandó eszköz számára nem értelmezhető. További probléma, hogy a szöveges leírások sokszor pontatlanok és félreérthetők. Ezért a következő részben ismertetjük azokat a nyelvi eszközöket, melyek az Eiffel programozási nyelven írt programok helyességi specifikációs részét képezik. Ezek a nyelvi eszközök megfelelnek mindkét feltételnek. Mint az Eiffel programozási nyelv részei a fordítóprogram

által értelmezhetők, továbbá segítségükkel pontosan és egyértelműen adható meg a programtól elvárt működés.

Az Eiffel nyelv helyességi specifikációja alatt a nyelv egy bizonyos részét értjük. A helyességi specifikációs nyelv a teljes Eiffel programozási nyelv egy valódi részhalmaza, mely elsősorban logikai kifejezésekre épül. Ezek a kifejezések deklaratív módon fogalmazzák meg a program működését. A specifikációs nyelv az Eiffel programozási nyelv részeként a „mit" kérdésre válaszol, a program által elvégzett feladatot egy magasabb elvonatkoztatási szinten határozza meg. Maga a helyességi leíró elemektől megfosztott Eiffel program a problémát imperatív módon oldja meg – a „hogyan" kérdésre ad választ – hasonlóan ahhoz, ahogy azt a legtöbb elterjedt programozási nyelv, például a Pascal, a C, a C++ vagy a Java teszi.

Megjegyezzük, hogy a helyességi specifikációs nyelv a program megvalósításával összefüggő, de attól mégis elvontabb, különálló dolog. Nem törvényszerű, hogy az Eiffel programok helyességi specifikációja is Eiffel nyelvi szerkezetekből álljon. Léteznek a programozási nyelvektől különálló specifikációs nyelvek is. Az egyik legismertebb talán a Z [Spi 92] és annak objektumközpontú változata az Object-Z [Smi 00]. Annak, hogy ez mégis így van praktikus okai vannak. Egyrészt megkönnyíti a fejlesztő feladatát, mert nem kívánja meg a programozási nyelven kívül egy specifikációs nyelv elsajátítását. Másrészt megfelel annak a követelménynek is, miszerint a helyességi specifikációs nyelv jól illeszkedjen a programozási nyelvhez. Az illeszkedés bizonyos szempontból szinte tökéletes, mert az Eiffel nyelv helyességi specifikációjában – amint ezt látni fogjuk – közvetlenül hivatkozhatunk nyelvi változókra, függvényekre.

13.1.1. Az Eiffel és az elsőrendű logika

A specifikációs nyelvek többnyire az elsőrendű logikára épülnek. Az elsőrendű logikai formulák segítségével könnyen megfogalmazhatók olyan állítások, melyek a program működését deklaratív módon adják meg. Ezért érdemes megvizsgálni, hogy milyen kapcsolat van az elsőrendű logika és az Eiffel nyelv specifikációs nyelvi lehetőségei között. A 13.1. táblázatban összevetjük az elsőrendű logika elemeit az Eiffel nyelvi megfelelőikkel.

Az Eiffelben kifejezhető jól formált formulák halmaza induktívan a következő módon adható meg:

- *True*, *False* jól formált formulák.
- *BOOLEAN* típusú állandók és változók jól formált formulák.
- Ha t_1 és t_2 változó, objektum, kifejezés vagy függvény, akkor $t_1 = t_2$, *equal*(t_1, t_2), *deep_equal*(t_1, t_2) jól formált formulák.
- Ha t_1, t_2,... t_n változó, objektum, kifejezés vagy függvény és P n argumentumú, *BOOLEAN* visszatérési értékkel rendelkező függvény, akkor $P(t1, t2, \ldots tn)$ jól formált formula.
- Ha f_1, f_2 formulák, akkor `not` f_1, f_1 `or` f_2, f_1 `and` f_2, f_1 `implies` f_2, f_1 `or else` f_2, f_1 `and then` f_2 jól formált formulák.

Elsőrendű logika	Eiffel	Megjegyzés
$\neg, \wedge, \vee, \Rightarrow, \equiv$	Rendre **not**, **and**, **or**, **implies**	Az ekvivalencia az $(a \equiv b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$ azonosság alkalmazásával adható meg.
$=$	$=$, *equal*, *deep_equal*	Objektumok értékeinek összehasonlítása esetén *equal*, vagy *deep_equal* szerkezetek alkalmazhatók.
Kvantorok ($\exists$, $\forall$)	Nincs	Közvetlenül nem írhatók. Függvények és ciklusok együttes alkalmazásával elméletileg kivitelezhetők.
Változó	Eiffel változó	
Igazságérték	*True*, *False*	
Konstansok	Konstans érték	
Függvény-szimbólumok	Eiffel függvények	Globális mellékhatás nélküli függvények, melyek az objektum attribútumait nem változtatják meg.
Predikátum-szimbólumok	*BOOLEAN* visszatérési értékkel rendelkező Eiffel függvények	Globális mellékhatás nélküli függvények, melyek az objektum attribútumait nem változtatják meg.
Kifejezés	Konstans, változó vagy függvény	Természetesen függvény argumentumában is szerepelhet függvény.

13.1. **táblázat:** Az elsőrendű logika elemei és a megfelelő Eiffel konstrukciók

Logikai műveletek

A figyelmes olvasó észrevehette, hogy a jól formált formulák definíciójánál a szokásos **and** és **or** logikai műveleteken kívül más műveleteket is alkalmaztunk. Az Eiffel nyelvben a megszokott logikai műveleteken kívül használhatók még az **and then**, **or else** és az **implies** műveletek is. Az **implies** művelet a logikai következtetés, ahogy az az összefoglaló táblázatból már kiderült. Az **and then** és **or else** rendre az **or** és **and** műveletek nem szigorú megfelelői.

A kifejezés kiértékelése mindkét művelet esetében szekvenciális. Ez alatt azt értjük, hogy először az első tag értékelődik ki. Ha a tag kiértékelése során a művelettel képzett logikai kifejezés értéke eldőlt, akkor nem kerül sor a második tag kiértékelésére. Például az **and then** esetében ha az első tagot kiértékelve hamis értéket kapunk, akkor nem kerül már sor a második tag kiértékelésére. Hasonlóan az **or else** logikai művelet esetében a második tag kifejezése csak abban az esetben értékelődik ki, ha az első tag értéke hamis.

Az *a* **and then** *b* logikai művelet az Eiffelben a következő függvénnyel adható meg:

```
function and_then(a, b : BOOLEAN) : BOOLEAN is
do
  if a then
     result := b
  else
    result := False
  end
end
```

Az *a* **or else** *b* egyenértékű a következő Eiffel nyelvű programrésszel:

```
function or_else(a, b : BOOLEAN) : BOOLEAN is
do
  if a then
    result := true
  else
    result := b
  end
end
```

Figyeljük meg, hogy ha mindkét tag kiértékelhető, akkor az **and then** és az **or else** műveletek jelentése rendre megegyezik az **and** és az **or** műveletekével (feltéve, hogy a műveletek nem rendelkeznek mellékhatással). A különbség a szigorú és nem szigorú műveletek esetében csak ott mutatkozik, ahol a második tag nem kiértékelhető (a tagot kiértékelve az eredmény nem definiált, például a második tag értékét előállító függvény futása nem fejeződik be). A szigorú műveletek esetében ha bármely tag nem definiált, akkor az egész kifejezés nem definiált. Míg a nem szigorú logikai műveletek esetében, ha az első tag kiértékelése eldönti a kifejezés logikai tartalmát, akkor a második tag figyelmen kívül hagyatik, függetlenül attól, hogy definiált-e vagy sem.

Az **and then** logikai művelet jól alkalmazható olyan esetekben, amikor a második feltétel kiértékelhetősége megköveteli az első feltétel teljesülését. Ha azt szeretnénk eldönteni, hogy *j* változó maradék nélkül osztható-e *i*-vel, akkor próbálkozhatunk az alábbi programrésszel.

```
if j \\ i = 0 then ...
```

A példában a \\ művelet az egész részes osztás maradékát képzi. Ez az esetek többségében helyesen működik, de figyelmen kívül hagyja az *i=0* esetet. Ekkor a program futása nullával való osztás miatt megszakad. Helyette alkalmazhatjuk az

```
if i /= 0 then
   if j \\ i = 0 then ...
```

szerkezetet. Ez már helyes és megfelelően működik. Sajnos azonban az eredeti feltételünk egy újabb **if**–**then**–**else** ággal bővült. Összetettebb feltételek ellenőrzése esetén a felesleges

if–**then**–**else** szerkezetek alkalmazása megnehezíti a feltétel megértését, a program karbantarthatóságát. Éppen az ilyen és ehhez hasonló feltételek elegáns megadására találták ki az **and then** műveletet. Segítségével a feltétel az első alaknak megfelelően írható, a különbség csak az, hogy helyesen is fog működni, hiszen a második tag kiértékelése csak abban az esetben történik meg, ha az *i* változó értéke nullától különböző.

if *i* /= *0* **and then** *j* \\ *i* = *0* **then** ...

A nem szigorú logikai műveletek az Eiffel nyelv számos programhelyességet támogató nyelvi eszközében jól használhatók. Fontos követelmény azonban, hogy pontosan tisztában legyünk az egyes logikai műveletek szemantikájával.

Logikai állítások

Az Eiffel nyelv helyességi specifikációs nyelvének legfontosabb alapelemei a logikai állítások. Ezek az állítások az Eiffel nyelvben megadható jól formált formulák, melyek a később bemutatandó metódusok elő- és utófeltételét, az osztály- és ciklusinvariánst alkotják.

A logikai állítások, mint ahogy a nevük is mutatja, *BOOLEAN* típusú kifejezések. Magát az állítást egy nem kötelező megnevezés vezeti be és egy megjegyzés zárja. A logikai állítások a program bizonyos pontján lévő logikai összefüggéseket írják le. Ezek az összefüggések vonatkozhatnak attribútumokra, helyi változókra, vagy függvények visszatérési értékeire. A logikai állítások a helyességi specifikáció alapját képezve deklaratív módon írják le a program működését, de ugyanakkor elősegítik a program megértését és jó támpontot nyújtanak a dokumentációnál is.

Például ha a program egy bizonyos részén a *csucsszam* változó a sokszög csúcsainak számát jelenti, és a specifikáció részeként azt a feltételezést szeretnénk megadni, hogy az alakzat háromszög vagy négyszög, akkor az az alábbi módokon történhet:

csucsszam = *3* **or** *csucsszam* = *4*

vagy

haromszog_vagy_negyszog: csucsszam = *3* **or** *csucsszam* = *4*

Fontos megjegyezni, hogy a feltételezés megadása azt jelenti, hogy ha a program helyesen működik, akkor az adott ponton a *csucsszam* változóban tárolt érték pontosan 3 vagy 4 lehet. Ha ez nem teljesül, akkor a program nem áll összhangban a helyességi specifikációjával, ami jelen esetben éppen a fenti logikai állítás.

A két kifejezés tartalmilag egyenértékű, különbség csak a megnevezésben mutatkozik. A megnevezés a fejlesztő számára ad információt a mögé írt állításról. Az első logikai állítás jelentése talán nem mindenkinek nyilvánvaló, ezért van szükség a logikai állítás megnevezéssel való ellátására.

Az első logikai állítás jelentése nem egyértelmű, míg a második állításé igen, az elé írt megnevezés miatt. Egy hasonlattal élve ez olyan, mintha egy állandó értéket magával az értékével, vagy pedig egy azonosítóval adnánk meg. Egy programkódban a 23-as szám sokféle jelentéssel bírhat. Lehet életkor, évszám stb. Ha azonban a `Bp_keruletek_szama` azonosítót használjuk ugyanazzal a 23-as értékkel, akkor az több tartalmat hordoz.

Az olvasó talán azt gondolhatja, hogy a megnevezés fölösleges, hiszen a logikai állítás után megjegyzés formájában is megadható a plusz információ. Ez a megállapítás részben helyes. Valóban lehetőségünk van a logikai állítás mögött elhelyezett megjegyzés formájában is utalni az állítás tartalmára, sőt sok esetben ez az egyetlen megoldás.

Vannak ugyanis olyan logikai állítások, melyek nem, vagy csak nagyon körülményesen fejezhetők ki a nyelvben. Láttuk, hogy az elsőrendű logika nem minden szerkezete alkalmazható az Eiffelben. Például kvantorokat és kvantorokat tartalmazó formulákat (közvetlenül) egyáltalán nem használhatunk. Azaz nem tudunk elegánsan és könnyedén olyan formulákat képezni, mint például „minden pont a sokszögön belül helyezkedik el".

De még ha lehetőségünk is lenne az elsőrendű logika teljes kifejezőerejét használni, akkor is bajban lennénk bizonyos esetekben. Ilyenkor nem marad más hátra, mint az üres logikai állítás megnevezéssel és megjegyzéssel. Ilyen lehet például:

graf_kormentes: -- g körmentes gráf

A fenti logikai állítás üres, hisz nem tartalmaz mást, mint megnevezést és megjegyzést. Így a fenti logikai állítás nem hordoz érdemi logikai tartalmat a program számára. Pontosabban az üres logikai állítások értéke definíció szerint *True*. Azaz a fenti logikai állítás egyenértékű a

graf_kormentes: True -- g körmentes gráf

logikai állítással. A fentihez hasonló üres logikai állítások természetesen nem lényegesek a helyességi specifikáció és a futtatói környezet szemszögéből. Használatuk azonban segítséget nyújt a dokumentációnál és a program megértéséhez is támpontot ad.

Felhívjuk a figyelmet, hogy a logikai állítások nem képezik részét a számítási folyamatnak, nincsenek semmilyen hatással (közvetlenül) a program futására, mindössze a program bizonyos pontjain mint a program működésével kapcsolatos deklaratív megfogalmazások szerepelnek.

Tehát a logikai állítások használata során nem arról van szó, hogy megvizsgáljuk, hogy a feltétel teljesül-e és ha igen, akkor folytatjuk az algoritmust az egyik ágon, ha nem, akkor pedig a másikon. Egy logikai állítás egy, a program belső működésére vonatkozó hipotézis. Ha a fenti példával élve a *haromszog_vagy_negyszog* megnevezéssel azonosított logikai állítást helyezzük el a program egy adott pontján, akkor azon a ponton feltételezzük, hogy a *csucsszam* változó értéke 3 vagy 4. Ez a feltételezés pedig a program algoritmusának ismeretén alapszik.

Ha azt szeretnénk eldönteni, hogy az alakzat háromszög vagy négyszög, és ha igen, akkor A algoritmust alkalmazni, ellenkező esetben B algoritmust alkalmazni, akkor nem logikai állításra van szükségünk. Ez egyszerűen megtehető más nyelvekhez hasonlóan egy **if**–**then**–**else** elágazással.

```
-- háromszög vagy négyszög
if csucsszam = 3 or csucsszam = 4 then
  -- A algoritmus
else
  -- B algoritmus
end
```

A logikai állításoknak a program futására kifejtett hatása a fordítási és futtatási paramétertől függ. Elégedjünk meg annyival, hogy a logikai állítások ellenőrzése engedélyezhető és letiltható. Ha a program futása során minden logikai állítás ellenőrzése letiltott, akkor a logikai állítások semmilyen hatással nincsenek a program végrehajtására, hiszen nem kerül sor a kiértékelésükre sem. Ha egy logikai állítás kiértékelése engedélyezett és a kiértékelés *False* értéket ad vissza, a program futása kivétel kiváltásával megszakad.

Visszatérve a logikai állításokban alkalmazott megnevezés és megjegyzés közötti különbségre, elöljáróban csak annyit jegyzünk meg, hogy ilyen esetekben, ha a program futása a logikai állítás kiértékelése miatt szakad meg, a futtató környezet a logikai állításra az állításnál megadott megnevezéssel hivatkozik. Ha ez a megnevezés hiányzik, a logikai állítás nehezebben azonosítható, ugyanis a megjegyzés nem jelenik meg a hibaüzenetben.

13.1.2. Példa

A specifikációs nyelvbe való rövid betekintés után az Eiffel programozási nyelv programhelyesség-ellenőrzést támogató nyelvi eszközeit egy egyszerű példán keresztül mutatjuk be. Példának a vermet, mint absztrakt adattípust választottuk. Azért esett erre a választásunk, mert a verem működése közismert és jól szemléltethető. A verem mint absztrakt adattípus az Eiffelben egy osztály lesz. Az osztályt lépésről lépésre építjük fel. Minden lépésnél bemutatjuk, hogy az Eiffel programozási nyelv milyen lehetőségeket biztosít a programhelyesség ellenőrzése során.

A verem mint absztrakt adattípus

Röviden fogalmazzuk meg, illetve pontosítsuk, hogy mit is értünk verem alatt. A verem, mint adattípus olyan struktúra, melybe azonos típusú elemek helyezhetők. A veremben az elemek LIFO (az angol Last In First Out szavak rövidítése) módon érhetők el, mely a következőket jelenti: a verembe helyezendő elem mindig a verem tetején foglal majd helyet; ha egy újabb elemet helyezünk el, akkor az ismét a verem tetején helyezkedik el, míg az előzőleg elhelyezett elem a második elem lesz a verem tetejétől számítva; a veremből mindig a legfelső (legutoljára elhelyezett) elem érhető (közvetlenül) el: ezt a legfelső elemet tudjuk kiolvasni, vagy törölni; innen ered az angol elnevezés is. A verem kezeléséhez az alábbi alapvető műveletekre lesz majd szükségünk:

- Új elem elhelyezése a veremben (*berak*).
- Aktuális elem törlése a veremből (*torol*).

A verem tartalmának kiolvasásához használjuk majd a következő lekérdezést:

- Aktuális elem kiolvasása a veremből (*elem*).

Természetesen a műveletek más halmaza is megadható, mi azonban most megelégszünk a fentiekkel. Belátható, hogy a felsorolt műveletekkel a verem kezelése megoldható és más összetettebb műveletek is felépíthetők segítségükkel. Például a legfelső elem kivétele a veremből megadható az aktuális elem kiolvasása és aktuális elem törlése műveletek egymás utáni alkalmazásával. Az érdekesség kedvéért definiáljunk egy *fordit* műveletet is, ami a veremben elhelyezett elemek sorrendjét cseréli meg. A legalsó elem lesz a legfelső, alulról a második lesz felülről a második és így tovább.

Praktikus okokból (például mert a számítógépek tárolókapacitása véges) szeretnénk korlátozni a verembe helyezhető elemek számát. Részben a korlátozás miatt a fenti műveleteken kívül még az alábbi lekérdezésekre is szükségünk lesz:

- A verem ürességének vizsgálata (*ures*). Igaz értéket ad vissza, ha a verem nem tartalmaz elemet.
- A verem telítettségének vizsgálata (*tele*). Igaz értéket ad vissza, ha a verembe nem helyezhető új elem.

Ezzel a végére is értünk a feladat megfogalmazásának.

Ábrázolás

Egy absztrakt adattípus megvalósítása során az absztrakt típust ábrázolni szükséges a már meglévő komponensekkel, osztályokkal. Mi a vermet, mint absztrakt adattípust, egy generikus tömbbel és két egész típusú változóval fogjuk ábrázolni. A tömb fogja tartalmazni a verem elemeit. Az *elemszam* változó megadja, hogy a veremben egy adott pillanatban hány elem található, míg a *kapacitas* a verembe helyezhető elemek legnagyobb megengedett számát jelzi. Az üres vermet a nulla elemszám jellemzi.

Az *elemszam* változó használatának előnye a hatékonyság. A veremből törölt elemeket fizikailag nem töröljük, hanem egyszerűen az *elemszam* változó értékét csökkentjük. Ez nem okoz problémát, mert a veremnek az elemszámnál nagyobb indexű elemei nem érthetők el. Ha pedig a verembe új elemet helyezünk, akkor az *elemszam* változót növeljük, majd felülírjuk a régi elemet.

```
class VEREM[G]
feature{NONE}
        tomb:       ARRAY[G]
        elemszam:   INTEGER
        kapacitas:  INTEGER
```

Figyeljük meg, hogy a változók szelektív láthatósági köre *NONE*, ami azt jelenti, hogy a változók csak az objektum metódusai számára láthatók.

Természetesen a *VEREM* osztály más reprezentációja is megadható. A verem ábrázolható listaként is tömb helyett, vagy örökölhetnénk a tömb (*ARRAY*) osztálytól ahelyett, hogy globális változóként használjuk.

13.1.3. Elő- és utófeltételek

Vegyük észre, hogy a fenti műveletek egy része nem értelmezett a verem minden állapota esetén. Az *ures* és *tele* metódusok meghívása bármely verem esetén a megfelelő értéket kell, hogy szolgáltassa, de például a *berak* és *torol* művelet már nem minden esetben értelmezhető. Egy üres veremből nem lehet újabb elemet törölni, míg egy teli verembe nem lehet újabb elemet helyezni.

Parciális és totális függvények

Egy-egy műveletre úgy is tekinthetünk, mint egy függvényre. Például a *berak* művelet egy kétváltozós függvény, melynek paraméterei egy verem és egy elem, visszatérési értéke pedig az elemmel bővített új verem.

$$berak : VEREM \times ELEM \mapsto VEREM$$

A *berak* művelet egy parciális függvényt valósít meg, mert nem minden bemenő paraméterre értelmezett a művelet. Ha az első argumentum egy teli vermet tartalmaz, akkor a művelet nem értelmezett, mert a teli verem már nem bővíthető újabb elemmel. Vegyük észre, hogy számunkra a parciális függvények használata nem kielégítő, egy program elkészítése során totális függvényeket kell alkalmaznunk, olyanokat, melyek minden engedélyezett paraméterkombináció esetében biztosan eredményt szolgáltatnak. A programmal szembeni elvárásunk az, hogy egyértelmű eredményt szolgáltasson. A megoldás az „engedélyezett" határozóban rejlik. Vegyük az alábbi matematikai függvényt:

$$f(x) = \frac{1}{x} \qquad x \in \mathbb{R}$$

Ez a függvény szintén parciális, mert $x = 0$ esetén nem értelmezett. Az f parciális függvényből viszonylag könnyen készíthető egy neki megfelelő f' totális függvény.

$$f'(x) = \frac{1}{x} \qquad x \in \mathbb{R}, x \neq 0$$

A programozás során is ezt a megoldást fogjuk alkalmazni. Egy művelet argumentumainak a halmazát, azaz a függvény értelmezési tartományát addig szűkítjük, amíg a műveletnek megfelelő függvény totális lesz.

Előfeltétel

A függvény értelmezési tartományára vonatkozó megszorítást előfeltételnek tekinthetjük. Az előfeltétel az értelmezési tartományon definiált karakterisztikus függvény, mely megadja az értelmezési tartomány azon elemeit, amelyekre a függvény értelmezett. (A karakterisztikus függvény a tartományhoz tartozó elemekhez igaz értéket, a többi elemhez hamis értéket rendel.) A fenti példában az $x \neq 0$ predikátumot az f' függvény előfeltételének tekinthetjük.

Természetesen az Eiffelben az előfeltételek nemcsak a függvény vagy művelet értelmezési tartományára adhatnak megszorítást, hanem figyelembe vehetnek globális attribútumokat és összefüggéseket is.

Az, hogy mit válasszunk a *berak* művelet előfeltételének, kézenfekvő. A *berak* művelet pontosan akkor hajtható végre, ha a verem nincs tele, azaz a művelet előfeltétele: **not** *tele*. Hasonlóképpen az elem lekérdezés előfeltétele: a **not** *ures* predikátum. Emlékezzünk vissza, hogy mindkét előfeltétel tulajdonképpen egy Eiffel nyelven megadott jól formált formula.

Utófeltétel

Szeretnénk azonban azt is, ha a művelet végrehajtásának eredményét is le tudnánk írni. Ezt az előfeltételekhez hasonlóan fogalmazhatjuk meg. A művelet végrehajtásakor megfigyelhetjük, hogy a behelyezett elem éppen a verem tetején helyezkedik el. Ezeket a művelet helyes eredményét meghatározó feltételeket a művelet utófeltételeinek nevezzük. Tegyük fel, hogy a *berak* művelet az *uj_elem* paraméterben kapja meg a verembe helyezendő új elemet. Ekkor a művelet utófeltétele lehet a következő formula: *verem.elem=uj_elem*.

Hoare-formulák

Általánosságban egy adott M művelet esetében a $\{E\}M\{U\}$ jelölést alkalmazzuk, ahol az E és U predikátumok rendre az M művelet elő- és utófeltételét jelölik. Az $\{E\}M\{U\}$ hármast Hoare-formulának nevezzük. Maga a formula pontosan akkor igaz, ha az E predikátum igaz az M művelet végrehajtása előtt és ha az M műveletet végrehajtjuk, akkor az befejeződik és a végrehajtása után az U predikátum igaz lesz. Megjegyezzük, hogy a Hoare-formula az előbbi értelmezés esetén úgynevezett teljes helyességet ír le. Létezik ennél gyengébb definíció is, mely nem követeli meg az M művelet befejezését. Ilyen esetekben részleges helyességről beszélünk.

A fejezetben mi a programhelyességgel, illetve annak ellenőrzésével nem elméleti, hanem gyakorlati szempontból foglalkozunk. Számunkra fontos, hogy a programok futása befejeződjön, így mi csak teljes helyességgel foglalkozunk. A továbbiakban helyesség alatt mindig teljes helyességet értünk.

Az $\{E\}M\{U\}$ Hoare-formula elő- és utófeltételéből álló $\langle E, U\rangle$ rendezett párost az M művelet helyességi specifikációjának is nevezzük. Az $x := x \cdot 2$ művelethez tartozhat például az $\langle x > 4, x > 7\rangle$ vagy a $\langle x > 4, x > 8\rangle$ helyességi specifikáció. Az $\langle x > 4, x > 7\rangle$ a művelettel együtt a $\{x > 4\}x := x \cdot 2\{x > 7\}$ Hoare-formulát adja. A formula jelentése: ha x változó értéke nagyobb, mint négy, akkor az $x := x \cdot 2$ művelet végrehajtása után biztos, hogy x változó értéke nagyobb lesz, mint hét.

A *berak* művelet esetében egy egyszerűsített Hoare-formula lehet például a következő:

$$\{\, \neg verem.tele \,\}\ \ verem.berak(uj_elem)\ \ \{\, verem.elem = uj_elem \,\}$$

Ennek jelentése: ha a verem nincs tele, akkor a *berak* művelet befejeződik és a végrehajtás után az új elem a verem tetején helyezkedik el. Emlékezzünk vissza, hogy az *elem* lekérdezés a verembe utoljára helyezett elemet adja vissza.

Vegyük észre, hogy ha az elő- és utófeltételeket nem használnánk, akkor nem lehetnénk biztosak benne, hogy a behelyezni kívánt elem valóban a verembe került, ráadásul a verem tetejére. Elképzelhető lenne ugyanis olyan megoldás, melynek *berak* eljárása ellenőrzi, hogy a verem tele van-e. Ha nem, akkor az új elemet elhelyezi, ha igen, akkor nem csinál semmit. Ilyen esetekben a *VEREM* osztályt használó programozó, mivel nem kap semmilyen visszajelzést, abban a hiszemben lenne, hogy az átadott elem a verembe került. Láthatjuk tehát, hogy az előfeltételek ellenőrzése milyen jelentőséggel bír. Hasonlóan, ha megvizsgáljuk a *berak* eljárás fent megadott utófeltételét, szembetűnik, hogy az eljárás garantálja, hogy engedélyezett meghívás esetén (amely azt jelenti, hogy a verem nincs tele) teljesíti a feladatát: az új elem valóban a verembe került, méghozzá pontosan a verem tetejére.

Elő- és utófeltételek az Eiffelben

Vizsgáljuk meg, hogy a fent megadott elő- és utófeltételeket hogyan kell megadnunk az Eiffelben. Az előfeltételeket a **require** (követel) kulcsszó vezeti be, míg az utófeltételek az **ensure** (biztosít). Sem az elő-, sem az utófeltételek megadása nem kötelező és amennyiben nem adjuk meg őket, úgy alapértékük a *True* logikai formula lesz.

```
berak(e : G) is
require
        nincs_tele: not tele
do
        elemszam := elemszam + 1
        tomb.put(e, elemszam)
ensure
        uj_elem_a_tetejen: elem = e
        nem_ures: not ures
        ...
end
```

Logikai állítások csoportja

Vegyük észre, hogy a *berak* művelet utófeltételében két logikai állítás is szerepelt. Egyetlen logikai állítás ugyanis nem minden esetben elegendő a programrész működését leíró feltétel megfogalmazására. Bár egy logikai állítás tartalmazhat több logikai kifejezést is a megfelelő logikai műveletekkel összekapcsolva (**or**, **and**, **and then**, **or else** stb.), ez az összekapcsolás nem minden esetben célravezető. A *berak* művelet utófeltételét megadhattuk volna az alábbi alakban is:

utofeltetel: elem = e **and not** *ures* **and** ...

Az ilyen összevont megadás azonban nem kielégítő. Ennek elsődleges oka az, hogy egy logikai állításhoz csak egy megnevezés társítható. A *berak* eljárás eredeti utófeltételében a két feltétel megfogalmazása jól szembetűnik, míg a második esetben összemosódik. Ezért az Eiffel nyelv lehetőséget biztosít a logikai állítások blokkba, csoportba való rendezésére.

Logikai állítások csoportján egy vagy több logikai állításnak a forráskódban szereplő fizikai sorrend alapján történő felsorolását értjük. A logikai állításokat, bár ez nem jelölt, az **and then** logikai művelet kapcsolja össze.

Old

Egy eljárás működésének leírása során gyakran szükségünk lehet egy változó korábbi értékére való hivatkozására. Az új érték és a korábbi érték ismeretében tudjuk ugyanis a változást leírni. Ehhez nyújt segítséget az **old** szerkezet. Az **old** konstrukció kizárólag utófeltételben alkalmazható. Az **old** operátor mögött globális változó állhat. A szerkezet hatásköre az adott eljárásra terjed ki és a mögötte álló változó az eljárás lefutása előtti értékére hivatkozik.

Emlékezzünk vissza, hogy az *elemszam* a veremben éppen tárolt elemek számát jelöli. A *torol* művelet utófeltétele az alábbiak szerint adható meg:

```
-- Aktuális elem törlése.
torol is
require
        nem_ures: not ures
do
        elemszam := elemszam - 1
ensure
        elemszam_csokkent:
           elemszam = old elemszam - 1
        ures_ha_egy_elem_volt:
           old elemszam = 1 implies ures
        nincs_tele:
           not tele
end
```

Az *elemszam* = **old** *elemszam* − *1* kifejezésben tehát az **old** *elemszam* az *elemszam* változónak az eljárás lefutása előtti értékére hivatkozik. A teljes kifejezés azt jelenti, hogy az elem törlésével a verem elemszáma pontosan eggyel csökkent. Megjegyezzük, hogy az **old** operátor a legmagasabb kiértékelési szinttel rendelkezik, azaz az **old** *elemszam* = *1* **implies** *ures* kifejezés egyenértékű az *(***old** *elemszam) = 1* **implies** *ures* kifejezéssel.

Strip

Láttuk, hogy az **old** szerkezet segítségével lehetőségünk van objektumok attribútumainak változását leírni. Gyakran szükség lehet annak a megadására is, hogy bizonyos globális változók értékei nem változtak a program futása során.

A *fordit* művelet módosítja ugyan az elemek sorrendjét, de nincs hatással a verem ábrázolására, így az elemszámra sem. A *fordit* eljárás utófeltételében ez a következőképpen adható meg:

```
tobbi_valtozatlan:  old elemszam = elemszam
                    and old kapacitas = kapacitas
                    and ...
```

Az **and** logikai művelet mögött felsorolható az összes olyan attribútum, ami nem változhat. Amennyiben a *VEREM* osztálynak számos olyan attribútuma van, ami nem változik, úgy ezeknek felsorolása és **and**-del való összekapcsolása aprólékos megoldás. Ráadásul ha az osztály új attribútummal bővül, az összes utófeltételt ki lehet egészíteni az új attribútumra vonatkozó logikai állítással.

A **strip** szerkezet alkalmazása ehhez nyújt segítséget. A **strip** az **old** szerkezettel együtt az utófeltételben alkalmazható. A **strip** operátor után zárójelben az osztály attribútumait lehet felsorolni. A **strip** operátor egy *ARRAY [ANY]* típusú tömböt határoz meg, melyben az adott osztály attribútum-halmazának komplementere szerepel. Azaz a tömb pontosan azokat a globális változókat tartalmazza, melyek az osztályhoz tartoznak, de nem szerepelnek a

zárójelben. Vegyük észre, hogy ha a **strip** kifejezésben a zárójelben semmilyen attribútumot nem soroltunk fel, azaz **strip**() kifejezést írunk, a tömb az osztály összes globális változóját tartalmazni fogja.

Emlékezzünk vissza, hogy a *VEREM* osztály elemeit egy *tomb* nevű globális változóban tároljuk.

A *VEREM* osztályunk esetében a **strip**(*tomb*) egy olyan tömböt definiál, mely tartalmazza az osztály minden globális változóját, kivéve a *tomb* változót. A *fordit* művelet működését ismerve nekünk pont erre van szükségünk. A *fordit* eljárás esetében utófeltétel a **strip** operátorral az alábbiak szerint adható meg:

tobbi_valtozatlan: *deep_equal*(**strip**(*tomb*), **old strip**(*tomb*))

Az **old** és **strip** szerkezetekkel képzett utófeltétel pontosan azt írja le, hogy az osztály semmilyen globális attribútuma nem változik, leszámítva azt, hogy az elemek sorrendje megváltozott.

Az utófeltételben alkalmazott *deep_equal* reláció objektumokon értelmezett egyenlőség reláció. Az objektumok szerkezetét rekurzívan hasonlítja össze. A sima egyenlőség reláció alkalmazása azért nem szerencsés, mert az az objektumoknál a hivatkozási értékek összehasonlítását jelenti.

Tervezési szempontok

A metódusok elő- és utófeltételeinek meghatározása során be kell tartanunk bizonyos szabályokat. Az első és talán legfontosabb szabály, hogy az előfeltételekben csak olyan logikai állítások szerepelhetnek, melyekben a használt változók, függvények (predikátumok) rendelkezésre állnak az eljárás hívója számára. Ennek oka az, hogy az eljárást használó ügyfelektől csak akkor követelhetjük meg az engedélyezett eljáráshívást, ha módjukban áll ellenőrizni annak előfeltételeit.

Ha a *VEREM* osztály *berak* műveletének előfeltételében szerepel a **not** *tele* predikátum, akkor ezt a predikátumot rendelkezésre kell bocsátanunk minden olyan osztály számára, melyek számára a *berak* eljárás is elérhető. Nem várható el ugyanis egy osztálytól a *berak* eljárás előfeltételeinek teljesítése és a helyes eljáráshívás, ha nem áll rendelkezésre számára a *tele* függvény.

Figyeljük meg, hogy nem követeljük meg ugyanezt az utófeltételekben szereplő egyedekre. Ez azért van így, mert az utófeltételek biztosítása a hívott eljárás feladata, így amennyiben a hívott eljárás rendben befejeződik, a hívó az utófeltételek teljesülésére nyugodtan építhet.

Megköveteljük azonban, hogy sem az elő-, sem az utófeltételekben nem szerepelhetnek olyan logikai állítások, melyek helyi változókra hivatkoznak. Előfeltételek esetén ez egyrészt következménye az előfeltétel rendelkezésre állási szabályának, másrészt az előfeltételek kiértékelése előtt a helyi változók legfeljebb csak alapértelmezett értékkel rendelkeznek, így használatuk nem túl lényeges.

Az utófeltételek esetében a megszorításnak elvi, programozástechnikai okai vannak. Egyrészt a műveletek az objektumokon okoznak állapotváltozást, az objektum állapotát pedig pontosan a globális változóinak együttese határozza meg. Ami számunkra egy művelet végrehajtása során érdekes, az kizárólag az, hogy az objektum milyen kezdőállapotból

milyen végállapotba jut, ehhez viszont nem szükséges helyi változókat figyelembe venni. Másrészt a helyi változók is az eljárás argumentumától, globális változóktól, esetleg más függvényhívások által kapnak értéket. Így ezek az értékek rendelkezésre állnak az utófeltételben is (ez bizonyos, mellékhatással rendelkező számításoknál nem mindig van így).

13.1.4. Osztályinvariáns

Érvényes objektumok

Korábban megadtuk a verem egy lehetséges ábrázolását. Tegyük fel, hogy a *tomb* nevű tömb az alábbi elemeket tartalmazza:

[1, 2, 3, 4, 5]

A *tomb* változó önmagában azonban nem írja le a verem objektum állapotát, hiszen az *elemszam* változótól függően a fenti tömb több vermet is ábrázolhat. Lehet akár üres, nulla elemű tömb, vagy akár teli tömb is öt elemmel. Sőt a verem állapotához szükséges a *kapacitas* változó ismerete is. A konkrét ábrázolás (a *VEREM* osztály megvalósítása) és az elvont fogalom (verem) között az úgynevezett reprezentációs függvény teremt kapcsolatot. A reprezentációs függvényről a típusok vonatkozásában az 5.1.2. alfejezetben találhat bővebb információt az olvasó. A reprezentációs függvény képez le a konkrét megvalósításról az absztrakt adattípusra. Esetünkben a reprezentációs függvény az alábbiak szerint adható meg:

$$rep : tomb \times elemszam \times kapacitas \mapsto VEREM$$

A konkrét ábrázolás és reprezentációs függvény önmagában nem elégséges a verem egy adott állapotának leírására. Pontosabban nem minden reprezentáció határoz meg érvényes verem objektumot, illetve absztrakt adattípust. Emlékezzünk vissza, hogy az *elemszam* változóban tároltuk a veremben éppen tárolt elemek számát, míg a *kapacitas* változó adott felső határt a verembe helyezhető elemek számára. Ha például az *elemszam* változó értéke 5 és a kapacitás 4, akkor az ábrázolás nehezen értelmezhető. Hasonlóképpen értelmezhetetlen egy negatív kapacitással rendelkező verem objektum.

Ilyen jellegű problémával már találkoztunk a műveletek működésének elemzése során. Vegyük észre, hogy a probléma jelen esetben is az, hogy a megadott reprezentációs függvényünk parciális függvény. Most is ugyanazt a trükköt alkalmazzuk, mint a metódusoknál: megpróbáljuk a függvényt totális függvénnyé alakítani.

Az ábrázolás értelmezésén kívül tehát szükségünk lesz olyan összefüggések megadására is, amelyek biztosítják, hogy a reprezentációs függvény totális legyen, ami egyben azt is jelenti, hogy az adott objektum érvényes. Más szóval: ne fordulhasson elő a fenti példák egyike sem. Ezeket a globális összefüggéseket egy absztrakt adattípushoz tartozó konkrét előfordulás esetén predikátumokkal jellemezhetjük. A mi esetünkben a következő észrevételeket tehetjük:

- A verem kapacitása pozitív egész szám kell, hogy legyen.
- A verem elemszáma 0 és a kapacitás közötti egész szám kell, hogy legyen (emlékezzünk vissza, hogy a nulla elemszám az üres vermet jelenti).

Az absztrakt adattípus ezen összefüggéseit, melyet axiómáknak is nevezünk, az Eiffelben a következőképpen adhatók meg:

```
invariant
  ervenyes_kapacitas:    kapacitas > 0
  ervenyes_elemszam:     elemszam >= 0
                    and elemszam <= kapacitas
  ...
end -- class VEREM
```

Az absztrakt adattípus tulajdonságait leíró predikátumokat az **invariant** kulcsszó vezeti be. Ezeket közösen osztályinvariánsnak nevezzük. Az invariáns kifejezés onnan ered, hogy olyan feltételeket határoz meg, melyek az osztály példányainak életciklusa alatt minden kitüntetett időpillanatban a példányokon végzett műveletek során megőrzik igaz értéküket. Más szóval ezek olyan logikai kifejezések, melyek mindegyike *True* logikai értékkel rendelkezik és változásokra érzéketlen.

Egy invariáns több logikai állításból áll össze, az osztályinvariánst képező logikai állítások pedig az **and then** logikai operátorral vannak összekapcsolva a forráskódban meghatározott fizikai sorrend szerint.

Vegyük észre, hogy az invariánsok egy adott ábrázolási séma esetében (*tomb*, *elemszam*, *kapacitas*) egyértelműen meghatározzák, hogy az adott reprezentáció melyik konkrét előfordulása megengedett. Úgy is fogalmazhatunk, hogy az osztályinvariánsok karakterizálják az adott ábrázolási séma példányait. Az *ervenyes_elemszam* predikátum például azt fogalmazza meg, hogy egy érvényes verem objektum esetében a verem elemszáma mindig nulla és a kapacitás között van. Ez az összefüggés a verem objektumon végzett bármilyen művelet tetszőleges sorrendben való végrehajtása során teljesülni fog.

Az osztályinvariánsok nemcsak az osztály attribútumaira adhatnak megszorítást, illetve azok között teremthetnek kapcsolatot. Segítségükkel lehetőségünk van az attribútumok és függvények közötti relációk megfogalmazására is. A *VEREM* osztály invariánsa bővíthető az alábbi logikai állításokkal:

```
ures_definicio: (ures implies elemszam = 0)
                  and (elemszam = 0 implies ures)
tele_definicio: (tele implies elemszam = kapacitas)
                  and (elemszam = kapacitas implies tele)
```

A *tele_definicio* például a *tele* metódus és az ábrázolás között teremt összefüggést. Egész pontosan azt fogalmazza meg, hogy a verem akkor és csak akkor van tele, ha az *elemszam* változó értéke megegyezik a *kapacitas* változó értékével.

Az osztályinvariánsok kiértékelése

A fentiekben használtuk a „kitüntetett időpillanat" kifejezést, mely alatt a következőket értjük. Az osztályinvariánsoktól nem követeljük és nem is követelhetjük meg, hogy minden időpillanatban igazak legyenek. Ennek oka az, hogy egy-egy bonyolultabb eljárás működése során előállhat a globális attribútumoknak olyan pillanatnyi kombinációja, amely mellett az osztályinvariánst kiértékelve az eredmény *False* lenne.

Később az osztályok helyességének meghatározásakor ismertetjük az osztály konstruktor metódusát is. Érdemes megfigyelni, hogy a konstruktor metódus futása során nem, csak a teljes konstruktor metódus befejezése után teljesülnek az osztályinvariánsok. Ez természetes, hiszen amint látni fogjuk, a konstruktor egyik alapvető feladata pontosan az objektum kezdeti értékeinek beállítása az osztályinvariánsok figyelembe vételével.

Az osztályinvariánsok kiértékelésére csak nyilvános eljárások meghívása előtt és után kerül sor, viszont megköveteljük, hogy ezekben a pillanatokban az osztályinvariáns igaz legyen. Ezeket a pillanatokat neveztük korábban kitüntetett időpillanatoknak. Azt, hogy a privát eljárások végrehajtása előtt és után nem követeljük meg az invariánsok fennállását, annak a következő okai vannak:

Az objektum egy absztrakt adattípus konkrét példánya. A külvilág számára a szolgáltatásai csak a nyilvános felületen keresztül érhetők el. Egy-egy eljárás belsejében megengedjük, hogy ideiglenesen megsértsük az invariánst. A privát eljárások tulajdonképpen a nyilvános metódusok számítási algoritmusának részét, kiterjesztését képezik, csak bizonyos szoftvertervezési megfontolások alapján kerültek dedikált eljárásokba. Mivel ezek az eljárások amúgy sem érhetőek el a külvilág számára és meghívásuk minden esetben egy nyilvános metóduson keresztül történik, nem kell, hogy megköveteljük az ilyen eljárásokban az osztályinvariáns fenntartását.

13.1.5. Ciklusok

Általános problémák

A rekurzív eljáráshívások mellett kétségtelenül a ciklusok okozzák a legtöbb fejtörést a programozóknak. A ciklusok írása során az alábbi jellemző hibák szoktak előfordulni:

- Helytelen inicializálás.
- Ciklus 1 ismétléssel kevesebb vagy több végrehajtása.
- Kicímzés a ciklusban használt struktúrából. A megengedettnél nagyobb vagy kisebb indexérték használata.
- „Végtelen" ciklus.

A fordit művelet

VEREM osztályunk *fordit* eljárásának alapötlete a következő. Az eljárás mindig két elemet fog felcserélni. Először a legalsó és legfelső elemet, majd a korábbi legalsó elem fölötti és a korábbi legfelső alatti elemet és így tovább. Ezt a cserét egészen addig ismételjük, amíg minden elemet fel nem cseréltünk. Ez akkor teljesül, amikor a felcserélendő két elem indexe ugyanaz, vagy szomszédos.

Gondoljuk át *VEREM* osztályunk *fordit* metódusának ciklusát. Ennél a ciklusnál is a fentihez hasonló problémák jelentkezhetnek. Előfordulhat, hogy a ciklus miatt a *fordit* metódus nem, vagy nem szabályosan fejeződik be. Lehet, hogy az eljárás terminál, de az elemek helytelenül cserélődnek fel. Előfordulhat az is, hogy az elemek nagy része felcserélődik, de az utolsó két elem nem (vagy a ciklus kihagyja az utolsó két elemet, vagy duplán cseréli meg). Látható, hogy egy ilyen egyszerű művelet, mint az elemek sorrendjének megfordítása is milyen sok problémát vet fel és mennyi körültekintést igényel.

Szerencsére az Eiffel nyelv ciklusszerkezete segítséget nyújt a ciklusok pontos és biztonságos megadására, hogy a fenti hibák ne fordulhassanak elő.

A *fordit* eljárás – leszámítva az elő- és utófeltételeket és változó-deklarációkat – csak magából a ciklusból áll. A következő oldal példaprogramja alapján vegyük szemügyre az Eiffel ciklusszerkezetét.

A ciklusban az alábbi változókat használjuk:

- Az *also_poz* és *felso_poz* a verem fel nem cserélt elemeinek alsó és felső indexét jelölik.
- Az *also_csere* és *felso_csere* változók a cserélt alsó és felső elemek indexét tárolják.
- Az *also_elem* és *felso_elem* a csere előtti alsó és felső elemet reprezentálják.

Az inicializáló blokk a **from** kulcsszó után írható. Ez állítja be a ciklus változóinak kezdeti értékét. Ez a mi esetünkben az alsó és felső pozíciók megadását jelenti.

Az **invariant** kulcsszó már az osztályok vizsgálatánál felmerült. A ciklusok esetében az invariánst ciklusinvariánsnak nevezzük, használata nem kötelező. Ez a ciklus azon tulajdonságait írja le, melyek a ciklus futása alatt változatlanok maradnak. A ciklusinvariáns is jól formált formulák halmaza. Nézzük, mit is jelent ez a mi esetünkben.

A felcserélendő elemeket két index azonosítja. Az *also_csere* az alsó cserélendő elem indexét adja meg, míg a *felso_csere* a felsőét. Az algoritmus alapján a cserélt elem alsó indexe minden pillanatban kisebb kell, hogy legyen, mint a cserélt felső elem indexe. Azt is tudjuk, hogy a kezdő alsó index értéke 1 és kezdő felső index értéke megegyezik az elemszámmal. Továbbá, mivel az egyik index csökken, a másik pedig nő, a kettő összegének bármely időpillanatban ki kell adnia az elemszámot (illetve eggyel nagyobb értéket, mivel az elemeket 1-től kezdtük számozni). A felső és alsó csereindexekre rendre az 1 (az első elem indexe) és az *elemszam* ad korlátozást.

A fenti invariáns megadásával biztosítottuk, hogy a cserélendő elemek valóban az alsó és felső tartományból kerülnek ki. Biztosítottuk azt is, hogy az alsó és felső cserepozíciót megadó indexek szerepe szimmetrikus. Nem fordulhat elő olyan eset, hogy az alsó index nőtt, a felső pedig nem, hiszen az indexek összege minden időpillanatban állandó. Tehát az alsó index csak akkor és annyival nőhet, amikor és amennyivel a felső index csökken. Megfogalmaztuk azt a feltételt is, hogy az indexek valóban csak a veremben elhelyezett elemekre mutathatnak.

Figyeljük meg, hogy a *fordit* eljárásban kihasználjuk az előfeltételt, miszerint a verem nem lehet üres. A ciklusinvariáns ugyanis üres verem esetén nem teljesülne, hiszen a *felso_poz* értéke 0 lenne, ami kisebb, mint a kezdő alsó pozíció.

Láthatjuk, hogy a ciklusinvariánsban megfogalmazott formulák a ciklus számos lényeges tulajdonságát adják meg. Segítségükkel a korábban felsorolt programozási hibák nagy része elkerülhető. Vegyük észre azonban, hogy az még nem biztosított, hogy minden elem felcserélődik, sőt a ciklus befejezéséről sem mondtunk még semmit. Vizsgáljuk meg először az Eiffel ciklusok terminálását.

A **variant** kifejezés éppen ehhez nyújt segítséget. A kulcsszó után egy egész típusú kifejezés adható meg. A kifejezéstől két dolgot követelünk meg:

- Értéke a ciklusmag lefutása előtt és után mindig pozitív egész legyen.
- Értéke folyamatosan minden ciklusmag lefutása után szigorúan csökkenjen.

```
-- Felcséreli a verem elemeinek sorrendjét.
fordit is
require
   nem_ures: not ures
local
   also_poz    : INTEGER
   felso_poz   : INTEGER
   also_elem   : G
   felso_elem  : G
   also_csere  : INTEGER
   felso_csere : INTEGER
do
   from
      also_poz    := 1
      felso_poz   := elemszam
      also_csere  := also_poz
      felso_csere := felso_poz
   invariant
      ervenyes_poziciok:
         also_csere + felso_csere = elemszam + 1
         and also_csere >= 1
         and also_csere <= felso_csere
         and felso_csere <= elemszam
   variant
      fennmarado_intervallum: felso_poz - also_poz + 1
   until (felso_poz - also_poz) < 1
   loop
      also_csere  := also_poz
      felso_csere := felso_poz
      also_elem   := tomb.item(also_csere)
      felso_elem  := tomb.item(felso_csere)
      tomb.put(felso_elem, also_csere)
      tomb.put(also_elem, felso_csere)
      check
         elemcsere:
            also_elem = tomb.item(felso_csere)
            and felso_elem = tomb.item(also_csere)
      end
      also_poz  := also_poz + 1
      felso_poz := felso_poz - 1
   end
ensure
   elemek_felcserelve:
      old tomb.item(1) = tomb.item(elemszam)
      and old tomb.item(elemszam) = tomb.item(1)
   tobbi_valtozatlan:
      deep_equal( strip(tomb), old strip(tomb) )
end
```

Belátható, hogy a fenti két tulajdonság elegendő a ciklus befejezéséhez, ugyanis a kifejezés értéke bármilyen nagy pozitív egész szám is, ha a ciklus során biztosított, hogy értéke folyamatosan és szigorúan csökken, akkor a csökkenés nem tarthat a végtelenségig. Más szavakkal a ciklus futása előbb vagy utóbb befejeződik.

A programhelyesség-bizonyítás témakörében jártasak bizonyára észrevették, hogy az Eiffel ciklusvariáns kifejezése tulajdonképpen a termináló függvény egy speciális esete, a termináló függvény értékkészlete pedig az $(N, <)$ rendezéssel megadott jól megalapozott halmaz.

A *fordit* eljárás esetében a ciklusvariánsunk az alábbi kifejezés:

variant
fennmarado_intervallum: felso_poz − also_poz + 1

A ciklus algoritmusát átgondolva megállapíthatjuk, hogy a fent megadott kifejezés maradéktalanul megfelel az elvárásoknak. Egyrészt mivel a verem nem lehet üres, a kezdeti felső és alsó pozíció különbsége pozitív egész. Másrészt a csere működéséből azt is tudjuk, hogy az alsó index folyamatosan nő, míg a felső csökken, így a ciklusvariáns minden lefutás során pontosan 2-vel csökken.

Természetesen választhattunk volna más kifejezést is, például a *felso_poz* változó, mint egész típusú kifejezés, önmagában is megfelelt volna, de ennek használata nem olyan szemléletes. A *felso_poz − also_poz + 1* kifejezés éppen a fennmaradó intervallumot reprezentálja, azt a tartományt, amelynek elemeit még nem cseréltük fel.

A kilépési feltételt az **until** kulcsszó után kell megadni. Felhívjuk a figyelmet, hogy sok más programozási nyelvvel ellentétben az Eiffelben a ciklusmag addig ismétlődik, amíg a kilépési feltétel hamis. Ez a mi esetünkben azt jelenti, hogy a ciklus akkor fejeződik be, amikor a fennmaradó tartomány üres lesz, más szavakkal az alsó és felső pozíció elérte vagy elhagyta egymást.

Első ránézésre ijesztőnek és feleslegesen bonyolultnak tűnik a teljes Eiffel ciklusszerkezet. Ne feledkezzünk meg azonban arról, hogy a ciklus definíciójában csak a kilépési feltétel kötelező, az összes többi szerkezeti elem elhagyható. Alkalmazásukat azonban mindenképpen javasoljuk. A ciklusinvariánssal és -variánssal kibővített ciklus szemléletesebb, jobban érthető. A megfelelő ciklusinvariáns és termináló függvény megalkotása türelmet és gyakorlást igényel, ugyanakkor elősegíti a probléma jobb átlátását. Ráadásul a teljes ciklusszerkezet alkalmazásakor a megfelelő futtatási kapcsolókat használva a futtató környezet jelzi számunkra, ha a ciklus működése nincs összhangban a ciklusra vonatkozó helyességi specifikációval. Így elkerülhetjük és kiküszöbölhetjük a bekezdés elején ismertetett jellemző problémákat.

A figyelmes olvasónak bizonyosan feltűnt, hogy a ciklus megadása során az *also_poz* és *felso_poz* változókon kívül használtuk még az *also_csere* és *felso_csere* változókat is. Ennek a viszonylagos bonyolultságnak az oka az, hogy az Eiffel ciklus kilépési feltételének ellenőrzésére a ciklusmag végrehajtása előtt kerül sor. A kilépési feltétel páros elemszám esetén éppen az, ha az alsó és felső pozíció átlépi egymást. Például egy négy elemű verem esetében az alsó pozíció 3 lesz, míg a felső 2. Emiatt az átlépés miatt azonban a ciklusinvariáns már nem lenne érvényes, ha ott is az *also_poz* és *felso_poz* változók szerepelnének. A két változó érté-

kének változása a ciklusvariáns kifejezés miatt is kívánatos, hiszen a ciklusszerkezetet leíró, fent ismertetett formulák megkövetelik a ciklusvariáns csökkenését minden esetben, még kilépéskor is. Vegyük észre azonban, hogy az *also_poz* és *felso_poz* változó csak technikai szereppel bír. A ciklus jelentését, az elemek cseréjét éppen az *also_csere* és *felso_csere* változók írják le, mégpedig szemléletesen és helyesen.

Azt, hogy minden elem felcserélődjön a ciklus kilépési feltétele biztosítja. A ciklus ugyanis pontosan akkor terminál, ha az *also_poz* és *felso_poz* változók azonos vagy szomszédos elemeket határoznak meg.

13.1.6. A check utasítás

Láttuk, hogy az Eiffel osztályaihoz osztályinvariáns adható meg. A metódusok elő- és utófeltétellel rendelkezhetnek. A ciklusok helyességének ellenőrzésekor segítséget nyújt a ciklusinvariáns és a terminálló függvény. A nyelv által támogatott további lehetőség, hogy egy metódus törzsében tetszőleges ponton elhelyezhetünk logikai állításokat. A logikai állítások megadására a **check** utasítás ad lehetőséget. A figyelmes olvasó észrevehette, hogy a *fordit* eljárás ciklus törzsében hogyan alkalmaztuk a **check** szerkezetet.

```
check
   elemcsere: also_elem = tomb.item(felso_csere)
              and felso_elem = tomb.item(also_csere)
end
```

A példa kissé erőltetett, mert az elemek cseréje egyszerű művelet. Összetettebb számításoknál azonban a **check** szerkezet jól alkalmazható.

A **check** utasításokkal nemtriviális feltételezéseket fogalmazhatunk meg és dokumentálhatunk. Természetesen a **check** utasítás nemcsak ciklusban fordulhat elő, hanem az eljárás törzsében bárhol. Ezek a feltételezések az eljárás előfeltételéhez hasonló feltételek, melyekre az eljárás működése során támaszkodunk. Például ha a metódus működése alatt biztosak vagyunk benne, hogy egy változó, például a *diszkriminans* nem vehet fel negatív értéket, akkor ezt az alábbiak szerint adhatjuk meg:

```
check
  pozitiv_diszkriminans: diszkriminans > 0
  -- mert a > 0 es c < 0 vagy fordítva
end

x1 := (-b + sqrt(diszkriminans)) / 2 * a
x2 := (-b - sqrt(diszkriminans)) / 2 * a
```

A logikai állítás után már a valós gyökök kiszámítása során alkalmazhatjuk az ismert formulát, mivel a program algoritmusát átgondolva arra a következtetésre jutottunk, hogy a diszkrimináns nem lehet negatív és ezt a következtetést logikai állítás formájában rögzítettük is.

Ismételten felhívjuk a figyelmet, hogy a **check** utasítás esetében, mint a többi helyességi specifikáció részét képező nyelvi szerkezeteknél, sem arról van szó, hogy ellenőrizzük,

hogy a diszkrimináns változó értéke pozitív-e, és ha igen, akkor kiszámoljuk a gyököket, egyébként pedig valami mást csinálunk. A **check** utasítás után szereplő logikai állítás egy hipotézis, melynek igaznak kell lennie a program futása során. Amennyiben ez nem teljesül, úgy a program nincs összhangban a specifikációval. A fenti példában a másodfokú egyenlet gyökeinek számítása során tudjuk, hogy az egyenlet a és c paraméterének előjele ellentétes, így nyugodtan alkalmazható a fenti logikai állítás. Ebben az esetben ugyanis a $diszkriminans = b^2 - 4ac$ biztos, hogy pozitív lesz.

A logikai állítások **check** formában való megfogalmazása több előnnyel is jár.

- A **check** utasításban szereplő állítások egyfajta dokumentációt biztosítanak a program megértéséhez.
- A fenti példában a hipotézis megfogalmazása után nem szükséges **if–then–else** ágakkal megvizsgálni, mi történik akkor, ha a diszkrimináns negatív. Így a program elegánsabb, jobban áttekinthető lesz.
- A hipotézis esetleges tévessége a program futása során kiderül(het).

13.1.7. Öröklődés

Az öröklődési mechanizmus tükrében számos érdekes kérdés merül fel a korábban ismertetett fogalmakkal kapcsolatosan. A leírtak alapvetően érvényesek és helyesek, de szükségünk van értelmezésük kibővítésére.

Metódusok átdefiniálása

Az öröklődés során lehetőségünk nyílik (sok egyéb lehetőség mellett) a metódusok átdefiniálására is. Maga az átdefiniálás többféleképpen is történhet (**redefine**, **rename**, **undefine**), de mi most csak az eredménnyel foglalkozunk, nevezetesen, hogy az öröklődés során képesek vagyunk (bizonyos megszorítások mellett) metódusokat átdefiniálni.

Vizsgáljuk meg a kérdést a típus-helyettesíthetőség szemszögéből. Megjegyezzük, hogy az Eiffelben az öröklődési reláció egyúttal típusmegfelelőséget is definiál. Ha C osztály B osztálytól örököl és c a C osztály egy példánya, akkor c használható minden olyan környezetben, ahol B osztály egy példányát feltételezzük.

Érdekes átgondolni, hogy mi történjen ilyen esetekben az elő- és utófeltételekkel. Tegyünk egy gondolatkísérletet. Ha az új osztály metódusának előfeltétele erősebb, szigorúbb, mint az ősosztályé, akkor a c objektum nem lenne használható a B példányai helyett, ugyanis ekkor c olyan feltételekre is építene, amit az ügyfél nem biztosít, hiszen az a B osztály példányát feltételezi. Ha ugyanazt vagy gyengébb előfeltételt választunk, akkor viszont a helyettesítés megengedett, hiszen a B osztály metódusának előfeltételei biztosítják a C osztály átdefiniált metódusának előfeltételeit is.

Hasonlóan érvelhetünk az utófeltétel esetében is. Ha a C osztály átdefiniált metódusának utófeltétele gyengébb lenne, akkor a c példányt használó ügyfél számára nem lennének biztosítottak a B osztály példányától elvárt utófeltételek. Ha azonban az átdefiniált eljárás utófeltétele ugyanaz vagy erősebb, akkor az ügyfelek alapozhatnak a B osztály eljárásának utófeltételére, hiszen a C osztály eljárásának utófeltétele biztosítja annak teljesülését, sőt még akár többet is.

A fentiekben a „gyengébb" és „erősebb" jelzőket intuitíven használtuk. Mindjárt kiderül, hogy pontosan miről is van szó.

Az $\langle E', U' \rangle$ specifikációt $\langle E, U \rangle$ helyességi specifikáció alspecifikációjának nevezzük, akkor és csak akkor, ha $E \Rightarrow E'$ és $U' \Rightarrow U$. A típus-helyettesíthetőség egyik feltétele az, hogy a helyettesíteni kívánt típus metódusainak specifikációja rendre alspecifikációja legyen a helyettesítendő típus metódusainak. Figyeljük meg, hogy az alspecifikáció maradéktalanul megfelel a típus-helyettesíthetőség szempontjából. Ha ugyanis a C osztály metódus $\langle E', U' \rangle$ helyességi alspecifikációval rendelkezik, akkor az az ügyfél, mely biztosítja az E előfeltételt, egyben biztosítja E'-t is, hiszen a definíció szerint $E \Rightarrow E'$. Hasonlóan, ha az ügyfél U utófeltételre épít, akkor U' utófeltétel megfelelő, mert $U' \Rightarrow U$.

A korábbi „terminológiánk" értelmében a fenti definícióban az E formula „erősebb", mint E', mert minden esetben, ha E igaz, akkor igaz E' is. Ugyanakkor az U formulát „gyengébb"-nek nevezzük U'-nél. Szerencsére viszonylag egyszerűen tudunk egy adott specifikációhoz alspecifikációt megadni. Könnyen belátható, hogy tetszőleges $\langle E, U \rangle$ helyességi specifikációra és F, T predikátumokra igaz, hogy $\langle E \vee F, U \wedge T \rangle$ alspecifikációja $\langle E, U \rangle$-nak.

A fentieket figyelembe véve az Eiffel nyelv az alábbi megszorítást teszi az átdefiniált eljárások helyességi specifikációjára. Az átdefiniált eljárások specifikációjában a **require** és **ensure** kulcsszavak helyett rendre **require else** és **ensure then** kulcsszavak használhatók. Legyen E és U rendre az átdefiniálandó metódus elő- és utófeltétele, jelölje E' és U' az átdefiniált metódus elő- és utófeltételét. Ekkor a metódus teljes elő- és utófeltétele a következő:

- Ha a **require else** kulcsszó hiányzik – vagy jelen van, de mögötte nem szerepel logikai kifejezés – akkor az előfeltétel E.
- Ha az **ensure then** kulcsszó hiányzik – vagy jelen van, de mögötte nem szerepel logikai kifejezés – akkor az utófeltétel U.
- Ha a **require else** kulcsszó jelen van és a mögötte szereplő előfeltétel E', akkor a teljes előfeltétel E **or else** E'.
- Ha az **ensure then** kulcsszó jelen van és a mögötte szereplő utófeltétel U', akkor a teljes utófeltétel U **and then** U'.

A továbbiakban egy eljárás elő- és utófeltételét a fenti értelemben használjuk.

Osztályinvariáns

Vegyük szemügyre most az osztályinvariánst. Tudjuk, hogy az Eiffelben az öröklődés kettős szerepet játszik. Egyrészt az öröklődés során lehetőségünk van az ősosztályok megvalósításának felhasználására, másrészt az öröklődés típus-helyettesíthetőséget is jelent.

Ha a C osztály a B osztálytól örököl, akkor a C osztály B egy speciális változatának tekinthető. Ha az öröklődést megfelelően alkalmazzuk, akkor a C osztályra természetesen helytállóak mindazok az osztályinvariánsban megfogalmazott feltételek, melyeket a B osztályban adtunk meg. Ugyanakkor a C osztálynak a B osztály speciális eseteként egyéb tulajdonságai is lehetnek. Az öröklődés során tehát a C osztálynak tartalmaznia kell mindazokat az invariánsokat, melyek jelen voltak az ősosztályokban.

Ha megadunk egy új *VEREM* osztályt (legyen *VEREM2*), ami a *VEREM* osztálytól örököl és csak páros értékű kapacitással rendelkezhet, akkor az új osztály osztályinvariánsa az alábbiak szerint nézhet ki.

```
invariant
    ervenyes_kapacitas:
        kapacitas > 0
    ervenyes_elemszam:
        elemszam >= 0
        and elemszam <= kapacitas
    ures_definicio:
        (ures implies elemszam = 0)
        and (elemszam = 0 implies ures)
    tele_definicio:
        (tele implies elemszam = kapacitas)
        and (elemszam = kapacitas implies tele)
    paros_kapacitas:
        kapacitas \\ 2 = 0
end -- class VEREM2
```

Vegyük észre, hogy az osztályinvariánsban felsoroltuk a *VEREM* osztályinvariánsait és kibővítettük a *paros_kapacitas* logikai állítással.

Az ősosztály invariánsainak felsorolása többszörös öröklődés és bonyolult osztályhierarchia esetén nagy odafigyelést igényel. Szerencsére az ősosztályok invariánsai algoritmikusan származtathatók. Az Eiffel nyelv és fejlesztői környezet nem is követeli meg tőlünk, hogy az új osztálynál felsoroljuk az ősosztályok invariánsait. Ha egy *C* osztály örököl más osztályoktól, akkor a *C* osztály osztályinvariánsa automatikusan kibővül azoknak az osztályoknak az osztályinvariánsaival, melyektől örököl. Az osztályinvariánst kibővítve az ősosztályok invariánsaival együtt teljes osztályinvariánsnak nevezzük. Az új *VEREM2* osztály esetében is elegendő az új összefüggéseket felsorolni.

```
invariant
    paros_kapacitas: kapacitas \\ 2 = 0
end -- class VEREM2
```

A *VEREM2* osztály teljes osztályinvariánsának előállítása már az Eiffel-fordító feladata.

Osztályinvariáns, elő- és utófeltétel

Vizsgáljuk meg most az osztályinvariánsok, illetve az elő- és utófeltételek kapcsolatát. Az osztályinvariáns értelmezése az öröklődés során számos érdekes kérdést vet fel. Említettük, hogy az osztályinvariánsnak teljesülnie kell minden nyilvános metódus meghívása előtt és után. Tudjuk azt is, hogy egy osztály invariánsa automatikusan kibővül az ősosztályok invariánsaival.

Legyen adott egy B osztály I osztályinvariánssal. Legyen M B osztály egy nyilvános metódusa a $\langle E, U \rangle$ specifikációval. Mivel M publikus eljárás, meghívásakor teljesülnie kell az $I \wedge E$, végrehajtása után pedig az $I \wedge U$ feltételnek.

Első pillantásra az osztályinvariáns feleslegesnek tűnhet, hiszen megfelelő elő- és utófeltételek választásával nincs rá szükség, de ez két dolog miatt nincs így. Az első egy módszertani ok. Az osztályinvariáns az osztály globális viselkedését leíró formulák csoportja. Elkülönítése az elő- és utófeltételektől az alábbi előnyökkel jár:

- A globális viselkedés külön kezelhető, jól látható.
- Osztályinvariáns nélkül a globális attribútumok módosítása esetén az összes nyilvános metódus elő- és utófeltételét módosítani kellene.

A második egy gondolatmenet következménye. Látni fogjuk, hogy az elő- és utófeltételek szerepe elkülönül az osztályinvariánsokétól. Vizsgáljuk meg a helyzetet ismét a típushelyettesíthetőség és az öröklődés szempontjából.

Tegyük fel, hogy C osztály örököl B osztálytól. C osztály teljes invariánsa legyen $I \wedge J$, ahol J az osztályban megadott invariáns. C örökölje változatlan formában az M metódust B osztálytól. Ekkor M végrehajtásának feltétele a C osztályban $I \wedge J \wedge E$, az eljárás végrehajtása után pedig $I \wedge J \wedge U$ feltételnek kell teljesülnie. Vizsgáljuk meg az M eljárás meghívásának feltételét B és C osztályban. Látható, hogy a C osztályban a $I \wedge J \wedge E$ feltétel erősebb, mint a B osztályban (ahol az előfeltétel $I \wedge E$), ez viszont ellentmond a helyettesíthetőség feltételének. Az utófeltételnél ez nem jelent problémát, hiszen $I \wedge J \wedge U \Rightarrow I \wedge U$ magától értetődően teljesül.

Megjegyezzük, hogy metódus-átdefiniálás esetén sem különbözne lényegesen a helyzet a fentitől. Ekkor ugyanis az M eljárás elő- és utófeltétele rendre az E', U' formulák lennének, de ilyenkor az $\langle E, U \rangle$ és $\langle E', U' \rangle$ helyességi specifikációkra önmagukban teljesülne az alspecifikáció reláció.

A fenti okok miatt más nyelvek, például a Sather [Gom 97] az alábbi megközelítést alkalmazzák. Csak absztrakt osztályok szerepelhetnek az öröklődési relációban ősökként. Ezzel a megszorítással, bár a fenti probléma nem oldódik meg, a helyettesítés során nem lép fel ellentmondás, hisz az absztrakt osztályoknak nem lehetnek példányaik. További technikai megfontolás a Sather részéről, hogy az osztályinvariánsokat csak az utófeltételekhez kapcsolja, csak azokkal együtt értékeli ki.

Érdemes elgondolkozni az előző kijelentésen. Egy program futása során a feladatot az objektumokon végzett műveletek sorozatai oldják meg. Ha egy nyilvános eljárás futása után igaz az utófeltétel és igaz az osztályinvariáns is, akkor az objektumon végrehajtandó újabb eljárás az előfeltételén kívül nyugodtan építhet az osztályinvariánsra is.

Érvelésünk során látszólag ellentmondásra jutottunk. Kiindulásként azt a matematikai, logikai ellentmondást használtuk, miszerint az öröklődés során az osztályinvariánssal kibővített elő- és utófeltételekre nem teljesül az alspecifikáció reláció. Láttuk azt is, hogy az osztályinvariáns utófeltételbe való beépítése nem okozhat problémát. Az így képzett, módosított utófeltétel megfelel az alspecifikáció követelményeinek. Ugyanakkor az objektum működését megvizsgálva arra a következtetésre jutottunk, hogy ha egy metódus végrehajtása után fennáll az osztályinvariáns, akkor egy újabb metódus előfeltételében építhet erre a tényre.

A látszólagos ellentmondást a következő egyszerű észrevétel oszlatja el. Vegyük észre, hogy az osztály (objektum) az osztályinvariáns fenntartását, teljesülését nem a külvilágtól várja, hanem maga az osztály (objektum) felelős a fenntartásáért. A belső mechanizmusok biztosítják, hogy megfelelő paraméterek esetén az invariánsok megőrződnek, még akkor is, ha ezek erősebbek, mint az ősosztályé, és még akkor is, ha az ősosztály példánya helyett használtak egy adott környezetben. A fenti okok miatt sem tekinthetünk az osztályinvariánsokra úgy, mint közös elő- és utófeltételekre. Az osztályinvariánsok annál többre hivatottak, mint ahogyan korábban, a reprezentációs függvény tárgyalásakor is kiderült.

13.2. A programhelyesség definiálása az Eiffelben

A helyességi specifikációt alkotó nyelvi elemek birtokában most már képesek vagyunk az Eiffel nyelven írt programok helyességéről érdemben nyilatkozni. Az általunk megfogalmazott helyesség a teljes helyességet fogja jelenteni.

13.2.1. Az attribútumok helyessége

Az attribútumok helyességét önmagában nem vizsgáljuk. Minden attribútumot helyesnek feltételezünk. Ezt a következők miatt tehetjük meg:

- Minden attribútum egy változó vagy objektum; értékének meghatározása nem tartalmaz nyelvi szintű számítást, ezért minden attribútum értékének meghatározása befejeződik.
- Az attribútumok konzisztenciáját (az osztályra, más attribútumokra, vagy metódusokra nézve) az osztály szintjén megfogalmazott invariánsok adják meg. Ezeket később tárgyaljuk.

Megjegyezzük, hogy előfordulhat olyan eset is, amikor egy globális attribútum eljárására vagy további attribútumára hivatkozunk. Ez vagy egy adott M eljárás törzsében fordulhat elő, vagy egy osztályinvariánsban. Az első esetben természetesen az adott eljárás helyessége függ az attribútum osztályának helyességétől. Az attribútum osztályának helyességét azonban nem attribútumszinten, hanem az adott metódus helyességének definiálásakor követeljük meg. Vegyük észre, hogy a második eset az attribútumok konzisztenciájának általánosítása.

13.2.2. A metódusok helyessége

A következő részben jelölje adott M metódus esetén B a metódus törzsét, E, U rendre annak teljes elő- és utófeltételét, I pedig a teljes osztályinvariánst.

A ciklusok helyessége

A ciklus működéséről közölt ismereteket összefoglalva és rendszerezve egy Eiffel nyelven megadott ciklus helyes működése formálisan az alábbi Hoare-formulákkal adható meg [Mey 91]. A ciklus akkor és csak akkor helyes a helyességi specifikációra nézve, ha az összes

alábbi Hoare-formula érvényes. Vegyük észre, hogy a definícióban implicit megköveteljük azt is, hogy a ciklusmag befejeződjön. Emlékezzünk vissza, hogy a $\{E\}B\{U\}$ Hoare-formula pontosan akkor igaz, ha E előfeltétel igaz, B utasítás végrehajtása befejeződik és igaz lesz az U utófeltétel is.

- $\{REQ\}INIT\{INV\}$
- $\{REQ\}INIT\{VAR \geq 0\}$
- $\{INV \wedge \neg EXIT\}BODY\{INV\}$
- $\{INV \wedge \neg EXIT \wedge VAR = v\}BODY\{0 \leq VAR < v\}$

Itt az INIT a ciklus inicializáló blokkja, a BODY a ciklusmag, az INV jelöli a ciklusinvariánst, az EXIT a kilépési feltételt, míg a VAR a terminálló függvényt (ciklusvariáns). A REQ egy logikai kifejezés, mely megadja, hogy a ciklus milyen feltételek mellett hajtható végre. Feltételezzük továbbá, hogy a formulákban szereplő v egész típusú változó a metódusban nem szerepel.

Természetesen egy ciklus helyességét nemcsak a fenti formulákhoz viszonyítva értelmezhetjük. A ciklustól elvárt tényleges működés helyességének belátása és ellenőrzése attól függ, hogy mennyire vagyunk képesek a ciklus belső működésének lényegét megfogni a rendelkezésre álló eszközökkel. Extrém esetben a *True* értéket, mint formulát, használhatjuk a ciklusinvariánsban is. Ebben az esetben a ciklusinvariáns nyilvánvalóan megfelel bármilyen követelménynek. Ugyanakkor a *True* igazságérték, mint formula, nem árul el semmit a ciklus működéséről.

Egymásba ágyazott ciklusok

Gyakran előfordul olyan helyzet, amikor a ciklusok egymásba ágyazása szükséges. Ilyenkor célszerű először a legbelső ciklust megvizsgálni. Ha sikerül olyan invariánst és variánst találni, melyek leírják a ciklus terminálását és működését, akkor ez a ciklus már mint atomi programblokk kezelhető. Ezek után a legbelső ciklus egyetlen utasításnak tekinthető és összpontosíthatunk az őt beágyazó ciklusra. Ezt az eljárást egészen addig ismételjük, amíg a legkülső ciklus helyességi specifikációjának megadására nem kerül sor. Természetesen minden esetben figyelembe kell venni a belső ciklus helyességi formuláinak és kifejezéseinek a külső ciklus változóitól való függőségét és fordítva.

Check utasítások

Emlékezzünk vissza, hogy egy metódus törzsében tetszőleges ponton megadhatók a program működésével kapcsolatos hipotézisek **check** utasítás formájában. Egy M metódust check-konzisztensnek nevezünk, ha bármely, az előfeltételeknek megfelelő paraméterkombinációra a futása során minden **check** utasításban megadott feltétel teljesül.

Kivételkezelés és helyesség

Az Eiffel-programozó a kivétel kezelésére egy speciális programblokkot írhat eljárásszinten. A kivételkezelő blokk vezérlési szerkezete alapvetően kétfajta ágra bontható. Az optimista

ágak **retry** utasítással végződnek és feltételezik, hogy bizonyos előkészületek után, melyeket maguk az ágak tartalmaznak, a teljes eljárás törzse újrafuttatható újabb kivétel kiváltása nélkül. Természetesen az újrafuttatáshoz szükséges, hogy a kivételkezelő blokk lefutása után az eljárás ismét meghívható legyen, azaz az ilyen ágaknak vissza kell tudniuk állítani az eljárás teljes előfeltételét és a teljes osztályinvariánst is nyilvános metódus esetében. Nem nyilvános metódus esetében megelégszünk az előfeltétel visszaállításával, hiszen ezeknél a metódusoknál nem követeljük meg az osztályinvariáns fennállását.

Amennyiben nincs esély az eljárás normális lefutására – akár az eljárás törzsének és a kivételkezelő blokk **retry** ágának nulla vagy többszöri futtatása után –, úgy a kivételkezelő blokkban a nem kezelhető kivételek esetére megfogalmazott utasítások hajtódnak végre, végül az eljárás kivétel kiváltásával ér véget. Mivel ilyen esetben az eljárás futása sikertelen, nincs lehetőség az utófeltétel teljesítésére, de a nyilvános metódusok esetén az osztály teljes konzisztenciájának visszaállítását megköveteljük.

A kivételkezelésre vonatkozó következő definíciót Meyer vezette be [Mey 91, Mey 00]. Azt mondjuk, hogy egy M publikus metódus kivételkezelése akkor és csak akkor konzisztens, ha a kivételkezelő blokk minden B ágára, ami nem **retry** utasítással ér véget, $\{True\}B\{I\}$, ami **retry** utasítással ér véget, $\{True\}B\{I \wedge E\}$ Hoare-formulák érvényesek. Egy M nem publikus metódus kivételkezelése akkor és csak akkor konzisztens, ha a kivételkezelő blokk minden B ágára, ami **retry** utasítással ér véget, $\{True\}B\{E\}$ Hoare-formula érvényes.

Figyeljük meg, hogy a kivételkezelő blokk minden ága a legszigorúbb előfeltétellel rendelkezik ($True$). Ez azért van így, mert a nem várt esemény (kivétel) után nem támaszkodhatunk sem az előfeltételre, sem az osztályinvariánsra, hiszen a kivétel során a program abnormális állapotba kerülhetett.

A metódusok helyessége

Egy M metódus akkor és csak akkor helyes a helyességi specifikációjára nézve, ha check-konzisztens, a metódusban alkalmazott minden ciklus helyes és az $\{E\}B\{U\}$ Hoare-formula érvényes.

Vegyük észre, hogy a fenti meghatározás nem követeli meg egy metódustól, hogy konzisztens legyen a kivételkezelésre nézve. Az utóbbi tulajdonságot azért nem vettük bele a definícióba, mert a kivételkezelés a valós életben fellépő futásidejű probléma.

A programok helyességének ellenőrzése tisztán matematikai, logikai érvelés, mely nincs összefüggésben a programok futása során felmerülő gyakorlati problémákkal. Ha a fenti definíció szigorúbb lenne és tartalmazná a kivételkezelésre vonatkozó konzisztenciát is, akkor egy adott eljárás esetében – az eljárás működésétől függően – olyan dolgokat is meg kellene vizsgálnunk, hogy mi történik, ha elfogy a memória, megsérül a merevlemez, amire éppen írunk, vagy hibázik a processzor. Látható, hogy ebben az esetben egy eljárás helyességének vizsgálata lényegesen bonyolultabb lenne, nem is beszélve arról, hogy még egy bonyolultabb elméleti modell birtokában sem tudnánk minden lényeges helyzetet megvizsgálni.

Megjegyezzük, hogy Meyer [Mey 91, Mey 00] helyességi feltételében szerepel a kivételkezelési konzisztencia is. Megítélésünk szerint azonban a kivételkezelésre vonatkozó konzisztencia a metódusok hibatűrését határozza meg. A nem előre látható kivételek a helyes

programműködéstől független események. Lényeges itt az előre nem látható jelző. Termé szetesen az olyan előre látható események, mint például a nullával való osztás, nem okoz hatják a program helytelen működését. Az említett esemény előre látható, így az eljárás író jának fel kell készülnie rá. A felkészülés pedig ebben az esetben nem kivételkezelést, hanen esetszétválasztást jelent. Az osztás során a nulla speciális értéket jelent, ezzel az algoritmu megfogalmazójának tisztában kell lennie a program fejlesztésekor.

13.2.3. Az osztályok helyessége

Az osztály konzisztenciája

Az Eiffelben minden változó típusától függő alapértékkel rendelkezik, melyet akkor ves fel, ha nem kap más értéket. Jelölje D az osztály alapértelmezett értékeit leíró feltételt. A C osztály akkor és csak akkor konzisztens I-re, mint teljes osztályinvariánsára, ha bármel $\langle E, U \rangle$ specifikációval rendelkező nyilvános metódus esetében a következő formulák érvé nyesek [Mey 91]:

- $\{D \wedge E\} B \{U \wedge I\}$ ha a metódus konstruktor
- $\{I \wedge E\} B \{U \wedge I\}$ ha a metódus nem konstruktor.

A fenti definíció jól érzékelteti az osztályinvariáns jelentését. A meghatározásban sze replő első feltételt mint indukciós alapállítást is értelmezhetjük, a második feltétel pedig a indukciós lépés, mely biztosítja, hogy amennyiben az indukciós hipotézis fennállt a művc let végrehajtása előtt, úgy egy konzisztens osztály esetében fenn fog állni a végrehajtás utá is.

Nem esett szó még a *VEREM* osztály konstruktoráról. A fenti definíció birtokába most megadjuk a konstruktort is.

```
-- Létrehozás: k tárolókapacitású verem létrehozása.
keszit(k : INTEGER) is
require
   ervenyes_kapacitas: k > 0
do
   create tomb.make(1, k)
   elemszam := 0
   kapacitas := k
ensure
   hely_lefoglalva:      tomb /= void
   uj_verem_ures:        ures
   kapacitas_beallitva: kapacitas = k
end
```

Figyeljük meg a konstruktor előfeltételét, mely megköveteli, hogy a verem kapacitás pozitív legyen. Ha az előfeltételt nem adnánk meg, akkor nem lenne biztosított az osztály invariáns fenntartása. Az utófeltételben a sikeres létrehozással kapcsolatos logikai állításoka soroltuk fel. Biztosítottuk, hogy az objektum számára a tárterület lefoglalható, az új verem objektum üres és a megfelelő kapacitással rendelkezik.

Osztályhelyesség

Egy C osztály akkor és csak akkor helyes a specifikációra nézve (metódusok elő- és utófeltételei, teljes osztályinvariáns, ciklusokban lévő állítások, ellenőrző utasítások), ha konzisztens és minden metódusa helyes.

13.2.4. A programok helyessége

Mielőtt rátérnénk az objektumorientált programhelyesség definíciójára, illetve az ezzel kapcsolatos ellenőrzés problémáira, ismerkedjünk meg néhány meghatározással.

Kapcsolatok

Egy adott programrészben A osztály ügyfélként, B osztály pedig kiszolgálóként szerepel, ha A valamely eljárásában közvetlenül vagy közvetetten hivatkozik B valamely attribútumára, eljárására, vagy A valamely attribútuma, metódusa B valamely metódusára, attribútumára épül közvetlenül vagy közvetetten.

Vegyük észre, hogy egy adott objektum (osztály) különböző relációkban ügyfélként és kiszolgálóként is szerepelhet.

A kapcsolat nemcsak osztályszinten, hanem az osztályok példányainak szintjén is értelmezhető.

Függőség

Azt mondjuk, hogy A osztály megvalósítása függ B osztályétól, ha van A osztály megvalósításában olyan programkörnyezet, melyben A és B rendre ügyfélként és kiszolgálóként fordul elő. A függőséget így jelöljük: $B \rightarrow_f A$. Jelentése A osztály megvalósítása függ B-től. A függőségi reláció reflexív és tranzitív. A függőség reflexív és tranzitív lezártját $\rightarrow_f^+$-vel jelöljük. Például az $A \rightarrow_f^+$ jelentése az összes osztály, melynek megvalósítása A osztályétól függ. Alkalmazzuk az $\rightarrow_f^+ A$ jelölést is azon osztályok halmazára, melyektől A osztály megvalósítása függ. A függőségi definíció következménye, hogy egy osztály mindig függ az ősosztályaitól.

Programhelyesség

Legyen adott egy P program, melynek futtatása C osztály egyik konstruktor eljárásának meghívásával jár. Ekkor azt mondjuk, hogy P program akkor és csak akkor helyes a specifikációra nézve, ha C osztály helyes. Megjegyezzük, hogy a C osztály pedig akkor és csak akkor helyes a specifikációra nézve, ha C osztályt alkotó, illetve a C osztály által használt osztályok mindegyike helyes. Más szavakkal a P program akkor és csak akkor helyes, ha $\rightarrow_f^+ C$ osztályok mindegyike helyes. A *VEREM* osztály esetében ez azt jelenti, hogy az osztály akkor és csak akkor helyes, ha

- az alkalmazott *ARRAY* osztály és *INTEGER* osztály helyes, valamint
- maga a *VEREM* osztály helyes.

13.3. A programhelyesség-ellenőrzés problémái

Ebben a bekezdésben röviden ismertetjük a programhelyesség-ellenőrzés problémáit és korlátait.

13.3.1. Függőségek

Többszörös függőségek

A korábban ismertetett megközelítés – miszerint a program helyességének ellenőrzése felbontható az egyes osztályok helyességének ellenőrzésére – szépséghibája, hogy a programösszetevők között fennálló függőségi viszonyok miatt nem minden esetben alkalmazható közvetlenül.

Az osztályok között ugyanis többszörös függőségi viszony is fennállhat és a függőség szimmetrikus is lehet. Egy adott C osztály helyességének ellenőrzése általában nem végezhető elkülönülten. A C osztály helyessége éppen a $\rightarrow_f^+ C$ osztályok halmazának helyességétől függ.

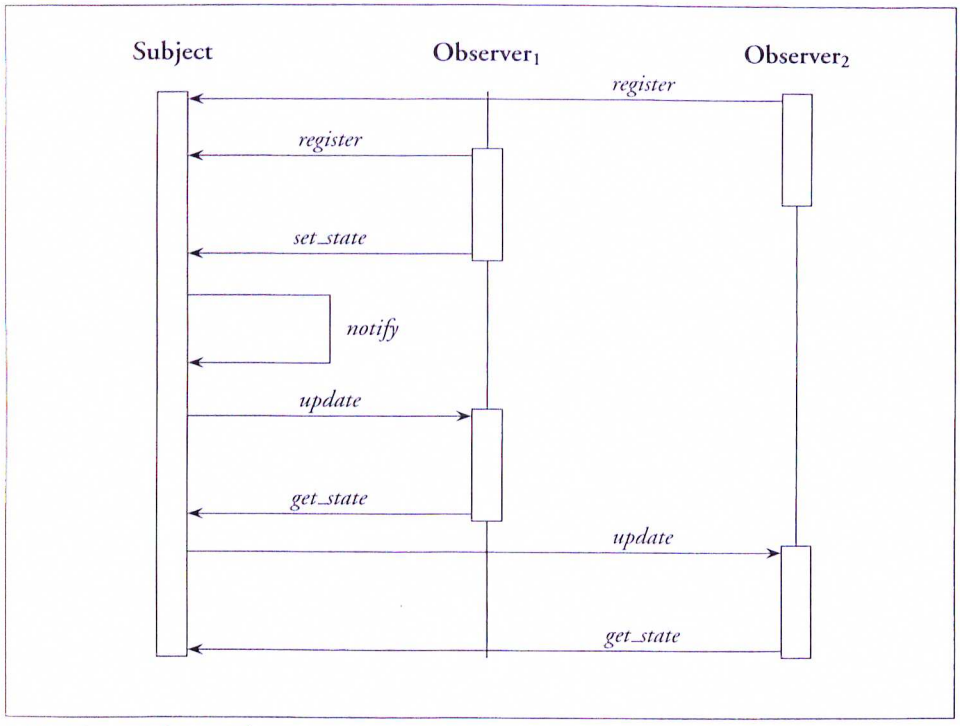
Például $A \rightarrow_f B$, $B \rightarrow_f C$ és $C \rightarrow_f B$ függőség esetén A, B és C osztályok helyességéről nem beszélhetünk külön-külön. A helyesség ellenőrzése során a B és C osztály, mint egy logikai egység kezelendő. A B és C osztályok helyességének ellenőrzéséhez előfeltétel továbbá, hogy A osztály helyessége már igazolt legyen, hiszen $\rightarrow_f^+ B = \{A, B, C\} = \rightarrow_f^+ C$.

A tárgy-vizsgáló minta

A 13.1. ábra egy ismert osztálytervezési minta szekvenciadiagramját [Gam 95] mutatja be UML jelölést alkalmazva. A diagramon jól látható, hogy a mintában szereplő két komponens, a Subject (tárgy) és az Observer (vizsgáló) között szimmetrikus függőségi viszony áll fenn, így helyességük csak együttesen értelmezhető.

- A vizsgáló osztály példányai, melyek egy konkrét tárgy állapotáról akarnak friss információval rendelkezni, bejegyzik magukat a vizsgáló osztály megfelelő példányánál (*subject.register(observer$_1$)*, *subject.register(observer$_2$)*). A bejegyzés a *subject* osztály példányánál általában egy listába való felfűzést jelent.
- Később az *observer$_1$* objektum módosítja a *subject* objektum állapotát (*subject.set_state()*).
- A *subject* objektum a módosításról a *notify* eljárás segítségével értesül, a *notify* eljárás pedig sorban meghívja *(observer.update)* az *observer* osztály azon példányait, melyek bejegyezték magukat (szerepelnek a listában).
- Végezetül minden értesített vizsgáló példány a *subject.get_state* eljárást meghívva értesül a módosult tárgy állapotáról.

Természetesen léteznek ennél jóval bonyolultabb tervezési minták is, ahol az osztályok közötti összefüggések és függőségek még összetettebbek.



13.1. ábra: Subject-Observer szekvenciadiagram

13.3.2. Helyességi specifikációs nyelv

Gyakorlati korlát

A fejezet elején összevetettük az elsőrendű logikát és az Eiffel helyességi specifikációs nyelv szerkezeteit. Láttuk, hogy az elsőrendű logika nem minden lehetősége támogatott közvetlenül az Eiffelben. A figyelmes olvasó észrevehette, hogy a *fordit* eljárás utófeltétele nem teljes körű, mert nem követeli meg, hogy a verem minden eleme megcserélődjön. Ennek oka az, hogy a specifikációs nyelv nem ad lehetőséget kvantorok használatára. Nem fogalmazhatunk meg olyan kijelentést, hogy „minden elem felcserélve".

Korábban óvatosan fogalmaztunk és azt mondtuk, hogy ezek a formulák nem támogatottak közvetlenül. A kvantorokat tartalmazó formulák ciklusokkal vagy rekurzív eljáráshívásokkal valósíthatók meg. Bizonyos egyszerű esetekben a „létezik olyan elem" típusú formulák kikerülhetők az alaposztályok eljárásainak használatával. Például a *LINKED_LIST* osztály rendelkezik egy *has* függvénnyel, mely igaz értéket ad vissza, ha a listában szerepel egy adott elem. Azaz például a formula, hogy „6 létezik az *l* listában" megadható az *l.has(6)* kifejezéssel.

Próbáljuk meg megadni Eiffelben a „minden elem felcserélt" utófeltételt. Mivel közvetlenül ilyen predikátum nem írható, függvény formájában oldjuk meg ezt. A *fordit* metódus utófeltétele tehát az alábbi logikai állítással bővül:

minden_elem_felcserelve: elemek_felcserelve

Eddig rendben is volna a dolog, de mielőtt elhamarkodottan ítélnénk, vizsgáljuk meg, hogyan adható meg az *elemek_felcserelve* függvény egy lehetséges változata.

```
elemek_felcserelve:BOOLEAN is
local
   elem_poz : INTEGER
do
   result := True
   from
      elem_poz := 0
   invariant
      ervenyes_elem: elem_poz >= 0
                     and elem_poz <= elemszam
   variant
      ellenorzott_elemek:   elemszam - elem_poz
   until elem_poz = elemszam
         or not result
   loop
      elem_poz:=elem_poz + 1
      if tomb.item(elem_poz) /= regi.item(elemszam + 1 - elem_poz)
      then
                result:=False
      end
   end
end
```

A függvény feltételezi, hogy rendelkezésre áll a *tomb* változó *fordit* művelet előtti állapota, melyet a *regi* nevű globális változóban tárolunk. Az *elemek_felcserelve* függvény helyesen működik, de a helyes működésnek belátása nem magától értetődő. Vegyük észre, hogy az univerzális kvantort egy ciklussal váltottuk ki. Természetesen ilyen esetben igazolni kell, hogy az új metódus valóban helyes, sőt be kell látni a ciklus befejezését is.

Általánosságban, ha összetettebb predikátumokat valósítunk meg *BOOLEAN* visszatérési értékű függvények formájában, akkor a predikátumok helyességét is külön igazolni szükséges.

A helyességi specifikáció jelentősége éppen abban rejlik, hogy a program működését egy magasabb elvonatkoztatású szinten tudjuk megadni, csupán a „mit" kérdésre koncentrálva, megfosztva az imperatív elemektől. Az *elemek_felcserelve* függvény ennek nem tesz eleget. Vegyük észre, hogy korábban is alkalmaztunk *BOOLEAN* visszatérési értékű függvényeket,

mint predikátumokat. Ilyen volt a *tele* vagy az *ures* predikátum. Ezeknek a predikátumoknak a helyessége azonban könnyen belátható.

```
tele : BOOLEAN is
do
   result := elemszam = kapacitas
end
```

A *tele* függvény helyessége azért is biztosított, mert lényegében minden imperatív szerkezettől mentes. Az *elemszam = kapacitas* maga is egy predikátum, méghozzá olyan, mely az elsőrendű logikának is eleget tesz. Az egyetlen imperatív elem, az értékadás, csak technikai jelentőséggel bír.

Összességében elmondható tehát, hogy az Eiffelben tetszőlegesen összetett predikátumokat fogalmazhatunk meg, de óvakodnunk kell az olyan függvényektől, melyek működése összetettebb: ciklusokat vagy rekurzív eljáráshívásokat tartalmaz.

Elméleti korlát

Az előbbieknél fontosabb korlátozást jelent, hogy az axiomatikus szemantika kifejezőereje gyengébb az operációs szemantikáénál [Mit 98], azaz még ha ideális specifikációs nyelv birtokában lennénk is, akkor sem lennénk képesek megfogalmazni deklaratív állításokkal a program működésének minden lényeges aspektusát.

13.4. A *VEREM* osztály

Végezetül megadjuk a *VEREM* osztály teljes Eiffel nyelvű megvalósítását.

```
indexing
   description: "tömb alapú LIFO verem"
   status:      "demo"
   names:       verem
   contents:    generikus
   date:        "$Date: 2003/06/11 17:01:10 $"
   revision:    "$Revision 1.00$"

class VEREM[G]

creation
   keszit

feature{NONE}
   tomb:      ARRAY[G] -- a verem adatait tömbben tároljuk
   elemszam:  INTEGER -- a veremben lévő elemek száma
   kapacitas: INTEGER -- a veremben elhelyezhető elemek száma
```

```
feature{ANY}
-- Létrehozás:
-- k tárolókapacitású verem létrehozása.
keszit(k : INTEGER) is
require
   ervenyes_kapacitas: k > 0
do
   create tomb.make(1, k)
   elemszam := 0
   kapacitas := k

ensure
   hely_lefoglalva: tomb /= void
   uj_verem_ures: ures
   kapacitas_beallitva: kapacitas = k
end


-- műveletek:
-- új elem elhelyezése
berak(e : G) is
require
   nincs_tele: not tele
do
   elemszam := elemszam + 1
   tomb.put(e, elemszam)

ensure
   uj_elem_a_tetejen: elem = e
   elemszam_nott: elemszam = old elemszam + 1
   kapacitas_valtozatlan: kapacitas = old kapacitas
   nem_ures: not ures
end


-- Aktuális elem törlése.
torol is
require
   nem_ures: not ures
do
   elemszam := elemszam - 1

ensure
   elemszam_csokkent: elemszam = old elemszam - 1
   ures_ha_egy_elem_volt: old elemszam = 1 implies ures
   kapacitas_valtozatlan: kapacitas = old kapacitas
   nincs_tele: not tele
end
```

```
-- Felcseréli a verem elemeinek sorrendjét.
fordit is
require
   nem_ures: not ures
local
   also_poz    : INTEGER
   felso_poz   : INTEGER
   also_elem   : G
   felso_elem  : G
   also_csere  : INTEGER
   felso_csere : INTEGER
do
   from
      also_poz    := 1
      felso_poz   := elemszam
      also_csere  := also_poz
      felso_csere := felso_poz
   invariant
      ervenyes_poziciok:
         also_csere + felso_csere = elemszam + 1
         and also_csere >= 1
         and also_csere <= felso_csere
         and felso_csere <= elemszam
   variant
      fennmarado_intervallum: felso_poz - also_poz + 1
   until (felso_poz - also_poz) < 1
   loop
      also_csere  := also_poz
      felso_csere := felso_poz
      also_elem   := tomb.item(also_csere)
      felso_elem  := tomb.item(felso_csere)
      tomb.put(felso_elem, also_csere)
      tomb.put(also_elem, felso_csere)
      check
         elemcsere:
            also_elem = tomb.item(felso_csere)
            and felso_elem = tomb.item(also_csere)
      end
      also_poz := also_poz + 1
      felso_poz := felso_poz - 1
   end
ensure
   elemek_felcserelve:
      old tomb.item(1) = tomb.item(elemszam)
      and old tomb.item(elemszam) = tomb.item(1)
   tobbi_valtozatlan:
      deep_equal(strip(tomb), old strip(tomb))
end
```

```
-- Lekérdezések:
-- Aktuális elem
elem : G is
require
   nem_ures: not ures
do
   result := tomb.entry(elemszam)
end

-- Állapotjelentések:
-- Üres-e
ures : BOOLEAN is
do
   result := elemszam = 0
end

-- tele van-e
tele : BOOLEAN is
do
   result := elemszam = kapacitas
end

invariant
   ervenyes_kapacitas:
      kapacitas > 0
   ervenyes_elemszam:
      elemszam >= 0
      and elemszam <= kapacitas
   ures_definicio:
      (ures implies elemszam = 0)
      and (elemszam = 0 implies ures)
   tele_definicio:
      (tele implies elemszam = kapacitas)
      and (elemszam = kapacitas implies tele)
end
```

Feladatok

1. Az alábbi feladat Meyertől származik [Mey 00]. Azért választottuk, mert rendkívül szemléletes és hozzájárul az elő- és utófeltételek megértéséhez.

   Tegyük fel, hogy a hatékonyság érdekében a munkáltatók az állásajánlataikban a munkakör leírását Hoare formulákkal, a megszokott $\{E\}M\{U\}$ formában adják meg. Ez a leírás segíti a munkavállalót abban, hogy gyorsan és céltudatosan kiválaszthassa a számára legmegfelelőbb munkakört. Az E előfeltétel leírja, hogy az M munkát milyen feltételek mellett kell teljesíteni, az U utófeltétel pedig a munka eredményére ad megszorítást.

   Adja meg a világ legkönnyebb és a világ legnehezebb munkakörét M erejéig!

2. Adja meg a mosógép egy lehetséges helyességi leírását!

3. Miért nem helyes az alábbi helyességi leírás?
{mosópor, vízlágyító, öblítő, elektromos áram, víz} mosás {tiszta ruha}

4. Mi a probléma az alábbi függvénnyel, amelyik egy másodfokú egyenlet valós gyökeit számolja ki? A paraméterekben az a, b és c változók a megszokott módon rendre a másodfokú tagot, az elsőfokú tag együtthatóját és az állandó tagot jelölik. Az x változóban x értéke szerepel. Az $x1$ és $x2$ globális változók.

```
x1:REAL
x2:REAL

gyok(a, b, c, x : REAL)
   require
      pozitiv_diszkriminans: b * b − (4 * a * c) > 0
   local d:REAL;
   do
      d := sqrt(b * b − (4 * a * c))
      x1 := (−b + d) / (2 * a)
      x2 := (−b − d) / (2 * a)
   ensure
      helyes_gyokok: a * x * x + b * x + c = a * (x − x1) * (x − x2)
   end
```

5. Adja meg az alábbi osztályok lehetséges kapcsolatait öröklődés szempontjából, ha az elő-, utófeltételekben és osztályinvariánsokban lévő formulák azonos nevű előfordulásai minden osztályban ugyanazt jelentik, és a formulák nem tartalmaznak függvényekre való hivatkozást! (Az osztályokban mindenhol a teljes elő-, utófeltételek és osztályinvariánsok szerepelnek.)

```
class A          class B          class C          class D
...              ...              ...              ...
m is             m is             m is             m is
require          require          require          require
 pre             pre               pre and pre'     pre or pre'
do               do               do               do
...              ...              ...              ...
ensure           ensure           ensure           ensure
 post            post and post'    post             post and post'
end              end              end              end
...              ...              ...              ...
invariant        end              invariant        invariant
  inv                               inv and more     inv
end                                end              end
```

14 Párhuzamos nyelvi elemek

Ebben a fejezetben röviden bemutatjuk a párhuzamos programozás legfontosabb elemeit. Megvizsgáljuk a kommunikációs és szinkronizációs modelleket és eszközöket: a tevékeny várakozást, a kölcsönös kizárást, a kritikus szakasz problémakörét, a szemafort és a monitort. Részletesebben is áttekintünk néhány programozási nyelvet (az Ada, a CSP, az Occam, a Java, a PVM és az MPI programozási nyelveket).

A *párhuzamos programozás* több részfeladat egyidőben történő végrehajtásához nyújt segítséget. Napjainkban egyre több feladat megoldása igényel ilyen programozást. A legjellemzőbb három eset, amelyben párhuzamos megoldásra van szükség, a következő:

1. A probléma logikai szerkezete igényli a felbontást (például kiszolgálóprogramok, grafikai alkalmazások). Ezekben az esetekben legalább két, nagymértékben független részfeladatot kell egyidőben megoldani. Ilyenkor gyakran a legegyszerűbb és leglogikusabb megoldás, ha a megoldóprogramban minden független részfeladatot önálló, a többi egységgel párhuzamosan futó egységként ábrázolunk.
2. A program több, fizikailag független eszközön fut (például számos internetes alkalmazás). Ebben az esetben szükségszerű a párhuzamos megvalósítás, hiszen a fizikailag független eszközökön futó programegységek csak így ábrázolhatók megfelelően.
3. A probléma sokkal hatékonyabban oldható meg párhuzamos eszközök felhasználásával. Ilyenkor a feladat szekvenciálisan is megoldható, de ha egyszerre több processzor dolgozik ugyanazon a problémán, hatékonyabb lesz a megoldás. Sok esetben ez a párhuzamos gyorsítás nagyon nagy is lehet.

A párhuzamosság bevezetésére az első próbálkozás az volt, amikor párhuzamosították a bemeneti-kimeneti műveletek végrehajtását a központi egység műveleteinek végrehajtásával. A következő nagyobb lépés az operációs rendszerek különböző műveleteinek párhuzamos végrehajtása volt. Ezáltal megjelent a *multiprogramozás*. A cél a gép erőforrásainak maximális kihasználása volt. Ugyanekkor vezették be a *folyamatok* (*processzek*) fogalmát. A folyamatok egy adott feladatot elvégző, önálló programpéldányok. Egy folyamatot azzal az utasítássorozattal határozunk meg, amelyet végrehajt. Azt is mondhatjuk, hogy egy folyamat olyan események sorozata, amelyeket az egyes utasítások végrehajtásával váltunk ki. A folyamatok viselkedésük alapján lehetnek egymástól függetlenek, egymással versengők és egymással együttműködők. Természetesen olyan gép használata esetén, amely csak egy processzort tartalmaz, csak látszólagos párhuzamosságot érhetünk el. Az ilyen rendszerekben a processzor folyamatosan váltogatva szolgálja ki az egyes folyamatokat, így egyszerre csak egyetlen folyamatot hajt végre, viszont adott időtartam alatt akár több száz folyamatot is kiszolgálhat. Valódi párhuzamos végrehajtást csak többprocesszoros rendszerekben érhetünk el.

A programozási nyelvekben a párhuzamosság több szinten is megvalósítható. A legelemibb szint az utasítás szintje, mely két vagy több utasítás párhuzamos végrehajtását jelenti. A párhuzamosságot az alprogramok szintjén is meghatározhatjuk, az úgynevezett *taszkokba* több utasítást vagy alprogramot csoportosíthatunk, majd ezeket párhuzamosan futtathatjuk. A legmagasabb szinten a párhuzamosan végrehajtható teljes programokat különböztetjük meg.

Ezek közül a nyelvi elemek bemutatásánál elsősorban az utasításszintű és az alprogramszintű párhuzamossággal foglalkozunk, azaz egy vagy több utasítást, illetve alprogramot magába foglaló egységek párhuzamosságát tárgyaljuk.

A párhuzamosság lehet *valódi* vagy *látszólagos* párhuzamosság. Valódi párhuzamosságról akkor beszélünk, ha a különböző egységeket fizikailag is különböző processzorokon hajtjuk végre, míg látszólagos párhuzamosságról akkor, ha az egységek egy részét ugyanazon a processzoron hajtjuk végre, időosztásos rendszer alapján. A folyamatok konkrét leképezése egy vagy több processzorra megvalósítási kérdés.

Kommunikációs és szinkronizációs modellek

A két alapvető probléma, amelyet meg kell oldanunk két vagy több folyamat elindításakor, a folyamatok közötti szinkronizáció és kommunikáció. A kommunikáció során egy folyamat hozzáfér egy másik, vele párhuzamosan futó folyamat által szolgáltatott információhoz. A kommunikáció megoldható *közös, osztott memóriaterület* használatával, vagy *üzenetküldéssel*.

Az első esetben a program változóinak egy része (esetleg minden változó) hozzáférhető egyszerre több folyamat számára. Ekkor két párhuzamos folyamat úgy kommunikálhat, hogy az egyik beírja az átadni kívánt információt egy változóba, a másik folyamat pedig kiolvassa onnan. Ennél a megoldásnál nagyon fontos a folyamatok összehangolása, mivel ha két folyamat például egyidőben ír ugyanarra a memóriaterületre (ugyanabba a változóba,) a kapott eredmény határozatlan érték lesz.

A második esetben az egyik folyamat üzenetet küld a másik folyamatnak. Az üzenetküldés lehet aszinkron vagy szinkron. Az *aszinkron kommunikációs* modellben, ha egy folyamat üzenetet küld, akkor nem várja meg, hogy a másik folyamat fogadja az adott üzenetet, a küldő folyamat végrehajtása csak az üzenet elküldésének idejére függesztődik fel. Ezzel szemben *szinkron kommunikáció* (*randevú*) esetén, ha egy folyamat kommunikálni akar egy másik folyamattal, akkor végrehajtása felfüggesztődik addig, amíg a megszólított folyamat alkalmas nem lesz az üzenetvételre. A két folyamat csak a kommunikáció befejeztével futhat tovább.

Kölcsönös kizárás és szinkronizáció

A kommunikáción kívül a másik kritikus probléma a szinkronizáció. A szinkronizáció olyan mechanizmus, mellyel a párhuzamos folyamatok összehangolhatók, azaz meghatározható a különböző folyamatokban szereplő műveletek relatív sorrendje.

A sorrend több esetben is fontos lehet. Ha az egyik folyamat üzenetet vár egy másik folyamattól, meg kell várnia, míg a másik elküldi az üzenetet. Ha pedig két folyamat

ugyanazt az erőforrást szeretné használni, az egyiknek meg kell várnia amíg a másiknak már nincs szüksége az adott erőforrásra.

A folyamatok összehangolásánál több olyan egyedi probléma is felmerülhet, amelyek elkerülésére a konkrét megvalósításkor figyelnünk kell. Az egyik ilyen a holtpont (deadlock) problémája. A holtpont olyan állapot, melynél a folyamatok körkörösen egymásra várakoznak, és egyik sem tud tovább haladni. Például tegyük fel, hogy van két folyamatunk és mindkettőnek szüksége van az *A* és a *B* erőforrásra is a működéséhez. Ebben az esetben például holtponthoz vezet, ha az egyik folyamat lefoglalja az *A* erőforrást és ezután megpróbálja lefoglalni a *B*-t, míg ezzel egyidőben a másik lefoglalja a *B* erőforrást és ezután megpróbálja lefoglalni az *A*-t. Ebben az esetben egyik folyamat sem tud tovább haladni.

A másik ilyen probléma az úgynevezett koplalás. Ebben az esetben több folyamat szeretné ugyanazt az erőforrást használni, de van, amelyik sohasem fér hozzá, mivel mindig leköti azt valamelyik másik folyamat. Például ez az eset következik be, ha két folyamat szeretne nyomtatni ugyanazon a nyomtatón, de az egyik folyamat folyamatosan nyomtat, ezért a másik egyáltalán nem tud nyomtatni.

A kommunikációs és szinkronizációs kérdéseket a fenti problémák elkerülésével kell megoldani.

A szinkronizációt (ha az adott nyelven ez rendelkezésre áll) megoldhatjuk randevú, azaz szinkron üzenetküldés segítségével is, de a szinkronizáció megvalósítására többféle, ettől eltérő módszert is kidolgoztak. Mielőtt a szinkronizációs módszerek közül a fontosabbakat áttekintenénk, bevezetünk néhány fontos fogalmat.

Osztott változók esetén felmerül az a kérdés, hogyan használhatják a folyamatok ezeket a változókat oszthatatlanul, azaz, hogy az egyes folyamatok ne akadályozzák meg egymást az osztott változók segítségével végrehajtott műveletekben. Például tegyük fel, hogy a P és a Q folyamat az $x := x + 1$ utasítást konkurens módon hajtja végre. Míg a P folyamat az x értékét olvassa a memóriából, addig a Q folyamat éppen az összeadást végzi, majd tárolja az eredményt, majd ezek után a P folyamat is tárolja a saját eredményét, ezáltal az x értéke valójában nem az elvárt lesz, csak 1-gyel nő, nem 2-vel. Így ha az értékadó utasítást az egyes folyamatok nem oszthatatlanul hajtják végre, információvesztés következhet be. Ha úgynevezett kölcsönös kizárást biztosítunk az x változóra, a folyamatok osztatlanul férnek hozzá. A kölcsönös kizárás segítségével elérhetjük az osztott változók biztonságos használatát. Kritikus szakasznak nevezzük a program forrásának azt a részét, melyben a program futása során egyidőben csak egy folyamat tartózkodhat (például közös memóriaterületre történő írás) [Pet 81].

A fenti példában az $x := x + 1$ utasítás a programban kritikus szakasz. A kölcsönös kizárás biztosítja, hogy egyidőben csak egy folyamat hajtja végre egy program kritikus szakaszát. Ahhoz, hogy a folyamatok számára biztosítsuk a kölcsönös kizárást, a folyamatok a kritikus szakasz előtt végrehajtanak egy belépési protokollt, a kritikus szakasz végrehajtása után pedig egy kilépési protokollt. Ezzel megakadályozzuk, hogy több folyamat tartózkodjon a kritikus szakaszban.

Tevékeny várakozás (busy waiting) • Amíg a szükséges helyzet elő nem áll, a folyamat dinamikusan várakozik, azaz folyamatosan vizsgálja, hogy a kívánt feltételek teljesülnek-e. Hátránya, hogy a folyamat várakozás közben foglalja a processzoridőt. A *tevékeny várako-*

zás leginkább a kritikus szakaszok kezelésére szolgál, ami az egyik alapvető szinkronizációs feladat.

Szemafor • A párhuzamos programozásban, valamint az operációs rendszerekben a szemafor nagyon fontos eszköz, mivel ennek segítségével védhetjük és vezérelhetjük azokat a folyamatokat, amelyek egy kritikus szakasz műveleteit kell végrehajtsák. A szemafort E. W. Dijkstra vezette be [Dij 68]. A legáltalánosabb formájában a szemafor egy egész típusú változó, melyhez egy várakozási sor tartozik. Egy *s* szemafornak két művelete van, a *P(s)* és a *V(s)* művelet. Ezek a műveletek atomi módon hajtódnak végre. Egy folyamat atomi művelete azt jelenti, hogy a művelet bármilyen közbeeső állapota nem látható a többi folyamat számára [And 91].

A szemafor műveleteit a következőképpen határozhatjuk meg:

P(s): Ha $s > 0$, akkor *s* értékét csökkenti eggyel, különben a *P(s)*-t végrehajtani akaró folyamat bekerül *s* várakozási sorába.

V(s): Ha van várakozó folyamat az *s*-hez tartozó várakozási sorban, akkor elindítja a legelsőt, különben *s* értékét növeli eggyel.

A szemaforhoz tartozó változó kezdeti értéke határozza meg, hogy egyszerre hány folyamat léphet be a kritikus szakaszba. A szigorú vagy bináris szemafor változójának csak két értéke lehet 0 vagy 1. Ebben az esetben, ha a szemafor változójának értéke 1, akkor a *V(s)* műveletnek nincs hatása.

A szemafor előnyei közé tartozik, hogy felhasználásával hatékony program készíthető, a várakozó folyamat nem foglalja a központi processzort (bekerül egy várakozási sorba). Viszont hátránya, hogy a strukturáltság ellen hat, és használatával könnyű holtpontot előidézni, azaz olyan állapotot, melynél a program nem tud tovább futni, mivel a párhuzamos folyamatok egymásra várakoznak és ezáltal mindegyik blokkolódik.

Kölcsönös kizárásra és holtpont elkerülésére a következőképpen használhatunk bináris szemafort:

```
s := 1;
P(s);       // Belépési protokoll végrehajtása.
...         // Kritikus szakasz végrehajtása.
V(s);       // Kilépési protokoll végrehajtása.
```

A bináris szemafort zárolónak (lock) is nevezik, melynek segítségével a folyamatok kölcsönös kizárását megvalósíthatjuk, ezért a *P(s)* műveletet zároló műveletnek, *V(s)*-t pedig nyitó műveletnek is nevezik.

Monitor • A monitor a szemafornál strukturáltabb párhuzamos programnyelvi elem, amely az adatszerkezeteket, a változókat és a rajtuk értelmezett műveleteket úgy zárja egységbe, hogy a monitorhoz tartozó eljárások közül egyidőben legfeljebb egy lehet aktív. A monitor belső adatszerkezetei a monitoron kívül csak a monitor által meghatározott műveleteken keresztül érhetők el. A monitoron belül használhatunk úgynevezett feltételes változókat is. Minden feltételes változóhoz tartozik egy várakozási sor és a szemaforhoz hasonlóan két megengedett művelete van: a *wait* és a *signal*.

A műveletek jelentése a következő:

c: condition;

wait(c): A hívó monitorfolyamat blokkolódik a *c*-hez tartozó várakozási sorban, újraindítani majd valaki más fogja.

signal(c): Ha van *c*-hez tartozó blokkolt folyamat a várakozási sorban, akkor az első folyamatot elindítja, különben nem történik semmi.

Első látásra a feltételes változók nagyon hasonlítanak a szemaforokhoz, azonban van egy nagyon fontos jelentésbeli különbség köztük. A feltételes változókhoz nem tartozik számláló, azaz nem gyűjti a beérkezett *signal* kérelmeket, mint ahogy azt a szemafor teszi. Ha egy feltételes változó *signal* műveletét akkor hívjuk meg, amikor épp nem várakozik senki az adott sorban, a *signal* elvész. [Hoa 74b]

Feltételes kritikus szakasz • A szemafor másik alternatívája a feltételes kritikus szakasz utasítás, melyet Brinch Hansen vezetett be körülbelül a monitorral egyidőben. Ez a szerkezet egy nyelvi elemekkel meghatározott kritikus szakasz, amelyben a folyamat a védett változóhoz csak ellenőrzötten férhet hozzá. A megadott feltételnek teljesülnie kell, mielőtt a vezérlés belép a kritikus szakaszba. A kritikus szakaszon kívül egyetlen folyamat sem férhet hozzá a védett változóhoz, és minden folyamat, amely hozzá akar férni a változóhoz, várakozik, amíg a feltétel igazzá nem válik, valamint amíg másik folyamat használja a változót. [Han 72]

Természetesen ezeken az eszközökön kívül létezik több, többnyire nyelvfüggő módszer is a szinkronizáció megoldására (például az Ada védett objektuma), melyekkel az adott nyelvnél foglalkozunk részletesen.

14.1. A párhuzamos nyelvek általános jellemzése

A párhuzamosságot támogató programozási nyelveket nagyon sok szempont alapján osztályozhatjuk.

A párhuzamos programok alapvetően nem determinisztikusak. Ez meghatározza a párhuzamos programozásban használt vezérlési típusokat. A párhuzamos programok vezérlésénél a fő szempont a feladat minél hatékonyabb, biztonságosabb megoldása, párhuzamos, elosztott erőforrásokkal rendelkező környezetben.

A párhuzamos programok vezérlésére különböző modellekkel próbálkoztak. A programozási nyelvekben található nyelvi elemeket a vezérlés alapján is csoportosíthatjuk [Pâr 92]. Az alábbiakban a vezérlési típusokat soroljuk fel, ezek mind alacsonyszintű, mind pedig magasszintű párhuzamosságra vonatkozhatnak. Az alább felsorolt modelleknek megfelelhet egy-egy programozási nyelvosztály is.

- *Ellenőrzésvezérelt* típusú nyelvek (control driven computation) esetén az elindított folyamatok ellenőrzése *fork, join, wait* típusú parancsokkal történik.
- Az *adatvezérlésű* modellben (data driven computation) a folyamatok végrehajtási sorrendje az adatok szerkezetétől, az adatok közötti összefüggésektől függ. A párhuzamos folyamatok végrehajtása akkor kerül sorra, amikor már minden adat a folyamat rendelkezésére áll. Ilyen modellen alapul például az Irvine Dataflow (ID) és a Value Algorithmic Language (VAL).

- Az *igényvezérlésű* modellben (demand driven computation) a folyamatok akkor hajtódnak végre, amikor ezeket egy másik folyamat kifejezetten kéri, így a vezérlést mindig a kérések határozzák meg. Erre épül például a Parlog vagy a Concurrent Prolog.

Egy másik típusú párhuzamos programozás a tömbök és mátrixok feldolgozását valósítja meg a párhuzamos architektúrájú gépeken. A gépek vektor- vagy mátrixprocesszoros gépek, melyek azonos szerkezetű és ugyanolyan természetű adatokkal dolgoznak. A folyamatok közötti kommunikáció szinkron kommunikáció. Ez a fajta párhuzamos programozás az egyik legrégebbi típusú, az ebbe a kategóriába tartozó nyelveknél alapvetően háromféle megközelítést különböztethetünk meg a párhuzamossági kérdések megoldására.

A különböző FORTRAN-változatok (FORTRAN IV, CFT, Cyber FORTRAN) a `do` ciklusból próbálják kiszűrni a párhuzamosítható utasításokat, ezáltal automatikusan határozzák meg a párhuzamosan végrehajtható részeket.

A párhuzamosság közvetlen megadásánál (CFD és DAF FORTRAN) már szükséges a párhuzamos számítógép szerkezetének ismerete, ezért az ilyen nyelvek szintaktikája és szemantikája elősegíti a gép architektúrája szerinti programok írását, de a programok nem lesznek hordozhatók.

Harmadik csoportba azokat a nyelveket sorolhatjuk, amelyek a párhuzamosságot már a gép architektúrájától függetlenül próbálják megvalósítani (például Actus).

A fent említett nyelvek bár nagyon korai nyelvek, olyan nyelvi elemeket tartalmaznak, melyeket később a modern programozási nyelvek átvettek és továbbfejlesztettek. Például a későbbi programozási nyelvek, melyek a párhuzamosságot az adatvezérlésű modell alapján valósítják meg (C*, Dataparallel-C) ilyen típusú tömbökkel dolgoznak, ezért itt röviden jellemezzük őket.

A Cyber 200 FORTRAN magasszintű programozási nyelv. A fordító állapítja meg, hogy a szekvenciális program mely részei hajthatók végre párhuzamosan. Azokat az egyszerű, valamint egymásba ágyazott `do` törzseket párhuzamosítja, amelyek tömbökkel dolgoznak. Ezt a műveletet *vektorizálásnak* nevezik [Cyb 83].

A CFT ugyanolyan elvek alapján dolgozik, mint a Cyber, azzal a különbséggel, hogy az egymásba ágyazott ciklusokat nem tudja feldolgozni. A két nyelvben írt programokat nem lehet más gépre átültetni, mivel csak azokon a gépeken futtathatók, amelyekre tervezték.

A CFD FORTRAN esetében a központi egység dekódolja az utasítást, és végrehajtja vagy a processzorokra küldi. A központi egység csak azokat az aritmetikai műveleteket végzi el, melyek a címszámításokhoz vagy a ciklusok kezeléséhez szükségesek. A nyelvet SIMD (egy utasítássorozat és több adatfolyam) típusú géphez készítették. A programozó dönti el, hogy az utasítást a központi egység vagy egy bizonyos processzor hajtja végre (ebben az esetben meg kell adni a processzor számát). A nyelvi elemek között találunk egy olyan `if` utasítást, amellyel vektorokat hasonlíthatunk össze és az összehasonlítás eredményének függvényében a vektorokat párhuzamosan meg is változtathatjuk. A nyelv rendelkezik továbbá az `and`, `or`, `not`, illetve az `any` és `all` operátorokkal; ez utóbbi kettővel feltételeket szabhatunk. A `wait` utasítás segítségével pedig az adatcseréket szinkronizálhatjuk.

A VS FORTRAN taszkokat használ, melyek saját memóriával és saját kóddal rendelkeznek. A párhuzamos kódot a programozó is megadhatja, vagy pedig a fordító deríti fel

a párhuzamosítható útvonalakat. Virtuális processzorokkal dolgozik, melyeket az operációs rendszer valódi processzorokhoz csatol, így ez a nyelv már valamennyire gépfüggetlen és a programok több számítógép-architektúrán futtathatók. A párhuzamos kódot a fordító automatikusan hozza létre, a `do` ciklusok és az iterációk már aszinkron módon futtathatók. A párhuzamosítás feltétele, hogy az iterációk egymástól függetlenek legyenek. Az utasítások: `originate`, `terminate`, `schedule`, `wait for task`, `wait for any task`, `wait for all tasks`. Párhuzamos útvonalakra a következő műveletekkel rendelkezik a nyelv: `parallel do`, `local`, `dobefore`, `doevery`, `doafter`, `exit`, `paralel sections`, `section`, `end sections`, `parallel call`, `wait for all calls`. A VS FORTRAN kritikus szakaszok végrehajtására blokkolási lehetőséget is nyújt.

Az Actus [Per 79, Per 83] Pascal nyelven alapuló, párhuzamos adatszerkezetekkel rendelkező programozási nyelv. Független a géptől, dolgozhat mind vektorprocesszorokkal, mind mátrixprocesszorokkal. A párhuzamosság közvetlenül is megadható, így az erőforrásokat maximálisan ki lehet használni. Az **array** párhuzamosan feldolgozható tömb, elemei lehetnek egyszerű vagy összetett típusúak. Párhuzamos tömbfeldolgozás esetén meg kell adni az első, illetve az utolsó elem indexét. A tömbökkel a *rotate* vagy a *shift* műveleteket végezhetjük.

Ezeket a nyelvek a többprocesszoros gépek programozási nyelveiként is említik, de mint láthattuk a folyamatok ugyanazt a műveletet hajtják végre ugyanolyan típusú adatcsomagokra, ezért néha szinkron nyelveknek is nevezik őket.

A modern párhuzamos programozási nyelvek a következő kérdésekre keresik a választ, amikor új jelentéssel rendelkező nyelvi elemet vezetnek be:

- A nyelv milyen szinten szeretné megvalósítani a párhuzamosságot?
- A nyelv a párhuzamosságot osztott memóriával vagy üzenetküldéssel valósítja meg?
- Léteznek-e mellékhatások?
- Milyen típusú kommunikációt szolgáltat a nyelv?
- Milyen alapvető műveleteknek megfelelő nyelvi elemeket tartalmaz a nyelv?

A programozási nyelveket így többféleképpen lehet osztályozni. A kritériumokat a fenti kérdésekre adott válaszok is meghatározhatják. Ugyanakkor vannak olyan nyelvek, amelyek több kategóriába is besorolhatók.

- Osztott változókkal rendelkező nyelvek:
  Pascal Plus [Wel 79], Concurrent Pascal [Her 79], Modula-2 [Wir 82, Bla 90].
- Üzenetküldéssel rendelkező nyelvek, ahol a kommunikálás send-receive típusú utasításokkal történik: CSP [Hoa 78], Occam [Bar 92].
- Osztott memóriát használó adatvezérlésű nyelvek:
  VAL [Ack 79], Dataflow [Arv 78], Lucid [Wad 85].
- Objektumorientált párhuzamos nyelvek: Emerald [Hut 87], Pool-T [Ame 87].
- Funkcionális programozási nyelvek, melyek párhuzamos nyelvi elemeket tartalmaznak: Clean [Pla 93], a különböző Haskell- vagy ML-változatok (Glasgow Parallel Haskell [Loi 01], pH [Nik 01], JoCaml [Fou 01]).

A *folyamat (processz)* a párhuzamos programozás egyik legfontosabb fogalma. A következőkben aszerint tekintünk át néhány programozási nyelvet, hogy milyen jelentéssel bír ez a fogalom. Ez a rövid áttekintés egyben a párhuzamos programozásban használt nyelvi elemek sokféleségét, illetve ugyanannak a fogalomnak a széles értelmezési skáláját is bemutatja, a teljesség igénye nélkül.

Mint látni fogjuk, nagyon sokszínű a folyamatok értelmezése, használata, konkrét megvalósítása. Ugyanakkor többféle elnevezéssel is találkozhatunk. A folyamatot az alábbiakban a folyamatok közötti kommunikáció szempontjából is láttatjuk, mivel ez az egyik legfontosabb jellemzője.

A folyamat fogalmára épít a Modula-2 [Wir 82, Bla 90]. A folyamatok osztott változók és az úgynevezett jelek segítségével kommunikálnak. A jelek a processzmodulokból exportált műveletek. Ezeket szinkronizálásra használják, ezek nem tartalmaznak (hordoznak) adatot. Egy folyamat küldhet jeleket (*send*) vagy pedig várhat jeleket (*wait*) egy másik folyamattól. A folyamat további műveletei: *startprocess*, mely egy új párhuzamos folyamatot indít, valamint az *awaited* művelet, mely egy adott jel jellemzője és azt mutatja, hogy van-e legalább egy folyamat, mely a jelre vár. A jelek abban különböznek a szemaforoktól, hogy egy olyan jel, amelyet egy folyamat sem vár, nulla műveletnek számít. A nyelv érdekességéhez tartoznak a korutinok, amelyek párhuzamosan futó folyamatok, egymást aktiválják. Amikor az egyik folyamat újból aktív lesz, onnan folytatódik, ahol az előző vezérléscserénél megszakadt. Az alábbi példa egy egyszerű termelő-fogyasztó feladat megoldása párhuzamos folyamatok segítségével:

```
MODULE ProcessDemo;
  FROM Terminal IMPORT
    Read, Write;
  FROM SYSTEM IMPORT
    PROCESS, NEWPROCESS, TRANSFER, ADR, SIZE;
  VAR p, c: PROCESS;
      workspace: ARRAY[1..100] OF CARDINAL;
      char: CHAR;
  PROCEDURE Consumer;
  BEGIN
    LOOP
      IF char="$" THEN EXIT END;
      Write(char);
      TRANSFER(c, p);
    END
  END Consumer;
BEGIN
  NEWPROCESS(Consumer, ADR(workspace), SIZE(workspace), c);
  LOOP
    Read(char);
    TRANSFER(p, c)
  END
END ProcessDemo.
```

A *TRANSFER(p1, p2)* megszakítja és felfüggeszti a *p1* folyamatot és átadja a vezérlést a *p2* folyamatnak.

Ugyancsak a korutin fogalmára épít a SIMULA 67 programozási nyelv [Lam 82]. A SIMULA 67 korutinok kapcsán az alábbiakban a vezérlés bonyolultságát is láthatjuk. A *korutin* valamilyen osztályba tartozó objektum, amely azonos vagy más osztályba tartozó objektumokkal működik együtt. A létrehozott objektum (*korutin*) alárendeltségi viszonyban van azzal az objektummal, amely létrehozta. A létrehozott objektum a *detach* utasítással adhatja vissza a vezérlést a létrehozó objektumnak. A létrehozó objektum a *call(X)* utasítással aktiválhatja újra a leválasztott objektumot. Az egyik korutinból (objektumból) a vezérlést a *resume(X)* utasítás segítségével adhatjuk át az *X* korutinnak (objektumnak). A vezérlés visszaadásakor az objektumok végrehajtása ott folytatódik, ahol a legutóbbi vezérlésátadáskor megszakadt. Egy korutin objektum élettartama a létrejöttekor kezdődik, majd az objektum elkezdi végrehajtani a törzsében található utasításokat. Az objektum ilyenkor működő állapotban van, és a generátorhoz (ahhoz az objektumhoz, ami létrehozta) rendelt objektumnak nevezzük.

A korutin egy *detach* utasítással lemondhat a futásról a létrehozó javára, ilyenkor az objektumot leválasztottnak, de még nem lezártnak nevezzük. Az objektum egy *resume(X)* utasítással egy másik korutin javára lemondhat a vezérlésről. Az objektum újra vezérlést kaphat egy rá vonatkozó *call(X)* (melyet a generátor hajthat végre) vagy egy *resume(X)* (melyet egy „testvérfolyamat" hajthat végre) utasítás hatására. Az objektum futása befejeződik, ha törzsének végrehajtása elérte a befejező **end** kulcsszót. Ilyenkor az objektum futását sem *call*, sem *resume* utasítással nem lehet felújítani. Az objektum azonban továbbra is létezik, változói és műveletei elérhetők.

A Portalban [Bus 85] a folyamatnak különböző állapotai lehetnek: aktiválható – azaz megkaphatja a vezérlést; felfüggesztett; várakozó – egy jelre vár; illetve külső – azaz egy monitoron kívül található. Egy Portal program a megszakításrendszert is kezelheti, így valós idejű programozásra is használható.

Az ALGOL 68-ban [Bra 79] a vesszővel elválasztott és **begin–end** közé írt utasítások párhuzamosan futtathatók. A **begin–end** addig nem fejeződik be, amíg minden vessző közötti egység – párhuzamos klóz – véget nem ér. A szinkronizálásra Dijkstra-szemaforokat vezettek be (*sema*). A szemaforokban a párhuzamos klózok elé ki kell tenni a *par* kulcsszót.

A Concurrent Eiffelben a párhuzamosságot egy kulcsszó valósítja meg, a *separate*. Egy *separate* objektum eljárásainak meghívása után a hívó metódus nem függesztődik fel, hanem folytatja tevékenységét; függvény esetén pedig a hívó megvárja az eredményt. *Separate* lehet egy osztály vagy metódus is. Egy ilyen objektum létrehozásakor az objektum metódusai gyakorlatilag az új „processzoron" futnak. Ha egy paraméter *separate*, akkor más objektum nem használhatja, mivel automatikus kizárás történik. Ha egy objektumnak mégis szüksége lenne rá, akkor az *exception* és a *yield* segítségével ez megtörténhet. Más párhuzamos objektumorientált Eiffel-változatok: Distributed Eiffel, Eiffel//.

A Modula-3-ban [Bösz 96], bár szintén a Modula nyelvcsaládhoz tartozik, a folyamatokat szálaknak nevezik. Ezek közös memóriaterületen kommunikálnak. A párhuzamosság eszközei a *Thread* modulban találhatók (*Fork*, *Join*).

```
...
TYPE
   Closure = Thread.Closure OBJECT
                a, b, eredmény: INTEGER;
             OVERRIDES
                apply := Start
             END;
VAR
   cl := NEW(Closure);
   tread: Thread.T;
   a: ARRAY [1..4] OF INTEGER;
   max: INTEGER;

PROCEDURE Start(cl: Closure): REFANY =
BEGIN
   cl.eredmény := Max(cl.a, cl.b);
   RETURN NIL
END START;

BEGIN
   cl.a := a[1]; cl.b := a[2];
   thread := Thread.Fork(cl);      (* 4 szám maximuma párhuzamosan *)
   max := Max(a[3], a[4]);
   EVAL Thread.Join(thread);
   max := Max(max, cl.eredmény);
```

A kölcsönös kizárást a *Lock* parancs és a *Mutex* típus valósítja meg. Az alábbi példa a *Lock* által védett helyfoglalást mutatja be:

```
...
TYPE
   Járat = MUTEX OBJECT
              n: CARDINAL;
              ...
           END;
PROCEDURE Foglalás(járatszám: Járat): BOOLEAN =
BEGIN
   LOCK járatszám DO
      IF járatszám.n > 0 THEN DEC(járatszám.n); RETURN TRUE
      ELSE RETURN FALSE
      END;
   END;
END Foglalás;
...
```

A *Channel* absztrakt típus segítségével csatornákat hozhatunk létre és üzeneteket küldhetünk a csatornákon.

Ugyancsak a szál fogalmával találkozunk a Java nyelvben [Har 98], ahol a folyamatok leírására a *Thread* osztályt használják, amelynek egy objektuma egy folyamatot jelöl. A folyamat törzse a *run* metódusban leírt kód. Egy programban tetszőleges számú szálat indíthatunk. Ezeknek megadhatjuk a prioritásukat, felfüggeszthetjük, újraindíthatjuk, megállíthatjuk őket. A szálakat a *join* segítségével szinkronizálhatjuk. A kommunikáció osztott változókon keresztül történik, a kölcsönös kizárás biztosítására a **synchronized** kulcsszó szolgál. Ha egy blokk vagy metódus **synchronized**, akkor a hozzá csatolt monitor biztosítja a kölcsönös kizárást (lásd a Java programozási nyelvről szóló részt).

A Delphiben a *Thread* osztályból képzett alosztályok párhuzamosan végrehajthatók. A szinkronizálásra és a kommunikálásra a főosztályban létre kell hozni egy eljárást, amely a vezérlésért felelős, majd a *synchronize(⟨eljárásnév⟩)* segítségével a párhuzamos alosztályok kommunikációját ellenőrizhetjük.

A BETA-ban [Mad 93] egy objektumból külön végrehajtási szálat lehet létrehozni az `P:@|Activity` utasítással. Egy ilyen szál párhuzamosan futhat más szálakkal. Két objektumot is végrehajthatunk párhuzamosan a `fork` segítségével. A `semaphore` mintában található a bináris szemafor szerkezete:

```
(# S: @semaphore;
   A: @ | (# do imp1; S.P; imp2; S.V; imp3 #);
   B: @ | (# do imp4; S.P; imp5; S.V; imp6 #);
do S.V; A.fork; B.fork
#)
```

Itt az `A` komponens `imp1`-et és `imp3`-et párhuzamosan hajthatja végre a `B` komponens bármely műveletével. `A` és `B` nem hajthatja végre párhuzamosan `imp2`-öt és `imp5`-öt.

A `Monitor` minta a kölcsönös kizárást biztosító monitort tartalmazza.

A szinkronizációt a `Port` nevű mintával valósíthatjuk meg, melynek `entry` műveletével a folyamatok szinkronizálását érhetjük el.

A folyamat egy másik lényeges megvalósítása a taszk. A PL/I [Gru 90] egyike azoknak a nyelveknek, amelyek korán bevezették a taszk fogalmát. A taszkok párhuzamosan végrehajtott programrészek. Egy `P` eljárás meghívhat egy `Q` eljárást, amely egyidőben fut a `P`-vel: `Call Q(`⟨*paraméterek*⟩`) Task(X)`. `Q` tájékoztatja `P`-t, hogy végrehajtotta feladatát, az `X` eseményváltozó (event variable) segítségével. `P` tesztelheti az `X`-et, vagyis ellenőrizheti, hogy `Q` végrehajtás közben van-e, vagy pedig befejeződött. `P` a `wait(X)` segítségével késleltetheti `Q`-t. `Q` örökölheti `P` minden változóját, így `P` és `Q` közös adatokat oszthat meg. A PL/I a konkurens programozást az utasítások szintjén is megengedi.

A folyamatokat az Ada programozási nyelvben is taszk-objektumok ábrázolják. A taszkok rendelkezhetnek úgynevezett belépési (entry) pontokkal, melyek egy másik taszkból meghívhatók. A taszknak van egy törzse, ez írja le azt a tevékenységet, amelyet a folyamat végez. A törzsben minden belépési ponthoz tartozik egy **accept** utasítás; amikor itt tart a törzs végrehajtása, akkor hajlandó elfogadni egy másik taszk hívását erre a randevúra. A kommunikáció szinkron, a hívó folyamat a randevú elfogadásáig és az **accept** utasítás végéig felfüggesztődik. A belépési pontnak lehetnek paraméterei (**in**, **out** vagy **in out** típusúak). Ha egy taszk meghívja egy másik taszk belépési pontját, akkor futása felfüggesztődik a

randevú létrejöttéig és kezeléséig. Ezt meg lehet előzni időzített belépési pont hívás alkalmazásával, amikor is megadhatjuk, hogy legfeljebb mennyi időt várjon a randevú elfogadására. Ha a hívott taszk ennyi idő alatt nem válik fogadóképessé, akkor a hívó taszk végrehajtása folytatódik. A részleteket lásd az Ada programozási nyelvről szóló külön fejezetben.

A folyamatok úgynevezett csatornákon keresztül is kommunikálhatnak, üzenetküldés segítségével. A CSP/K nyelv, valamint az SP/K nyelv a PL/I kiterjesztései. Az üzenetküldés szinkron: `Q ! x` – egy `P` processz az `x` üzenetet küldi `Q`-nak; `P ? x` – `Q` az `x` változóba helyezi `P` üzenetét. A folyamatot a következőképpen írjuk le: `név: procedure options (concurrent)`. A `név: process;` deklarálással belépési pontot adhatunk meg. (A CSP nyelvről később még szólunk.)

Az Occam [Bar 92] programozási nyelvnek két alapeleme van: a folyamat (`proc`) és a csatorna (`chan`). Hat konstruktorral rendelkezik: a `Seq` egymás után (szekvenciálisan) helyezi el a folyamatokat, a `Par` párhuzamosan, az `Alt` alternatív módon hajtja végre feltétel teljesülése szerint, az `if` azt a folyamatot hajtja végre, amelyre a kifejezés értéke igaz, a `while` ismételt végrehajtást jelöl, a `for` pedig 0-tól `n`-ig azonos folyamatokat hoz létre. A változók bevezetése a következőképpen történik: `var változó:` (a : jellel kapcsolunk egy változót egy folyamathoz.) A folyamat deklarálása így történik: `proc processz`, a csatornáé pedig így: `chan csatorna`. A kommunikáció üzenetküldéssel valósul meg. (Bővebben lásd az Occam nyelvről szóló részt.)

Csatornát használ a Par C is, mely a szabványos C kibővítése a csatorna adattípussal és a **par**, illetve a **select** utasításokkal. A folyamatok közötti kommunikáció a csatorna vagy a globális változó segítségével történik. A **par** formailag a **for**-hoz hasonlít és a párhuzamos folyamatok elindítását eredményezi. A csatornán bármilyen típusú adat közvetíthető, a **select** pedig az Ada **select**-jéhez hasonlít.

A Pascal nyelvcsaládban például a Super Pascal használ csatornákat. A `forall` segítségével indítjuk el ugyanazt az eljárást több példányban, a `parallel` utasítással pedig párhuzamos utasításokat hajthatunk végre. A folyamatok csatornákon keresztül kommunikálnak. A kommunikálás nyelvi elemei az `open` (új csatorna létrehozása), `send` (egy kifejezést egy megadott csatornára küld), illetve a `receive` (leolvas egy értéket a megadott csatornáról) eljárások.

A folyamatok kapcsolattartásának fontos eleme a monitor, mely ugyanúgy többféle értelmezéssel bír. A Concurrent Pascalban [Her 79] a monitor a folyamatok közötti osztott változók egységbe zárása, az osztály pedig absztrakt adattípus. Az osztály példánya egy folyamathoz vagy monitorhoz csatolható. A monitor biztosítja adatainak védelmét, a monitor eljárásai pedig egy várakozási sorban állnak, így egy adott pillanatban csak egy eljárás futtatható. A monitor egy eljárása felfüggeszthető a `delay`-jel vagy folytatható a `continue`-val, az előkészítés pedig az `init`-tel történik.

A Portal ugyancsak Pascal típusú nyelv és a konkurens programozásra szintén a monitort használja. A Modula-2-ben az osztott változókat a monitorok tartalmazzák, ezek segítségével kommunikálnak a folyamatok. A CSP/K és az SP/K a monitort használja a kölcsönös kizárásra és a `wait-signal`-t a szinkronizálásra.

Külön kategóriába tartoznak az úgynevezett funkcionális nyelvek, melyekről a funkcionális programozási nyelvekről szóló fejezetben olvashatunk. A funkcionális nyelvek két legfontosabb művelete a kifejezések egyszerűsítése és kiértékelése. Ezekben a nyelvekben egy függvényhívást úgy tekinthetünk, mint egy processzt. A függvények és a kifejezések kiértékelése bizonyos annotációk, illetve kiértékelési módszerek hatására párhuzamosan történhet. A folyamatok dinamikusan hozhatók létre (például VAL [Ack 79], ParAlf, Concurrent Clean [Pla 93], Distributed Haskell, JoCaml [Fou 01]).

A logikai nyelveknél a szabályok kiértékelése párhuzamosan is történhet. Ilyen például a Concurrent Prolog, illetve a Parlog.

A Parlog logikai nyelvet az *And/Or* párhuzamosság jellemzi. Az *And* a , műveleti jellel adható meg és párhuzamosan (balról jobbra) értékeli ki a kifejezéseket. A klózokat . vagy ; választja el egymástól. A . műveleti jellel elválasztottakat párhuzamosan bizonyítja, ha pedig ez nem sikerül, akkor a ; jel utániakat próbálja bizonyítani (*Or* párhuzamosság). A párhuzamos egységek közös változókon keresztül kommunikálnak egymással. A szinkronizáció automatikus. Az együttműködés aszinkron, a párhuzamos egységek dinamikusan hozhatók létre és a rekurzió is megengedett.

A szekvenciális objektumorientált nyelvekre az alábbi három feltétel teljesül: a program futása pontosan egy objektum létrehozásával kezdődik. Amikor egy objektum üzenetet küld egy másik objektumnak, akkor az eredmény megérkezéséig nem csinál semmit, azaz megvárja az üzenet fogadását és feldolgozását. Egy objektum csak akkor aktív, amikor egy bejövő üzenet feldolgozására egy metódust futtat.

A párhuzamosságot úgy lehet elérni, hogy a fenti feltételek valamelyikét elhagyjuk, azaz több folyamat lehet aktív egyszerre. Az üzenetküldő nem várja meg az üzenet fogadását és feldolgozását, hanem az üzenet elküldése után rögtön folytatja futását (aszinkron kommunikáció). Az objektumok nem várnak passzívan az érkező üzenetekre, hanem minden objektumnak saját törzse van. Az objektum létrehozásakor elkezdődik a törzs végrehajtása, a többi objektum törzsével párhuzamosan. A törzs meghatározott pontjain megszakítható, hogy a beérkező üzenetekre válaszoljon. Ez randevú formájában történik, azaz a küldő és a fogadó szinkronizál.

A Pool-T [Ame 87] párhuzamos egységei az objektumok. A kommunikáció randevú útján vagy üzenetek küldésével lehetséges. Az objektumok dinamikusan hozhatók létre és aszinkron módon működnek.

Párhuzamos környezetekre is hozunk néhány példát. A Linda program például kupacok gyűjteménye. Kétféle kupac van: aktív és passzív. Az aktív kupacok a folyamatok, a passzív kupacok pedig az adatok. Egy passzív kupac elemi típusok tetszőleges rekordja lehet. A kupacok egy úgy nevezett kupactérben (*Tuple Space*) helyezkednek el. A kupactérben lehet több teljesen azonos kupac is. Ha egy folyamat (aktív kupac) információt akar küldeni egy másiknak, akkor létrehoz egy kupacot és kihelyezi a kupactérbe, a másik folyamat pedig bevonja vagy csak elolvassa azt. Egy kupactérbeli kupac minden folyamat számára elérhető, de nem tartozik egyikhez sem.

Műveletek a kupactéren:

eval *(processznév)*: Az utasítás elindítja a megnevezett folyamatot.

out *(P1, . . .Pn)*: Az **out** utasítás végrehajtása egy *(P1, . . .Pn)* kupac beszúrását jelenti a kupactérbe. Az **out**-tal aktív kupacot is rakhatunk a kupactérbe, ez úgy történik, hogy a kupac egyes elemeinek értékét egy-egy folyamat számítja ki, és amikor az összes ilyen folyamat befejeződött, akkor a kapott eredménnyel a kupac „belefagy" a kupactérbe (passzív kupaccá válik).

in *(P1, . . .Pm)*: Az **in** utasítás a passzív kupacok beemelésére szolgál. A paraméterlistában kétféle elem lehet: érték és változó. Az **in** a kupactérben egy olyan kupacot keres, amely komponenseinek típusa rendre megegyezik a paraméterlistában megadott elemek típusával, továbbá az érték szerint megadott paraméterek értéke is egyenlő a megfelelő kupacelemek értékével. Ha nem talál ilyen kupacot, a hívó folyamat felfüggesztődik, amíg nem lesz egy megfelelő kupac. A megtalált kupacot (ha több ilyen is van, akkor nem determinisztikus módon az egyiket) eltávolítja a kupactérből, és a ?-ek után megadott változók felveszik a megfelelő kupacelem értékét.

rd *(P1, . . .Pk)*: Ugyanolyan, mint az **in** utasítás, de a kupacot nem távolítja el a kupactérből.

A Linda-val rokon környezetek: Piranha, Trellis, Linda Program Builder.

A Strand környezet párhuzamos taszkokat használ. A szinkronizáció osztott változókon keresztül történik, a folyamatok ezeken keresztül kommunikálnak. A Strand programok csak párhuzamos felépítésűek lehetnek, amelyben a programelemek egymással párhuzamosan futnak.

A CC++ komponenseket épít be. A *Parbegin–parend* szerkezettel párhuzamos blokkot hozhatunk létre. A szinkronizáció a *sync* típusú változókkal történik. A CC++ a párhuzamos komponenseket szekvenciális komponensekkel ötvözi. A komponensek együttműködését atomi függvényekkel valósítja meg.

Heterogén hálózatokban az MPI és a PVM nyújt párhuzamossági eszközöket folyamatok létrehozására, a kommunikáció szinkronizációjára, illetve párhuzamos programok fejlesztésére. (Bővebben lásd a megfelelő részeket.)

14.2. Ada

Az Adában [Nyé 98, Koz 93, Whe 96, Bar 96] a párhuzamosság kifejezésének nyelvi eszköze a **task** és a védett (**protected**) objektum.

14.2.1. Taszkok

A taszkok párhuzamosan végrehajtott egységek. Egy taszkot egy logikai processzor hajt végre azáltal, hogy sorban végrehajtja a taszk törzsének utasításait a szekvenciális programoknál megszokott módon.

Az egyes logikai processzorok egymástól függetlenek, párhuzamosan futtatják a taszkokat. A logikai egységek leképezésének módja fizikai processzorokra a nyelv implementációjához tartozó kérdés.

A taszkok egymástól függetlenül működnek és a szinkronizációs pontokon kommunikálnak egymással. A taszkok aszinkron módon együttműködő programegységek. A taszkok szinkronizálása a taszkok randevúival történik, ami egy hívást kiadó, illetve fogadó taszk között zajlik le.

Az Ada programozási nyelv szinkronizációt kifejező nyelvi elemei az **entry**, az **accept**, a **delay**, a **select** és az **abort**.

A taszkok az úgynevezett belépési pontokon (**entry**) keresztül kommunikálhatnak, így egy taszk, ha adatokat szeretne átadni vagy kapni egy másik taszktól, meghívja a taszk belépési pontját. A hívott taszkban az **entry**-hez legalább egy **accept** utasítás is tartozik és a hívás hatására az **accept** törzsébe fogalt utasítások hajtódnak végre. Az **entry** egy eljárás vagy függvény fejlécéhez hasonlít, mivel paraméterei is lehetnek, amelyek a fent említett taszkok közötti információáramlást szolgálják.

A következőkben a taszkegységekben kommunikációra és a taszkok szinkronizálására használható párhuzamos nyelvi elemeket ismertetjük. Ezek a következők: a **task** egység, mint az Ada programozási nyelvben a párhuzamosságot kifejező legfontosabb elem; az **entry** és a hozzá tartozó **accept**, melyek a taszkok közötti kapcsolatok leírásához szükséges elemek; a **select**, melynek összetett szemantikája lehetővé teszi a kommunikációs típusok megadását; a **delay** az időzítési kérdések megoldására, valamint az **abort** utasítás egy taszk végrehajtásának megszüntetésére. Célunk ezeknek az alapvető elemeknek a bemutatása, leírása néhány példa segítségével.

14.2.2. A taszk specifikációja és törzse

Az Adában a folyamatot a **task** programegységgel hozzuk létre, amely a taszk specifikációból és a taszk törzséből áll. A specifikáció és a törzs szerepe a taszk által végzett műveletek leírása. A **type** alapszó nélküli specifikáció egyedi taszkot definiál, amely a *névtelen taszk típushoz* tartozik. Ha egyedi taszkot hozunk létre, ezt úgy tehetjük meg, hogy először megadjuk a taszk specifikációját:

```
task Hello is
   entry üzenet(s: String := "Hello world!");
end Hello;
```

ezután a **body**, azaz a törzs részbe írjuk a taszk által végzett utasításokat:

```
task body Hello is
begin
   accept üzenet(s: String := "Hello world!") do
      put_line(s);
   end üzenet;
end Hello;
```

Amennyiben nem egyedi taszkot definiálunk, hanem taszk típust, viselkedési mintát szeretnénk létrehozni, akkor a specifikáció a **task type** alapszavakkal kezdődik. Ezzel olyan taszk típust definiálhatunk, amellyel taszk objektumokat hozhatunk létre a definíció példányosításával.

A taszk típus értékei olyan taszkok, amelyek **entry** pontjai a specifikációs részben felsorolt belépési pontok, a taszkok törzsét pedig ugyanúgy a **task body** definiálja.

Példa taszk típusok specifikálására:

```
task type Erőforrás is
   entry Lefoglal;
   entry Felszabadít;
end Erőforrás;

task type Vezérlés is
   entry Olvas (C: out Character);
   entry Ír (C: is Character);
end Vezérlés;

task type Üzenetrögzítő is
   entry hívó_azonosító(i: Integer);
   entry felvesz(s: Üzenet);
end Üzenetrögzítő;
```

A taszk törzse definiálja a taszk végrehajtásának hatását. A taszk törzsének végrehajtása a taszk elindításának hatására történik, a taszk elindítása pedig a program deklarációs részének kiértékelése után, a program törzsének végrehajtása előtt történik (részletesebben lásd később).

Példák egyedi taszkokra és taszk törzsekre:

```
task Termelő;
task Fogyasztó;
Üzenetrögzítők: array (1 .. 10) of Üzenetrögzítő;

task body Termelő is
   C: Character;
begin
   loop
      Termel(C);
      Tároló.Betesz(C);
   end loop
end Termelő;

task body Fogyasztó is
   C: Character;
begin
   loop
      Tároló.Kivesz(C);
      Fogyaszt(C);
   end loop;
end Fogyasztó;
```

```
task body Üzenetrögzítő is
begin
   accept hívó_azonosító(i: Integer) do
      put_line("Az " & Integer'Image(i) & ". számú ügyfél hívott.");
   end hívó_azonosító;
   accept felvesz(s: Üzenet) do
      put_line(Üzenet'Image(s));
   end felvesz;
end Üzenetrögzítő;
```

14.2.3. Taszk típusok, taszk objektumok

Minden taszk egy objektum, melyet *taszk objektumnak* nevezünk. Mint minden objektumnak, a taszknak is van típusa. Az ugyanolyan típusú taszk objektumok ugyanazt az utasítássorozatot hajtják végre, önálló módon.

Ha a taszk objektum olyan objektum, vagy olyan objektum komponense, amelyet

- egy objektumdeklarációban vezettünk be, a taszk objektum értékét a deklaráció kiértékelése nyomán határozzuk meg;
- egy mutató (**new**) kiértékelése nyomán hoztunk létre, a taszk objektum értékét a mutató kiértékelésekor határozzuk meg.

A taszk típus **limited** típus, azaz a taszk objektumokra nem alkalmazhatjuk az értékadás és a =, /= műveleteket, így műveleteinek halmaza lényegében a taszk attribútumaira korlátozódik. A taszk típusú objektumok értéke tehát állandó a program futása során. Taszk típusú objektum nem lehet alprogram **out** típusú vagy **generic in** típusú paramétere, de egy függvény visszaadhat taszk típusú objektumot, cím szerinti átadással. A taszk típusú objektumot ugyanúgy deklaráljuk, mint bármilyen más típusú objektumot.

```
T1, T2: Taszktípus;
-- létrehozhatunk taszkokból álló tömböt
type TaszkTömb is array(1 .. 10) of Taszktípus;
-- taszkmutatókat is létrehozhatunk az access segítségével
-- access-ről bővebben lásd a mutatókról szóló fejezetet
type TaszkMutató is access Taszktípus;
-- létrehozhatjuk dinamikusan a taszkot
TM: TaszkMutató := new Taszktípus;
```

Példák taszk változók deklarálására:

```
type Mutató is access Üzenetrögzítő;
Tömb: array (1 .. 10) of Üzenetrögzítő;
ÜzenetrögzítőMutató: Mutató := new Üzenetrögzítő;
```

Minden taszk objektumnak saját példánya van a taszk típusban deklarált változókból. Egy taszk típushoz tartozó új taszk objektum a többi objektummal párhuzamosan hajtja

végre a taszk törzsének utasításait. Egy taszk objektum a taszk helyi változói mellett használhatja a taszk törzsén kívül deklarált globális változókat is, melyek osztott változók is lehetnek.

Egy taszknak lehet diszkriminánsa része is. A diszkriminánsra ugyanazok a szabályok érvényesek, mint a tömb, illetve a rekord típus diszkriminánsaira. Ha egy taszknak van diszkriminánsa, akkor ez a taszk típus egy objektumának létrehozásakor értékelődik ki. A taszk objektum élettartama alatt a diszkrimináns értéke nem változik.

```
task type Azonosító (Diszkriminánsː Típus) is
   entry Deklarálás;
end Azonosító;
```

A diszkrimináns olyan paraméter, amely befolyásolja a taszk objektumok végrehajtását. Egy taszk típus objektumai ugyanazokat az utasításokat hajtják végre, a diszkrimináns azonban különböző értékeket vehet fel a különböző objektumoknál, így az utasítások végrehajtása különbözhet a diszkrimináns kezdeti értéke és a kommunikáció miatt.

Például ha különböző tömbökbe író taszk objektumokat akarunk létrehozni, akkor a tömb nevét diszkriminánsként adhatjuk meg:

```
task type T(Tnév: access String) is
   entry Betesz(K: Character);
end T;
```

Diszkriminánsos taszk nem lehet egyedi taszk. Példányosításkor írhatjuk, hogy:

```
A: T(Tnév => Karakterek'Access);
B: T(Tnév => Számjegyek'Access);
```

ahol *Karakterek* és *Számjegyek* két *String*.

Egy taszk törzsét helyettesítheti a következő deklarálás:

```
task body Név is separate;
```

amely azt jelöli, hogy a törzs külön forrásfájlban szerepel. A taszk törzsét tartalmazó forrásfájl pedig a **separate**(⟨*taszkot deklaráló eljárásnév*⟩) sorral kezdődik.

14.2.4. Taszkok elindítása, végrehajtása

A taszk végrehajtását a taszk törzse határozza meg. A végrehajtás kezdeti lépése a taszk elindítása (ez a deklarációs rész kiértékelését is magában foglalja).

A taszk elindítása az objektumot bevezető programegység törzse deklarációs részének feldolgozása után, a törzs első utasításának végrehajtása előtt történik.

Ha a deklaráció egy **package** specifikációs részében van, az elindítás a törzs deklarációs részének kiértékelése után és a taszk törzsének végrehajtása előtt történik. Az a taszk objektum vagy objektumelem, amit mutatóval hozunk létre, a mutató kiértékelésekor jön létre, az objektum nem taszk típusú komponenseinek inicializálása után.

Ha egy taszk elindítása közben hiba történik, a taszk befejezett (*komplett*) állapotba kerül, az elindítás befejezése után pedig a taszkot tartalmazó programegység törzsében az

első utasítást közvetlenül megelőző helyen, illetve a mutató kiértékelésének helyén *Tasking_Error* lép fel. Egy befejezett állapotba került taszkot már semmilyen módon nem lehet újra elindítani.

14.2.5. Taszkok lezárása (terminálása)

A taszk objektumok függőségi viszonyban állnak más programegységekkel. Minden taszk függ legalább egy szülőtől. A szülő lehet taszk, végrehajtás alatt álló blokk, alprogram vagy könyvtári csomag. A függés közvetlen az alábbi két esetben:

- A taszk szülője az a taszk(objektum), blokk, alprogram vagy könyvtári egység, amely a taszk objektum deklarációját tartalmazza.
- Ha egy taszk mutató (allokátor) kiértékelésével jött létre, akkor az **access** típus deklarációját tartalmazó taszk, blokk, alprogram vagy könyvtári egység a taszk szülője.

Ha egy taszk függ egy blokktól vagy alprogramtól, akkor a blokkot végrehajtó, illetve az alprogramot hívó taszktól, blokktól, alprogramtól vagy könyvtári egységtől is függ (közvetve). Ha egy *T1* taszk függ a *T2* taszktól, akkor a *T1* leszármazottja a *T2*-nek. Ha a *T1* taszk függ a *T2* taszktól, a *T2* pedig a *T3* taszktól, akkor a *T1* is függ a *T3*-tól.

A taszk befejezetté válik, ha véget ér a taszk törzsében levő utasítások végrehajtása, vagy ha a végrehajtás közben hiba lép fel és nincs a taszkban hibakezelő programrész, vagy ha van, és annak végrehajtása befejeződik.

Nem szabályos leállás (**abort**) esetén a leállított taszknak nincs lehetősége további randevúban részt venni, így esetleg a lekötött erőforrások sem szabadulnak fel. Az **abort** utasítás hatására a leállított taszk *abnormális taszkká* válik, majd az első szinkronizációs pont (belépésipont-hívás, másik taszk indításának kezdete, saját aktivizálásának vége, **select**, **abort**, **delay** utasítás, kivételkezelő vagy **accept** utasítás kezdete és vége) elérése előtt kompletté válik. Egy taszk önmagát is leállíthatja (öngyilkos taszk).

Egy taszk lezárul (*terminál*), ha:

- komplett, és az összes tőle függő taszk már lezárult, terminált.
- végrehajtása egy **terminate** utasításhoz ért és a taszk olyan programegységtől függ, amely már komplett (a végrehajtása befejeződött, tehát például nem könyvtári csomag) és a szülőtől függő összes leszármazott terminált már, komplett vagy szintén egy **terminate** utasításnál várakozik.
- **abort** következtében abnormális állapotba jutott, kompletté vált és törlődött a várakozási sorokból.

14.2.6. A taszkok attribútumai

A taszkokhoz is tartoznak attribútumok:

TaskName'Callable: logikai érték, ami pontosan akkor hamis, ha a taszk már kompletté vált, terminált vagy abortált (vagyis már nem lesz többé hívható).
TaskName'Terminated: logikai érték, ami pontosan akkor igaz, ha a taszk már terminált.
EntryName'Count: a megnevezett **entry**-hez tartozó **accept**-re várakozó hívások száma.

14.2.7. Entry, entry-hívás, accept utasítás

Egy taszknak lehetnek belépési pontjai is. Egy taszk kommunikálhat egy másik taszkkal, azáltal hogy meghívja azt (hasonlóan egy eljáráshíváshoz).

Amint már említettük, elsősorban az **entry** hívás biztosítja a taszkok szinkronizálását és adatcseréjét. Az **entry** deklarációja hasonlít az alprogram deklarációjához, de csak taszkban megengedett. A belépési pont tartalmazhat paramétereket, az **entry** formális és aktuális paramétereire pedig ugyanaz érvényes, mint az alprogramok paramétereire (a belépési pontnak nem lehet **access** típusú paramétere). A belépési pont specifikációja szabja meg a neki megfelelő belépési ponton történő kommunikáció irányát és az átadott üzenetek típusát.

Például a bevezetőben szereplő *Hello* taszknak van egy *üzenet* nevű belépési pontja, melyet a következőképpen hívhatunk meg: *Hello.üzenet(str);* ahol *str* a belépési pont paramétere.

Minden **entry**-hez tartozik legalább egy külön **accept** utasítás, mely azokat a műveleteket határozza meg, amelyeket akkor kell elvégezni, ha az **entry** hívása megtörtént.

A belépési pont segítségével egy taszk egy másik taszkkal közvetlen módon kommunikálhat. Ha egy taszk meghívja egy másik taszk belépési pontját, és ez válaszol a hívásra, akkor randevú jön létre a két taszk között. Amikor két taszk kommunikálni akar egy randevú alkalmával, akkor az első taszk, amely a találkozási ponthoz ér, vár addig, amíg a másik is a találkozási ponthoz ér.

Amikor egy taszk hívást intéz egy másik taszkhoz, és a hívott taszk végrehajtása még nem jutott el egy megfelelő **accept** utasításhoz, a hívó felfüggesztődik. Ha egy taszk végrehajtása egy **accept** utasításhoz ért és nincs a megfelelő **entry**-re hívás, akkor ez a taszk is felfüggesztődik.

Ha egy taszk meghívja egy másik taszk belépési pontját, akkor ez addig vár, amíg a második taszk fogadja hívását, ami azt jelenti, hogy a második taszk a hívott belépési pontnak megfelelő **accept** utasítás végrehajtásához kell érkezzen. Egy taszk saját belépési pontjának hívása holtpontot eredményez!

Egy **entry** hívásnál, ha a hívott taszk az **entry**-hez tartozó **accept** utasítás végrehajtásánál tart, végrehajtódik az **accept** utasítás törzse és a hívó taszk felfüggesztődik. Ezáltal valósul meg a két taszk között a *randevú*. Az **accept** törzs végrehajtása után mindkét taszk párhuzamosan fut tovább, egymástól függetlenül folytatva az utasítások végrehajtását.

Ha több taszk hívja ugyanazt az **entry**-t, miközben a végrehajtás nem érte el az **accept** utasítást, a hívások egy várakozási sorba kerülnek. Minden **entry**-hez külön várakozási sor tartozik. Egy adott belépési ponton kommunikálni kívánó taszkok érkezési sorrendben várakoznak (FIFO).

A megfelelő **accept** utasítás minden egyes végrehajtása kivesz egy várakozót a sorból. A várakozási sorból két módon törlődnek elemek: a legrégebben várakozó taszk hívásának fogadásával, ha a hívott taszk a belépési ponthoz tartozó **accept** utasításhoz ér (sikeres randevú), illetve a várakozásra szabott idő leteltével, vagy a várakozó taszk szabálytalan leállása (**abort**) következtében (elmaradt randevú). Egy adott pillanatban egy belépési pontnál várakozó taszkok száma az *E'Count* várakozási sor attribútum segítségével kérdezhető le.

Példa az **accept** utasítás használatára a *Tároló* taszk, mely egy elemet tud tárolni és melyet a *Termelő* és a *Fogyasztó* taszk hív:

```
task body Tároló is
   T: Character;
begin
   loop
      accept Betesz(C: in Character) do
         T := C;
      end Betesz;
      accept Kivesz(C: out Character) do
         C := T;
      end Kivesz;
   end loop;
end Tároló;
```

Az **accept** utasítás része a taszk törzsében található utasítások sorozatának. Egy belépési ponthoz több **accept** utasítás is tartozhat. Hívás esetén azt az **accept** utasítást hajtja végre, amelyhez a hívott fél hamarabb érkezett.

Az **accept** utasítás paraméterei megegyeznek az **entry**-ben található paraméterekkel. Randevú esetén a két taszk a paraméterek segítségével kommunikálhat, értékeket adhat át egymásnak. Ha az **accept** utasítás paramétere **in** típusú, a paraméter értéke bemásolódik a formális paraméterbe, ha **in out** típusú, az eredmény visszamásolódik a hívó fél megfelelő aktuális paraméterébe. Amíg az **accept** törzsének végrehajtása be nem fejeződik, a hívó fél várakozik.

Minden olyan tevékenységet, amelyet egy taszk végrehajthat a randevún kívül is, az **accept** törzsén kívül kell elhelyezni, hogy a hívó fél ne várakozzon és folytathassa tevékenységét.

Egy **accept** utasítást felhasználhatunk arra is, hogy a hívó tevékenységét szabályozzuk. Egy paraméter nélküli **accept** utasítással, melynek törzsében csak a **null** utasítás szerepel, elérhetjük, hogy a hívó taszk csak azután folytassa tevékenységét, hogy a hívott fél a randevút fogadta. Egy *T* taszk **accept** utasítása csak a *T* taszk törzsében szerepelhet, nem szerepelhet alprogramban, csomagban, vagy olyan taszkban, amely be van építve a *T*-be.

Lényeges megjegyezni az eljáráshívás és a randevú közötti különbségeket. Eljáráshívás esetén a hívott eljárás törzse a hívó eljárás kódjának részévé válik. Az eljárás helyi változói megfelelő lokális változókként jönnek létre a hívóban. Egyszerre több folyamat is végrehajthatja ugyanazt az eljárást, vagyis ugyanazon eljárásnak egyszerre több példánya is lehet aktív. Randevú esetén két aktív folyamat kapcsolódik össze. Az interakció szinkron szimmetrikus módon megy végbe, azaz akkor, ha mindkét fél készen áll rá. A taszktörzs deklarációs részében definiált objektumokra a kölcsönös kizárás automatikusan biztosított.

14.2.8. Belépésipont-családok

Az **entry** deklaráció bevezetheti a belépési pontok családját is. Egy **entry**-család tulajdonképpen egy belépési pontokból álló tömb. Ezeket egyetlen belépési pontként írhatjuk meg és, hasonlóan egy tömb elemeihez, a belépési pontokat a tömb indexe különbözteti meg egymástól. A családot diszkrét tartománnyal jelöljük ki. A család egy **entry**-jére indexeléssel

hivatkozhatunk. A törzsön kívül az **entry**-nevekre minősített formában kell hivatkozni, ahol az előtag a taszk objektum neve. A belépési pontok családjának alkalmazásával növelhetjük a belépési pontok számát és vele együtt a várakozási sorok számát. Egy várakozási soron belül csak érkezési sorrendben fogadhatók a hívások, több várakozási sor esetén azonban ettől eltérő kiszolgálási sorrend is lehetséges. Egy belépésipont-családot a következőképpen deklarálhatunk:

entry *Család(Diszkrét_típus) (Paraméterek);*

Majd a *Család* belépési pont meghívása a következőképpen történik:

Család(Kifejezés);

(A *Kifejezés* értéke a *Diszkrét_típus*-hoz kell tartozzon.)

14.2.9. A select utasítás

A randevú lehetővé teszi két taszk kommunikálását, de ez a módszer kissé nehézkes, mert egy taszk addig vár, amíg egy másik taszk nem fogadja hívását és fordítva, ha egy taszk fogadhat egy hívást, akkor addig vár, amíg egy másik taszk meghívja őt.

A **select** utasítás megkönnyíti a randevút, mely az **accept**, **delay** és **terminate** utasításokkal együtt lehetővé teszi, hogy egy taszk egy adott ponton többféle hívást várjon és fogadjon. Ugyanakkor különböző feltételek segítségével a hívásokat szabályozni is lehet. A **select** utasítás biztosíthatja azt is, hogy egy hívott taszk automatikusan lezáruljon, valamint azt is lehetővé teszi, hogy egy taszk folytassa tevékenységét, ha bizonyos idő elteltével nem jön létre a randevú.

Ezáltal a **select** az **accept**, **delay** és **terminate** utasításokkal együtt különféle randevúk megoldására alkalmazható, például taszkok szelektív hívására, szelektív várakozásra, feltételes **entry** utasításra és időhöz kötött belépésipont-hívásra.

14.2.10. Szelektív hívásfogadás

Tegyük fel, hogy szavakat és számokat akarunk kiírni a képernyőre. Ahhoz, hogy a betűk ne keveredjenek össze a számjegyekkel, szükségünk van arra, hogy a számot, illetve a karakterláncot kiíró taszkoknak kölcsönös kizárást biztosítsunk a képernyőre való kiírásnál:

```
task Buffer is
   entry Szám(I: in Integer);
   entry Karakterlánc(K: in String);
end Buffer;
```

A szelektív várakozással ez úgy oldható meg, hogy a **select** utasításba írt **accept** ágak közé **or**-t teszünk.

```
task body Buffer is
begin
   loop
      select
         accept Szám(I: in Integer) do
            put(Integer'Image(I));
         end Szám;
      or
         accept Karakterlánc(K: in String) do
            put(K);
         end Karakterlánc;
      end select;
      new_line;
   end loop;
end Buffer;
```

A taszk **select** utasításában több **accept** ág lehet, a különböző alternatívákat az **or** választja el egymástól, így egyszerre kettőnél több belépési pontján érkező hívásokra is tud figyelni.

A fenti példában a taszk, amikor a **select** utasításhoz ér, megvárja, hogy valamelyik **entry** ága meghívódjon. Ha egy taszk meghívta valamelyik **entry** pontját, akkor a neki megfelelő **accept** alternatívát hajtja végre. Ha a hívás hamarabb történt, akkor a hívó megvárja, amíg a hívott taszk a **select** utasításhoz ér.

Ha a **select** utasításban szereplő belépési pontok közül egynél többhöz tartozik nem üres várakozási sor, akkor a végrehajtásra kerülő **accept** utasítás kiválasztása nem determinisztikus.

14.2.11. Feltételes hívásfogadás

A feltételes hívásfogadás esetén a különböző **select** alternatívák végrehajtását bizonyos feltételekhez köthetjük, úgynevezett őrfeltételekkel egészíthetjük ki. Ha a **select** utasítás valamelyik ága elé a **when** *feltétel* => szerkezetet illesztjük, akkor szabályozhatjuk, hogy az utasítás az adott ágon milyen esetben fogadjon hívást.

Egy ágat *nyitottnak* nevezünk, ha nincs benne őrfeltétel (**when**), vagy ha a feltétel értéke igaz, különben *zártnak* nevezzük.

A **select** utasítás végrehajtásának kezdetén egyetlen egyszer értékelődnek ki a belépési pontok feltételei. Feltételes hívásfogadás esetén egy taszk csak azokat a hívásokat tudja fogadni, amelyeknek egy nyitott ág felel meg, zárt ág nem választható ki a **select** utasítás végrehajtásánál. Értelemszerűen az őrfeltételek ismételt kiértékelésére akkor kerülhet sor, ha a **select** utasítás egy ciklus magjában helyezkedik el. Ha minden ág zárt, *Program_Error* kivétel lép fel.

14.2.12. A terminate alternatíva

A taszkok többségénél a **select** utasítást egy **loop** utasítás tartalmazza, ami azt jelenti, hogy ha nincs **terminate** ág a taszk törzsében, akkor a taszk soha nem ér véget, nem adja át a vezérlést egy másik programegységnek. Ezáltal az eljárás, amelyben a taszk objektumot deklaráltuk, szintén soha nem ér véget. A **select** utasítás **terminate** alternatívája biztosítja, hogy a taszk törzsének végrehajtása lezárul. A **terminate** alternatívát használhatjuk feltétellel (**when** ⟨*feltétel*⟩ => **terminate**;) vagy pedig feltétel nélkül.

Egy taszk, miután fogadta minden taszk hívását a taszk specifikációjában megadott belépési pontok szerint, azaz minden **entry** várakozási sora kiürült és minden taszk lezárult, végrehajtja a **terminate** alternatívát.

```
select
   accept A1(X: in Üzenet) do
      -- az X üzenetet írjuk
   end A1;
or
   accept A2(Y: out Üzenet) do
      -- az Y üzenetet olvassuk
   end A2;
or
   terminate;
      -- a taszk véget ér, ha se nem olvasunk, se nem írunk üzenetet
end select;
```

14.2.13. A delay és az abort utasítás

A **delay** *t;* utasítás felfüggeszti a taszkot legalább *t* másodpercre. Az Adában maga a főprogram is egy taszk, ezért a **delay** utasítás használható taszkon kívül is, ekkor a főprogramot függeszti fel az adott időre. Az idő megadására az Ada bevezette a valós *Duration* típust, az idő kezelésére pedig a *Calendar* csomagot. A **delay** *t*-nek a **select** utasításon belüli használata lehetővé teszi, hogy amennyiben *t* időn belül semmilyen ág sem hajtható végre, akkor a taszk a **delay** után megadott utasításokat végezze el. Az időt a **select** utasítás megkezdésétől számítja. A **delay** alternatívát is valamilyen feltételhez köthetjük.

Az **abort** *T;* utasítás *abnormális* állapotba hozza a megnevezett taszkot. Ebben az állapotban a taszk már nem vehet részt randevúban. Egy megszakított taszk lezárult (komplett) állapotba kerül.

14.2.14. Az else alternatíva

A **select** utasítás egy másik alternatívája az **else** ág. A **select** utasítás az **else** ágon levő utasításokat akkor hajtja végre, ha a taszk semmilyen belépésipont-hívást nem tud kiszolgálni

azonnal (azaz a hívás megtörtént, de a neki megfelelő **accept** ág nem fogadóképes vagy létezik nyitott ág, de nincs neki megfelelő hívás).

A szelektív várakoztatásnak legalább egy olyan ága kell legyen, amely **accept** utasítást tartalmaz, ezen kívül lehet egy olyan ága, amely **delay**, **terminate** vagy pedig **else** utasítást tartalmaz.

A **select** utasítás végrehajtásakor kiértékelődnek a feltételek és így kiderül, hogy mely ágak nyitottak. Ezután végrehajtódik egy nyitott ág, vagy ha ilyen nincs, az **else** ág. Ezzel az utasítás végrehajtása befejeződik. Ha több nyitott ág is van, a választás a következő szabályok szerint történik:

- ha van nyitott **accept** ág, amelynél randevú jön létre, az hajtódik végre;
- ha nincs nyitott **accept** ág, akkor egy nyitott **delay** ág választódhat ki;
- az **else** ág csak akkor választódhat ki, ha mindegyik ág zárt;
- a **terminate** ág nem választódhat ki addig, amíg van a taszknak nem üres **entry** sora.

Példa a szelektív várakoztatásra:

```
task body Erőforrás is
   Foglalt: Boolean := False;
begin
   loop
      select
         when not Foglalt => accept Lefoglal do
            Foglalt := True;
         end Lefoglal;
      or
         accept Felszabadít do
            Foglalt := False;
         end Felszabadít;
      or
         terminate;
      end select;
   end loop;
end Erőforrás;
```

Az egymástól különböző belépési pontokhoz tartozó **accept** utasítások egymásba ágyazhatók, illetve egy **accept** utasítás törzse tartalmazhat belépésipont-hívást (*egymásba ágyazott randevú*).

Egy adott belépési ponthoz tartozó randevú befejeződéséig ugyanezen belépési ponthoz tartozó újabb randevú azonban nem jöhet létre (azonos belépési ponthoz tartozó **accept** utasítások nem ágyazhatók egymásba). A belső randevúra esetleg nagyon sokáig kell várni, ez megakadályozza a külső randevú befejeződését (*rejtett várakozás*). A belső randevú mindig feltételes legyen, de jobb, ha nincs.

14.2.15. A feltételes entry hívás

Az eddigi fejezetekben azt mutattuk be, hogy egy taszk hogyan tud szelektív módon válaszolni az **entry**-hívásokra, egy taszk azonban szelektív módon meg is hívhat más taszkokhoz tartozó belépési pontokat.

Ha egy belépésipont-hívás fogadására nem akarunk tetszőlegesen hosszú ideig várakozni, akkor *feltételes* vagy *időhöz kötött belépésipont-hívást* alkalmazunk. A feltételes belépésipont-hívás ciklusba szervezése azonban hatékonysági okokból nem kívánatos (busy waiting). Egyszerre egynél több belépési pont hívása nem megengedett, helyette *lekérdezés* alkalmazható (polling).

A feltételes **entry** hívás esetén a **select** utasításnak van egy **entry** hívást tartalmazó ága és egy **else** ága. Ebben az esetben **entry** hívás csak akkor történik, ha a randevú azonnal létre is jön. Nem valósul meg az **entry** hívás, ha a randevú nem jöhet létre azonnal (például mert a taszk végrehajtása még nem tart az **accept** utasításnál, vagy az **accept** ága nem nyitott, vagy az illető **entry**-hez tartozó várakozási sor nem üres). Ha a hívás nem történik meg, helyette az **else** ág hajtódik végre.

```
procedure Próba(R: in Erőforrás) is
begin
   loop
      select R.Lefoglal;
      else null; -- várakozás Foglalt = True -ra
      end select;
   end loop;
end Próba;
```

14.2.16. Időhöz kötött entry hívás

Az időhöz kötött **entry** hívásnál a **select** utasításnak két alternatívája van: egy **entry** hívást és egy **delay** utasítást tartalmazó ág. A hívás akkor következik be, ha a randevú még a **delay** utasításban megadott idő lejárta előtt létrejön. Ha az adott idő alatt a randevú nem jön létre, akkor a **delay** ág hajtódik végre.

```
select
   put_line("Legfeljebb 4 másodpercet várok!");
or
   delay 4.0;
end select;
```

14.2.17. Kivételkezelés

A kivételes esetek továbbterjedésének szabályai:

- Ha hiba keletkezik taszk indításakor, akkor *Tasking_Error* váltódik ki a szülőben.
- Ha a belépésipont-hívás abnormális, terminált vagy komplett taszkra történik, akkor *Tasking_Error* keletkezik a hívóban.
- Egy kivételes eset egy randevú belsejében, amelyet nem kezeltek a randevú végéig, kivételes eset lesz a hívóban és a fogadóban is.

14.2.18. Példák

Kölcsönös kizárás

Párhuzamos rendszerekben szükség van olyan eszközre, amely biztosítja, hogy amikor egy közös objektumot használ egy taszk, addig a többi taszk ne férjen hozzá. Az Adában a kölcsönös kizárást többféleképpen, és nagyon egyszerűen meg lehet valósítani.

Először az osztott objektumot magában a taszkban helyezzük el. Itt *T* egy korábban meghatározott típus, *Osztott* pedig az osztott objektum neve.

```
task Osztott_Változó is
   entry Módosít(X: in T);
   entry Olvas(X: out T);
end Osztott_Változó;

task body Osztott_Változó is
   Osztott: T;
begin
   loop
      select
         accept Módosít(X: in T) do
            Osztott := X;
         end Módosít;
      or accept Olvas(X: out T) do
            X := Osztott;
         end Olvas;
      or terminate;
      end select;
   end loop;
end Osztott_Változó;
```

Az osztott objektum lehet globális objektum is, ilyenkor a kölcsönös kizárást szemafor használatával valósíthatjuk meg. Egy szemaforhoz egy *P* és egy *V* művelet tartozik. A *P* művelet csak az első őt meghívó taszkot engedi tovább, a többit várakoztatja, a *V* művelet pedig mindig újraindít egy *P* által felfüggesztett taszkot.

```
task type Szemafor is
   entry P;
   entry V;
end Szemafor;

task body Szemafor is
begin
   loop
      accept P;
      accept V;
   end loop;
end Szemafor;
```

Az osztott objektum használata előtt tehát mindig meg kell hívni a *P* műveletet, amikor pedig befejeződtek az osztott objektumon végzett műveletek, akkor meg kell hívni a *V* műveletet, ezzel az objektum használatára vonatkozóan a kölcsönös kizárást biztosítottuk.

```
   Osztott: T;
   S: Szemafor;
begin
   ...
   S.P;
   Osztott := F(Osztott);
   S.V;
   ...
```

A szemafor nem túlságosan megbízható szinkronizációs eszköz, ha ugyanis valahol kifelejtjük a *V* művelet meghívását, ez teljesen összezavarja a program működését.

Termelő-fogyasztó feladat

A termelő–fogyasztó feladatokban kétfajta taszk szerepel: az egyik, ami bizonyos típusú adatokat előállít, a másik pedig az, amelyik feldolgozza ezeket.

```
with Osztott_Sor;
package Int_Osztott_Sor is new Osztott_Sor(Integer, 128);

task body Termelő is
   C: Integer;
begin
   loop
      -- C előállítása
      Int_Osztott_Sor.Sor.Betesz(C);
   end loop;
end Termelő;

task body Fogyasztó is
   C: Integer;
begin
   loop
      Int_Osztott_Sor.Sor.Kivesz(C);
      -- C felhasználása
   end loop;
end Fogyasztó;
```

A még fel nem dolgozott adatokat egy sorban kell tárolni, s ehhez a sorhoz mind a két fajta taszknak hozzá kell tudni férni, sőt, a hozzáférésre biztosítani kell a kölcsönös kizárást. Egy ilyen sor kezelését mutatjuk meg az alábbi példában.

```
generic
   type Elem is private;
   Sor_Hosszúság: Natural;
package Osztott_Sor is
task Sor is
   entry Kivesz (D: out Elem);
   entry Betesz (D: in Elem);
end Sor;
end Osztott_Sor;

package body Osztott_Sor is
task body Sor is
   type P is new Natural range 0 .. Sor_Hosszúság;
   subtype Aktuális_Tároló is P range 1 .. P'Last;
   Tároló: array (Aktuális_Tároló) of Elem;
   Benne: P := 0;
   Inx, Outx: Aktuális_Tároló := 1;
begin
   loop
      select
         when Benne > 0 =>
            accept Kivesz (D: out Elem) do
               D := Tároló(Outx);
            end Kivesz;
            Outx := Outx mod P'Last + 1;
            Benne := Benne - 1;
         or
         when Benne < P'Last =>
            accept Betesz (D: in Elem) do
               Tároló(Inx) := D;
            end Betesz;
            Inx := Inx mod P'Last + 1;
            Benne := Benne + 1;
         or terminate;
      end select;
   end loop;
end Sor;
end Osztott_Sor;
```

Védett objektumok

A védett objektumok [Nyé 98] olyan adatokat tartalmaznak, amelyekhez a taszkok csak a védett műveletek segítségével férhetnek hozzá. A védett adatokat a **protected** típus **private** részében helyezhetjük el.

Három fajta védett műveletet írhatunk egy védett objektumba: védett belépési pontokat, védett eljárásokat és védett függvényeket. A védett belépési pont megegyezik egy taszk őrfeltétellel ellátott belépési pontjával. Ha az őrfeltétel igaz, akkor a hívó taszk végrehajthatja a belépési ponthoz tartozó törzset. Amennyiben az őrfeltétel hamis, akkor ugyanúgy, mint a taszkoknál, a hívás bekerül a belépési pont várakozási sorába, amíg az őrfeltétel igazzá válik. A védett eljárásnak írási-olvasási joga van a **private**-ben meghatározott változókra. Amikor egy taszk egy védett eljárást hív meg, semmilyen más taszk nem férhet hozzá a védett változókhoz, azaz a védett objektum kölcsönös kizárást biztosít a **private** változói számára. A védett függvénynek csak olvasási joga van, a védett függvényt egyszerre több taszk is meghívhatja.

A belépési pont abban különbözik az eljárástól, hogy van várakozási sora, illetve őrfeltételt kell hozzárendelni.

Védett objektumok nagyon sok feladat megoldására használhatók kényelmesen, segítségükkel szinkronizációs szerkezeteket építhetünk fel. Az alábbi példa bemutatja, hogyan készíthetünk védett objektumok segítségével egy bináris szemafort.

```
protected type Erőforrás is
   entry Lefoglal;
   procedure Elenged;
private
   foglalt: Boolean := False;
end Erőforrás;

protected body Erőforrás is
   entry Lefoglal when not foglalt is
      begin
         foglalt := True;
      end Lefoglal;
   procedure Elenged is
      begin
         foglalt := False;
      end Elenged;
end Erőforrás;
```

A következőképpen használhatjuk a szemafort:

```
Nyomtató: Erőforrás;
begin
   Nyomtató.Lefoglal;
   Nyomtat;
   Nyomtató.Elenged;
end program;
```

Védett egység *belépésipont-családot* is meghatározhat. A következő példa azt mutatja be, hogyan használhatók a belépésipont-családok prioritások kezelésére. A *Kijelző* mindig a legfontosabb üzenetet írja ki:

```
type Prioritás is (Magas, Közepes, Alacsony);
protected Kijelző is
   entry Üzenet (Prioritás) (Szöveg: in String);
   function Mehet (Aktuális: Prioritás) return Boolean;
end Kijelző;

protected body Kijelző is
   entry Üzenet (for Aktuális in Prioritás) (Szöveg: in String)
      when Mehet(Aktuális) is
   begin
      Put_Line(Prioritás'Image(Aktuális) & Ascii.HT & Szöveg);
   end Üzenet;

   function Mehet (Aktuális: Prioritás) return Boolean is
      Fontosabb: Prioritás := Prioritás'Last;
   begin
      while Fontosabb > Aktuális loop
         if Üzenet(Fontosabb)'Count > 0 then return False; end if;
         Fontosabb := Prioritás'Pred(Fontosabb);
      end loop;
      return True;
   end Mehet;
end Kijelző;
```

A taszkok az alábbi módon küldhetnek üzenetet a kijelzőre:

```
Kijelző.Üzenet(Magas)("Nekem van igazam!");
```

A belépésipont-család minden tagjához külön várakozási sor tartozik.

Ugyanúgy, mint ahogy lehet taszk objektumokat és taszk típusokat írni, készíthetünk *védett objektumokat* és *védett típusokat* is. A védett típusokból statikusan (változódeklarációval) és dinamikusan (allokátorral) is létrehozhatunk védett objektumokat. A védett típusok megadása formailag abban különbözik a védett objektumokétól, hogy a **protected** kulcsszó után szerepel a **type** kulcsszó is.

A védett típusok deklarációjában szerepelhetnek diszkriminánsok is. A lehetőségek ugyanazok, mint a taszkok és a rekordok esetében: diszkriminációs diszkrét típus és mutatódefiníció lehet. A következő példában a *Szemafor* típusnak egy diszkriminánsa is van:

```
protected type Szemafor ( Max: Positive := 1 ) is
   entry P;
   procedure V;
private
   Nr: Natural := 0;
end Szemafor;          -- A program a következő oldalon folytatódik.
```

```
   -- Folytatás az előző oldalról.
protected body Szemafor is
   entry P when Nr < Max is
   begin
      Nr := Nr + 1;
   end P;

   procedure V is
   begin
      Nr := Nr - 1;
   end V;
end Szemafor;
```

14.3. CSP

A CSP (Communicating Sequential Processing) nyelvet C. A. R. Hoare írta le 1978-ban [Hoa 78]. A CSP a bemeneti és kimeneti utasításokat a programozás alapelemeinek tekinti (éppúgy mint például az értékadást), a szekvenciális programrészek párhuzamos kompozícióját pedig a programkészítés egy alapvető eszközének. Igazából a CSP ebben a formában nem kezelhető programnyelvként, inkább csak modellként, hiszen csak a párhuzamosság szempontjából fontos elemeket tartalmazza és nem foglalkozik például a megvalósítás problémáival. Ennek ellenére érdemes vele foglalkoznunk, mivel több későbbi nyelv (például az Occam) tervezésére hatással volt.

A CSP a folyamatok kommunikációira a bemeneti és kimeneti utasítások egyszerű formáit vezeti be. A bemeneti műveletet ?-lel jelöljük. Például a

```
f2 ? uz()
```

bemeneti műveletet végrehajtó folyamat az `f2` folyamattól egy `uz()` üzenet érkezését várja. Ennek párja a kimeneti művelet, melyet !-el jelölünk. Például a

```
f1 ! uz()
```

műveletet végrehajtó folyamat az `f1` folyamatnak egy `uz()` üzenetet küld egy absztrakt kommunikációs csatornán keresztül. Kommunikáció úgy jöhet létre, hogy az egyik folyamat megnevez egy másikat, mint üzenetének célját (kimeneti művelet), a megnevezett folyamat pedig az előzőt, mint lehetséges feladót (bemeneti művelet). Azaz például az előbbi példában említett bemeneti műveletet az `f1` folyamat hajtja végre, a példaként leírt kimeneti műveletet pedig az `f2` folyamat.

A kommunikáció szinkron, az elsőnek kommunikálni kívánó folyamatnak meg kell várnia a másodikat. A kommunikációnál mintaillesztés történik. A küldött, illetve fogadott kifejezések csak akkor illeszkednek, ha konstruktoraik neve rendre megegyezik, azaz azonos típusúak. Mindkét előbbi példában az `uz()` üres kifejezéslistával rendelkező összetett változó szerepel, melynek konstruktora az `uz` azonosító.

A bemeneti vagy kimeneti utasítás sikertelen, ha a partnerként megnevezett másik folyamat már befejeződött. Hasonlóan problémát okoz, ha a kimenetként megadott kifejezés definiálatlan. Például, ha a

```
x ! (3*a+5)
```

kifejezésben az a definiálatlan.

Nézzük ezeken kívül még milyen utasítások találhatók a CSP-ben. A nemdeterminisztikusság kezelésére szolgál az *őrfeltételes utasítás*, amely két részből áll, egy őrfeltételből és egy utasítássorozatból. Az őrfeltétel tartalmazhat logikai kifejezést és bemeneti utasításokat. Az őrfeltételes utasítás utasítássorozata akkor hajtódhat végre, ha az őrfeltétel értéke igaz. A bemeneti utasítást tartalmazó őrfeltétel akkor igaz, ha a benne szereplő logikai kifejezés igaz, és a forrásként megnevezett folyamat még nem fejeződött be és kész megfelelő üzenet küldésére. Az őrfeltételekben kimeneti utasítás nem szerepelhet. Például a

```
f ? uz1() ⟶ f ? uz2()
```

utasítás azt jelenti, hogy ha az `f` folyamat fogadott egy `uz1` üzenetet, akkor ezek után egy `uz2` üzenetet fog fogadni. Az első üzenet fogadása az őrfeltétel a második üzenet fogadásához.

A *párhuzamos utasítás* (`| |`) az alkotó szekvenciális utasítássorozatok párhuzamos, egyidejű végrehajtását definiálja. A folyamatok egyszerre indulnak, és a párhuzamos utasítás akkor ér véget, ha minden alkotó folyamat befejeződött. A folyamatok függetlenek, nem használhatnak közös változókat, pontosabban egyik sem hivatkozhat olyan változóra, amely értékadás bal oldalán jelenik meg bármely más folyamatban.

Az *alternatív utasítás* (`[ ]`) véges számú őrfeltételes utasításból épül fel. Az alternatív utasítás végrehajtásakor azon őrfeltételes utasítások közül, melyek őrfeltétele igaz, tetszőlegesen kiválasztódik egy, és az ahhoz tartozó utasítások hajtódnak végre. Ha egyetlen őrfeltétel sem igaz a végrehajtáskor, az utasításnak nincs semmilyen hatása. Ha az utasítás összes nem hamis őrfeltétele olyan bemeneti utasításokat tartalmaz, amelyek forrása ugyan nem kész a megfelelő üzenet küldésére, de még nem terminált, az alternatív utasítás végrehajtása felfüggesztődik.

Az *iteratív utasítás* (`*`) alternatív utasításokból épül fel. Addig hajtódnak végre egymás után az alternatív utasítások, ameddig az összes őrfeltétel hamissá nem válik. Ekkor az iteratív utasítás terminál.

Az utasítássorozatok címkével láthatók el. Ha a címke együtt jár egy tartománnyal, például

```
f(i:0..4)::F
```

akkor `F` folyamatok sorozata indul el egyidőben, mindegyik azonos kóddal, de az `i` változót a megfelelő sorszámmal helyettesítve.

Végül nézzük meg, hogyan fest az étkező filozófusok problémája CSP-ben. A feladat szerint van öt filozófusunk, akik gondolkodással és étkezéssel töltik a napjaikat. A filozófusoknak közös étkezőszobájuk van, kör alakú asztallal. Az asztal körül öt szék áll. Minden filozófusnak megvan a saját széke, amit használhat. Az asztal közepén van egy nagy tál

spagetti és öt darab villa, minden két hely között egy-egy. Ha egy filozófus megéhezik, akkor bemegy a szobába, leül a saját székére és felveszi a bal oldalán fekvő villát. A spagetti elfogyasztásához viszont szüksége van a másik mellette fekvő villára is, így azt is felveszi. Amikor végzett az evéssel, visszateszi a villákat és elhagyja a szobát. Mindig nyilvántartjuk, éppen hány filozófus van a szobában.

```
FIL    = *[ ... amíg az i. filozófus él ... ⟶
            GONDOLKODIK;
            szoba!belép();
            villa(i)!felvesz();villa((i+1) mod 5)!felvesz;
            ESZIK;
            villa(i)!lerak();villa((i+1) mod 5)!lerak();
            szoba!kilép()
          ]
VILLA = *[ fil(i)?felvesz() ⟶ fil(i)?lerak()
         [] fil((i-1) mod 5)?felvesz() ⟶ fil((i-1) mod 5)?lerak()
          ]
SZOBA = létszám : integer; létszám := 0;
        *[ (i:0..4)létszám<4 ⟶ (fil(i)?belép()⟶létszám:=létszám+1)
         [] (i:0..4)fil(i)?kilép() ⟶ létszám:=létszám-1
          ]
[ szoba::SZOBA || villa(i:0..4)::VILLA || fil(i:0..4)::FIL ]
```

A fenti megoldásnál a SZOBA folyamatnál a belépés előfeltétele az, hogy a létszám változó értéke, azaz a szobában lévők száma kisebb legyen négynél. Ezzel kizárjuk azt a lehetőséget, hogy az összes filozófus egyszerre lépjen be a szobába és vegye fel a bal oldali villáját, majd hiába várakozzon arra, hogy felvehesse a jobb oldali villáját. Ez egy tipikus holtponthelyzet lenne, azaz olyan helyzet, ahol a folyamatok körkörösen egymásra várakoznak.

Hoare megalkotta az itt ismertetett nyelv egy újabb változatát is 1985-ben [Hoa 85], amely csak egy-két pontban tér el a korábbitól.

14.4. Occam

Az Occam nyelvet pontosan azzal a céllal hozták létre, hogy segítségével könnyen írhassanak konkurens algoritmusokat. A nyelvben egy alkalmazás úgy képzelhető el, mint folyamatok gyűjteménye, mely folyamatok konkurensen hajtódnak végre [Bar 92].

Mint már említettük a párhuzamos programozás világában az egyik legfontosabb feladat a folyamatok közötti kommunikáció megvalósítása. Az Occam nyelvben a kommunikáció csatornákon keresztül történik, méghozzá olyan csatornákon keresztül, melyek egyirányúak és nem használnak átmeneti tárat. A kommunikáció típusát, azaz hogy milyen típusú elemek haladhatnak keresztül a csatornán, a csatornaprotokoll határozza meg, melyet a csatorna deklarálásakor adhatunk meg. A fentiek az Occam nyelv erősen típusos voltából következnek. A kommunikációt két alapvető művelet segítségével valósíthatjuk meg. Az egyik a kimeneti művelet, mellyel valamilyen kifejezés által meghatározott értéket rakhatunk a csatorna egyik oldalára. A kimeneti művelet addig várakozik, amíg egy megfelelő bemeneti művelet le nem veszi az értéket a csatornáról. A művelet általános alakja:

```
csatorna ! kifejezés
```

A másik a bemeneti művelet, mellyel levehetünk egy elemet a csatorna másik oldaláról, és berakhatjuk azt egy megfelelő típusú változóba. A bemeneti művelet addig várakozik, amíg le nem tud venni egy elemet. A művelet általános alakja:

```
csatorna ? változó
```

Vizsgáljuk meg egy kicsit részletesebben az átküldéskor használható típusokat és a csatornaváltozók deklarációját. Az alaptípusok a `BOOL`, `BYTE`, `INT16` (-32768-tól 32767-ig), `INT32` (-2^{31}-től $2^{31} - 1$-ig), `INT64` (-2^{63}-tól $2^{63} - 1$-ig), `REAL32`, `REAL64` (32, illetve 64 biten ábrázolt valós számok). Occam-ban például egy `INT32` típusú változót a következőképpen deklarálhatunk:

```
INT32 x:
```

Ennek megfelelően egy egyszerű csatorna deklarációja, melyen 32 bites valós számokat küldhetünk át a következőképpen alakul (nevezzük a csatornaváltozót `in`-nek):

```
CHAN OF REAL32 in:
```

Az Occam nyelv legújabb változatában már létrehozhatunk bonyolultabb adatszerkezeteket is, például tömböket, literálokat, rekordokat. Így például olyan csatornát is deklarálhatunk, melyen olyan szöveget szeretnénk átküldeni, melyek hosszát is csak a küldéskor akarjuk meghatározni. Ekkor olyan csatornát hozunk létre, melyen egyszerre lehet átküldeni egy egész értéket és egy tömböt:

```
CHAN OF INT::[]BYTE in:
```

Ilyenkor a kimeneti művelet például a következő lehet:

```
in ! 5 "Küldendő szöveg."
```

a bemeneti pedig például a következő alakú lehet:

```
in ? hossz::tmb
```

A fenti esetben a `hossz` változóba az 5 érték, míg a `tmb` változóba a `"Külde"` karakterlánc kerül (az elküldött tömb első 5 értéke).

Miután megoldottuk a kommunikációt, nézzük, hogyan érhetjük el, hogy az egyes folyamatok egymással párhuzamosan fussanak. Ehhez először vizsgáljuk meg, hogy egyáltalán hogyan kapcsolhatunk össze több folyamatot egy futási egységgé. Erre a legegyszerűbb módszer a `SEQ` kulcsszóval való összekapcsolás, amely szekvenciális végrehajtást eredményez:

```
SEQ
   screen ! '?'
   keyboard ? char
   screen ! char
   screen ! cr
   screen ! lf
```

A program beolvas egy karaktert és kiírja azt, valamint egy újsort a képernyőre.

Ennél számunkra most érdekesebb a PAR kulcsszóval történő összekapcsolás, amely párhuzamos végrehajtást eredményez:

```
WHILE next <> eof
  SEQ
    x := next
    PAR
      in ? next
      out ! x*x
```

A példaként leírt program először értékül adja az x változónak a next változó értékét, majd párhuzamosan beolvassa az in csatornáról a next változó új értékét, és kiírja az out-ra az x*x-et, és mindezt addig csinálja, míg a next változóba fájlvége jel nem kerül.

Párhuzamos végrehajtásnál azok a változók, amelyek valamelyik folyamatban értéket kapnak (értékadásban vagy bemeneti utasításban) nem szerepelhetnek egyetlen másik párhuzamos folyamatban sem. Ezért az alábbi példa hibás:

```
PAR                     -- Ez a kompozíció hibás.
  SEQ
      hiba := 42        -- A hiba változó egynél több...
      c ! 42
  c ? hiba              -- ... párhuzamos folyamatban szerepel.
```

Hasonlóan, egy csatornát, amelyet az egyik folyamatban egy bemeneti műveletben használunk nem használhatunk másik párhuzamos folyamat bemeneti műveletében. Ugyanez igaz a kimeneti műveletben használt csatornaváltozókra is. Így az alábbi példa is *hibás*:

```
PAR                     -- Ez a kompozíció hibás.
  c ! 0                 -- A c csatornát több folyamat...
  SEQ
    c ? x
    c ? y
  c ! 1                 -- ... kimeneti műveletében is használjuk.
```

Párhuzamos végrehajtás esetén gyakran nagyon hasonló folyamatokat akarunk egymással párhuzamosan futtatni. Ilyenkor sokszor egyszerűbben is leírhatjuk a PAR szerkezetet. A példában leírt két szerkezet egyenértékű egymással.

```
PAR i = 3 FOR 4
    user[i] ! message
```

```
PAR
    user[3] ! message
    user[4] ! message
    user[5] ! message
    user[6] ! message
```

Érdekes vezérlési szerkezet még az ALT, amely a CSP alternatív utasításának megfelelője. Például ezt a szerkezetet használhatjuk, ha két bemeneti csatornánk van és a kimeneti csatornára küldött adat attól függ, melyiken kaptunk adatot.

```
ALT
  ⟨logikai feltétel⟩ & in1 ? next
        out ! x*x
  in2 ? next
        out ! x+5
```

Természetesen, mint látható, a CSP-hez hasonlóan itt is állhat logikai feltétel is az őr-feltételben. Ha az alternatívák hasonlók, akkor a PAR-hoz hasonlóan az ALT szerkezetet is leírhatjuk egyszerűbben:

```
ALT i=1 FOR 2
  in[i] ? next
        out ! next
```

A fenti példa egyenértékű a következővel:

```
ALT
  in[1] ? next
        out ! next
  in[2] ? next
        out ! next
```

14.5. Java

A Java nyelvben [Nyé 01] két lehetőségünk van szálak létrehozására. Az elsőben egy olyan osztályt használunk, ami a *java.lang.Thread* osztályt (a Java szálakat megvalósító osztályát) terjeszti ki. Amikor egy ilyen osztályból példányosítunk, kapunk egy szálat, ami viszont még nem fog futni. Ehhez meg kell hívnunk a *Thread* osztályból örökölt *start()* metódust. Ez ad kezdeti értéket a szálunknak és indítja el annak *run()* metódusát, ami tulajdonképpen a szál főprogramja. A *Thread* osztály *run()* metódusa eredetileg üres, a leszármazott osztály *run()* metódusát felül kell definiálnunk a megfelelő működés érdekében.

```
class UjSzal extends Thread {
  public void run() {
    System.out.println("Ez a" + getName() + "nevű példaszál.");
  }
}
public class ThreadPelda {
  static public void main(String args[]){
      UjSzal x, y;
      x = new UjSzal();
      y = new UjSzal();
      x.start();
      y.start();
  }
}
```

Ennek a módszernek van egy nagy hátránya is, miszerint a *Thread* osztályt kell kiter jeszteni, a Java pedig nem támogatja a többszörös öröklődést, azaz csak egy osztályból szár maztathatunk egyszerre. Ez például appletek készítésekor jelent problémát, hiszen ilyenko az *Applet* osztályt mindenképpen ki kell terjesztenünk, így nem örökölhetünk a *Thread* osztályból is.

Ez az oka a másik módszer bevezetésének, amely a *Runnable* interfész megvalósítás igényli. Az osztályokon kívül a nyelv további fontos elemei az interfészek. Az interfész egy új referencia típus, amely absztrakt metódusok deklarációjának és konstans értékeknek a összessége. Az osztályokhoz képest itt hiányoznak a valódi, változtatható adattagok és metódusok megvalósításai. Egy osztály megvalósít egy interfészt, ha az összes, az interfész által specifikált metódushoz megvalósítást ad. Ezáltal az absztrakt program konkréttá vá lik. Bárhol, ahol egy adott interfész szerepel típusként, az interfészt megvalósító bármilyen osztály példánya felhasználható. Fontos megjegyeznünk, hogy egy osztály tetszőleges számú interfészt megvalósíthat.

A *Runnable* interfész csak a *run()* metódust definiálja, ezen interfész megvalósításával olyan osztályokat határozhatunk meg, amelyek nem a *Thread* osztály leszármazottai, mégis szálként képesek futni.

A két módszer közötti fő különbség abban áll, hogy a *Runnable* interfészt megvalósító osztályok csak a *run()* metódushoz férnek hozzá, a többi *Thread* osztálybeli metódushoz nem. Amikor az interfészt használjuk a szál definiálásához, akkor is létre kell hozni egy *Thread* osztállyal, vagy egy alosztályával deklarált valódi szálat, csak a konstruktorban meg kell adni a *Runnable* objektumra vonatkozó hivatkozást. Így amikor a szál elkezd futni, a *Runnable* objektum *run()* metódusa kapja meg a vezérlést.

```
class UjSzal implements Runnable {
  public void run() {
     System.out.println("Ez a" + Thread.currentThread().getName()
                        + "nevű példaszál.");
  }
}

public class RunnablePelda {
  static public void main(String args[]){
        UjSzal usz;
        Thread x, y;

        usz = new UjSzal();
        x = new Thread(usz);
        y = new Thread(usz);
        x.start();
        y.start();
  }
}
```

A szálak csoportba (*ThreadGroup* osztály) szervezhetők, így egyszerre kezelhetők. Egy szál csak egy csoporthoz rendelhető, és a hozzárendelés nem változtatható meg. A szálcsoport újabb szálcsoportokat is tartalmazhat, így hierarchikusan egymásra épülve a csoportok faszerkezetet alakíthatnak ki. A *ThreadGroup* osztályban számos hasznos eljárás található. A *getThreadgroup()* eljárás segítségével lekérdezhetjük egy szál csoportját. Az *activeCount()* eljárással megtudhatjuk, hogy egy adott csoporthoz, illetve a faszerkezetben alatta elhelyezkedő csoportokhoz hány szál tartozik. A *getName()* művelettel megkaphatjuk a csoport nevét, a *getParent()* és a *parentOf()* eljárásokkal pedig feltérképezhetjük a csoportok közötti hierarchiát.

A Java 2-es változatától bevezettek egy új, érdekes osztályt, a *ThreadLocal* osztályt. Ezzel az osztállyal olyan változókat lehet meghatározni, melyek osztályszintű, statikus változók, de szálanként más-más lokális értéket tárolnak. Az ebből az osztályból, vagy ennek alosztályából példányosított **static** változó abban tér el a többi osztályszintű változótól, hogy minden szál, amely a *set()* vagy *get()* eljárásokkal hozzáfér, a saját lokális másolatát látja, illetve módosítja.

A szálak a *Thread* osztály *interrupt()* eljárásával szakíthatóak meg, a *ThreadGroup* osztály hasonló metódusával pedig a csoport összes futó szálát megszakíthatjuk. A *Thread* osztály *isInterrupted()* függvényével le is kérdezhetjük, hogy egy adott szál meg van-e szakítva.

Háttérben futó feladatok ellátására démon szálakat használhatunk. A *Thread* osztály *setDaemon()* eljárásával állíthatjuk be egy szálnál, hogy az démonként üzemeljen-e, vagy sem. A beállítást csak a szál elindítása előtt lehet elvégezni. Azt, hogy egy szál démon-e, az *isDaemon()* függvénnyel kérdezhetjük le. A démon szálak egyedül abban különböznek a többi száltól, hogy csak akkor állnak meg, ha az összes nem démon szál a programban befejezte működését.

A *join()* művelettel gondoskodhatunk arról, hogy egy szál megvárja, amíg egy másik befejezi a működését. Az eljárás addig vár, míg a megnevezett szál le nem áll, és csak ezután tér vissza, hogy a hívó szál folytathassa futását. Az eljárásnak megadható egy időparaméter is, hogy csak véges ideig várjon a másik szálra. Ha az adott idő alatt a másik szál nem áll le, a hívó szál automatikusan folytatódik. A művelet kiválthatja az *InterruptedException* kivételt, ha valaki megszakítja azt a szálat amire várunk, így a kivétel kezelésről gondoskodnunk kell.

Ha a szálak egyprocesszoros környezetben futnak, a párhuzamosságnak csak az illúzióját keltik, hiszen valójában ekkor egyszerre csak egy szál futhat. Ekkor valahogyan meg kell határoznunk, milyen sorrendben hajtjuk végre a szálakat. A Javában az a szál futhat előbb, melynek nagyobb a prioritása. Ha több azonos prioritású szál is van, akkor azok *Round-Robin*[1] módszer szerint futhatnak. Ezt bizonyos mértékben befolyásolhatjuk is, mivel a *Thread* osztály *yield()* eljárásának hatására a hívó szál a futási sor végére kerül, és az azonos prioritású szálak közül a következő kezd futni. Tíz különböző prioritási szint van, a *Thread.MIN_PRIORITY*-tól a *Thread.MAX_PRIORITY*-ig. Ezek értékei rendre 1 és 10; minél magasabb az érték, annál nagyobb a prioritás. A szálak prioritása indításuk előtt és futásuk alatt is beállítható a *setPriority()* eljárással, a prioritás értéke pedig lekérdezhető a *getPriority()* függvénnyel. A *ThreadGroup* osztálybeli *setMaxPriority* és *getMaxPriority* eljá-

[1] Ez azt jelenti, hogy „körkörösen" egymás után futnak, azaz a sorban utolsó után ismét az első következik.

rásokkal a csoportokra vonatkozóan is beállíthatjuk, illetve lekérdezhetjük a legmagasabb prioritási értéket.

A legtöbb többszálú programban létfontosságú a szinkronizáció megoldása. A Java szálak a monitor modellt használják a programkódok kritikus részeinek szinkronizálására. A monitor alapvetően zárolás, vagyis amikor a szál eljut a kritikus részhez, magához rendeli az ahhoz tartozó monitort. Ha ekkor egy másik szál is be akarna lépni ebbe a kritikus részbe, várnia kell addig, amíg a monitor újra szabad nem lesz. Egy szinkronizált metódusban levő programrészlet meghívhat egy másik metódust ugyanabban az objektumban blokkolás nélkül. Ugyanez érvényes a hívó szál objektumaira is, melyeket a szál blokkolt és még nem eresztette el őket. Azaz a monitorok újrahívhatóak (a kód újrabeléptethető, újrameghívható). A szinkronizáció megmarad, ha egy nem szinkronizált metódust hívunk meg a blokkból, illetve a nem szinkronizált metódusok konkurensen futhatnak bármikor.

A Java nyelvben megkülönböztetünk objektumszintű és osztályszintű monitort. Az objektumszintű monitor példányosítódik, ha az objektum legalább egy **synchronized** metódussal rendelkezik. Ha egy szál meghív egy ilyen metódust, akkor ugyanennek az objektumnak egyik **synchronized** metódusát sem hívhatja meg másik szál. Ha a szál egy ugyanilyen osztályhoz tartozó, de másik objektum **synchronized** metódusát hívja meg, akkor azt használhatja, hiszen a szál csak az adott objektumhoz tartozó monitort foglalja le. Az osztályszintű monitor példányosítódik egy osztály számára, ha az osztálymetódus definíciójában **static synchronized** módosító van. Ekkor az osztály összes objektuma ezen az osztályszintű monitoron osztozik az ilyen metódusok hívásakor.

Esetenként szükség van arra, hogy egy szál megvárja, míg egy másik elér munkájának egy adott pontjára. Ilyenkor kell alkalmaznunk a monitorokat kiegészítő feltételes változókat. A Java objektumokhoz is tartoznak ilyen feltételes változók. Ezeket nem érhetjük el közvetlenül, csak a *wait()*, *notify()* és *notifyAll()* eljárásokon keresztül. Ezeket az eljárásokat az *Object* osztály tartalmazza, tehát bárhol használhatjuk őket. Az egyetlen megkötés, hogy ezek az eljárások csak **synchronized** eljárásban, vagy blokkban szerepelhetnek.

Az a szál, amelyik meghívja a *wait()* eljárást, blokkolódik, és addig vár, amíg nem jeleznek neki, hogy tovább futhat. A blokkolódással egyidejűleg elengedi az előzőleg lekötött monitort. Amikor jeleznek neki, hogy tovább futhat, újra megpróbálja magához kötni a monitort, és ha sikerült, folytatja a futását. A *notify()* eljárás szolgál arra, hogy egy várakozó szálat tovább engedjünk. Amikor az adott szál megkapja a jelzést, megpróbál tovább futni az előbb leírt módon. Ha arra van szükségünk, hogy ne csak egy, hanem az összes szál feléledjen, a *notifyAll()* eljárást használhatjuk, melynek hatására az összes szál, amely egy objektumban várakozik, feléled és tovább fut.

Lehetőségünk van arra is, hogy bizonyos időhöz kössük a szál várakozását. Vagyis ha az adott ideig nem érkezik *notify()* jelzés, a szál automatikusan újraéled, hasonlóan a *join()* műveletnél leírtakhoz.

Nézzünk egy példát Java szinkronizációra:

```
// Java termelő–fogyasztó probléma:
public class Bounded_Buffer {
        private int[ ] pool;
        private int top, bottom, nums, size;

        public Bounded_Buffer (int s) {
           top=bottom=nums=0;
           size = s;
           pool = new int[s];
         }

        public synchronized boolean isEmpty( ) {
          return nums == 0;
        }

        public synchronized boolean isFull( ) {
          return size == nums;
        }

        public synchronized void put (int x) {
          while (isFull( ))
          try { wait( ); }
            catch (InterruptedException e) {};
          pool[top] = x;
          top = (top + 1) % size;
          nums++;
          notify( );
        }

        public synchronized int get ( ){
          while (isEmpty( ))
          try{ wait( ); }
            catch (InterruptedException e) {}
          int x = pool[bottom];
          bottom = (bottom + 1) % size;
          nums--;
          notify( );
          return x;
        }
}
// A program a következő oldalon folytatódik.
```

```
// Folytatás az előző oldalról.
public class Producer extends Thread {
    private Bounded_Buffer queue;

    public Producer(Bounded_Buffer queue) {
      this.queue = queue;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
          queue.put(i);
          System.out.println("Termelő berak: " + i);
          try {sleep((int)(Math.random() * 100));}
            catch(InterruptedException e) {
                System.err.println("Hiba " + e.toString());
            }
        }
    }
}

public class Consumer extends Thread {
    private Bounded_Buffer queue;

    public Consumer(Bounded_Buffer queue) {
      this.queue = queue;
    }
    public void run() {
      int value = 0;
      for (int i = 0; i < 10; i++) {
          value = queue.get();
          System.out.println("Fogyasztó kivesz: " + value);
      }
    }
}

public class Producer_Consumer {
    static public void main (String s[ ]) {
      Bounded_Buffer sharedBuffer = new Bounded_Buffer (100);
      Producer producer = new Producer (sharedBuffer);
      Consumer consumer = new Consumer (sharedBuffer);
      consumer.start();
      producer.start();
    }
}
```

A példában egy termelő és egy fogyasztó folyamat szerepel. A termelő adatokat pakol egy sorba, a fogyasztó pedig adatokat vesz ki a sorból. Ha a sor megtelik, a termelő folyamat nem tud újabb adatot rakni a sorba. Ebben az esetben ez a szál blokkolódik (*put* műveletben szereplő *wait()* metódus), ameddig a fogyasztó ki nem vesz egy adatot a sorból és fel nem ébreszti (*get* műveletben szereplő *notify()* metódus). Hasonlóan, ha a sor üres, a fogyasztó nem tud kivenni elemet a sorból, ezért blokkolódik (*get* műveletben szereplő *wait()* metódus), ameddig a termelő nem rak egy elemet a sorba és fel nem ébreszti (*put* műveletben szereplő *notify()* metódus).

14.6. C alapú könyvtárak

14.6.1. MPI

Az MPI (Message Passing Interface) [MPI 94] egy eljáráskönyvtár-szabvány, amely a programok közötti üzenetek segítségével történő kommunikáció biztosítását írja le. Mi itt elsősorban a C nyelvhez tartozó könyvtárral foglalkozunk, azaz a példaként leírt utasítások a C nyelv szintaxisát követik.

Az MPI segítségével többek között megoldhatjuk a folyamatok közötti pont–pont kommunikációt, a csoportok kezelését, a csoportkommunikációt és az üzenetekben szereplő típusok kezelését.

Az MPI első szabványa nem határoz meg eljárásokat folyamatok indítására, illetve dinamikus kezelésére. Ezzel a problémával csak az MPI-2 szabvány foglalkozik. Az MPI-1 szabvány szerinti rendszerben a folyamatok önálló, nem feltétlenül megegyező programok, melyeket egyszerre indítanak.

Az MPI könyvtár használatához kezdetben végre kell hajtanunk az *MPI_init*[2] kezdeti értéket adó utasítást, az eljárásnak pedig a parancssori argumentumokat kell átadnunk. A program normális befejezése esetén a lefoglalt erőforrásokat az *MPI_Finalize*[3] eljárással szabadíthatjuk fel. Hiba esetén a program az *MPI_Abort*[4] eljárással szakítható meg. Az eljárás segítségével nem csak egyetlen program, hanem egy egész programcsoport is leállítható (erre szolgál a *comm*) paraméter.

Az MPI-2-es szabvány [MPI 95] lehetővé teszi folyamatok létrehozását az alkalmazás elindítása után is, valamint olyan módszerről is gondoskodik, melynek segítségével az újonan létrehozott folyamatok és a már elindított folyamatok kommunikálni képesek egymással, még abban az esetben is, ha nincsenek szülő–gyermek viszonyban.

Új MPI folyamatokat az *MPI_Comm_spawn*[5] eljárással indíthatunk el. Az eljárás megpróbál *maxprocs* számú, a *command* paraméterben megadott programot elindítani *argv* paraméterekkel, létrehozza a kommunikációs lehetőséget az új folyamatok között és visszaad egy külső kommunikációs környezetet (lásd később) a szülő- és a gyermekfolyamatok között. Például, ha el akarjuk indítani a `proba` nevű programot `-gridfile` és `elsoproba.grd`

[2] *int MPI_init (int *argc, char *argv[])*

[3] *int MPI_Finalize (void)*

[4] *int MPI_Abort(MPI_Comm comm, int errorcode)*

[5] *int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int errcodes[])*

paraméterekkel, a következőképpen tehetjük meg (természetesen a „. . .” helyére a kívánt paramétereket behelyettesítve):

```
char parancs[ ] = "proba";
char *argv[ ] = {"-gridfile", "elsoproba.grd", NULL};
MPI_Comm_spawn(proba, argv, . . .);
```

Ehhez hasonló az *MPI_Comm_spawn_multiple*[6] eljárás, amely különböző programokat indít el (vagy ugyanazokat, de más paraméterezéssel) ugyanazon *MPI_COMM_WORLD*-be (lásd később) helyezve azokat és visszaad egy külső kommunikációs környezetet. Így például két programot egyidőben, mondjuk a *proba* programot az előbbi paraméterekkel, és egy *teszt* nevű programot *elsoteszt.grd* paraméterrel a következőképpen indíthatunk el:

```
char *array_of_commands[2] = {"proba", "teszt"};
char **array_of_argv[2];
char *argv0[ ] = {"-gridfile", "elsoproba.grd", (char *)0};
char *argv1[ ] = {"elsoteszt.grd", (char *)0};
array_of_argv[0] = argv0;
array_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, array_of_argv, . . .);
```

A gyermek folyamatoknál az *MPI_Comm_get_parent*[7] függvénnyel megkaphatjuk az adott folyamat szülő külső kommunikációs környezetét (amelyet az *MPI_Comm_spawn* vagy *MPI_Comm_spawn_multiple* meghívása után megkapott).

Az MPI-2 a meglévő csoport fogalmat (lásd később) használja a folyamatok ábrázolására. Egy folyamatot egy (csoport, sorszám) páros egyértelműen azonosít, viszont egy folyamathoz tartozhat több ilyen páros is, mivel benne lehet több csoportban.

Most térjünk vissza az MPI-1 által is biztosított eszközökre. A könyvtár által biztosított legegyszerűbb kommunikációs eljárások egyirányú (a küldőtől a fogadó felé irányuló), sorrendtartó, pont–pont kommunikációt valósítanak meg. A legegyszerűbb ilyen eljárás az *MPI_Send*[8]. Ez az eljárás egy adott típusú adatvektor elküldésére képes. Paraméterként meg kell adnunk az adatvektort tartalmazó tárterület címét, az üzenet típusát, a fogadó sorszámát és az üzenet továbbítására szolgáló kommunikációs környezetet. (A kommunikációs környezetek részletes leírását lásd később.) Az eljárás blokkoló és átmeneti tárat nem használó, aszinkron kommunikációt valósít meg.

Az üzenet típusát az *MPI_Datatype* típusú paraméterrel adhatjuk meg. A következő alaptípusok használhatók:
MPI_CHAR, *MPI_SHORT*, *MPI_INT*, *MPI_LONG*, *MPI_UNSIGNED_CHAR*, *MPI_UNSIGNED_SHORT*, *MPI_UNSIGNED*, *MPI_UNSIGNED_LONG*, *MPI_FLOAT*, *MPI_DOUBLE*, *MPI_LONG_DOUBLE*, *MPI_BYTE*, *MPI_PACKED*.

---

[6] *int MPI_Comm_spawn_multiple(int count, char *a_commands[], char **a_argv[], int a_maxprocs[], MPI_Info a_info[], int a_root, MPI_Comm comm, MPI_Comm *intercomm, int errcodes[])*

[7] *int MPI_Comm_get_parent(MPI_Comm *parent)*

[8] *int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

Az alaptípusokat felhasználva mi is létrehozhatunk bonyolultabb adattípusokat egy *MPI_Datatype* típusban, melyeket azután az *MPI_Type_commit*[9] eljárással engedélyeztethetünk a teljes rendszerben. Sikeres engedélyeztetés után az új típus is használható további összetett típusok létrehozására.

A legegyszerűbb eljárás, mellyel új típust hozhatunk létre, az *MPI_Type_Contiguous*[10]. Ezzel egy már ismert típusból adott hosszúságú, a memóriában folytonosan elhelyezkedő elemeket tartalmazó vektor hozható létre. Ennél valamivel bonyolultabb eljárás az *MPI_Type_vector*[11], amelynél az elemeknek nem kell feltétlenül folytonosan elhelyezkedniük, hanem egyforma hosszú blokkokban lehet azokat tárolni, melyek hossza és a közöttük levő eltolás mértéke megadható (az eltolás értéke az elemek adattípusának méretével mérendő). E két eljáráson kívül még számos típuskonstrukciós eljárást tartalmaz a könyvtár, melyekkel gyakorlatilag bármilyen típus leírható, de ezeket most nem vizsgáljuk meg részletesen.

Az adattípusok által lefoglalt memória felszabadítására az *MPI_Type_free*[12] eljárás használható. Ha egy kommunikációs eljárás éppen használja az adott adattípust, akkor az még rendben lefut, és csak utána szabadul fel a hozzá tartozó tárterület.

A különböző processzorok közötti kommunikációban általában nincs adattípusegyeztetés, csak az üzenet hossza számít, így a programozóra van bízva annak ellenőrzése, hogy az adattípusok a küldő és a fogadó oldalon megegyeznek.

Abban az esetben, ha az üzenet nem állítható elő egy lépésben, akkor azt valamilyen köztes tárolóhelyen kell raktározni, és utána egy lépésben elküldeni. Egy üzenet összeállítására az *MPI_Pack*[13] eljárás használható, amely gyakorlatilag az *MPI_Send*-hez hasonló formában megadott adatokat másol egy előre lefoglalt tárterületre. Az eljárást többször meghívva, tetszőleges adatokat pakolhatunk egy üzenetbe, amit az *MPI_Send* eljárással küldhetünk el az üzenet adattípusaként az *MPI_Packed* paramétert megadva. Egy ilyen üzenetből az *MPI_Unpack*[14] eljárással lehet kipakolni az adatokat. A csomagolásos üzenetösszeállítás, illetve -vétel leginkább a PVM módszerére hasonlít (lásd később), azzal a különbséggel, hogy itt a programozónak kell kezelnie az üzenettárat. A becsomagolt üzenetek küldésénél az MPI semmilyen típusellenőrzést nem tud végezni.

Az elküldött üzenetet a fogadó oldalon például az *MPI_Recv*[15] eljárással vehetjük, amely blokkoló üzenetvételre szolgál. Az üzenet forrása és típusa előírható, de az *MPI_ANY_SOURCE*, illetve *MPI_ANY_TAG* paraméter segítségével bárkitől bármilyen üzenetet lehet fogadni. Az üzenetről információkat az eljárás *status* paraméteréből lehet kinyerni, melynek *MPI_SOURCE*, *MPI_TAG*, illetve *MPI_ERROR* mezőiből egy ismeretlen üzenet küldőjét, típusát, illetve a vétel sikerességét lehet megállapítani.

---

[9] *int MPI_Type_commit(MPI_Datatype *datatype)*

[10] *int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)*

[11] *int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)*

[12] *int MPI_Type_free(MPI_Datatype *datatype)*

[13] *int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)*

[14] *int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)*

[15] *int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*

Nézzünk egy egyszerű üzenetküldéses példát, amelyben a *0* indexű folyamat küld üzenetet az *1*-es indexű folyamatnak. A példában használt *MPI_Comm_rank* eljárással a folyamat sorszáma kérdezhető le (részletesen lásd később).

```
#include "mpi.h"
main(int argc, char **argv)
{
    char uzenet[20];
    int sorszam;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &sorszam);
    if (sorszam == 0) {
        strcpy(uzenet,"Helló egyes!");
        MPI_Send(uzenet, strlen(uzenet), MPI_CHAR,
                 1, 59, MPI_COMM_WORLD);
    } else {
        MPI_Recv(uzenet, 20, MPI_CHAR, 0, 59,
                 MPI_COMM_WORLD, &status);
        printf("Az üzenet: %s\n", uzenet);
    }
    MPI_Finalize();
}
```

Mint már említettük, az *MPI_Send* aszinkron és átmeneti tárat nem használó üzenetküldést valósít meg. Az MPI azonban másfajta kommunikációt megvalósító eljárásokat is biztosít, melyek alakja csak nevében tér el az *MPI_Send*-től. Az *MPI_Bsend* eljárás átmeneti tárat használó kommunikációt valósít meg, ebben az esetben az eljárás az üzenet továbbítása nélkül is véget érhet, mivel annak tartalma még megőrződhet egy helyi tárolóban. Az *MPI_Ssend* szinkron módú üzenetküldést valósít meg, ez azonban nem jelent feltétlenül szinkron kommunikációt, ugyanis az MPI-nál a szinkron mód azt jelenti, hogy a küldő félnek meg kell várnia, amíg a másik oldal elindít egy megfelelő fogadó (például *MPI_Recv*) eljárást, viszont nem feltétlenül kell megvárnia annak befejeződését. Az MPI szinkron mód csak akkor jelent szinkron kommunikációt is, ha mindkét oldal blokkoló eljárásokat használ. Végül az *MPI_Rsend* úgynevezett várt kommunikációt valósít meg, mely csak akkor sikeres, ha az eljárás végrehajtásakor a fogadó oldal már végrehajtott egy *MPI_Recv* eljárást (azaz gyakorlatilag már várja az üzenetet). Ellenkező esetben a művelet eredménye definiálatlan. Mindhárom művelet az *MPI_Send*-hez hasonlóan blokkoló.

Az üzenetküldő műveleteknek vannak nem blokkoló változataik is. Ez az MPI-ben azt jelenti, hogy az eljárást két részre lehet bontani egy kezdeti értéket adó részre, illetve egy, a befejeződést vizsgáló részre. Ennek akkor van értelme, ha a művelet sokáig tartana és a várakozás helyett lehet más számítást végezni. A nem blokkoló műveletek akkor lehetnek hatékonyak, ha azokat a program egy másik végrehajtási szála, esetleg egy másik processzor hajtja végre. A végrehajtás módszerének részletei megvalósításfüggők.

Nem blokkoló üzenetküldést az *MPI_Isend*[16] eljárással lehet megvalósítani, melynek

[16] *int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)*

paraméterezése csak az utolsó paraméterben tér el a blokkoló változattól. Az eljárás lefutásakor kezdeti értéket ad az utolsó paraméterként kapott *MPI_Request* struktúrának, ami tárolja a háttérben futó vagy függőben levő üzenetküldés jellemzőit. Ennek vizsgálatával állapítható meg, hogy az üzenetküldés sikeresen befejeződött-e, vagy valamilyen hiba történt. Az eljárás a kezdeti értékadás után visszatér és az üzenet küldése a program további futásával párhuzamosan történik. A küldendő üzeneteket az üzenetküldés sikeres befejezéséig nem tanácsos megváltoztatni, mivel nem meghatározott, hogy a rendszer mikor olvassa ki azokat.

A nem blokkoló üzenetküldésnek is megvannak az előbbi változatai, az *MPI_Ibsend* átmeneti tárat használó, az *MPI_Issend* szinkron, az *MPI_Irsend* pedig várt kommunikációt valósít meg.

Nem blokkoló üzenetvételt valósít meg az *MPI_Irecv*[17] eljárás, mely az *MPI_Recv*-től csak az utolsó, *MPI_Request* típusú paraméterével tér el.

Egy háttérben futó eljárás vizsgálatára több lehetőségünk is van. Az *MPI_Wait*[18] eljárással meg lehet várni a művelet végét, blokkolva a programot, az *MPI_Test*[19] eljárással pedig a program blokkolása nélkül meg lehet vizsgálni, hogy egy művelet befejeződött-e már.

Nézzük meg az előbbi példát nem blokkoló üzenetküldéssel és üzenetvétellel:

```
#include "mpi.h"
main(int argc, char **argv)
{
    char uzenet[20];
    int sorszam;
    MPI_Request req;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &sorszam);
    if (sorszam == 0) {
        strcpy(uzenet, "Helló egyes!");
        MPI_Isend(uzenet, strlen(uzenet), MPI_CHAR, 1, 59,
                MPI_COMM_WORLD, &req);
        /* ... egyéb számolások ... */
        MPI_Wait(&req, &status);
    } else {
        MPI_Irecv(uzenet, 20, MPI_CHAR, 0,59,MPI_COMM_WORLD,&req);
        /* ... egyéb számolások ... */
        MPI_Wait(&req, &status);
        printf("Az üzenet: %s\n", uzenet);
    }
    MPI_Finalize();
}
```

[17] *int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)*

[18] *int MPI_Wait(MPI_Request *request, MPI_Status *status)*

[19] *int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)*

Az üzenetekről azok vétele nélkül is szerezhetünk információkat. Az *MPI_Probe*[20] és a nem blokkoló *MPI_Iprobe*[21] eljárásokkal egy üzenetről annak vétele nélkül megvizsgálható, hogy megérkezett-e már, mekkora a mérete, mi a típusa és ki a feladója.

Az MPI a pont–pont kommunikáción kívül a csoportkommunikációra is biztosít lehetőséget. Ebben az esetben több folyamat együttesen vesz részt a kommunikációban és a részt vevő folyamatok mindegyike egyszerre kell meghívja az eljárásokat. Az eljárások nagy része egyszerű algoritmusokat valósít meg, melyek az eddigi eszközökkel is megoldhatók lennének, de ezeknél egyszerűbb az írásmód, az ilyen kommunikációs eljárás a program többi kommunikációs eljárásától teljesen el van választva és egyes megvalósításokban ezek az eljárások gyorsabbak a pont–pont üzenetküldéssel megírt változatoknál.

Az egyik csoportkommunikációs eljárás az *MPI_Barrier*[22], melynek segítségével több folyamatot lehet egymással szinkronizálni. Az eljárás addig várakozik, ameddig az összes, az adott kommunikációs környezetben levő folyamat meg nem hívja ugyanezt az eljárást.

Az *MPI_Bcast*[23] eljárással egy kijelölt folyamat adatterületét másolhatjuk le a műveletben részt vevő összes folyamat adatterületére. Ez az eljárás például közösen használt változók, paraméterek értékének elterjesztésére szolgálhat.

Például 100 darab **int** értéket küldhetünk a *0* sorszámú folyamattól a csoport összes folyamatának:

```
MPI_Comm comm;
int tomb[100];
...
MPI_Bcast( tomb, 100, MPI_INT, 0, comm);
```

Természetesen a *comm* és *tomb* paramétert megfelelően be kell állítani a kipontozott részben. Az *MPI_Gather*[24] eljárás adatok összegyűjtésére szolgál. Minden folyamat egy adott mennyiségű adatot küld, ami egy kijelölt folyamatban egy vektorban összegyűlik. Az adatelem helyét a küldő folyamat sorszáma határozza meg. A küldött és fogadott adatok mennyiségének meg kell egyeznie. (Az eljárás hatása egyenértékű azzal, hogy a kijelölt folyamat ciklusban fogad adatokat a többi folyamattól és elhelyezi azokat egy vektorban.)

Például minden folyamat küldhet 100 darab **int** értéket a *0* sorszámú folyamatnak:

```
MPI_Comm comm;
int meret,kuldotomb[100];
int *fogadotomb;
...
MPI_Comm_size( comm, &csoportmeret);
rbuf = (int *)malloc(csoportmeret*100*sizeof(int));
MPI_Gather(kuldotomb, 100, MPI_INT, fogadotomb,
        100, MPI_INT, 0, comm);
```

[20] *int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)*

[21] *int MPI_IProbe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)*

[22] *int MPI_Barrier(MPI_Comm comm)*

[23] *int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

[24] *int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)*

Természetesen a paramétereket itt is megfelelően be kell állítani a kipontozott részben.

Ennek fordítottja az *MPI_Scatter*[25] eljárás, amely adatok szétosztására szolgál, és egy vektorban megadott adathalmazt oszt el az indexek alapján a kommunikációban részt vevő folyamatok között.

```
MPI_Comm comm;
int csoportmeret, *kuldotomb;
int fogadotomb[100];
...
MPI_Comm_size( comm, &csoportmeret);
kuldotomb = (int *)malloc(csoportmeret*100*sizeof(int));
...
MPI_Scatter(kuldotomb, 100, MPI_INT,
            fogadotomb, 100, MPI_INT, 0, comm);
```

Természetesen a paramétereket itt is megfelelően be kell állítani a kipontozott részekben.

Az *MPI_Alltoall*[26] eljárás segítségével minden, a kommunikációban részt vevő folyamat minden másiknak más-más adatokat tud küldeni. Hatása gyakorlatilag olyan, mintha az *MPI_Gather* és az *MPI_Scatter* eljárásokat egyszerre hajtanánk végre.

Az *MPI_Reduce*[27] és az *MPI_Scan*[28] eljárások asszociatív és kommutatív műveletek végrehajtására használhatók a folyamatokban tárolt lokális adatokon. Az elsőnél a végeredmény egy kijelölt folyamatba érkezik meg, míg a másodikban a részeredmények kiszámítására is sor kerül és egy adott indexű folyamatban az indexig kiszámolt részeredmény tárolódik majd eredményként (a 0 indexű folyamatban a bemenő érték, míg a legnagyobb indexű folyamatban az *MPI_Reduce* eljárás eredményeképp kapott érték fog tárolódni).

Bizonyos esetekben (például nagy programok fejlesztése, eljáráskönyvtárak írása) nagy segítséget jelenthet, ha a kommunikáció egy részét elválaszthatjuk a program többi részétől. Erre adnak lehetőséget a folyamatcsoportok, illetve a kommunikációs környezetek.

Folyamatok csoportba rendezésével bizonyos esetekben a rendszer számára is nagyobb lehetőséget biztosítunk az optimalizációra, valamint a programozónak sem kell a kód ismétlésével többször hivatkoznia önálló folyamatokra. A csoportok folyamatok rendezett halmazai, melyekben minden folyamatnak van egy sorszáma. A sorszámozás nullával kezdődik és folyamatos. A csoportok az *MPI_Group* átlátszatlan típussal írhatók le. Az ilyen leírás csak az adott folyamaton belül érvényes és nem adható át másik folyamatnak. Létezik két különleges konstans az *MPI_GROUP_EMPTY* és az *MPI_GROUP_NULL*; az első az üres, míg a második a nem létező csoport jelölésére szolgál. A csoportok kialakítása lokális,

[25] *int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)*

[26] *int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

[27] *int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_OP op, int root, MPI_Comm comm)*

[28] *int MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_OP op, MPI_Comm comm)*

azaz nem igényel kommunikációt más folyamatokkal. A létrehozott csoportok folyamatonként különbözhetnek.

Az *MPI_Comm_group*[29] eljárással lehet megkapni egy kommunikációs környezethez tartozó csoportot. A létező csoportokra minden halmazműveletet el lehet végezni, amivel kialakíthatók új csoportok. Például vehetjük két halmaz metszetét (*MPI_Group_intersection*[30]), unióját (*MPI_Group_***union**[31]), különbségét (*MPI_Group_difference*[32]), hozzátehetünk egy folyamatot egy halmazhoz (*MPI_Group_incl*[33]), vagy kivehetünk egy folyamatot egy halmazból (*MPI_Group_excl*[34]).

A csoportokat össze lehet hasonlítani (*MPI_Group_compare*[35]), meg lehet tudni méretüket (*MPI_Group_size*[36]), valamint le lehet kérdezni egy adott folyamat sorszámát a csoportban (*MPI_Group_rank*[37]). A nem használt csoportokat az *MPI_Group_free*[38] eljárással lehet felszámolni.

Az üzenetküldési rétegek segítségével egymástól független kommunikációs szinteket alakíthatunk ki egy programon belül, akár azonos folyamatok között is. Üzeneteket alapesetben csak egy üzenetküldési rétegen belül lehet küldeni, egy adott rétegben utazó üzenet más rétegben lévő folyamat számára nem látható és annak működését nem zavarja. Üzenetküldési rétegek magukban nem hozhatóak létre, csak egy csoportra vonatkozóan lehet ilyen tulajdonságot definiálni.

Az MPI-ben a csoportok és üzenetküldési rétegek összefogására szolgálnak a kommunikációs környezetek. Minden kommunikációban meg kell adni egy ilyen kommunikációs környezetet. Inicializáláskor létrejön egy *MPI_COMM_WORLD* kommunikációs környezet, amelyben minden folyamat benne lesz. Új kommunikációs környezetet létre lehet hozni egy csoportból az *MPI_Comm_create*[39] eljárással, vagy egy másik kommunikációs környezetből, az új kommunikációs környezet létrejöttekor a benne levő folyamatok kialakítanak egy új üzenetküldési réteget is, azaz az új kommunikációs környezetben küldött üzenetek nem keveredhetnek a korábbiakkal.

Az *MPI_Comm_dup*[40] eljárással egy meglevő kommunikációs környezetet másolhatunk le. Ennek hatására a régi kommunikációs környezet folyamatcsoportja megmarad, de létrejön egy új üzenetküldési réteg is. Az *MPI_Comm_split*[41] művelettel részekre bonthatunk egy kommunikációs környezetet. Minden, a műveletben részt vevő folyamat megmond-

[29] *int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)*
[30] *int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)*
[31] *int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)*
[32] *int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)*
[33] *int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)*
[34] *int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)*
[35] *int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)*
[36] *int MPI_Group_size(MPI_Group group, int *size)*
[37] *int MPI_Group_rank(MPI_Group group, int *rank)*
[38] *int MPI_Group_free(MPI_Group *group)*
[39] *int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)*
[40] *int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)*
[41] *int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)*

hatja, melyik új kommunikációs környezetben szeretne benne lenni, és milyen sorszámot szeretne viselni.

A kommunikációs környezetekhez tartozó csoportokról a csoportoknál vizsgált eljárásokhoz hasonló műveletekkel szerezhetünk információkat (*MPI_Comm_size*[42], *MPI_Comm_rank*[43] és *MPI_Comm_compare*[44])

Egy kommunikációs környezetet, amelyre nincs szükségünk, az *MPI_Comm_free*[45] eljárással szüntethetünk meg.

Az eddig tárgyalt kommunikációs környezetek olyan kommunikációt tesznek lehetővé, melyben a kommunikáló folyamatok ugyanannak a csoportnak tagjai, ezért hívhatjuk őket „belső kommunikációs környezeteknek" is.

Ezen kívül szükségünk lehet pont–pont kommunikációra két különböző csoportba tartozó folyamat között is. Erre szolgál a „külső kommunikáció" és a „külső kommunikációs környezet". A pont–pont belső és külső kommunikáció szintaxisa megegyezik. A külső kommunikációs környezetet nem lehet csoportkommunikációra használni, és egy kommunikációs környezet vagy külső vagy belső kommunikációs környezet, mindkettő nem lehet. Egy külső kommunikációs környezetet használó kommunikációs eljárás garantáltan nem zavar egyetlen olyan kommunikációs eljárást sem, amely másik kommunikációs környezetet használ.

Az *MPI_Comm_Test_Inter*[46] függvény segítségével lekérdezhetjük, hogy egy adott kommunikációs környezet külső vagy belső kommunikációs környezet-e. A függvény igaz értéket ad vissza, ha a kommunikációs környezet külső, és hamis értéket, ha belső.

Külső kommunikációs környezetet az *MPI_Intercom_Create*[47] eljárással hozhatunk létre két már létező belső kommunikációs környezetből. Az eljárás feltétele, hogy a két létező kommunikációs környezethez tartozó csoport két kiválasztott tagjának (*local_leader* és *remote_leader*) tudnia kell egymással kommunikálni, valamint a csoportoknak diszjunktnak kell lenniük.

Az *MPI_Intercomm_merge*[48] eljárás egy külső kommunikációs környezetből hoz létre egy belső kommunikációs környezetet, összekapcsolva a külső kommunikációs környezethez tartozó helyi és távoli csoportot.

Az MPI-2 szabvány az itt tárgyalt alapfüggvényeken kívül még számos új szolgáltatást is biztosít (például kommunikáció olyan folyamatok között, melyeknek nincs közös kommunikációs környezete; távoli memóriaeléréses kommunikáció stb.), amiket most (helyszűke miatt) nem tárgyalunk.

---

[42] *int MPI_Comm_size(MPI_Comm comm, int *size)*

[43] *int MPI_Comm_rank(MPI_Comm comm, int *rank)*

[44] *int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)*

[45] *int MPI_Comm_dup(MPI_Comm *comm)*

[46] *int MPI_Comm_test_inter(MPI_Comm comm, int *flag)*

[47] *int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)*

[48] *int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)*

Végül nézzünk egy egyszerű MPI példát:

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char *argv[ ])
{
    const int tag = 42;
    int id, taskszam, kod_id, cel_id, hiba, i;
    MPI_Status status;
    int msg[2];

    hiba = MPI_Init(&argc, &argv);
    if (hiba != MPI_SUCCESS) {
        printf("MPI inicializálási hiba!\n");
        exit(1);
    }
    hiba = MPI_Comm_size(MPI_COMM_WORLD, &taskszam);
    hiba = MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (taskszam < 2) {
        printf("Legalább 2 processzor szükséges
                a program futtatásához\n");
        MPI_Finalize();
        exit(0);
    }
    if (id == 0) {
        for (i = 1; i < taskszam; i++) {
            hiba = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE,
                            tag, MPI_COMM_WORLD, &status);
            kod_id = status.MPI_SOURCE;
            printf("Fogadott üzenet %d %d a %d
                    folyamattól\n", msg[0], msg[1], kod_id);
        }
    } else {
        msg[0] = id;
        msg[1] = taskszam;
        cel_id = 0;
        hiba = MPI_Send(msg, 2, MPI_INT, cel_id,
                        tag, MPI_COMM_WORLD);
    }
    hiba = MPI_Finalize();
    if (id == 0) printf("Vége\n");
    exit(0);
}
```

A programban mindegyik 0-tól különböző azonosítójú folyamat elküldi a saját azonosítóját és a folyamatok számát a 0 azonosítójú folyamatnak, amely fogadja az üzeneteket és kiírja azokat a képernyőre.

14.6.2. PVM

A PVM (Parallel Virtual Machine) szoftverrendszert azzal a céllal hozták létre, hogy hálózatba kapcsolt számítógépeket egyetlen nagy, párhuzamos virtuális számítógépként kezelhessünk.

A PVM rendszer két részből áll. Az első része egy démon program, amelynek kezdetben mindegyik, a rendszerbe bekapcsolt számítógépen elindul egy-egy példánya. Ennek a programnak a virtuális gép létrehozása és működtetése a feladata. A rendszerbe több, akár különböző típusú számítógépet kapcsolhatunk. A rajta futó programok szempontjából ezek után a rendszer egy nagy, elosztott memóriájú virtuális számítógépnek látszik. A virtuális gépben a folyamatoknak egyértelmű azonosítójuk van (**tid**), amelyet a folyamatok kezelésében és a kommunikációban is használhatunk.

A másik rész az eljáráskönyvtár, amelyben a felhasználó által hívható, a folyamatindítás és -kezelés, valamint a kommunikáció megvalósítására szolgáló függvények találhatóak. Ebben a részben csak a C nyelvű könyvtárral foglalkozunk. A teljesség igénye nélkül tekintsük át, milyen párhuzamosságot támogató függvények találhatók a C könyvtárban. Nézzük először a folyamatvezérlésre szolgáló függvényeket.

A *pvm_mytid*[49] eljárás lefuttatásával a program egy PVM folyamattá válik és bekerül a virtuális gépbe. A visszatérési érték lesz a folyamat **tid**-je. Az eljárás többször is meghívható, de a kezdeti értéket adó lépéseket csak az első híváskor hajtja végre.

A *pvm_exit*[50] függvény meghívásával a program kilép a rendszerből. Ezek után folytathatja a működését, de PVM hívásokat nem hajthat végre.

A *pvm_spawn*[51] eljárás segítségével új folyamatokat indíthatunk. A folyamat leírásához a folyamatot leíró futtatható program nevét és parancssori argumentumait kell megadnunk. Meghatározhatjuk azt is, hol induljanak el az új folyamatok, de ennek kiválasztását a rendszerre is bízhatjuk. Ha az eljárás sikeresen elindította a folyamatokat, akkor azok azonosítóit (**tid**-jeit) adja vissza.

A *pvm_kill*[52] függvénnyel leállíthatunk egy futó folyamatot. A folyamat az őt futtató operációs rendszer módszerével lesz leállítva.

A folyamatok kezeléséhez bizonyos információkra is szükségünk lehet. A *pvm_parent*[53] függvény annak a folyamatnak az azonosítóját adja vissza, amely a hívót *pvm_spawn*-nal elindította, vagy a *PvmNoParent* értéket, ha ilyen nem volt. Nézzük, hogyan jelenhetnek meg ezek a függvények egy programon belül:

---

[49] *int tid = pvm_mytid(void)*
[50] *int info = pvm_exit(void)*
[51] *int numt = pvm_spawn(char *task, char **argv, int flag, char *where, int ntasks, int *tids)*
[52] *int info = pvm_kill(int tid)*
[53] *int tid = pvm_parent(void)*

A szülőfolyamat:

```
#include "pvm3.h"
#include <stdio.h>
#define PROCSZAM 5
main()
{
  int mytid;
  int i, msgtype, adatszam = MERET/PROCSZAM;
  mytid = pvm_mytid();
  ...
  pvm_spawn("gyermek", (char **)0, PvmTaskDefault,
            "", PROCSZAM, task_ids);
  ...
  pvm_exit();
}
```

A gyermekfolyamat:

```
#include "pvm3.h"
#include <stdio.h>
main()
{
  int mytid;
  int szulo;
  mytid = pvm_mytid();
  ...
  szulo = pvm_parent();
  ...
  pvm_exit();
}
```

A *pvm_pstat*[54] segítségével folyamatok állapotát kérdezhetjük le, azaz megtudhatjuk, hogy az adott folyamat fut-e még vagy létezik-e egyáltalán, a *pvm_tasks*[55] függvénnyel pedig a virtuális gép összes folyamatáról megtudhatjuk, hogy mi az azonosítója, a szülője azonosítója, a folyamat állapota, a hozzá tartozó démon azonosítója és a folyamat neve.

A könyvtár a folyamatindító és -kezelő függvényeken kívül kommunikációt megvalósító függvényeket is tartalmaz. A kommunikáció a PVM-ben üzenetek küldésével és fogadásával valósítható meg. Vizsgáljunk meg most néhány olyan függvényt, amely az üzenetküldést, illetve -fogadást segíti.

Egy üzenet elküldéséhez először létre kell hoznunk egy küldő tárat. A PVM-ben mindig egy aktív küldő tár van, melyben az elküldendő üzenetet lehet összeállítani és egy aktív

[54] *int pstat = pvm_pstat(int tid)*
[55] *int info = pvm_tasks(int which,int *ntask,struct pvmtaskinfo **taskp)*

fogadó tár, amelyben a legutóbb fogadott üzenet található. A *pvm_mkbuf*[56] függvényel hozhatunk létre új küldő tárat és állíthatjuk be az üzenet összeállításakor használt módszert.

A *pvm_initsend*[57] eljárás törli az aktuális küldő tárat és létrehoz egy újat. Ha csak egy tárat használunk, akkor az eljárást mindig meg kell hívni egy üzenet összeállítása előtt.

Ha már létrehoztunk egy tárat, akkor összeállíthatjuk benne az elküldeni kívánt üzenetet. A csomagoló eljárások segítségével adatokat rakhatunk be az előzőleg létrehozott küldő tárba. Egyszerű változók és tömbök becsomagolása könnyű, de a bonyolultabb adatszerkezeteket elemenként kell belerakni egy üzenetbe. A rendszer a következő típusokhoz szolgáltat csomagoló függvényeket: az egész számok variációi (*pvm_pkint*, *pvm_pkshort*, *pvm_pklong*, *pvm_pkuint*, *pvm_pkushort*, *pvm_pkulong*), komplex számok (*pvm_pkcplx*, *pvm_pkdcplx*), lebegőpontos számok (*pvm_pkdouble*, *pvm_pkfloat*), bájt (*pvm_pkbyte*). A csomagoló függvényekkel egy tömböt is be lehet rakni a tárba.

A csomagoló függvények szintaxisa a következő:

```
int info = pvm_pkbyte(char *cp, int nitem, int stride);
int info = pvm_pkint(int *np, int nitem, int stride);
...
```

C-beli szöveg a *pvm_pkstr*[58] függvénnyel csomagolható be, ez nem tárolja a szöveg hosszát, csak 0 kódú karakterrel jelzi a végét.

A fentieken kívül üzenetek összeállítására használható még a *pvm_packf*[59] eljárás. Az eljárást a C *printf* függvény formátuma szerint lehet üzenetek összeállítására használni, azzal a különbséggel, hogy itt tömböket is kezelhetünk. (Egyedi változóknál értéket, tömbök esetén pedig az első elem címét kell átadni a további paraméterekben.)

Miután összeállítottuk az üzenetet, elküldhetjük azt egy másik folyamatnak. Ezt legegyszerűbben a *pvm_send*[60] eljárással tehetjük meg, melynek segítségével egy adott azonosítójú folyamatnak tetszőleges üzenettípussal elküldhetjük az aktív üzenettárban összeállított üzenetet.

A *pvm_mcast*[61] eljárás segítségével egyszerre több folyamatnak is elküldhetjük ugyanazt az üzenetet. Hatása egyenértékű a *pvm_send* többszöri meghívásával, de egyes esetekben gyorsabb.

Az előzőekben leírt tár kezdeti értékadását, feltöltését és a csomag elküldését egyetlen lépésben is elvégezhetjük a *pvm_psend*[62] eljárás segítségével. Ez az eljárás egy előre megadott adattípusból (bájt, komplex szám, lebegőpontos szám és egész szám típusok) álló vektor elküldésére képes. Az eljárás az aktív tárat nem változtatja meg.

---

[56] *int bufid = pvm_mkbuf(int encoding)*
[57] *int bufid = pvm_initsend(int encoding)*
[58] *int info = pvm_pkstr(char *cp)*
[59] *int info = pvm_packf(const char *fmt, ...)*
[60] *int info = pvm_send(int tid, int msgtag)*
[61] *int info = pvm_mcast(int *tids, int ntask, int msgtag)*
[62] *int info = pvm_psend(int tid, int msgtag, void *vp, int count, int type)*

Adott azonosítójú folyamattól adott típusú üzenetet a *pvm_recv*[63] eljárás segítségével fogadhatunk. A két paraméter mindegyikét helyettesíthetjük -1-gyel, amivel tetszőleges folyamattól, illetve tetszőleges típusú üzenetet is fogadhatunk. Például a *pvm_recv(-1,-1)* az első beérkező üzenetet fogadja. Az eljárás addig blokkolja a hívó folyamatot, amíg a megfelelő üzenet meg nem érkezik. Ha az üzenet megérkezik, az aktív vételi tárat törli és az új üzenetre állítja. Nézzünk egy egyszerű üzenetküldést:

A szülőfolyamat:

```
main()
{
   int cc, az, msgtag;
   char tomb[100];

   printf("Az azonosítóm: t%x\n", pvm_mytid());
   cc = pvm_spawn("gyermek", (char**)0, 0, "", 1, &az);
   if (cc == 1) {
      msgtag = 1;
      pvm_recv(az, msgtag);
      pvm_upkstr(tomb);
      printf("A t%x azonosítójú folyamattól %s üzenet jött.\n",
            az, tomb);
   } else
      printf("Hiba! Nem lehet a gyermekeket elindítani!\n");
   pvm_exit();
}
```

A gyermekfolyamat:

```
#include "pvm3.h"
main()
{
   int szuloaz, msgtag;
   char tomb[100];

   szuloaz = pvm_parent();
   strcpy(tomb, "Helló! ");
   msgtag = 1;
   pvm_initsend(PvmDataDefault);
   pvm_pkstr(buf);
   pvm_send(szuloaz, msgtag);
   pvm_exit();
}
```

[63] *int bufid = pvm_recv(int tid, int msgtag)*

Hasonló eljárás a *pvm_nrecv*[64] is, de ez az eljárás nem blokkolja a folyamatot, ha még nem érkezett meg az üzenet, hanem 0 értékkel tér vissza.

A *pvm_probe*[65] segítségével lekérdezhetjük, hogy érkezett-e a paraméterezésnek megfelelő üzenet. Ha érkezett, az eljárás nullától különböző visszatérési értéket ad.

Ezen műveletek pontos megértéséhez fontos tudnunk, hogy egy PVM program valójában minden hozzá érkező üzenetet fogad, de azokat egy várakozási sorba rendezi. A fenti műveletek ebben a sorban keresnek, illetve ebből a sorból vesznek ki műveleteket.

A *pvm_recv* eljáráshoz hasonló a *pvm_trecv*[66] is, de itt előírható egy türelmi idő, ameddig az eljárás várakozik a még meg nem érkezett üzenetre.

Miután fogadtuk az üzenetet, az összeállításhoz hasonló eljárásokkal lehet az adatokat kipakolni az aktív vételi tárból. A támogatott adattípusok: az egész számok variációi (*pvm_upkint*, *pvm_upkshort*, *pvm_upklong*, *pvm_upkuint*, *pvm_upkushort*, *pvm_upkulong*), komplex számok (*pvm_upkcplx*, *pvm_upkdcplx*), lebegőpontos számok (*pvm_upkdouble*, *pvm_upkfloat*), bájt (*pvm_upkbyte*). Szöveg kicsomagolására a *pvm_upkstr*, formátum szerinti kicsomagolásra pedig a *pvm_unpackf* eljárás alkalmazható. Az eljárások szintaxisa is hasonló a csomagoló függvényekéhez:

```
int info = pvm_upkint(int *np, int nitem, int stride);
int info = pvm_upkstr(char *cp);
...
```

Vigyázzunk arra, hogy a PVM nem ellenőrzi, hogy azt pakoljuk-e ki, amit beleraktunk egy üzenetbe, erről a felhasználónak kell gondoskodnia.

Az üzenetek küldésének egylépéses változatához (*pvm_psend* eljárás) hasonlóan az üzenetek vételére is létezik egylépéses eljárás (*pvm_precf*)[67], amely az üzenetek vételének és kicsomagolásának műveleteit fogja össze

A folyamatokat a PVM esetében is csoportokba rendezhetjük. Egy adott folyamat a *pvm_joingroup*[68] eljárással csatlakozhat egy csoporthoz és a *pvm_lvgroup*[69] eljárással hagyhatja el azt. Egy folyamat egyszerre több csoportnak is tagja lehet és bármikor kiléphet, illetve újra beléphet egy adott csoportba.

Több folyamat szinkronizációjára a *pvm_barrier*[70] eljárás szolgál, amely addig várakozik, amíg az adott csoportban adott számú folyamat meg nem hívja ezt a függvényt.

Csoportbeli folyamat a többi adott csoportbeli folyamatnak a *pvm_bcast*[71] függvénnyel küldhet üzenetet.

---

[64] *int bufid = pvm_nrecv(int tid, int msgtag)*
[65] *int bufid = pvm_recv(int tid, int msgtag)*
[66] *int bufid = pvm_trecv(int tid, int msgtag, struct timeval *tmout)*
[67] *int info = pvm_precf(int tid, int msgtag, void *vp, int count, int type, int *rtid, int *rtag, int *rcount)*
[68] *int inum = pvm_joingroup(char *group)*
[69] *int info = pvm_lvgroup(char *group)*
[70] *int info = pvm_barrier(char *group, int count)*
[71] *int info = pvm_bcast(char *group, int msgtag)*

Végül nézzünk egy egyszerű PVM példát, amelyben kétfajta folyamat szerepel: a *szülő-* és a *gyermekfolyamat*. A szülőfolymat elindít 5 gyermekfolyamatot és mindegyiknek elküldi egy vektor adott részét. A gyermekek kiszámolják a nekik küldött vektorban lévő elemek összegét, majd az összeget visszaküldik a szülőnek. A szülő összeadja az így kapott összegeket, majd a végeredményt kiírja a képernyőre.

A szülőfolyamat:

```
#include "pvm3.h"
#include <stdio.h>
#define MERET 500
#define PROCSZAM 5

main()
{
    int mytid, task_ids[PROCSZAM];
    int tar[MERET], ertek[PROCSZAM], sum = 0;
    int i, msgtype, adatszam = MERET/PROCSZAM;

    mytid = pvm_mytid();

    for (i = 0; i < MERET; i++)
        tar[i] = (i % 20) + 2;

    pvm_spawn("gyermek", (char **)0, PvmTaskDefault, "",
              PROCSZAM, task_ids);

    for (i = 0; i < PROCSZAM; i++) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&adatszam, 1, 1);
        pvm_pkint(&tar[adatszam*i], adatszam, 1);
        pvm_send(task_ids[i], 1);
    }

    msgtype = 11;
    for (i = 0; i <PROCSZAM; i++) {
        pvm_recv(task_ids[i], msgtype);
        pvm_upkint(&ertek[i], 1, 1);
        sum += ertek[i];
    }

    printf("Az összeg: %d \n", sum);

    pvm_exit();
}
```

A gyermekfolyamat:

```
#include "pvm3.h"
#include <stdio.h>

main()
{
    int mytid, msgtype;
    int j, sum, *tar;
    int adatszam, szulo;

    mytid = pvm_mytid();

    pvm_recv(-1, -1);
    pvm_upkint(&adatszam, 1, 1);
    tar = (int *) malloc(adatszam*sizeof(int));
    pvm_upkint(tar, adatszam, 1);

    sum = 0;
    for (j = 0; j < adatszam; j++)
        sum += tar[j];

    msgtype = 11;
    szulo = pvm_parent();
    pvm_initsend(PvmDataRaw);
    pvm_pkint(&sum, 1, 1);
    pvm_send(szulo, msgtype);

    pvm_exit();
}
```

Feladatok

1. Írjunk Ada programot a termelő–fogyasztó probléma megoldására.
2. Valósítsuk meg a kölcsönös kizárást Ada taszkok segítségével.
3. Írjunk Ada programot a monitor fogalmának bemutatására.
4. Írjunk CSP, illetve MPI programot a termelő–fogyasztó probléma megoldására.
5. Írjunk Java, illetve PVM programot az étkező filozófusok problémájának megoldására. Lehetőség szerint készítsünk holtpontmentes programot.

15 Programkönyvtárak

Jelen fejezet célja, hogy megismertesse az olvasóval, hogyan érdemes programkönyvtárakat tervezni és fejleszteni. Ezen belül részletesen kifejtjük a programkönyvtárakkal szemben támasztott követelményeket, a könyvtárak tervezési szempontjait, valamint az objektumorientált könyvtártervezéshez szükséges ismereteket. Említésre kerülnek majd néhány elterjedt programozási nyelv beépített programkönyvtárainak legfőbb jellemzői is.

A programozási nyelvek, illetve környezetek rendszerint szabványos beépített könyvtárakkal rendelkeznek, melyek az alapvető szolgáltatásokat valósítják meg. Nagyobb programrendszerek fejlesztése során azonban gyakran előfordulhat, hogy saját, speciális API-t (alkalmazás-programozói interfész) kell létrehoznunk. Jelen fejezet célja az ehhez szükséges tervezés szempontjainak ismertetése.

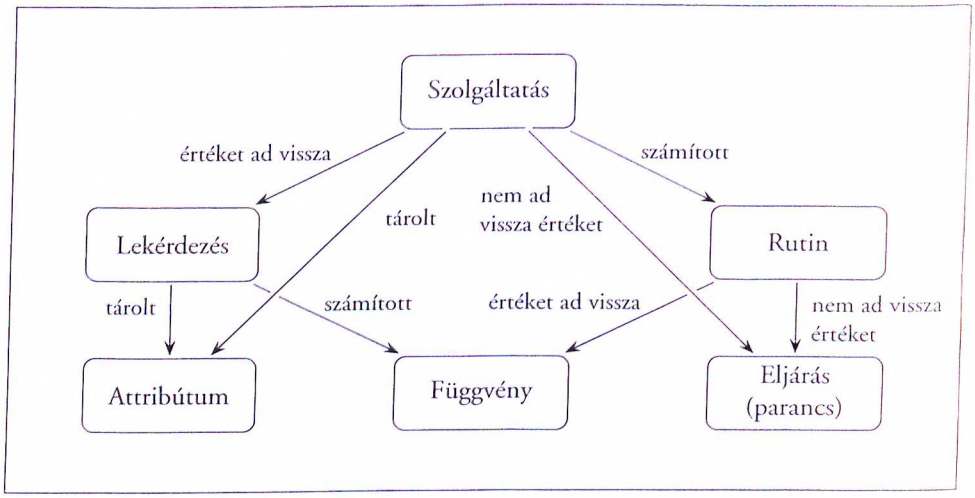
Ezen szempontok közül a legfontosabbak: helyesség, hatékonyság, megbízhatóság, kiterjeszthetőség, újrafelhasználhatóság, jól dokumentáltság. Részletesen foglalkozunk az osztályok szerkezetének, csoportosításának és méretének, a biztonsági elemeknek, az elnevezési szabályoknak és a memóriakezelésnek a megtervezésével.

Példaként ismertetjük a Java [Nyé 01], a C++ [Str 00], az Eiffel [Mey 91] és még néhány jellemző nyelv szabványos könyvtárainak megközelítését. Ezen kívül kitérünk az objektumorientáltságtól független programozási paradigmák (funkcionális, aspektusorientált stb. programozás) legjellemzőbb képviselőire is.

A továbbiakban a programkönyvtárakat olyan alprogram-, modul-, osztály-, illetve adattípus-gyűjteményeknek tekintjük, amelyek egy jól körülhatárolható programozói szolgáltatáscsoportot megvalósító programkódot tartalmaznak, és egységes (manapság általában objektumorientált) felületet (interfészt) biztosítanak a felhasználó programozók számára. Éppen ezért nem nevezhetők végfelhasználói programoknak, sokkal inkább a programozók munkáját megkönnyítő (szoftver-)építőköveknek tekinthetők. A hangsúly itt a gyűjtemény jellegen van: a már korábban tárgyalt modulok tervezési alapelvei a programkönyvtárakra is érvényesek, csak mindazt egy még magasabb, átfogóbb szinten kell alkalmazni.

A programkönyvtárak által nyújtott szolgáltatásokat a 15.1. ábrán látható módon csoportosíthatjuk.

További csoportosítási lehetőséget biztosít a csomagok (Java) vagy clusterek (Eiffel) használata. A programkönyvtárak és csomagok 1–sok, illetve sok–1 kapcsolatban állhatnak egymással. A csomagok általában egy az egyben képeződnek le a fizikai tárolóterületre, például a fájl- vagy könyvtárszerkezetre.



15.1. ábra: A programkönyvtárak által nyújtott szolgáltatások

15.1. Követelmények a programkönyvtárakkal szemben

Akármilyen céllal is készítünk programkönyvtárat – legyen az egy grafikai eszközkészlet vagy egy üzleti célú rendszer illesztőfelülete más üzleti célú rendszerekhez –, egyaránt fontos, hogy megfelelő minőségű legyen. Vannak a szoftvertermékekkel szemben támasztott általános követelmények, amelyeknek természetesen a programkönyvtáraknak is meg kell felelniük, valamint vannak olyan, egyéb tulajdonságok, igények, amelyek kifejezetten a programkönyvtárak készítői számára fogalmazódnak meg.

Mielőtt felsorolnánk a programkönyvtárakkal szemben támasztott követelményeket, megpróbáljuk felvázolni, hogy maguk a könyvtárat készítő személyek milyen tulajdonságokkal jó, ha rendelkeznek.

15.1.1. A jó könyvtárfejlesztő tulajdonságai

Programkönyvtárakat készíteni gyakorlatilag egyet jelent nagy, összetett szoftverrendszerek fejlesztésével. Ennek következtében az alábbi követelmények mindenkire vonatkoznak, aki bonyolult, sok részből álló szoftvertermék tervezésében vagy fejlesztésében vesz részt.

Az alábbi lista a teljesség igénye nélkül született, de a legfontosabb dolgokat tartalmazza. Tehát a jó programkönyvtár-fejlesztő:

- absztrakciós készséggel rendelkezik – képes arra, hogy az egyedi jelenségekből általánosítson;
- van érzéke a részletekhez – egy könyvtárban ugyanis minden kis tulajdonság számít (itt nincs olyan, hogy „elég jó", teljes kell, hogy legyen a könyvtár);

- alaposan ismeri a nyelv minden részletét – előnyeit és hátrányait, tudja, hogy milyen célra mi a legalkalmasabb eszköz;
- rendszerető – képes az osztályozásra, hogy mindennek megtalálja a helyét (nem praktikus ugyanis, ha egy programkönyvtár elemei következetlenül vannak elnevezve vagy rendszerezve);
- van irodalmi érzéke – a leíró megjegyzéseinek megfelelő stílusa és eleganciája van;
- elsőrendű programozó és tervező – át tudja látni a könyvtár későbbi használójának igényeit (hol van szükség általánosságra, és hol kell inkább hatékonynak lenni stb.);
- csapatjátékos – nem szabad, hogy egyéni stílusa érvényesüljön, a cél az egységes és következetes használati stílus (kreativitása a könyvtár által nyújtott szolgáltatásokban jöjjön elő).

Aki a fenti tulajdonságokkal és képességekkel rendelkezik, az jó eséllyel vághat neki, hogy a következőkben bemutatott követelményeknek a lehető legteljesebb mértékben megfelelő programkönyvtárat hozzon létre.

15.1.2. Alapvető minőségi követelmények

Az alapvető követelmények megegyeznek a szoftvertervezés általános szempontjaival: helyesség, hatékonyság, megbízhatóság, kiterjeszthetőség és újrafelhasználhatóság (lásd a 2. oldalt). A programkönyvtárak esetében azonban ezek az igények erőteljesebben jelentkeznek.

Helyesség

A programtermék helyessége azt jelenti, hogy a program pontosan megoldja a feladatot, megfelel a kívánt specifikációnak. Ez az első és legfontosabb kritérium, hiszen ha egy program nem úgy működik, ahogy kellene, akkor az egyéb szempontok nyilván nem sokat számítanak.

A programkönyvtárak esetében összetettségük miatt különösen fontos a részletes, mindenre kiterjedő, jól átgondolt tervezés és specifikáció. Ennek megfelelően különösen ügyelni kell arra – akár nyelvi eszközök segítségével, akár „adminisztratív" eszközökkel, úgymint CASE eszközök[1], round-trip engineering[2], szoftverminőség-biztosítási módszerek és szab-

[1] Computer-Aided Software Engineering. Olyan szoftvereszközök, melyek bizonyos tervezési és fejlesztési módszereket alkalmazva a rendszertervezés és fejlesztés folyamatai közül jó néhányat automatizálnak, például kódgenerátorok és kódvisszafejtés alkalmazásával. Mostanság a legelterjedtebb modellezési szabvány az UML, az ezt alkalmazó legnépszerűbb CASE eszközök: ArgoUML, Oracle Designer, Paradigm+, Rational Rose, Software Through Pictures, Together.

[2] Azt a folyamatot jelöli, amelyben kódelemző eszközök segítségével öröklött rendszerek forráskódját egy megadott tervezési jelölésrendszer szerinti modellé alakítjuk, majd a rendszert újra vagy tovább tervezve a végterméket oly módon generáljuk vissza forráskóddá, hogy annak eredeti elemei (a modellben nem jelölt programrészek) ne vesszenek el.

ványok (CMM[3], SPICE[4], Bootstrap[5], RUP[6], ISO 9001[7] stb.) –, hogy a készülő szoftvertermék mindenben maradéktalanul megfeleljen a tervezés során előírt kívánalmaknak.

Hatékonyság

A program hatékonysága a futási idővel és a felhasznált tárterület méretével arányos. Minél gyorsabb (CPU idő, I/O, hálózati forgalom), illetve minél kevesebb memóriát és háttértárat használ, annál hatékonyabb.

Ezek a követelmények sokszor egymás ellen hatnak. A gyorsabb futást sokszor nagyobb memóriaigény ellensúlyozza, és fordítva.

Bjarne Stroustrup, a C++ nyelv megalkotója szerint a hatékonyság és az általánosság egymásnak ellentmondó követelmények. Szerinte programkönyvtárak készítésekor a hatékonyság szem előtt tartása fontosabb, mivel ellenkező esetben a felhasználók (alkalmazásfejlesztők) elkezdenének bizonyos célokra saját, hatékonyabb eljárásokat írni. Emiatt sérülne a szabványosság elve, növekedne az inkompatibilis szoftvertermékek száma.

Ennek ellentmond az a nézőpont, hogy ha egy szolgáltatást nem kellően általánosan terveztek, hanem túl sok a hatékonyság miatti „bedrótozás", akkor bizonyos esetekben az alkalmazásfejlesztők szintén elkezdik saját programrészekkel helyettesíteni a könyvtár elemeit, hogy egyedi igényeiket kielégíthessék. Nyilvánvaló, hogy minden esetre alkalmazható jó megoldás nem létezik erre a problémára. A programkönyvtár tervezőjének a könyvtár fő céljait kell szem előtt tartania, amikor az erre vonatkozó döntését meghozza.

A C++ nyelvet egyértelműen a hatékonyságot igénylő feladatokra használják leginkább, ily módon az arra épülő programkönyvtárak is inkább azt tartják szem előtt. Például a nyelvhez szorosan kapcsolódó szabványos könyvtárból (Standard Template Library) hatékonysági okokból szinte teljesen száműzték az öröklődés alkalmazását, ily módon nyilvánvalóan veszítve az általánosságból. Magasabb szintű nyelveknél könnyen elképzelhető, hogy éppen az általánosság az, amire inkább szükség van.

Megbízhatóság

Egy programot megbízhatónak nevezünk, ha abnormális – a specifikációban nem leírt – helyzetekben nem történik katasztrófa, hanem valamilyen „ésszerű" működést mutat.

A fenti meghatározás jól tükrözi, hogy a megbízhatóság messze nem olyan precíz fogalom, mint a helyesség. Mondhatnánk persze, hogy ha pontosabb specifikációt írtunk volna,

[3] Capability Maturity Model. A Carnegie Mellon University által kifejlesztett módszer, melynek lényege, hogy a szoftverfejlesztéssel foglalkozó szervezeteket egy ötfokozatú skálán minősítik, aszerint, hogy a minőségbiztosítás mennyire épült be az általuk fejlesztett szoftverek életciklusába.

[4] Software Process Improvement and Capability dEtermination, az ISO által szabványosított CMM-hez hasonló, annál általánosabb és összetettebb módszer.

[5] Európai cégek konzorciuma által kifejlesztett, a SPICE-szal és a CMM-mel rokon módszer.

[6] Rational Unified Process. Az UML-re épülő, teljes életciklust lefedő szoftver-előállítási folyamat, amely a Rational termékeivel együttműködve segíti a tervezők, fejlesztők és tesztelők munkáját.

[7] Minőségbiztosítási szabvány, mely a fentiekkel ellentétben nem a szoftvergyártás folyamatára, hanem az elkészült termékre vonatkozik.

akkor a megbízhatóság pontosan a helyességet jelentené, de a gyakorlat azt mutatja, hogy mindig vannak olyan esetek, amelyeket egy specifikáció nem ír le explicit módon.

Például a C nyelvben, ha egy tömbnek n bájtot kell tudnia tárolni, és a specifikációból kimaradt, hogy a túlcsordulás megelőzésére ellenőrző feltételt kell alkalmazni, akkor ha a programozó nem figyelt oda a megbízhatóságra, és hagyja, hogy az $n + 1$-edik bájt is adott esetben felülíródjon, akkor vagy nem várt módon fog működni a program (egy másik változót írunk felül), vagy az operációs rendszertől kapott védelmi hibaüzenet lesz az eredmény.

Egy sokak által használt programkönyvtár esetében ezért igen fontos szempont a megbízhatóság és a biztonság. Ennek megvalósítása elő- és utófeltételekkel, valamint invariánsokkal történhet. Mindez a programkönyvtár írói és használói között tulajdonképp egy szerződés létrejöttét jelenti, ahol az előfeltétel a felhasználó kötelezettsége és a programozó biztosítéka, míg az utófeltétel a programozó kötelezettsége és a használó biztosítéka. Így a programozónak nem kell „defenzíven" programoznia, (minden lehetséges hibalehetőségre és bemenetre felkészülve) és a kód is hatékonyabb és egyszerűbb lesz. A lehetséges kétfajta megközelítési mód:

- Toleráns megközelítés: a programkönyvtári rutinoknak nincs előfeltételük (vagy csak gyenge) és minden lehetséges bemenetre valamilyen módon reagálnak. Ennek persze az a következménye, hogy a bemenet sokszor felesleges alapos ellenőrzése miatt erősen csökken a kód hatékonysága.
- Követelődző megközelítés: minden rutin szigorú előfeltételekkel rendelkezik, ezek biztosítása a felhasználó felelőssége. Ilyenkor nyugodt szívvel eltekinthetünk a bemeneti adatok ellenőrzésétől, viszont hibás bemenő adatok esetén nem tudunk felelősséget vállalni programunk viselkedéséért.

Mint látható, mindkét esetnek megvannak az előnyei és hátrányai. Az ajánlott megközelítés:

- Csak az absztrakt művelet logikailag helyes elvégzéséhez szükséges előfeltételeket ellenőrizzük.
- Csak azt ellenőrizzük, ami ha nem teljesülne, súlyosan befolyásolná a hatékonyságot.

Az így létrejött szerződés bármilyen megsértése programhibát jelez. Az előfeltétel sérülése felhasználó oldali hibát, míg az utófeltétel vagy az invariáns sérülése programkönyvtár-hibát jelez. Ezért a programkönyvtár kifejlesztése után a terjesztés előtt a hatékonyság érdekében elég csak az előfeltétel teljesülésének vizsgálatát bekapcsolni.

Kiterjeszthetőség és karbantarthatóság

A kiterjeszthetőség arra vonatkozik, hogy milyen könnyű a programterméket a specifikáció változásához igazítani.

A felhasználó gyakran igényli a programtermék továbbfejlesztését, módosítását, új környezeti feltételekhez igazítását. Egyes felmérések szerint a szoftvertermékekkel kapcsolatos költségek 70 %-át karbantartásra fordítják, így érthető, hogy ez a követelmény a program minőségét jelentős mértékben befolyásolja.

A kiterjeszthetőségnek különösen nagy programrendszerek, programkönyvtárak írásánál van jelentősége, hiszen kis programoknál általában semmilyen változtatás nem túl bonyolult. A kiterjeszthetőség növelése szempontjából a két legfontosabb alapelvnek a tervezési egyszerűséget és a decentralizációt (minél önállóbb modulok, osztályok, komponensek létrehozását) tekinthetjük (lásd a következő részt, amely az objektumorientált programkönyvtár-tervezésről szól).

Újrafelhasználhatóság

Manapság, amikor rendkívül nagy a kifejlesztendő szoftvertermékek mennyisége, nem engedhető meg, hogy mindig minden rendszert „a nulláról" kelljen felépíteni. Egyre nagyobb az igény az újrafelhasználható komponensek[8] iránt.

Ha egy szoftverrendszert úgy tervezünk meg, hogy szem előtt tartjuk az újrafelhasználhatóság szempontjait, az némileg több befektetést igényel, de a következő fejlesztésnél mindez megtérül, mivel jó néhány dolgot készen kapunk.

15.1.3. A programkönyvtárakra vonatkozó egyedi követelmények

Az alábbi követelmények kifejezetten azokra a szoftvertermékekre vonatkoznak, amelyeket nem végfelhasználók, hanem más szoftverfejlesztők használnak. Mivel a programkönyvtárak kifejezetten ilyen célból készülnek, mindez különösen érvényes rájuk nézve.

A könyvtárakban található osztályoktól az alábbi tulajdonságok várhatók el:

- könnyű és intuitív használat,
- azonnali, széleskörű felhasználhatóság szoftverrendszerek széles körében,
- magas szintű dokumentáció,
- hordozhatóság és kompatibilitás.

A következőkben részletesen áttekintjük ezeket a követelményeket.

Könnyű és intuitív használat

A programkönyvtárak által felkínált szolgáltatásoknak könnyen megérthetőnek, és néhány használat után jól megjegyezhetőnek kell lenniük.

A könnyű felhasználhatóságot többféle szempontból vizsgálhatjuk. Az egyik ezek közül a szolgáltatások elkülöníthetősége. Amennyiben nagyon sok hasonló szolgáltatás található egy könyvtárban, a felhasználó számára igen nehéz a pontosan megfelelő kiválasztása. Általában elmondható, hogy előnyösebb kevés, majdnem teljesen ortogonális[9] szolgáltatás definiálása – minden esetben egy jól meghatározott szerep betöltésére, az optimális megvalósítást követve.

[8] Komponensen több dolgot szoktak érteni, például van, aki a futási időben megfelelő interfészeken (például CORBA, COM, EJB) keresztül összekapcsolódni képes szoftverelemeket hívja komponenseknek, van aki pedig már egy programnyelvi szabványos típust (például C++ Standard Template Library vagy Java API-beli osztályt) komponensnek hív. Az elnevezés mindkét összefüggésben megállja a helyét, mi ez esetben inkább a forráskód szintjén összekapcsolható komponenseket értjük alatta.

[9] Különböző funkcionális szempontokat, dimenziókat megvalósító, lásd aspektusorientált programozás és nézőpontok többdimenziós szétválasztása

Az általánosság (a felhasználási lehetőségekhez való alkalmazkodás) és az egyszerűség (minden feladatot elemi műveletek kombinálásával lehet elvégezni) ellentmondanak egymásnak. A profi programozóknak szükségük van a primitív műveletekre a hatékonyság miatt, az alkalmi programozóknak viszont a kényelmesség miatt szükségük van általánosabb összetevőkre és a gyakran használt konvenciókhoz való alkalmazkodásra.

Egy programkönyvtár használatát szintén nagymértékben elősegíti az egységes elnevezési konvenció. Ez következetes névhasználatot jelent, kialakítása a szolgáltatások szintjén pedig azon alapul, hogy kategorizálnunk kell a megvalósított adattípusokon elvégezhető műveleteket, például így:

- Az adatelemek lekérdezésére/elérésére használt szolgáltatások neve kezdődjön a *get* előtaggal.
- Az adatelemek tárolására használt szolgáltatások neve kezdődjön a *put* előtaggal.
- Az adatelemek törlésére használt szolgáltatások neve kezdődjön a *remove* előtaggal.
- A logikai értéket visszaadó szolgáltatások neve kezdődjön az *is* előtaggal.

Természetesen ez a lista még tetszőlegesen folytatható. Ehhez még csak annyit tennénk hozzá, hogy ezen szavak mind az angolból származnak, mivel a programozás terén kétségtelenül az angol nyelv az egyeduralkodó.

Ezek alapján már könnyen kialakítható egy egységes elnevezési rendszer. Az előtagokhoz hasonlóan az adattípusok azonos tulajdonságaira is használjunk egységes elnevezéseket, például így:

- Adott struktúra kapacitására: *capacity*.
- Adott struktúra elemszámára: *count*.
- Adott adatstruktúra üres mivoltára: *empty*; stb.

Így például olyan szóösszetételeket használhatunk szolgáltatások elnevezésére, mint a *setCapacity*, *getCount*, vagy *isEmpty*. Megjegyzendő, hogy a szóösszetevők nagybetűvel kezdése tovább könnyíti névkonvenciónk elfogadását.

Ezen példák alapján a következő irányelveket fogalmazhatjuk meg:

- A név legyen rövid (rendszerint egy szó), de beszédes.
- Egy szolgáltatás neve ne tartalmazzon utalást a szolgáltatást tartalmazó osztályra (kivéve többszörös öröklődés esetén az átnevezésekkor).
- Az osztályok neve mindig főnév vagy főnévi szerkezet legyen.
- A parancsok (eljárások) nevei legyenek (felszólító módú) igék, esetleges főnévi kiegészítővel.
- A nem logikai lekérdezések nevei legyenek főnevek vagy főnévi szerkezetek.
- Egy logikai lekérdezés neve legyen olyan melléknév, amely igen/nem választ sugall, vagy használjuk az *is* előtagot.
- A szolgáltatás nevét a célobjektum szemszögéből lehessen értelmezni, mivel egy szolgáltatás hívásakor mindig létezik egy objektum (a célobjektum), aminek a szolgáltatásáról szó van.
- Az összetartozó művelet- és lekérdezéspárok elnevezéseinek szótöve legyen azonos (extendible – extend).

Azonnali, széleskörű felhasználhatóság

Az újrafelhasználhatóság a szoftvertermékek azon képessége, hogy egészben vagy részben újrafelhasználhatók új alkalmazásokban, szoftverrendszerek széles körében.

Azt a tapasztalatot szeretnénk hasznosítani, hogy a szoftverrendszerek sok eleme közös mintát követ. El kell kerülni, hogy a már megoldott problémákat újra kidolgozzuk.

Ez a kérdés különösen fontos, nem is elsősorban egy egyedi programtermék előállításánál, hanem ha a szoftverkészítés globális optimalizációját tekintjük, hiszen minél inkább lehetőségünk van arra, hogy újrafelhasználható komponensek segítségével oldjuk meg feladatainkat, annál több energiánk marad a többi minőségi jellemző javítására is (azonos költség mellett).

Magas szintű dokumentáció

A jól használható programkönyvtárak különösen fontos jellemzője a pontos, jól szervezett dokumentáció.

Ennek feladata, hogy a felhasználó (programozó) könnyen eligazodjon a felajánlott szolgáltatások sokaságában, és a rendelkezésre álló lehetőségeket optimálisan használhassa ki.

A dokumentáció megírásánál ezért vegyük figyelembe a következő szempontokat:

- A forrásszöveg nem elég absztrakt. A felhasználó számára fontos információkon kívül az alacsonyszintű implementációt is tartalmazza, ezért használata túl sok információ átfésülését jelentené, vagy arra sarkallná a programozókat, hogy nem nyilvánosnak szánt implementációs lehetőségeket is kihasználjanak. Előnye viszont, hogy mindig aktuális.
- Különálló dokumentáció esetén nem biztosított a szoftver és a dokumentáció konzisztenciája. Előnye, hogy csak a felhasználó számára fontos dolgokat tartalmazza.

Ezen két szempontot figyelembe véve megfogalmazhatjuk az ajánlott „belső dokumentáció" elvét: a dokumentáció a programszövegbe legyen beágyazva. Ennek nagy előnye, hogy a forrásszöveg és a dokumentáció mindig konzisztens marad (mivel elvileg a kód módosítása közvetlenül együtt jár a dokumentáció megváltoztatásával is), valamint hogy a dokumentáció előállítása segédprogramokat felhasználva automatizálható (például javadoc)

Hordozhatóság és kompatibilitás

A hordozhatóság arra vonatkozik, mennyire könnyű a programot más géphez, konfigurációhoz vagy operációs rendszerhez – általában különböző környezetekben való futtatáshoz – átalakítani.

A kompatibilitás azt mutatja meg, milyen könnyű a szoftvertermékeket egymással kombinálni. A programokat általában nem légüres térben fejlesztjük, a hatékonyságot nagyságrendekkel növelheti, ha a készített szoftver egyszerűen összekapcsolható más rendszerekkel, valamint jól illeszkedik a háttér-infrastruktúrához (operációs rendszer, adatbázis-kezelő, webkiszolgáló, 3D API, XML séma stb.).

A hordozhatóság és a kompatibilitás gyakran ellentmondanak egymásnak, ugyanis ha például egy programkönyvtárnak már a tervezésekor szem előtt tartották egy bizonyos operációs rendszer minden lehetőségének hatékony kihasználását, akkor nyilvánvalóan egy másik operációs rendszerre már a sima átültetés is nehézségekbe ütközhet, a hatékonyság mértéke pedig egészen biztosan nagyon különböző lesz. Ugyanez fordítva is igaz: ha egy API-t maximálisan hordozhatóra tervezünk, akkor valószínűleg nem tudjuk maximálisan kihasználni az egyes környezetek adta lehetőségeket.

Az egészséges kompromisszum létrehozásánál a legfontosabb az adott könyvtár fő céljának szem előtt tartása, és ennek megfelelően annak eldöntése, hogy inkább a hordozhatóság vagy a kompatibilitás élvezzen előnyt.

15.1.4. A követelmények teljesítésének feltételei

A fenti követelmények következményeképpen az alábbi tulajdonságok várhatók el egy jó programkönyvtártól:

- Konzisztencia – a könyvtár minden komponense (típusok, osztályok, modulok és csomagok) átfogó, koherens tervezés eredménye kell hogy legyen, és számos szisztematikus, explicit és egységes konvenciót kell követnie.
- Jó minőségű, homogén komponensek kellenek.
- A tervezéskor alkalmazzunk objektumorientált megközelítést, hiszen az az adatabsztrakció támogatása miatt különösen alkalmas programkönyvtárak készítésére.

A fenti kritériumok egy részének egyidejű teljesítése igen nehéz. Könnyen lehetséges például, hogy a legáltalánosabb algoritmus nem a leghatékonyabb sok speciális, de gyakran előforduló esetben.

A követelmények jelentős csoportjának – gondoljunk itt elsősorban az újrafelhasználhatóság, kiterjeszthetőség és kompatibilitás szempontjaira – megvalósítását legjobban azzal támogathatjuk, ha könyvtárainkat a 9. fejezetben tárgyalt modularitás elveinek megfelelően tervezzük. Mint ahogy azt már ott is említettük, az objektumorientált megközelítés megfelelő eszközt ad a kezünkbe a moduláris tervezés kiegészítésére és az ott fellépő problémák hatékony kezelésére, ezáltal kiválóan alkalmas jó minőségű programkönyvtárak előállítására. A továbbiakban rövid áttekintést adunk az objektumorientáltságról a programkönyvtárak szemszögéből.

15.2. Objektumorientált programkönyvtár-tervezés

Egy objektumorientált programkönyvtár használhatóságát, minőségét, kiterjeszthetőségét és újrafelhasználhatóságát igen jelentős mértékben befolyásolja az, hogy hány osztályból áll (a lehetőségekhez és a feladathoz mérten), ezek az osztályok mekkorák, és milyen szerkezetű hierarchiában helyezkednek el, milyenek az egymáshoz fűződő kapcsolataik.

Ennek megfelelően a következő néhány oldalon felvázoljuk, milyen megközelítést érdemes választani, amikor a programkönyvtárak osztályhierarchiáját megtervezzük. Ezután áttekintjük, hogy egy osztályon belül célszerűen hány szolgáltatás szerepelhet, valamint hogy

a szolgáltatások kialakításánál milyen paraméterszámot jó választanunk (természetesen az adott feladat jellemzőinek figyelembe vételével). Végül felsorolunk néhány jellemző osztályfajtát az osztályhierarchiában való elhelyezkedésük és fő technikai jellemzőik alapján.

15.2.1. Osztályhierarchia

Egy programkönyvtár elkészítése objektumorientált megközelítés esetén egy osztályhierarchia kialakítását is jelenti. A használhatóság szempontjából nem lényegtelen ezen hierarchia bonyolultsága vagy egyszerűsége.

Az objektumorientált szemléletnek megfelelően igyekezzünk minél általánosabb osztályokat bevezetni, és az öröklődést felhasználva hozzuk létre a szükséges speciális osztályokat. Ezáltal az osztályhierarchia karbantartása is leegyszerűsödik.

Általánosítás

Az alábbi, Objective-C nyelvű példaprogramban szeretnénk szemléltetni az általánosítás (generalization) helyes megközelítését.

E két rövid programrészlet egy általános ablakot és egy párbeszédablakot megvalósító osztály létrehozását szemlélteti.

```
// Helytelen megközelítés
@interface MainWindow : Object
  // ...
  - show;
  - moveToX: (int) x Y: (int) y;
  - onExit;
@end

@interface DialogBox : Object
  // ...
  - show;
  - moveToX: (int) x Y: (int) y;
  - onOK;
@end
```

```
// Helyes megközelítés
@interface Window : Object
  // ...
  - show;
  - moveToX: (int) x Y: (int) y;
@end

@interface MainWindow : Window
  // ...
  - onExit;
@end

@interface DialogBox : Window
  // ...
  - onOK;
@end
```

Az első esetben helytelenül, az általánosítás alkalmazása nélkül hozunk létre két osztályt, amelyek hasonlítanak egymásra, mivel bizonyos szempontból ugyanannak az általános fogalomnak (ablak) a konkrét, specializált megtestesítői.

Mivel ebben az esetben nem hoztunk létre általános ablak osztályt, bizonyos jellemzők (*show* és *moveToX:Y:* metódusok) mindkét helyen, egyformán szerepelnek. Ez azt eredményezi, hogy ha ezeken a szolgáltatásokon változtatni szükséges, akkor ezeket a változásokat párhuzamosan két helyen kell végrehajtani, ami egyrészt plusz munkát eredményez, tehát

nehezíti a karbantarthatóságot, másrészt, ha programozói figyelmetlenségből csak az egyik helyen változik meg az adott viselkedés, akkor ellentmondás lép fel, ami rontja a program minőségét.

A második programrészletben már szerepel a *Window* osztály, a másik kettő ennek a leszármazottjaként jön létre. Ez egyrészt a közös viselkedés egy helyre való kiemelését jelenti, ami javítja a karbantarthatóságot és a program minőségét, másrészről az újrafelhasználhatóságot is segíti, ugyanis egy olyan általános ablak típust hoztunk ily módon létre, amelyből kiindulva később tetszőleges egyéb ablakosztályokat származtathatunk.

Konkretizálás

Az alábbi Smalltalk [Gol 83] nyelvű példaprogram szintén arra példa, hogy megfelelő általános ősosztályok megválasztásával a konkrét típusok konkretizálással (specialization) jóval egyszerűbben hozhatók létre, mintha egymástól függetlenül, az öröklődés lehetőségeivel nem élve kellene megvalósítani azokat.

```
class Allat superclass Object
  instance variables
    szin
    szuletesiEv
    elohely
  class methods
    new ...
  instance methods
    eszik ...
    iszik ...
    alszik ...

class Kutya superclass Allat
  instance variables
    veszettsegElleniOltasDatuma
  class methods
    new ...
  instance methods
    setal ...

class Tehen superclass Allat
  instance variables
    napiTejAdag
  class methods
    new ...
  instance methods
    kerodzik ...
```

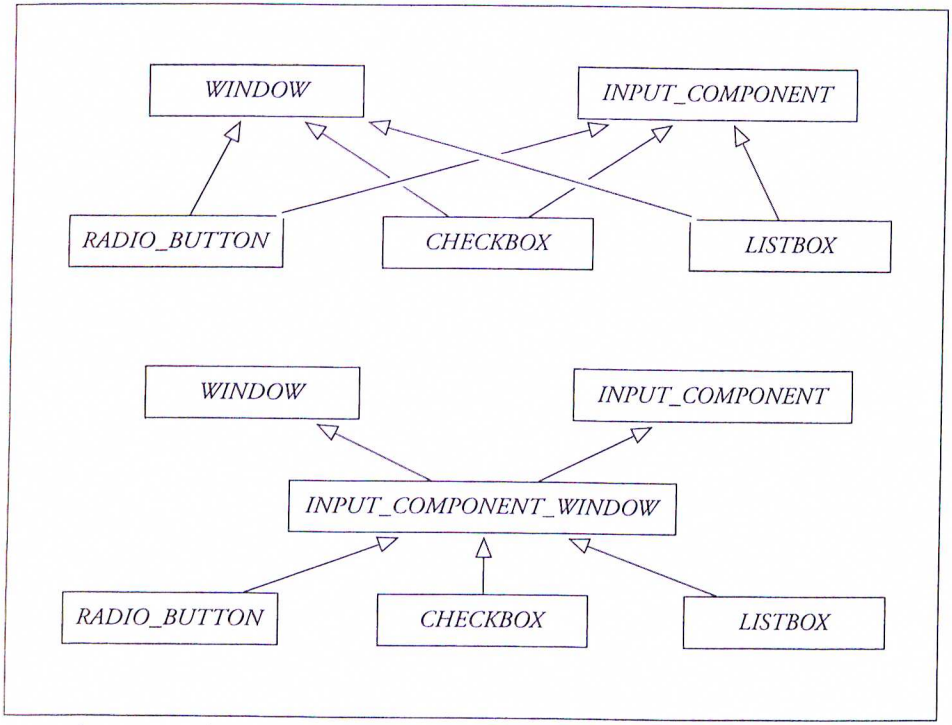
A példa alapján jól látható, hogy lényegesen rövidebb kódot kapunk, ha az általánosítás–specializáció útját követjük, hiszen ha nem emeltük volna ki a közös részt, akkor azt duplán kellett volna beírnunk. Ráadásul ha például az aranyhörcsögöt is szeretnénk példánkban implementálni, akkor még többet nyerünk a közös rész kiemelésével, hiszen azt három helyett csak egyszer kell megadnunk.

Egyszerű vagy bonyolult hierarchia

Egy új osztály bevezetésének további oka lehet az osztályhierarchia egyszerűsítése. Ennek főleg többszörös öröklődés alkalmazása esetén van jelentősége, hiszen egy további csomópont osztály bevezetése kevesebb öröklődési relációt eredményez.

Az alábbi Eiffel nyelvű példa célja ennek bemutatása. A példát a 15.2. ábrán egy UML diagramon is bemutatjuk, az osztályhierarchiák áttekinthetőségének javítása céljából.

```
-- Helytelen megközelítés
class RADIOBUTTON inherit WINDOW INPUT_COMPONENT
  -- ...
end

class CHECKBOX inherit WINDOW INPUT_COMPONENT
  -- ...
end

class LISTBOX inherit WINDOW INPUT_COMPONENT
  -- ...
end

-- Helyes megközelítés
class INPUT_COMPONENT_WINDOW
  inherit WINDOW INPUT_COMPONENT
  -- ...
end

class RADIOBUTTON inherit INPUT_COMPONENT_WINDOW
  -- ...
end

class CHECKBOX inherit INPUT_COMPONENT_WINDOW
  -- ...
end

class LISTBOX inherit INPUT_COMPONENT_WINDOW
  -- ...
end
```



15.2. ábra: Az Eiffel példaprogram két változatának UML diagramjai

A példából is látható, hogy ebben az esetben a kód mérete nem csökkent, de áttekinthetősége jelentősen megnőtt azáltal, hogy jóval kevesebb öröklődési reláció keletkezett, valamint logikailag is átláthatóbb lett a program.

Ez utóbbi oka az, hogy egy olyan típusegyüttest emeltünk ki külön osztályba, mely több helyen is közösen szerepel, tehát erősen valószínűsíthető, hogy a program logikájában önálló fogalomként is megállja a helyét.

15.2.2. Az osztályok mérete

A továbbiakban áttekintünk néhány fogalmat az osztályok méretével kapcsolatban, majd néhány statisztikai adat segítségével támpontot igyekszünk nyújtani a programkönyvtárakban lévő osztályok méretének megválasztásához.

Sima, közvetlen és növekményes méret

Egy osztály méretét a következő módokon lehet meghatározni:

- Közvetlen méret = az osztály által bevezetett közvetlen új (nem absztrakt és nem örökölt vagy újradefiniált) szolgáltatások száma.

- Növekményes méret = az örökölt és újradefiniált szolgáltatások száma.
- Sima (teljes) méret = az osztály által megvalósított szolgáltatások száma = közvetlen + növekményes méret.

Az alábbiakban egy C# [Sch 02] nyelvű példán mutatjuk be ezeknek a mérőszámoknak az alkalmazását.

```
using System;

public class Telefonkeszulek {
  public string Telefonszam() {
    string telefonszam;
    // Telefonszám értékének meghatározása.
    return telefonszam;
  }
  public void HivasMegszakitas() {
    telefonkagylo.Letesz();
  }
}

public class HordozhatoTelefonkeszulek : Telefonkeszulek {
  public void HivasMegszakitas() {
    nagyPirosGomb.Megnyom();
  }
  public int AntennaHossz() {
    int hossz;
    // Hossz értékének meghatározása.
    return hossz;
  }
}
```

Ebben a programban a *HordozhatoTelefonkeszulek* osztály közvetlen mérete 1, az újonnan bevezetett *AntennaHossz* metódus miatt. Az újradefiniált *HivasMegszakitas* metódust is beszámítva a növekményes méret 2, és az öröklött *Telefonszam* metódussal együtt megkapjuk a sima méretet, ami 3.

Külső és belső méret

Másik szempont lehet, hogy az adott mérőszám tartalmazza-e a nem exportált jellemzőket: eszerint megkülönböztetünk külső és belső méretet (a külső csak a publikus tagokat, míg a belső az összeset magában foglalja).

A külső és a belső méret tehát akkor nem tér el jelentősen egymástól, ha a publikus jellemzőit az osztály nem saját privát eszközeivel implementálja, hanem azok megvalósítását más osztályokhoz delegálja.

Kezelhetőség miatti felső korlát

Ha szabályokat szeretnénk megfogalmazni arra vonatkozóan, hogy mekkora legyen egy osztály, akkor meg kell határozni, hogy mely metrika szerint szeretnénk ezen követelményeket megadni.

A teljes méretre nyilván nincs felső határ, mivel ez nagy mértékben függ az ősosztályok számától és bonyolultságától. A közvetlen, illetve a növekményes méret az, ami szerepel az osztály szövegében. Természetes megközelítésnek tűnik, ha a növekményes mérettel dolgozunk, hiszen ez az, ami ténylegesen az osztály forráskódjában szereplő jellemzőkre vonatkozik.

Egy adott méreten felül az osztály kezelhetetlenné válik. Ezt a problémát két módon lehet megközelíteni:

- *Minimalista nézőpont*: egy programkönyvtár osztálya a megvalósított típusnak csak a legalapvetőbb szolgáltatásait tartalmazza, redundáns (azaz ami alapvető szolgáltatások használatával is megvalósítható) metódusokat pedig nem. Ez persze a programkönyvtár írójának kedvez, hiszen kevesebb munkát és egyszerűbb karbantartást jelent.
- *„Bevásárlólista" nézőpont*: minden olyan szolgáltatást vegyünk bele az osztályunkba, ami az alábbi szempontoknak megfelel:
  - A szolgáltatás beleillik az absztrakt adattípus megvalósítási sémájába.
  - Nem rontja el az osztály helyességét, azaz az osztályinvariánst.
  - Hasznos funkcionalitást valósít meg.
  - Nem duplikál már meglévő szolgáltatást.
  - Megfelel a könnyen kezelhetőség követelményeinek.

A gyakorlatban általában a programozási nyelvek beépített könyvtáraitól elvárt szempont a minimalista nézőpont, azaz hogy egy műveletet csak egyfajtaképpen lehessen hatékonyan elvégezni, míg az egyéb programkönyvtárak tervezésekor a felhasználás megkönnyítése érdekében inkább a bevásárlólista nézőpont kell, hogy érvényesüljön.

A következő példaprogram a bevásárlólista nézőpontot mutatja be:

```
(* Objective CAML példaprogram *)
module Tablazat :
sig
  val cella_szinezes : int int int -> unit
  val sor_szinezes : int int -> unit
  val oszlop_szinezes : int int -> unit
  val mindent_szinez : int -> unit
  (* egyéb függvények *)
end =
struct
  (* implementáció *)
end
```

Szolgáltatások száma	Osztályok gyakorisága (%)
0 – 5	40 – 60
6 – 10	10 – 20
11 – 15	5 – 10
16 – 20	5 – 10
21 – 40	5 – 15
41 – 80	5 – 10
81-től	0 – 5

15.1. táblázat: Eiffel programkönyvtári osztályok méretének eloszlása

A példabeli *Tablazat* típus egy táblázatkezelő egyszerűsített modelljét valósítja meg. Egyaránt tartalmaz olyan funkciót, mellyel az egész táblázatot egyszerre beszínezhetjük, valamint soronként és oszloponként színező, sőt olyan metódus is megtalálható a modulban, amellyel csak egyetlen adott cellát színezhetünk be. A bemenő paraméterek értelemszerűen a sorok és oszlopok sorszámát, valamint a színkódot tartalmazzák.

Rendező elvek

Túl sok szolgáltatás a programkönyvtár használatának megtanulhatóságát is kedvezőtlenül befolyásolja, ennek kezelhetősége érdekében a következő rendező elvek betartása ajánlott:

- Minden szolgáltatást egy jól meghatározott szerződéssel adjunk meg, csak a specifikációjuk, nem pedig az implementációjuk alapján használjuk fel őket.
- Az osztálydokumentáció egységes és könnyen áttekinthető formátumú legyen, tartalmazza minden szolgáltatás szerződését és rejtse el az implementációs részleteket.
- A szolgáltatások nevei szigorú elnevezési konvenciót kövessenek.
- A szolgáltatásokat csoportosítsuk pontosan definiált kategóriák szerint, ezen kategóriák minden osztálynál egyezzenek meg, sorrendjük a dokumentációban legyen azonos, egy kategórián belül pedig a szolgáltatások névsorrendben következzenek.

Példaadatok

Általában kijelenthető, hogy egy 80 szolgáltatásnál többet tartalmazó osztály esetében már érdemes az osztályhierarchia továbbbontását meggondolni.

A 15.1. táblázatban szereplő adatok az Eiffel programkönyvtári osztályok méretének alapján az ajánlott méreteloszlást mutatják.

15.2.3. A szolgáltatások mérete

Mivel a programkönyvtárak használata a megvalósított szolgáltatások meghívásából áll, igen fontos, hogy a használó pontosan ismerje egy szolgáltatás paramétereinek számát, típusát és sorrendjét. Ezért a felhasználás szempontjából egy szolgáltatás méretét tulajdonképpen a megadandó paraméterek száma jellemzi.

A méret erősen függ a programkönyvtár feladatától is, hiszen egy grafikus felületet megvalósító, vagy statisztikai számításokat végző osztály metódusai rengeteg argumentumot igényelhetnek.

Nagyszámú argumentum esetén érdemes azokat a következő két csoportra osztani:

- Paraméter: olyan argumentum, amelynek értékével valamilyen műveletet kell elvégezni.
- Opció: olyan argumentum, amely akár el is hagyható, ugyanis rendelhető hozzá egy alapértelmezett érték. Csak a paraméterek feldolgozását befolyásolja. Egy szolgáltatás fejlesztése során a paraméterlista lehetőleg ne változzon, míg új opciókat nyugodtan fel lehet venni.

Ezekből következik, hogy a szolgáltatás méretének csökkentése érdekében csak paramétereket vegyünk fel az argumentumlistába. Az opciókat pedig külön attribútummal reprezentáljuk, és adjunk hozzá új beállító, illetve lekérdező szolgáltatásokat. Ez az úgynevezett opció-beállítási módszer. Ez persze felveti az opciót reprezentáló új attribútum globalitásának kérdését, sőt, növeli az osztály méretét is.

Az alábbi PHP nyelvű példaprogram az opció-beállítási módszert hivatott illusztrálni:

```
<?php
class FileKezelo {                    // Hagyományos osztály
  function Megnyit($filenev ,$mod, $path_beszamitasa) {
    return fopen($filenev, $mod, $path_beszamitasa);
  }
}

class FileKezeloOpciokkal {  // Opciókat külön kezelő osztály
  var $m_mod;
  var $m_path;
  function ModBeallit($mod) {
    $this->m_mod = $mod;
  }
  function PATHBeallit($path_beszamitasa) {
    $this->m_path = $path_beszamitasa;
  }
  function ModLekerdez() {
    return $this->m_mod;
  }
  function PATHLekerdez() {
    return $this->m_path;
  }
  function Megnyit($filenev) {
    return fopen($filenev, $this->m_mod, $this->m_path);
  }
}
// (A program a következő oldalon folytatódik.)
```

Argumentumszám	Szolgáltatások gyakorisága (%)
0	50 – 60
1	30 – 35
2	5 – 15
3	0 – 5
4	0 – 5
5-től	0 – 5

15.2. táblázat: Paraméterszámok eloszlása az Eiffel programkönyvtári osztályok szolgáltatásainál

```
// (Folytatás az előző oldalról.)
// Hagyományos osztály használata
$fk = new FileKezelo;
$fk->Megnyit("/etc/passwd", "r", 0);

// Opciókat külön kezelő osztály használata
$fko = new FileKezeloOpciokkal;
$fko->ModBeallit("r");
$fko->BlokkmeretBeallit(0);
$fko->Megnyit("/etc/passwd");
?>
```

A programban a *FileKezelo* osztály egyetlen metódust tartalmaz, melynek három argumentuma van, míg a *FileKezeloOpciokkal* osztály ugyanazon metódusa csak egy paramétert tartalmaz, a másik két korábbi argumentum pedig opcióként külön tagváltozóba került, és lekérdező, valamint beállító metódusokat is adtunk hozzájuk.

Példaadatok

A szolgáltatások átlagos paraméterszáma rendszerint 0,5 körül van. Ötnél több paraméter esetén már meggondolandó, hogy nem lehetne-e a szolgáltatást tovább bontani. A szolgáltatások lekérdezés–parancs aránya kb. 60–40 %.

A 15.2. táblázatban szereplő adatok az Eiffel programkönyvtári osztályok szolgáltatásai méretének alapján az ajánlott méreteloszlást mutatják.

15.2.4. Az osztályok fajtái

A továbbiakban a programkönyvtárakban fellelhető osztályok néhány fajtáját jellemezzük.

Konkrét típusok

Minden programkönyvtár-hierarchiában találunk olyan típusokat, melyek alapvető adatszerkezeteket ábrázolnak (lista, karakterlánc, dátum, komplex szám stb.). Ezeket nevezzük konkrét típusoknak.

A konkrét típusok a következőképp jellemezhetők:

- szoros illeszkedés egy konkrét fogalomhoz és a megvalósítás módjához;
- érthetőség, önálló használhatóság;
- erősen implementációfüggők, ezért hatékony a megvalósításuk, de bármilyen módosításukkor minden felhasználói kódot újra kell fordítani;
- minimális mértékben függnek más osztályoktól;
- különállóan (izoláltan) is fordíthatók és használhatók, ugyanakkor nem lehet tőlük örökölni;
- konkrét elnevezésük az absztrakt tulajdonság ellenpárja, mivel semmilyen általános vagy elvont (absztrakciós) tulajdonsággal nem rendelkeznek, inkább az alacsonyszintű programozásban használatosak.

Az alábbi példaprogram a Java nyelv azon lehetőségét mutatja be, amelynek segítségével olyan osztályokat hozhatunk létre, amelyek olyannyira konkrétak, hogy explicit módon megtiltjuk a belőlük való származtatást. Ezt a **final** kulcsszó használatával érhetjük el.

Az első kódrészletben a véglegességet egy egész osztályra alkalmazzuk.

```
// konkrét osztály
final class SakkAlgoritmus {
  // . . .
}
```

Ha egy olyan osztályt szeretnénk lefordítani, amely a fenti osztályt ősként jelöli meg, a következő hibaüzenetet kapjuk a fordítótól:

```
SakkAlgoritmus.java:6: Can't subclass final classes: class SakkAlgoritmus
class MegJobbSakkAlgoritmus extends SakkAlgoritmus ...
      ^
1 error
```

A következő példában pedig csak egy adott metódust véglegesítünk:

```
// Ősként használható osztály végleges metódussal.
class SakkAlgoritmus {
    // ...
    final void kovetkezoLepes(SakkFigura mozgatandoFigura,
                              TablaPozicio ujPozicio) {
        // ...
    }
}
```

Ebben az esetben az osztályból származtathatunk, de a **final**-ként jelölt metódust nem definiálhatjuk felül.

Absztrakt típusok

Absztraktnak olyan osztályokat nevezünk, amelyekre azért van szükség, hogy az osztályhierarchiában egy fontos logikai egységet testesítsenek meg, de példányosításukra közvetlenül nem kerülhet sor, kizárólag leszármazottaikon keresztül. Szerepük rendszerint egy adott interfész bevezetése, például halmaz, lista, fa stb.

Az absztrakt típusokat konkrét, belőlük származtatott típusok segítségével lehet implementálni. Az absztrakt típusok tulajdonságai:

- azonos elérési módhoz többfajta implementáció is létezhet;
- hatékony tárkihasználás és elfogadható futási idő a virtuális metódusok segítségével.

Az absztrakt osztályok létrehozását több nyelv is támogatja megfelelő kulcsszavakkal. Például a Java, az Ada 95, a C# nyelvek az **abstract** kulcsszót tartalmazzák, a C++ pedig az *= 0* végződéssel jelöli, ha egy tagfüggvény „pure virtual" (tisztán virtuális), azaz absztrakt. Általában lehetőség van arra, hogy megjelöljük, absztrakt vagy konkrét műveletről van-e szó.

Az alábbi Ada 95 nyelvű példaprogram egy absztrakt halmaz típust mutat be, egy konkrét megvalósítással (hasítótábla alapú halmaz típus).

```
package Halmaz is
  type THalmaz is abstract tagged null record;
    function Meret(h: in THalmaz) return Integer is abstract;
    function UresE(h: in THalmaz) return Boolean;
    procedure Hozzaad(h: in out THalmaz, elem: in Float) is abstract;
    procedure Kitorol(h: in out THalmaz, elem: in Float) is abstract;
    -- Egyéb absztrakt műveletek specifikációja.

  type THashTablaHalmaz is new THalmaz with private;
    function Meret(h: in THashTablaHalmaz) return Integer;
    procedure Hozzaad(h: in out THashTablaHalmaz, elem: in Float);
    procedure Kitorol(h: in out THashTablaHalmaz, elem: in Float);
    -- Egyéb műveletek specifikációja.
  private
    -- A THashTablaHalmaz típus megvalósítása.
end Halmaz;

package body Halmaz is
  -- A THalmaz típus műveleteinek megvalósítása.
  function UresE(h: in THalmaz) return Boolean is
    begin
      return Meret(h) = 0;
    end UresE;
```

```
   -- A THashTablaHalmaz típus műveleteinek megvalósítása.
function Meret(h: in THashTablaHalmaz) return Integer is
   m : Integer := 0;
  begin
     -- A méret kiszámítása.
    return m;
  end Meret;

procedure Hozzaad(h: in out THashTablaHalmaz, elem: in Float) is
  begin
     -- Elem hozzáadása.
  end UresE;

procedure Kitorol(h: in out THashTablaHalmaz, elem: in Float) is
  begin
     -- Elem eltávolítása.
  end UresE;

  -- Egyéb műveletek megvalósítása.
end Halmaz;
```

Csomópont típusok

A csomópont típusok az öröklési hierarchia belső pontjait alkotják. Ahogy azt már korábban bemutattuk (lásd 15.2.1), az osztályhierarchia egyszerűsítésénél játszanak fontos szerepet. A csomópont osztályok jellemzői a következők:

- Az ősosztályok szolgáltatásait implementálják, ugyanakkor annak interfészét kibővítik virtuális metódusokkal is, melyeket maguk is implementálnak.
- Függnek az őseiktől.
- Lehet belőlük örökölni.
- Példányosíthatók.

Példa lehet egy geometriai alkalmazásban a négyszög típus, mely leszármazottja a síkidom osztálynak, ugyanakkor tovább finomítható (paralelogramma, trapéz stb.).

Kövér interfészek

Ez egy olyan interfésztípus, amely nem függ más osztályoktól, rengeteg szolgáltatást vezet be, ezek közül pedig csak a legfontosabbakhoz ad implementációt. A speciálisabb szolgáltatásokat virtuálisként deklarálja, és hogy lehessen példányosítani, ad hozzá egy üres implementációt, ami rendszerint egy hibaüzenetet ad, jelezvén, hogy az adott szolgáltatást még nem implementálták.

Példa: általános tároló (container) osztályok, melyek az elemek tárolását saját maguk valósítják meg, viszont a tárolt elemek rendezésére, vagy adott feltételnek megfelelő részhalmazok felsorolására már nem adnak implementációt.

Alkalmazáskeretek

Ezen absztrakt osztályok tulajdonképp egy mini alkalmazást valósítanak meg. Az osztályon belül csak a programlogika implementált, minden egyéb alapműveletet elvégző és ezen működést paraméterező metódus virtuális.

Példa: szűrő osztály, amely egy bemeneti adatfolyam minden elemén végiglépkedve adott feltétel teljesülésekor bizonyos műveleteket elvégez. A bemeneti adatfolyamon történő végigléptetést, adott tulajdonság fennállásának ellenőrzését, illetve a kívánt művelet meghívását és az esetleges hibakezelést maga az osztály valósítja meg.

Leíró osztályok

Egy absztrakt interfész és annak konkrét implementációja közötti kapcsolat a program futása során állandó, sőt, az implementáló osztály mérete is állandó, de felmerülhet az igény adott interfészen keresztül változó méretű osztályok kezelésére is. Erre ad megoldást a leíró osztályok használata. Ekkor ugyanis az implementáció két részre bomlik: az aktuális reprezentációra és a reprezentáció elérését biztosító leíró (handle) objektumra. Ez tulajdonképp a mutató típus objektumorientált megvalósításának tekinthető.

Példa: fájlleíró osztályok segítségével tetszőleges méretű adatfájlt lehet reprezentálni.

15.3. Új paradigmák

A kilencvenes években jelentek meg azok az új elméletek és módszerek, amelyek az OOP-től eltérő módszerekkel, vagy azt kiterjesztve továbbfejlesztették a programozási nyelvek modularitását. Ezek használatával tovább javíthatók a programkönyvtárak minőségi jellemzői.

Példaként említhetjük az aspektusorientált programozást (AOP), melynek lényege, hogy az osztályhierarchiát átszövő aspektusokat külön moduláris egységekbe emeljük ki, vagy a generatív programozást (GP), melynek lényege, hogy egy alkalmazási terület (domain) modellezését követően olyan programkönyvtárakat vagy komponenseket hozunk létre, melyek felhasználásával a konkrét probléma ismeretében egyszerűen és a lehető legteljesebb mértékben lehet a rendszer tervéből programkódot generálni. Ez utóbbival rokon az intencionális programozás (IP), mely a megfelelő módon tervezett és strukturált absztrakciók könyvtáraiból képes teljes kódgenerálásra.

A fenti elméletekre és azok nyelvi megvalósításaira az aspektusorientáltságról szóló fejezetben térünk ki részletesebben.

15.4. Általános programkönyvtárak

Mint korábban már említettük, a magasszintű nyelvek, illetve fejlesztőkörnyezetek az általuk megvalósított szolgáltatásokat rendszerint hozzájuk tartozó általános programkönyvtárak formájában kínálják fel. Ilyenkor maga a programnyelv valójában csak annak alaputasítás- és kulcsszó-készletére korlátozódik, amely önmagában már túl kevés lenne egy mindennapi probléma megoldásához. Ezt úgy is mondhatnánk, hogy egy adott nyelv általános programkönyvtárának szolgáltatásgazdagsága erősen meghatározza a programnyelv felhasználhatóságát. Ilyen magasszintű programnyelv például a Java, amely közismerten elterjedt és rengeteg szolgáltatást tartalmazó általános programkönyvtárral rendelkezik.

Ha jobban belegondolunk, az alacsonyszintű nyelvek – például az assembly – esetében is tulajdonképp előregyártott programkönyvtár-szolgáltatásokat veszünk igénybe – például megszakítások formájában. Ez talán azért nem tűnhet fel elsőre, mert az így használt programkönyvtár nem adathordozón, hanem a számítógép ROM-jában található, ezért mindenféle szerkesztési művelet nélkül hozzáférhetünk a szolgáltatásaihoz.

Az általános programkönyvtárak tehát adott programozási nyelven/nyelvhez biztosítanak olyan alapvető szolgáltatásokat, melyekre szinte minden programozónak szüksége lehet. Jó példa erre a program és a külvilág közti kommunikáció megvalósítása, hiszen adatbevitel és eredménykijelzés nélkül csak igen kevés program tudna megélni. Ugyanakkor egy programkönyvtár szolgáltatásait akár más programnyelvekből is igénybe lehet venni. Ez történhet alacsony szinten, amikor is a lefordított modulokat más nyelvből, de az eredeti nyelv szintaxisának megfelelően hívunk meg (például amikor egy C program egy Pascal szignatúrával definiált külső rutint hív meg, vagy gondoljunk a Java nyelv *native* lehetőségére), vagy a szolgáltatás meghívása történhet egy interpreteren keresztül, de tágabb értelemben gondolhatunk például a CORBA szolgáltatások igénybevételére is, ahol a közös leíró (IDL) nyelvnek köszönhetően teljesen szétválik az adott szolgáltatás definíciója és implementációja.

15.4.1. Adatszerkezetek

Az egyik alapvető dolog, amit egy általános programkönyvtártól elvárhatunk, hogy az a programozó munkáját segítendő, gyakran használt összetett adatszerkezetek és struktúrák megvalósítását kínálja. Ugyanis a kiindulási programnyelv konkrét alaptípusaiból építkező bonyolultabb típusokhoz a nyelv csak a típuskonstrukció eszközeit biztosítja, ugyanakkor ezen összetett típusokra elég gyakran szükségünk van, ami erősen indokolja azok programkönyvtárba való felvételét.

Jó példa erre a Java *Collection Framework* megoldása vagy a C++ *STL* könyvtára, melyek objektumok „gyűjteményét", úgynevezett konténereket valósítanak meg, amilyen például a verem (stack), a kupac (bag) vagy a vektor. A fő tervezési szempont ebben az esetben a konténereket alkotó alaptípusok, az összetett konténertípusok valamint az azokon elvégezhető műveletek ortogonalitása volt. Ez azt jelenti, hogy bármilyen alaptípusból lehet konténertípust képezni, valamint a konténer elemeihez csak kötött módon, az úgynevezett iterátorok segítségével lehet hozzáférni. A konténer elemeinek rendezése például nem a konténer fel-

adata, azt egy tetszőleges külső algoritmus elvégezheti az iterátorokat felhasználva, ugyanis maga a konténer csakis az elemeinek megadását, elérését és módosítását biztosítja.

15.4.2. I/O

Egy másik alapvető programozói feladat az adatok bevitele és a kimenő adatok megjelenítése. Ezt természetesen nem várhatjuk el magától a programnyelvtől, hiszen rendszerint minden adatkivitel és -bevitel eleve az operációs rendszer feladata. Egy általános programkönyvtár tehát csak megkönnyíti az operációs rendszer ezen szolgáltatásainak igénybevételét. Gondoljunk csak bele, hogy szinte minden programnyelv esetében ez történik, amikor fájlokat kezelünk, vagy be, illetve kimeneti csatornákat olvasunk vagy írunk, vagy esetleg amikor számítógéphálózatokon keresztül kommunikálunk.

15.4.3. Memóriakezelés

A programkönyvtár-tervezés egyik kulcsfontosságú kérdése a memóriakezelés. A két alapvető probléma, amire oda kell figyelni:

- memóriafoglalás új objektumok létrehozásakor;
- a memória felszabadítása.

Míg az első az operációs rendszer feladata, a második problémára két megoldási irányzat is született:

- Automatikus szemétgyűjtés (például Java, Eiffel, Smalltalk). Ez rendszerint referenciák bevezetését és a mutató típus megszüntetését jelenti. Ennek előnye, hogy a programozónak nem kell törődnie a memóriakezeléssel, hátránya viszont, hogy a szemétgyűjtés automatizmusa miatt biztos, hogy nem optimális konkrét alkalmazások esetében.
- Objektumok kézzel történő felszabadítása. Hatékonyabb, de sokkal veszélyesebb, hiszen ekkor minden felelőség a programozó kezében van, és még a legapróbb figyelmetlenség is a memória „elszivárgásához", rosszabb esetben pedig nem megismételhető programhibákhoz vezethet.

Mint látható, ezen két probléma jellege természetesen a használt programnyelvtől is függ. Még egyszer szeretnénk itt kihangsúlyozni, mennyire fontos, hogy programkönyvtárunk a memóriakezelés szempontjából biztonságosan és hatékonyan viselkedjen, hiszen kódunk rengeteg felhasználási helye miatt egy esetleges hibának is megsokszorozódik a kihatási területe.

15.5. Programkönyvtárak életciklusa

Mivel a programkönyvtárak olyan szoftvertermékek, melyek alkalmazói programok írását hivatottak megkönnyíteni, az egész életciklusuk alatt a legfontosabb szempont, aminek meg kell felelniük, az újrafelhasználhatóság. A tervezésük során tehát olyan hozzáférési felületet kell definiálni, ami hosszú távon stabil marad és nyitva hagyja a továbbfejlesztés lehetőségét

is. A kezdeti fázis után, ha már több helyen is felhasználták programkönyvtárunkat, egyre fontosabb szerephez jut a karbantartás kérdése. Itt az okozza a problémát, hogy a továbbfejlesztések és bővítések hatására az eddig használt szolgáltatások működése nem változhat meg (kivéve persze, ha a változtatás hibajavítás céljából történt). A továbbiakban végigkísérjük a programkönyvtárak életciklusát és megvizsgáljuk, hogyan tudunk legkönnyebben megfelelni az eddig tárgyalt minőségi kritériumoknak.

15.5.1. Tervezési fázis

Az eddig tárgyalt moduláris és objektumorientált megközelítés alapelveinek alkalmazása mellett még érdemes a következő dolgokat is figyelembe venni:

- Az adatstruktúrák problémája: a változtatásokra, de főleg a bővítésre az adatstruktúrák a legérzékenyebbek. Gondoljunk csak bele, mennyi gondot okozhat, ha egy rekord típust további mezőkkel bővítve megnő a típus tárigénye. Ezért az adatstruktúrák bővítési lehetőségére már a tervezés fázisában is gondolni kell, például úgy, hogy kezdetben üres, használaton kívüli mezőket is felveszünk a típusba, melyeket majd egy esetleges továbbfejlesztés során használunk fel. Persze akkor már azon is érdemes elgondolkodni, hogyan lehet majd az adatstruktúrák verzióit megkülönböztetni. Erre az egyik lehetőség, hogy a várt verzió számát közli az alkalmazói program a könyvtárral vagy minden műveletnél külön paraméterként megadva azt (ez persze csökkenti a hatékonyságot a plusz paraméterátadás miatt), vagy a programkönyvtár adatstruktúráinak használata előtt egy speciális szolgáltatással beállítjuk az igényelt verziót. Ekkor azonban a programkönyvtárnak a feladata, hogy nyilvántartsa az esetleges párhuzamos felhasználók által kért verziószámokat, ami csak bonyolítja a kód megvalósítását. Ilyenkor a legegyszerűbb megoldás, ha minden kimenő adatstruktúra önmaga tárolja saját verziószámát, ez viszont a tárigényt fogja megnövelni. Bár ez a probléma az objektumorientált nyelvek esetében kevesebb gondot okoz, mindenképp érdemes a következő elvet megfogadni:

- Könyvtárunk csak közvetett elérési lehetőséget biztosítson a (globális) változókhoz, illetve mezőkhöz! Ilyen esetekben inkább vezessünk be lekérdező és beállító szolgáltatásokat, ezáltal is elfedve az adatmezők és struktúrák belső megvalósítását, valamint így felügyelhetjük az adatokhoz történő hozzáférést.

- A szolgáltatások elnevezésekor használjuk egyedi azonosítóneveket, például következetesen tegyünk ki minden név elé egy előtagot (például a PVM párhuzamos programozást segítő programkönyvtár szolgáltatásainak nevei mind a *pvm_* előtaggal kezdődnek), vagy ha a nyelv támogatja, vezessünk be külön névteret (a C++-ban a *namespace* kulcsszóval, vagy a Javában a csomagok segítségével).

- A programkönyvtár hozzáférési felülete (API) csak a valóban nyilvános szolgáltatásokat fedje fel, ezáltal a megvalósításhoz használt belső kód változása kevésbé lesz kihatással a felhasználó kódrészekre.

15.5.2. Megvalósítási fázis

Ebben a fázisban a legfontosabb szempont a szolgáltatások hatékony és hibamentes implementálása. A következő alapelvek ehhez nyújtanak segítséget:

- A szolgáltatásokat a legszigorúbb módon definiáljuk. Alaposan gondoljuk meg, hogy egy művelet osztály- vagy objektumszintű legyen, és minél pontosabban adjuk meg a lehető legszűkebb módosítókat. Kívülről nem elérhető szolgáltatásokat például jelöljünk meg privátként, és azt is mindenképp jelezzük (Javában például a *final* kulcsszóval) ha valamely metódust véglegesíteni szeretnénk. A fordító csak így tud optimális kódot generálni.
- Az előző elv folyománya, hogy a megvalósítás csak azokat a műveleteket tegye kívülről is elérhetővé, amelyek dokumentáltan beletartoznak könyvtárunk szolgáltatásába. Így biztosítható, hogy senki ne használjon nem dokumentált, a belső implementációhoz tartozó funkciókat.

15.5.3. Karbantartási fázis

Minden nagyobb programkönyvtár életében ez a leghosszabb fázis. A karbantartás két módon, hibajavítás és továbbfejlesztés formájában történhet. A hibajavítás legfontosabb alapelve, hogy azt mindenképp el kell végezni, nem szabad olyan kifogásoknak engedni, hogy esetleg pont a hibás viselkedésre épülhet valamely felhasználói program. Hiszen a hibát felfedező programozó feladata nem annak kihasználása lenne, hanem jeleznie kéne azt a programkönyvtár készítője felé.

Továbbfejlesztéskor a legfontosabb szempont a verziókezelés megoldása, valamint a módosítások elvégzésekor a kompatibilitás lehető legnagyobb mértékű megtartása. Ebben segíthet a következő alapelvek betartása:

- Új szolgáltatások bevezetésekor meg kell oldanunk, hogy az új verziót használó programok a könyvtár régi verziójával el se induljanak. Erre alkalmazhatjuk a tervezési fázisnál az adatstruktúrák problémájára adott megoldásokat.
- Ha valamely szolgáltatás szignatúrája megváltozik, kompatibilitási okokból a régi változatot is hagyjuk meg. Ilyenkor a régi változat hívásakor rendszerint az újonnan hozzájött paraméter valamilyen alapértelmezett értéket fog kapni, majd hívásátirányításra kerül sor. Eközben akár egy figyelmeztető jelzésre is sor kerülhet, melynek hatására a felhasználó programozók előbb-utóbb áttérnek az új változat használatára. Ez történhet akár a programnyelv támogatásával is (lásd a Java *deprecated* kulcsszavát: az így megjelölt metódusok hívásakor már a fordító is figyelmeztető üzenetet ad).
- Programkönyvtárunk minden újabb verzióját mindig alaposan teszteljük le. Ehhez a fejlesztéssel párhuzamosan mindig készítsünk megfelelő tesztkörnyezetet is, amit a könyvtárral együtt építhetünk ki. Az új funkciók tesztelése érdekében mindig bővítsük tesztkörnyezetünket is, a már meglévő szolgáltatások újratesztelésével pedig az esetleges kereszthatások ellen védekezhetünk.

16 A funkcionális programozási nyelvek elemei

A deklaratív programozási nyelvekben a program, a számítási folyamat leírása deklarációk halmaza. A funkcionális program típus-, osztály- és függvénydeklarációk, illetve definíciók sorozatából, valamint egy kezdeti kifejezésből áll. A program végrehajtása nem más, mint a kezdeti kifejezés kiértékelése. A funkcionális nyelvek matematikai számítási modellje a λ-kalkulus. Ebben a fejezetben áttekintjük a funkcionális programozás leglényegesebb fogalmait, történeti előzményeit, matematikai hátterét, kifejezőerejét, hatékonyságát, a funkcionális stílust, az absztrakciót támogató nyelvi elemeket, az ilyen nyelveken írt programok olvashatóságát, módosíthatóságát, megbízhatóságát.

A funkcionális nyelvek elemeinek tárgyalásakor több nyelv (SML, Miranda, Clean, Haskell) elemeit mutatjuk be példákon keresztül, hogy ezzel is kifejezésre juttassuk, hogy elsősorban a funkcionális programozási stílus, illetve a modern funkcionális programozási nyelvekben előforduló közös nyelvi elemek megismerése a cél [Hud 89, Hor 97, Hor 99b], nem pedig egy konkrét nyelv kimerítő tárgyalása [Cla 95, Har 01, Han 00, Tho 99, Pla 99][1].

16.1. Bevezetés

A funkcionális programozás matematikai alapja a Church által 1932–33-ban kidolgozott számítási modell, a *λ-kalkulus* [Bar 84, Csö 01]. A funkcionális programozási nyelvek szemantikáját leggyakrabban λ-kalkulus segítségével szokták definiálni. Turing megmutatta, hogy a λ-kalkulusban definiálható, a nem negatív egészeken értelmezett és hatékonyan kiértékelhető függvények ugyanazok, mint amelyek az imperatív nyelvek számítási modelljeként ismert *Turing géppel* is kiszámíthatók. Funkcionális programozási eszközökkel tehát minden olyan feladat megoldható, amelyik megoldható imperatív nyelven és viszont. A funkcionális programozás modellje egyidős az imperatív programozás modelljével, a Turing géppel, és az első funkcionális nyelv (a LISP) is kortársa (1956–1962) az első magasszintű imperatív programozási nyelveknek (FORTRAN, ALGOL 60). Mindez nem véletlen, a két szemlélet ugyanis nem ellentétes egymással, hanem sokkal inkább egymást kiegészítő. A programozási feladatok megoldása imperatív és funkcionális gondolkodásmódot egyaránt megkövetel.

A számítógépek teljesítménynövekedése és a funkcionális programozási nyelvek fordítási technikájának fejlődése együttesen azt eredményezte, hogy a funkcionális nyelven írt programok hatékonysági szempontból is lényegében egyenértékűek az imperatív nyelven fejlesztett programokkal. Például a Clean nyelv fordítója által előállított kód futási ideje alig haladja meg a C nyelven készített programváltozatét.

[1] Ez a fejezet részben az OTKA T037742. számú pályázat támogatásával és keretében készült.

A funkcionális nyelvek gyakorlati jelentősége egyre nagyobb, az objektumorientált megközelítést jól kiegészítő jellegük miatt is. Számos szoftverfejlesztő cég használ általános célú vagy saját célra alkotott funkcionális nyelvet egyes különösen összetett részfeladatok megoldására. Ilyen alkalmazási területek például a távközlés (Erlang, Ericsson), vagy a természetes nyelvű szövegek elemzése (Haskell, Lolita), a forgalomirányító rendszerek (Clean), a felhasználói felületek létrehozása.

Az ugyanannak a feladatnak a megoldására funkcionális nyelven írt programkód általában lényegesen rövidebb, kifejezőbb, olvashatóbb és könnyebben módosítható, mint az imperatív nyelven kódolt programszöveg. Ennek az az oka, hogy a tisztán funkcionális nyelvekben az imperatív nyelvekből ismert változófogalom nem létezik, így a kódrészletek mellékhatásokat sem okozhatnak. Egy programrészlet megváltoztatásának hatása sokkal kisebb körben érvényesül, így könnyebben követhető. Funkcionális nyelvek alkalmazásával a nagy programok bonyolultsága lényegesen csökkenthető, a fejlesztéshez szükséges idő lerövidül, a program megbízhatóbb, kevesebb hibát tartalmaz.

A funkcionális programozási nyelvek elterjedését jelenleg leginkább a megfelelően képzett és jó absztrakciós készségekkel rendelkező fejlesztők hiánya akadályozza.

16.1.1. A funkcionális programozási stílus

Funkcionális szempontból nem a program működésének egyes lépései fontosak, hanem a programnak az a hatása, amely a program programfüggvényével (hatásrelációjával) [Fót 83] vagy viselkedési relációjával jellemezhető [Hor 94]. Ha programok helyességét vizsgáljuk, akkor a programfüggvényt, a viselkedési relációt vetjük össze a feladattal. A megoldó program egy összetevője mellékhatást okozhat, ha a feladat felbontása, finomítása során nem megfelelő módon választjuk meg az egyes részfeladatokat[2]. A mellékhatások azt eredményezik, hogy a program helyessége csak nagyon körülményesen igazolható, a programkonstrukciókra vonatkozó levezetési szabályok csak nehezen (azaz a mellékhatások figyelembevételével), vagy egyáltalán nem alkalmazhatók. Az utóbbi esetben lehetetlenné válik az elő- és utófeltételekre támaszkodó szintézis, illetve helyességbizonyítási módszer [Tho 90, Hor 99a, Mol 01, Hor 02b]. Tág értelemben *funkcionális programozási stílusról* beszélünk, ha a megoldó programokat mellékhatásoktól mentes összetevőkből szabályos programkonstrukciókkal építjük fel.

A feladat leírásához használt elő- és utófeltételek a változók[3] értékeire vonatkozó kikötéseket tartalmaznak. Ezen kikötések gyakran függvénykompozíciók értékére vonatkoznak egyenlőségek formájában. A függvénykompozíció alakja sok esetben megfelel a megoldó program felépítésének. A megoldást tehát kereshetjük oly módon is, hogy a feltételeket szabályos függvénykompozíciókkal építjük fel elemi függvényekből. Ezzel egyben közvetett módon azt is meghatározzuk, hogy milyen módon juthatunk el a feladat előfeltételével jellemzett állapotokból az utófeltétellel leírt állapotok egyikébe. Másképpen fogalmazva meghatározzuk az utófeltételben szereplő függvények kiszámítási módját a rendelkezésre álló

[2] Mellékhatás akkor léphet fel, ha nem bővített identitás vagy nem vetítéstartó a feladatot leíró reláció a részfeladat altere és kiegészítő altere felett [Fót 83].

[3] A változókat az állapottér projekciós függvényeinek tekintjük.

elemi függvények segítségével. Ha a függvénykompozíciókban *magasabbrendű függvényeket* is használhatunk, azaz olyan függvényeket, amelyeknek az argumentumai és az értéke függvények is lehetnek, akkor egy egységes funkcionális szemlélethez jutunk[4] [Kis 93].

16.1.2. Funkcionális program felépítése és kiértékelése

A *funkcionális program* típus-, osztály- és függvénydeklarációk, illetve definíciók sorozatából, valamint egy *kezdeti kifejezésből* áll. A program végrehajtása nem más, mint a kezdeti kifejezés kiértékelése. A kiértékelést úgy képzelhetjük el, mint a kezdeti kifejezés átírása a benne szereplő függvények szövegszerű behelyettesítésével (16.1.2. példa). Az átírási lépések pontos jelentését a nyelv modellje határozza meg. A funkcionális nyelven írt programok végrehajtási modellje minden esetben egy konfluens redukciós rendszer (átíró rendszer). Ha egy átíró rendszer *konfluens*, akkor az egyes részkifejezések átírásának sorrendje a végeredményre nincs hatással, a sorrend legfeljebb azt befolyásolja, hogy az átírási lépéssorozattal eljutunk-e a végeredményig. Konfluens rendszer a λ-kalkulus, illetve léteznek konfluens kifejezésátíró, és gráfátíró rendszerek is[5]. Több nyelvben, például az SML-ben, a Haskellben és a Cleanben megadhatók a λ-kalkulus nyelvtani szabályainak közvetlenül megfelelő kifejezések is [Bar 84].

Egyszerű *függvénydefiníciók* megadásakor a függvény neve és az *argumentumot* azonosító *formális paraméter* áll a definiáló egyenlőség bal oldalán, a függvényértéket meghatározó kifejezés, a *függvény törzse* pedig az egyenlőségjel jobb oldalán. A függvény definíciója egy rögzített funkcionális nyelv és kiértékelési módszer mellett egyben meghatározza a függvényérték kiszámításának módját és költségét is adott argumentumértékre. Megjegyezzük, hogy egyes, matematikai értelemben jól definiált függvények funkcionális programozási nyelven hatékonysági vagy kiszámíthatósági szempontok miatt nem adhatóak meg matematikai definíciójukkal azonos alakban.

A továbbiakban megadjuk néhány egyszerű függvény definícióját[6].

(Miranda, Clean és Haskell): Egyszerű függvénydefiníciók

```
nulla x = 0
id x = x
inc x = x + 1
square x = x * x
squareinc x = square (inc x)
fakt n = product [1..n]
```

[4] Hudak [Hud 89] nyomán belátható, hogy a funkcionális szemléletben valóban megfogalmazható az imperatív programozásban megszokott *elemi programok* [Fót 83] (SKIP, ABORT, értékadás), illetve programkonstrukciók (szekvencia, elágazás, ciklus) mindegyike. Az értékadás például mint a változókon értelmezett *magasabbrendű függvény* definiálható.

[5] Miranda programok λ-kalkulusra fordítására mutat példát [Pla 93]. A Haskell programok jelentését két lépésben határozzák meg. Megadják a nyelv úgynevezett magjának szemantikáját λ-kalkulusban, majd a további nyelvi elemek jelentését a nyelv magjára támaszkodva [Pey 99]. A Clean nyelv szemantikáját egy gráfátíró rendszer határozza meg. Hatékony kiértékelési módszereket: kifejezésátíró rendszereket (TRS), gráfátíró rendszereket (GRS) ismertet [Pla 93].

[6] A fejezet példaprogramjai előtt zárójelben mindig feltüntetjük, hogy az adott példa forrásszövegét melyik funkcionális nyelv (vagy nyelvek) fordítója fogadja el. SML: Moscow ML 2.0; Clean: Concurrent Clean 1.3, 2.1; Haskell: GHCi 5.02.2 for Haskell 98.

A nulla függvény nem függ az `x` paramétertől, konstans függvény, értéke `0`. Az `id` identitásfüggvény a paraméterének értékét adja vissza. Az `inc` függvény (egész típusú) argumentumánál eggyel nagyobb értéket ad eredményül. A `square` függvény az argumentumának négyzetét határozza meg, a `squareinc` pedig az `inc` és `square` függvények kompozíciója. A `fakt` a faktoriális függvény.

(SML): Egyszerű függvénydefiníciók

```
fun square (x) = x * x;
val area = fn  r => pi * r * r
fun plusz a b  =  a + b
```

Az `area` függvény definíciójában az argumentumokat az `fn` kulcsszó[7] és a `=>` jel között helyeztük el. A `fun` kulcsszó alkalmazásával a függvénydefiníció egyszerűbb alakban is felírható, erre példa a `square` és `plusz` függvények definíciója. Az `area` függvény az `r` sugarú kör területét, a `plusz` függvény az argumentumainak összegét határozza meg.

A funkcionális program végrehajtása a kezdeti kifejezésből kiinduló *redukciós* vagy más néven *átírási* lépések sorozata. Egy-egy redukciós lépésben egy – valamely *kiértékelési stratégia* szerint kiválasztott – redukálható részkifejezésben, a *redexben* szereplő függvényhívás helyettesítődik a függvény *törzsében* megadott kifejezéssel a formális és aktuális paraméterek megfeleltetése mellett. *Normál formájú* egy kifejezés, ha további redukcióra nincs lehetőség, ez az átírási lépéssorozat végeredménye.

Megadunk néhány kezdeti kifejezést:

(Miranda, Clean és Haskell): Kezdeti kifejezések

```
Start = sqrt 5.0        // Clean
main = product [1..10] -- Haskell
squareinc 7
```

A normál formák: `2.236068`, `3628800`, `64`.

(SML): Kezdeti kifejezések

```
Math.sqrt 5.0;
```

A normál forma a `2.2360679775` valós érték[8].

Egy-egy kifejezésben gyakran egynél több redex is van, és a kiértékelési módszer határozza meg, milyen sorrendben kerül sor ezek átírására. Bizonyos esetekben egyidőben vagy időben átfedve, egymással párhuzamosan több részkifejezés is redukálható, meggyorsítva ezzel a program végrehajtását.

Konfluens rendszerekben a kifejezések normál formája nem függ a kiértékelési sorrendtől, azaz a normál forma egyértelmű [Bar 84], de nem minden kifejezésnek van normál formája. A kiértékelési módszertől is függ, hogy a normál formához eljutunk-e. A Miranda, a Clean, illetve a Haskell által is alkalmazott *lusta kiértékelés* a kifejezések kiértékelésekor

[7] Az `fn` kulcsszó a λ-kalkulus lambda operátorának felel meg.

[8] Az SML értelmezőben csak a `Math` modul betöltése után (`load "Math"`) alkalmazhatjuk a négyzetgyökvonást az `5.0` valós értékre. A sor végére `;-t` kell tenni, ez a *kiértékelő jel*, aminek hatására az értelmező megkezdi a kiértékelést.

először a kifejezéssel ekvivalens λ-kifejezésben a legbaloldalibb legkülső redexet[9] helyettesíti (azaz a függvénydefiníciót alkalmazza elsőként, ha a kifejezés függvényszimbólummal kezdődik) és az argumentumok kiértékelését csak szükség esetén végzi el. A lusta kiértékelés *normalizáló kiértékelési módszer* (Curry és Feys, 1958.), azaz mindig megtalálja a normál formát, ha az létezik. Az ML vagy a LISP által alkalmazott *mohó kiértékelés* a legbaloldalibb legbelső redexszel[10], azaz az argumentumok redukálásával kezdi. A mohó kiértékelés gyakran hatékonyabb, de nem mindig ér véget akkor sem, ha létezik normál forma. Ezért a mohó (mohó kiértékelést alkalmazó) nyelvek gyakran tartalmaznak olyan nyelvi elemeket, amelyek egyes kifejezések lusta kiértékelését írják elő (ilyen nyelv például az SML, ahol létezik a lusta lista), illetve a lusta nyelvek gyakran alkalmaznak eljárásokat arra vonatkozóan, hogy szabad-e egy kifejezést hatékonyan, mohó stratégiával kiértékelni (ilyen például a Clean, amelynek fordítója mohóságvizsgálatot végez, illetve a programozó mohóságjelölést alkalmazhat.

Az alábbi két példán bemutatjuk a két legjellemzőbb kiértékelési módszert, a mohót és a lustát. Az egyes átírási lépésekben a függvényeknek az 591. oldalon megadott definícióit helyettesítjük be.

Redukció - mohó

```
squareinc 7
-> square (inc 7)
-> square (7 + 1)
-> square 8
-> 8 * 8
-> 64
```

Redukció - lusta

```
squareinc 7
-> square (inc 7)
-> (inc 7) * (inc 7)
-> (7 + 1) * (7 + 1)
-> 8 * 8
-> 64
```

16.1.3. A modern funkcionális nyelvek főbb jellemzői

Tisztán funkcionális egy programozási nyelv, ha nyelvi elemei felhasználásánál mellékhatások biztosan nem lépnek fel, illetve az előző értéket megsemmisítő (destructive) értékadás vagy más imperatív programozásra jellemző nyelvi elem nem áll rendelkezésre.

Ellentétben például a LISP-pel és az SML-lel, a Hope, a Miranda, a Haskell, a Concurrent Clean tisztán funkcionális nyelvek.

Az alábbiakban bemutatjuk a modern, tisztán funkcionális programozási nyelvek legfontosabb jellemzőit.

Hivatkozási helyfüggetlenség (referential transparency) • A kifejezések értéke hivatkozásuk előfordulási helyétől független, azaz ugyanaz a kifejezés a program szövegében mindenhol ugyanazt az értéket jelöli. A függvények kiértékelésének nincs mellékhatása, azaz egy függvény kiértékelése nem változtathatja meg egy kifejezés értékét. A tisztán funkcionális program *változói*[11] valójában állandók. A változók értéke – akárcsak a matematikában – esetleg még nem ismert, de egyértelmű, semmiképpen sem változhat meg a program

[9] A legbaloldalibb redex olyan redex, amelyik nincs egy másik redex belsejében (tehát legkülső), és az ilyen redexek között balról a legelső.

[10] A legbelső redex olyan redex, amelyik belsejében nincs másik redex (tehát legbelső), és az ilyen redexek között balról a legelső.

[11] Például a függvénydefiníciókban használt formális paraméterek.

végrehajtása során. Ezen a tulajdonságon alapszik a program funkcionális helyességét igazoló úgynevezett *egyenlőségi érvelés* (equational reasoning) alkalmazhatósága, azaz például az alábbi programrészletben

```
... x + x ...
      where  x = f a
```

az `x` minden *szabad előfordulása* helyébe szövegszerűen behelyettesíthető az `f a` kifejezés a `where` hatáskörén belül, és `f a` értéke minden esetben azonos.

Szigorú, statikus típusosság • Típusdeklarációk megadása nem kötelező, de megköveteljük, hogy a *Hindley–Milner-féle korlátozottan polimorfikus típusrendszer* alapján a kifejezések típusa a *típuslevezetési szabályok* által meghatározott legyen. Ez azt jelenti, hogy egy adott kifejezés *legáltalánosabb típusát* a fordítóprogram általában még akkor is meg tudja határozni a benne szereplő részkifejezések típusa alapján, ha a programszöveg készítője nem deklarálta a kifejezés típusát. Biztosítottak a nyelvi eszközök absztrakt és algebrai adattípusok leírásához is.

Magasabbrendű függvények (higher order functions) • A függvények ugyanolyan értékek, mint az elemi típusértékhalmazok elemei. Magasabbrendű függvénynek nevezzük azokat a függvényeket, amelyeknek valamelyik argumentuma vagy értéke maga is függvény. Magasabbrendű függvények segítségével biztosítható a program *modularitása*, illetve a *funkcionális absztrakció* elmélyítése. Például: `twice f x = f ( f x )`

A magasabbrendű függvények alkalmazása hatással van a számítási folyamatra is. Mohó kiértékelés esetén például egy függvény kiértékelésére előbb kerülhet sor, ha egy magasabbrendű függvény argumentumaként jelenik meg. Magasabbrendű függvények használatával a számítási folyamatot jól meghatározott szakaszokra bonthatjuk.

Curry módszer, részleges függvényalkalmazás • Ha függvények eredménye függvény is lehet, akkor nincs szükség a *többváltozós függvény* fogalmára. Haskell B. Curry módszere alapján minden függvényt tekinthetünk egyváltozósnak. Ha egy függvénynek több változója van, akkor Curry módszere szerint csak az első változót tekintjük a függvényhez tartozónak. Az első argumentum megadásával már egy újabb függvényhez jutunk, amelyet a következő argumentumra alkalmazhatunk. Többváltozós függvény alkalmazását tekinthetjük egyváltozós függvények alkalmazása sorozatának. Ebben a szemléletben például az összeadás műveletét is egyváltozósnak tekintjük oly módon, hogy az összeadás első argumentumát tekintjük a + művelet egyetlen argumentumának, értéke pedig egy újabb függvény. A `(+) 1` függvény például megegyezik az `inc` függvénnyel. Többváltozós függvény *részleges alkalmazásáról* beszélünk, ha – balról jobbra haladva – argumentumainak egy részének rögzítésével egy függvényt kapunk eredményül.

Függvények alkalmazása önmagukra • Lehetőség van *rekurzív és kölcsönösen rekurzív* függvénydefiníciók megadására.

Lusta (lazy) kiértékelés – mohó (eager, strict) kiértékelés • A modern funkcionális programozási szemléletben a kifejezések jelentését a lusta kiértékelési módszer határozza meg. A függvények argumentumai akkor értékelődnek ki, ha ténylegesen szükség van az értékükre a normál forma eléréséhez. Lusta redukciós módszert alkalmaz a Miranda, a Clean

és a Haskell is. Mohó kiértékelésről beszélünk, ha az argumentumot mindig kiértékelik, mielőtt a függvény alkalmazásának megfelelő redukcióra sor kerülne. Ilyen kiértékelési módszert alkalmaz például a LISP, az SML és a Hope.

Zermelo-Fraenkel halmazkifejezések • Iteratív adatszerkezet elemeinek és azok sorrendjének megadására alkalmas, a matematikában halmazok megadásánál alkalmazott jelölésrendszernek megfelelő nyelvi eszköz a Zermelo-Fraenkel halmazkifejezés. A végtelen adatszerkezetek (listák, halmazok, sorozatok, vektorok) kiértékelése lusta kiértékelési módszerrel történik. A

```
[ x * x | x <- [1..], odd(x) ]
```

kifejezés egy olyan végtelen listát határoz meg, amelyben rendre szerepelnek a páratlan (`odd`) természetes számok (`[1,3,5 ..]`) négyzetei. A teljes program futásának befejeződése az eredményt mint argumentumot felhasználó függvény igényeitől függ (modularitás), a listát pedig lustán, azaz csak addig értékeljük majd ki, ameddig szükség van a benne szereplő elemekre.

Argumentumok mintaillesztése • A függvények definiálásakor megadhatunk mintákat[12] (lásd a 16.2.3. részt) a formális paraméter helyén. Amennyiben az aktuális paraméter egy megadott mintára illeszkedik, a függvény értékét az ehhez a mintához tartozó függvénytörzsváltozat határozza meg (pontos szabályt lásd a 16.2.3. részben). Például:

```
fac 0      = 1
fac n|n>0  = n * fac (n-1)
```

A `fac` függvény definíciójában az első minta a `0` érték. Ha az aktuális paraméter illeszkedik a `0` mintára, azaz nullával egyenlő, akkor a függvény értéke `1`.

Margó szabály (off-side rule) (Landin) • Összetartozó kifejezések csoportjának azonosítására és deklarációk *hatáskörének* korlátozására alkalmas a bal oldali margó szélességének változtatása. A margó szabály a blokkszerkezet kialakításának egyik nyelvi eszköze. Egy deklaráció hatásköre lokális egy, a forrásszövegben őt megelőzőre nézve, ha a deklaráció úgy követi az előzőt, hogy beljebb kezdődik. Minden nyelv leírása megadja a margó szabály alkalmazásának pontos definícióját[13]. A hatáskörök egymásba is ágyazhatóak, oly módon, hogy egyre inkább jobbra haladva kezdjük írni a deklarációkat. Az alábbi példában az `add4` függvény egy lokális `succ` függvényt alkalmaz, amely különbözik az `add` függvényben használt `succ` függvénytől:

```
add4 = twice succ
     where
       succ x = x + 2
add  = ... succ
```

[12] Kevés kivételtől eltekintve mintában csak a formális paraméter típusának algebrai típusdefiníciójában megadott konstruktorokat használhatjuk, lásd a 16.4.2. részt.

[13] Ha az egyes megjelenített karakterek szélessége nem állandó, nagyon nehéz megállapítani a deklarációk hatáskörét. A forrásszövegen végrehajtott szövegszerkesztési lépések következtében is megváltozhat egyes deklarációk hatásköre például akkor, ha egyes azonosítókat hosszabbakra cserélünk.

I/O modell • A modern funkcionális nyelvek rendelkeznek valamilyen I/O modellel is. Ilyen modell például az *IO monád*, az *egyszeresen hivatkozott, egyedi környezet* (unique environment), az *adatáramlási IO modell* (stream of request, stream of response), vagy a vele egyenértékű *folytatás* (continuation) modell (lásd a 16.7. részt).

16.1.4. A funkcionális nyelvek rövid áttekintése

A LISP-et (LISt Processing), az első olyan nyelvet, amelynek λ-kalkulus volt a számítási modellje, John McCarthy alkotta meg 1956–1962 között az MIT-n. Azóta számos LISP-változat jött létre, például a Common LISP (DARPA, 1981), amely procedurális és objektum-elvű elemeket (CLOS, Common Lisp Object System) is magában foglal, vagy a Scheme nyelv (Steele, Sussman, 1975), amelyet inkább az egyetemeken használnak és CAD rendszerekben alkalmaznak. Az első típusos funkcionális nyelv az ML (Meta Language), amely eredetileg az Edinburghban tételbizonyításra tervezett LCF (Logic for Computable Functions) rendszer metanyelve volt. A nyelvet R. Milner tervezte a 70-es évek közepén. A Hope (Burstall, 1980) megalkotása után alkotta Milner, Tofte és Harper az SML-t (Standard ML) 1983 és 1990 között. Az SML legújabb, átdolgozott szabványa 1997-ben készült el [Mil 97, Har 01, Han 00]. ML-változat a Caml (INRIA, 1984–1990, a Coq tételbizonyító alapnyelve) és az Objective Caml (a Caml Light továbbfejlesztése, INRIA, 1990–). Az ML-változatok nem tisztán funkcionális nyelvek, imperatív nyelvi elemeket (például frissíthető változókat) is tartalmaznak. Az ISWIM (If You See What I Mean, Landin, 1966) volt az első lusta kiértékelést alkalmazó tisztán funkcionális nyelv, amely a margó szabályt is bevezette. D. Turner az ISWIM-ből kiindulva több nyelvet is tervezett, az ugyancsak lusta kiértékelésű SASL-t (Single Assignment Language, Turner, 1976), a KRC-t (Kent Recursive Calculator, Turner, 1981), majd 1985–86-ban a Mirandát [Mir 90, Tur 86, Kis 93, Cla 95]. A Miranda üzleti termék, a Research Software Ltd. tulajdona. A Miranda elemeit felhasználó nyelv a Haskell (1990) [Pey 99, Hud 99, Tho 99].

A Haskell megalkotásáról 1987-ben döntöttek egy funkcionális programozási konferencián (FPCA '87). A nyelvet Haskell Brooks Curry matematikusról nevezték el. Legújabb definíciója a Haskell 98 szabvány. A tervező csoport tagjai között a világ számos egyetemétől, illetve kutatóintézetéből találunk kutatókat (J. Hughes, S. Peyton Jones, P. Hudak, K. Hammond, E. Meijer, J. Peterson, P. Wadler és mások). A Haskell esetében alapkövetelményként fogalmazták meg, hogy a nyelv

- egyaránt alkalmas legyen oktatási és kutatási feladatok megoldására, illetve nagy alkalmazói programok fejlesztésére,
- legyen formálisan definiált szintaxisa és a szemantikája,
- szabadon elérhető legyen,
- feleljen meg széles körben elfogadott alapelveknek,
- egységes fejlődési irányt szabjon az egymástól kisebb-nagyobb mértékben különböző modern funkcionális nyelveknek.

A Concurrent Clean (Plasmeijer, Nijmegen, 1987) [Pla 01, Pla 99] egy gráfátíró kísérleti nyelvből (a LEAN-ból) fejlődött ki, tisztán funkcionális, alapvetően lusta kiértékelést alkalmazó nyelv. Jelenlegi változata (a Clean 2.0) közel áll a Haskellhez [Heg 02], de több olyan nyelvi elemet is tartalmaz, amelyet a Haskell 98 nem.

A függvénykompozíció asszociatív művelet, ezért eredményének kiszámítása jól párhuzamosítható. A kifejezések normál formája, ha létezik, független a kiértékelési módszertől (legalábbis a konfluens átíró rendszerekben), így a funkcionális nyelven írt programok jól párhuzamosíthatók. A legtöbb nyelvnek létezik olyan változata (Concurrent Clean [Kes 96], GpH - Glasgow Parallel Haskell [Tri 98], Distributed Haskell [Huc 01], Eden, Concurrent ML, JoCaml [Fou 01] stb.), amely nyelvi elemekkel is támogatja annak meghatározását, hogy melyik részkifejezést célszerű (spekulatív módon) párhuzamosan vagy elosztott módon kiszámítani.

16.2. Egyszerű funkcionális programok

16.2.1. Egyszerű függvények definíciója

A függvények definíciója egy vagy több olyan egyenlőségből áll, amelyek bal oldalán a függvény neve és *formális paraméterei*, jobb oldalán pedig a függvény értékét és kiszámítási módját is meghatározó kifejezés, a *függvénytörzs* áll. A *kifejezés* állhat egyetlen értékből, a formális paraméterből vagy egy *függvény alkalmazásából* annak aktuális paraméterére (amely lehet a definiált függvény formális paramétere is). A kifejezések kiértékelése során a legmagasabb *prioritása* a függvényalkalmazásnak van. Megadhatunk az argumentumokra vonatkozó *feltételeket* és *mintákat* is, amelyek az aktuális és a formális paraméterek *illesztését* határozzák meg. A függvény értelmezési tartománya és értékkészlete is a deklaráció része, ez azonban egyszerűbb esetekben elhagyható.

A függvények definiálásakor a formális paraméterek leírására változóneveket vezetünk be. A változó deklarációjának *hatásköre* ez esetben a változót bevezető egyenlőségre terjed ki. A deklarált függvénynevek hatásköre alapesetben a modulra, egyes esetekben a teljes programra terjed ki. A deklarációk hatásköre szűkíthető a *margó szabály*, illetve lokális deklarációk megadását jelző kulcsszavak (`where`, `local`, `let`, # stb.), valamint a modulok közötti export-import mechanizmus alkalmazásával.

Rekurzív függvénydefiníciót is megadhatunk. Hatékonysági szempontból előnyös lehet, ha a rekurzív hivatkozás egyszeres és a definíció végén helyezkedik el. *Esetszétválasztással* meg kell adnunk az alapesetet, különben a kiértékelés nem ér véget.

Feltételes kifejezéssel megadott rekurzív függvény

```
fakt n  = if n==0 then 1 else n * fakt (n-1)   -- Haskell
fakt n  = cond (n=0) 1 (n*fakt(n-1))           || Miranda
fakt n  = if (n==0) 1  (n * fakt (n-1))        // Clean
fun fakt n = if n=0 then 1 else n * fakt (n-1) (*SML*)
```

16.2.2. Feltételek megadása

Az esetszétválasztást a matematikában megszokott módon is megadhatjuk az argumentumokra vonatkozó *feltételekkel*. A feltételeket a függvény alkalmazásakor az aktuális paraméter értéke alapján, megadásuk sorrendjében vizsgálja meg a kiértékelő rendszer. Az első igaz feltételhez tartozó egyenlőséget alkalmazza, ha pedig nincs ilyen[14], akkor valamilyen hibaüzenettel leáll a kiértékelés vagy egy kivételkezelő kiértékelésére kerül sor (lásd a 16.8. részt).

[14] Ilyen esetben egy parciális függvényt definiáltunk.

A feltételek megadásának sorrendje szemantikai szempontból lényeges. Ha a sorrendet megváltoztatjuk, általában a definíció jelentése is megváltozik.

(Clean és Haskell): Esetszétválasztás, rekurzív függvény

```
fakt n | n==0 = 1
       | n>0  = n * fakt (n-1)
```

(Miranda): Esetszétválasztás, rekurzív függvény

```
lnko a b = lnko (a-b) b, if a>b
         = lnko a (b-a), if a<b
         = a,           otherwise
```

16.2.3. Mintaillesztés

Mintaillesztés alkalmazásával is szétválaszthatunk eseteket. Mintaillesztés esetén azt vizsgálja a kiértékelő rendszer, hogy az aktuális paraméter értéke vagy alakja megfelel-e valamelyik definiáló egyenlőség bal oldalán megadott mintának. A minták és a hozzájuk tartozó egyenlőségek sorrendje meghatározza a mintaillesztés eredményét. Ha a sorrendet megváltoztatjuk, általában a definíció jelentése is megváltozik. A kiértékelő rendszer az első illeszkedő mintához tartozó egyenlőséget alkalmazza, ha pedig nincs ilyen, hibaüzenettel leáll[15]. Minták megadása helyett mindig megadhatunk ekvivalens *feltételsorozatot*, de ez fordítva nem igaz. A mintaillesztés, mint nyelvi elem nem szükséges ahhoz, hogy teljes funkcionális nyelvet alkossunk, de nagymértékben egyszerűsíti a függvénydefiníciók megadását és javítja a kód olvashatóságát. Módosíthatósági szempontból hátrányos, hogy a sorrendben előbb megadott minta módosítása kihat a később megadott minták jelentésére.

(Miranda, Clean és Haskell): Mintaillesztés

```
fakt 0  = 1
fakt n  = n * fakt (n-1)
```

Mintaillesztés és feltételek egyszerre is alkalmazhatók:

(Clean és Haskell): Mintaillesztés és feltételek

```
fakt 0         = 1
fakt n | n > 0 = n * fakt (n-1)
```

Mintaillesztésre az SML is biztosít nyelvi elemet:

(SML): Mintaillesztés

```
fun fakt 0  = 1
      | n   = n * fakt (n-1)
```

[15] A mintaillesztések kiértékelésének módja lényegesen egyszerűbb és hatékonyabb a logikai programozási nyelvekben alkalmazottnál, mert a mintában – néhány kivételtől eltekintve – csak a formális paraméter típusának algebrai típusdefiníciójában megadott konstruktorokat használhatjuk, lásd a 16.4.2. részt

16.3. Függvények típusa, magasabbrendű függvények

A függvények típusát értelmezési tartományuk és értékkészletük megadásával határozzuk meg. Az értelmezési tartomány és az értékkészlet közé -> jelet teszünk, illetve az SML-ben az értelmezési tartományt és változóját zárójelek közé is tehetjük, az értékkészletet pedig a zárójel után kettősponttal elválasztva adhatjuk meg.

Függvény típusa

```
ketszer :: num->num          || Miranda
ketszer :: Num a => a -> a   -- Haskell
fakt :: !Int -> Int          // Clean
val fakt : int -> int        (* SML *)
fun fakt (x:int):int         (* SML *)
```

A Mirandában a numerikus típusok közös azonosítója a `num`. A Haskell nyelv megkülönbözteti az egyes numerikus típusokat egymástól, de lehetőséget ad arra, hogy a numerikus típusok `Num` *osztályának* minden a elemtípusára egyszerre definiáljunk egy függvényt. A típusosztályokat a későbbiekben részletesen tárgyaljuk.

A funkcionális nyelvekben és a λ-kalkulusban szigorú matematikai értelemben minden függvénynek legfeljebb egy argumentuma van. *Többváltozós függvények* megadása *magasabbrendű függvények* segítségével lehetséges, a Haskell B. Curryről elnevezett módszer alapján (Schönfinkel, Curry, Feys). *Elsőrendű* az a függvény, amelynek sem argumentuma, sem értéke nem függvény.

Többváltozós függvény típusa

```
plusz :: Num a => a -> a -> a -- Haskell
plusz :: num -> num -> num    || Miranda
plusz :: Int Int -> Int       // Clean
plusz a b = a + b
val plusz : int -> int -> int = (* SML *)
     fn a => fn b =>  a + b     (* SML *)
```

A `plusz` függvény egyetlen argumentuma `a`, eredménye pedig az a függvény, amely az `a`-nak megfeleltetett aktuális paraméterhez adja hozzá azt a számértéket, amelyet az új függvény egyetlen formális paramétere, a `b` azonosít. `plusz :: Int -> Int -> Int` jelentése tehát azonos `plusz :: Int -> (Int -> Int)`-tel. A `plusz 5` egy szabályos függvényalkalmazás, amelynek eredménye egy `Int -> Int` típusú függvény. `plusz 5 6` egyenlő `(plusz 5) 6`-tal. A többváltozós függvények típusdefiníciói tehát jobbról balra *asszociatívak*, míg alkalmazásuk balról jobbra asszociatív: `f :: a -> b -> c` egyenértékű `f :: a -> (b -> c)` -vel, azaz `f` egy olyan függvény, amelynek argumentuma `a` típusú, eredménye pedig egy olyan függvény, amelynek értelmezési tartománya `b`, értékkészlete `c`. Az `f x y` függvényalkalmazás ennek megfelelően egyenértékű `(f x) y`-nal.

A Curry módszer alkalmazásával többváltozós függvényekből könnyen készíthetünk olyan példányokat, amelyekben néhány paraméter értékét rögzítjük (részleges alkalmazás). Ilyen függvény az `inc`.

(Miranda, Clean és Haskell): Részlegesen alkalmazott függvény

```
inc :: num -> num                || Miranda
inc :: Num a => a -> a           -- Haskell
inc :: Int -> Int                // Clean
inc x = (+) 1 x                  // Miranda, Clean, Haskell
inc = (+) 1                      // Clean, Haskell
fun inc (x:int):int = plusz 1 x (* SML *)
val inc : int->int = secl 1 op+ (* SML *)
```

Az összeadás műveleti jelét (+) általában infix helyzetben alkalmazzuk, azaz a két összeadandó között helyezzük el. A `plusz` függvény definíciója – a függvényekre általánosan érvényes – prefix alkalmazáshoz készült. A legtöbb nyelvben azonban a bináris műveleteknél lehetőség van arra, hogy infix operátort prefix helyzetben használjunk és viszont. Például: `(+) 2 3` (Clean, Haskell, Miranda), `2 'plusz' 3` (Haskell), `2 $plusz 3` (Miranda), `op+ (2,3)` (SML). Az `inc` függvény `(+) 1`, illetve `secl 1 op+` alakú[16] definíciói éppen ezt a lehetőséget használják fel, hogy az infix bináris + művelet első argumentumát az összeadás művelet prefix változatát alkalmazva rögzítik.

Megjegyezzük, hogy egy függvény alkalmazásának nem csak az értéke lehet függvény, hanem megfelelő módon definiált függvény argumentuma is lehet függvény. Ezt mutatjuk be a következő példán, és a 602. oldal hasonló nevű példáján.

(Miranda, Clean és Haskell): Függvény, mint argumentum

```
nullaban :: Num a => (a -> a) -> a   -- Haskell
nullaban :: (num -> num) -> num      || Miranda
nullaban :: (Int -> Int) -> Int      // Clean
nullaban f  = f 0
fun nullaban (f: int -> int) : int
            = f 0                    (* SML *)
nullaban inc
```

A fenti példában megadott függvény típusdefiníciójában szereplő zárójelek nem hagyhatók el. A zárójelek elhagyása maga után vonná az asszociativitási szabály alkalmazását és teljesen megváltoztatná a definíció értelmét. A zárójelek megadásával jelölnünk kell, hogy a függvény paramétere nem egy számérték, hanem függvény.

16.3.1. Egyszerű típuskonstrukciók

A funkcionális nyelvekben is rendelkezésünkre áll a direkt szorzat, illetve iterált *típuskonstrukció*. Rendezett n-eseket, illetve véges vagy végtelen a *sorozatokat*[17] képezhetünk. A rendezett n-eseket gömbölyű `()`, a sorozatokat pedig szögletes `[]` zárójelek közé zárjuk.

[16] `fun secl x f y = f(x,y)` az `f` bal oldali argumentumának lekötésére használható SML függvény.

[17] A funkcionális nyelveket ismertető könyvekben a sorozatok helyett *listákról* beszélnek, utalva a megvalósítás módjára. Egyes funkcionális nyelvekben - amelyek nem statikusan típusosak - lehetséges, hogy egy lista elemei különböző típusúak legyenek. Ezt sem a Miranda, sem a Clean, sem a Haskell, sem az SML nem engedi meg.

Rendezett n-esek

A rendezett n-esek összetevőit mintaillesztéssel, vagy előre definiált szelektorfüggvényekkel (`fst`, `snd`, `#i`, stb.) érhetjük el.

(Clean és Haskell): Rendezett pár legnagyobb közös osztója

```
lnko :: (a,a) -> a | - , < , == a    // Clean
lnko (a,b) | a>b = lnko (a-b,b)
           | b>a = lnko (a,b-a)
           | a=b = a                 -- Haskell
           | a==b = a                // Clean
```

(SML): Rendezett pár legnagyobb közös osztója

```
lnko : int * int -> int
fun lnko (0,b) = b
  | lnko (a,b) = lnko(a mod b, b);
```

A rendezett n-esek között kitüntetett szerepe van a rendezett nullásnak, amelynek egyetlen összetevője sincs. A `()` érték az SML `unit` típusának egyetlen típusértéke.

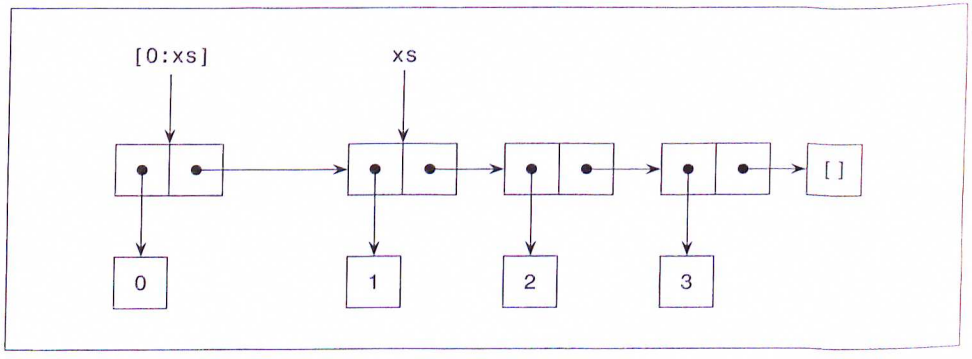
Sorozatok

A `:`, illetve az SML-ben a `::` *konstruktorok* alkalmasak arra, hogy egy elemből és egy sorozatból egy újabb, az eredeti sorozatot az adott elemmel balról kiterjesztő sorozatot készítsenek. A konstruktorokat mintákban is használhatjuk, így mintaillesztéssel könnyen eldönthető, hogy a sorozat üres-e (`[]`), valamint a nem üres (`[a:x]`) sorozat (`a`) első eleme, illetve a fejelem nélküli maradék (`x`) sorozat egyszerűen a minta megadásával meghatározható.

Sorozat elemeinek összege

```
fun osszeg (a::x) = a + osszeg x
  | osszeg []     = 0;                (* SML *)

osszeg [] = 0
osszeg (a:x) = a + osszeg x           -- Haskell
osszeg [a:x] = a + osszeg x           // Clean

osszeg [1,2,3,4,5,6,7,8,9,10]
osszeg [1..10]                        -- Haskell, Clean
```

A 16.1. ábra a sorozatok láncolt listaként történő ábrázolását mutatja be. Az ábrán megfigyelhetjük, hogy a lista elemeit külön tároljuk, a lista vázában csak az elemek címe található. Az alábbiakban megadjuk néhány olyan függvény definícióját Clean nyelven, amelyeket listák esetén gyakran használunk. `hd` az első, a `last` a sorozat utolsó elemét adja vissza. A `tl` a sorozat fejelem nélküli részsorozatát adja eredményül. Mivel a tisztán funkcionális nyelvekben a függvényeknek nincs mellékhatásuk, egy lista kezdőrészét (`init`) (az utolsó elem nélküli részét) csak úgy kaphatjuk meg, ha a régi lista vázát megőrizve a kezdőrészhez teljesen új vázat építünk. Az eredeti lista vázát nem módosíthatjuk, az utolsó



16.1. ábra: Sorozatok belső ábrázolása.

értékre mutató vázelemet nem szakíthatjuk le[18], mert azzal megsértenénk a hivatkozási helyfüggetlenséget.

```
hd [a:x]  = a                  tl [a:x] = x
hd [] = abort "hd of []"       tl []  = abort "tl of []"
last [a]  = a                  init [] = []
last [a:tl] = last tl          init [x]  = []
last [] = abort "last of []"   init [x:xs] = [x: init xs]
```

A `map` függvény segítségével elemenkénti feldolgozást adhatunk meg. A `map` argumentuma az egyes elemekre alkalmazott függvény. A `modseq` függvényt a `map` segítségével definiáljuk: egy számértékeket tartalmazó listából az argumentumként kapott bináris művelet és egy számérték segítségével elemenként határozza meg az új listát. Figyeljük meg, hogy a definícióban mindkét oldalról elhagyhattuk az utolsó paramétert, a listát.

Függvény, mint argumentum

```
modseq ::(num->num->num)->num->[num]->[num]          || Miranda
modseq :: Num a => (a->a->a)->a->[a]->[a]            -- Haskell
modseq :: (Int Int -> Int) Int->([Int]->[Int])       // Clean
modseq f c = map (f c)
modseq :(int->int->int)->int->int list -> int list;  (* SML *)
fun modseq f c = map (f c);                          (* SML *)

modseq plusz 5 [1,2,3]  // = [6,7,8]
modseq plusz 5 []       // = []
```

Sorozatokat *generálhatunk* is korábban definiált sorozatokból logikai kifejezések, *szűrők* megadásával.

(Miranda, Clean és Haskell): Szám osztói

```
osztok n = [i | i<-[1..n], n `mod` i == 0 ]     -- Haskell
osztok n = [i | i<- [ 1 .. n ]; n mod i = 0 ]   || Miranda
osztok n = [i \\ i<-[1..n] | n mod i == 0 ]     // Clean
```

[18] A sorozat belső ábrázolását megvalósító láncolt lista felépítéséhez használt mutatókhoz egyébként sem férhetünk hozzá.

Egy n szám osztóinak sorozatát úgy kapjuk meg, ha 1-től n-ig előállítjuk az összes i értéket az i<-[1..n] generátorral, majd az n mod i == 0 szűrő alkalmazásával kiválogatjuk azokat, amelyekkel n-t osztva 0 maradékot kapunk. A .. szimbólummal (a *pont-pont-jelöléssel*) véges vagy végtelen számtani sorozatokat adhatunk meg, a számtani sorozat differenciáját pedig az első két elem különbségével adhatjuk meg. Ha nem adjuk meg a második elemet, akkor a differencia 1, ha nem adjuk meg az utolsó elemre vonatkozó felső korlátot, akkor a sorozat végtelen. Egynél több generátort is megadhatunk, amelyek vagy egymásba ágyazva, vagy párhuzamosan futva állítják elő az elemeket:

(Clean): Generátorok

```
[ (x,y) \\ x <- [1..4], y <- [1..x] | x+y > 3 ] // beágyazott
[ (x,y) \\ x <- [1..4] & y <- [6..8] ]          // párhuzamos
```

Rekordok

A rendezett *n*-esek mellett a Cleanben és az SML-ben névvel ellátott mezőkkel rendelkező rekordokat is használhatunk. Ha egy rendezett n-es nagyon sok összetevőből áll, akkor nagyon körülményes egyes elemeinek mintaillesztéssel történő elérése. Ha a direkt szorzat típuskonstrukció értékeinek elemeit névvel ellátott szelektorfüggvénnyel azonosíthatjuk, akkor a programszöveg áttekinthetőbb és módosíthatóbb lesz. Az SML-ben és a Cleanben nagyon hasonló a rekord típus, a rekord típusú érték és a rekordminta megadásának alakja, így most csak egy Clean nyelven írt példát mutatunk.

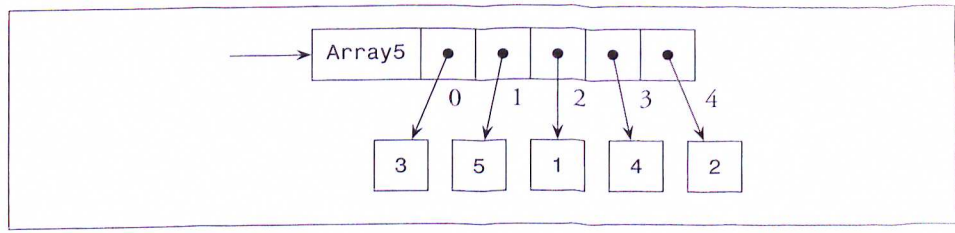
(Clean): Rekord

```
:: Point = {  x       ::  Real
           , y       ::  Real
           , visible ::  Bool
           }
:: Vector = { dx       ::  Real
           , dy       ::  Real
           }
Origo :: Point
Origo = { x = 0.0
        , y = 0.0
        , visible = True
        }
```

Mintaillesztés esetén elegendő, ha csak azokat a rekordmezőket adjuk meg a mintában, amelyek a függvény törzsében is szerepelnek.

(Clean): Mintaillesztés, rekordminta

```
IsVisible :: Point -> Bool
IsVisible {visible = True} = True
IsVisible _                = False
```



16.2. ábra: Dobozolt tömb

A szelektorfüggvények jelölésére a Cleanben alkalmazhatjuk a más nyelvekben megszokott '.' (pont) jelölést, míg az SML-ben a `#<mezőnév>` alakú függvényekkel határozhatjuk meg egy rekord valamely összetevőjének értékét.

```
xcoordinate :: Point -> Real
xcoordinate p = p.x
```

A Clean nyelvben lehetőségünk van arra is, hogy egy rekord típusú érték definíciójában egy másik rekordra hivatkozzunk. A `hide p` egy olyan rekord, amelynek minden mezője megegyezik a `p` rekord megfelelő mezőjének értékével, *kivéve* az `&` jel után felsorolt `visible` mezőt.

(Clean): A kivéve művelet

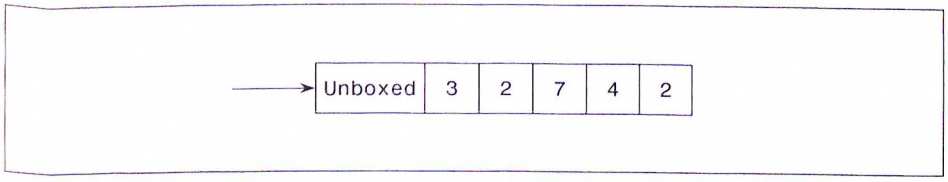
```
hide :: Point -> Point
hide p = { p & visible = False }

Move :: Point Vector -> Point
Move p v = { p & x = p.x + v.dx, y = p.y + v.dy }
```

A `Move` függvény *nem* tolja el a `p` pontot a síkon a `v` eltolásvektorral, hanem meghatároz egy olyan *új* pontot, amely a `p`-ből úgy származtatható, hogy minden mezőjének értéke ugyanaz, *kivéve* az `x` és `y` koordinátát.

Tömbök

Láncolt listák használatakor a sorozat elemeinek elérése csak úgy lehetséges, ha a listát az adott elemig bejárjuk. Tömböknél a kiértékelő rendszer címaritmetikát alkalmaz, az egyes elemeket azonos idő alatt érjük el. Az SML és a Clean nyelv – továbbá a Haskell egyes kiterjesztései is – tartalmazzák ezt a nyelvi elemet.

Az SML tömbök a nyelv imperatív részéhez tartoznak, az elemek értéke változhat. Cleanben csak az *egyszeresen hivatkozott* (unique, lásd a 16.6.1. részt) típusú tömbök elemei kaphatnak új értéket. Ilyenkor a hivatkozási helyfüggetlenség nem sérül. Az egyszeres hivatkozást a tömb típusának megadásánál `*`-gal jelezzük. A frissíthető tömböket a 16.6. részben mutatjuk be.



16.3. ábra: Dobozolatlan tömb

A Clean megkülönbözteti a *dobozolatlan* és a *dobozolt* tömböket, annak megfelelően, hogy a vázban tároljuk az elemeket vagy a listákhoz hasonlóan a váz csak az elem címét tartalmazza. A dobozolatlan tömbök típusdefiníciójában a nyitó kapcsos zárójel után egy # jel áll.

(Clean): Dobozolt tömb

```
Array5 :: *{Int}
Array5 = {3,5,1,4,2}
```

(Clean): Dobozolatlan tömb

```
Unboxed :: {#Int}
Unboxed = {3,2,7,4,2}
```

A tömbök elemeit az imperatív nyelvekben megszokotthoz hasonló módon választhatjuk ki. A tömb első elemét a `0` indexértékhez rendelték hozzá.

```
Array5.[1]+Unboxed.[0]
```

A Cleanben a tömböket generátorok segítségével is megadhatjuk. Az első példában az `[1,2,3]` lista elemeit a `<-` listagenerátor alkalmazásával sorra kiolvassuk a listából és elhelyezzük a `narray` tömbben. A második példában az `Array5` tömb elemeit olvassuk ki a `<-:` tömbgenerátor alkalmazásával és elhelyezzük az `nlist` listában.

```
narray = { e \\ e <- [1,2,3] }
nlist = [ e \\ e <-: Array5 ]
```

16.3.2. Lokális deklarációk

A Mirandában, Cleanben és a Haskellben a `where` kulcsszó használatával *lokális* függvényeket vezethetünk be. A `where` *hatáskörét* a *margó szabály* határozza meg. Az SML-ben a `local, in, end` szerkezetben helyezhetünk el egy deklarációhoz tartozó lokális deklarációkat, a `let, in, end` szerkezetben pedig egy kifejezésben szereplő lokális deklarációkat adhatunk meg. `let` kifejezéseket Haskellben és Cleanben is használhatunk. Cleanben a `let` kifejezéseknek több, formai szempontból is és jelentését tekintve is különböző alakja létezik. A # jel például bevezetheti egy függvénytörzs előtt az abban használt lokális állandó

deklarációját. A where kifejezésben megadott lokális definíciókban a tartalmazó függvény formális paramétereire közvetlenül hivatkozhatunk.

(Miranda): Lokális deklarációk

```
masodfoku :: num->num->num->[num]
masodfoku a b c = error "nem másodfokú",          if a = 0
                = error "komplex gyökök",         if delta < 0
                = [-b/(2*a)],                     if delta = 0
                = [-b/2*a + radix/(2*a),
                   -b/2*a - radix/(2*a) ],        otherwise
                where
                delta = b * b - 4 * a * c
                radix = sqrt delta
```

(Haskell): Lokális deklarációk

```
masodfoku :: Float->Float->Float->[Float]
masodfoku a b c | a == 0        = error "nem másodfokú"
                | delta <  0    = error "komplex gyökök"
                | delta == 0    = [-b/(2*a) ]
                | delta >  0    = [-b/(2*a) + radix/(2*a),
                                   -b/(2*a) - radix/(2*a) ]
                                where
                                delta  = b* b - 4.0 * a * c
                                radix  = sqrt delta
masodfoku 1 (-3) 2    -- Megj.: a masodfoku 1 -3 2  rossz!
```

(Clean): Lokális deklarációk

```
masodfoku :: Real Real Real -> (String,[Real])
masodfoku a b c
 | a == 0.0     = ("nem másodfokú",[])
 | delta < 0.0  = ("komplex gyökök",[])
 | delta == 0.0 = ("egy gyök",[-b/(2.0*a)])
 | delta > 0.0  = ("ket gyök", [(-b+radix)/(2.0*a),
                                (-b-radix)/(2.0*a)])
 where
     delta = b*b-4.0*a*c
     radix = sqrt delta
```

(SML): Lokális deklarációk

```
fun masodfoku a b c   =
   let
      val    delta  = b* b - 4.0 * a * c;
      val radix  = Math.sqrt delta;
   in
      if  a = 0.0           then ("nem másodfokú",[])
      else if  delta  < 0.0 then ("komplex gyökök",[])
      else if  delta  = 0.0 then ("egy gyök",[-b/(2.0*a)])
      else  ("ket gyök", [(-b+radix)/(2.0*a),
                          (-b-radix)/(2.0*a)])
   end
```

16.3.3. Egy érdekesebb példa: királynők a sakktáblán

Helyezzünk el egy n*n-es sakktáblán n királynőt úgy, hogy egyik sem üti a másikat! Két királynő akkor üti egymást, ha azonos oszlopban, azonos sorban vagy azonos átlón helyezkednek el. Adjuk meg az összes megoldást!

A megoldásokat n hosszúságú sorozatok sorozata formájában adjuk meg. Egy-egy megoldás azt írja le, hogy balról jobbra az egyes oszlopokban hányadik sorban helyezzük el a királynőket. Egy oszlopba tehát semmiképpen sem kerül két királynő.

Az alábbi SML-megoldás a `List.tabulate(n, fn x => x+1)`[19] sorozat, azaz az `[1,2,..,n]` sorozat permutációi között keresi a megoldásokat, így azonos sorban nem lehet két királynő. Egy-egy permutáció vizsgálatakor (`diag`) csak azt kell ellenőrizni, hogy nincs-e két királynő azonos átlón. A jó permutációkat az `accuqueens` gyűjti össze, a `cycle` pedig a következő permutációt készíti el. `accuqueens` és `cycle` kölcsönösen rekurzív, ennek megfelelően az `and` kulcsszó köti össze a két deklarációt. `@` két sorozatot fűz össze, a `(right as r::rr)` *összefogó minta* pedig lehetővé teszi, hogy ugyanarra a sorozatra a `right` azonosítóval hivatkozzunk, amelynek első elemét `r`-rel, maradékát pedig `rr`-rel jelöljük.

(SML): Királynők

```
local
 fun diag' d1 d2 []       = true
  | diag' d1 d2 (q1::qr) =
     d1<>q1 andalso d2<>q1 andalso
            diag' (d1+1) (d2-1) qr
 fun diag mid right = diag' (mid+1) (mid-1) right

 fun accuqueens []    tail res = tail :: res
  | accuqueens (x::xr) tail res = cycle [] x xr tail res
 and cycle left mid [] tail res =
          if diag mid tail then
             accuqueens left (mid :: tail) res else res
  | cycle left mid (right as r::rr) tail res =
          cycle (mid::left) r rr tail
               (if diag mid tail then
              accuqueens (left@right) (mid :: tail) res
                else res)
in
    fun queens n =
      accuqueens (List.tabulate(n, fn x => x+1)) [] []
end
```

Az alábbi program fokozatosan próbálja kiterjeszteni a megoldást jobbról balra, oszlopról oszlopra haladva úgy, hogy az új királynő a már elhelyezett királynők egyikét se üsse.

[19] A `tabulate` függvény *n* elemű listát készít, a lista elemeinek értékeit a függvényargumentum [0..n-1] sorozatra történő elemenkénti alkalmazásával kapjuk meg.

Megfigyelhetjük, hogy generátorok alkalmazásával nagyon tömör és jól olvasható megoldáshoz jutunk:

(Haskell): Királynők

```
doqueens  :: Int -> [[Int]] doqueens n = queens n n
 -- n királynő, n*n-es táblán

queens :: Int Int -> [[Int]]
queens n 0      = [ [] ]
queens n (m+1) = [ q:b | b<-queens n m, q<-[0..n-1], safe q b ]
```

A queens függvény megkeresi a következő oszlopban a jó helyeket. queens m az összes jó elhelyezés az utolsó m oszlopban, amelyek közül b egy jó elhelyezés. safe q b teljesül, ha a q-adik sorba tett királynő nem üti azokat a korábbiakat, amelyeket a b tartalmaz.

```
safe :: Int -> [Int] -> Bool
safe q b  = and [ not (checks q b i) | i<-[0.. length b - 1 ]]
```

A safe függvény ellenőrzi, hogy a q-adik sorba tett királynő nem üti a b - belieket. i befutja a b-beli oszlopindexeket. check q b i teljesül, ha az aktuális oszlop q-adik sorába tett új királynő üti a b felállás i-edik oszlopában levőt. Az and függvény elemenkénti felhasználás, értéke igaz, ha az argumentumában megadott sorozat minden eleme igaz.

```
checks ::  Int -> [Int] -> Int -> Bool
checks q b i = (q == (b !! i)) || (abs(q - (b !! i))==(i+1))
```

b !! i jelentése: a b sorozat i-edik eleme, az i-edik királynő helye. || a logikai vagy művelet, abs az abszolútérték függvény.

```
doqueens 8
```

(Miranda): Királynők

```
queens n 0 = [ [] ]
queens n (m+1) = [ q:b | b <- queens n m; q <- [0..n-1]; safe q b ]
safe q b = and [ ~checks q b i | i <- [0..#b -1 ] ]
checks q b i = (q = b!i) \/ abs(q - b!i)=(i+1)
queens 8 8
```

(Clean): Királynők

```
queens_ n 0 = [[]]
queens_ n m = [[q:b] \\ b <- queens_ n (m-1), q <- [0..n-1] | safe q b]
safe q b = and [not (checks q b i) \\ i <- [0 .. (length b)-1] ]
checks q b i = (q == b!!i) || (abs(q-b!!i)==(i+1))
queens n = queens_ n n
Start = (length(queens 8),queens 8)
```

A lusta kiértékelés lehetőséget ad *rekurzív hívások helyettesítésére* végtelen sorozatok felhasználásával.

(Miranda, Clean és Haskell): Fibonacci számok

```
fib :: Int -> Int    -- Miranda: num->num
fib 0      = 1       -- elhagyható
fib 1      = 1
fib 2      = 2
fib n = flist!!(n-2) + flist!!(n-1) -- Miranda: !! helyett !
          where
          flist = map fib [ 0 .. ]
```

A `map` függvény elemenként alkalmazza az első argumentumként kapott függvényt a másodikként kapott sorozatra. Az `flist` tehát tartalmazza a Fibonacci számokat rendre a `0`-diktól kezdve az *összes* természetes számra. Így az `n`-edik Fibonacci szám könnyen kiszámítható, mint az `flist` sorozat `n-1`-edik és `n-2`-edik elemének összege. Az `flist` lista összes elemét természetesen nem számítjuk ki, a lusta kiértékelés miatt csak azokat határozzuk meg, amelyekre valóban szükség van.

16.4. Típusok és osztályok

Ebben a részben bemutatjuk, hogy a funkcionális nyelvekben milyen nyelvi elemek segítik az adatabsztrakciót, hogyan lehet absztrakt, algebrai típusokat definiálni, típusosztályokat megadni és magasabbrendű típusokat alkalmazni.

16.4.1. Polimorfizmus, típusosztályok

A korábbiakban már több olyan függvénnyel találkoztunk, amely egymástól különböző típusú argumentumokra is alkalmazható. Ezeket a függvényeket *polimorfnak* nevezzük. Ilyen például a sorozatok első elemét visszaadó `hd` függvény, amely tetszőleges (nem üres) listára alkalmazható, függetlenül a lista elemeinek típusától. A függvény megvalósítása sem függ a típustól, ezért definíciója is megadható típusfüggetlen formában. A függvény típusának deklarációjakor egy *típusváltozót* használunk, ezzel fejezzük ki, hogy a függvény polimorf, bármilyen konkrét típusra alkalmazható.

```
hd  :: [a] -> a    // Miranda, Clean, Haskell
hd (x:xs) = x
```

A `+` műveletet is számos konkrét típus esetén használhatjuk, például egészekre, valósakra, de akár logikai értékekre is értelmezhetjük. Az összeadás minden esetben két azonos típusú értékből határoz meg egy harmadik, ugyanolyan típusú értéket, és alkalmazása is egységesen infix. A `hd` függvénynél megismert egyszerű polimorfizmussal szemben az összeadás

megvalósítása minden konkrét típus esetén más és más. Például egész, illetve valós értékeket bináris alakban általában különböző módon ábrázoljuk, így az összeadás műveletét is másképp kell elvégezni. A polimorfizmus ebben az esetben tehát más jellegű, itt valójában több, egymástól különböző függvényre használunk azonos jelölést, a művelet jelét többszörösen használjuk (ad hoc polimorfizmus, túlterhelés). Az SML csak egyes előre definiált függvények esetében engedi meg azonosítók többszörös használatát. A Cleanben és a Haskellben a programozó is deklarálhat többszörösen használt azonosítókat, de csak akkor ha a megvalósításukban különböző függvények néhány lényeges dologban azért megegyeznek (paraméterek száma, infix alkalmazás, asszociativitás, stb.). Ebben az esetben ezen műveleteknek létezik egy közös, típusfüggetlen, absztrakt szignatúrája.

A függvény *szignatúrájának* nevezzük paraméterei és eredménye típusának leírását. A + művelet absztrakt leírását típusváltozó segítségével fogalmazhatjuk meg, kicsit leegyszerűsítve például a `(+) :: a a -> a` alakban. Ebből az absztrakt szignatúrából az a típusváltozó konkrét típussal történő helyettesítésével, *példányosítással* kapjuk az egyes típusokhoz tartozó különböző összeadás műveletek szignatúráit. Ezek a műveletek megvalósításukban is eltérnek egymástól, ezért példányosításkor meg kell adnunk az adott típus értékeire vonatkozó összeadás művelet definícióját is. Az absztrakt szignatúrát Cleanben és Haskellben a `class`, a példányosítást az `instance` kulcsszó vezeti be. Az absztrakt szignatúrák és azok példányosításának megfelelő nyelvi elemek SML-ben nem az osztályok, hanem az SML modulnyelvéhez tartozó szignatúrák, struktúrák és funktorok (lásd a 16.5. részt).

Egy `class` deklaráció több egymáshoz tartozó absztrakt szignatúrát is összefoghat. Egy adott `class` deklaráció absztrakt szignatúráihoz tartozó példányok halmazát *típusosztálynak* nevezzük[20]. Ebben az esetben az absztrakt szignatúra definiálja azt is, hogy a művelet infix és balról asszociatív, amit az `infixl` kulcsszó utolsó betűje azonosít (az angol left szó kezdőbetűje). Meghatározzuk a művelet precedenciaszintjét is, ez az összeadás esetében 6.

(Clean): Típusosztály és példányosítás

```
class (+) infixl 6 a :: !a !a -> a // absztrakt (+)
double :: a  -> a | + a
double n :== n + n
 instance + Bool
 where
 (+) :: !Bool !Bool -> Bool   // példány
 (+) True b = True
 (+) a    b = b
```

A `double` függvény is polimorf, de nem kell példányosítani. A `double` definíciója a + definíciójától függ, ezt a szignatúrájában jelezzük is. Azt mondjuk, hogy a `double` egy *származtatott függvény*, értelmezése attól függ, hogy az adott típushoz definiáltuk-e, és hogy hogyan definiáltuk a + *típusosztály* megfelelő példányát.

[20] A típusosztály elemei függvények, absztrakt függvények különböző konkrét megfelelői. A *függvényosztály* elnevezés alkalmazása ebből a szempontból kifejezőbb lenne. Típusosztályról abban az értelemben beszélhetünk mégis, hogy meghatározható azon konkrét típusok halmaza is, amelyekre a `class` deklarációban megadott absztrakt függvények mindegyikének létezik példánya.

A több szignatúrát összefogó `Num` típusosztályra Haskell nyelvű példát mutatunk:

(Haskell): Numerikus típusosztály
```
class Num a where
 (+)    :: a -> a -> a
 negate :: a -> a
```

A Haskell és a Clean típusrendszere megkülönbözteti a típusállandókat a típusváltozóktól. A típusváltozók kis betűvel kezdődnek. A fenti példákban a típusváltozó. Típusállandók például a `Char`, `Int`, `Integer`, `Float`, `Double` és `Bool` típusok.

Az osztályok azonosítói általában nagybetűvel kezdődnek. Egy típus vagy függvény deklarációjánál *osztálykörnyezetet* adhatunk meg, azaz a típus vagy függvény deklarációjában, definíciójában szereplő típusváltozók lehetséges értékét osztályok megadásával korlátozhatjuk:

(Clean és Haskell): Osztálykörnyezet megadása
```
double :: Num a => a -> a -- Haskell
double :: a -> a  | +  // Clean
double x = x + x
```

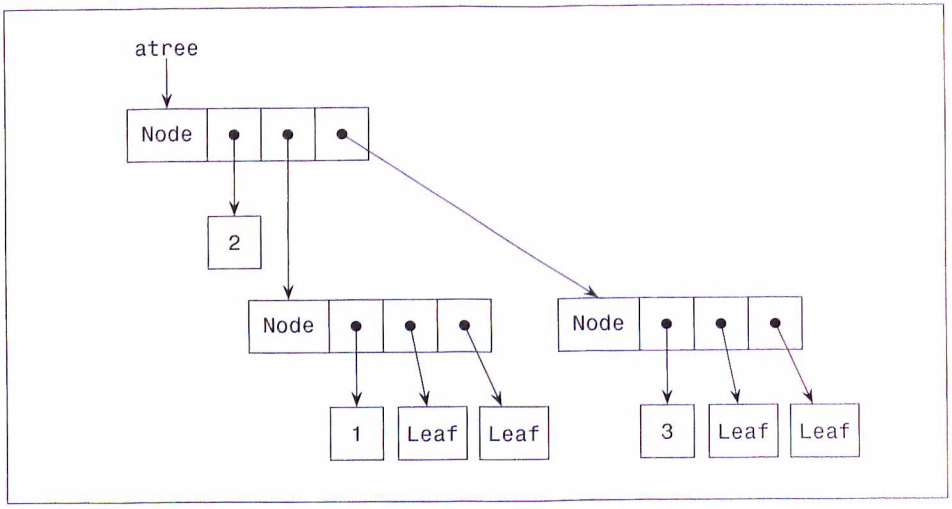
A `double` tehát a `Num` típusosztályhoz tartozó `Int` típusú értékekre alkalmazható, de a típusosztályhoz általában nem tartozó `Char` típus értékeire nem.

Egy `C1` osztálykörnyezet megadásával azt írjuk elő, hogy az `a` típusváltozó értéke csak olyan t típus lehet, amely a `C1` osztályhoz tartozik.

A függvényalkalmazás fogalmának felel meg a *típusalkalmazás* fogalma. Ha a t_1 típus például $k_1 \longrightarrow k_2$ alakú és a t_2 típus k_1-re illeszkedik, akkor a $t_1\ t_2$ típuskifejezés értéke a k_2 típus (típuslevezetés). A típusváltozók általában implicit univerzális kvantor hatáskörében állnak.

Egy kifejezés típusának mindig levezethetőnek kell lennie a Hindley–Milner féle típuslevezetési eljárás szerint. Az eredmény függ a típuskörnyezettől, amely meghatározza a kifejezésben szereplő szabad változók típusát, illetve az osztálykörnyezettől, amely a típusváltozók értékét korlátozza. A polimorf `double` függvény típusa általánosabb az `Int -> Int` típusnál, a legáltalánosabb típusleírás `a a -> a | + a` alakú. A kifejezés legáltalánosabb típusa a kifejezés alaptípusa, amely az osztálykörnyezet segítségével általában akkor is levezethető, ha a kifejezésben szereplő egyes függvénynevek többszörösen használtak (több típushoz is tartoznak). Néhány esetben meg kell azonban adnunk a szignatúrát, mert a fordító nem tudja meghatározni a legáltalánosabb típust. Az azonosítók többszörös használata viszonylag gyakori oka az automatikus típuslevezetés sikertelenségének, az SML éppen ezért korlátozza ezt a lehetőséget.

16.1. definíció: *Általánosabb típus.* Jelöljön u, v típusváltozó halmazokat, c_1, c_2 ezen típusváltozókra vonatkozó osztálykörnyezeteket, t_1, t_2 pedig ezeket a típusváltozókat is tartalmazó típuskifejezéseket. A $\forall u.c_1 \Rightarrow t_1$ típus(kifejezés) általánosabb a $\forall w.c_2 \Rightarrow t_2$ típusnál, ha van olyan S helyettesítés, amely az u-beli típusváltozókat úgy helyettesíti, hogy t_2 megegyezik $S(t_1)$-gyel és ha c_2 teljesül, akkor $S(c_1)$ is.



16.4. ábra: Fa típusú érték: `atree`.

Típusokat megadhatunk algebrai adattípusként, származtatott típusként, és típusszinonímaként. A típuskonstrukció-osztályokat az algebrai adattípus fogalmának bevezetése után ismertetjük.

16.4.2. Algebrai adattípus

Algebrai adattípus deklarációjában egyidejűleg vezetünk be egy új típust (típuskonstrukciót) és annak adatkonstruktorait. Kivétel nélkül minden, az adott típushoz tartozó érték a típus deklarációjakor megadott valamelyik adatkonstruktor függvény alkalmazásával áll elő. A következő példában egy felsorolási típus algebrai típusdefinícióját adjuk meg. A `Nap` konstans (nulláris) típuskonstruktor, a `Het`, a `Kedd`, stb. pedig konstans adatkonstruktor függvények, melyek értéke `Nap` típusú.

```
:: Nap = Het | Kedd | Sze | Csu | Pen | Szo | Vas
```

A `Tree` típuskonstrukció tetszőleges alaptípusból bináris fa típust hoz létre; a `Tree` típuskonstruktor egyváltozós, argumentuma az `a` típusváltozó. A típuskonstruktorokat tekinthetjük magasabbrendű típusoknak is, melyek egy típusból egy másik típust hoznak létre. A típuskonstrukció adatkonstruktorai közül a `Node` adatkonstruktor-függvény háromváltozós, első paramétere egy `a` alaptípusú érték, amit a fa csúcsában tárol, második és harmadik paramétere pedig egy-egy `Tree` típusú érték. A deklaráció arra is példa, hogy a típuskonstruktort rekurzívan is használhatjuk.

```
:: Tree a = Node a (Tree a) (Tree a)
          | Leaf

atree = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)
```

A konstruktorok hatásköre és láthatósága kiterjed arra a teljes modulra, amely a deklarációt tartalmazza.

A Mirandában a `::=` jel vezeti be az algebrai adattípust és a `*` jelöli a típusváltozót.

(Miranda): Algebrai adattípus

```
bool ::= True | False
nat ::= Zero | Suc nat
list * ::= Nil | Cons * (list *)
color ::= Red | Orange | Yellow | Green | Blue | Indigo | Violet
bool_or_num ::= Left bool | Right num
tree * ::= Nilt | Node * (tree *) (tree *)
```

A Haskellben a `data`, az SML-ben a `datatype` kulcsszó vezeti be az algebrai típusdefiníciót. A halmaz típus algebrai definíciójára alább egy Haskell nyelvű példát adunk.

(Haskell): Algebrai adattípus

```
data Eq a => Set a = NilSet | ConsSet a (Set a)
```

Az adatkonstruktorok típusa:

```
NilSet :: Set a
ConsSet :: Eq a => a -> Set a -> Set a
```

A `Set` típuskonstruktor és a `ConsSet` konstruktor egyaránt csak az `Eq` típusosztályhoz tartozó típus esetében alkalmazható. A

```
data ( C1 a b , C2 b ) => T a b = ...
```

általános alakú típusdefinícióban a típusváltozók lehetséges értékeire osztálykörnyezet megadásával megszorításokat írhatunk elő.

Az egyes konstruktorokhoz tartozó mezőknek nevet is adhatunk:

```
data ( C1 a b , C2 b ) => T a b = K1 {f1 :: a, f2 :: b} ...
```

Ily módon Haskellben is létrehozhatunk rekordra emlékeztető olyan direktszorzat adattípust, ahol mintaillesztést alkalmazva mezőnevekkel dolgozhatunk.

(Haskell): Rekord

```
data Szemely = Ferfi {nev::[Char], kor::Int}
             | No {nev::[Char]}
```

Ezzel valójában egy variáns rekordot hoztunk létre. Ekkor anélkül, hogy előre tudnánk, hogy az aktuális `x Szemely` férfi vagy nő, egyetlen `case` kifejezéssel definiálhatunk egy olyan típusértéket, ahol a név mező értéke `"Gabi"`:

```
case x of Ferfi _ kor -> Ferfi "Gabi" kor
          No _    -> No "Gabi"
```

Az alábbi SML példa a lista típus egy lehetséges algebrai definíciója. Az SML-ben a típusváltozót mindig egy ' jel előzi meg és a típusváltozó a típuskonstruktor előtt van. Az adatkonstruktort az `of` kulcsszó választja el argumentumától.

(SML): Algebrai adattípus

```
datatype 'a List = Nil | Cons of 'a * 'a List
```

A Mirandában az `abstype` kulcsszó segítségével deklarálhatunk absztrakt adattípust, míg az SML-ben, Cleanben és Haskellben a modulrendszer az adatabsztrakció megvalósításának eszköze (lásd a 16.5. részt).

(Miranda): Absztrakt, algebrai adattípus specifikáció

```
abstype tree *
with mirror :: (tree *) -> (tree *)
     empty  :: (tree *)

numtree ::= tree num
```

A most bevezetett fa típushoz mindössze két műveletet deklaráltunk, az `empty`-t, amelynek értéke egy üres fa, illetve a `mirror`-t, amely egy tetszőleges fát tükröz. A `tree` típuskonstrukció algebrai típusdefiníciója rejtett, az adatkonstruktorok hatásköre pedig korlátozott, a program más részeiben nem használhatók.

```
tree * == Nilt | Node * (tree *) (tree *)
mirror Nilt = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x )
```

16.4.3. Típusszinonimák

Típusszinonima deklarációja nem vezet be új típust; a szinonima ekvivalens az eredeti típuskifejezéssel, szintaktikai rövidítésnek tekinthető. Típusszinonimára vonatkozó rekurzió, illetve példányosítás nem megengedett.

(Miranda): Típusszinonimák

```
string == [char]
matrix == [[num]]
```

(Haskell): Típusszinonimák

```
type List = []
type Rec a = [Circ a]
data Circ a = Tag [Rec a]
```

16.4.4. Származtatott típusok

A Haskellben az eredeti típussal azonos reprezentációjú, de nem ekvivalens típust kapunk a `newtype` kulcsszó használatával végzett származtatással. A származtatott típusok esetében a példányosítás is megengedett.

16.4.5. Típuskonstrukció osztályok

A Cleanben készíthetünk olyan osztályokat is, amelyek példányait nem típusokra, hanem típuskonstrukciókra definiáljuk. Ilyen a `length` vagy a `map` függvény, amely egy tetszőleges adatszerkezet elemeinek számát határozza meg, illetve tetszőleges adatszerkezetre elemenként alkalmaz egy függvényt.

Az osztály deklarációjában a `t` típuskonstruktor változó szerepel, amely felveheti a lista vagy a fa[21] típuskonstruktor értékét.

(Clean): Típuskonstrukció osztály

```
class map t :: (a->b) (t a) -> (t b)
instance map []
   where map f l = [ f e \\ e <-l]
:: MTree a = NNode a [MTree a]
instance map MTree
   where map f (NNode el ls) =
            NNode (f el) (map (map f) ls)
```

A Haskellben a `Functor` monadikus osztály (lásd a 16.6. részt) általánosítja a `map` függvényt típuskonstruktorokra, `fmap` néven.

16.5. Modulok

Míg a Haskellben csak egyfajta modul létezik, a Cleanben definíciós és implementációs modulokat hozhatunk létre. A definíciós modulhoz általában tartozik egy implementációs modul is, ketten együtt alkotnak egy egészet. A definíciós modul interfészt nyújt az implementációs modul felhasználói számára; ők csak azt látják a modulból, amit a definíciós modulban megadtunk. A definíciós/implementációs modulpár a Clean nyelvben az adatabsztrakció eszköze. Cleanben a modulok az `import` utasítás használatával hivatkoznak egymásra, az importálás tranzitív. A Haskellben a definíciós modult az export lista helyettesíti, ennek segítségével adjuk meg, hogy milyen függvények, illetve adattípusok legyenek a modulon kívül láthatók. Az export lista a modul fejlécében helyezkedik el.

(Haskell): Modulok

```
module Uj ( fv1 ) where
fv1  x = x
fv2  x = 3 + x
```

Mindenki, aki az `Uj` modult importálja, csak az `fv1`-et fogja látni, az `fv2`-t nem. Ha nincs export lista, minden, amit a modulban definiáltunk látható kívülről.

Mindkét nyelv lehetőséget ad arra is, hogy egy modulból ne importáljunk mindent, amit az exportál, hanem csak azt, amire valóban szükségünk van.

A Haskell és az SML lehetőséget ad minősített nevek használatára is. Ezek akkor hasznosak, ha más modulokban definiált azonosítókat vegyesen szeretnénk használni. Előfordulhat ugyanis, hogy ugyanaz az azonosító több importált modulban is szerepel, esetleg más

[21] Ebben a példában egy új – nem bináris – fa típuskonstrukció szerepel.

funkcióval. A minősített név a következő alakú: *minősítő.azonosító*; a minősítő általában a modul neve, de megadhatunk más nevet is a modul importálásakor.

16.5.1. Absztrakt algebrai adattípus

Az alábbi példa segítségével bemutatjuk, hogy a Clean modulszerkezetének segítségével hogyan valósíthatunk meg absztrakt adattípust. A verem típus reprezentációját az implementációs modulban adjuk meg, típusszinonima formájában. A reprezentációt a definíciós modul nem tartalmazza, tehát a vermet reprezentáló listát az implementációs modulon kívül más modulokból nem érhetjük el, csak a definíciós modul által exportált absztrakt műveleteket használhatjuk.

(Clean): Modulok és absztrakt adattípus

```
definition module stack
:: Stack a
Push :: a (Stack a) -> Stack a
Pop  :: (Stack a) -> Stack a
top  :: (Stack a) ->  a
Empty :: Stack a

implementation module stack
:: Stack a :== [a]
Push :: a (Stack a) -> Stack a
Push e s = [e:s]
Pop  :: (Stack a) -> Stack a
Pop [e:s] = s
top  :: (Stack a) -> a
top [e:s] = e
Empty :: Stack a
Empty = []

module teszt
import StdEnv, stack
Start = top (Push 1 Empty)
```

Az SML modulszerkezete gazdagabb, a modulok kezeléséhez használható nyelvi elemeket az SML modulnyelve írja le [Har 01]. A teljes modulnyelv leírása meghaladja a fejezet kereteit, csak néhány fontos nyelvi elemet emelünk ki.

A Clean definíciós moduljainak a *szignatúrák*[22] felelnek meg, a *struktúrák* pedig az implementációs moduloknak. Fontos eltérés azonban, hogy az SML ismeri a szignatúrakifejezés és a struktúrakifejezés fogalmait is, amelyeknek értékük is van és argumentumként is átadhatók.

A szignatúrákat tekinthetjük a struktúrák típusleírásának. Minden struktúrának létezik elsődleges, levezethető szignatúrája, amely a (preorder) megfelelési reláció [Har 01] szerint összehasonlítható egy specifikációt megjelenítő szignatúrával.

[22] Itt a szignatúra szó az SML nyelvi elemét azonosítja. Az SML szignatúrái egyfajta általánosítását jelentik a függvények szignatúráinak, egy egész struktúra interfészét írják le.

A szignatúrákra vonatkozóan kétféle öröklődési relációt is értelmez az SML modulnyelve. Egy szignatúra kiterjeszthet (tartalmazhat) egy másikat, illetve finomítása (specializációja) lehet egy másiknak (lásd a következő példát). A kiterjesztés alkalmas lehet új műveletek bevezetésére, a finomítás pedig a reprezentáció módját előírva, a szignatúrát a benne deklarált absztrakt típusokra adott definíciókkal egészítheti ki.

(SML): Szignatúrák – kiterjesztés, finomítás

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end

signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
  end

signature QUEUE_AS_LISTS =
  QUEUE where type 'a queue = 'a list * 'a list
```

Az importálás megfelelője a szignatúrák közötti öröklődési, illetve a nyitott struktúra (open), amelynek alkalmazása esetén az importált struktúra teljes belső tartalma láthatóvá válik az azt használó modulban.

Egy struktúra akkor felel meg egy szignatúrának, ha elsődleges szignatúrája tartalmazza mindazt, amit a szignatúra előír .

Az ORDERED szignatúra az ORD osztály (lásd a 16.4. részt) SML-megfelelője, amelynek egy lehetséges megvalósítása a LessInt struktúra. A megfelelési relációt fejezi ki a : jelölés, amit úgy is olvashatunk, hogy a LessInt struktúra ORDERED típusú.

(SML): Szignatúra és példány

```
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
  end
structure LessInt : ORDERED =
  struct
    type t = int
    val eq = (op =)
    val lt = (op <)
  end
```

Ahogy az osztályokhoz több példány is tartozhat, úgy a szignatúráknak is több különböző megvalósítása lehet egyszerre. Ezek közül néhány egy-egy másik struktúrában beágyazva is megjelenhet.

A szignatúrák implementációja esetén a megfelelő struktúra létrehozásakor a `:>` előírás alkalmazásával rendelkezhetünk arról is, hogy a struktúrából – a struktúra használatakor – csak azok az egyedek legyenek majd láthatók, amelyek a szignatúrában is megjelennek. Az ily módon létrehozott struktúra az *adatabsztrakció* eszköze. Az alábbi példa az absztrakt sor megvalósítását mutatja be [Har 01]. A sort listák rendezett párja reprezentálja. A `QUEUE_WITH_EMPTY` szignatúra (lásd a 617. oldal példáját) tartalmazta a típusspecifikációt, amelynek úgy felel meg a `Queue` struktúra, hogy a `:>` előírás szerint belső szerkezete *átlátszatlan* marad.

(SML): Modulok és absztrakt adattípus

```
structure Queue :> QUEUE_WITH_EMPTY =
  struct
    type 'a queue = 'a list * 'a list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    exception Empty
    fun remove (nil, nil) = raise Empty
      | remove (bs, f::fs) = (f, (bs, fs))
      | remove (bs, nil) = remove (nil, rev bs)
  end
```

Az SML-ben arra is lehetőségünk van, hogy modulokat struktúrákkal paraméterezzünk. A paraméterezett modul nyelvi megfelelője a `functor`, amely az Ada sablonhoz hasonló eszköz. A következő példán [Har 01] megfigyelhetjük, hogy a `DICT` szignatúrának megfelelő `DictFun` funktort a `K` struktúrával paraméterezzük. A `dict` típus egy olyan típuskonstrukciót specifikál, amelyben az elemeket, amelyek (kulcs, érték) párból állnak, kulcs szerint kereshetjük. A `DictFun` struktúra egy olyan algebrai adattípussal megfogalmazott adatszerkezettel reprezentálja a szignatúrában megadott típust, amelyben a kulcs típuson értelmezett rendezési reláció alapján a keresés hatékonyan megvalósítható. Ezért a kulcs típust leíró, paraméterként kapott `K` struktúráról kikötjük, hogy megfelel az `ORDERED` szignatúrának (a definíciót lásd a 617. oldalon). A reprezentáció és az implementáció a `:>` előírás szerint átlátszatlan, a megvalósítás részleteit most mi is mellőzzük.

A `K` aktuális értéke lehet például a `LessInt` struktúra is (617. oldal), amely az egész számokra a szokásos módon értelmezi a rendezési relációt. Megjegyezzük, hogy a `LessInt` úgy felel meg a az `ORDERED` struktúrának, hogy a struktúra belső szerkezete látható marad (a `:` előírásnak megfelelően), tehát nem rejtjük el, hogy a (`K.t`) kulcs típust az egész számok reprezentálják.

Az `LtIntDict` struktúra a `DictFun` funktor egy olyan példánya lesz, ahol a kulcs típus az egész számok típusa, míg a (kulcs,érték) párban elhelyezett értékek típusa (az `LtIntDict.dict` típuskonstruktor paramétere) továbbra is szabadon megválasztható.

(SML): Funktor

```
signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
  end

functor DictFun (structure K : ORDERED) :> DICT
  where type Key.t = K.t =
struct
  structure Key : ORDERED = K
  datatype 'a dict = ...  (* reprezentáció *)
  val empty = Empty
  fun insert  ... =  ...  (* implementáció *)
  fun lookup  ... =  ...
end

structure LtIntDict = DictFun (structure K = LessInt)
```

16.6. Frissíthető változók, monádok, mellékhatás

A bemeneti/kimeneti műveletek szükségszerűen mellékhatással járnak, megváltoztatják a program környezetét, a képernyő tartalmát, háttértárolón elhelyezett fájlokat, stb. A tisztán funkcionális nyelvekben is vannak olyan elemek, amelyeknek van mellékhatásuk, különben nem lehetne ezeken a nyelveken a külső környezetre ható programot írni. A mellékhatásokat lehetőség szerint a programszöveg jól azonosítható részeire kell korlátozni, és a hivatkozási helyfüggetlenség sérülését is el kell kerülni.

16.6.1. Egyszeresen hivatkozott változók

A Clean tervezői bizonyos korlátok között megengedik az előző értéket megsemmisítő értékadást. Egy *egyszeresen hivatkozott* (unique) objektumnak újra értéket adhatunk, de csakis akkor, ha az objektumra csak összesen egyetlen helyen hivatkozunk. Ekkor ezen a helyen biztonságosan megváltoztathatjuk az egyed által elfoglalt memóriaelemben tárolt értéket, hiszen senki más nem használja többé a korábbi értékét, a hivatkozási helyfüggetlenség ilyenkor nem sérül. A régi egyed többé nem létezik, az általa korábban elfoglalt memóriaterületet a szemétgyűjtő helyett rögtön egy új egyednek adjuk át. Ez hatékonysági szempontból gyakran előnyös lehet, például nagy adatszerkezetek használata esetén. Ha az adatszerkezet egy olyan változatára van szükségünk, amelyik csak egyetlen helyen különbözik az

egyszeresen hivatkozott eredetitől, akkor a megfelelő elem felülírása megengedett és hatékonyan megvalósítható. Fájlműveletek esetén is hasonló a helyzet, nagy fájlok másolatainak elkészítése helyett a fájl megfelelő rekordját írhatjuk felül.

A Cleanben az egyszeresen hivatkozott vázzal rendelkező tömbök tartalmát is újra definiálhatjuk. Az alábbi példában a `mArray5` egy olyan tömb, amely váza a memóriában az egyszeresen hivatkozott vázzal rendelkező `Array5` tömb vázának helyére kerül, és elemei megegyeznek a `Array5` tömb elemeivel, kivéve a 3-as és 4-es indexhez tartozó értékeket, amelyek új értékét az & jel után adtuk meg. Az `mArray` tömbben minden egyes érték más lesz, mint az `mArray`-ben azonos indexhez tartozó érték[23].

(Clean): Tömbök

```
Array5 :: *{Int}
mArray5 = { Array5 & [3]=3, [4]=4 }
mArray  = { Array5 & [i]=k \\ i <- [0..4] & k <-[80,70..]}
```

Azt, hogy nincs-e egynél több hivatkozás az egyedre, a fordító ellenőrzi. Minden „értékadáskor" új nevet is kellene adunk a létrejövő új egyednek, de mivel több hivatkozás már nincs az adott nevű régi egyedre, egy lokális definíció alkalmazásával a nevet is gyakran újra felhasználjuk. Ettől kezdve a név már az új egyedet azonosítja. Ezzel a módszerrel az imperatív stílushoz hasonlóan programozhatunk (amikor ez feltétlenül szükséges), anélkül, hogy a tisztán funkcionális programozás kereteit átlépnénk. A megsemmisítő értékadások használatához szükség van sorrendjük meghatározására is, különben nem lehetne tudni, hogy az újra felhasznált nevek éppen melyik értékadás eredményét jelölik. Erre mutatunk példát a 623. oldal *Párbeszéd* nevű programjában.

16.6.2. Monádok

A monád[24] egy kategória-elméleti fogalom, három művelet algebrai tulajdonságainak leírására szolgál [Bar 90]. A Haskellben ennek az algebrai fogalomnak egy programozási nyelvi megfelelőjével találkozunk.

A Haskell három monadikus osztályt definiál: a `Functor` (16.4.5.), `Monad` és `MonadPlus` osztályokat. Ezeket olyan típuskonstrukciókkal példányosíthatjuk, mint például az I/O típuskonstrukció (lásd a 16.7. részt).

A monadikus osztályok absztrakt műveleteihez olyan axiómák tartoznak, amelyeket az osztály minden példányosításakor figyelembe kell venni. Például a `Functor` monád `fmap` függvényére teljesülnie kell az elemenként feldolgozható függvényekre [Fót 83] ismert szabályoknak: `fmap id = id`, `fmap (f . g) = fmap f . fmap g`.

A `Monad` és `MonadPlus` monádosztályok példányaihoz tartozó alapműveletek a monád értékeinek, az úgynevezett *monadikus műveleteknek* az[25] összekapcsolását segítik, ezáltal a monadikus műveletek kiértékelésének sorrendjét határozzák meg. A következő monadikus

[23] A második & jel a `//` után megadott két listagenerátor párhuzamos működését írja elő, így a tömb értéke `[80,70,60,50,40]` lesz.

[24] A monád szó szinonimájaként a szakirodalomban használják a triple-t is.

[25] Különbséget teszünk a monád három alapművelete és a monád értékei, a *monadikus* műveletek között!

műveletnek (például I/O művelet) mindig meg kell várnia a megelőző monadikus művelet eredményét. A monadikus műveletek objektumként is viselkedhetnek, a művelet argumentuma és eredménye egy belső állapotot is tartalmazhat, amelyet a művelet megváltoztat. A monadikus műveletek tehát lépésről lépésre egymásnak adhatják tovább a mindig egyszeresen hivatkozott, belső állapotteret. Az I/O műveletek például a külső környezetet viszik magukkal.

A `Monad` osztálynak két fő művelete van. A `return` művelet monadikus értékké alakítva, becsomagolva adja vissza az argumentumként kapott típusértéket, a `>>=` alapművelet pedig két – azonos típuskonstrukcióhoz tartozó – monádérték, azaz két monadikus művelet összekapcsolását biztosítja. A `>>` változat úgy kapcsol össze monadikus műveleteket, hogy az első eredményét a második nem használja fel.

A `fail` alapművelet hibakezelésre alkalmas. A `do` kulcsszó segítségével monadikus műveleteket egyszerűbb formában fűzhetünk egymás után, a `do` hatáskörét a margó szabály határozza meg.

```
do  e1 ; e2       =  e1 >>      e2
do  p <- e1; e2   =  e1 >>= (\v -> case v of p -> e2;
                                             _ -> fail "s")
```

A Haskellben kizárólag a monádok okozhatnak mellékhatásokat, így a mellékhatást okozó programrészek jól azonosíthatók és a mellékhatás a monádokon belülre korlátozódik. Az `IO` monádot (lásd a 16.7. részt) használjuk a 622. és a 632. oldal példáiban. A monádok kifejezőereje nagy, számos nyelvi konstrukció felépítésére alkalmasak [Hud 99]. A monádok segítségével például kis, imperatív célnyelvek szemantikáját adhatjuk meg és az ezen nyelveken írt programok kiértékelését végezhetjük el. Ilyen imperatív célnyelvnek tekinthetjük például a Haskellben az előre definiált `IO` monádot is, amelynek segítségével a programszöveg jól elhatárolható részleteiben imperatív stílusban programozva végezhetünk I/O műveleteket.

16.6.3. Frissíthető változók

Az SML-ben a `ref` kulcsszó jelöli a frissíthető változókra (cellákra) hivatkozó mutatókat. A `val r = ref 0`, `val r = !r + 1` SML függvények kiértékelése során az `r : int ref` mutató először a `0 : int` értékre mutat. Az `!r` explicit hivatkozás-feloldással kiolvassuk ezt az értéket és felül is írhatjuk. Az `r` által mutatott memóriacella új értéke `1` lesz.

Frissíthető elemeket tartalmazó tömböket az `Array` modul segítségével használhatunk. Az `array` függvény adott méretű tömböt hoz létre és meghatározza a kezdőértékét, a `sub` függvény alkalmazásával az `i`-dik elemet érhetjük el, az `update` függvénnyel pedig frissíthetjük az értékét:

```
val m : int array = Array.array (size, initv)
   Array.sub (memopad, i)
Array.update (memopad, i, value);
```

16.7. Interaktív funkcionális programok

A Haskell nyelv a program külső állapotterét az `IO` monádok belsejében rejti el. Erre az állapottérre a programozó közvetlenül nem is hivatkozik az I/O műveletek során.

Az `IO` monád a `Monad` osztály `IO a` típuskonstrukcióval való példányosításával jön létre. A `getChar :: IO Char` és a `putChar :: Char -> IO ()` műveletek is az `IO` monádhoz tartoznak, utóbbi alaptípusa az egységelemet tartalmazó rendezett nullás típus. A két I/O művelet értékének alaptípusa különböző ugyan, ennek ellenére összekapcsolhatók, mert mindkettő az `IO` monádhoz tartozik. A `getLine`, `putLine` műveletek a `getChar`, `putChar` segítségével építhetők fel. Az alábbi példa azt mutatja be, hogy a `do` kulcsszó segítségével imperatív stílusban írhatunk I/O programokat a Haskellben.

(Haskell): Egyszerű karakteres io

```
helloworld :: IO ()
helloworld = putStr "Hello world"

read2lines :: IO ()
read2lines
    = do linea <-  getLine
         lineb <-  getLine
         putStrLn  (reverse lineb)

getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)


toScreen :: IO ()
toScreen = print 5
```

A Clean nyelv a program külső állapotterét a `*World` (külvilág) absztrakt típusban rejti el, amelynek típusértékei csak egyszeresen hivatkozhatóak. A Haskell `IO` monáddal szemben a Cleanben a programozónak közvetlenül kell hivatkoznia a külső állapottérre, vigyázva arra, hogy ezek a hivatkozások mindig egyszeresek legyenek (különben a fordító hibát jelez). Az alábbi program típusleírásából látszik, hogy a `Start` függvény egy új „világot" készít a paraméterként kapott „külvilágból". Az új külső környezet felhasználói események eredményeként jön létre lépésről lépésre. A `stdio` függvény segítségével egy olyan új `w` világot kapunk[26], amelyben a `console` konzol már megnyitott állapotban van. Azt ezt követő lépések során az újabb és újabb konzolra karakteres üzenetek kerülnek, illetve a felhasz-

[26] A `w` változó *nem* kapott új értéket, a baloldalon álló `w` egy új azonosító, amelynek lokális hatáskörén belül a külső `w` nem látszik.

náló karakteres formában megadott válasza alapján a következő konzol[27] és a name lokális konstans értékét egyidejűleg definiáljuk.

(Clean): Párbeszéd

```
module helloconsole
import StdEnv
Start :: *World -> *World
Start w
    # (console,w) = stdio w
      console      = fwrites "Írja be a nevét!:\n" console
      (name,console) = freadline console
      console = fwrites ("Örömmel látom " +++ name +++ "!") console
      (_,console) = freadline console
      (ok, nw) = fclose console w
    | not ok = abort "error"
    | otherwise = nw
```

Az SML a PRIM_IO, STREAM_IO, IMPERATIVE_IO modulokkal biztosít adatáramlás alapú I/O lehetőséget. A STREAM_IO felület puffereli a bemenetet és a kimenetet is. Az IMPERATIVE_IO modul a már egyszer megnyitott adatfolyamok dinamikus átirányítását is megengedi. A TEXT_IO modul a STREAM_IO karakteres I/O kezelésére készült példánya.

(SML): Adatáramlás alapú I/O

```
let
  ins = TextIO.stdIn
  outs = "output.txt"
in
  TextIO.openIn(ins);
  TextIO.openOut(outs);
  let
    line = TextIO.inputLine()
  in
    TextIO.output(outs,line);
    TextIO.output1(outs,#"\n");
    TextIO.flushOut(outs);
  end
end
```

Az SML-nek és a Haskellnek nincs szabványos grafikus I/O könyvtára, a Clean Object IO könyvtára [Ach 00] segítségével viszont könnyen kezelhetünk menüket, párbeszédablakokat, ablakokat. Az alábbi példán megfigyelhetjük, hogy a felhasználói felületet algebrai típusokhoz (például Dialog) tartozó értékek definiálásával írhatjuk le. Az egyes felhasználói akciókhoz, például az ablak bezárásához (WindowClose) függvényeket (például

[27] A console sem kap új értéket, mindig új konzol jön létre.

`CloseProcess`) rendelhetünk hozzá, amelyeket az `Object IO` eseménykezelője értékel ki. A `startIO` függvény argumentumában adott `openDialog` függvény kiértékelésének megkezdésével, a párbeszédablak felépítésével indul a felhasználói felülethez rendelt interaktív folyamat.

(Clean): Object IO

```
module hellooio
import StdEnv, StdIO
Start :: *World -> *World
Start world
 = startIO NDI Void (snd o openDialog undef hello) [] world
 where
   hello = Dialog "" (TextControl "Hello world!" [])
                [WindowClose (noLS closeProcess)]
```

16.8. Kivételkezelés

A Miranda és a Clean kivételkezelése lényegében csak arra ad lehetőséget, hogy az `error`, illetve `abort` függvény segítségével a program írója saját maga által megfogalmazott hibaüzenetet küldjön a felhasználónak, mielőtt a program kiértékelése megszakad.

A Haskell `Monad` és `MonadPlus` osztályai (16.6.2. rész) tartalmaznak egy `fail` alapműveletet, amelynek segítségével lehetőségünk van kivételkezelésre monádokban. Az `IO` monád (16.7. rész) tartalmaz is kivételkezeléshez szükséges műveleteket. A kivételek ebben az esetben az `IOError` előre definiált absztrakt típushoz tartoznak. A `catch` függvény segítségével az egyes I/O hibákhoz kivételkezelőt rendelhetünk, az `ioError` függvény pedig a nem kezelt kivétel továbbadására szolgál [Hud 99].

Az SML kivételkezelése általánosabb, az Adáéval rokon. A programozó az `exn` típushoz tartozó olyan kivételeket definiálhat[28], amelyek lehetnek konstans függvények vagy konstruktorfüggvények[29]. A deklarált hibákat ki lehet váltani, és a hívó a kiváltott üzenetekre kivételkezelő függvények kiértékelésével válaszolhat. Az alábbi példában [Har 01] a faktoriális (parciális) függvényhez deklarálunk kivételt, amelyet akkor váltunk ki, ha argumentuma negatív érték. A `factorial_driver` függvény több más lehetséges kivétellel együtt kezeli az általunk deklarált kivételt is.

(SML): Kivételkezelés

```
exception Factorial
local
    fun fact 0 = 1
      | fact n = n * fact (n-1)
in
```

[28] Az `exn` típus kiterjeszthető típus, ami azt jelenti, hogy adakonstruktorai a típuskonstruktor megadásától elkülönítve, később is deklarálhatóak.

[29] A kivétel kiváltásakor az `exn` értéket előállító adatkonstruktor adott típusú értéket csomagol be.

```
    fun checked_factorial n =
        if n >= 0 then
            fact n
        else
            raise Factorial
end
fun factorial_driver () =
    let
        val input = read_integer ()
        val result = makestring (checked_factorial input)
        val _ = print result
    in
        factorial_driver ()
    end
    handle EndOfFile => print "Done.\n"
      | SyntaxError =>
        let val _ = print "Syntax error.\n" in factorial_driver () end
      | Factorial =>
      let val _ = print "Out of range.\n" in factorial_driver () end
```

16.9. Párhuzamos kiértékelés és elosztott programok

Az alábbiakban olyan nyelvi elemeket mutatunk be Clean, Haskell, illetve JoCaml példákon keresztül, amelyek segítségével osztott, párhuzamos programokat készíthetünk funkcionális stílusban. A nyelvek kifejező ereje különböző, az egyes elemek absztrakciós szintje eltérő. Bemutatjuk a legalacsonyabb absztrakciós szintet képviselő annotációkat, az azokra épülő kiértékelési módszereket, az explicit üzenetküldés eszközeit, a csatornákat, a mobil funkcionális kód megfelelőit, a `Dynamic`-ot és az ágenseket. Ezek hatékonyságban, alkalmazhatóságban, és megvalósításukban is nagyon különböznek egymástól. A megoldandó feladatnak megfelelően választhatunk a bő kínálatból és hozhatunk létre párhuzamos és elosztott alkalmazásokat.

A függvények kompozíciója asszociatív, így a funkcionális nyelvű programok kiértékelése jól párhuzamosítható. A legfontosabb kérdés annak eldöntése, hogy mely részkifejezések kiértékelését célszerű párhuzamos vagy elosztott módon elvégezni.

A párhuzamos és elosztott funkcionális programozás világában többféle irányvonal létezik [Zsó 02]: új absztrakt nyelvi elemek bevezetése, párhuzamos és elosztott függvénykiértékelés létrehozása, a kiértékelési módszerek módosítása, dinamikusan összeszerkesztett mobil kód, valamint TCP/IP kommunikációs protokoll használata. Ezeket a nyelvi lehetőségeket mutatjuk be Concurrent Clean [Kes 96, Ach 95, Hor 00, Ser 99, Ach 00] és Distributed Haskell with Ports [Huc 01], valamint JoCaml[30] [Fou 01] példák segítségével.

[30] ML változat, amely imperatív és objektumorientált elemeket is tartalmaz.

16.9.1. Párhuzamos és elosztott programozás Concurrent Cleanben

Elsőként a Concurrent Clean nyelvben található, a párhuzamosságot kifejező egyik legrégebbi funkcionális nyelvi elemet szeretnénk bemutatni, az annotációkat [Kes 96]. Az annotációk segítségével meghatározhatjuk, hogy a kiértékelő rendszer egy függvény vagy egy kifejezés mely részeit értékelje ki párhuzamosan. A lusta nyelvekben általában csak olyan kifejezések kiértékelésére kerül sor, amelyek értékére már szükség van. A párhuzamos kiértékelés kivételt képez ez alól a szabály alól. Az annotációk segítségével, *spekulatív módon* meghatározhatjuk, hogy szerintünk mely részkifejezések kiértékelését érdemes párhuzamosan előre elkezdeni, hogy amikor később szükség lesz a kifejezés értékére, akkor annak normál formája már rendelkezésre álljon. Három típusú annotációt különböztetünk meg:

- `I`: összefésüléses annotáció, mely lehetővé teszi egy kifejezés párhuzamos kiértékelését ugyanazon a processzoron, ahol az `I` annotációt tartalmazó kifejezés kiértékelése folyik. A két kiértékelő folyamat egymással párhuzamosan, felváltva dolgozik[31].
- `P`: párhuzamos annotáció, mely egy kifejezés ugyanazon, vagy lehetőség szerint egy másik processzoron történő párhuzamos kiértékelését jelöli.
- `P at procid`: párhuzamos kiértékelést végez megadott processzoron.

A következő függvény annotációk segítségével számítja ki a Fibonacci szám értékét:

(Clean): Annotációk

```
Nfib :: Int -> Int
Nfib n   | n < 2  = 1
                   =   {| P |}Nfib (n - 1) +  Nfib (n - 2) + 1
```

A fenti példában a `P` annotáció hatására egy új kiértékelő folyamat indul el, azaz `Nfib (n - 1)` kiértékelése az `Nfib (n - 2)` kiértékelésével párhuzamosan történik.

Az annotációk segítségével kiértékelési módszereket (stratégiákat) fogalmazhatunk meg magasabbrendű függvények segítségével [Tri 98, Hor 00]. Két elemi kiértékelési módszer, a `'seq'` (a szekvenciális kiértékelési módszer) és a `'par'` (a párhuzamos kiértékelési módszer) segítségével összetett kiértékelési módszereket építhetünk fel. Például az összetett `parlist` módszer segítségével párhuzamos `map` függvényt írhatunk, amely az eredménylista elemeit egymással párhuzamosan határozza meg. Az egyes elemeket pedig a `parlist` paramétereként átvett `s` kiértékelési módszer szerint számoljuk ki.

(Clean és Haskell): Összetett kiértékelési módszer

```
parmap :: (Strategy b) (a -> b) [a] -> [b]
parmap s f x = map f x 'using' parlist s
```

Egy másik párhuzamos programozási lehetőség a Concurrent Cleanben a csatornák használata. Az üzenetküldés kétféle típusát különböztetjük meg: két különálló program közötti kommunikációt, illetve egy programon belüli párhuzamos szálak üzenetváltását [Ser 99].

[31] Az összefésüléses szemantika határozza meg a párhuzamos kiértékelés eredményét.

Az első esetben két absztrakt csatorna adattípust használunk: az `(SChannel Int)` és az `(RChannel Int)` típust, amelyek `Int` alaptípusú értékek egyirányú továbbítására, küldésére, illetve fogadására alkalmas csatornák típusát jelölik. Az üzeneteket a fogadó csatorna listában tárolja, érkezési sorrendben. Csatornát a `createRChannel`-lel hozunk létre[32], a `findSChannel`-lel, illetve `lookupSChannel`-lel pedig megkeressük a már létrehozott csatornát, neve alapján, a másik oldalon. A programok egymással a `send`, a `receive` és az `available` könyvtári függvények kiértékelésével kommunikálnak. Az alábbi program a termelő–fogyasztó feladat egy megoldása csatornák segítségével. A `produce` függvény rekurzív módon 10 adatot állít elő, a `consume` függvény pedig ugyancsak 10 adatot fogad.

(Clean): Csatornák programok között

```
// 1. a termelő program.
Start :: *World -> *World
Start w
   # (sc, w) = findSChannel "Consumer" w
   = produce sc 10 1 w

produce :: (SChannel Int) Int Int *World -> *World
produce sc i n w
   | i == 10 = w
   | otherwise = produce sc (i - 1) (n + 1) (send sc n w)

// 2. a fogyasztó program.
Start :: *World -> (Int, *World)
Start w
   # (maybe_rc, w) = createRChannel "Consumer" w
   = consumer maybe_rc w
where
   consumer Nothing w = abort "already exists"
   consumer (Just rc) w = consume rc 10 0 w

consume :: (RChannel Int) Int Int *World -> (Int, *World)
consume rc i r w
   # (n, rc, w) = receive rc w
   | i == 0  = (r + n, w)
   | otherwise = consume rc (i - 1) (r + n) w
```

A második esetben lusta kiértékelésű szálakat hozhatunk létre a `newThread` függvénnyel ugyanazon a processzoron. A szálak közötti üzenetváltást ugyancsak a `send`, a `receive` és az `available` függvényekkel valósíthatjuk meg.

Példaprogramként ismét a termelő–fogyasztó feladat egy megoldását mutatjuk be, ez alkalommal a termelő és a fogyasztó két különálló szálnak felel meg. A két szál törzse ugyan-

[32] Amennyiben csak helyi használatra szeretnénk egy csatornát létrehozni, akkor azt a `newChannel`-lel hozzuk létre.

azon programhoz tartozik, de egy szálat elindíthatunk egy másik processzoron is. Ezt a `newThreadAt pid` segítségével tehetjük meg:

(Clean): Csatornák szálak között

```
module ProdCons
import StdEnv, StdParallel, StdThread, StdChannel

Start w = startProcessorsW' myStart w

// Létrehoz egy csatornát a pid azonosítóval rendelkező processzoron.
// A csatorna a két külön szálon futó produce, illetve a consume függvényeket köti össze.
myStart :: (Set ProcId) *World -> (Int, *World)
myStart pids w
  # (sc, rc, w) = newChannel w
    w          = newThreadAt pid (produce sc 10 1) w
  = newThread' (consume rc 10 0) w
where
  pid = pickFromSet pids 0

// A termelő (producer) függvény.
produce :: (SChannel Int) Int Int *env -> *env | ThreadEnv env
produce sc i n env
  | i == 0  = env
  | otherwise = produce sc (i - 1) (n + 2) (send' n sc env)

// A fogyasztó (consumer) függvény.
consume :: (RChannel Int) Int Int *env -> (Int, *env) | ThreadEnv env
consume rc i r env
  # (n, rc, env) = receive' rc env
  | i == 1  = (r + n, env)
  | otherwise = consume rc (i - 1) (r + n) env
```

Csatornákat használ a TCP/IP protokoll alapú párhuzamos funkcionális programozás is [Ach 00, Tan 99]. A hálózat bármely gépét IP címe szerint azonosíthatjuk, majd a géphez rendelt csatornákon üzeneteket küldhetünk a vele összekötött gépeknek. Miután létrehoztunk egy kliens-szerver kapcsolatot két gép között, ezek a csatornákon keresztül a következőképpen küldhetnek egymásnak üzeneteket:

(Clean): TCP alapú kommunikáció

```
// A kliens elküldi a "hello server" üzenetet a TCP_SChannel-len keresztül.
clientSend :: TCP_SChannel *World -> (TCP_SChannel, *World)
clientSend sChannel world=send (toByteSeq "hello server")
                           sChannel world
```

```
// A szerver TCP_SChannel-len fogadja az üzenetet.
serverReceive ::
  String TCP_RChannel *World -> (TCP_RChannel, *World)
serverReceive expectedMessage rChannel world
  # (message, rChannel, world)    = receive rChannel world
  | toString message <> expectedMessage = abort "wrong message"
  | otherwise                     = (rChannel, world)
```

A Clean nyelv szigorúan statikus típusrendszerét egészíti ki a `Dynamic` típus [Pil 99, Pla 01]. A `dynamic` függvény segítségével tetszőleges típusú[33] kifejezést átalakíthatunk `Dynamic` típusúvá, illetve később a `Dynamic` értéket a tartalmazott érték *típusára való mintaillesztéssel* visszakaphatjuk. A `sendDynamic` és `receiveDynamic` függvényekkel lehetőségünk van arra is, hogy Clean programok konstansokat és mobil kódrészleteket küldjenek és fogadjanak [Pla 01, Hor 02a]. A `writeDynamic` és `readDynamic` függvények alkalmasak arra, hogy fájlba írjunk ki, fájlból olvassunk be `Dynamic` típusú kifejezéseket. Az alábbiakban egy példán azt mutatjuk be, hogy egy Clean program hogyan épít fel egy fát, amelyet fájlba ment; egy másik program elkészít egy olyan függvényt, amely megszámolja egy fa leveleit és a függvényt kiírja egy fájlba, végül egy harmadik program beolvassa a fát a fájlból és alkalmazza rá az ugyancsak fájlból beolvasott függvényt, majd kiírja az eredményt.

Az első program létrehozza a `tree2` fát, becsomagolja a `dynamic` függvénnyel egy `Dynamic` típusú értékbe, végül kiírja egy fájlba a `writeDynamic`-kal.

(Clean): `Dynamic` típusú érték készítése

```
module v
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
  #! (ok, world) = writeDynamic (p +++ "value")
                   DynamicDefaultOptions dt world
  | not ok = abort "could not write dynamic"
  = (dt, world)
  where  dt = dynamic (Node 99 tree2 tree2)
       tree2 = (Node 2 (Node 1 Leaf Leaf) Leaf)
          p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
module f
import StdDynamic, StdEnv
:: Tree b = Node b (Tree b) (Tree b) | Leaf
Start world
  #! (ok,world) = writeDynamic (p +++ "function")
                   DynamicDefaultOptions dt world
  | not ok = abort "could not write dynamic"
  = (dt,world)
  where  dt = dynamic count_leafs
         p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
```

[33] Minden olyan típusérték `Dynamic` típusúvá alakítható, amely a `TC` típusosztályba tartozik, azaz létezik típuskódja.

Az `apply` alkalmazza a fájlból beolvasott függvényt az ugyancsak fájlból beolvasott fára, majd kiírja az eredményt. A beolvasás során *típusokra vonatkozó mintaillesztéssel* ellenőrizzük, hogy a beolvasott függvény és a beolvasott érték típusa megegyezik-e az elvárt típusokkal.

(Clean): `Dynamic` felhasználása

```
module apply
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
   # (ok,f,world)  = readDynamic (p +++ "function") world
   | not ok            = abort " could not read"
   # (ok,v,world) = readDynamic (p +++ "value") world
   | not ok            = abort " could not read"
   # result = apply f v;
   = (result, world);
where
     apply (f :: (Tree Int) -> Real) (v :: (Tree Int)) = f v
     apply _ _ = abort "unmatched"
     p = "C:\\Clean 2.0\\Examples\\Dynamic\\"
```

16.9.2. Distributed Haskell with Ports

A Distributed Haskell könyvtár [Huc 01] a Haskell funkcionális programozási nyelv egy kiterjesztése, amelynek használatával osztott Haskell programokat írhatunk. A kiterjesztés lehetőséget ad arra, hogy a programok között egyirányú, pont–pont kapcsolatot létesítsünk kapuk (portok) segítségével. Nézzük meg, milyen alapvető nyelvi elemek állnak ehhez rendelkezésünkre. A könnyebb megértés céljából vegyünk egy egyszerű példaalkalmazást, melyben egy szerver és két (azonos kódú) kliens programunk van. A kliensek egy paraméterként megadott számot küldenek el a szervernek, a szerver visszaküldi a klienseknek az elküldött két szám összegét, azok pedig kiírják az összeget a képernyőre.

Ahhoz, hogy a kliensek üzenetet tudjanak küldeni a szervernek, először a szerveroldalon létre kell hoznunk egy kaput. Ezt a `newport` függvénnyel tehetjük meg, amellyel bármilyen olyan típushoz, amely a Haskell `Readable` és `Showable` osztályába is beletartozik létrehozhatunk kaput, amelyen az adott típusú adat küldhető. A kaput csak az a program (és ha esetleg a programon belül több szál van, akkor csak az a szál) olvashatja, amely létrehozta azt. A kliensekkel valamilyen módon tudatnunk kell, hogy melyik az a kapu, amelyre a szervernek üzenetet küldhetnek. Ehhez a szerveroldalon a `registerPort` függvénnyel be kell jegyeznünk a szerverkaput valamilyen (a kliensek által ismert) néven.

(Haskell): Port létrehozása és bejegyzése

```
serverPort <- newPort
registerPort serverPort "ServerPort"
```

Bejegyzés után a kliensoldalon a `lookupPort` függvénnyel lekérdezhetjük az adott névhez tartozó kaput. A néven kívül szükségünk van a szervert futtató gép IP címére is (a példánkban a szerveroldali IP címet a kliensek parancssori argumentumként kapják meg). Ezek után a klienseknek már elég információ áll a rendelkezésükre ahhoz, hogy elküldjék az argumentumként kapott számot a szervernek. Mivel az üzenetküldés csak egyirányú, a klienseknél is létre kell hoznunk egy-egy kaput, amelyre a szerver elküldheti az eredményül kapott értéket. Ugyanakkor tudatnunk kell a szerverrel, hogy melyek ezek a kapuk. Erre a legegyszerűbb módszer, ha a kliensek a paraméterként kapott számokkal együtt az általuk létrehozott kapuk leírását is elküldik a szervernek. A kapukra való adatküldésre a `writePort` függvény szolgál. A függvénynek a küldeni kívánt adatot, illetve azt a kaput kell megadnunk paraméterül, amelyre az adatot küldeni szeretnénk.

(Haskell): Adatok elküldése a szerver számára

```
[host,numstr] <- getArgs
serverPort <- lookupPort host "ServerPort"
inPort <- newPort
writePort serverPort (num,inPort)
```

A szerver az előzőekben leírt kapubejegyzés után a `readPort` függvény meghívásával olvasni próbál egy számból és egy kapuleírásból álló adatpárt a szerver kapuról. A `readPort` függvény akkor ad vissza adatot, ha valaki küld valamit a `writePort` függvénnyel az adott portra, amíg ez meg nem történik, a program kiértékelése felfüggesztődik a függvényhívásnál. Miután a szerver sikeresen olvassa az egyik kliens által küldött értékeket, még egyszer meghívja ugyanazt a függvényt, hogy a másik kliens adatait is megkapja.

(Haskell): A kliensoldali adatok fogadása

```
(num1,outPort1) <- readPort serverPort
(num2,outPort2) <- readPort serverPort
```

Ha ez is sikeresen megtörténik, akkor a már ismert `writePort` függvénnyel elküldi a kapott számok összegét a számokkal egyidejűleg kapott kapukra.

(Haskell): Az összeg visszaküldése

```
writePort outPort1 (num1 + num2)
writePort outPort2 (num1 + num2)
```

A kliensek a fentiekben leírt adatküldés után megpróbálnak egy `Int` típusú értéket olvasni a `readPort` függvénnyel az általuk létrehozott kapuról, majd miután ez sikerül, kiírják a kapott értéket a képernyőre.

(Haskell): Az eredmény fogadása és kiírása

```
sum <- readPort inPort
putStrLn (show sum)
```

Ezek után nézzük meg a teljes programokat:

(Haskell): Szerver

```
import IO
import Port
import System

main = do
     serverPort <- newPort
     registerPort serverPort "ServerPort"
     (num1,outPort1) <- readPort serverPort
     (num2,outPort2) <- readPort serverPort
     writePort outPort1 (num1 + num2)
     writePort outPort2 (num1 + num2)
```

(Haskell): Kliens

```
import IO
import Port
import System

main = do
     [host,numstr] <- getArgs
     serverPort <- lookupPort host "ServerPort"
     inPort <- newPort
     client inPort serverPort (read numstr)

client :: Port Int -> Port (Int,(Port Int)) -> Int -> IO ()
client inPort serverPort num = do
     writePort serverPort (num,inPort)
     sum <- readPort inPort
     putStrLn (show sum)
```

16.9.3. A JoCaml párhuzamos és osztott nyelvi elemei

A JoCaml [Fou 01] nem tisztán funkcionális nyelv, imperatív és objektumorientált elemeket is tartalmaz. A JoCaml nyelvben a mobil kód megfelelője az ágens. A nyelv tervezésének három fő szempontja volt: eszközök biztosítása ágensek egyszerű létrehozására, összetett elosztott számítások kifejezhetősége, az absztrakt nyelvi eszközök jelentésének pontos leírhatósága. A legfontosabb két absztrakt nyelvi elem a csatorna és az absztrakt hely fogalma. Szinkron és aszinkron csatornát a `let def` utasítás segítségével hozhatunk létre, aszinkron csatorna azonosítóját egy `!` követi. Aszinkron csatornán a csatornához rendelt folyamatok segítségével kezelhetünk adatokat.

```
let def echo! X = print_int x;
val echo : <<int>>
```

A spawn az aszinkron csatornához rendelt folyamatpéldányból kifejezést állít elő, melynek kiértékelésekor egy konkurens folyamat indul el. Az üzenetküldést az alábbi példán

szemléltetjük, mely párhuzamos folyamatokhoz csatolt csatornák értékét írja ki, azaz az 1, 2, 3 egész számokat előre nem ismert sorrendben:

```
spawn{
   let x = 1 in
   {let y = x + 1 in echo y | echo (y + 1)} | echo x }
```

A konkurens programozás összetett vezérlési és adatkezelési struktúráit szinkronizációs mintákkal írhatjuk le. Csatornahalmazokhoz tartozó kommunikációs események között fennálló konfliktust, illetve szinkronizációt fejezhetünk ki velük. Szinkronizációs minták mindig egyetlen `let def` kifejezésben egyidejűleg definiált csatornákra vonatkoznak. Mintákat a | párhuzamos kompozíció művelettel írhatunk le, csatornarészhalmazokra vonatkozó, több alternatív mintát pedig az `or` művelettel kapcsolhatunk össze . Konkurens környezetben használt számlálóra példa a `count` aszinkron csatorna, amelyre küldött üzeneteket vagy az `inc`, vagy a `get` szinkron csatornákra érkező üzenetekkel szinkronizálunk. Egy kezdeti `count 0` üzenet után az így kapott konkurens rendszer az `inc` üzenet hatására növeli a számlálót, önmagának küldve újabb `count` üzenetet és abban tárolva a számláló aktuális értékét. A `get` üzenet hatására visszaadja a számláló értékét, egyben önmagának egy `count` üzenet küld, ezzel biztosítva a további helyes működést.

```
let def count! n | inc () = count (n + 1) | reply to inc
     or count! n | get () = count n | reply n to get
spawn {count 0}
```

A JoCaml-ban elosztott programokat is írhatunk. A folyamatok vándorolhatnak egyik gépről a másikra. Kezdetben a különböző gépeken futó folyamatok között nincs kapcsolat, a partnerek névszolgáltató segítségével találnak egymásra.

Az absztrakt hely fogalma magában foglalja az ágens kódjának és aktuális fizikai helyének együttesét. Ily módon a kód mozgása során a JoCaml nyelv absztrakt eszközöket biztosít a kompozíció, a kommunikáció és a belső állapot helyfüggetlen kezelésére és távoli szolgáltatások igénybevételére. Az ágensek összetevői együtt mozognak az őket tartalmazó szülő ágenssel, a kommunikációs csatorna pedig fennmarad a helyüket időközben változtató ágensek között is, és az ágensek abból az állapotból folytatják működésüket, ahol a helyváltoztatás előtt voltak.

A következő példában az egyik folyamat bejegyzi az `f` közös erőforrást `square` néven, a másik pedig megkeresi és felhasználja:

```
spawn {let def f x=reply x*x in Ns.register "square" f vartype;}
spawn {let sqr=Ns.lookup "square" vartype in print_int(sqr 2); exit 0;}
```

Absztrakt helyeket a `let loc` ⟨*azonosító*⟩ `do {}` konstrukcióval hozhatunk létre. Egy fizikai helyet mint egy absztrakt hely aktuális megfelelőjét, azaz mint az ágens aktuális fizikai helyét azonosíthatjuk. Erre használjuk majd a `here` üres ágenst:

```
let loc here do {
  Ns.register "here" here vartype;
  Join.server ()
}
```

A `mobile` ágens lekérdezi a névszolgáltatótól a `here` ágens adatait, majd átvándorol a `here` aktuális fizikai helyére és ennek mindenkori helyén, alágenseként végzi el a számításokat.

```
let loc mobile
 do {
    let here = Ns.loopkup "here" vartype in
    go here;
    let sqr = Ns.lookup "square" vartype in
    let def sum (s,n)=reply (if n=0 then s else sum(s+sqr n, n-1)) in
    let result = sum (0, 5) in
    print_string ("q: sum 5= " ^ string_of_int result ^ "\n");
     flush stdout;
 }
```

Letölthető ágenseket is definiálhatunk oly módon, hogy az ágensnek paraméterként adjuk meg, hogy híváskor hova vándoroljon, illetve adatvezérelt vándorlást is megvalósíthatunk.

Feladatok

1. Adjuk meg egy szám prímosztóit!
2. Döntsük el egy számról, hogy tökéletes szám-e!
3. Rendezzük nagyság szerint növekvő sorrendbe egy sorozat elemeit!
4. Keressük meg egy sorozat legkisebb elemét! A rendezési reláció legyen a függvény argumentuma!
5. Határozzuk meg egy sorozat elemeinek átlagát!
6. Döntsük el, hogy egy karaktersorozat előfordul-e egy másikban!
7. Készítsünk absztrakt, algebrai zsák típust!
8. Írjunk interaktív funkcionális programot! A képernyő tetején jelenjen meg egy menüsor, annak egyik legördülő menüjéből nyissunk meg egy párbeszédablakot. A párbeszédablakban legyen két szerkeszthető beviteli mező, egy válaszsor és egy gomb. A beviteli mezőben olvassunk be két egész számot és a gomb megnyomására összegüket jelenítsük meg a válaszsorban.

17 Logikai programozás

Ha adott egy rendszer logikai formulákkal definiált modellje, kiegészítendő vagy eldöntendő kérdéseket fogalmazhatunk meg bizonyítandó állítások formájában. Kiegészítendő kérdés esetén a megfelelő tulajdonságú objektumok, tervek stb. meghatározása számítási folyamatnak tekinthető, ez a logikai programozás (LP) alapgondolata.

Egy logikai program tehát ugyanúgy, mint egy automatikus tételbizonyító rendszer, axiómák halmaza, amelyhez egy következtető gép is tartozik. Attól elsősorban a következtető gép egyszerűsége, a számítási folyamat átláthatósága, irányíthatósága, garantálható terminálása és nagyobb hatékonysága különbözteti meg.

Ebben a fejezetben áttekintjük a logikai programozás kialakulását, alapfogalmait, mindmáig legfontosabb megvalósítását, a Prolog nyelvet, a Prolog programozás legalapvetőbb módszereit, a logikai programozás irodalmát, néhány kiterjesztését és újabb irányzatokat. Különös hangsúlyt fektetünk a Prolog programozás gyakorlati módszereire, helyes és hatékony programok készítésére.

A logikai programozás (LP) gyökerei Hilbertig nyúlnak vissza, aki a naiv halmazelmélet ellentmondásossága miatt hirdette meg programját a matematika axiomatizálására és *mechanikus tételbizonyítási módszerek* kidolgozására.

A rezolúciós algoritmus [Rob 65] megszületése ezen kutatás egyik mérföldköve. Az algoritmusból kinyerhető válaszok ugyanis a rezolúciót konstruktív bizonyítási eszközzé teszik, és így a bizonyítás adott tulajdonságú objektumok megkeresésének, vagy kiszámításának tekinthető[1].

A bizonyítás vagy számítás algoritmusa a rezolúcióban és általában mindenféle mechanikus tételbizonyító rendszerben a feladat logikai (deklaratív) leírásából és a rendszer vezérlési módjából áll:

Algoritmus = Logika + Vezérlés.

Néhány sikertelen amerikai kísérlet után, a hetvenes évek elején Robert Kowalski felismerte, hogy erre az algoritmusfogalomra általános célú programozási nyelv is alapozható lenne. Alain Colmerauer és kollégái tervezték meg az első elfogadható hatékonyságú LP nyelv, a *Prolog* részleteit és ugyanők készítették el hozzá Marseille-ben, 1972-ben az első értelmezőprogramot (interpretert) is. Gyakorlati céljuk egy adatbáziskezelő természetes nyelvű felületének megalkotása volt, és ez is lett a Prolog nyelv első alkalmazása[2].

Az első hatékony Prolog-fordítót David H. D. Warren és munkatársai Edinboroughban a hetvenes évek második felében fejlesztették ki. Teljesítménye alapján összemérhető volt a kor legjobb Lisp rendszereivel. Megteremtette a Prolog *de facto* szabványát, meghatározta további fejlődésének irányát és közvetve a nyelvnek a kilencvenes évek közepére kidolgozott *ISO* szabványát is. Sajnos, ez még nem terjed ki a nyelv minden részletére (például a modulrendszerre). Ilyen esetekben a manapság talán legelterjedtebb változatot, a svéd SICStus Prolog rendszert vesszük alapul [Spl 02], mivel ez a megvalósítás eddig is iránymutatónak bizonyult a szabvány kialakítása szempontjából.

[1] Az elsőrendű predikátumkalkulus és a rezolúciós algoritmus alapszintű ismeretét a továbbiakban feltételezzük [Fek 90, Nil 82, Kow 79]. A tárgyalás fő vonala azonban enélkül is érthető lesz.

[2] A Prolog név a *Programmation en Logique* (francia) kifejezésből származik.

A Prolog a mai napig messze a legszélesebb körben használatos LP nyelv. Viszonylagos sikerének titka abban rejlik, hogy tervezői jó gyakorlati érzékkel kötötték meg azokat a kompromisszumokat, amelyeket elméleti oldalról gyakran és hevesen bírálnak, de amelyek lehetővé teszik – megfelelő programozási stílus mellett – a gyors programfejlesztést kevés hibával, hatékony kódot állítva elő. Valóban, talán nincs még egy olyan programozási nyelv, ahol a jó stílus ennyire alapvető lenne.

Ebben a fejezetben inkább a Prolog nyelv *alapvető* sajátosságainak és programozási módszereinek bemutatására törekedtünk – amennyire ez az adott keretek között lehetséges volt –, mint a témakör általános áttekintésére, mert ezt a témában járatlan olvasó szempontjából előnyösebbnek ítéltük. Reményeink szerint a fejezet végén elegendő információ található az ismeretek kiszélesítéséhez szükséges irodalom felkutatásához is.

Helyszűkében nem ismertethettünk hosszabb példaprogramokat. Mellőznünk kellett a haladó programozási módszerek ismertetését, úgymint az indeterminisztikus programozást, a részleges adatstruktúrák használatát, a metaprogramozást, a logikai nyelvtanokat stb. Tudatosan kihagytuk a *vágó* (cut) utasítás tárgyalását, amely – tapasztalataink szerint – kezdő kézben több kárt mint hasznot hoz. Ehelyett inkább a strukturált programozási stílust és a programok helyességét, hatékonyságát kevésbé veszélyeztető *feltételes célokat* választottuk, bár ezek túlhasználására is nagy lehet a kísértés a hagyományos programozási stílusokhoz szokott programozók esetében.

17.1. Logikai programok

A logikai program egy modellre vonatkozó *állítások* (axiómák) egy halmaza. Az állítások a modell objektumainak tulajdonságait és kapcsolatait, szaknyelven *relációit* írják le. Ha például adottak állítások, amelyek arra vonatkoznak, hogy ki kinek az apja, ezek együtt az *apja* nevű, *kettő* aritású relációt írják le. Ha megmondjuk, hogy kik a férfiak, akkor az erre vonatkozó állítások együtt a *férfi* nevű, *egy* aritású relációt írják le. (Lásd a 17.1.1. paragrafust.)

Az állítások egy adott relációt meghatározó részhalmazát *predikátumnak* nevezzük. A program futtatása minden esetben egy az állításokból következő tétel konstruktív bizonyítása, azaz – a logikai programozásban szokásos szóhasználattal – a programnak feltett *kérdés* vagy más néven *cél* megválaszolása. Ennek során a predikátumok eljárásokként működnek.

A ma használatos LP nyelvekben minden, a predikátumokat alkotó *állítás tény* vagy *szabály* lehet. Az állításokat és a kérdéseket (azaz célokat) együtt *mondatoknak* nevezzük (a kérdések ugyan nem a program részei). Az egyes LP nyelvekben a program mondatai még a programban leírt relációkra vonatkozó *deklarációk* és a program betöltésekor végrehajtandó *direktívák* is lehetnek. A deklarációk a predikátumok, mint eljárások futtatásának módját pontosítják, míg a direktívák magát a futtatási környezetet állítják be. A Prologban a direktívák és a deklarációk is *:-cél.* alakúak. Minden mondatot pont, és legalább egy elválasztó karakter (újsor, helyköz, vagy tabulátor) zár le. A logikai programok *állításait* a szakirodalomban *definit klózoknak* vagy *Horn klózoknak* is nevezik.

17.1.1. Tények

A legegyszerűbb logikai programok csak tényeket tartalmaznak[3]. A tények formailag atomi formulák, például:

```
apja('Ábrahám','Izsák').     apja('Ábrahám','Ismáel').
apja('Ábrahám','Ismeretlen').
apja('Izsák','Jákób').       apja('Izsák','Ézsaú').

anyja('Sára','Izsák').       anyja('Hágár','Ismáel').
anyja('Rebeka','Jákób').     anyja('Rebeka','Ézsaú').

férfi('Ábrahám').  férfi('Izsák').  férfi('Ismáel').
férfi('Jákób').    férfi('Ézsaú').

nő('Sára').        nő('Hágár').     nő('Rebeka').
nő('Ismeretlen').
```

Ezek a program objektumaira vonatkozó egyszerű tulajdonságokat, illetve az objektumok között egyszerű kapcsolatokat fejeznek ki. A legegyszerűbb kérdésekre példák:

```
| ?- apja('Ábrahám','Izsák').
yes
| ?- anyja('Sára','Jákób').
no
```

A `| ?-` a Prolog rendszer (és fejlesztői környezet) parancskérő jele. Ide gépeljük be – a mindig ponttal és `<Enter>`-rel lezárt – utasításainkat (például programbetöltés), és a betöltött programok predikátumaira (eljárásaira) vonatkozó kérdéseinket. A `yes`, illetve a `no` a Prolog rendszer válaszai aszerint, hogy sikerült-e bizonyítania az állítást. Itt a bizonyítás persze csak egylépéses, hiszen vagy megtaláljuk a kérdésnek megfelelő tényt, vagy nem. (Ne keverjük össze az *utasításokat* – amelyek formailag Prolog célok – a *direktívákkal*, amelyek formailag – de nem tartalmilag – a deklarációkkal egyeznek meg (17.1.).)

Ha azt kérdezzük, hogy: *Van-e olyan X, akinek Izsák az apja?*, akkor a fenti programnak megfelelően két megoldást kaphatunk (ahol `X` a kérdés ismeretlenje, konkrétan egy egzisztenciálisan kvantált változó):

```
| ?- apja('Izsák',X).
X = 'Jákób' ? ;    X = 'Ézsaú' ? ;
no
```

A fenti eredmények az `apja('Izsák',X)` cél és a megfelelő tények egyesítéséből származnak. A kérdőjelek arra vonatkoznak, hogy kérünk-e újabb megoldást. A `;` részünkről azt

[3] A példaprogramokat – itt, az általános részben és később is – SICStus Prolog 3.9 nyelven írtuk le. A példákban a változóneveket nagybetűvel vagy aláhúzásjellel kezdődő azonosítókkal, a neveket pedig kisbetűvel kezdődő azonosítókkal vagy aposztrófok közé zárt karakterláncokkal jelöltük.

jelenti, hogy igen. Ha nem kérünk több megoldást, ezt egy <Enter>-rel jelezhetjük, és ezt a Prolog fejlesztői környezet a yes válasszal nyugtázza. A fenti példában a no válasz azt mondja, hogy nincs több megoldás.

Az eddig tárgyalt elemi célok és-kapcsolataival *összetett célokat*, vagy más néven *célsorozatokat* képezhetünk. Az összetett célok elemi kérdéseit pedig *részcéloknak* is nevezzük. Például:

```
| ?- apja('Ábrahám',X), nő(X).
X = 'Ismeretlen' ? ;
no
```

Ilyenkor a célsorozat megoldásai a részcélok közös megoldásai. Úgy is fogalmazhatunk, hogy a később megoldott részcél a korábban megoldott részcél megoldásaira további megszorítást (és így szűrőt is) tartalmaz, már amennyiben van közös változójuk.

A célok változói mindig *egzisztenciálisan* kvantáltak. A program futtatása a kérdés konstruktív megválaszolása, azaz változói lehetséges értékeinek meghatározása.

17.1.2. Szabályok

Az egynél hosszabb célsorozatok valójában új relációkat határoznak meg. A fenti `apja('Ábrahám',X), nő(X)` célsorozat például a bibliai Ábrahám lányaira vonatkozik. Az új relációnak formálisan is nevet adva a logikai programokat alkotó állítások másik osztályának egy példányához, egy *szabályhoz* juthatunk:

```
'Ábrahám lánya'(X) :- apja('Ábrahám',X), nő(X).
```

Ennek olvasata: `X` *Ábrahám lánya,* **ha** *Ábrahám* `X` *apja* **és** `X` *nő.* A szabályokban szereplő változók, mint a fenti példából is látható, *univerzálisan* kvantáltak. A szabályok általános alakja:

$$A \text{ :- } B_1, B_2, \ldots, B_n \qquad \text{ahol } n > 0, \quad A, B_1, \ldots, B_n \text{ atomi formulák.}$$

Egy szabály következmény része (fent A) a szabály *feje*, feltétel része (fent $B_1, \ldots, B_n$) pedig a szabály *törzse*. A tényeknek csak fejük van. Úgy is tekinthetjük, hogy törzsük üres, vagy logikailag igaz (`true`).

A tények is tartalmazhatnak (univerzálisan kvantált) változókat. Azt például, hogy mindenki szereti Sárát, így is kifejezhetjük:

```
szereti(_Bárki,'Sára').
```

Így nem kell az univerzum minden elemére felsorolnunk a vonatkozó állítást. Van azonban egy lényeges különbség. Ha most megkérdezzük, hogy ki szereti Sárát:

```
| ?- szereti(Ki,'Sára').
true ? ;
no
```

A `true` válasz azt jelenti, hogy a megtalált megoldás nem példányosította a `Ki` változót. Ezt úgy értelmezhetjük, hogy a válaszban nem helyettesített változó tetszőlegesen példányosítható. Az már természetes, hogy az általános megoldás mellett további megoldást nem találtunk.

A relációkat és az ezeket definiáló *predikátumokat* nevükkel és aritásukkal jellemezhetjük, *név/aritás* formában. (Az *aritás* egy predikátum argumentumainak (paraméterhelyeinek) száma. Az *argumentum* az a szövegpozíció, ahová a paramétert írjuk.)

Egy relációt gyakran nem egyetlen állítás határoz meg. Ilyenkor a reláció az állítások által definiált relációk uniója. Minden változó hatóköre pontosan az őt tartalmazó mondat. Például a `szülője/2` reláció az `anyja/2` és az `apja/2` relációk uniója:

```
szülője(X,Y) :- anyja(X,Y).
szülője(X,Y) :- apja(X,Y).

fia(X,Y) :- szülője(Y,X), férfi(X).

lánya(X,Y) :- szülője(Y,X), nő(X).

nagyszülője(X,Y) :- szülője(X,Z), szülője(Z,Y).
```

A `fia/2` reláció viszont a `szülője/2` és a `férfi/1` relációk metszete, és hasonlóan adódik a `lánya/2` reláció is. Az utolsó szabály eltér az előzőektől abban, hogy a jobb oldalán új változó is előfordul. Így kétféle olvasata is van:

- Minden `X,Y,Z`-re `nagyszülője(X,Y)`, *ha* `szülője(X,Z)` *és* `szülője(Z,Y)`.
- Minden `X,Y`-ra `nagyszülője(X,Y)`, *ha* van olyan `Z`, hogy `szülője(X,Z)` *és* `szülője(Z,Y)`.

A szabályok fenti olvasatait mindegyik esetben *deklaratív* olvasatnak nevezzük. Ezzel szemben áll a *procedurális* olvasat, ami a logikai programok futtatásához, vagyis a konstruktív bizonyítási eljáráshoz kapcsolódik. Minden esetben felteszünk egy kérdést. Ez a bizonyítandó cél(sorozat). Ezután a részcélok bizonyítása, és így kiejtése a feladat. A bizonyítást akkor fejeztük be sikeresen, ha az összes részcélt kiejtettük, azaz bizonyítottuk, és így csak az *üres célsorozat maradt.*

A bizonyítás történhet például *felülről lefelé*, vagy *alulról felfelé*, de egyéb bizonyítási módok is találhatók például [Kow 79]-ben.

1. Az *alulról felfelé* való bizonyítás azt jelenti, hogy a szabályok feltételeivel sorban tényeket egyesítünk, sorban kiejtve ezeket. Így a szabályokból újabb tényeket kapunk, mígnem ezek sorban egyesíthetők lesznek a célban szereplő atomi formulákkal[4]. Amennyiben ezt elértük, a cél következik a programból. Ilyenkor azonban a bizonyítás nehezen irányítható a cél felé, ezért a gyakorlatban a *felülről lefelé* való, azaz a céltól a tények felé haladó következtetések általában jobban kezelhetőek. A Prologban is ezt valósították meg.

[4] Az egyesítéshez mindig a legáltalánosabb egyesítő helyettesítést alkalmazzuk.

2. A *felülről lefelé* való bizonyítás nagy vonalakban azt jelenti, hogy a cél *atomi formuláival* valamilyen *állításfejeket* (tényeket illetve szabályfejeket) egyesítünk[4]. (Az egyesítésben részt vevő mondatoknak a rezolúcióhoz hasonlóan itt is változóidegeneknek kell lenniük, azaz nem lehet közös változójuk. Ezt az egyesítésben résztvevő állítás, azaz tény vagy szabály változóinak átnevezésével biztosíthatjuk.) Tény egyesítésével kiejtjük a megfelelő részcélt a cél(sorozat)ból, szabályfej egyesítésével pedig a szabály törzsével helyettesítjük azt, az egyesítő helyettesítést mindkét esetben a kapott célsorozatra alkalmazva: Ez egy *célredukciós lépés*.
Ha ilyen célredukciós lépések sorozatával az összes részcélt kiejtettük, akkor bebizonyítottuk a célt. A bizonyítás közben végzett változó-helyettesítéseknek az eredeti cél változóira vonatkozó eredményei adják a konstruktív bizonyítás által meghatározott, a feltételeknek eleget tevő objektumokat (termeket).
Tegyük fel például, hogy adott a $Q_1, Q_2, \ldots, Q_n$ célsorozat. Ha most adott az ettől változóidegen A tény, és $Q_1\theta = A\theta$, ahol a θ az A és a Q_1 atomi formulák legáltalánosabb egyesítő helyettesítése, akkor elég $(Q_2, \ldots, Q_n)\theta$ bizonyítása. Ez a bizonyítandó célra vonatkozó *célredukciós lépés* egyik esete. Ha pedig adott a $Q_1, Q_2, \ldots, Q_n$ célsorozattól változóidegen A :- $B_1, \ldots, B_m$ szabály, továbbá θ az A és Q_1 atomi formulák legáltalánosabb egyesítő helyettesítése, akkor elég $(B_1, \ldots, B_m, Q_2, \ldots, Q_n)\theta$ bizonyítása. Ez a bizonyítandó célra vonatkozó *célredukciós lépés* másik esete. Mindkét esetben, ha a célredukciós lépés eredménye a C célsorozat, amelyet a φ helyettesítéssel bizonyíthatunk, azaz $C\varphi$ igaznak bizonyul, akkor ezzel $(Q_1, Q_2, \ldots, Q_n)\theta\varphi$ is bizonyítást nyert, azaz a $\theta\varphi$ helyettesítés az eredeti kérdés megoldása.

Mindkét említett bizonyítási mód helyes és teljes, azaz pontosan azokat a célokat lehet velük bebizonyítani, amelyek a programból következnek.

A fentebb vázolt hatékonysági megfontolás és a Prolog nyelvhez való közelítés okán a továbbiakban a célokat mindig *felülről lefelé* fogjuk bizonyítani. Ehhez a bizonyítási módhoz kapcsolódik az állítások *procedurális* olvasata is:

- Az A tény azt jelenti, hogy az A részcél közvetlenül megoldható.
- Az A :- B_1, B_2, ... , B_n szabály azt jelenti, hogy az A részcél megoldható a B_1, B_2, ... , B_n célok megoldásával.

A fentiek szerint a `nagyszülője/2` predikátum egyetlen állításának procedurális olvasata a következő: *A* `nagyszülője(X,Y)` *cél megoldásához oldjuk meg a* `szülője(X,Z)` *és a* `szülője(Z,Y)` *célokat.*

Egy részcéllal esetleg több szabályfej illetve tény is egyesíthető. A lehetséges levezetések ezért faszerűen elágazhatnak. Az így létrejövő faszerkezet gyökere az eredeti cél, csúcsai a levezetés során előállított célsorozatok, élei a célredukciós lépések, és levelei az üres célsorozatok, a *megoldáslevelek*, illetve az olyan ágak, ahol a célredukcióra kiválasztott részcél nem redukálható, a *fail-levelek*. Minden megoldáslevélhez az eredeti kérdés egy-egy megoldása tartozik. Az így meghatározott fát *levezetési* vagy *bizonyítási fának*, illetve *keresési fának* vagy *keresési térnek* is nevezik.

Az alábbi példában a fa szintjeit a sorok behúzása jelzi. Minden lépésben a célsorozat első részcélját választjuk ki, és ezt próbáljuk ténnyel való egyesítés esetén kiejteni, illetve szabályfejjel való egyesítés esetén a szabálytörzzsel helyettesíteni. Ez a *legbal* részcélkiválasztási módszer. A keresési fában a célredukciós lépések során alkalmazott változóhelyettesítések közül mindig csak az aktuális célsorozat változóira vonatkozó helyettesítéseket jelöljük, mégpedig *változó<-helyettesítő_term* formában. Ha egy részcél és egy szabályfej vagy tény egyesítése során két változót kell egyesítenünk, az egyszerűség kedvéért mindig a részcél változóját helyettesítjük a szabályfej vagy tény változója helyébe.

Világos, hogy egyes lépésekben másik részcélt is választhatnánk a célredukcióhoz, és akkor az alábbitól különböző keresési fát kapnánk.

Kérdés, hogy nem kapnánk-e más megoldást. Szerencsére igaz az, hogy a különböző részcélkiválasztási módszerek ugyan különböző keresési fákhoz vezetnek, de a fák ugyanazokat a megoldásokat tartalmazzák. Sajnos a különböző keresési fák általában különböző méretűek, mint azt bárki maga is ellenőrizheti, ha az alábbi példában más részcélkiválasztási módszert választ.

```
?- nagyszülője('Ábrahám',X).
    szülője('Ábrahám',Z),szülője(Z,X).
      anyja('Ábrahám',Z),szülője(Z,X). % meghiúsul
      apja('Ábrahám',Z),szülője(Z,X).
       { Z <- 'Izsák' }
        szülője('Izsák',X).
          anyja('Izsák',X).        % meghiúsul
          apja('Izsák',X).
            { X <- 'Jákób' }       % 1. megoldás
            { X <- 'Ézsaú' }       % 2. megoldás
       { Z <- 'Ismáel' }
        szülője('Ismáel',X).
          anyja('Ismáel',X).       % meghiúsul
          apja('Ismáel',X).        % meghiúsul
       { Z <- 'Ismeretlen' }
        szülője('Ismeretlen',X).
          anyja('Ismeretlen',X).   % meghiúsul
          apja('Ismeretlen',X).    % meghiúsul
```

A fenti `nagyszülője('Ábrahám',X)` kérdésre a *legbal* részcélkiválasztó módszer a lehető legkisebb keresési fát állítja elő. Könnyen látható azonban, hogy például a `nagyszülője(X,'Jákób')` kérdésre egy olyan módszer szolgáltatná a legkisebb keresési fát, amely mindig a *legjobboldali* részcélt választaná. Tapasztalatainkat összegezve azt mondhatjuk, hogy mivel a `nagyszülője(X,Y)` típusú kérdésekre az első célredukciós lépés után mindig egy `szülője(X,Z), szülője(Z,Y)` típusú célsorozatot kapunk, a továbbiakban mindig azt a részcélt érdemes választani, amelyben kevesebb a szabad változó. (A szabad változó olyan változó, amelyet még nem helyettesítettünk állandó vagy összetett termmel.) Ez a módszer más célsorozatoknál is figyelemreméltó, mert a kevesebb szabad változót tar-

talmazó részcél általában kevésbé elágazó keresési fához vezet. Ez természetesen csak heurisztika.

Általában az is előfordulhat, hogy ugyanarra a kérdésre az egyik keresési fa véges, míg a másiknak vannak végtelen ágai. Ilyenkor a megoldás keresése elkalandozhat egy végtelen ágon, ami végtelen kereséshez vezethet. Ez azt jelenti, hogy részcélkiválasztási módszerünket úgy célszerű megválasztani, hogy a keresési fa lehetőleg véges legyen. Ha azonban – mint jelen esetben – nincsenek rekurzív szabályaink, a keresési fa véges.

17.1.3. Rekurzív szabályok

Tegyük fel, hogy azt szeretnénk leírni, X mikor őse Y-nak. Világos, hogy akkor őse, ha szülője, vagy valamelyik ősének a szülője. Látható, hogy a relációnak két esete van, és az egyik eset rekurzív. Ezt ennek megfelelően egy nem rekurzív és egy rekurzív szabállyal fejezhetjük ki:

```
őse(X,Y) :- szülője(X,Y).
őse(X,Y) :- szülője(X,Z), őse(Z,Y).
```

Ha most úgy képzeljük, hogy a `szülője/2` reláció egy irányított kört nem tartalmazó gráfot ír le, amelynek éleit éppen a `szülője/2` reláció adja meg, csúcsait pedig az élek végpontjai, akkor az `őse/2` reláció ezen a gráfon a nem üres, irányított utakat adja meg. Ezek az utak a *szülője* kapcsolat természeténél fogva körmentesek és felülről korlátos hosszúságúak.

Tekintsük most az `őse('Ábrahám',X)` kérdést. Könnyen ellenőrizhető, hogy a keresőfa előállításához célszerű a legbal részcélkiválasztási módszert alkalmazni, mert így a `szülője(X,Y)` célokban `X` már mindig ismert lesz, tehát a keresőfa nem lesz túl széles, azaz az elágazások száma az egyes csúcsokban nem lesz túl nagy, és a keresőfa véges is lesz, hiszen minden rekurzív hívásban egy, az `'Ábrahám'` pontból induló irányított út egy élét dogozzuk fel, és sem végtelen irányított utak, sem irányított körök nincsenek a gráfban.

Ha most például az `őse(X,'Izsák')` kérdést tekintjük, és a túl széles keresőfák ellen védekezésül az adódó célsorozatokból a jobb oldali részcélt választjuk, mivel abban a második paraméter már meghatározott, akkor végtelen keresési fát kapunk. Ha a célredukcióban következetesen a második szabályt részesítjük előnyben az elsővel szemben, rögtön a végtelen ágra kerülünk. Ha következetesen az első szabályt részesítjük előnyben, először megtaláljuk a megoldásokat, de utána, ha tovább keresünk, a végtelen ágra kerülünk. Mivel általánosságban egy számítás nem tudhatja, hogy mikor találja meg az utolsó megoldást, csak azt, hogy megoldást talált, illetve azt, hogy bejárta-e a keresési teret, ez nagyon kellemetlen tud lenni. Ha erre a kérdésre is a *legbal* részcélkiválasztási módszert alkalmazzuk, akkor az első `szülője(X,Y)` hívás lerögzít valahol a gráfon egy kezdőélet, és a további útkeresés az `'Izsák'` csúcshoz innét folytatódik. Ily módon a keresési fa szélesebb lesz, mint az előbb, de véges marad.

Megfogalmazhatnánk, mint heurisztikát, azt, hogy egy célsorozatból mindig a nem rekurzív predikátummal megoldható részcélt válasszuk. Ez gyakran segítene, de ilyenkor egyrészt felmerülhet, hogy több heurisztika esetén hogyan rangsoroljunk, másrészt mi a hely-

zet, ha több rekurzív relációra is van részcél a célsorozatban, és mindegyik végtelen keresési fához vezethet? Például a fenti `őse/2` relációt a következőképp is átfogalmazhatnánk:

```
őseOO(X,Y) :- szülője(X,Y).
őseOO(X,Y) :- őseOO(X,Z), őseOO(Z,Y).
```

Ebből azt a tanulságot vonhatjuk le, hogy a várható kérdésekre jól illeszkedő formalizálás fontosabb, mint a túlfinomult, és ezért a programozó számára követhetetlen működésű részcélkiválasztó módszer. A módszer egyszerűsége hatékonysági szempontból is fontos lehet, és különösen előtérbe kerül, ha LP nyelvet szeretnénk tervezni.

17.2. Bevezetés a Prolog nyelvbe

A Prolog nyelv tervezői a fentiek miatt a *legbal* részcélkiválasztási módszer mellett döntöttek (lásd a 643. oldalt). A választható állításokat pedig a Prolog gép a felírás sorrendjében próbálja ki, egy visszalépéses kereséssel járva be a keresési fát, amelynek ágait természetesen nem építi fel előre, és visszalépéskor le is bontja, így tartva korlátok között a futó program memóriaigényét.

Ha már a Prolog gépet így határoztuk meg, maga a Prolog nyelv inkább tekinthető LP nyelvnek, mint egy tételbizonyító programhoz használt formulaleíró nyelvnek. Az ugyanis, hogy egy logikai formális nyelv melyik a kettő közül, a szerző véleménye szerint a hozzátartozó következtető gép kifinomultságán múlik. Ez a gép a Prolog esetében a lehető legegyszerűbb, és így működése könnyen követhető, a következtetés egyszerű eszközökkel irányítható, annak végessége könnyen biztosítható, hatékonysága előre kiszámítható. Ezek a feltételek pedig alapvetőek egy programozási nyelv esetében. Mint látni fogjuk, a Prolog részletes kidolgozása szerint is egy magas szintű programozási nyelv.

A Prolog programok helyességét mindig a megoldandó célokra vonatkoztatjuk. A parciális helyesség egyszerűen a formalizálás helyességét jelenti. A megállási probléma legegyszerűbb megoldása, ha bebizonyítjuk a kívánt típusú célsorozatokra a keresési fák végességét. Ha nincs rekurzív szabályunk, a közvetett rekurziót is ide értve, akkor a fa végessége automatikusan teljesül. Ha vannak rekurzív szabályok, akkor hasznos például, ha találunk egy nemnegatív egész értékű függvényt, ami a keresési fa csúcsain értelmezett, a fában lefelé haladva nem növekszik, és minden rekurzív hívásnál szigorúan csökken.

Az `őse/2` reláció első formalizálása esetén például a feldolgozatlan út (644. oldal) lehetséges legnagyobb hossza egy megfelelő függvény. Ez ugyanis tetszőleges `őse(X,Y)` típusú kérdés esetén a `szülője(X,Z)` típusú részcélok feldolgozásával mindig eggyel csökken. Mivel nem lehet negatív, a keresési fa véges. A második formalizálás (lásd az `őseOO/2` predikátumot a 645. oldalon) esetében végtelen a keresési fa, nem is találhatunk megfelelő függvényt.

A programozás rendesen a *fejlesztői környezetben* történik, egy belső vagy külső szövegszerkesztő felhasználásával. Az is lehet, hogy a Prolog rendszer (például SICStus) és a szövegszerkesztő (például Emacs) egy mindkettő által támogatott szabványos felületen keresztül

érintkezik, és a Prolog környezet a szövegszerkesztőn belül indul el, annak lehetőségeit (automatikus tördelés, színezés; fordítás, betöltés, forráskód szintű nyomkövetés stb.) kihasználva. A Prolog rendszer rendesen egy parancsablakban indul el (bár grafikus felülete is lehet). Ide gépeljük be célsorozatainkat, azaz programbetöltő, fordító és egyéb utasításainkat, valamint a betöltött programokra vonatkozó kérdéseinket. Alapértelmezés szerint (a grafikus, fájl és egyéb különleges felületektől eltekintve) itt kapjuk meg a rendszer válaszait is. Futtatható fájl készítése [Spl 02] esetén a programnak legalább egy, a betöltéskor végrehajtandó célt (direktívát (17.1.)) kell tartalmaznia; ez(ek) vezérli(k) működését. Egy program tetszőlegesen sok fájlból állhat. A fejlesztői környezetbe a programok többféleképpen is betölthetők. Az értelmező módban való betöltéshez egyszerűen megadjuk a Prolog parancskérő jelénél (`?-`) a betöltendő forrásállományok listáját (nevük rendesen `*.pl` alakú), például `?- [a,b,c].`, ha a program az `a.pl`, `b.pl` és `c.pl` állományokból áll. A betöltött program tetszőleges predikátuma tesztelhető, azaz feltehetünk rá vonatkozó kérdéseket.

Tegyük fel, hogy eddigi predikátumaink az `ny.pl` fájlban vannak:

```
| ?- [ny].
{consulting /home/at/pp/ny/ny.pl...}
{consulted /home/at/pp/ny/ny.pl in module user,
 70 msec 4864 bytes}
yes
| ?- őse(Ős,'Jákób').
Ős = 'Rebeka' ? ; Ős = 'Izsák' ? ;
Ős = 'Sára' ? ;   Ős = 'Ábrahám' ? ;
no
```

A betöltött program tetszőleges predikátuma *közvetlenül* tesztelhető tehát, mivel a Prologban szimbolikusan tetszőleges adatszerkezetet leírhatunk.

17.3. A logikai programok objektumai

A logikai programok objektumait *termekkel* írjuk le. Nincs szabványos LP termfogalom, de ez minden LP nyelvben hasonló, ezért – az ismétléseket elkerülendő – az ISO Prolog szabványnak megfelelően tárgyaljuk.

Egy Prolog term lehet *egyszerű term*, azaz *változó* vagy *állandó* (*konstans*); továbbá *összetett term*, azaz *struktúra*.

Egy *állandó* lehet *név* vagy *szám*, ami lehet *egész* vagy *lebegőpontos szám*. A számokat a szokásos módon írhatjuk.

A *nevek* kisbetűvel kezdődő azonosítók, vagy aposztrófok közé zárt karaktersorozatok is lehetnek. Nevek még a `+-*/\^<>=~:.?@#&$` karakterekből álló speciális jelsorozatok. (Például: `=< @>= ?- *$` stb.) Végül van még négy különleges név, ezek a `;` (or,else,elsif), a `!` (cut), a `[]` (nil) és a `{}` (empty).

A *változónevek* nagybetűvel, vagy aláhúzásjellel kezdődő azonosítók. A *változók* a matematikai egyenletek *ismeretleneinek* felelnek meg, azaz egy-egy ismeretlen objektumot jelöl-

nek. A számítások célja a kérdések változói, azaz ismeretlenei lehetséges értékeinek meghatározása. Ilyen módon nem lehetséges és nincs is értelme, hogy értékadó utasítást írjunk. Ha egy logikai változót behelyettesítünk, ez a továbbiakban a helyettesítő termtől megkülönböztethetetlen, hacsak vissza nem lépünk a helyettesítés elé, amikor is a változó elveszti értékét. Ez a procedurális nyelvekhez szokott programozónak idegen lehet, de a logikai programozás gyakorlása során természetessé válik. Azzal, hogy értékadás helyett változóhelyettesítéssel dolgozunk, talán a legveszélyesebb programozási hibaforrást küszöböltük ki.

Széles körben elterjedt jelölési szokás szerint azokat a változókat, amelyek helyettesítési értéke érdektelen, aláhúzásjellel kezdődő nevekkel jelöljük:

```
apa(Valaki) :- apja(Valaki,_Gyermek).
```

Ha programunk egy mondatában egy változónak egyetlen előfordulása van, értéke érdektelen, ugyanis nincs hova továbbítanunk[5]. Az egyetlen aláhúzásjelből álló *névtelen változó* minden előfordulása – még egyetlen mondaton belül is – más-más logikai változót jelöl, ez tehát csak érdektelen változókat jelölhet. *Minden más változó hatóköre az őt tartalmazó mondat.* Például:

```
apa_és_fiú(Valaki) :-
    apja(Valaki,_), szülője(_,Valaki).
```

Az *összetett termek* strukturált információ ábrázolására szolgálnak. Alakjuk $f(t_1, ..., t_n)$, ahol f tetszőleges *név*, $t_1, ..., t_n$ tetszőleges termek, és az f/n *függvényszimbólum* vagy *funktor* a $t_1, ..., t_n$ termeket egyetlen összetett termbe fogja össze[6]. Bármelyik t_i term lehet állandó, változó, vagy összetett term is. A változómentes termeket *alaptermeknek* nevezzük. (Az alaptermek ezért állandók vagy alaptermekből felépített összetett termek.)

Összetett termek segítségével tehát rekurzív adatszerkezeteket is leírhatunk. Egy üres fát reprezentálhatunk például az `o` állandóval, egy nem üres fát pedig `fa(`*gyökér*`,` f_1`, ..., ` f_n`)` alakban, ahol minden f_i egy-egy részfát jelöl.

A `fa(4,fa(2,fa(1,o,o),fa(3,o,o)),fa(5,o,o))` term ekkor egy kettő mélységű bináris rendezőfát ábrázol.

17.4. Listakezelés rekurzív LP módszerekkel

A *listákat* ezután unáris fáknak tekinthetjük. Egy *valódi lista*[7] lehet *üres*, vagy *nem üres*. A *nem üres listákat* szabványosan a `'.'/2` függvényszimbólum segítségével építjük fel, a `[]`

[5] Például a SICStus Prolog fordító feltételezi, hogy a nagybetűvel kezdődő azonosítók lényeges változókat jelölnek, ezért ezekre, ha egy állításban csak egy előfordulást talál, figyelmeztetést küld, segítve ezzel a gépelési hibák kiküszöbölését. A másik oldalon, ha egy a SICStus Prolog parancskérő jelénél (lásd a 646. oldalon) begépelt kérdésben egy változó neve aláhúzásjellel kezdődik, a rendszer cél megoldása után ennek értékét nem írja automatikusan ki.

[6] A függvényszimbólumokat tehát – a predikátumokhoz vagy relációkhoz hasonlóan – nevükkel és aritásukkal jellemezzük *névlaritás* formában.

[7] *Valódi lista*: szemben az alább bevezetendő *részleges listákkal.*

üres listából kiindulva (`[]` olvasata `nil`). Egy `.(X,Xs)` term pontosan akkor *valódi lista*, ha `Xs` is valódi lista.

Az `[1,2,3]` listát így `.(1,.(2,.(3,[])))` alakban írhatnánk le. Tekintettel a listák jelentőségére, a programok olvashatóságát növelendő egy első ránézésre talán kicsit bonyolult, de nagyon hasznos jelölésrendszert vezettek be.

A nem üres listák `.(X,Xs)` konstruktorát `[X|Xs]` alakban, a listák első néhány elemét pedig vesszővel elválasztva írhatjuk.
Az `[X1,X2,...,Xn|[]]` jelölésből `[]` elhagyható. Eszerint

```
| ?- .(X,Xs)==[X|Xs], [X1|[X2|Xs]]==[X1,X2|Xs],
     [X1,X2,X3|[]]==[X1,X2,X3],
      .(1,.(2,.(3,[])))==[1,2,3].
true
```

(Az `X==Y` Prolog hívás pontosan akkor sikeres, hogy az `X` és `Y` termek *azonosak* (17.7.2).) Az `[X|Xs]` valódi lista első eleme vagy más néven feje, az `X`, tetszőleges term, akár szabad változó vagy lista is lehet. Az `[X|Xs]` valódi lista maradéka `Xs`, maga is valódi lista kell legyen. Például `[a|[_]]` lista, de `[[_]|a]` nem az, mert az `a` név nem lista.

Egy term pontosan akkor *részleges lista* (parciális lista), ha szabad változó, vagy `[X|Xs]` alakú, ahol `Xs` részleges lista. Eszerint egy term pontosan akkor *részleges lista*, ha ő maga nem valódi lista, de megfelelő változóhelyettesítéssel azzá tehető.

A valódi és a részleges listákat együtt a továbbiakban *listáknak* nevezzük. Mint hamarosan látni fogjuk, a listák mindkét osztálya alapvető jelentőségű.

Végül, az *alaplista* egy változómentes lista, azaz olyan lista, ami egyúttal alapterm is. Az alaplisták tehát a valódi listák egy valódi részhalmazát adják.

Tegyük fel például, hogy `Xs` szabad változó. Ekkor `Xs` és `[1,2|Xs]` *részleges listák*, de `[Xs]` és `[1,2,Xs]` egy, illetve három elemű *valódi listák*. Az előbbi négy lista egyike sem alaplista. Ha viszont `Xs==3`, akkor `Xs` és `[1,2|Xs]` sem listák, de `[Xs]` és `[1,2,Xs]` alaplisták. Az alábbi programmal

```
list([]).
list([_X|Xs]) :- list(Xs).
```

a `list(L)` cél *valódi listával* paraméterezve változóhelyettesítés nélkül sikeres lesz. *Részleges listára* végtelen sok megoldása van, ezek az `L`-ből példányosítható valódi listák legáltalánosabb esetei. Például:

```
| ?- list(L).
L = [] ? ;  L = [_A] ? ;  L = [_A,_B] ? ;
L = [_A,_B,_C] ? ; L = [_A,_B,_C,_D] ? <Enter>
yes
```

A `list(L)` cél, ha nem listával paraméterezzük, meghiúsul.

A fenti jelölésekkel nézzünk most néhány predikátumot. Ezek segítségével bemutatjuk a listakezelő (és általában a rekurzív adatszerkezeteket kezelő) logikai programokban szokásos alapvető programozási módszereket.

17.4.1. A rekurzív keresés

Tekintsük először az *eleme* reláció klasszikus leírását.
(A % jel után a sor hátralevő része a Prologban megjegyzésnek számít.)

```
% member(X,Xs) :- X eleme az Xs listának.
member(X,[X|_Xs]).
member(X,[_X|Xs]) :- member(X,Xs).
```

Egy lista elemeit tehát a lista első eleme, és a maradékában lévő elemek adják. A predikátumhívás előfeltételének tekinthetjük, hogy ezen kérdés második paramétere egy valódi lista legyen. Részleges lista esetén ugyanis a vonatkozó cél keresési tere végtelen lesz. Valódi listánál a rekurzióban a lista hosszának csökkenése garantálja a keresési fa végességét. Ezt mutatják a következő futtatási eredmények is. (Próbáljuk meg felrajzolni az alábbi célok keresési fáját!)

```
| ?- member(X,[1,2,3]).
X = 1 ? ;  X = 2 ? ;  X = 3 ? ;  no
| ?- member(2,[1,2,3,X,4]).
true ? ;  X = 2 ? ;  no
```

A megfelelő elemet itt közvetlenül, vagy rekurzió segítségével közvetetten próbáljuk megkeresni. Az itt alkalmazott programozási módszert ezért *rekurzív keresésnek* nevezzük. (Figyeljük meg, miképpen alkalmaztuk a *rekurzív keresést* eddigi rekurzív predikátumainkban.)

17.4.2. Az eredmény fokozatos közelítése

A következő predikátum a listaösszefűzés klasszikus programja. Használhatjuk egy lista részekre bontására is. Vegyük észre, hogy két állításunk van, aszerint, hogy az első lista üres-e vagy sem. Ez gyakori szervezési elv a predikátumok megfogalmazásakor.

```
% append(Xs,Ys,XsYs) :-
%    Az Xs és Ys listák összefűzöttje az XsYs lista.
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Az üres lista és bármely Ys lista összefűzésének eredménye Ys. Egy nem üres [X|Xs] lista és bármely Ys lista összefűzésének eredményében az első elem X, míg a maradék az Xs és Ys összefűzésének eredménye. Mivel a rekurzióban az első és a harmadik lista hossza is csökken, a keresési fa végességéhez elegendő, hogy a hívásban ezen paraméterek valamelyike valódi lista legyen. A második paraméter mindegyik esetben lehet valódi vagy részleges lista is.

Vegyük most az `append([1,2],[3,4],Zs)` kérdés keresési fáját. (Feltesszük, hogy az LP rendszer az állítások minden `V` változóját minden célredukciós lépésben `Vi`-re nevezi át, ahol *i* a célredukciós lépés sorszáma.)

```
append([1,2],[3,4],Zs)
  { Zs <- [1|Zs1] }
    append([2],[3,4],Zs1)
      { Zs1 <- [2|Zs2] }
        append([ ],[3,4],Zs2)
          { Zs2 <- [3,4] }          % (megoldás)
% Végeredményben:
Zs=[1|Zs1]=[1|[2|Zs2]]=[1,2|Zs2]=[1,2|[3,4]]=[1,2,3,4]
```

Egyrészt tehát a keresési fa lineáris, mert mindenütt csak *egy* állítás feje illeszkedett a célra. Másrészt az összefűzött listát több lépésben, részleges listákon keresztül közelítettük. Minden lépésben – az aktuális részleges lista végére vonatkozó újabb változóhelyettesítéssel – a kimenő lista egyre nagyobb részét határoztuk meg, mígnem megkaptuk a végeredményként adódó összefűzött listát.

Ezt a programozási módszert, amikor tehát az eredmény adatszerkezetet felülről lefelé építjük fel, először bizonyos részleteit határozatlanul hagyva, majd ezeket a részleteket újabb és újabb változóhelyettesítésekkel egyre pontosítva, az *eredmény fokozatos közelítésének* nevezzük.

(Próbáljuk meg felrajzolni az alábbi kérdések keresési fáját! Figyeljük meg, hogyan érvényesül a számításokban az eredmény fokozatos közelítése! Az első feladatban a végeredmény is részleges lista, a másodikban pedig egy lista kettévágására elágazó keresési fát, és eredményül a lehetséges vágásokat kapjuk.)

```
| ?- append([1,2],Ys,Zs).
Zs = [1,2|Ys] ? ;
no
| ?- append(Xs,Ys,[1,2,3]).
Xs = [ ], Ys = [1,2,3] ? ;
Xs = [1], Ys = [2,3] ? ;
Xs = [1,2], Ys = [3] ? ;
Xs = [1,2,3], Ys = [ ] ? ;
no
```

17.4.3. Az akkumulátor módszer

A rekurzív predikátumokban alkalmazott harmadik számítási eljárás az *akkumulátor módszer*. Ezt a következő példaprogram illusztrálja.

```
% rev_app(Xs,Ys,Zs) :-
%       Az Xs lista fordítottját az Ys listával összefűzve kapjuk a Zs listát.
rev_app([ ],Ys,Ys).
rev_app([X|Xs],Ys,Zs) :- rev_app(Xs,[X|Ys],Zs).
```

Az üres lista fordítottjának és bármely `Ys` listának az összefűzöttje az `Ys` lista. Egy nem üres `[X|Xs]` lista fordítottjának és bármely `Ys` listának az összefűzöttje az `Xs` lista fordítottjának és az `[X|Ys]` listának az összefűzöttje.

Mivel a rekurzióban az első lista hossza csökken, a keresési fa végességéhez elegendő, hogy a hívás első paramétere valódi lista legyen. Ha az első paraméter részleges lista, a keresési fa nyilván végtelen lesz. Második és a harmadik paraméterként tetszőleges listákat várunk, ezek tehát lehetnek valódi vagy részleges listák, esetleg szabad változók is.
Nézzük most a `rev_app([1,2],[3,4],Zs)` kérdés keresési fáját.

```
rev_app([1,2],[3,4],Zs)
  rev_app([2],[1,3,4],Zs)
    rev_app([ ],[2,1,3,4],Zs)
      { Zs <- [2,1,3,4] }          % megoldás
```

A második, az *akkumulátor argumentumban* építjük fel az eredmény adatszerkezetet alulról fölfelé. Így az eredmény a rekurzió alján ebben az akkumulátor argumentumban áll elő. Az eredmény visszaadásához ezért szükségünk van egy harmadik, *alagút argumentumra*. A harmadik paraméter tehát változatlanul végigmegy egészen a rekurzió aljáig, és csak itt egyesítjük az akkumulátor értékével. Az *akkumulátor* és *alagút* argumentumokat együtt *akkumulátor párnak* nevezzük, mert a programokban mindig így, párban fordulnak elő.

A gyakorlati programokban gyakran írunk olyan predikátumokat, amelyekben – egyetlen predikátum argumentumait tekintve – több akkumulátor párt, illetve fokozatos közelítéssel előállított paramétert is találhatunk.

Megfigyelhetjük például, hogy a `?- append(Xs,Ys,[1,2,3])` célban – míg `Xs` fokozatos közelítéssel áll elő –, addig `Ys` alagútként, a harmadik paraméter pedig valamiféle negatív akkumulátorként viselkedik. (Nem *felépíti*, hanem *lebontja* a bemenő paraméterből az alagúton visszaküldeni kívánt adatszerkezetet.)

17.4.4. Az általánosítás módszere

Az alábbi predikátum segítségével az *általánosítás* módszerére adunk példát. Ilyenkor a problémát egy általánosabb feladatra való visszavezetéssel oldjuk meg. Ez természetesen nem a logikai programozás sajátsága; az emberi, illetve gépi problémamegoldás minden területén használjuk. Programkészítésnél gyakran hasznos, ha egy rekurzív eljárással az eredeti feladatnál általánosabbat oldunk meg, majd megfelelően paraméterezve hívjuk meg.

```
% reverse(Xs,Ys) :- Az Xs lista fordítottja az Ys.
reverse(Xs,Ys) :- rev_app(Xs,[ ],Ys).
```

Az `Xs` lista fordítottja az `Ys` lista, ha `Xs` fordítottját az üres listával összefűzve az `Ys` listát kapjuk.

Ez a predikátum is örökli tehát `rev_app/3`-tól azt a tulajdonságot, hogy ha a rá vonatkozó kérdés első paramétere valódi lista, akkor ezen cél keresési tere véges lesz; míg ha a rá vonatkozó cél első paramétere részleges lista, akkor ezen cél keresési tere végtelen lesz.

Feltéve, hogy `Xs` szabad változó, a `reverse([1,2,3],Xs)` cél rendben kiszámolja egyetlen, `Xs=[3,2,1]` megoldását, és keresési tere is véges lesz. Természetesen a

`reverse(Xs,[1,2,3])` kérdés egyetlen megoldása is ez, de keresési tere végtelen. A Prologban, mivel a `rev_app/3` kódjában a rekurziót leállító tény megelőzi a rekurzív szabályt, első válaszként ez utóbbi kérdésre is megtaláljuk ezt a megoldást, de ha további megoldást is keresünk, akkor állandóan visszalépünk, és egyre hosszabb változólistákat próbálunk a második paraméterrel egyesíteni, ami persze mindig sikertelen lesz. Ezzel szemben:

```
| ?- reverse([X,Y,Z],[1,2,3]).
X = 3, Y = 2, Z = 1 ? ;
no
```

Általában a fák kezelésénél is az itt megismert négy rekurziófüggő programozási módszert használjuk.

17.5. A Prolog gép

A teljes Prolog nyelv metalogikai és logikán kívüli nyelvi elemeket is tartalmaz. Ebben fejezetben az ezektől mentes *tiszta Prolog* általános végrehajtási mechanizmusát ismertetjük, majd ezt finomítjuk tovább, a mai Prolog megvalósítások irányában.

A tiszta Prolog program felhasználói predikátumok halmaza, amelyre kérdéseket tehetünk fel. Minden predikátum állítások (tények és szabályok) sorozata. Egyetlen szabálytörzs sem tartalmaz olyan célt, amely beépített eljárást (predikátumot) hívna meg.

17.5.1. Tiszta Prolog programok végrehajtása

Mint már említettük, a Prolog gép egy program végrehajtása során a feltett kérdés keresési fáját járja be, mégpedig preorder módon (17.2). Közben mindig csak a fa aktuális ágát tartja nyilván. Az aktuális ág minden csúcsát a hozzá tartozó célsorozat, és a célsorozat első elemi részcéljához (predikátumhívásához) tartozó predikátumnak a még ki nem próbált állításai címkézik, ez utóbbiak felírásuk sorrendjében. Az aktuális ág éleit a megfelelő célredukciós lépéshez felhasznált tény vagy szabály mellett a kapcsolódó egyesítő helyettesítésben behelyettesített célsorozatbeli változók halmaza címkézi. Vegyük észre, hogy nem szükséges az egyesítő helyettesítések feljegyzése, és – mint látni fogjuk – explicit kiszámítása sem, csak végrehajtásuk. A tiszta Prolog programok végrehajtásának alábbi algoritmusában – mint később meg is indokoljuk (17.5.2) – az *egyesítés* helyett az *illesztés* szót használjuk.

1. Kezdetben a keresési fa gyökerét az eredeti kérdés (célsorozat) címkézi. Ekkor a keresési fa gyökere az aktuális csúcs. (Mindig a gyökértől legtávolabbi csúcs lesz az aktuális).
2. Ha az aktuális csúcshoz az üres célsorozat tartozik, megoldást találtunk. Ez az eredeti kérdés változóinak aktuális helyettesítése. Ilyenkor a megoldást kiírjuk, majd megkérdezzük a felhasználót, hogy kér-e további megoldást. Ha igen, akkor (9)-től folytatjuk (visszalépés). Ha nem, akkor *sikeresen befejeztük a keresést.*
3. Ha az aktuális csúcshoz nem az üres célsorozat tartozik, akkor most az aktuális csúcsot az azt címkéző célsorozat első elemi részcéljához tartozó predikátum állításlistájával is felcímkézzük. Az állítások mindegyikének változóit átnevezzük úgy, hogy a kérdéstől változóidegen legyen.

4. Ha az aktuális állításlista üres, akkor zsákutcába jutottunk, és (9)-től folytatjuk (visszalépés).
5. q legyen az aktuális állításlista első eleme. Töröljük q-t e listáról.
6. Az aktuális célsorozat első predikátumhívását megpróbáljuk illeszteni q fejével. (Egy tény feje önmaga, törzse pedig üres. Egy *a:-b,...,z.* állítás feje *a*, törzse pedig a *b,...,z* célsorozat.) Az illesztés algoritmusát lásd a 17.5.2. pontban.
7. Ha az illesztés meghiúsul, (4)-től folytatjuk.
8. Ha az illesztés sikeres, egy célredukciós lépést végzünk: Az illesztés során behelyettesített változók minden, célsorozatbeli illetve állításbeli előfordulása is megfelelőképpen helyettesítődik. (Ezt rendesen a változók láncolt ábrázolása biztosítja.) A keresési fa éppen nyilvántartott ágát egy új csúccsal bővítjük. Az ide vezető élet az aktuális célsorozatnak az illesztés során behelyettesített változói halmazával és q-val címkézzük. Az új csúcs első címkéjét úgy kapjuk, hogy az aktuális célsorozat első predikátumhívását a q törzsével helyettesítjük, ahol tehát az illesztő helyettesítést már minden résztvevőn elvégeztük. Ezután az új csúcs lesz az aktuális, és (2)-től folytatjuk.
9. *Visszalépés:* Ha az aktuális csúcs a keresési fa gyökere, a program végrehajtása *meghiúsulással ér véget.* Különben töröljük a keresési fa aktuális csúcsát, és a fölötte lévő lesz az új aktuális csúcs. Közben a két csúcsot összekötő élet címkéző változóhalmaz elemeinek behelyettesítését töröljük. Ezután (4)-től folytatjuk.

17.5.2. Az illesztés algoritmusa

Mint láttuk, a Prolog gép működésének kulcsműveletei a célredukció és a visszalépés. A célredukció kulcsa viszont a kiválasztott részcél és állításfej egyesítése a legáltalánosabb egyesítő helyettesítés algoritmusa szerint. Mivel azonos funktornevű és aritású atomi formulákat kell egyesítenünk, a feladat azonos hosszúságú termsorozatok egyesítése. ISO Prologban, és így a modern Prolog változatokban is, hatékonysági okokból az egyesítő algoritmus alábbi, egyszerűsített változata használatos. A továbbiakban ezt nevezzük *illesztésnek*:

1. Ha az illesztendő termsorozatok üresek, készen vagyunk, különben illesztjük a két sorozat első elemét (2) szerint. Ezután folytatjuk az illesztést a maradék sorozatokra (1) szerint.
2. Ha mindkét term állandó, akkor attól függően, hogy különbözők vagy azonosak, az illesztés meghiúsul, vagy változóhelyettesítés nélkül sikeres lesz.
3. Ha egyik term állandó, a másik összetett, az illesztés meghiúsul.
4. Ha mindkettő összetett term, akkor azonos funktornév és aritás esetén paramétereik sorozatait (1) szerint illesztjük. Különböző funktornév vagy aritás esetén az illesztés meghiúsul.
5. Ha mindkét term változó, bármelyiket helyettesíthetjük a másikkal. (Hatékonysági okokból általában az állításfej változóját helyettesítjük a célsorozat változójával.)
6. Ha csak az egyik term változó, akkor ezt helyettesítjük a másik (*állandó* vagy *összetett*) termmel.

A fenti algoritmus változóhelyettesítései a célredukciós lépésben részt vevő célsorozatban és állításban a helyettesített változó minden előfordulására végrehajtódnak.

A célsorozat helyettesített változóit minden behelyettesítésnél feljegyezzük. Ha az illesztés végül meghiúsul, a célsorozat változóinak behelyettesítéseit e feljegyzés alapján megszüntetjük. Ha az illesztés végül sikeres lesz, a keresési fa éppen nyilvántartott ágának megfelelő élét e feljegyzés alapján felcímkézzük a célsorozat behelyettesített változóinak halmazával (17.5.1).

A fenti algoritmus (1)-es pontjában a termsorozatok megfelelő termpárjainak illesztése más sorrendben, vagy akár korutinszerűen is végrehajtható.

Vegyük észre, hogy az illesztés fenti algoritmusának (6)-os pontjában változó és összetett term illesztése esetén az eredeti egyesítő algoritmus szerint csak akkor hajthatnánk végre a változóhelyettesítést, ha a változó a struktúrában nem fordul elő. Ez a *változóelőfordulás-ellenőrzés*. Ez az eredeti egyesítő algoritmus szerves része [Fek 90, Kow 79, Der 96, Ste 94]. Tegyük fel például, hogy adott az `eq(X,X).` tény, és feltesszük az `eq(Y,f(Y))` kérdést. Ez a klasszikus célredukció szabályai szerint meghiúsul, a Prolog változat szerint azonban az `{X<-f(Y),Y<-f(Y)}` helyettesítést állítja elő. Ez az eredeti definíció szerint nem egyesítő helyettesítés. A Prolog szabvány szerint meghatározatlan az eredmény, ha az illesztő algoritmus során egy változó az őt tartalmazó struktúrával kerül szembe [Der 96]. A mi illesztő algoritmusunk speciálisabb a szabványosnál [Der 96], amely ebben az esetben akár végtelen ciklusba is kerülhet. Kikerüljük ugyanis az illesztő helyettesítés előállítását. Így ugyan jobban megkötjük a Prolog fordító megvalósítójának kezét, de egyrészt elkerüljük a végtelen ciklusokat, másrészt Colmerauer [Col 82] szellemében azt mondhatjuk, hogy egy `{Y<-f(Y)}` helyettesítés egy ciklikus `Y=f(Y)` termet hoz létre. A SICStus Prolog például eszközöket is ad az ilyen típusú termek kezelésére [Spl 02].

A megközelítés hátránya, hogy a ciklikus termekkel dolgozó programok általában nem jól hordozhatók a különböző Prolog változatok között. Másrészről viszont vegyük észre, hogy a *változóelőfordulás-ellenőrzés* elhagyása egyike azoknak a zseniális kompromisszumoknak, amelyek a Prologot gyakorlatilag használható programozási nyelvvé teszi. Enélkül ugyanis egy változó és egy összetett term egyesítésének műveletigénye a struktúra méretével lenne arányos, míg így – a Prolog termeket láncoltan ábrázolva – az illesztés az összetett termtől független, állandó költséggel, valójában egyetlen pointerállítással megoldható.

17.5.3. NSTO programok

Ezeknél nincs szükség változóelőfordulás-ellenőrzésre. A továbbiakban tehát megvizsgáljuk, hogyan írhatunk olyan programokat, amelyekben a predikátumhívások során az eredeti egyesítő algoritmus sem végezne sikeres *változóelőfordulás-ellenőrzést*. Az ilyen predikátumhívásokat *NSTO* céloknak (not subject to occurs check) nevezzük, különben *STO* célokról beszélünk. Ha egy programban minden cél NSTO, akkor maga a program is NSTO. Különben a program STO.

Ezzel kapcsolatosan egy elégséges feltételt fogalmazunk meg. (További, bonyolultabb feltételek találhatók még [Der 93]-ban.)

Ha egy program minden állítása eleget tesz az alábbi feltételek *valamelyikének*, és a beépített eljárásokra vonatkozó hívások is NSTO célok, akkor a program maga is NSTO.

1. Az állításfejben és a vele kapcsolatos kérdésekben is paraméterként csak egyszerű termek (állandók és változók) használatosak.
2. Az állításfej nem tartalmaz kettőzött változót, azaz olyan változót, amelynek egynél több előfordulása lenne.
3. Az állításra vonatkozó célok nem tartalmaznak kettőzött változót.

A fenti feltételek[8] felhívják figyelmünket a programban lehetséges kritikus pontokra. Ezeken a pontokon az alábbi példa szerint járhatunk el. Tegyük fel, hogy a `member/2` predikátumot a szokásos módon írtuk le:

```
% member(X,Xs) :- X eleme az Xs listának (NSTO célokhoz).
member(X,[X|_Xs]).
member(X,[_X|Xs]) :- member(X,Xs).
```

Látható, hogy ennek első állításával lehet probléma az NSTO tulajdonság szempontjából, hiszen kettőzött változót tartalmaz. Gyakorlati problémák esetén ilyenkor majdnem mindig teljesül az NSTO tulajdonság harmadik elégséges feltétele, esetünkben az, hogy a `member(Y,Ys)` célban az `Y` és `Ys` együtt sem tartalmaz kettőzött változót.

Ha a fenti feltételek alapján sem lehetünk biztosak abban, hogy az NSTO tulajdonság teljesül, és más, saját ötletünk sincs ennek bizonyítására (5. feladat), vagy ha éppen az bizonyítható, hogy egy cél STO, célszerű a kérdésben a fenti `member/2` predikátum helyett az alábbi `eleme/2` predikátumot meghívni:

```
% eleme(X,Xs) :- X eleme az Xs listának (STO változat).
eleme(X,[Z|_Xs]) :- unify_with_occurs_check(X,Z).
eleme(X,[_X|Xs]) :- eleme(X,Xs).
```

Itt a `unify_with_occurs_check(X,Z)` cél a Prolog megfelelő, szabványos beépített eljárását hívja meg, amely az `X` és `Z` Prolog termeket változóelőfordulás-ellenőrzéssel egyesíti, feltéve, hogy van egyesítő helyettesítésük, különben meghiúsul:

```
| ?- member(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
X = f(f(f(f(f(f(f(f(f(f(...)))))))))) ? ;
no
| ?- eleme(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
no
```

[8] Eddigi programjainkban a programot elindító kérdést minden esetben a Prolog fejlesztői környezetéből, interaktívan tettük föl. Ez alapján úgy tűnhet, mintha most a programok használatát önkényesen korlátozni akarnánk. A gyakorlati programozásban azonban csak a tesztelési szakaszban használjuk így programjainkat, ily módon a procedurális nyelvekben szokásos tesztágyak megírását elkerülve. A végső program – mint a fejlesztői környezettől független alkalmazás – tartalmazza az indító direktívát (17.1) is, környezetével pedig a szokásos felületi eszközök segítségével tartja a kapcsolatot. Így az állításokra vonatkozó lehetséges hívások behatárolhatók.

```
| ?- member(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z), Y = g(Z) ? ;
X = g(g(g(g(g(g(g(g(g(...)))))))))),
Y = g(g(g(g(g(g(g(g(g(...)))))))))) ? ;
no
| ?- eleme(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z), Y = g(Z) ? ;
no
```

Általában, ha egy programban használunk összetett termeket, akkor azokra az állításokra, amelyek fejében kettőzött változók szerepelnek, és a rájuk vonatkozó Prolog célok NSTO tulajdonságát nem tudjuk bizonyítani, a következő átalakítást végezzük el: míg az állításfejben az NSTO tulajdonság szempontjából kritikus kettőzött változót találunk (jelölje most `X`), egyik előfordulását nevezzük át tetszőleges, az adott állításban még nem használt változónévre (jelölje most `Z`), majd az állításstörzs elejére szúrjunk be egy új `unify_with_occurs_check(X,Z)` hívást.

Ilyen módon programunk minden predikátumhívása NSTO lesz, kivéve a fenti átalakítással előállított `unify_with_occurs_check(X,Z)` hívásokat. Programunk így az elsőrendű logika szabályainak megfelelően működik, és hordozhatósága sem sérül. A Prolog beépített eljárásai esetében a hívások NSTO tulajdonsága általában automatikusan teljesül. Az ebben a munkában tárgyalt beépített eljárások közül kivételek az `(=)/2` és `(\=)/2` (17.6, 17.6), `arg/3` (17.7.3), `read/2` (17.9.2), `retract/1` és `retractall/1` (17.9.3), `findall/3` (17.10), valamint a `catch/3` (17.11) predikátumok. Az ezekre vonatkozó hívások STO/NSTO tulajdonsága a fentiekhez hasonlóan ellenőrizhető és kezelhető.

Az itt ismertetett módszer lényege, hogy a klasszikus – és költséges – változóelőfordulás-ellenőrzéses egyesítést a programoknak csak azon – elvétve előforduló – pontjain alkalmazzuk, ahol az valóban szükséges.

17.5.4. Első argumentum indexelés

Az első argumentum indexelés – ha programunk kódolásánál figyelembe vesszük – nagyban növelheti a program hatékonyságát.

A választási pontok létrehozása és kezelése ugyanis viszonylag költséges. Predikátumhívásaink azonban gyakran *determinisztikusak*, ami alatt a Prologban azt értjük, hogy pontosan egy állításfej illeszkedik rájuk. (Egy hívás *indeterminisztikus*, ha több állításfejjel is illeszthető. Ilyenkor ugyanis a program futása elvileg többféleképpen is folytatódhatna, bár a Prolog-fordító az indeterminisztikusságot az ismert visszalépéses vezérléssel oldja fel.) Ha a Prolog gép felismeri egy predikátumhívás determinisztikusságát, nem hoz létre választási pontot. Az első argumentum indexelés pedig éppen a determinisztikusság felismerését támogatja. Lássuk, hogyan:

A 17.5.1. pontban ismertetett programvégrehajtó algoritmus (3). lépésében a keresési fa aktuális csúcsába felvettük az aktuális célsorozat első részcéljához tartozó predikátum összes

állításának listáját. Ezzel – feltéve, hogy a lista egynél több elemet tartalmaz – egy választási pontot hoztunk létre, ahol tehát a keresési fa elágazik a predikátumhíváshoz illeszkedő állításfejek szerint.

A fenti listára nyilván elég lett volna azokat az állításokat felvenni, amelyek feje illeszthető a kérdéses részcéllal. Ezeket előre meghatározni azonban túl sok spekulatív munkával[9] jár, ezért a legtöbb Prolog megvalósítás egy kompromisszumos megoldást alkalmaz, amit *első argumentum indexelésnek* nevezünk. Ennek lényege:

1. Abban az esetben, ha a hívás első paramétere állandó, csak azok az állítások kerülnek a listára, amelyek fejének első formális paramétere ugyanaz az állandó, vagy változó.
2. Ha a hívás első paramétere összetett term, akkor csak azok az állítások kerülnek a listára, amelyek fejének első formális paramétere változó, vagy ezen formális paraméter fő funktora megegyezik a hívás első paraméterének fő funktorával.
3. Ha a hívás első paramétere változó, akkor a hivatkozott predikátum minden állítása a listára kerül.

A lehetséges listákat a Prolog-fordító rendesen már fordítási időben létrehozza, és egy táblázatot készít belőlük, ahonnan a megfelelő lista kiválasztása futás közben már elhanyagolható műveletigénnyel lehetséges[10].

A fentiek szerint a `list/1`, `append/3`, `rev_app/3` predikátumok (17.4) futása során – ha a megfelelő célok első paramétereit valódi listákként adjuk meg – minden eljáráshívás determinisztikus lesz, és egyetlen (rekurzív) hívás sem hoz létre választási pontot. Ha ugyanis a cél első paramétere üres lista, mindegyik esetben az első, míg ha nem üres lista, a második állításra indexelünk. A `member/2` predikátumot az első argumentum indexelés nem érinti, mivel mindkét állítása fejrészének első paramétere változó.

17.5.5. Utolsó hívás optimalizáció

Ha a (17.5.1) részben ismertetett Prolog gép működésének (5)-ös pontjában az aktuális állításlista egyelemű volt, akkor első elemének törlése után üres állításlista maradt, ami azt jelenti, hogy egy későbbi esetleges visszalépés ezen a csúcson a (4)-es pont szerint újabb visszalépést eredményez. Ezért a Prolog gép algoritmusának (8)-as pontjában ilyenkor megtehetjük, hogy a keresőfa éppen ekkor nyilvántartott ágának új csúccsal való bővítése helyett az aktuális csúcsot írjuk felül, az ide vezető élet címkéző változóhalmazhoz pedig hozzávesszük azokat a változókat is, amelyek az eredeti változatban az új csúcshoz vezető élet címkéznék. Ilyenkor tehát a keresőfa aktuális ága nem lesz hosszabb.

Ha tehát egy predikátumhívást megvalósító állítástörzs utolsó részcéljának meghívása pillanatában a tartalmazó predikátum meghívása óta nincs élő választási pont, akkor ez az utolsó hívás újrahasznosíthatja a szülője (a tartalmazó predikátum) meghívása által igényelt

[9] Spekulatív munka: amit nagy valószínűséggel részben megtakaríthatnánk.

[10] A változó-átnevezések ugyanúgy oldhatók meg, mint a procedurális nyelveknél. Másrészt, a többszörös argumentum-indexelés éppen a táblázat méretének gyors növekedése miatt nem szokásos.

memóriát. Ezt *utolsó hívás optimalizációnak* nevezzük. Ha itt a szülő predikátum önmagát hívja, akkor ez a végrekurzió lényegében véve ciklusként működhet, azaz *végrekurzió-optimalizáció* alkalmazható.

Mivel a `list/1`, `member/2`, `append/3` és `rev_app/3` predikátumok (17.4) futása során a rekurzív hívás pillanatában nincs élő választási pont, mindegyik esetben végrekurzió-optimalizációt végezhetünk.

Amikor pedig a `reverse/2` predikátum hívja meg a `rev_app/3` eljárást, utolsó hívás optimalizáció alkalmazható.

17.6. Vezérlésmódosítás a Prologban

Az eddig tárgyalt tiszta Prolog nyelvben nem tudjuk megfogalmazni a tagadást. Azt például, hogy két term illeszthető (`A=B`), könnyen leírhatjuk, például az `X=X.` tény segítségével. Ennek tagadását azonban (`A\=B`) még nem tudjuk megfogalmazni[11]. Hasonlóképpen, a `member/2` predikátumot definiáltuk a (17.4.1) részben, de a `nonmember/2` predikátumhoz eddigi eszközeink elégtelennek bizonyulnak. Ebből már következik, hogy például két rendezetlen lista unióját vagy metszetét sem tudjuk megfogalmazni, hiszen itt meg kéne mondani, mi legyen, ha az egyik lista egy eleme a másik listán nem található.

Állításaink – tényeink és szabályaink – általában nem alkalmasak arra, hogy negatív információt következtessünk ki programunkból. Legfeljebb azt tudjuk eldönteni, hogy egy állítás következik-e a programból. (Például ha az állítás – mint cél – keresési tere véges.) Ha viszont egy ilyen eldönthető kérdésről feltehető, hogy bizonyíthatósága egybeesik igazságával, bizonyíthatatlanságából egyben negáltjára következtethetünk. (Ilyenek például a (17.4) listakezelő predikátumaira vonatkozó kérdések.) Ha tehát rendelkezni tudnánk arról, hogy mi történjen, ha egy célsorozat sikeres (illetve ha meghiúsul), valamelyest korlátozott, de gyakorlati eszközünk lenne a tagadás kezelésére.

17.6.1. Feltételes célok

A strukturált programozási stílus szempontjából talán a legfontosabb vezérlésmódosító eszközök a *feltételes célok*. Alapformájuk:

```
( ha -> akkor
; különben
)
```

Ahol a *ha*, az *akkor* és a *különben* is tetszőleges Prolog célsorozatok lehetnek. Amennyiben a *ha* célsorozat sikeres, a feltételes cél megoldásait az *akkor* célsorozat megoldásai adják. Egyébként a feltételes cél megoldásait a *különben* célsorozat megoldásai szolgáltatják.

A fenti megfogalmazásból következik, hogy a *ha* célsorozatba sikeres végrehajtása esetén sem lehetséges a visszalépés. Ilyenkor ugyanis a *ha* célsorozat esetleges választási pontjait a Prolog rendszer közvetlenül annak első sikeres végrehajtása után levágja. A *különben* ág

[11] Az `(=)/2` és a `(\=)/2` valójában a szabványos Prolog nyelv [Der 96] beépített predikátumai, ezért nem definiálhatjuk őket felül. A Prolog célokban mindkettő neve közbevetett (infix) módon írható (17.8).

helyére – külön zárójelpár nélkül – feltételes cél is írható, hasonlóan az ALGOL, Pascal, vagy a C nyelvek többirányú elágazásaihoz. A

(*ha* -> *akkor*)

feltételes cél is megengedett. Mivel ennek nincs *különben* ága, ott megoldások sincsenek. Így ez a fajta feltételes cél egyenértékű a

(*ha* -> *akkor* ; fail)

szerkezettel, ahol a `fail/0` beépített eljárás, amelynek hívása mindig meghiúsul. Ha azt akarjuk, hogy a *ha* meghiúsulása esetén a feltételes cél sikeres legyen, akkor a

(*ha* -> *akkor* ; true)

szerkezetet használjuk, ahol a `true/0` beépített eljárás, amelynek hívása mindig sikeres.

A jelen alfejezet (17.6) bevezetőjében említett problémák néhány lehetséges megoldása ezután a következő.

```
% X\=Y :- X nem illeszthető Y-nal.
X\=Y :- ( X=Y -> fail ; true ).

% nincs_benne(Xs,Y) :- Az Xs listán nem található Y.
nincs_benne([ ],_Y).
nincs_benne([X|Xs],Y) :-
    ( X = Y -> fail
    ; nincs_benne(Xs,Y)
    ).

% nonmember(X,Xs) :- nincs_benne(Xs,X).
nonmember(X,Xs) :- ( member(X,Xs) -> fail ; true ).

% metszet(Xs,Ys,Zs) :-
%    Az Xs lista Ys listán nem szereplő elemei elhagyásával adódik a Zs lista.
metszet([ ],_Ys,[ ]).
metszet([X|Xs],Ys,Zs) :-
    ( member(X,Ys) ->
        Zs = [X|Ms], metszet(Xs,Ys,Ms)
    ; metszet(Xs,Ys,Zs)
    ).

% unió(Xs,Ys,Zs) :-
%     Az Xs lista Ys listán nem szereplő elemeit sorban Ys elé kapcsolva kapjuk
%     a Zs listát.
unió([ ],Ys,Ys).
unió([X|Xs],Ys,Zs) :-
    ( member(X,Ys) -> unió(Xs,Ys,Zs)
    ; Zs = [X|Us], unió(Xs,Ys,Us)
    ).
```

Vegyük észre, hogy a `nincs_benne/2`, a `metszet/3` és az `unió/3` rekurzív predikátumok argumentumsorrendjét az első argumentum indexelés kihasználása vezérelte (17.5.4). Mindegyik rekurzív hívásra végrekurzió-optimalizáció végezhető (17.5.5). Az ezeket néhol megelőző `Zs=[X|Vs]` alakú célok, az egyenlőség paramétereinek *illesztését* célozzák.

A szabványos Prolog nyelv ugyanis *nem* tartalmaz értékadó utasítást, hiszen a logikai változók az állítások *ismeretlenjei* (17.3), [Der 96, Kee 90, Ste 94], ami azt jelenti, hogy a program írásakor ismeretlen, a program által kiszámítandó objektumot jelölnek. Ha tehát egyszer behelyettesítjük a logikai változót, azaz meghatározzuk a programban kódolt relációk által definiált, de eddig ismeretlen értékét, nincs értelme azt felülírni. Tipikus kezdő Prolog programozói hiba az `Xs=[Y|Xs]` cél és hasonlók használata. Ez a Prolog cél biztosan nem fogja végrehajtani a programozó által óhajtott értékadást. Ha például az `Xs` alapterm, az `Xs` és az `[Y|Xs]` illesztése nyilván meghiúsul. Ha az `Xs` szabad változó, akkor a cél STO, ami a Prologban általában nem kívánatos stb.

A `member(X,Xs)` cél (17.4.1) futtatása – visszalépések során – e kérdés minden lehetséges megoldását felsorolja. Első megoldása után általában választási pontot hagy maga után. Ha azonban csak ellenőrzésre szeretnénk használni, ez gyakran zavaró lehet. Befejezésül két különböző megoldást adunk erre a problémára. Az első azt példázza, hogy a feltételes szerkezet (azaz feltételes cél) a felesleges választási pontoktól is megszabadít bennünket, ahogy fenti példáinkban is. A második azt mutatja, hogy a meghiúsulást nem kell expliciten programoznunk. Ha ez zavar bennünket – mint ez hagyományos programozási nyelveken nevelkedett programozókkal gyakran megesik – hatékonysági okokból jobb, ha a felesleges állításokat az alábbi példának megfelelően megjegyzésbe tesszük.

```
member1(X,Xs) :- ( member(X,Xs) -> true ).

% member_check(_X,[]) :- fail.
member_check(X,[Y|Ys]) :-
    ( X = Y -> true
    ; member_check(X,Ys)
    ).
```

17.6.2. Tagadás

Tegyük fel, hogy szeretnénk definiálni a `nőtlen_hallgató/1` Prolog predikátumot, feltéve, hogy adottak a `hallgató/1` és a `nős/1` predikátumok. Az előző részben mondottak alapján nyilvánvalóan adódik az alábbi megoldás.

```
nőtlen_hallgató(X) :- hallgató(X), nőtlen(X).

nőtlen(X) :- ( nős(X) -> fail ; true ).

hallgató('Péter'). hallgató('János').
hallgató('Jakab').
nős('Péter').      nős('József').
```

Megfigyelhetjük, hogy a `(\=)/2` a `nonmember/2` és a `nőtlen/1` közös sémája a következő:

```
nem(P) :- ( P -> fail ; true ).
```

Kérdés, hogy a tagadásnak ez az általánosabb formája értelmezhető-e Prolog predikátumként.

Világos, hogy a Prolog célok formailag összetett termek vagy esetleg nevek. Ez lehetővé teszi, hogy egy Prolog célt futási időben határozzunk meg – hisz ez a kiszámító részprogram számára közönséges Prolog term –, majd mint *metacélt* hívjuk meg. A metacél a forráskódban általában változóként, de egy Prolog célsorozat elemeként jelenik meg. Az `X=p(Y),X` célsorozat például jelentésében egyenértékű a `p(Y)` céllal. A Prolog szabályok törzsében megjelenő metacélok egyben a szabályfej formális paraméterei is lehetnek, mint a fenti `nem(P)` fejű szabály esetén is. Ha egy szabály törzsében megjelenő metacél egyben a szabályfej formális paramétere, a szabályt tartalmazó predikátumot *metapredikátumnak* nevezzük, mint például a `nem/1` predikátumot. A megfelelő paraméterhelyet – mint a `nem/1` predikátum egyetlen argumentumát – *meta-argumentumnak* nevezzük. A meta-argumentumokat a metapredikátum hívásakor célokkal paraméterezhetjük.

A `nőtlen_hallgató/1` predikátumot így is megadhatnánk:

```
nőtlen_hallgató(X) :- hallgató(X), nem(nős(X)).
```

A `nem/1` metapredikátum egyetlen formális paramétere `P`, amely a szabálytörzsben metacélként hívódik meg: ha `P` sikeres, esetleges választási pontjait a feltételes szerkezet levágja, majd a feltételes cél *akkor* ágán a `fail` cél hajtódik végre és a `nem(P)` hívás meghiúsul. Ha `P` meghiúsul, a feltételes cél *különben* ágán a `true` cél hajtódik végre és a `nem(P)` hívás sikeres lesz. Összegezve, `nem(P)` pontosan akkor sikeres, ha a `P` cél meghiúsul (és így `P`-nek egyetlen megoldása sem adódik).

Ez a *negáció, mint meghiúsulás* elve. Ez *nem* logikai tagadás, de viszonylag hatékonyan valósítható meg[12], és némi programozói önfegyelemmel alkalmazva általában kiváltja azt.

Figyeljük meg először a `nem(P)` cél négy tulajdonságát:

I. Sosem hagy maga után választási pontot.
II. Függetlenül a `P` céltól, sosem példányosítja annak változóit.
III. `nem(P)` pontosan akkor sikeres, ha `P` keresési fája véges, és nem tartalmaz megoldást.
IV. `nem(P)` pontosan akkor hiúsul meg, ha `P` keresési fája tartalmaz megoldást, de az első megoldás ág előtt nincs végtelen ág.

(Vegyük észre, hogy ezek a megfigyelések akkor is érvényesek, ha a tagadás implicit, azaz a konkrét esetben feltételes céllal programozzuk, mint fentebb a `(\=)/2`, a `nonmember/2` és a `nőtlen/1` predikátumoknál.)

(II)-ből világos, hogy egy negált Prolog cél sosem ad megoldás(oka)t, azaz nem lehet (rész)eredmények kiszámítására, hanem csak szűrésére, azaz ellenőrzésére használni. Ezzel

[12] Meg kell jegyeznünk, hogy bár a `nem(nős(X))` és a `nőtlen(X)` ugyanazt jelentik, a `nem(nős(X))` a mai Prolog rendszerekben általában kevésbé hatékony, mert a metacélok legtöbbször nem optimalizálhatók a program fordítása során.

összhangban, nem mindegy, hogy egy negált cél hol helyezkedik el a célsorozatban, hiszen az elvárt működéshez általában szükséges, hogy változói példányosítva legyenek. Például

```
| ?- nőtlen_hallgató(X).
X = 'János' ? ;  X = 'Jakab' ? ; no
```

függetlenül attól, hogy melyik változatot használjuk. Ha azonban a szabálytörzs két részcélját fölcseréljük (megint mindegy, hogy melyik változatban), akkor a

```
| ?- nőtlen_hallgató(X).
no
```

eredményt kapjuk. Ilyenkor ugyanis a `nőtlen(X)`, illetve a `nem(nős(X))` cél meghiúsul, hiszen a `nős(X)` – lévén `X` még szabad változó – sikeres lesz.

Általában a Prolog tagadás logikai értelemben vett helyességének *elégséges* feltétele, hogy a negált cél a meghívás pillanatában ne tartalmazzon változót, és keresési fája véges legyen[13].

A `nem/1` predikátumban fent leírt tagadást a Prologban a `(\+)/1` beépített eljárás hatékonyabban valósítja meg. Ez előtag műveleti jelként is használható, mint a `nőtlen_hallgató/1` predikátum alábbi verziójában (lásd még (17.8)):

```
nőtlen_hallgató(X) :- hallgató(X), \+ nős(X).
```

17.7. A Prolog metalogikai predikátumai

A gyakorlati programozásban gyakran van szükségünk olyan információkra, számításokra, amelyek eddigi eszközeinkkel nem, vagy nem elég hatékonyan fejezhetők ki. Ilyenek az aritmetikai számítások, ismeretlen adatok típusának meghatározása, összehasonlításuk, ismeretlen szerkezetű termek kezelése stb.

17.7.1. Aritmetika

A Prologban a szokásos aritmetikai műveleti jelek, mint a `+`, `-`, `*`, `/`, `**` stb. csupán közbevetett (infix) illetve előtag (prefix) műveleti *jelként* írható függvényszimbólumok. Ennek az az oka, hogy a nyelv támogatja a szimbolikus számításokat, mint például a `-x^2+3*x-4` polinom deriválása vagy integrálása. Ebből a szemléletből fakad, hogy például a `2+3` kifejezés itt csak a `+(2,3)` termet jelenti, és a Prolog rendszer általában nem, kizárólag csak *aritmetikai argumentumokban* értékeli ki. Ezek

- az `is/2` beépített eljárás jobb oldali argumentuma, és
- az `(=:=)/2`, `(=\=)/2`, `(<)/2`, `(>)/2`, `(=<)/2`, `(>=)/2` aritmetikai összehasonlító predikátumok argumentumai.

[13] A tagadással kapcsolatban a *zárt világ feltételezéssel* élünk. Ez röviden azt jelenti, hogy feltételezésünk szerint elegendő információ áll rendelkezésünkre, azaz az aktuális modellben minden állítás, ami nem következik a programunkból, hamis.

A fenti eljárások nevei a hívásokban közbevetett (infix) jelöléssel írhatók. A ⟨*term*⟩ `is` ⟨*exp*⟩ cél tehát először kiértékeli az ⟨*exp*⟩ aritmetikai kifejezést, majd az eredményt megpróbálja *illeszteni* a ⟨*term*⟩ Prolog termmel. *Aritmetikai kifejezés* egy egész (`integer`) vagy lebegőpontos (`float`) szám, illetve egy összetett (`compound`) term, amelynek funktora aritmetikai függvényszimbólum (ezek a szokásos aritmetikai műveleti jelek, logaritmikus, trigonometrikus stb. függvényjelek [Spl 02, Der 96]), paraméterei pedig aritmetikai kifejezések. (Ez azt jelenti, a Prolog célok *aritmetikai argumentumait* paraméterező kifejezések változóit a célok meghívásáig megfelelően be kell helyettesíteni.) Például (`//` az egészosztás, `floor` az egészrész):

```
| ?- X is -2**3, Y is floor(cos(0))+3*(8-7//2).
X = -8.0,  Y = 16
| ?- X = -2**3, Y = floor(cos(0))+3*(8-7//2).
X = -2**3,  Y = floor(cos(0))+3*(8-7//2)
| ?- 5 is 2+3, 5.0 =:= 2+3.
yes
| ?- 2+3 is 2+3.
no
| ?- 5.0 is 2+3.
no
```

Tipikus kezdő Prolog programozói hiba az `X is X+1` cél és hasonlók használata. Ez a Prolog cél biztosan nem fogja végrehajtani a programozó által óhajtott értékadást. Ha például a fenti utasítás meghívásakor az `X` értéke egy szám, az `X+1` értékének és az `X` értékének illesztése meghiúsul. Ha pedig az `X` szabad változó, akkor az `X+1` kiértékelése az `instantiation_error` kivételt váltja ki (17.11).

Fel kell még hívnunk a figyelmet arra, hogy az aritmetikai összehasonlító predikátumok csak valódi aritmetikai kifejezések számértékeit hajlandók összehasonlítani. Például az `Y=:=X+1` utasítás *nem* használható az `Y is X+1` helyett az `X+1` értékének kiszámítására:

```
| ?- Y is 6+1.
Y = 7
| ?- Y =:= 6+1.
{INSTANTIATION ERROR: _157=:=6+1 - arg 1}
```

A most következő két predikátum egyenlő hosszúságú valódi számlistákkal ábrázolt vektorok összegét és skaláris szorzatát számolja ki.

```
% add(V1,V2,V) :-
%    A V vektor a V1 és V2 vektorok összege.
add([],[],[]).
add([X|Xs],[Y|Ys],[Z|Zs]) :-
   Z is X+Y, add(Xs,Ys,Zs).
```

```
% mult(V1,V2,S) :-
%    S a V1 és V2 vektorok skaláris szorzata.
mult([],[],0).
mult([X|Xs],[Y|Ys],S) :-
   mult(Xs,Ys,S0), S is S0+X*Y.
```

A fenti predikátumok determinisztikusak az első argumentum indexelés szerint. `add/2`-re így végrekurzió-optimalizáció is alkalmazható. `mult/2` is átalakítható ilyenné, ha bevezetünk egy akkumulátort, amiben a részösszegeket gyűjtjük, vagyis általánosítjuk a feladatot úgy, hogy a két vektor skaláris szorzatát egy kezdőértékhez kell hozzáadni:

```
mult(V1,V2,S) :- mult(V1,V2,0,S).

% mult(V1,V2,A,S) :- S = A+V1*V2.
mult([],[],A,A).
mult([X|Xs],[Y|Ys],A0,S) :-
    A1 is A0+X*Y, mult(Xs,Ys,A1,S).
```

17.7.2. Termek típusa és összehasonlítása

Egy Prolog term (lásd a 17.3. pontot) lehet *változó* (`var`), *állandó* (`atomic`), vagy *összetett*, azaz *struktúra* (`compound`). Az állandók és a struktúrák együttes típusneve `nonvar`.

Egy *állandó* lehet *név* (`atom`), vagy *szám* (`number`), ami lehet *egész* (`integer`) vagy *lebegőpontos* (`float`).

Zárójelben megadtuk a típusok eredeti Prolog-beli neveit is, mert ezek egyben a megfelelő (egy aritású) típusellenőrző Prolog predikátumok nevei is. Például a `nagyszülője/2` predikátumot (17.1.2) hatékonyabbá tehetjük a következő módon (az Olvasóra bízzuk a változtatás magyarázatát):

```
nagyszülője(X,Y) :-
   ( var(X), nonvar(Y) -> szülője(Z,Y), szülője(X,Z)
   ; szülője(X,Z), szülője(Z,Y)
   ).
```

A Prologban két tetszőleges term összehasonlítható. Két term azonosságát vizsgálják az `(==)/2` és a `(\==)/2` beépített predikátumok. Eszerint az

```
X \== Y, X=Y, X==Y.
```

célsorozat sikeres, hiszen a két változó eredetileg különbözik, de az illesztésük során azonosakká válnak.

A *termek szabványos rendezésében* a változók megelőzik a lebegőpontos számokat, ezek az egészeket, ezek a neveket, ezek pedig az összetett termeket. Az `A @< B` cél pontosan akkor sikeres, ha a termek szabványos rendezése szerint `A` megelőzi `B`-t, azaz `A` kisebb mint `B`.

A valós számok egymás között számértékük szerint rendezettek, és az egész számokról is ezt mondhatjuk, de a valósak kisebbek, mint az egészek:

```
| ?- 5 < 5.1, 5.1 @< 5, 9.9e99 @< -999999999.
yes
```

A *nevek* az őket alkotó karakterláncok szerint ábécésorrendbe rendezettek, ahol az egyes karaktereket ASCII kódjaik szerint hasonlítjuk össze. A *struktúrák* elsősorban aritásuk, másodsorban nevük szerint rendezettek; végül paramétereiket kell sorban összehasonlítani a lexikografikus rendezettség szabályai szerint, a paraméterpároknál újra a szabványos term rendezést alkalmazva.

A változók rendezése minden esetben meghatározott, de rendszerfüggő. A változók és a változót tartalmazó struktúrák rendezettsége a változóhelyettesítések során változhat.

A szabványos rendezettség szerint termjeinket a `(@>)/2`, `(@<)/2`, `(@>=)/2`, `(@=<)/2`, beépített predikátumok segítségével hasonlítjuk össze. Például:

```
% rendbeszúr(Ys,X,Zs) :- Ys (szabványos term rendezés szerint) monoton
%          növekvő valódi Prolog listába X rendezett beszúrásával adódik Zs.
rendbeszúr([],X,[X]).
rendbeszúr([Y|Ys],X,Zs) :-
    ( X @=< Y -> Zs = [X,Y|Ys]
    ; Zs = [Y|Us], rendbeszúr(Ys,X,Us)
    ).
```

17.7.3. Termműveletek

Az eddigi példákban feltettük, hogy a programok által feldolgozandó termek szerkezetét jól ismerjük. Ez azonban a gyakorlatban nincs mindig így (például ha egy fájlból jövő bemenetet kell feldolgozni). Ilyenkor a termeket a termműveletek segítségével szedhetjük részekre, majd esetleg újra összerakjuk őket.

A struktúrák kezeléséhez a legfontosabbak a `functor/3` és az `arg/3` beépített eljárások.

A `functor(Term,Funktornév,Aritás)` hívás segítségével meghatározhatjuk vagy ellenőrizhetjük a *nonvar* `Term` funktor nevét és/vagy aritását. (Az állandók aritása nulla.) Ha a híváskor a `Term` változó, a `Funktornév` név, az `Aritás` pedig pozitív egész, a `Term`-ben létrejön a megfelelő összetett term, paramétereiben új változókkal.

Az `arg(I,Struktúra,Arg)` cél illeszti `Arg`-gal a `Struktúra` összetett term `I`-edik paraméterét, ahol `1 =< I =< Aritás`.

Így tehát *ki is nyerhetjük* vagy *ki is tölthetjük* egy összetett term paramétereit (általában miután egy `functor(Term,Funktornév,Aritás)` hívás segítségével meghatároztuk a term aritását, vagy éppen létrehoztuk a struktúrát):

```
% cserél(TO,X,Y,T) :-
%    T a TO másolata, kivéve, hogy X TO-beli előfordulásainak T-ben Y felel meg.
%    (Elégséges előfeltétel: TO és Y alaptermek; vagy T szabad változó és nem fordul
%    elő sem TO-ban sem Y-ban.)
cserél(TO,X,Y,T) :-
    ( TO == X -> T = Y
    ; compound(TO) ->
        functor(TO,F,N), functor(T,F,N),
        arg_csere(N,TO,X,Y,T)
    ; T = TO
    ).
% A program a következő oldalon folytatódik.
```

```
% Folytatás az előző oldalról.
arg_csere(N,T0,X,Y,T) :-
   ( N > 0 ->
      arg(N,T0,A), cserél(A,X,Y,B), arg(N,T,B),
      N1 is N-1, arg_csere(N1,T0,X,Y,T)
   ; true
   ).

% Teszt:
| ?- cserél(a(X,nil,b(nil,2,B,nil)),nil,[],T).
T = a(X,[],b([],2,B,[]))
```

Állandók szétszedésére és összerakására szolgálnak az `atom_codes(Név,Karakterlista)` *(atom=név)* és a `number_codes(Szám,Karakterlista)` predikátumok. (További termkezelők [Der 96, Spl 02]-ben.) Ezek segítségével tehát bármilyen szövegkezelő műveletet listakezelésre vezethetünk vissza. Például:

```
% aA(A,B) :- B az A név másolata, kivéve, hogy nagybetűssé tettük
%             (á kódja 0'á stb.).
aA(A,B) :-
   atom_codes(A,Cs), cC(Cs,Ds), atom_codes(B,Ds).

cC([],[]).
cC([C|Cs],[D|Ds]) :- bB(C,D), cC(Cs,Ds).

bB(C,D) :-
   ( 0'a=<C, C=<0'z -> D is C-0'a+0'A
   ; éÉ(C,D) -> true
   ; D = C
   ).

éÉ(0'á,0'Á). éÉ(0'é,0'É). éÉ(0'í,0'Í).
éÉ(0'ó,0'Ó). éÉ(0'ö,0'Ö). éÉ(0'ő,0'Ő).
éÉ(0'ú,0'Ú). éÉ(0'ü,0'Ü). éÉ(0'ű,0'Ű).

% Teszt:
| ?- aA(ó_mély_gyönyörű,Ő).
Ő = 'Ó_MÉLY_GYÖNYÖRŰ'
```

17.8. Prolog műveleti jelek

Említettük, hogy bizonyos (például az összehasonlító) predikátumok nevei közbevetett (infix) módon írhatók, és némely aritmetikai függvényszimbólumot is műveleti jelként jelölhetünk, noha mindezek – környezetfüggetlenül tekintve – csupán összetett termek függ-

vényszimbólumai. Ez pedig a Prolog *műveleti jel fogalmán* alapul. Eszerint bizonyos *neveket* műveleti jelként definiálhatunk.

1200 prioritási szint van. Az első szinten lévő műveleti jelek kötnek a legerősebben, míg az 1200-adik szinten lévők a leggyengébben. Egy műveleti jel a Prologban *nem csinál semmit*, hanem – megfelelően használva – a program olvashatóságát javító *jelölési lehetőség*. Például `a+b*c == +(a,*(b,c))`, a 400 prioritású * műveleti jel ugyanis erősebben köt, mint az 500 prioritású +. Egy term prioritása nulla, ha változó vagy állandó, zárójelbe tettük, listaként vagy a szokásos függvényjelöléssel – $f(t_1, \ldots, t_n)$ formában – írtuk le. Egy zárójelbe nem tett term prioritása fő funktorának prioritásával egyezik meg, amennyiben azt műveleti jelként írtuk le.

Egy *műveleti jel* lehet *közbevetett* (infix), *előtag* (prefix), vagy *utótag* (posztfix), aszerint, hogy két paramétere közé, egyetlen paramétere elé vagy mögé írjuk. Ugyanaz a név legfeljebb két műveleti jelhez tartozhat. Ilyenkor az egyik közbevetett, a másik előtag vagy utótag. Egy közbevetett műveleti jel lehet `yfx` (balról jobbra zárójeleződő, például +, -), `xfy` (jobbról balra zárójeleződő, például ->, ;), vagy `xfx` (mindenképpen zárójelezendő, például =, \=) módú. Itt `f` a függvényszimbólumot, azaz a műveleti jelet, `x` és `y` a paramétereit, vagyis az operandusait jelöli; y = yes: a megengedő oldal, ahol tehát a műveleti jellel kisebb vagy egyenlő prioritású kifejezés is lehet; míg x = áthúzás: ennek az oldalnak a prioritása kisebb kell legyen. Például `a+b+c==(a+b)+c`, `a+b` prioritása ugyanis 500, ami a 2. + jel bal oldalán megengedett, de `b+c` prioritása is 500, ami az 1. + jel jobb oldalán nem megengedett. Az előtag műveleti jelek `fx` vagy `fy`, az utótag műveleti jelek `xf` vagy `yf` módúak lehetnek a fenti logika szerint.

A műveleti jeleket az `op(Prioritás,Mód,Név)` utasítással hozhatjuk létre, illetve definiálhatjuk felül a fentiek szerint. (A `Név` itt egyetlen név, vagy valódi névlista is lehet.) Ha a `Prioritás`-nak `0`-át adunk meg, ez az adott nevű és módú műveleti jel törlését eredményezi. Ezt az utasítást általában *direktívaként* (17.1) írjuk le, és így műveleti jeleinket a program betöltésekor hozza létre. Például:

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200, fx, [ :-, ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op( 900, fy,  \+ ).
:- op( 700, xfx, [ =, \=, ==, \==, @<, @>, @=<, @>=,
                =.., is, =:=, =\=, <, >, =<, >= ]).
:- op( 500, yfx, [ +, -, /\, \/ ]).
:- op( 400, yfx, [ *, /, //, mod, rem, <<, >> ]).
:- op( 200, xfx, ** ).   :- op( 200, xfy, ^ ).
:- op( 200, fy, [ -, \ ]).
```

A fentiekben a Prolog ISO-szabványos, előre adott műveleti jeleit vezettük be. Mivel a vessző közbevetett műveleti jel (ilyenkor aposztrófok nélkül írható) és paraméter- illetve listaelem-elválasztó is, ezért a paraméterlisták és a szabványos Prolog listák elemeinek prioritása kisebb kell legyen, mint `1000`, hogy a listaelem-elválasztó vesszők és a közbevetett

műveleti jel vesszők össze ne keveredjenek egymással. (Ha tehát egy paraméter vagy listaelem prioritása elérné az ezret, zárójelbe kell tenni.)

Látható, hogy a mondatok felírásához szükséges jelek – mint a `:-` szimbólum és a vessző – is műveleti jelek. Műveleti jel továbbá a negációt jelölő `\+` és az `(If->Then;Else)` szerkezet összekötő jelei is. Környezetfüggetlenül tekintve tehát nem csak az atomi formulák paraméterei, hanem a Prolog célok, sőt, programjaink mondatai is termek. Ugyanaz a term adat, ha egy állításfej vagy egy predikátumhívás egy paramétere, cél, ha egy célsorozat eleme, és mondat, ha a program mondatainak sorában foglal helyet, szabályosan ponttal és helyközzel lezárva. A tagadást például – ha nem lenne beépített predikátum – a

```
\+ Cél :-
    ( Cél -> fail
    ; true
    ).
```

állítással definiálhatnánk. Ebben a `Cél` első előfordulása Prolog adatot, konkrétan formális paramétert, míg második előfordulása Prolog célt jelöl. (Vegyük észre, hogy a fenti esetben a feltételes szerkezetet nem lenne szükséges zárójelbe tenni, a program jó olvashatósága miatt mégis minden esetben ragaszkodunk a feltételes szerkezetek szokásos zárójelezéséhez és behúzásához.) A szabványos, funkcionális termjelöléssel élve a fenti állítást az alábbi formában is leírhatnánk (bár ezt legfeljebb a próba kedvéért javasoljuk):

```
:-(\+(Cél),;(->(Cél,fail),true)).
```

Az, hogy a Prologban – formai értelemben – nincs éles határvonal program és adat között, jelentősen megkönnyíti a tanuló programok, értelmező- és fordítóprogramok és általában a nyelvi feldolgozó programok készítését, főként ha a tárgy és a célnyelv is megfelel a Prolog termfogalmának [Sze 01]. (Ha ez valamelyik esetben nem így van, *logikai nyelvtanok* segítségével könnyíthetjük meg munkánkat [Der 93, Spl 02, Ste 94, Sze 01].)

17.9. A Prolog logikán kívüli predikátumai

Ezek valójában *csak* műveleti jelentéssel bíró eljárások. Deklaratív olvasatuk nincs. Kívül esnek a logikai programozási modellen, a gyakorlati programozás szempontjából mégis a Prolog nyelv nélkülözhetetlen elemei.

Ide tartoznak a mellékhatásos predikátumok, úgymint a környezet kezelő (interface) predikátumok különböző fajtái és a programmódosító predikátumok, amelyek tanuló programok készítését, általában a program keresési fájának alternatív ágai között az információ átadását támogatják [Kee 90, Ste 94]. Az ezen predikátumok által okozott mellékhatás gyakran éppen az adott predikátumhívás lényege, és sosem léptethető vissza.

A környezetkezelő predikátumok fajtái általában a programbetöltő, adatbeolvasó és kiíró, hálózati kapcsolatokat és más nyelvű programmokkal az adatforgalmat kezelő predikátumok [Spl 02, Sze 01]. Ez utóbbiak a különböző megvalósításokban tipikusan a C/C++, Java, BASIC, Pascal, Tcl/Tk stb. programnyelvekhez, valamilyen adatbáziskezelőhöz (ennek segítségével intelligens adatbáziskezelő felületet készíthetünk) valamilyen GUI (grafi-

kus felhasználói felületi) eszközhöz (például Tcl/Tk), illetve néhány Prolog-kiterjesztés kiegészítő (például funkcionális, objektumorientált, constraint kezelő, logikai nyelvtan stb.) összetevőjéhez biztosítják a kapcsolatot, és egyéb, a programfejlesztést támogató eszközökkel együtt a rendszerkönyvtárakban foglalnak helyet.

17.9.1. Programbetöltés

A programbetöltés adott forrásfájl vagy tárgykód a Prolog fejlesztői környezetbe való betöltését jelenti.

Egyetlen szabványos predikátuma az `ensure_loaded/1`, amelynek paramétere egyetlen fájlnév, vagy fájlnevek listája is lehet. Minden fájlra nézve ellenőrzi annak korát, és csak akkor tölti be, ha szükséges. Némely rendszerben csak *direktívaként* (17.1) [Der 96], általában azonban korlátozás nélkül hívható, és a betöltött kódot a memóriába fordítja.

A legtöbb rendszerben ezen kívül még rendelkezésünkre áll az értelmező módban való betöltés (`consult/1` vagy egyszerűen a betöltendő fájlok listája (17.2)) és a fordításos betöltés (`compile/1`) lehetősége, lefordított tárgykód előállítása, betöltése, speciális modul(fájl)kezelők, (`use_module/2` ... (17.12.2)) stb. [Spl 02, Sze 01]. Az értelmező módban betöltött predikátumok listázhatók (`listing/1`), nyomkövethetők (`trace/0`, `notrace/0`, stb.), míg a fordításos betöltés nagyságrenddel gyorsabb, optimalizált kódot eredményez [Spl 02, Sze 01].

17.9.2. Bemenet és kimenet

Csak a soros elérésű szövegfájlok kezelésének alapvető predikátumait tekintjük át. A szövegállományokat `read`, `write`, vagy `append` módban, mégpedig az `open(Fájl,Mód, Csatorna)` utasítással nyithatjuk meg. Ezután műveleteink a csatornára vonatkoznak, egészen annak lezárásáig (`close/1`). A szabványos bemeneti és kimeneti csatornákat nem kell és nem is lehet megnyitni és lezárni. Ezekre `user` néven hivatkozhatunk. Az I/O utasítások karakter vagy term szintűek [Der 96, Spl 02].

A `peek_code(F,C)` igaz, ha az `F` csatorna aktuális karakterének kódja `C` (megpróbálja illeszteni `C`-vel). A `get_code(F,C)` cél hasonló, de hatására az aktuális karakter a következő lesz. Az `at_end_of_stream(F)` igaz, ha az `F` csatornát már végigolvastuk (`F\=user`). A `put_code(F,C)` utasítás kiírja a `C` kódú karaktert, míg a `nl(F)` sort emel. Például:

```
% appf(F1,F2) :- F1 szövegfájl végére beszúrja F2-t.
appf(F1,F2) :-
   open(F1,append,A), open(F2,read,R), af(A,R).

af(A,R) :-
   get_code(R,C),
   ( C == -1 -> close(A), close(R) % R vége
   ; put_code(A,C), af(A,R)
   ).
```

A `read(F,T)` hívás egy teljes termet olvas be, és megpróbálja illeszteni `T`-vel. Mivel tetszőleges term tovább folytatható egy közbevetett műveleti jellel és egy kapcsolódó termmel, a beolvasandó termet az állításokhoz hasonlóan mindig egy pont és legalább egy elválasztó karakter zárja le:

```
| ?- read(user,X), read(user,Y).
|: 12.  a+b*c.
X = 12,  Y = a+b*c
```

A `write(F,T)` utasítás kiírja a `T` *tetszőleges* Prolog termet (akár teljes listát is), lezáró pont nélkül. A neveket (atomokat) azonban sosem teszi aposztrófok közé, hiszen ezek általában kiírandó üzeneteket ábrázolnak:

```
| ?- write(user,'A Pi értéke = '), read(user,Pi).
A Pi értéke = 3.14159265358979323846.
Pi = 3.141592653589793
```

Ha egy másik Prolog program által visszaolvasható kimenetet szeretnénk, a `writeq(F,T)` utasítást használjuk:

```
| ?- writeq(user,apja('Izsák','Jákób')),
     write(user,'.'), nl(user).
apja('Izsák','Jákób').
```

17.9.3. Önmódosító programok

Az eddigiekben kimondatlanul csak a statikus Prolog predikátumokkal foglalkoztunk. Egy predikátum ugyanis alapértelmezés szerint statikus, és kódja a programbetöltés után már nem változik. Dinamikusként is deklarálhatjuk, például a `:-dynamic(p/2).` deklaráció hatására a `p/2` predikátum dinamikus, azaz futási időben módosítható lesz; így programunk új állításokat is meg tud tanulni.

A deklarációnak mindig meg kell előznie a hivatkozott predikátum első állítását. Egy deklarált – és így létező – predikátumhoz nem szükségszerűen tartozik állítás. Ha egy létező predikátumhoz nem tartozik állítás, természetesen minden rá vonatkozó hívás meghiúsul, de ha a predikátum nem létezik, ez az `existence_error` kivételt (17.11) váltja ki.

A dinamikus predikátumokhoz újabb állításokat is adhatunk. Az `asserta(K)` utasítás hatására az állítás feje által meghatározott predikátum első állítása lesz a `K` állítás, míg az `assertz(K)` hatására a megfelelő predikátum utolsó állítása lesz.

Tegyük fel például, hogy az `e.pl` fájl tartalma a következő:

```
:- dynamic(elérhető/2).
elérhető(X,X).
```

és az alábbi módon dolgozzuk fel (feltesszük, hogy az `él/2` predikátum egy körmentes irányított gráfot határoz meg):

```
| ?- [e].
{consulting /home/at/pp/pny/lp/e.pl...}
{consulted /home/at/pp/pny/lp/e.pl in module user,
 10 msec 240 bytes}
yes
| ?- assertz((elérhető(X,Y):-él(X,Z),elérhető(Z,Y))).
yes
| ?- listing(elérhető/2). % elérhető/2 listázása}
elérhető(A, A).
elérhető(A, B) :-
        él(A, C),
        elérhető(C, B).
```

A dinamikus állítások törlése a `retract(ÁllításSéma)` utasítással lehetséges, ahol az `ÁllításSéma` *fej* része név vagy összetett term kell legyen, ami a `retract` utasítás által hivatkozott predikátumot azonosítja. A `retract(ÁllításSéma)` cél törli a hivatkozott predikátum első olyan állítását, amely illeszthető az `ÁllításSéma` termmel. Visszaléptetéssel akár az összes illeszkedő állítás törölhető:

```
| ?- retract((elérhető(X,Y):-Törzs)).
Y = X,   Törzs = true ? ;
Törzs = él(X,_A),elérhető(_A,Y) ? ;
no
```

Ennek példájára – ha nem volna beépített predikátum – a következőképpen írhatnánk le a `retractall(P)` utasítást, amely az összes, a `P` predikátumhívásra illeszkedő *fej* részű állítást törli:

```
retractall(P) :- retract((P:-_)), fail.
retractall(_P).
```

Ha egy nem létező predikátumra meghívjuk a fenti programmódosító utasítások valamelyikét (`asserta(Állítás)`, `assertz(Állítás)`, `retract(ÁllításSéma)`, illetve a `retractall(ÁllításFejSéma)`), akkor ez a nem létező predikátum dinamikus predikátumként létrejön:

```
% betölt(F) :- Beolvassuk az F fájl mondatait, és dinamikus predikátumokat
%             hozunk létre belőlük. Csak az állításokat vesszük figyelembe.
betölt(F) :- open(F,read,Cs), töltő_ciklus(Cs).

töltő_ciklus(Cs) :-
   read(Cs,Mondat),
   ( Mondat == end_of_file -> close(Cs) % Cs vége.
   ; Mondat = :-(_) -> töltő_ciklus(Cs)
   ; assertz(Mondat), töltő_ciklus(Cs)
   ).
```

Leggyakrabban a program futása során kiszámított, kikövetkeztetett információk, lemmák és negatív lemmák megőrzésére, újra való kiszámításuk elkerülésére használunk dinamikus predikátumokat.

Globális változóként való használatuk általában ellenjavallt. Ez egyrészt procedurális programozási stílust eredményezhet, másrészt nagyon költséges, mert például egy predikátum minden módosítása az első argumentum indexeléshez használt táblázatának újraépítését igényli, minden *assert* a beszúrandó állítást – függetlenül a benne tárolt adatszerkezetek méretétől – le kell másolja stb.

17.10. Prolog célok megoldásainak összegyűjtése

Egy Prolog célnak több megoldása is lehet. Mivel azonban ezek a keresési fa különböző ágain állnak elő, a visszalépéses keresés miatt egyszerre sosem állnak rendelkezésünkre. A célok megoldásait összegyűjtő beépített predikátumok közül [Der 96, Kee 90, Spl 02, Ste 94, Sze 01] talán a legfontosabb és legegyszerűbb a `findall(Mit,Honnan,Hová)` utasítás. Itt a `Honnan` egy tetszőleges cél, illetve zárójelezett célsorozat. Ennek a megoldásait a `Mit` term szerint összeállítva a `Hová` listába gyűjtjük össze.

Ha például – a 17.1.2. részben ismertetett módon – adott a `szülője/2` reláció, kiszámíthatjuk bárki szülei, illetve gyermekei listáját:

```
szülei(X,Ys) :- findall(Y,szülője(Y,X),Ys).
gyermekei(X,Ys) :- findall(Y,szülője(X,Y),Ys).

% Teszt:
| ?- szülei('Izsák',L).
L = ['Sára','Ábrahám']
| ?- gyermekei('Izsák',L).
L = ['Jákób','Ézsaú']
```

A `findall(Mit,Honnan,Hová)` cél a `Honnan` cél(sorozat) megoldásait abban a sorrendben gyűjti össze, amelyben azokat megkapjuk. Ha leírjuk a (fél)testvér relációt,

```
testvér(X,Y) :- szülője(Z,X), szülője(Z,Y), X\==Y.
```

akkor a féltestvérek csak egy, míg a teljes testvérek két szülőn keresztül testvérek. Ha ezek után egy *findall* utasítás segítségével leírjuk a

```
% testvérei(X,Ys) :- X testvérei Ys elemei.
testvérei(X,Ys) :- findall(Y,testvér(X,Y),Ys).
```

relációt, akkor a teljes testvérek megkettőzve szerepelnek az `Ys` listán:

```
| ?- testvérei('Izsák',L).
L = ['Ismáel','Ismeretlen']
| ?- testvérei('Jákób',L).
L = ['Ézsaú','Ézsaú']
```

Ha ez nem kívánatos, a legtöbb Prolog-megvalósításban rendelkezésre áll a `sort(Xs,Ys)` predikátum, amely az `Xs` listát rendezi szigorúan monoton növekvően a `(@<)/2` reláció (17.7.2) szerint az `Ys` listába, az esetleges elemkettőzéseket értelemszerűen megszüntetve. Ennek segítségével könnyen kitalálhatjuk a

```
% gyűjt(Mit,Honnan,Mibe) :-
%    Honnan megoldásainak halmaza Mit szerint Mibe.}
gyűjt(Mit,Honnan,Mibe) :-
   findall(Mit,Honnan,Hová), sort(Hová,Mibe).
```

predikátumot. Ezt használva a `testvérei/2`-ben a `findall/3` helyett a megfelelő megoldást kapjuk:

```
% testvérei(X,Ys) :- X testvéreinek halmaza Ys.}
testvérei(X,Ys) :- gyűjt(Y,testvér(X,Y),Ys).
```

Ez tehát már nem ad megkettőzött testvéreket:

```
| ?- testvérei('Jákób',L).
L = ['Ézsaú']
```

17.11. Kivételkezelés a Prologban

A hiba- és kivételkezelés (exception handling) minden modern programozási nyelvnek, így az ISO Prolognak is fontos részét képezi.

A tiszta Prologban egy predikátumhívás – feltéve, hogy keresési tere véges – meghiúsul vagy sikeresen lefut, esetleg választási pontot hagyva maga után. A beépített eljárások némelyike azonban bizonyos bemenő értékekre nem értelmezhető. Ilyen például az `arg(I,T,A)` hívás, ha `T` egyszerű term vagy `I` negatív; az `X is Exp` cél, ha `Exp` nem aritmetikai kifejezés; vagy a `read(X)` utasítás, ha formailag hibás termet kellene beolvasnia. Ilyenkor – más magasszintű nyelvekhez hasonlóan – az aktuális eljáráshívás a megfelelő *kivételt* (exception) váltja ki, és a hívás – sikeresség vagy meghiúsulás helyett – ezzel ér véget.

A kivételt mindig egy *állandó* vagy *összetett* term (nonvar term) jelöli, és mesterségesen is kiváltható a `throw(`*Kivétel*`)` utasítás segítségével, ahol *Kivétel* a kivételt jelölő nonvar term[14].

A kezeletlen kivételek mint futási hibák lépnek fel. Ezeket a Prolog fejlesztői környezet – a megfelelő hibaüzenet kiírásával – saját párbeszédciklusa szintjén kezeli.

A kivételkezelés eszköze a `catch(`*Cél*`,`*KivételMinta*`,`*KezelőCél*`)` utasítás[15].

A *catch* utasítások pontosan azon kivételek kezelésére szolgálnak, amelyek az első argumentumban levő *Cél* kiértékelése során lépnek fel. A *catch* utasítások első argumentumát ezért *védett* (protected) argumentumnak nevezzük.

[14] throw = dob, vet, hajít.

[15] catch = elfog, elkap, megragad, megcsíp.

A *catch* utasítás tehát először meghívja az első, védett argumentumát paraméterező *Cél* célsorozatot.

Amennyiben a *Cél* kiértékelése során nem lép fel kivétel, vagy ezt (ezeket) a *Cél* maga kezeli, akkor teljesen a *Cél* működése határozza meg a *catch* hívást.

Ha azonban a *Cél* kiértékelése kezeletlen kivételt vált ki (nevezzük ezt *E*-nek), akkor először is készítünk *E*-ről egy friss másolatot[16], majd a *Cél* kiértékelése során végrehajtott összes változóillesztést töröljük, az itt létrehozott választási pontokkal, a hívási verembe betöltött hívási keretekkel, és általában minden visszaléptethető hatással együtt.

Ezután *E* friss másolatát és a *KivételMinta* termet illesztjük.

Ha az illesztés sikeres, akkor a *KezelőCél* célsorozat határozza meg a *catch* utasítás további viselkedését.

Ha azonban *E* friss másolata és a *Kivételminta* illesztése meghiúsul, akkor a környezet szempontjából úgy tekinthető, hogy a *catch* hívás váltotta ki az *E* kivételt. Ilyenkor azt mondjuk, hogy a *catch* utasítás *tovább propagálja* a kivételt.

Végül, ha *E* friss másolatának a *Kivételmintával* való illesztése ugyan sikeres, de a *KezelőCél* maga is kivált egy kezeletlen *H* kivételt, akkor a környezet szempontjából úgy tekinthető, hogy a *catch* utasítás váltotta ki a *H* kivételt. Ilyenkor is tovább propagálja tehát a kivételt.

A beépített eljárások mindig `error(IsoErr,ImpErr)` alakú kivételeket váltanak ki, ahol `IsoErr` a szabvány hibatermje, míg `ImpErr` rendszerfüggő:

```
| ?- catch( X is a, error(IsoErr,ImpErr), true ).
ImpErr = domain_error(_A is a,2,expression,a),
IsoErr = type_error(evaluable,a/0)
```

Az `olvas(Term)` cél beolvas egy tetszőleges termet a szabványos bemenetről (ez alapesetben a billentyűzet), és az aktuális bejövő sor maradékát eldobja. Ha a bejövő term szintaktikusan hibás, kiírja az ISO szabványnak megfelelő hibatermet, majd újra olvas.

```
olvas(Term) :-
   catch( ( read(user,Term), skip_line(user) ) ,
          error(Hiba,_), ( ír(Hiba), olvas(Term) )
        ).
ír(Term) :- writeq(user,Term), nl(user).

| ?- olvas(Term).
|: a*(b+ .
syntax_error('. cannot start an expression')
|: a*(b+c.
syntax_error(') or operator expected')
|: a*(b+c).
Term = a*(b+c)
```

[16] friss másolat = fresh copy : olyan másolat, amelyben a változók helyett módszeresen új, eddig nem használt változókat vezetünk be. Egy változó különböző előfordulásai helyett természetesen ugyanazt az új változót vezetjük be, míg különböző változók helyett különbözőket.

17.12. Prolog modulrendszerek

Mivel a Prolog modulrendszerét még nem szabványosították, ezt megvalósításfüggően tárgyaljuk, fejezetünk bevezetője (637. oldal) szellemében a SICStus Prologra összpontosítva [Spl 02]. További összehasonlító elemzés található [Sze 01]-ben. A modulrendszer lehet *név* vagy *eljárás (predikátum)* alapú.

17.12.1. A név alapú modulrendszer

A név alapú modulrendszerben minden név (atom) alapértelmezés szerint lokális. Például az `alma` név alapértelmezés szerint minden modulban különböző. A lokalitás feloldása rendszerfüggő. Az *LPA Prolog Professional*-ben például kétszintű a modulrendszer[17]. Az alapértelmezett gyökérmodul alatt vannak az esetleges felhasználói modulok. Ezek a neveket a gyökérmodulba exportálhatják, illetve innét importálhatják. A kifejezetten modulba nem zárt Prolog kód is alapértelmezés szerint a gyökérmodulba töltődik. Ez a modulrendszer egyszerű, de ha egy nevet exportálunk illetve importálunk, akkor minden lehetséges aritással, mint adatnevet és mint predikátumnevet is exportáltuk illetve importáltuk. Az adatnevek lokalitása kényelmetlen, és nem világos, mi a különbség módszertanilag egy *szám* (ami globális) és egy *név* (ami alapértelmezés szerint lokális) között.

17.12.2. Az eljárás alapú modulrendszer

Az eljárás alapú modulrendszerben – amit a SICStus Prolog leírása [Spl 02] alapján tárgyalunk, de használatos még például az SWI és Quintus Prolog rendszerekben is [Sze 01] – a fentiekkel szemben a nevek, és általában minden term globális, a predikátumok (eljárások) azonban forrásmoduljukra nézve lokálisak. Alapértelmezett a `user` modul: ha nem használjuk a modulrendszert, minden kód ebbe töltődik. Ennek ellenére minden modul azonos szinten van.

A modulfájlok első mondata rendesen egy moduldeklaráció

`:- module(` ⟨*modulnév*⟩`,` ⟨*exportlista*⟩ `).`

formában, ahol a ⟨*modulnév*⟩ Prolog atom, az ⟨*exportlista*⟩ pedig *név/aritás* alakú predikátum-specifikációk valódi listája: az itt felsorolt eljárások lesznek a nevezett modul nyilvános predikátumai. Ezek lehetnek az adott modulban definiált, vagy ide importált predikátumok is. Ezek importálhatók egy másik modulba, például a

`use_module(` ⟨*modulfájlnév*⟩`,` ⟨*importlista*⟩ `)`

beépített eljárás segítségével, amit általában *direktívaként* (17.1) hívunk meg, és szükség esetén a fájl betöltését is elvégzi.

Tegyük fel most, hogy adott a `graph.pl` modulfájl, benne a `gráf` modullal, amely – `él/2` nyilvános predikátumában – egy irányított gráf leírását tartalmazza, vagyis az `él(X,Y)` pontosan akkor igaz, ha az `(X,Y)` a gráf egy éle. Feltesszük továbbá, hogy az `él(X,Y)` cél

[17] Ez az MProlog modulrendszerének [Sze 01] továbbfejlesztése.

a gráf éleinek ellenőrzésére és visszadására is alkalmas. A legegyszerűbb ilyen gráfleírások Prolog tényekként sorolják fel az éleket:

```
él(a,b).  él(b,c).  él(b,d).
él(c,a).  él(c,e).  él(d,e).
```

Írjunk most egy modult, amely egyetlen nyilvános predikátumával körmentes utakat keres a fenti gráfon:

```
:- module( keres, [ út/3 ] ).

:- use_module( graph, [él/2] ).
:- use_module( library(lists), [member/2,reverse/2] ).

% út(A,Z,Út) :-
%    Út egy A → Z körmentes út (csúcsok listája).
út(A,Z,Út) :- útja(A,[],Z,Út).

% útja(A,Ősök,Z,Út) :-
%    Az Ősök csúcslista fordítottjának és egy A → Z útvonalnak a konkatenáltja
%    az Út körmentes út.
útja(A,Ősök,A,Út) :- reverse([A|Ősök],Út).
útja(A,Ősök,Z,Út) :-
   él(A,B),                  % él egy közbenső csúcsba
   B\=A,                     % nem hurokél
   \+member(B,Ősök),         % nem ciklus
   útja(B,[A|Ősök],Z,Út).
```

Keressünk utakat. Figyeljük meg, hogy a megoldások között az első a *tetszőleges, de egyetlen csúcsból álló út*, mint általános megoldás (vö. 641. oldal):

```
| ?- út(_A,_Z,Út).
Út = [_A] ? ; Út = [a,b] ? ;
Út = [a,b,c] ? ; Út = [a,b,c,e] ? ;
Út = [a,b,d] ? ; Út = [a,b,d,e] ? ;
Út = [b,c] ? <Enter> % Elég volt!
yes
```

Bizonyos alkalmazásokban szükséges lehet, hogy a gráfot leíró állományt a `keres` modulnak kívülről adhassuk meg. Ehhez elég modulunk első két mondatát megváltoztatni:

```
:- module( keres, [ új_gráf/1, út/3 ] ).
új_gráf(Állomány) :- use_module( Állomány, [él/2] ).
```

Inicializáljuk most modulunkat a fenti gráffal, majd gyűjtsük össze az összes körmentes utat (bárhonnét bárhová), és írassuk ki az eredményt (kiíratáskor a változók belső neve jelenik meg):

```
| ?- új_gráf('../lp/graph').
{module gráf imported into keres}
yes
| ?- findall(Út,út(_,_,Út),_Utak), write(_Utak).
[[_759],[a,b],[a,b,c],[a,b,c,e],[a,b,d],[a,b,d,e],
 [b,c],[b,c,a],[b,c,e],[b,d],[b,d,e],
 [c,a],[c,a,b],[c,a,b,d],[c,a,b,d,e],[c,e],[d,e]]
true ? ; no
```

17.12.3. A forrásmodul megváltoztatása

A fent vázolt, szigorú modulrendszer alapvető problémája, hogy a modulok lokális predikátumai nem tesztelhetők közvetlenül, azaz tesztelésükhöz a kód megváltoztatása lenne szükséges. A SICStusban ezért – és még később részletezendő okokból – lehetséges a célok forrásmoduljának megváltoztatása: a célok mindig a megfelelő modulnévvel minősíthetők,

⟨*modulnév*⟩:⟨*cél*⟩

formában. Például a `keres` modul (676. oldal) `útja/4` predikátumát közvetlenül tesztelhetjük (annak ellenére, hogy ez a modulnak *nem* nyilvános predikátuma):

```
| ?- keres:útja(d,[b,a],E,Út).
E = d,   Út = [a,b,d] ? ;
E = e,   Út = [a,b,d,e] ? ;
no
```

Értelmező módban való betöltés (17.9.1) esetén ki is listázhatjuk a betöltött alakot a `listing(keres:útja/4)` utasítás segítségével.

A SICStus modulrendszere tehát *nem* a predikátumok elrejtésére, hanem *csak* véletlen ütközéseik megelőzésére szolgál.

Míg a fenti probléma általában mindenféle Prolog modulrendszerre jellemző, a most következő az eljárás alapú modulrendszer sajátja, és a metapredikátumokkal kapcsolatos. Tegyük fel például, hogy szeretnénk egy modul nyilvános szolgáltatásává tenni a `mindre1(P,Q)` predikátumot, amely `P` minden megoldásához végrehajtja a `Q` utasítást, `Q` első megoldására szorítkozva. Kódja megadható

```
% mindre1(P,Q) :- P minden megoldására Q is megoldható azaz P implikálja Q-t.
mindre1(P,Q) :- \+ ( P, \+Q ).
```

formában. Használatára egy egyszerű példa a következő:

```
| ?- mindre1(member(X,[1,2,3]),(write(X),write(' '))).
1 2 3
```

Ha most a `mindre1/2` az `m` modul nyilvános predikátuma, amit a `h` modulból hívtunk meg, a `P` és `Q` formális paramétereket paraméterező célsorozatok – mint fent a `member(X,[1,2,3])` – az `m` modulban hívódnak meg, ahol lehet, hogy nem léteznek, vagy

ami még rosszabb, esetleg egészen más a definíciójuk, mint a h modulban. A probléma egy lehetséges megoldása, hogy a hívást paraméterező célsorozatokat minősítjük a hívó modul nevével. Ez azonban nem túl elegáns, és ráadásul hibaveszélyes megoldás, hiszen a hívó modul nevét programfejlesztés közben megváltoztathatjuk. Ezért a predikátumok egyes argumentumaira *modulnév-kiterjesztés* deklarálható. A kiterjesztendő paramétereket ezekben a *metapredikátum-deklarációkban* kettősponttal (vagy egész számmal), a többieket tetszőleges más névvel stb. jelöljük [Spl 02]:

```
:- module( m, [ mindre1/2, gyűjt/3, be/1, be/0 ] ).

:- meta_predicate mindre1(:,:), gyűjt(?,:,?), be(:).
:- dynamic aktuális_fájl/1.

mindre1(P,Q) :- \+ ( P, \+Q ).

gyűjt(Mit,Honnan,Mibe) :-
   findall(Mit,Honnan,Hová), sort(Hová,Mibe).

be(F) :- compile(F), legyen_aktuális(F).

be :- aktuális_fájl(F), compile(F).

legyen_aktuális(F) :-
   retractall(aktuális_fájl(_)),
   asserta(aktuális_fájl(F)).
```

Ez azt jelenti, hogy a `mindre1(C,D)` hívás mindkét paraméterét a fordító automatikusan minősíti a hívó modul nevével, a `gyűjt(M,C,L)` hívásnak (lásd a 673. oldalt) azonban csak a második paraméterét. A két hívás tehát a modulnév-kiterjesztés után `mindre1(h:C,h:D)`, illetve `gyűjt(M,h:C,L)` lesz.

Modulnév-kiterjesztést igényelhetnek még az állításokat és a programfájlneveket (valamint a *blackboard* kulcsokat [Spl 02]) továbbító paraméterek. A SICStus beépített eljárásait ugyanis a `prolog` modulban definiálták, mint annak nyilvános predikátumait. Ezeket minden modul automatikusan látja. A programmódosító (`asserta/1`, `assertz/1`, `retract/1` stb.) predikátumoknak pedig tudniuk kell, hogy melyik modul predikátumait módosítsák. Hasonlóképpen, a programbetöltő (`consult/1`, `compile/1`, `use_module/2`, stb.) predikátumoknak is tudniuk kell, melyik modulba töltsék be a hívásukat paraméterező programfájlt, illetve modulfájl betöltése esetén melyik modulba importálják annak nyilvános predikátumait [Spl 02]. Ez a modul mindegyik esetben alapértelmezés szerint a beépített eljárást hívó forrásmodul, ezért mindegyik esetben elegendő, hogy a beépített eljárások megfelelő argumentumaira a `prolog` modulban modulnév-kiterjesztést deklaráltak [Spl 02]. Így a vonatkozó predikátumhívásokra a fordító modulnév-kiterjesztést végez, és a meghívott eljárások megkapják a szükséges információt. Általában, ha a SICStus egy beépített predikátumának egy argumentuma modulnév-kiterjesztést igényel (lásd [Spl 02, Sze 01]) akkor a hívó predikátum megfelelő formális paraméterére is modulnév-kiterjesztést célszerű deklarálni.

Tegyük fel például, hogy olyan fordítóutasítást szeretnénk írni, amely emlékszik az utoljára lefordított állomány nevére, és alapértelmezés szerint ezt fordítja újra. Erre vonatkoznak a fenti példában a `be/[0,1]` predikátumok. (A feladattal kapcsolatban tudnunk kell, hogy a programbetöltő predikátumok alapértelmezés szerint ugyan a hívó modulba töltik be a programfájlt, illetve modulfájl betöltése esetén ide importálják ennek nyilvános predikátumait; de ha a programfájl nevét egy modulnévvel minősítjük a szokásos *modulnév:fájlnév* alakban, akkor a betöltés, illetve importálás a minősítő modulba történik.) A `be(F)` cél, mint látjuk, az `F` fájl lefordítása után feljegyzi, azaz aktuálissá teszi annak nevét, és a `be` cél ezután az `F` fájlt fordítja újra. A `be/[0,1]` predikátumokkal lefordított fájlokra az `m` modulban hajtjuk végre a `compile(F)` hívást. A megfelelő metapredikátum-deklaráció nélkül ezért a `be` utasítások az `m` modulba fordítanának, kivéve, ha a `be/1` predikátumot `be(h:f)` alakban hívjuk meg. A deklaráció hatására azonban a `be(F)` cél `F` aktuális paramétere automatikusan minősítve lesz a cél forrásmoduljának `h` nevével, és a cél végül `be(h:F)` alakban hívódik meg. Ha tehát `F` egy programfájl neve, a `h` modulba fordítunk. Ha viszont például `F = k:f`, a cél a modulnév-kiterjesztés után `be(h:k:f)` lesz. Ilyenkor a belső minősítés érvényes, vagyis a `k` modulba fordítunk. Az automatikus modulnév-kiterjesztés tehát az explicit minősítést nem bírálja felül. (Vegyük észre, hogy a fenti példában nem csak a fájlnevet, hanem annak modulnév-előtagját is feljegyezzük. Így a legközelebbi `be` hívás annak forrásmoduljától függetlenül pontosan az utolsó `be(F)` hívás hatását ismétli meg.)

Befejezésül megjegyezzük, hogy a forrásmodul explicit megváltoztatásának lehetőségével a forrásprogramon belül nem tanácsos gyakran élni, mert ez programunk modulrendszerének lerombolásához vezethet.

17.13. Kitekintés

Ebben a rövid bevezetőben természetesen nem nyílt mód a logikai programozás átfogó tárgyalására, de még a Prolog nyelvnek és/vagy programozási módszereinek részletes ismertetésére sem. Az alábbi információk, reméljük, megfelelően segítik azt, aki a témában jobban el kíván mélyülni.

17.13.1. Ajánlott irodalom

A logikai alapok olvasmányos forrásai [Fek 90, Nil 82]. Egy az itt tárgyaltnál általánosabb megközelítést mutat be Kowalsky [Kow 79]. A logikai programozás elméleti vonatkozásait tárgyalja Lloyd [Llo 87]. A Prolog programozás módszertana szempontjából alapvető, kezdőknek és középhaladóknak Sterling és Shapiro könyve [Ste 94], haladóknak O'Keefe műve [Kee 90]. Az ISO Prolog szabvány megismeréséhez ajánljuk Deransart, Ed-Dbali és Cervoni közös alkotását [Der 96]. Warren absztrakt gépét, ami a Prolog megvalósításának szokásos eszköze, és az ezzel kapcsolatos problémákat jól érthetően tárgyalja [Kac 91]. Az ebben a fejezetben használt SICStus Prolog-változat felhasználói kézikönyve [Spl 02]. Több mint 500, hálózatról letölthető gyakorlati Prolog alkalmazást tartalmaz a Prolog 1000 adatbázis (Prolog 1000 database).

17.13.2. Klasszikus LP-kiterjesztések

A logikai programozás nem maradt elszigetelt tudományág. A másik alapvető deklaratív programozási megközelítéssel, a funkcionális programozással való egyesítésére kezdettől fogva történtek kísérletek, amelyek jól működő rendszerekhez vezettek [DeG 86, Han 94]. Ezen egyesített, funkcionális-logikai nyelveknek a gyakorlati programozásban való használatát elsősorban talán egy (de facto) szabvány és a megfelelő képzettségű programozók hiánya akadályozza.

Hasonló problémák – meggyőződésünk szerint – a Prolog nyelv széleskörű használatának akadályai is, hiszen a mai napig csak a nyelv magját sikerült szabványosítani, és talán még a modulrendszer szabványának késlekedésénél is nagyobb probléma a szabványos C/C++, Java, adatbáziskezelő, grafikus stb. felületek hiánya. Az, hogy a szükséges felületek a komolyabb Prolog rendszerek részei, csekély vigasz, ha tudjuk, hogy a gyakorlati alkalmazók nem szívesen függnek egy konkrét megvalósítástól, hisz az különböző okokból bármikor kiszorulhat a piacról. Nem véletlen, hogy a Prolog-megvalósítások száma viszonylag magas, de talán fél tucat – évtizedesnél régebbi – Prolog-fejlesztő *van jelen* az üzleti életben. Az, hogy a Prolog programok lassúak, inkább költött, mint valós érvelés. Egyrészt, meglepő módon még mindig a hatvanas évek amerikai próbálkozásainak kudarca kísért [Ste 94], másrészt a túlterhelt programozóknak a Prolog nélkül is épp elég követni a változásokat. Egy modern Prolog rendszer, teljesítménye alapján a szégyen veszélye nélkül összemérhető mondjuk egy Java- vagy Lisp-megvalósítással. Természetesen egy-egy Prolog-hacker sokat tehet a nyelv rossz hírének fenntartásáért.

Érdekes kísérletek a Prolog objektumorientált kiterjesztései [Mos 94, Spl 02]. Jól alkalmazhatók például szimulációs feladatokban és szakértői rendszerek építésénél. Hasonló elvű, de kimondottan szakértői keretrendszer Phil Vasey *flex* nevű, frame alapú szakértői keretrendszere, amely az *LPA* és a *Quintus* Prologok kiterjesztéseiként áll rendelkezésre. Jelentős üzleti alkalmazásairól tudunk, intelligens lekérdező és nyilvántartó felületek, diagnosztizáló, ütemező stb. rendszerek készítésére használják.

Majdnem minden modern Prolog-megvalósításban rendelkezésünkre állnak – a nyelv beépített kiterjesztéseként – a DCG nyelvtanok (Definit Clause Grammars). Ezek a logikai nyelvtanok elméleti szempontból a nullás típusú nyelvtanok, gyakorlati szempontból az attribútum nyelvtanok kifejezőerejével bírnak. Alkalmazásuk nagyban megkönnyíti nyelvi elemzők, szövegelőállítók, fordítóprogramok, formális és élő nyelvű programfelületek stb. készítését, amelyek a Prolog nyelvhez amúgy is jól illeszkedő és alapvető alkalmazási területeket jelentenek. Még a hagyományos listakezelési feladatok egy része is elegánsan és hatékony kóddal kezelhető a segítségükkel.

17.13.3. Az ötödik generációs számítógépek és programjaik

Talán a legismertebb, a Prolog alkalmazásaként elhíresült kezdeményezés az az 1981-ben elindított japán kutatási program, amely új típusú, úgynevezett *ötödik generációs*, logikai alapú gépi nyelven programozható sokprocesszoros hardver, és erre épülő intelligens számítógépek, robotok kifejlesztését tűzte maga elé. A fejlesztés alapnyelve egy idő után valójában

egyfajta GHC (Guarded Horn Clause) nyelv lett, amely egyrészről támogatja ugyan a részcélok párhuzamos végrehajtását, de nem támogatja a visszalépéses keresést.

Japán ugyan 1993-ban bejelentette a kutatás sikeres lezárását, de az valójában nem hozta meg a remélt eredményeket. Egyrészről nyilvánvalóvá lett, hogy az általános célú hardverek mai fejlődési üteme mellett nem érdemes egyedi hardvert készíteni (mire elkészül, elavul), másrészt, a visszalépéses keresésről való lemondással az LP éppen az intelligens keresést igénylő alkalmazásokban veszíti el előnyét a funkcionális programozással szemben.

17.13.4. Újabb irányzatok

A logikai programozás újabb irányzatai közül ígéretes fejlesztést jelent a Mercury nyelv. Ez egy tiszta LP nyelv, amit kifejezetten nagy programok hatékony fejlesztésére dolgoztak ki. Még a környezetkezelő predikátumokat is logikai alapon oldották meg. Modulrendszere, erős, polimorfikus típusrendszere, erős módrendszere (egy hívásban egy paraméter csak tisztán *in* vagy *out* módú lehet), és a kötelezően megadandó determinizmus-információk alapján fordítója kiemelkedően hatékony tárgykódot hoz létre. A típusosság növeli a tárgykód biztonságát, a Prologhoz viszonyítva több programozói hibára derül már fordítási időben fény. Hátránya, hogy a Prologhoz képest megszorított LP modellre épül, ami megnehezíti a programírást, jobban behatárolja a programozó lehetőségeit, és meggátolja néhány általánosan elfogadott, hatékony LP módszer alkalmazását, mint például az eredmény fokozatos közelítéssel való előállítása (17.4.2). Ezek a hiányosságok a nyelvnek és programozási módszertanának további fejlesztését kívánják.

Erőteljes kutatási irányzat a párhuzamos LP visszalépéses kereséssel párosított változata. A két alapvető megközelítés a *vagy-párhuzamos*, amikor tehát a keresési fa egyes ágait derítjük fel, amennyire lehet párhuzamosan, valamint az *és-párhuzamos*, amikor a célsorozat egyes részcéljainak megoldását párhuzamosítják. Előbbire példa az *Aurora* LP rendszer (Warren, Gigalips terv), és a SICStus Prolog *Muse* változata, a kettő egyesítésére Warren *Andorra-I* rendszere.

A CLP (Constraint Logic Programming) irányzat a nyolcvanas évek végétől fejlődött ki, és napjainkra a logikai programozás legsikeresebb irányzatává nőtte ki magát [Hen 86, Mar 98, Sze 01, Spl 02]. Elsősorban a legkülönfélébb optimalizációs feladatok megoldására használják. Lényege, hogy a feladatleírás deklarativitását és a keresés intelligenciáját úgy növeli, hogy egy-egy speciális feladatkörben felhasználja más tudományterületek megoldó módszereit. A CLP program különös alapelemei olyan célok, amelyek démonként viselkednek, előre adott értelmezési tartományokkal; és változóik behelyettesítésekor, illetve változóik értelmezési tartományának szűkülésekor egymással kölcsönhatásban szűkítik tovább azok értelmezési tartományát, és így a keresési teret. Mindez ráépül a hagyományos, visszalépéses keresésre. Ha egy változó értelmezési tartománya üressé válik, ez is visszalépést eredményez. Ilyen módon a SICStus Prolog is háromféle különböző CLP-kiterjesztést egyesít magában [Spl 02]. Nagyban növelte a CLP népszerűségét, hogy a francia Ilog cég CLP rendszerét a C++ nyelvre építette.

17.14. Összefoglalás

Napjainkra, a sikerek és kudarcok eredőjeként, a logikai programozás helye az összetett, több (egymással kapcsolatot tartó) aktív objektumot tömörítő rendszerek elemeinek *gondolkodó összetevőiben* látszik körvonalazódni. Segítségével elegánsan és hatékonyan fejleszthetők intelligens adatbázis-felületek, keresést, akár hálózati keresést igénylő rendszerek, különböző fajta, nyelvi feldolgozást, élő nyelvi felületet, fordítást igénylő programok, szakértői rendszerek, természetes intelligenciánkat fejlesztő játékprogramok, szimbolikus számításokat igénylő, képlet-átalakító, illetve egyszerűsítő programok, egyenlet és egyenlőtlenségrendszer-megoldók, optimalizáló programok stb.

A logikai programozás a problémáknak a hagyományos programozási nyelvektől eltérő megközelítését kívánja, de éppen ezért új, elegáns megoldásokhoz segít bennünket.

Feladatok

1. Alkalmazzuk a Prolog gép célredukciós algoritmusát (17.5.1. pont) a 17.4.2. részben föltett `append(Xs,Ys,[1,2,3])` Prolog kérdésre.

2. Írjunk listakezelő predikátumokat egy lista kezdőszeleteinek, zárószeleteinek, folytonos részlistáinak és részsorozatainak előállítására. A predikátumok adott listakezdőszelet stb. esetén a kérdéses tulajdonság ellenőrzésére is alkalmasak legyenek. Figyeljük meg, hogy a predikátumok megírásakor mikor milyen programozási módszert (az eredmény fokozatos közelítése, akkumulátor módszer stb.) alkalmaztunk. Vizsgáljuk meg a kapcsolódó predikátumhívásokra a keresési fák végességét.

3. Tekintsük a bináris fáknak a 17.3. rész végén tárgyalt ábrázolását. Írjuk meg ezzel az ábrázolással a keresőfák szokásos műveleit (Üres fa létrehozása illetve ellenőrzése, adott (kulcsú) elem keresése, beszúrása és törlése.)

4. Implementáljuk az általunk ismert listarendező algoritmusokat Prologban. Ügyeljünk arra, hogy az algoritmusok aszimptotikus hatékonysága ne romoljon.

5. Tekintsük a `member/2` predikátumot (17.5.3, 655. oldal). Adjunk alternatív, elégséges feltételeket a `member(Y,Ys)` cél NSTO tulajdonságához. (Vegyük észre, hogy ha egy állításfej kettőzött változót tartalmaz, de a rá vonatkozó célban az állításfej minden változója legfeljebb egy előfordulásának nem alapterm felel meg, a cél NSTO. Megfigyelhetjük azt is, hogy amennyiben `member(Y,Ys)` célban `Ys` valódi lista, akkor az adott esetben az STO tulajdonság csak `Y`, és az `Ys` egy elemének illesztésével jöhet létre.)

6. Adjunk elégséges feltételeket az `append(Xs,Ys,Zs)` (17.4.2), a `rev_app(Xs,Ys,Zs)` (17.4.3) és a `reverse(Xs,Ys)` (17.4.4) célok NSTO tulajdonságához.

7. A 17.5.3. rész végén (656. oldal) felsorolt beépített Prolog predikátumok STO módon is meghívhatók, mert ezekben a program bizonyos termjeit illesztjük egymással. Fogalmazzunk meg elégséges feltételeket a megfelelő célok NSTO tulajdonságára.

8. Vizsgáljuk meg a (17.6.1) rész predikátumaira hivatkozó `nincs_benne(Xs,Y)`, `nonmember(X,Xs)`, `metszet(Xs,Ys,Zs)` és `unió(Xs,Ys,Zs)` célokat a keresési tér végessége szempontjából. Adjunk erre elégséges feltételeket. Fogalmazzunk meg elégséges feltételeket a fenti célok NSTO tulajdonságára is.

9. Keressük meg a `setof/3` ISO Prolog predikátum leírását a rendelkezésre álló szakirodalomban [Der 96, Spl 02, Ste 94, Sze 01]. Megállapíthatjuk, hogy hasonló a 17.10. rész `gyűjt/3` predikátumához, de kifejezőereje nagyobb. (A szerző mérései szerint azonban kevésbé hatékony.)

10. Tegyük fel, hogy (17.10) `gyűjt/3` eljárása egy modul nyilvános predikátuma. Adjuk meg az ehhez szükséges metapredikátum-deklarációt.

11. Próbáljuk meg egy sorkezelő modulban megvalósítani a *sor* adattípus szokásos műveleteit. (Ha a nyilvánvalóan adódó valódi listás ábrázolást választjuk, nem minden művelet költsége lesz állandó. Ha az ábrázolásban rendezett párokat használunk, amelyek első tagja egy részleges lista (17.4), második tagja pedig a részleges lista végén lévő változó, sikerülhet a sor hosszától független, állandó műveletigény megvalósítása.)

12. A (17.12.2) részben megírtuk az `út/3` predikátumot, amely képes egy tetszőleges véges irányított gráf mélységi bejárására. Írjunk hasonló predikátumot, amely a gráfokat adott csúcstól kezdve szélességi módon járja be. Használjuk fel a 11. feladatban elkészített sorkezelő modult. Vegyük észre, hogy ez az eljárás képes két csúcs között a legrövidebb út meghatározására.

18 Aspektusorientált programozás

Ezen fejezet fő célja az aspektusorientált programozás, mint új programozási paradigma bemutatása. Miután bemutattuk a legfőbb mozgatórugókat, amelyek ezen újfajta megközelítés létrejöttéhez és elterjedéséhez vezettek, ismertetjük, hogy miben rejlik az aspektusorientált programozás újszerűsége. Ezt követően kerül sor néhány olyan programozási nyelv megemlítésére, melyek az aspektusorientált programozást támogatják, ezek közül részletesebben az AspectJ nyelvvel foglalkozunk. Végül szemügyre veszünk néhány, az aspektusorientált programozással rokonságban álló szoftverfejlesztési megközelítést.

A 70-es években az úgynevezett *szoftverkrízis* alapjaiban változtatta meg a számítógép-programozás megközelítését és módszereit, elterjesztve a strukturált, moduláris, majd objektumorientált programozást. Ugyanígy napjainkban is egyre inkább érezhető, hogy a programok fejlesztői nem képesek a megfelelő tempóban és minőségben felvenni a versenyt egyrészt a Moore-törvénnyel[1], másrészt az informatikai eszközök egyre szélesebb körű elterjedésének sebességével.

Az OOP-n túl

Az objektumorientált programozás annak idején talán a lehető legjobb választ adta az áttekinthető és ily módon biztonságosabb programok írására vonatkozó igényre, valamint a kód-újrafelhasználhatóság követelményére, ez okozta széleskörű elterjedését és de facto egyeduralmát. Mindezt tovább fokozta a modellezésben végbement szabványosodási folyamat az UML révén.

Általános módszerként és programozási nyelv központú megközelítésként az OOP (és az UML) azonban nem oldott meg két problémát. Egyrészt, mivel csupán az általános szoftver-modellezést segíti a megfelelő módon, eredeti formájában csak csekély mértékben alkalmas arra, hogy az egyes alkalmazási területek (például üzleti intelligencia, 3 dimenziós grafikai modellezés, távközlési protokollok vezérlése, banki számlavezetés) és szoftverfejlesztési aspektusok, szempontok (például naplózás, biztonsági szolgáltatások kezelése, hálózati kommunikáció) absztrakcióját megoldja, illetve szabványosításukra módot adjon.

Másrészt, az UML alapú round-trip engineering (programkód generáló és elemző) megoldások csak részben alkalmasak az öröklött rendszerek új programokkal való integrációjára és a különböző platformokra és nyelvekre épülő összetett szoftverrendszerek egységes tervezésére és generálására.

Ezekre a problémákra számos új módszertan igyekszik megoldást találni. Ilyen új módszertan például az aspektusorientált programozás [Kic 97], a vonatkozások többdimenziós

[1] Gordon Moore, az Intel korábbi vezetője fedezte fel a 70-es években azt a szabályszerűséget, mely szerint az informatikai eszközök számítási kapacitása másfél évenként megduplázódik. A tézis beigazolódott, és a törvény előreláthatólag még néhány évtizedig érvényben is marad.

szétválasztása (angolul multi-dimensional separation of concerns), az intencionális (szándék alapú, szándékorientált) programozás [IPF 03, Sim 99], vagy a generatív programozás [Cza 00]. Mindezen módszertanok közül a leginkább kiforrott és a legjobban elterjedt az aspektusorientált programozás.

Aspektusorientált programozás

Az aspektusorientált programozás egy olyan programozási paradigma, amely a kilencvenes évek közepén született meg. Manapság a programozási nyelvekkel kapcsolatos kutatások középpontjában áll, és minden bizonnyal a közeljövőben széles körben el fog terjedni. Az aspektusorientált szoftverfejlesztéssel kapcsolatos módszertanokról, nyelvekről és eszközökről (többek között azokról is, amelyek ebben a fejezetben előfordulnak) az ilyen irányú kutatásokat nyomonkövető [AOSD 03] nyújt egy részletes hivatkozási listát.

Az aspektusok olyan funkcionálisan összefüggő programrészletek, amelyek egy, az AOP lehetőségeit nélkülöző nyelven írt programban a forráskód különböző részein szétszórtan[2] találhatók meg. A megközelítés célja, hogy ezen összefüggő részek egy helyre gyűjtésével a programtermék (kód, terv) bonyolultságát csökkentse, karbantarthatóságát, olvashatóságát javítsa. Ily módon az aspektusok egy újfajta modularizációs eszközt valósítanak meg, mellyel tökéletesebbé válhat a szoftver különböző „vonatkozásainak" szétválasztása (principle of separation of concerns [Dij 76]).

Számos aspektusorientált módszer és eszköz létezik napjainkban. Ezek közül a legismertebb a Java nyelvhez készült AspectJ, melyre külön is kitérünk.

18.1. Az AOP áttekintése

A szoftvertervezési módszerek és a programozási nyelvek kölcsönösen támogatják egymást. A modellezési és tervezési módszertanok a rendszereket fokozatosan egyre kisebb egységekre bontják. A programozási nyelvek ezzel ellentétes irányú megoldásokat alkalmaznak: lehetővé teszik a fejlesztőknek, hogy a rendszer alegységeit képviselő absztrakciókat hozzanak létre, majd ezeket különféle módokon kombinálva összeállítsák a kész szoftverterméket.

Egy tervezési módszer és egy programozási nyelv abban az esetben tud jól együttműködni, ha a nyelv olyan absztrakciós és kompozíciós szolgáltatásokat biztosít, melyek könnyen használható és természetes módon támogatják azokat az elemi egységeket, amelyekre a tervezési módszer lebontja a rendszereket.

Ebből a szempontból az objektumorientált, procedurális és funkcionális programozási nyelvek (nevezzük most az egyszerűség kedvéért ezeket hagyományos programozási nyelveknek) legnagyobb része hasonló módon működik. Azok a rendszertervezési módszerek, amelyek a hagyományos programozási nyelvekkel egyidejűleg fejlődtek ki, a programot viselkedés vagy funkcionalitás alapján bontják részekre. Ezt a megközelítést szokás funkcionális dekompozíciónak hívni. Természetesen a dekompozíció jellege nyelvenként és paradig-

[2] A szakirodalomban a *scattered code* és a *tangling code* kifejezéseket használják. Az előbbi a funkcionálisan összefüggő kódrészletek „szétszóródását", a nem megfelelő modularizáltságot jelenti, míg az utóbbi ennek következménye: az „összegubancolódás" a programtermék megnövekedett komplexitására utal.

mánként jelentős eltéréseket mutathat, de a lebontáshoz használt nyelvi elemek (függvény, eljárás, modul, objektum) a nagy rendszer *funkcionális* egységei.

Ez utóbbi megállapítás túlságosan is nyilvánvalónak tűnhet, de az ad neki jelentőséget, hogy a jelen fejezetben tárgyalt aspektusorientált programozás a fenti funkcionális dekompozícióra „merőlegesen" bontja elemekre a programokat.

18.1.1. Aspektusok és komponensek

Minden programnak vannak olyan részei, melyek a funkcionális dekompozíció során kapott alprogramokat, objektumokat vagy modulokat az éppen vizsgált funkcionalitástól függetlenül szövik át. Gyakran szoktak ilyen elemek lenni a szinkronizációval foglalkozó, a hibakezelő, a hálózati kommunikációt bonyolító vagy a naplózó programrészek.

Ezek a részletek sok helyen, szinte vagy teljesen azonos módon szerepelnek, tehát feleslegesen növelik a kód méretét, de a hagyományos, funkcionális dekompozíción alapuló tervezéskor nem szoktak külön programegységekbe kerülni, mert nem képviselnek a rendszer funkcionális logikai szerkezetében önálló elemeket. Ez azt eredményezi, hogy ha a fent említett tevékenységek jellegében változás következik be, az sok helyen egyszerre történő programmódosítást követel meg, ami rontja a program karbantarthatóságát.

E jelenség megfelelő módon való kezelése céljából jött létre az aspektusorientált programozás, amelyet a következőképpen definiálhatunk:

- Nevezzük *komponensnek* azon programegységeket, amelyek a rendszer funkcionális dekompozíciójakor természetes módon előállnak. (Objektum, metódus, eljárás stb.)
- Nevezzük *aspektusnak* azokat a programrészleteket, melyek a funkcionális dekompozíció során nem állnak elő természetes módon programegységként, hanem a rendszer viselkedését és hatékonyságát egyéb rendszerezett módon befolyásolják.
- Ezek után az aspektusorientált programozás célját a következő módon határozhatjuk meg: válasszuk szét a rendszerünkön belül a komponenseket és aspektusokat olyan módszerek és eszközök segítségével, melyek lehetővé teszik a megfelelő absztrakciót és a rendszer hatékony kompozícióját.

A fenti definícióból is látható, hogy két dologban különböznek leginkább az AOP nyelvek a hagyományos programozási nyelvektől. Egyrészt a programok dekompozíciójára, komponensek és aspektusok absztrakciójára is lehetőséget adnak (tehát például önálló programegységben fogalmazhatjuk meg a naplózással kapcsolatos összes teendőt), másrészt pedig lehetőséget adnak a komponensek és az aspektusok természetes és egyszerű módon való összeillesztésére.[3] Ez utóbbit a szakirodalomban szövésnek (weaving) nevezik, ami történhet előfordító vagy fordító segítségével (ilyen az AspectJ is), vagy akár futási időben (mint például a JMangler vagy a PROSE esetében).

[3] A hagyományos programozási nyelvek is tartalmaznak elszigetelt aspektus-szerű elemeket. Ezek közül a legnyilvánvalóbb párhuzamnak a kivételkezelés aspektusorientált módon történő értelmezése tűnik. Például a Javában különálló moduláris egységben írhatjuk le a különböző fajtájú kivételek viselkedését (*Exception* osztályok), valamint explicit módon utasítást adhatunk a funkcionális és kivételkezelő részek kombinációjára (**try**, **catch**). Természetesen az AOP ennél összetettebb dolgokra is képes (hiszen itt még a kompozícióra a programozónak explicit módon kell utasítást adnia).

18.1.2. Aspektusleíró nyelvek

Annak érdekében, hogy az aspektusorientált megoldások természetes módon együtt tudjanak élni a meglévő nyelvekkel, olyan nyelveket szokás definiálni, melyek egy hagyományos nyelvre ráépülve, azt kiterjesztve teszik lehetővé az aspektusok meghatározását és a program hagyományos részeivel való összekapcsolását. Ezeket a nyelvkiegészítéseket hívjuk aspektusleíró nyelveknek. Az aspektusleíró nyelvek lehetnek akár speciális, akár általános célúak. Az előbbiek közé tartoznak például szinkronizációs nyelvek, elosztott programozást támogató nyelvek, monitorozást lehetővé tevő nyelvek (lásd például [Lop 97, Guly 99a]), de ide sorolhatók akár a manapság oly népszerű Enterprise JavaBeans technológia részét képező telepítési leírások (deployment descriptor) is, amelyekben például megfogalmazhatjuk az EJB-k üzleti metódusainak tranzakciós aspektusait [EJB 03]. Az általános célú aspektusleíró nyelvek nem szakosodnak egy bizonyos fajta aspektus leírására, hanem tetszőleges funkcionalitást megvalósító aspektusok megfogalmazását és komponálását teszik lehetővé. Ilyen nyelv az AspectJ is.

A fentiek függvényében egy alkalmazás aspektusorientált megvalósításának követelményei a következők:

(i.a) Rendelkezésünkre áll egy hagyományos, komponensek leírására alkalmas nyelv.

(i.b) Kéznél van egy vagy több aspektusleíró nyelv.

(ii) Birtokunkban vannak ezen nyelvek kombinációjához a megfelelő szövőprogramok.

(iii.a) Megírjuk a komponenseket a hagyományos nyelven, azok funkcionális szerepére összpontosítva.

(iii.b) Megírjuk az aspektusokat a megfelelő aspektusleíró nyelv(ek) használatával.

Az aspektusleíró nyelvek általában látható módon különböznek a hagyományos nyelvektől, ugyanis természetüknél fogva gyakran egyaránt tartalmaznak deklaratív és imperatív részeket. Az imperatív részek szerepe nyilvánvaló: ezekben írhatjuk le azokat a műveleteket, melyekből az egyes aspektusok megvalósítása előáll. A deklaratív részeket általában olyan szabályok alkotják, amelyek a szövőprogramot vezérlik azon tekintetben, hogy a hagyományos nyelven leírt mely komponensek mely részeivel és hogyan kell összeszőnie az aspektusok viselkedését. Ezekre a következő szakaszban részletesen kitérünk.

18.1.3. Aspektusszövők

Az aspektusszövők (*aspect weaver*) feladata, hogy a hagyományos és az aspektusleíró nyelvek segítségével készült programrészek felhasználásával összeállítsák a teljes rendszert. A szövők működését az úgynevezett *kapcsolódási pontok* (join points) vezérlik. Ezek a hagyományos nyelvek szemantikájára épülve határozzák meg, hogy az aspektusok hol kapcsolódhatnak be a komponensek működésébe.

Ilyen kapcsolódási pontok lehetnek például a komponensek közötti adatfolyamok vagy a metódusok belépési pontjai. Ez a két példa is mutatja, hogy a kapcsolódási pontokat nem feltétlenül explicit, szintaktikus nyelvi elemekkel határozzák meg, hanem a nyelv szemantikájának (a változók láthatóságára és hatókörére vonatkozó szabályok, a polimorfizmus és a dinamikus összekapcsolás stb.) ismeretében.

Az aspektusszövők a fentiek következtében oly módon működnek, hogy először feltérképezik a komponens program (lehetséges) kapcsolódási pontjait, majd ezekre ráillesztik az aspektusok viselkedését.

18.2. Az AspectJ bemutatása

Az AspectJ [Kic 01] a Java nyelv természetes kiterjesztése: minden Java program egyben AspectJ program is, az AspectJ által előállított programok minden JVM-en képesek futni, a Java fejlesztőeszközök természetes módon kiterjeszthetők AspectJ fejlesztőeszközökké, valamint az AspectJ szintaxisa olyan módon terjeszti ki a Javát, hogy a Java programozók a nyelv természetes részének tekintsék a kiterjesztéseket.

Az AspectJ-t a XEROX PARC (Palo Alto Research Center) kezdte kifejleszteni Gregor Kiczales vezetésével. Ezen sorok írásakor az AspectJ fejlesztését az Eclipse projekt [Ecl 03] koordinálja.

Teljes leírást nem célunk e helyt adni, csupán érzékeltetni szeretnénk az AspectJ megközelítését, sajátos aspektusorientált felfogását.

18.2.1. Az AspectJ elemei és főbb tulajdonságai

A nyelv fő elemei: *dinamikus kapcsolódási pontok* (dynamic join point), *eseményleíró kifejezések* (pointcut), *tanácsok* (advice), valamint *bevezetések* (introduction), amelyek *aspektusokká* (aspect) kombinálhatók.

- A *dinamikus kapcsolódási pontok* a programfutás olyan eseményeihez kötődnek, mint a metódushívás, a metódushívás fogadása, a metódus futása, az attribútumok lekérdezése, a kivételek kiváltása, az osztályok és objektumok inicializációja stb.
- Az *eseményleíró kifejezések* a dinamikus kapcsolódási pontok halmazait fogalmazzák meg. Eseményleíró kifejezésekből újabbakat képezhetünk különböző halmazműveletek segítségével.
- A *tanácsok* olyan metódusszerű műveletek, melyek egy eseményleíró kifejezéshez rendelt (az események előtt és/vagy után végrehajtott) tevékenységeket írják le.
- A *bevezetések* az aspektusok megvalósításához szükséges tagok (segédváltozók, segédmetódusok) bevezetését teszik lehetővé.
- Az *aspektus* olyan moduláris egység, amely az előző három elemből épül fel.

Annak érdekében, hogy futás közben egyértelműen eldönthető legyen, hogy az egy kapcsolódási pontra illesztett (akár több aspektusból származó) tanácsok milyen sorrendben futhatnak le, az aspektusok között és az aspektusokon belül jól definiált precedenciarelációk állnak fenn. Ezen kívül az aspektusok között öröklődési reláció is definiálható.

18.2.2. Egy rövid AspectJ példa

Az alábbi példaprogram azt hivatott bemutatni, hogyan lehet egyszerűen megvalósítani a naplózást az AspectJ konstrukcióinak segítségével.

Az alábbi *EgyszerűHibaNaplózás* aspektusban megadjuk, hogy a *hu.elte.inf* csomag minden osztályának minden publikus metódusában egy *Error* típusú kivétel kiváltása a naplóba való írást eredményezzen.

```
aspect EgyszerűHibaNaplózás {
    Napló napló = new Napló();

    pointcut publikusHívások():
        receptions(public * hu.elte.inf.*.*(..));

    after() throwing (Error e): publikusHívások() {
        napló.naplóz(e);
    }
}
```

A *napló* változó deklarálása és inicializálása egy bevezetés, mely formáját tekintve egy hagyományos tagváltozó-deklaráció. A létrejött változó ebben az esetben nem egy objektumnak vagy egy osztálynak, hanem magának az *EgyszerűHibaNaplózás* aspektusnak lesz egy tagja. Lehet egyébként az AspectJ-ben olyan bevezetéseket is írni, melyek egy vagy több osztály új példány- vagy osztályszintű tagját hozzák létre.

A *publikusHívások* eseményleíró kifejezés metódushívások egy halmazát határozza meg: ezt jelzi a *receptions* kulcsszó használata. A kulcsszó után a számunkra fontos metódusok jellemzése történik, melyhez helyettesítő (joker) karaktereket is használhatunk. Az eseményleíró kifejezésben összegyűjtött metódusok publikusak, és a *hu.elte.inf* csomag osztályaiban szerepelnek. Azt, hogy a csomag mely osztályáról van szó, nem rögzítjük: erre utal a második * helyettesítő karakter. A harmadik * jelzi, hogy a metódus nevére sem teszünk megkötést, míg az első * és a `(..)` a tetszőleges visszatérési értéket és a tetszőleges formális paraméterlistát jelenti.

Az utolsó elem az aspektusban a tanács, amely a viselkedést adja meg (ez a naplóba írás), valamint azt, hogy az említett *publikusHívások* eseményleíró kifejezés által meghatározott metódushívásoknál mely időpontban fusson le a viselkedés (egy *Error* kivétel kiváltását követően). Az aspektusszövő tehát azokra a dinamikus kapcsolódási pontokra fogja a naplózást ráilleszteni, ahol a szóban forgó metódusokban fellép a megadott nevű kivétel.

18.2.3. Fejlesztőeszközök és rokon nyelvek

Ezen sorok írásakor is már több fejlesztőeszközhöz (például az Eclipsehez, a JBuilderhez, a Forte 4J-hez, a SunOne Studiohoz, az Emacshez) létezik AspectJ kiterjesztés, és folyamatosan dolgoznak más eszközökhöz történő illesztésen is. Kifejlesztettek továbbá egy „saját" fejlesztőeszközt is az AspectJ-hez, melyet AspectJ Browser névre kereszteltek. Ennek segítségével a szokásos módon böngészhetjük a program nyelvi elemeit. A Java alprogramoknál megjeleníthetjük a hozzájuk kapcsolódó aspektusokat, és viszont.

Időközben más nyelvekhez is születtek az AspectJ-re hasonlító kiterjesztések. Ilyen nyelvek például a C, a C++, a C#, a Perl, a Python, a Ruby vagy a Smalltalk. Ezek a kiterjesztések azonban nem érték még el azt a fejlettségi, támogatottsági és elterjedtségi szintet, mint az AspectJ.

18.3. Az AOP-vel rokon paradigmák és megvalósításaik

Az alábbiakban néhány, az AOP-vel rokon paradigmát tekintünk át. Ezek között van olyan, amely az AOP előzményének tekinthető (adaptív programozás), van olyan, amelynek az AOP egy speciális esete (vonatkozások többdimenziós szétválasztása, generatív programozás), és van olyan is, amely hasonló célokkal született, mint az AOP, de más elven alapszik.

18.3.1. Vonatkozások többdimenziós szétválasztása (MDSC)

Ezt a módszertant, ami az aspektusorientált programozás általánosításának tekinthető, az IBM fejlesztette ki Harold Ossher és Peri Tarr vezetésével. A vonatkozások többdimenziós szétválasztása (Multi-Dimensional Separation Of Concerns, MDSC, [MDSC 02]) egy olyan paradigma, mely az elsődleges dekompozíció uralmát (Tyranny of the Dominant Decomposition) igyekszik megdönteni. Ez alatt azt kell érteni, hogy egy rendszer megtervezésekor az alrendszereket nem csak egyféleképpen lehet, sőt kell kialakítani, hanem többféleképpen, mégpedig többféle vonatkozásrendszernek (dimenziónak) megfelelően. A korábban már említett funkcionális dekompozíció csak egy szemszögből (a vonatkozások egy dimenziójában) bontja fel a rendszert komponensekre. Emellett az áttekinthető, könnyen módosítható, kis bonyolultságú kód kialakításához egyéb dimenziók szerinti dekompozícióra is szükség van. Például egy objektumorientált módszertan alkalmazása esetén a dekompozíció osztályokat hoz létre, melyekben különféle tevékenységeket leíró műveletek kapnak helyet. Emellett szükséges lehet az is, hogy ugyanezt a rendszert más dekompozíciók segítségével, az előző dekompozícióra „merőlegesen" bontsunk részekre. Mutassuk be ezt egy példa segítségével [Oss 99].

Tegyük fel, hogy egy aritmetikai kifejezéseket kezelő programot szeretnénk kifejleszteni, melynek segítségével kifejezések szintaktikus helyességének ellenőrzését, valamint kifejezések kiértékelését és megjelenítését lehet elvégezni. Az objektumorientált megközelítés segítségével létrejött modell egymással különböző (öröklődési, aggregációs, asszociációs) kapcsolatban álló osztályokat fog tartalmazni, mint például Kifejezés, Literál, Változó, Operátor, Unáris Operátor, Bináris Operátor stb. Ezekben az osztályokban szerepelni fognak az Ellenőriz, Kiértékel és Megjelenít műveletek az adott osztályra specializált változatai. Egy lehetséges alternatív dekompozíció a rendszert épp ezen három feladatkör szerint bonthatná részekre. A rendszer egyéb aspektusai (szinkronizáció, naplózás stb.) újabb dimenzióit jelenthetik a dekompozíciónak.

A vonatkozások többdimenziós szétválasztásának elve alapján a szoftverfejlesztés megkönnyítése érdekében a különböző dimenziók szerinti dekompozíciókat egymással egyenrangúnak kell tekinteni, és a szoftver életciklusa során az általuk kialakított komponenseket (az úgynevezett vonatkozásokat) önállóan kezelhetőnek, elkülöníthetőnek kell tartani. Ezt a célt megfelelő programnyelvi és eszköztámogatással lehet elérni.

Egy lehetséges megvalósítása a fent említett elveknek a hiperterek (hyperspaces) használata. Ez képezi alapját a HyperJ nyelvnek is, mely a Java nyelv MDSC alapú kiterjesztése.

Szubjektumorientált programozás (SOP)

A *szubjektumorientált programozás* [SOP 03] módszertanát szintén az IBM fejlesztette ki (W. Harrison, H. Ossher). Ez a módszertan volt a közvetlen elődje a vonatkozások többdimenziós szétválasztásának. A módszertan központi fogalma a szubjektum, mely osztályok és osztályrészletek egy gyűjteménye. Egy részfeladat megoldásához szükséges kódrészletek egy szubjektum formájában fogalmazhatók meg. A szubjektumok különféle kompozíciós szabályok segítségével állíthatók össze alkalmazássá. Végeredményben egy szubjektum egy aspektusnak is tekinthető.

Egy objektumorientált elvek alapján felépített rendszerben az objektumok a valódi világ objektumait modellezik. Ennek következményeképpen egy objektum több feladat (vagy programfunkció) megoldásában is részt vehet. Az objektumban tárolt adatok bizonyos részei és az objektum bizonyos műveletei az egyik, míg más adatok és műveletek egy másik feladat megoldásához szükségesek. Ezt úgy is megfogalmazhatjuk, hogy ugyanaz az objektum több funkció alapján, több szemszögből is vizsgálható, azaz több szubjektum kialakításában is szerepet játszhat. A szubjektum tehát egy részleges objektummodell, amely egy adott funkcionalitás megvalósításához szükséges osztályok releváns részeit (adattagokat, műveleteket) és releváns (öröklődési, aggregációs, asszociációs) kapcsolatait foglalja magába. (Egy hasonló fogalom a mixin, mely egy jól meghatározott, általában kisebb méretű funkcionalitást megvalósító osztályrészletet jelöl. A mixineket komponálni szokás más mixinekkel és osztályokkal, így állítva elő többrétű funkcionalitással rendelkező objektumokat.)

A szubjektumorientált programozás nyelvfüggetlen, bár objektumorientált megközelítéshez lett kialakítva. Egy megvalósítását a C++ nyelvhez, illetve az IBM VisualAge for C++ fejlesztőeszközhöz készítették el.

18.3.2. Adaptív programozás (AP)

Ez a paradigma az AOP egy elődje, egyben az AOP egy speciális esetének is tekinthető. A bostoni Northeastern Egyetemen fejlesztették ki Karl Lieberherr vezetésével [Dem 03]. Az *adaptív programozás* elve az, hogy a tevékenységeket megfogalmazó kódot úgy kell elkészíteni, hogy az független legyen attól a struktúrától, amelyen a tevékenység dolgozni fog. Ily módon lehetőség nyílik a funkcionalitás és a szerkezet (mint egy rendszer két fontos aspektusának) szétválasztására.

Ez a szemlélet megfelel *Demeter törvényének*, mely egy általános objektumorientált tervezési elv, és amely azt követeli meg, hogy a műveletek minél kevesebb információt használjanak a rendszer struktúrájáról. Például egy objektumorientált programban a metódusok kódjába nem égetjük be az osztályhierarchiát, hanem úgy készítjük el azokat, hogy azok *alkalmazkodjanak* a szerkezethez. Az adaptív programozás módszertana azt javasolja, hogy a tényleges számítást végző metódusokat kódoljuk csak le hagyományos módon (például az adott objektumorientált nyelven), míg azt a kódot, amely előkeresi azokat az objektumokat, amelyekre a fenti metódusokat végre kell hajtani, gráfbejárási stratégiák (traversal

strategy) segítségével fogalmazzuk meg. A bejárási stratégiák az objektumok közötti (öröklődési, aggregációs és asszociációs) kapcsolatokat leíró gráf bizonyos tulajdonságú csúcsainak megkeresését teszik lehetővé.

Az adaptív programozás elveit a C++ nyelvhez készített Demeter/C++, illetve a Java nyelvhez készített DemeterJ és utódja, a DJ valósítja meg.

18.3.3. Kompozíciós szűrők (CF)

Ismét egy olyan módszertant vázolunk fel, amely az objektumorientált paradigma hiányosságait igyekszik kiküszöbölni. A *kompozíciós szűrők* (composition filters) az objektumok között váltott üzenetek kézbesítésébe avatkoznak be, ily módon egészítve ki egy rendszert újabb funkcionalitással, tevékenységek újabb aspektusaival. A szűrők segítségével modularizáltan fogalmazhatjuk meg programunk viselkedésének bizonyos vonatkozásait. A szűrők az objektumok kimenő és bemenő üzeneteit módosíthatják, azaz, aspektusorientált szóhasználattal élve, a kapcsolódási pontok az üzenetküldések és -fogadások. Aspektusok megfogalmazása mellett a kompozíciós szűrők egyéb technikák, például delegáció és dinamikus öröklődés megvalósítására is használhatók.

A kompozíciós szűrők Mehmet Aksittól, a twentei egyetemről származnak [CF 01]. Több megvalósítása is létezik, ezek közül kiemelnénk a Sina nyelvet, illetve a C++ nyelvhez és a CORBA-hoz készített kiterjesztéseket. Egyéb érdekes próbálkozások a Java nyelvhez készített ConcernJ és ComposeJ.

Fontosnak tartjuk megjegyezni, hogy az Enterprise JavaBeans technológia [EJB 03] is szoros kapcsolatban áll a kompozíciós szűrőkkel [Cza 00]. Az EJB-k segítségével szerveroldali komponenseket készíthetünk, melyek egy J2EE rendszer szolgáltatásait szabványos interfészeken keresztül vehetik igénybe. Egy bean kódja áll egyrészt az üzleti logika megvalósításából, melyet Java nyelven készítünk el, másrészt az úgynevezett telepítési leírásból, mely a komponens különböző egyéb aspektusait (perzisztencia, tranzakcionalitás, konkurrencia, biztonság, elosztott kommunikáció) fogalmazza meg. A komponens telepítésekor a keretrendszer a telepítési leírás segítségével csomagoló objektumokat (lényegében véve kompozíciós szűrőket) generál, melyek segítenek megvalósítani a komponens fent említett aspektusait.

18.3.4. Generatív programozás (GP)

A *generatív programozás* (Generative Programming) egy viszonylag új keletű megközelítés, alapműve ugyanezzel a címmel 2000-ben jelent meg [Cza 00]. A paradigma lényege, hogy a szoftverfejlesztést megpróbáljuk nagyban automatizálni. Ebből kifolyólag a generikus programozás olyan módszerek és eszközök gyűjtőfogalma, amelyek segítenek megtervezni, legyártani, konfigurálni és összeépíteni a rendszer komponenseit. Az aspektusorientált programozás csak egy ezek közül a módszerek közül. Ide sorolhatjuk még a generikus programozást, a metaprogramozást, az intencionális (szándékorientált) programozást stb. is. Foglaljuk össze pár mondatban, hogy mit adhat a generatív programozás a szoftverfejlesztőknek a rendszeranalízis és -tervezés, valamint a programozás (program megvalósítás) terén.

A *generikus programozás* (generic programming) egy olyan programmegvalósítási technológia, amelyre a generatív programozás jelentős mértékben épít. Ez a technológia lehetővé teszi algoritmusok és adatszerkezetek absztrakt megfogalmazását (így lehetőséget nyújt arra, hogy rendszerkomponensek készítésekor a lehető legnagyobb mértékben elvonatkoztathassunk az érdektelen részletektől), emellett az elkészített absztrakt komponensek konfigurálhatóságát és együttműködésüknek megszervezését is biztosítja. A generikus programozás egy szép példája a C++ Standard Template Library, mely a típussal való paraméterezésen és a polimorfizmuson (lásd a 11. fejezetet) kívül egyéb technikákat – iterátorokat, adaptereket, függvényobjektumokat, trait-eket – is alkalmaz.

A *metaprogramozás*, mely egy másik fontos eszköze a generatív programozásnak, azt jelenti, hogy programokat program segítségével manipulálunk (például konfigurálunk, adaptálunk), instrumentálunk (például tesztelési vagy profilozási céllal) vagy generálunk. Ezen utóbbihoz tartozik a makrók használata és az aspektusorientált technikák is. A fordítóprogramok és az előfordítók is metaprogramoknak tekinthetők. Reflektív programok, melyek saját magukat vizsgálják (önelemzés, introspection) vagy módosítják (intercession), szintén metaprogramok. (A Smalltalk és a CLOS nyelvek például széles körű segítséget nyújtanak reflektív programok készítéséhez, míg a Java nyelv csak az önelemzést teszi lehetővé.) Végül hadd említsük meg a metaprogramozás egy további érdekes példáját, a C++ *template*-eken alapuló template metaprogramozást (C++ template metaprogramming), mely segítségével C++ programunk bizonyos részeit fordítási időben hajthatjuk végre [Ale 01].

A generatív programozás módszertana azt javasolja, hogy az egyes alkalmazási területek (domain) tudását rugalmasan paraméterezhető és kombinálható szabványos könyvtárakban („aktív könyvtárak") írjuk le, valamint lehetőséget adjunk arra, hogy a rendszerfejlesztők (tervezők) az adott szakterületnek megfelelő modellezési leírónyelveket (az UML-lel szemben, ami túl általános), és ezekhez az aktív könyvtárak lehetőségeit nagyban kihasználó kódgenerátorokat használhassanak. Ily módon egyre inkább megközelíthetővé válik az az állapot, hogy a programkód 100%-ban generálódjon (innen a paradigma elnevezése) a rendszertervből.

18.3.5. Intencionális programozás (IP)

Az *intencionális (szándékorientált) programozás* a Microsoft Research kutatási projektjeként indult a 90-es évek elején Charles Simonyi vezetésével. Ezen megközelítés célja egyrészt a programterv és a kód közötti éles határok eltüntetése, valamint az, hogy tetszőleges szak- és részterület absztrakcióit lehessen vele hatékonyan leírni, és az absztrakciókat rugalmasan és hatékonyan lehessen kombinálni. Emiatt az intencionális programozást szokás a generatív programozás egyik alfajának tekinteni.

Az „intencionális programozás" elnevezés a kognitív pszichológiából származik, és azt hivatott megfogalmazni, hogy az absztrakció és a reprezentáció ideális szimbiózisának következtében kevésbé tűnik el a program megálmodójának eredeti szándéka a kódolás során, mint a hagyományos módszerek esetében.

Miután a Microsoft leállította az IP projektet, nem sokkal később Charles Simonyi és Gregor Kiczales megalakították az Intentional Software Corporationt, mely az IP ötletét komplett fejlesztőeszköz formájában igyekszik megvalósítani.[4] Kiczales részvétele a cégben

[4] Az IP fejlesztések egy jelentős része Magyarországon folyik a NETvisor Kft. és az ELTE Általános Számítástudományi Tanszékének közreműködésével.

jelzi, hogy elsősorban az AOP felhasználóit célozzák meg az új megközelítéssel, mivel az IP alkalmas az aspektusorientáltság leírására.

18.3.6. További ígéretes kezdeményezések

Az alábbiakban a teljesség igénye nélkül felvázolunk néhány kezdeményezést, melyek a fentiekhez hasonló célokból születtek.

Eclipse Universal Modeler

Az IBM Research kutatóinak célja, hogy az Eclipse nevű univerzális fejlesztőeszközre épülve olyan eszközt adjanak, mellyel tetszőleges absztrakciós mechanizmus és jelölésrendszer leírható [Ecl 03].

Az eszköz szabványos (OMG, azaz Object Management Group és W3C, azaz World-Wide Web Consortium) protokollokat és alapinfrastruktúrát használ (XMI azaz XML Metadata Interchange, MOF, azaz Meta-Object Factory, SVG, azaz Scalable Vector Graphics) a metaadatok leírására, valamint a diagramok ábrázolására.

Jackpot

Ez a projekt az EUM-hez hasonlóan univerzális modellezőeszköz kifejlesztését célozza, a NetBeans fejlesztőeszköz alapjain. A projekt vezetője a Sun Microsystemsnél James Gosling, a Java egyik megalkotója, aki korábban az Emacs kifejlesztésében is részt vett.

Az UML fejlődése

Az UML is abban az irányban fejlődik, hogy a tervezés során előálló modellből egyre teljesebb programkódot lehessen vele előállítani. Ilyen irányú kezdeményezés az UML viselkedés-szemantika, egy teljes UML alapú szoftverplatform kifejlesztésének terve. Ivar Jacobson (az UML alapját képező három objektumorientált szoftver-modellezési jelölésrendszer és módszertan egyikének, az Objectory-nak a kidolgozója) szavaival élve: néhány év múlva mindenki UML-ben fog programozni, a mostani programnyelvek nélkül. Ez kísértetiesen hasonlít az IP célkitűzésére.

Szintén Ivar Jacobson eredménye a WayPointer nevű rendszer, amely intelligens ágenseket[5] alkalmaz újszerű módon, melyek az UML modellezés különféle területein segítik a rendszertervezőket és fejlesztőket [Guly 99b], a Rational Rose (egy UML modellező, tervező és kódgeneráló eszköz) és a RUP (Rational Unified Process, az UML-re épülő, a szoftver előállításának teljes életciklusát lefedő módszertan) eszközöket használva. Ezt a segítséget kihasználva a tervezők olyan UML modelleket állíthatnak elő, amelyek lehetővé teszik a kódgeneráló képességek lehető legteljesebb, optimális kihasználását.

[5] Az ágensek – egy lehetséges meghatározás szerint – olyan önálló programkomponensek, amelyek környezetük állapotára és viselkedésére reagálva, az őket vezérlő szabályok, szándékok, érzelmek által meghatározott módon képesek intelligens viselkedésre. Ebben a konkrét esetben olyan ágensekről van szó, melyek az UML lehetőségeit és annak helyes és teljes használatát leíró szabályrendszer és tudásbázis által vezérelve a rendszertervezőknek aktuális javaslatokat tesznek, próbálják „kitalálni" a következő vagy a hiányzó lépést.

19 Programozási nyelvek rövid leírásai

Ebben a fejezetben a manapság leggyakrabban használt programozási nyelvekről igyekeztünk rövid leírást adni, a teljesség igénye nélkül.

Az alábbiakban néhány nyelvet szeretnénk külön-külön bemutatni. A teljességre nem törekedtünk, mint ahogy nem is törekedhettünk, célunk csupán annyi, hogy a programozási nyelvek sokszínűségét bemutassuk, és azt a szemléletet erősítsük, miszerint nem feltétlenül kell mindent egy aktuálisan divatos és elterjedt nyelven megvalósítani, hanem lehetőségünk van az adott feladathoz és annak környezetéhez igazodva ideális megoldást találni.

19.1. Ada 95

Az Ada programozási nyelvet egy nemzetközi tervezőcsoport, a CII-Honeywell Bull készítette a DoD (Department of Defense, az Egyesült Államok védelmi minisztériuma) megrendelésére. A csoport vezetője a francia Jean Ichbiah volt, aki a LIS programozási nyelvet is tervezte. A nyelv nevét Lady Ada Augusta Byronról kapta, akit a világ első programozójának tartanak. A tervezésnél alapnyelvnek a Pascalt választották, de néhány vonást egyéb nyelvekből – például a Modula-2, ALGOL 68, SIMULA 67, Alphard és CLU nyelvekből – is átvettek.

A tervezés szempontjai a következők voltak:

- megbízhatóság;
- továbbfejleszthetőség;
- következetesség;
- erős típusosság.

Az ANSI 1983-ban fogadta el az Ada szabványát, ISO szabvány pedig 1987-ben lett. Ennek a szabványnak a felülvizsgálatával született meg az Ada 95 [Nyé 98].

Az Ada programozási nyelv – főleg nagyobb programrendszerek fejlesztésében – nagyon sikeresnek bizonyult. Bár eredetileg katonai rendszerek fejlesztésére akarták használni, más felhasználói körökben (kereskedelem, telekommunikáció) is hamar elterjedt. A felhasználói kör kiszélesedése azonban újabb kívánalmakat és követelményeket állított a programozási nyelvvel szemben.

1988-ban indult a szabvány felülvizsgálata, az Ada Board ajánlása a következő irányelveket emelte ki:

- objektumorientált programozás támogatása;
- interface lehetőség más programnyelvekkel;
- hierarchikus könyvtárstruktúra;
- fejlettebb taszkütemezés, osztott objektumok könnyebb, megbízhatóbb használata;
- sablonok paraméterezési lehetőségeinek bővítése.

Az új szabvány kidolgozásánál nagyon fontos volt az Ada 83-mal való visszamenőleges kompatibilitás (ez néhány ritka kivétellel sikerült is), valamint a '83-as Ada nagy vívmányainak (biztonságosság, megbízhatóság stb.) megőrzése. Fontosnak tartották, hogy ez a bővítés egyszerű legyen, így összesen hat új kulcsszót vezettek be.

Végül 1995-re megszületett az Ada 95, az Ada 83 méltó utódja, amely természetesen szintén ISO szabvány lett. A következő felülvizsgálatot 2005-re tervezik...

A nyelv központi honlapja: `http://www.adahome.com/`

19.2. ALGOL 60

Az ALGOL (ALGOrithmic Language) egyike azon nyelveknek, amelyet a tudományos számítások támogatására terveztek.

Az ALGOL 60-ban vezették be a blokkszerkezet lehetőségét, a strukturált feltételes vezérlésátadó utasításokat és a rekurzió nyelvi megvalósítását. Ez volt az első nyelv, amely lehetőséget ad a felhasználónak, hogy bevezesse saját adattípusait, valamint ahol a kulcsszavakat már fenntartott szavakként is kezelték.

Ez volt az első nyelv, amelynek szintaxisát John Backus és Peter Naur az azóta BNF-ként (Backus Naur Form) elterjedt jelöléssel formálisan megadta.

A nyelv központi honlapja:
`http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html`

19.3. ALGOL 68

Az ALGOL 68 az ALGOL 60 továbbfejlesztése. Az ALGOL 68 tervezői egy újfajta nyelvtant alkalmaztak: a környezetfüggetlen, kétszintű, ún. Wijngaarden-nyelvtan segítségével definiálták a nyelvet, ami nemcsak a szintaxis, de a szemantika egy részének megadására is alkalmas volt. Így például a deklarációk és automatikus típuskonverziók szabályait is a nyelvtan segítségével adták meg. A fogalmakra új elnevezéseket vezettek be, talán ez is szerepet játszott abban, hogy a nyelv sohasem terjedt el széles körben, mára szinte kihalt. Ugyanakkor számos jellemzője nagy hatást gyakorolt későbbi nyelvek tervezőire (például Bjarne Stroustrupra, a C++ tervezőjére).

A nyelv központi honlapja: `http://www.algol68.org/`

19.4. BASIC

Kemény János (John G. Kemény) és Tom Kurtz vezetésével 1964-ben fejlesztették ki Dartmouthban a BASIC (Beginner's All Purpose Symbolic Instruction Code = kezdők általános célú szimbolikus utasításkódja) nyelvet, azzal a céllal, hogy „a számítógép minden egyetemi hallgató (és minden diák) számára hozzáférhetővé váljék"[1]. A nyelv korszakalkotó jelentősége, hogy a számítógépek használata ezáltal ténylegesen az átlagember – és nem csak a matematikusok és mérnökök – számára is megszokottá vált.

A nyelv központi honlapja: `http://www.fys.ruu.nl/~bergmann/basic.html`

19.5. BETA

A BETA nyelvet Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen és Kristen Nygaard tervezték a SIMULA 67-ből kiindulva.

Ez a modern objektumorientált nyelv bevezet egy fontos absztrakciót: a „mintát". A BETA-ban minden „minta": az osztály, az eljárás, a függvény, a párhuzamos folyamatok és a kivételek is. Minta-öröklődés van, így az öröklődés is sokkal szélesebbkörűen használható, mint a többi nyelvben (lásd [Mad 93]).

A nyelv központi honlapja: `http://www.daimi.au.dk/~beta/`

19.6. C

A C nyelvet Dennis Ritchie fejlesztette ki az AT&T Bell Labs-nál a 70-es évek elején [Ker 89]. A nyelv létrehozásának fő mozgatórugója az volt, hogy a UNIX operációs rendszer olyan programozási nyelven íródhasson, amely egyrészt kellően magasszintű ahhoz, hogy nagy mértékben hordozhatók legyenek a benne írt programok, másrészt kellően alacsonyszintű ahhoz, hogy lehessen benne olyan operációs rendszert és alkalmazásokat fejleszteni, amelyek a hardver-erőforrások adta lehetőségeket a legteljesebb mértékben kihasználják.

A fenti szempontoknak megfelelően a C nyelvben minden olyan konstrukció megtalálható, amely az akkori korszerű magas szintű, azaz strukturált, procedurális és moduláris programozási nyelvekre jellemző volt. Ugyanakkor az erőforrásokhoz való alacsony szintű hozzáférés területén a C fejlesztők szabad kezet kapnak. Konkrétan, a memóriacímek (mutatók) kezelésének rugalmassága, a viszonylag gyenge típusosság, valamint a nyelvből szinte teljesen kimaradt futási idejű ellenőrzések jelentik a hatékonyság kulcsát.

Ez utóbbiak rejtik ugyanakkor a nyelv alkalmazásának legnagyobb veszélyforrásait is, ugyanis a C nyelv nem kellően átgondolt alkalmazásával, valamint a módszeres tesztelés mellőzésével igen könnyen lehetséges megbízhatatlan, hibás programkód előállítása.

A 60-as és 70-es években szokásos, „beszédes" szintaxisú nyelvekkel ellentétben a C-nek szokatlanul tömör a szintaxisa (például **begin** és **end** kulcsszavak helyett { és } szerepel). Ez a tulajdonság – bármennyire is nehezen olvashatóvá teszi a C nyelvű forráskódokat –

[1] idézet Kemény Jánostól Marx György előadásában, lásd [Mar 93]

nagy népszerűségre tett szert a programozók körében, olyannyira, hogy számos későbbi programozási nyelvben C-szerű nyelvi elemeket vezettek be.

A nyelv komoly szerepet játszott a UNIX elterjedésében, de később számos további operációs rendszert, rendszerprogramot és alkalmazást írtak C nyelven.

A nyelv központi honlapja: `http://cm.bell-labs.com/cm/cs/cbook/`

19.7. C++

A C++ programozási nyelvet Bjarne Stroustrup fejlesztette ki az AT&T Bell Labs-nál, a 80-as évek elején [Str 00][C++ 98], a C nyelv továbbfejlesztéseképpen.

A nyelv létrehozásának fő oka az akkoriban egyre inkább terjedő objektumorientált paradigma volt. Ennek megfelelően a C++ néhány lényeges dologgal egészíti ki a C-t:

- Adatabsztrakció
- Objektumorientált programozás többszörös öröklődéssel
- Operátor túlterhelés
- Sablonok
- Kivételkezelés
- Adatfolyamok
- Szabványos könyvtár

Mindezek mellett a C++ megőrizte a C egyszerűségét és hatékonyságát azáltal, hogy felülről kompatibilis vele, és az új konstrukciók tervezésénél is igyekeztek ugyanazokat a fő szempontokat figyelembe venni – azaz a cél minél magasabb szintű konstrukciók alkalmazása a hardver-erőforrások maximális kihasználásával. Ennek megfelelően minden szintaktikailag helyes C program egyben egy szintaktikailag helyes C++ program is.

Annak következtében, hogy a C++ nyelv az objektumorientáltságot nem öncélúan, hanem a C-t, mint sokak által ismert és elismert nyelvet kiterjesztve vezette be, nagy mértékben hozzájárult ahhoz, hogy a kilencvenes évekre az objektumorientáltság szinte egyeduralkodó szoftverfejlesztési megközelítéssé váljon.

A C++ jelentőségét mutatja az is, hogy a XXI. század elejének legszélesebb körben alkalmazott programozási nyelve, a Java (és annak alteregója, a C#), belőle kiindulva, és hozzá nagyon sok dologban hasonló módon került kifejlesztésre.

A nyelv központi honlapja: `http://www.research.att.com/~bs/C++.html`

19.8. C#

A C# (ejtsd: szí sarp) a Microsoft által kifejlesztett általános célú imperatív programozási nyelv [Sch 02].

A nyelv a C++ kifejezőerejét igyekszik ötvözni a Visual Basic egyszerű használatával. A szintaktikus alapokat a C++-tól kölcsönzi, míg az egyszerűséget a Visual Basic-től. A C#-ot sokan hasonlítják még a Javához is, nem teljesen alaptalanul. A nyelv támogatja az objektumorientált programfejlesztést, valamint tartalmazza a C++ programnyelvi eszközeinek

jelentős részét és néhány újszerű elemet is. A C# egyik legfontosabb hiányossága, hogy nem támogatja sablonok készítését.

A nyelv központi honlapja:
`http://msdn.microsoft.com/vcsharp/`

19.9. Clean

A Concurrent Clean (Plasmeijer, Nijmegen, 1987) [Pla 01, Pla 99] egy gráfátíró kisérleti nyelvből (a LEAN-ból) fejlődött ki, tisztán funkcionális, alapvetően lusta kiértékelést alkalmazó nyelv. Jelenlegi változata (a Clean 2.0) közel áll a Haskellhez [Heg 02], de több olyan nyelvi elemet is tartalmaz, amelyet a Haskell'98 nem.

A nyelv központi honlapja: `http://www.cs.kun.nl/~clean/`

19.10. CLU

Az 1970-es években, elsősorban a programozási módszertan oktatásának támogatására tervezték és fejlesztették Barbara Liskov vezetésével a Massachusetts Institute of Technology (MIT) egyetemen a CLU nyelvet. Ez volt az első megvalósított programozási nyelv, mely nyelvi eszközöket adott az adatabsztrakcióhoz, azaz olyan fordítási egységek megvalósítását tette lehetővé, melyek csak a típusspecifikációt teszik elérhetővé és elrejtik a megvalósításhoz szükséges egyéb információkat. Ezeket a fordítási egységeket *cluster*-nek nevezték, innen ered a nyelv neve (lásd [Lis 86]). A CLU ezen kívül jelentős újításokat vezetett be például a kivételkezelés, az iterátorok és a sablonok területén is, s bár maga a nyelv nem terjedt el, számos programozási nyelv tervezője merített ötleteket belőle.

A nyelv központi honlapja: `http://www.pmg.lcs.mit.edu/CLU.html`

19.11. COBOL

1959-ben a Pennsylvaniai Egyetemen fogalmazták meg azt a véleményt, hogy szükség lenne egy adatfeldolgozásra orientált, nyilvántartások kezelésére alkalmas, gépfüggetlen programozási nyelvre. Olyan nyelvet akartak, amelyen csak a feladatot kell megfogalmazni(!), s a számítást a gép ez alapján már elvégzi.

A létrehozott Előkészítő Bizottság feladatául azt tűzték ki, hogy tanulmányozza át a létező nyelveket, és hozza létre az új nyelv „minimális programját". Megegyeztek a COBOL (COmmon Business Oriented Language) névben, s 1968-ban jelent meg az USA Standard COBOL c. specifikáció, majd 1974-ben ennek kis mértékben módosított változata, amely jelenleg a legelterjedtebb COBOL változat. Azóta megjelent a COBOL-85 szabvány, mely a nyelvet objektumorientált elemekkel bővítette. A már megírt COBOL programok nagyrészt a COBOL-74 szabvány szerint készültek, a fordítóprogramok legtöbbje is ezt a változatot ismeri (lásd [Bak 74]).

A nyelv készítőinek a kitűzött célokat sikerült elérniük. Az algoritmus helyett a feladat középpontba állítása egy sajátos, de a kijelölt feladatokra nagyon jól alkalmazható nyelvet

eredményezett. Több ponton, így például a típusok esetében sokkal jobb a COBOL programok átvihetőségének lehetősége, mint a legtöbb – akár egy évtizeddel később készült – nyelv esetében; a többi nyelvi elem gépfüggetlenségét jól mutatja, hogy – ellentétben például a FORTRAN-nal [Lőcs 77] – a programíráshoz a programozónak egyáltalán nem kell tudnia azt, milyen gépi kódú programot készít a fordítóprogram a programjából.

A nyelv központi honlapja: `http://www.cobolreport.com/`

19.12. Delphi

A grafikus alkalmazások megjelenésével igény mutatkozott aziránt, hogy olyan fejlesztői környezetek jelenjenek meg a szoftver piacon, amelyek segítségével ezek gyorsan készíthetők. A Borland cég ennek támogatására 1995-ben jelentette meg a Delphi első verzióját [Lis 00]. Előnyei közé sorolható a form-központúság, a gyors fordító, az egyszerű adatbáziskezelés, a Windows környezet széleskörű támogatása, az operációs rendszer nyújtotta lehetőségek magasszintű kihasználása, és nem utolsósorban a komponenseken alapuló alkalmazásfejlesztés.

A Delphi programozási nyelve az Object Pascal, alapvetően a Turbo Pascal 7.0 nyelv továbbfejlesztett verziója. A Delphi megalkotói a következő szempontokat tartották szem előtt a nyelv tervezésekor:

- Windows alatti alkalmazások egyszerű és gyors elkészítése
- objektumorientált jellemzők támogatása
- kliens/szerver adatbázis-kezelés lehetősége

Később megjelent a Delphi Linux-os fejlesztést támogató verziója, a Kylix is.

A nyelv központi honlapja: `http://www.borland.com/delphi/`

19.13. Eiffel

Az Eiffel programozási nyelvet az 1980-as évek második felében tervezték Bertrand Meyer vezetésével [Mey 91]. Az Eiffel egy olyan objektumorientált nyelv, amelyet termelési környezetekben, minőségi szoftver tervezéséhez és implementálásához ajánlanak. A nyelv elkészítésének szempontjai a következők voltak: támogassa újrafelhasználható, kiterjeszthető, helyes, megbízható, hordozható és hatékony programok írását. Mindezt lehetőleg egyszerűen, elegánsan és könnyen kezelhetően igyekeztek megoldani.

Az Eiffel nyelv ősei: a SIMULA 67, az ALGOL W, az Alphard, a CLU, az Ada és a Z specifikációs nyelv. Ezenkívül egy sor új szempontot figyelembe vettek az öröklődés, a típuskezelés, az exception kezelés és az állítások területén, valamint az előző nyelvekből átvett jellemzőket egységes egésszé konstruálták.

Az Eiffel nyelvet már úgy tervezték, hogy gyakorlatilag elválaszthatatlan a környezetétől, amit köré építettek. Használata elképzelhetetlen a kiterjedt könyvtári osztályok rendszere nélkül. Szorosan kötődik a LACE nyelvhez is (Language for Assembling Classes in Eiffel), ami tulajdonképpen egy makefile-hoz hasonló módon írja le a fordításra és futtatásra

vonatkozó megkötéseket. Az Eiffel fejlesztőkörnyezet még számos eszközt kínál, amelyek igyekeznek megkönnyíteni a nyelv hatékony használatát.

A nyelv központi honlapja: `http://www.eiffel.com/`

19.14. FORTRAN

A FORTRAN a numerikus számításokhoz használt első és mindmáig a legelterjedtebb nyelv. A neve is innen ered: FORTRAN = FORmula TRANslator. Az első változat tervezése 1954-ben kezdődött, John Backus és az IBM nevéhez fűződik, 1957-re készült el a fordítója. 1964-ben készítette el az ASA (American Standards Association) a FORTRAN nyelv első szabványát, ezt azóta több is követte, említésre érdemes a FORTRAN 90 definiálása, mely többek között a rekurzió lehetőségével bővítette a nyelvet. A párhuzamosságot támogató bővítések szabványosítására megszületett a HPF (High Performance Fortran). Jelenleg dolgoznak a FORTRAN 200x szabvány kialakításán, mely kibővíti majd a nyelvet többek között kivételkezelési és objektumorientált lehetőségekkel, és javítja a C nyelvvel való együttműködést (lásd [FOR 03]).

A FORTRAN tudományos céllal jött létre, így szövegfeldolgozási képességei erősen korlátozottak. Az újabb verzióknak köszönhetően ma is élő nyelv: fizikai, kémiai, statisztikai, szociológiai, meteorológiai stb. számítások egy részét ma is FORTRAN-nal végzik.

A nyelv központi honlapja: `http://www.fortran.com/`

19.15. Haskell

A Haskell megalkotásáról 1987-ben döntöttek egy funkcionális programozási konferencián (FPCA '87). A nyelvet Haskell Brooks Curry matematikusról nevezték el. Legújabb definíciója a Haskell'98 szabvány. A tervező csoport tagjai között a világ számos egyetemé-ről, illetve kutatóintézetéből találunk kutatókat (J. Hughes, S. Peyton Jones, P. Hudak, K. Hammond, E. Meijer, J. Peterson, P. Wadler és mások). A Haskell esetében alapkövetelményként fogalmazták meg, hogy a nyelv

- egyaránt alkalmas legyen oktatási és kutatási feladatok megoldására, illetve nagy alkalmazói programok fejlesztésére,
- legyen formálisan definiált szintaxisa és a szemantikája,
- szabadon elérhető legyen,
- feleljen meg széles körben elfogadott alapelveknek,
- egységes fejlődési irányt szabjon az egymástól kisebb-nagyobb mértékben különböző modern funkcionális nyelveknek.

A nyelv központi honlapja: `http://www.haskell.org/`

19.16. Java

A Java [Nyé 01] objektumorientált nyelvként indult, amit digitális TV-t vezérlő apró készülékek programozására találtak ki. Ez a projekt gyakorlatilag elhalt, de a nyelv – köszönhetően néhány fejlesztő előrelátásának – tovább élt.

A WWW születésének idejében a Netscape-nek nagyon megtetszett az új nyelv (nem utolsósorban azért, mert lehetővé tette a statikus weboldalak életre keltését), így elkezdte beépíteni a saját böngészőjébe is. Látva a Netscape törekvéseit, a Microsoft is beépítette a Java támogatást saját böngészőjébe, ami mellékhatásként a Java még könnyebb elterjedését segítette.

A Java nyelv főbb jellemzői a következők:

- Objektumelvűség. A Javában gyakorlatilag minden objektum (osztály). Nincs osztályon kívüli (globális) változó, mint például a C++-ban.
- Öröklődés. Minden objektumosztály egy öröklődési fában jól meghatározott helyen szerepel, ennek a fának a gyökere egy **Object** osztály. Nincs többszörös öröklődés, van viszont helyette **interface**.
- Többszálúság. A Java támogatja a programok több szálon való futtatását. Az egyes szálak csoportokba szervezhetőek, prioritással láthatóak el.
- Könnyen tanulhatóság. A nyelv sok elemet vett át más nyelvekből (főleg a C/C++-ból), így szintaxisának megértése nem nehéz.
- Biztonságosság. A nyelv szerkezetéből adódóan „megnehezíti" a hibás programok írását. Nincsen például pointer-aritmetika (helyette referenciák vannak).
- Kivételkezelés. Jól fejlett kivételkezelése révén a nyelv lehetőséget ad a fellépett hibák, kivételek hatékony kezelésére, ezzel is segítve a programfejlesztést és a biztonságosságot.
- Automatikus memóriakezelés. A program memóriaterületet allokálhat, de nem szabadíthat fel – ez a GC (Garbage Collector, szemétgyűjtő) dolga. Ilyen módon elkerülhető például a már felszabadított objektumokra történő hivatkozás, ami ismét csak a biztonságosságot növeli.
- Virtuális gép. A Java programok egy úgynevezett virtuális gépben futnak, ami a nyelvet maximálisan hordozhatóvá teszi.
- Szabványos objektumkönyvtárak. A Java programok írói ezeknek köszönhetően biztosan tudhatják, hogy programjaik milyen szolgáltatásokat vehetnek biztosan igénybe (például grafikus felület kezelése, I/O, stb.), ami a fejlesztést nagyban megkönnyíti.

A nyelv központi honlapja: `http://java.sun.com/`

19.17. LISP

A LISP az első funkcionális nyelv. Az 1950-es évek végén John McCarthy és csapata dolgozta ki az MIT-n. A 70-es években bekövetkezett hardverfejlődés után rövid idő alatt hatalmas népszerűségre tett szert, különösen a mesterséges intelligencia alkalmazásoknál.

Sok változat alakult ki, a ma már szabványnak tekinthető Common Lisp elég későn vált elterjedtté (lásd [Ste 90]). A Common Lispnek objektumelvű kiterjesztése is készült, a CLOS. Ez alapvetően fontos volt, indoklásként elég csak arra utalnunk, hogy az objektumelvű programozás előfutárának tekinthető kereteket már régóta sikerrel alkalmazzák a mesterséges intelligencia alkalmazásoknál.

Fontos megemlíteni, hogy a LISP interpreteres nyelv.

A nyelv központi honlapja: `http://www.lisp.org/`

19.18. Maple

A matematika alkalmazásaival foglalkozó emberek régi álma az intelligens számológép, amely nem csak számokat, hanem képleteket és formulákat is képes kezelni. A szimbolikus számítások egyik eszköze a Maple V. Ez a nyelv napjainkban az egyik legjobban használható formula-manipulációs rendszer.

A Maple V lehetőségei az intelligens számológép funkciótól a rendszer profi szintű programozásáig, az elemi kalkulus használatától a differenciálegyenletek numerikus megoldását elvégezni képes függvények leírásáig terjed. A különböző algebrai struktúrák kezeléséhez, a függvényközelítésekhez és a lineáris algebrai számításokhoz nagy segítséget nyújt. Az eredmények szemléltetését grafikus eszközök használatával teszi még érzékelhetőbbé és érthetőbbé.

A nyelv központi honlapja: `http://www.maplesoft.com/`

19.19. Modula-2

A Modula-2 nyelvet Niklaus Wirth, az ALGOL W, a Pascal és az Oberon tervezője fejlesztette ki az 1970-es években Zürichben.

A Modula-2 egyszerűen tanulható, ugyanakkor megfelelő eszközökkel rendelkezik nagy rendszerek fejlesztéséhez valamint valósidejű beágyazott rendszerek készítéséhez is.

A nyelv központi honlapja: `http://www.modula2.org/`

19.20. Modula-3

A Modula-3-t az 1980-as évek végén a DEC (Digital Equipment Corporation) és az Olivetti közös fejlesztéseként, a Modula-2-re alapozva hozták létre. Támogatja a modulokra alapozott szoftvertervezést, az egyszeres öröklődést, a kivételkezelést, a párhuzamosságot. A modulokat interfészekkel is paraméterezhetjük, így hozhatunk létre benne sablonokat.

A nyelv központi honlapja:

`http://www.research.compaq.com/SRC/modula-3/html/home.html`

19.21. Objective-C

Az Objective-C programozási nyelv a C++-hoz hasonlóan a C nyelv objektumorientált kiterjesztésének tekinthető. Fejlesztői azzal a céllal hozták létre a 80-as évek közepén, hogy a C nyelvnek egy olyan objektumorientált kiterjesztése valósuljon meg, amely kevés új nyelvi elemet tartalmaz, valamint különösen alkalmas dinamikus viselkedésű programok fejlesztésére, azaz a programozó saját felelősségére lemondhat arról, hogy a fordítóprogram működése során bizonyos programhibákat kiderítsen, ezzel szemben hatékonyabb lehet a programja. Míg a C++ a SIMULA 67-féle, az Objective-C a Smalltalk-féle objektumorientált iskolát követi.

A C++-hoz hasonlóan az Objective-C is a Java nyelv egyik ősének tekinthető, ennek ékes példája a *protocol* fogalma, amit a Javába **interface** néven vettek át a többszörös öröklődés problémáinak megkerülésére.

A nyelv központi honlapja: `http://www.dekorte.com/Objective-C/`

19.22. Pascal

A Pascal nyelvet Niklaus Wirth az ALGOL alapján 1970-re fejlesztette ki. A nyelv könnyen elsajátítható, a strukturált programtervezést jól támogatja. Az egész világon nagyon gyorsan elterjedt, hatékony eszköze a programozás oktatásának.

A nyelv központi honlapja: `http://www.merlyn.demon.co.uk/pascal.htm`

19.23. Perl

A Perl [Sch 01] nyelv első verzióját 1991-ben készítette el Larry Wall, a NASA támogatásával. A nyelvet elsősorban szöveg-feldolgozásra szánták. Sokkal fontosabbnak tekintették a hatékonyságot mind a kód megírásában, mind a kész program futtatásakor, mint a szépséget. A nyelv neve is a fentebb elmondottakra utal, hiszen a Perl a „Practical Extraction and Report Language"-ből képzett betűszó. Igaz, Larry Wall először „Pathologically Eclectic Rubbish Lister"-nek titulálta.

A nyelv nagy előnye a platformfüggetlenség: létezik az összes UNIX-ra, Linuxra, VMS-re, OS/2-re és a Microsoft termékeire is.

A Perl nyelv egy interpretált, illetve betöltéskor fordított nyelv. Eredetileg rendszeradminisztrációs feladatok megkönnyítésére írta Larry Wall, mert nem volt kedve a meglévő eszközök korlátaival bajlódni. A nyelv meglévő eszközökre lett alapozva: C, sed, awk és sh programokra. A LISP ismerete is sokat segíthet a listakezelések megértésében.

Perl-ben csak a számítógép hardver-korlátai érvényesülnek: egy teljes fájlt képes beolvasni egy szöveg változóba (ha van elég memória), tetszőleges mélységű rekurzió futtatható benne (ha van türelmünk és memóriánk). Asszociatív tömbök elérését hasítótáblákkal gyorsítja (ami meglepően hatékony programok írását teszi lehetővé). Nagyon gyors és rugalmas mintaillesztő algoritmusa van szövegek keresésére és cseréjére. Képes bináris adatokkal is dolgozni, és ezekből bonyolult adatstruktúrákat felépíteni. Az adminisztrációs feladatok

megkönnyítésére az asszociatív tömbökhöz adatbázis fájlokat rendelhetünk, melyek szerkezetét egy gyakorlott programozó maga is megadhatja. Az 5-ös verziótól kezdve már használhatjuk a moduláris programozást támogató nyelvi konstrukciókat, sőt már objektumorientált eszközöket is.

A nyelv központi honlapja: `http://www.cpan.org/`

19.24. PHP

Hivatalos nevén „PHP: Hypertext Preprocessor", azonban már régen kinőtte ezt az utótagot. Mára már a PHP a legelterjedtebb tartalomgenerátor a HTML-oldalakhoz, a PHP-t használó weboldalak száma több millióra tehető. A népszerűség oka abban keresendő, hogy a nyelv (amint azt a neve is jelzi) kezdettől fogva a HTML oldalakba ágyazásra lett tervezve, a fejlesztőkörnyezetek is eleve úgy vannak kialakítva, hogy webszerverhez kapcsolódnak, és a programot ezen keresztül futtatják, az eredményt pedig weboldalként jelenítik meg.

A széleskörű használat következményeként rengeteg kiegészítése készült, adatbáziskezeléstől képkonvertáláson át grafikus kezelőfelületig sok mindent tudunk készíteni a segítségével.

A PHP a Java, C és Perl nyelvekre épül. A Perl nyelvhez való hasonlósága mind szintaktikájában, mind kialakításában nagyon szembetűnő (akár egy valamelyest átalakított Perlként is jellemezhető – a Perl nyelvnek is létezik hasonló kiegészítése Embedded Perl néven).

A nyelv központi honlapja: `http://www.php.net/`

19.25. PL/I

Az ALGOL, a FORTRAN és a COBOL előnyös tulajdonságainak egyesítésével tervezték meg az 1960-as évek elején a PL/I-t, abban a reményben, hogy ez lesz majd az igazi programozási nyelv, a *Programming Language No. 1*. A nyelv rendelkezik az ALGOL 60 nyelv blokkstruktúrájával, vezérlési utasításaival, a FORTRAN *FORMAT*-os input-outputjával, valamint a COBOL hordozható típusaival és fájl-kezelésével. Ezeken kívül bevezették a dinamikus adatszerkezetek kezelésére kialakított pointereket, a rugalmas stringműveleteket, és hibakezelésre alkalmas mechanizmust. Lehetővé tették az eljárások önálló fordítását is (lásd [Koz 92a]. A nyelvnek még egy fokozatosan felépített részhalmaz sorozatát (ez volt az SP/k k=1..8-ra) is létrehozták, hogy támogassák a programozás oktatását.

A nyelv központi honlapja: `http://home.nycap.rr.com/pflass/pli.htm`

19.26. Python

Guido van Rossum és csapata 1991 óta fejleszti a Python nyelvet, ezt a hatékony, dinamikus típusossága miatt is külön figyelemre méltó, objektumorientált szkriptnyelvet.

A nyelv központi honlapja: `http://www.python.org/`

19.27. SIMULA 67

A SIMULA 67 (SIMUlation LAnguage) nyelvet Dahl, Myhrhaug és Nygaard tervezte (lásd [Dah 85]). Az ALGOL 60 továbbfejlesztéseként olyan nyelvet alkottak, melyben hatékonyan lehet komplex, interaktív rendszereket szimulálni.

Ez az első objektumorientált nyelv, itt vezették be az osztály, az öröklődés – a SIMULA szóhasználatával: prefixelés –, a polimorfizmus és a dinamikus kötés fogalmait.

A kétirányú ciklikus listákat kezelő szabványos *SIMSET* osztály, valamint az ezt felhasználó *SIMULATION* osztály, mely nyelvi eszközöket vezet be egy folyamat modellezésére, központi szerepet kap a szimulációk megvalósításában.

A nyelv központi honlapja:

`http://www.engin.umd.umich.edu/CIS/course.des/cis400/simula/simula.html`

19.28. Smalltalk

Az 1970-es évek elején a Xerox Palo Alto Research Center kutatóközpontban Alan Kay, Daniel H. Ingalls és Adele Goldberg vezetésével egy olyan rendszer kialakításán dolgoztak, amellyel az emberek hatásosan és könnyen kezelhetnék a számítógépet. Így született meg a Smalltalk, mely noha a SIMULA 67 után a második objektumorientált nyelv, de az első, amelyet teljesen ezen paradigma alapján terveztek. A Smalltalkban minden objektum – az osztályok is, a kódblokkok is, sőt a Smalltalk fordító is. A nyelvnek objektumoktól független része minimális, még a hagyományos vezérlési szerkezeteket (elágazás, ciklus) is mint objektumnak küldött üzeneteket valósították meg, szinte az értékadás az egyetlen művelet, amit nem így kezelnek.

A Smalltalk volt az első nyelv, amelyhez integrált fejlesztői környezetet is terveztek, s ez adta az alapját a későbbi ablakozó rendszerek kidolgozásának.

A nyelv központi honlapja: `http://www.smalltalk.org/`

19.29. SML

Az első típusos funkcionális nyelv az ML (Meta Language). Ez eredetileg az Edinburghban tételbizonyításra tervezett LCF (Logic for Computable Functions) rendszer metanyelve volt. Az ML nyelvet R. Milner tervezte a 70-es évek közepén, az SML-t (Standard ML) a Hope (Burstall, 1980) létrehozása után alkotta Milner, Tofte és Harper 1983 és 1990 között. Az SML legújabb, átdolgozott szabványa 1997-ben készült el [Mil 97, Har 01, Han 00]. ML-változat a Caml (INRIA, 1984-1990, a Coq tételbizonyító alapnyelve) és az Objective Caml (a Caml Light továbbfejlesztése, INRIA, 1990-). Az ML-változatok nem tisztán funkcionális nyelvek, imperatív nyelvi elemeket (például frissíthető változókat) is tartalmaznak.

A nyelv központi honlapja: `http://www.standardml.org/`

19.30. SQL

A relációs adatbáziskezelés a 80-as években terjedt el, és vele párhuzamosan az SQL. Az SQL szó a Structured Query Language (Strukturált Lekérdező Nyelv) megnevezés rövidítése. A név is mutatja, hogy a relációs adatbáziskezelés egyik kulcskérdése, hogy mennyire hatékonyan lehetséges az adattáblákból a kívánt eredményt lekérdezni.

Az SQL nyelv két fő részből áll: a DML (Data Manipulation Language, azaz Adatkezelő Nyelv) az adatok lekérdezésére, valamint adattáblákba való beszúrására, módosítására és törlésére szolgál; a DDL (Data Definition Language, azaz Adat-definíciós Nyelv) az adattáblák, nézetek, indexek és egyéb adatbázisbeli objektumok létrehozására, módosítására és törlésére szolgál.

Az alap (ANSI szabvány) SQL nyelv csupán alapvető, deklaratív elemeket tartalmaz, nem célja, hogy összetett algoritmusokat lehessen vele megvalósítani. Emiatt számos olyan kiegészítés, más nyelvvel való integrált megoldás született, amely az imperatív, procedurális nyelvek előnyeit az SQL adatbáziskezelő képességeivel egyesíti. Az ilyen megoldásokat beágyazott SQL-nek (embedded SQL) hívják.

A beágyazott SQL-nek két fő fajtája van: az egyiknél az adatbáziskezelő rendszeren belüli, úgynevezett tárolt eljárások tartalmazzák az SQL utasításokat (például az Oracle PL/SQL és a Microsoft Transact-SQL nyelvek esetében); a másik megközelítés szerint az adatbáziskezelő rendszeren kívüli API segítségével tudunk SQL utasításokat kiadni és az adatbázishoz kapcsolódni, egy általános célú nyelvbe ágyazottan (például Pro C, ODBC, JDBC segítségével).

Az objektum-relációs és objektumorientált adatbáziskezelők elterjedésének következtében az SQL továbbfejlesztésre került, valamint jónéhány, az SQL-t alapul vevő, vagy arra hasonlító új adatbáziskezelő nyelv alakult ki. Mindezek ellenére továbbra is az SQL a legszélesebb körben alkalmazott nyelv az adatbázisokban tárolt információ lekérdezésére és manipulációjára.

A nyelv központi honlapja: `http://www.sql.org/`

19.31. Tcl

A Tcl (Tool command language) egy egyszerű interpretált nyelv, amely kevés, de sokoldalú elemre épít. A LISP, a C és a unix C-shell egyes elemeit egyesíti.

Magasszintűsége folytán gyors és kényelmes programfejlesztést tesz lehetővé. Mindez nem kell, hogy a hatékonyság rovására legyen, mert a Tcl szoros szimbiózisban él a C-vel. Sőt, ha egy alkalmazás alapvető funkcionalitását C nyelven, magasabb szintű részét és felhasználói felületét pedig Tcl-ben írjuk meg, a Tcl-t mint makrónyelvet adhatjuk a felhasználó kezébe, nagyfokú rugalmasságot biztosítva ezzel.

A Tcl legfontosabb kiterjesztése a Tk toolkit, amellyel a Tcl alkalmassá válik grafikus felhasználói felületek építésére.

Amit a nyelv nem támogat: absztrakt adattípusok, operátorok túlterhelése, objektumok, osztályok, öröklés, OOP, helyességbizonyítás, párhuzamosság, perzisztencia.

A nyelv központi honlapja: `http://www.tcl.tk/`

20 Függelék

Dec	Hex	Kar		Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar
0	0x00	NUL	Null	32	0x20	Space	64	0x40	@	96	0x60	‘
1	0x01	SOH	Start of heading	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	Start of text	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	End of text	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	End of transmission	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	Enquiry	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	Acknowledge	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	Bell	39	0x27	’	71	0x47	G	103	0x67	g
8	0x08	BS	Backspace	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT	Horizontal tab	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0a	LF	Line feed, new line	42	0x2a	*	74	0x4a	J	106	0x6a	j
11	0x0b	VT	Vertical tab	43	0x2b	+	75	0x4b	K	107	0x6b	k
12	0x0c	FF	Form feed, new page	44	0x2c	,	76	0x4c	L	108	0x6c	l
13	0x0d	CR	Carriage return	45	0x2d	-	77	0x4d	M	109	0x6d	m
14	0x0e	SO	Shift out	46	0x2e	.	78	0x4e	N	110	0x6e	n
15	0x0f	SI	Shift in	47	0x2f	/	79	0x4f	O	111	0x6f	o
16	0x10	DLE	Data link escape	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	Device control 1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	Device control 2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	Device control 3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	Device control 4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	Negative acknowledge	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	Synchronous idle	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	End of trans. block	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	Cancel	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	End of medium	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1a	SUB	Substitute	58	0x3a	:	90	0x5a	Z	122	0x7a	z
27	0x1b	ESC	Escape	59	0x3b	;	91	0x5b	[	123	0x7b	{
28	0x1c	FS	File separator	60	0x3c	<	92	0x5c	\	124	0x7c	|
29	0x1d	GS	Group separator	61	0x3d	=	93	0x5d	]	125	0x7d	}
30	0x1e	RS	Record separator	62	0x3e	>	94	0x5e	^	126	0x7e	~
31	0x1f	US	Unit separator	63	0x3f	?	95	0x5f	_	127	0x7f	DEL

20.1. **táblázat:** Az ASCII kódtábla

Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar
160	0xa0		176	0xb0	°	192	0xc0	À	208	0xd0	Ð	224	0xe0	à	240	0xf0	ð
161	0xa1	¡	177	0xb1	±	193	0xc1	Á	209	0xd1	Ñ	225	0xe1	á	241	0xf1	ñ
162	0xa2	¢	178	0xb2	²	194	0xc2	Â	210	0xd2	Ò	226	0xe2	â	242	0xf2	ò
163	0xa3	£	179	0xb3	³	195	0xc3	Ã	211	0xd3	Ó	227	0xe3	ã	243	0xf3	ó
164	0xa4	¤	180	0xb4	´	196	0xc4	Ä	212	0xd4	Ô	228	0xe4	ä	244	0xf4	ô
165	0xa5	¥	181	0xb5	µ	197	0xc5	Å	213	0xd5	Õ	229	0xe5	å	245	0xf5	õ
166	0xa6	¦	182	0xb6	¶	198	0xc6	Æ	214	0xd6	Ö	230	0xe6	æ	246	0xf6	ö
167	0xa7	§	183	0xb7	·	199	0xc7	Ç	215	0xd7	×	231	0xe7	ç	247	0xf7	÷
168	0xa8	¨	184	0xb8	¸	200	0xc8	È	216	0xd8	Ø	232	0xe8	è	248	0xf8	ø
169	0xa9	©	185	0xb9	¹	201	0xc9	É	217	0xd9	Ù	233	0xe9	é	249	0xf9	ù
170	0xaa	ª	186	0xba	º	202	0xca	Ê	218	0xda	Ú	234	0xea	ê	250	0xfa	ú
171	0xab	«	187	0xbb	»	203	0xcb	Ë	219	0xdb	Û	235	0xeb	ë	251	0xfb	û
172	0xac	¬	188	0xbc	¼	204	0xcc	Ì	220	0xdc	Ü	236	0xec	ì	252	0xfc	ü
173	0xad	-	189	0xbd	½	205	0xcd	Í	221	0xdd	Ý	237	0xed	í	253	0xfd	ý
174	0xae	®	190	0xbe	¾	206	0xce	Î	222	0xde	Þ	238	0xee	î	254	0xfe	þ
175	0xaf	¯	191	0xbf	¿	207	0xcf	Ï	223	0xdf	ß	239	0xef	ï	255	0xff	ÿ

20.2. táblázat: Az ISO 8859 1 (Latin-1) kódtábla ASCII-től eltérő nyomtatható karakterei

Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar
160	0xa0		176	0xb0	°	192	0xc0	Ŕ	208	0xd0	Đ	224	0xe0	ŕ	240	0xf0	đ
161	0xa1	Ą	177	0xb1	ą	193	0xc1	Á	209	0xd1	Ń	225	0xe1	á	241	0xf1	ń
162	0xa2	˘	178	0xb2	˛	194	0xc2	Â	210	0xd2	Ň	226	0xe2	â	242	0xf2	ň
163	0xa3	Ł	179	0xb3	ł	195	0xc3	Ă	211	0xd3	Ó	227	0xe3	ă	243	0xf3	ó
164	0xa4	¤	180	0xb4	´	196	0xc4	Ä	212	0xd4	Ô	228	0xe4	ä	244	0xf4	ô
165	0xa5	Ľ	181	0xb5	ľ	197	0xc5	Ĺ	213	0xd5	Ő	229	0xe5	í	245	0xf5	ő
166	0xa6	Ś	182	0xb6	ś	198	0xc6	Ć	214	0xd6	Ö	230	0xe6	ć	246	0xf6	ö
167	0xa7	§	183	0xb7	ˇ	199	0xc7	Ç	215	0xd7	×	231	0xe7	ç	247	0xf7	÷
168	0xa8	¨	184	0xb8	¸	200	0xc8	Č	216	0xd8	Ř	232	0xe8	č	248	0xf8	ř
169	0xa9	Š	185	0xb9	š	201	0xc9	É	217	0xd9	Ů	233	0xe9	é	249	0xf9	ů
170	0xaa	Ş	186	0xba	ş	202	0xca	Ę	218	0xda	Ú	234	0xea	ę	250	0xfa	ú
171	0xab	Ť	187	0xbb	ť	203	0xcb	Ë	219	0xdb	Ű	235	0xeb	ë	251	0xfb	ű
172	0xac	Ź	188	0xbc	ź	204	0xcc	Ě	220	0xdc	Ü	236	0xec	ě	252	0xfc	ü
173	0xad	-	189	0xbd	˝	205	0xcd	Í	221	0xdd	Ý	237	0xed	í	253	0xfd	ý
174	0xae	Ž	190	0xbe	ž	206	0xce	Î	222	0xde	Ţ	238	0xee	î	254	0xfe	ţ
175	0xaf	Ż	191	0xbf	ż	207	0xcf	Ď	223	0xdf	ß	239	0xef	ď	255	0xff	˙

20.3. táblázat: Az ISO 8859 2 (Latin-2) kódtábla ASCII-től eltérő nyomtatható karakterei

Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar
128	0x80	Ç	160	0xa0	á	192	0xc0	└	224	0xe0	α
129	0x81	ü	161	0xa1	í	193	0xc1	┴	225	0xe1	ß
130	0x82	é	162	0xa2	ó	194	0xc2	┬	226	0xe2	Γ
131	0x83	â	163	0xa3	ú	195	0xc3	├	227	0xe3	π
132	0x84	ä	164	0xa4	ñ	196	0xc4	─	228	0xe4	Σ
133	0x85	à	165	0xa5	Ñ	197	0xc5	┼	229	0xe5	σ
134	0x86	å	166	0xa6	ª	198	0xc6	╞	230	0xe6	µ
135	0x87	ç	167	0xa7	º	199	0xc7	╟	231	0xe7	γ
136	0x88	ê	168	0xa8	¿	200	0xc8	╚	232	0xe8	Φ
137	0x89	ë	169	0xa9	⌐	201	0xc9	╔	233	0xe9	θ
138	0x8a	è	170	0xaa	¬	202	0xca	╩	234	0xea	Ω
139	0x8b	ï	171	0xab	½	203	0xcb	╦	235	0xeb	δ
140	0x8c	î	172	0xac	¼	204	0xcc	╠	236	0xec	∞
141	0x8d	ì	173	0xad	¡	205	0xcd	═	237	0xed	φ
142	0x8e	Ä	174	0xae	«	206	0xce	╬	238	0xee	ε
143	0x8f	Å	175	0xaf	»	207	0xcf	╧	239	0xef	∩
144	0x90	É	176	0xb0	░	208	0xd0	╨	240	0xf0	≡
145	0x91	æ	177	0xb1	▒	209	0xd1	╤	241	0xf1	±
146	0x92	Æ	178	0xb2	▓	210	0xd2	╥	242	0xf2	≥
147	0x93	ô	179	0xb3	│	211	0xd3	╙	243	0xf3	≤
148	0x94	ö	180	0xb4	┤	212	0xd4	╘	244	0xf4	⌠
149	0x95	ò	181	0xb5	╡	213	0xd5	╒	245	0xf5	⌡
150	0x96	û	182	0xb6	╢	214	0xd6	╓	246	0xf6	÷
151	0x97	ù	183	0xb7	╖	215	0xd7	╫	247	0xf7	≈
152	0x98	ÿ	184	0xb8	╕	216	0xd8	╪	248	0xf8	°
153	0x99	Ö	185	0xb9	╣	217	0xd9	┘	249	0xf9	∙
154	0x9a	Ü	186	0xba	║	218	0xda	┌	250	0xfa	•
155	0x9b	¢	187	0xbb	╗	219	0xdb	█	251	0xfb	√
156	0x9c	£	188	0xbc	╝	220	0xdc	▄	252	0xfc	^{n}
157	0x9d	¥	189	0xbd	╜	221	0xdd	▌	253	0xfd	^{2}
158	0x9e	Pt	190	0xbe	╛	222	0xde	▐	254	0xfe	■
159	0x9f	ƒ	191	0xbf	┐	223	0xdf	▀	255	0xff	

20.4. táblázat: Az IBM Codepage 437 karakterei a 128-as pozíciótól

Dec	Hex	Kar		Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar
0	0x00	NUL	Null	64	0x40		128	0x80		192	0xc0	{
1	0x01	SOH	Start of heading	65	0x41		129	0x81	a	193	0xc1	A
2	0x02	STX	Start of text	66	0x42		130	0x82	b	194	0xc2	B
3	0x03	ETX	End of text	67	0x43		131	0x83	c	195	0xc3	C
4	0x04	PF	Punch off	68	0x44		132	0x84	d	196	0xc4	D
5	0x05	HT	Horizontal tab	69	0x45		133	0x85	e	197	0xc5	E
6	0x06	LC	Lower case	70	0x46		134	0x86	f	198	0xc6	F
7	0x07	DEL	Delete	71	0x47		135	0x87	g	199	0xc7	G
8	0x08	GE		72	0x48		136	0x88	h	200	0xc8	H
9	0x09	RLF		73	0x49		137	0x89	i	201	0xc9	I
10	0x0a	SMM	Start of manual message	74	0x4a	¢	138	0x8a		202	0xca	
11	0x0b	VT	Vertical tab	75	0x4b	.	139	0x8b		203	0xcb	
12	0x0c	FF	Form feed	76	0x4c	<	140	0x8c		204	0xcc	
13	0x0d	CR	Carriage return	77	0x4d	(	141	0x8d		205	0xcd	
14	0x0e	SO	Shift out	78	0x4e	+	142	0x8e		206	0xce	
15	0x0f	SI	Shift in	79	0x4f	|	143	0x8f		207	0xcf	
16	0x10	DLE	Data link escape	80	0x50	&	144	0x90		208	0xd0	}
17	0x11	DC1	Device control 1	81	0x51		145	0x91	j	209	0xd1	J
18	0x12	DC2	Device control 2	82	0x52		146	0x92	k	210	0xd2	K
19	0x13	TM	Tape mark	83	0x53		147	0x93	l	211	0xd3	L
20	0x14	RES	Restore	84	0x54		148	0x94	m	212	0xd4	M
21	0x15	NL	New line	85	0x55		149	0x95	n	213	0xd5	N
22	0x16	BS	Backspace	86	0x56		150	0x96	o	214	0xd6	O
23	0x17	IL	Idle	87	0x57		151	0x97	p	215	0xd7	P
24	0x18	CAN	Cancel	88	0x58		152	0x98	q	216	0xd8	Q
25	0x19	EOM	End of medium	89	0x59		153	0x99	r	217	0xd9	R
26	0x1a	CC	Cursor control	90	0x5a	!	154	0x9a		218	0xda	
27	0x1b	CU1	Customer use 1	91	0x5b	$	155	0x9b		219	0xdb	
28	0x1c	IFS	Interchange file separator	92	0x5c	*	156	0x9c		220	0xdc	
29	0x1d	IGS	Interchange group separator	93	0x5d	)	157	0x9d		221	0xdd	
30	0x1e	IRS	Interchange record	94	0x5e	;	158	0x9e		222	0xde	
31	0x1f	IUS	Interchange unit separator	95	0x5f	¬	159	0x9f		223	0xdf	

20.5. táblázat: Az EBCDIC kódtábla

Dec	Hex	Kar		Dec	Hex	Kar	Dec	Hex	Kar	Dec	Hex	Kar
32	0x20	DS	Digit select	96	0x60	-	160	0xa0		224	0xe0	\
33	0x21	SOS	Start of significance	97	0x61	/	161	0xa1	~	225	0xe1	
34	0x22	FS	Field separator	98	0x62		162	0xa2	s	226	0xe2	S
35	0x23			99	0x63		163	0xa3	t	227	0xe3	T
36	0x24	BYP	Bypass	100	0x64		164	0xa4	u	228	0xe4	U
37	0x25	LF	Line feed	101	0x65		165	0xa5	v	229	0xe5	V
38	0x26	ETB	End of transmission block	102	0x66		166	0xa6	w	230	0xe6	W
39	0x27	ESC	Escape	103	0x67		167	0xa7	x	231	0xe7	X
40	0x28			104	0x68		168	0xa8	y	232	0xe8	Y
41	0x29			105	0x69		169	0xa9	z	233	0xe9	Z
42	0x2a	SM	Set mode	106	0x6a		170	0xaa		234	0xea	
43	0x2b	CU2	Customer use 2	107	0x6b	,	171	0xab		235	0xeb	
44	0x2c			108	0x6c	%	172	0xac		236	0xec	
45	0x2d	ENQ	Enquiry	109	0x6d	_	173	0xad		237	0xed	
46	0x2e	ACK	Acknowledge	110	0x6e	>	174	0xae		238	0xee	
47	0x2f	BEL	Bell	111	0x6f	?	175	0xaf		239	0xef	
48	0x30			112	0x70		176	0xb0		240	0xf0	0
49	0x31			113	0x71		177	0xb1		241	0xf1	1
50	0x32	SYN	Synchronous idle	114	0x72		178	0xb2		242	0xf2	2
51	0x33			115	0x73		179	0xb3		243	0xf3	3
52	0x34	PN	Punch on	116	0x74		180	0xb4		244	0xf4	4
53	0x35	RS	Reader stop	117	0x75		181	0xb5		245	0xf5	5
54	0x36	UC	Upper case	118	0x76		182	0xb6		246	0xf6	6
55	0x37	EOT	End of transmission	119	0x77		183	0xb7		247	0xf7	7
56	0x38			120	0x78		184	0xb8		248	0xf8	8
57	0x39			121	0x79		185	0xb9		249	0xf9	9
58	0x3a			122	0x7a	:	186	0xba		250	0xfa	|
59	0x3b	CU3	Customer use 3	123	0x7b	#	187	0xbb		251	0xfb	
60	0x3c	DC4	Device control 4	124	0x7c	@	188	0xbc		252	0xfc	
61	0x3d	NAK	Negative acknowledge	125	0x7d	'	189	0xbd		253	0xfd	
62	0x3e			126	0x7e	=	190	0xbe		254	0xfe	
63	0x3f	SUB	Substitute	127	0x7f	"	191	0xbf		255	0xff	EO

20.6. táblázat: Az EBCDIC kódtábla (folytatás)

Irodalomjegyzék

[Aba 98] Abadi, M. – Cardelli, L.: *A Theory of Objects.*
Corrected second printing, Springer-Verlag, New York, 1998.

[Ach 95] Achten, P. M. – Plasmeijer, M. J.:
Concurrent Interactive Processes in a Pure Functional Language.
van Vliet, J. C. ed.: Proceedings Computing Science in the Netherlands, CSN'95, Jaarbeurs Utrecht, The Netherlands, November 27–28 1995, Stichting Mathematisch Centrum, Amsterdam, pp. 10–21.

[Ach 00] Achten, P. – Wierich, M.: *A Tutorial to the Clean Object I/O Library.*
University of Nijmegen, Netherlands, 2000.
`http://www.cs.kun.nl/~clean`

[Ack 79] Ackermann, W. B. et al.: *A Value-Oriented Algorithmic Language.*
MIT Laboratory of Computer Science, TR-218, June 1979.

[ACM 89] *Special Issue on Programming Language Paradigms.*
ACM Computing Surveys vol. 21 no. 3, 1989.

[Ada 83] *Ada '83 Language Reference Manual.*
Ada Information Clearinghouse (publ.), 2003.
`http://archive.adaic.com/standards/83lrm/html/lrm-02-01.html`

[Ada 95] *Annotated Ada Reference Manual*, ISO/IEC 8652:1995.
`http://www.ada95.com`

[Aho 88] Aho, A. V. – Kernighan, B. W. – Weinberger, P. J.:
The AWK Programming Language. Addison-Wesley, Reading, Mass., 1988.

[Ale 01] Alexandrescu, A.:
Modern C++ Design (Generic Programming and Design Patterns Applied).
Addison-Wesley Professional, Reading, Mass., 2001.

[Ame 87] America, P.: *POOL-T: A parallel object-oriented language.*
Object-Oriented Concurrent Programming, MIT Press, Cambridge, Mass, pp. 199–220., 1987.

[And 91] Andrews, G. R.: *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, 1991.

[Ang 97] Angster Erzsébet: *Az objektumorientált tervezés és programozás alapjai.* Gábor Dénes Főiskola, Budapest, 1997.

[AOSD 03] *Az Aspect-Oriented Software Development honlapja.* `http://www.aosd.net/`, 2003.

[Arv 78] Arvind et al.: *An Asynchronous Programming Language for a Large Multiprocessor Machine.* TR114a, Dept ISC, UC Irvine, Dec 1978.

[Aug 99] Augustsson, L.: *Cayenne.* Göteburgs Universitet, Sweden, 1999. `http://www.math.chalmers.se/~augustss/cayenne/`

[Bak 74] Bakos Tamás: *A COBOL programozási nyelv.* Műszaki Könyvkiadó, Budapest, 1974.

[Bar 84] Barendregt, H. P.: *The Lambda-Calculus, its Syntax and Semantics.* Amsterdam, North-Holland, 1984.

[Bar 90] Barr, M. – Wells, Ch.: *Category Theory for Computer Science.* Prentice Hall, London, 1990.

[Bar 92] Barrett, G.: *Occam3 Reference Manual.* SGS-Thomson Microelectronics, Bristol, 1992.

[Bar 96] Barnes, J.: *Programming in Ada95.* Springer-Verlag, New-York, 1996.

[Bar 01] Barr, G.: *Error – Error/Exception Handling in an OO-ish Way.* The Comprehensive Perl Archive Network, 2001. `http://search.cpan.org/author/UARUN/Error-0.15/Error.pm`

[Bír 97] Bíró Ernő – Kovács Lehel: *A programozási nyelvek alapjai.* Komp-Press, Kolozsvár, 1997.

[Bla 90] Blaschek, G. – Pomberger, G.: *Introduction to programming with Modula-2.* Springer-Verlag, Berlin, 1990.

[Blo 01] Bloch, J.: *Effective Java Programming Language Guide.* Addison-Wesley Professional, Reading, Mass., 2001.

[Bor 99] *Borland Delphi 5 Quick Start.* `http://info.borland.com/techpubs/delphi/delphi5/`, 1999.

[Bow 02] Bowen, J.: *Logic Programming.* `http://www.comlab.ox.ac.uk/archive/logic-prog.html`, 2002.

[Böh 66] Böhm, D. – Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, vol. 9, no. 5, pp. 366–371., 1966.

[Bösz 96] Böszörményi, L. – Weich, C.: *Programmieren mit Modula-3.* Springer-Verlag, Berlin, 1996. In English: *Programming in Modula-3.* Springer-Verlag, Berlin, 1996.

[Bra 79] Brailsford, D. F. – Walker, A. N.: *Introductory Algol 68 programming.* Ellis Horwood series in Computer Science, Chichester, 1979.

[Bra 98] Bracha, G. – Odersky, M. – Stoutamire, D. – Wadler, P.: *Gj: Extending the Java Programming Language with Type Parameters.* Unpublished draft, March 1998.

[Bra 99] Brandel, M.: *1963: ASCII Debuts.* 1999. `http://www.bobbemer.com/brandela.htm`

[Bri 75] Brinch Hansen, P.: The Programming Langugage Concurrent Pascal. IEEE Transactions on Software Engineering, vol. SE-1, no. 2., pp. 199–207., 1975.

[Bru 02] Bruce, K. B.: *Foundations of Object-Oriented Languages and Types.* MIT Press, Cambridge, Massachusetts, 2002.

[Bud 91] Budd, T.: *An Introduction to Object-Oriented Analysis.* Addison-Wesley, Reading, Mass., 1991.

[Bus 85] Businger, A.: *PORTAL Language Description.* Springer-Verlag, LNCS 198, Berlin, 1985.

[C++ 98] *C++ Standard ISO/IEC 14882:1998(E).* ISO, 1998. `http://www.comnets.rwth-aachen.de/doc/c++std/`

[C# Ref 03] *C# Programmer's Reference.* `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcoriCProgrammersReference.asp`, 2003.

[Can 00] Cantù, M.: *Delphi 5 Mesteri szinten.* Kiskapu Kiadó, Budapest, 2000.

[Can 03] Cantù, M.: *Essential Pascal.* `http://www.marcocantu.com/`, 2003.

[Car 85] Cardelli, L. – Wegner, P.: On Understanding of Types, Data Abstraction and Polymorphism. *ACM Computing Surveys,* vol. 17, no. 4, pp. 471–522., December 1985.

[Cat 01] Catambay, B.: *Pascal standards.* Pascal ISO 7185:1990, Extended Pascal ISO 10206:1990. `www.pascal-central.com/standards.html`, 2001.

[Cez 95] Cezzar, R.: *A Guide to Programming Languages: Overview and Comparison.* Artech House, Boston, 1995. Series title: The Artech House Computer Science Library.

[CF 01] *Composition Filters.* `http://trese.cs.utwente.nl/composition_filters/`, 2001.

[Char 02] *Character Set Standards.* `http://www.diffuse.org/chars.html`, 2002.

[Cla 95] Clack, C. – Myers, C. – Poon, E.: *Programmieren in Miranda.* Prentice Hall, München, 1995.

[Coh 96] Cohen, N. H.: *Ada as a Second Language.* 2nd ed., McGraw-Hill, New York, 1996.

[Col 82] Colmerauer A.: Prolog and Infinite Trees. *Logic Programming,* Clark, K. L. – Tarnlund, S.–A. (eds) pp. 231–251., Academic Press, London, 1982.

[Cso 92] Csombók, J: *Párhuzamos programok írását támogató nyelvi elemek.* Szakdolgozat, ELTE TTK ITCS, 1992.

[Csö 96] Csörnyei, Z.: *Bevezetés a fordítóprogramok elméletébe, I–II.* Egyetemi jegyzet, ELTE TTK ITCS, Budapest, 1996.

[Csö 01] Csörnyei Z.: *Lambda-kalkulus.* Előadás jegyzet (kézirat). `http://people.inf.elte.hu/csz/lk-jegyzet.html`

[Cyb 83] *CDC Cyber 200 FORTRAN Reference Manual.* Sunnyvale, California, 1983.

[Cza 00] Czarnecki, K. – Eisenecker, U.: *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, Reading, Mass., 2000.

[Czy 98a] Czyborra, R.: *The ISO 8859 Alphabet Soup.* `http://czyborra.com/charsets/iso8859.html`, 1998.

[Czy 98b] Czyborra, R.: *Unicode Transformation Formats: UTF-8 & Co.* `http://czyborra.com/utf/`, 1998.

[Dah 72] Dahl, O. J. – Dijkstra, E. W. – Hoare, C. A. R.: *Structured Programming.* Academic Press, New York, 1972.
Magyarul: *Strukturált programozás.* Műszaki Könyvkiadó, Budapest, 1978.

[Dah 85] Dahl, O. J. – Myhrhaug, B. – Nygaard, K.: *SIMULA – Common Base Language.* Revised ed., Norwegian Computing Center, Oslo, October 1970.
Magyarul: *A SIMULA 67 nyelv definíciója.* Budapesti Műszaki Egyetem, 1985.

[DeG 86] DeGroot, D. – Lindstrom, G.: *Logic Programming: Functions, Relations and Equations.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[Dem 03] *Demeter: Aspect-Oriented Software Development.* `http://www.ccs.neu.edu/research/demeter/`, 2003.

[Der 93] Deransart, P. – Maluszyńsky, J.: *A Grammatical View of Logic Programming.* MIT Press, Cambridge, Massachusetts, 1993.

[Der 96] Deransart, P. – Ed-Dbali, A. A. – Cervoni, L.: *Prolog: The Standard (Reference Manual).* Springer-Verlag, Berlin, 1996.

[Dij 62] Dijkstra, E. W.: *A Primer of ALGOL 60 Programming, Together with Report on the Algorithmic Language ALGOL 60.* Published for the Automatic Programming Information Centre, Brighton College of Technology, Eng., by Academic Press, London, New York, 1962.

[Dij 68] Dijkstra, E. W.: Co-operating Sequential Processes.
Genuys, F. (ed.): *Programming Languages,*, pp. 43–112., Academic Press, New York, 1968.

[Dij 76] Dijkstra, E. W.: *A Discipline of Programming.*
Prentice Hall, Englewood Cliffs, NJ, 1976.

[Don 80] Donzeau-Gouge et al.: *Formal Definition of the ADA Programming Language.*
Honeywell, Cii Honeywell Bull, INRIA, 1980.

[Ecl 03] *Az Eclipse projekt honlapja.*
`http://www.eclipse.org/`, 2003.

[Eek 90] van Eekelen, M. et al.: *Concurrent Clean.*
Technical Report no. 90–20, University of Nijmegen, November 1990.

[Eck 00] Eckel, B.: *Thinking in C++, vol. I.: Introduction to Standard C++.*
2nd ed., Prentice Hall, Englewood Cliffs, NJ, 2000.

[EJB 03] *Enterprise JavaBeans Specification.*
Version 2.1, Sun Microsystems, 2003.
`http://java.sun.com/products/ejb/docs.html`

[Fek 90] Fekete István – Gregorics Tibor – Nagy Sára:
Bevezetés a mesterséges intelligenciába. Műszaki Könyvkiadó, Budapest, 1990.

[Fin 94] Finne, S. – Jones, S. P.: *Programming Reactive Systems in Haskell.*
Proceedings of the Glasgow Functional Programming Group Workshop, Ayr, Scotland, Sept. 1994, Springer-Verlag, Berlin.

[FOR 03] *J3 – Fortran Standards.*
`http://www.j3-fortran.org`, 2003.

[Fót 83] Fóthi Ákos: *Bevezetés a programozáshoz.*
Egyetemi jegyzet, ELTE TTK, Budapest, 1983.

[Fou 01] Fournet, C. – Le Fessant, F. – Maranget, L. – Schmitt, A.:
The JoCaml language – Documentation and User's Manual.
INRIA Rocquencourt, Jan. 2001.

[Fri 01] Friedman, D. P.: *Essentials of Programming Languages.*
2nd ed., MIT Press, Cambridge, Mass., 2001.

[Fro 96] Frohner Ákos:
Párhuzamos programozást támogató nyelvi eszközök összehasonlítása.
Diplomamunka, ELTE TTK ITCS, 1996.

[Gam 95] Gamma, E. – Helm, R. – Johnson, R. – Vlissides, J.:
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, Reading, Mass., 1995.

[Gol 83] Goldberg, A. – Robson, D.:
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley, Reading, Mass., 1983.

[Gom 97] Gomes, B. – Stoutamire, D. – Weissman, B. – Klawitter, H.: *A Language Manual for Sather 1.1.* International Computer Science Institute, TR 97-037, Oct. 1997

[Gre 02] Green, R.: *How To Write Unmaintainable Code.* Canadian Mind Products, 2002. `http://mindprod.com/unmain.html`

[Gri 81] Gries, D.: *The Science of Programming.* Springer-Verlag, New York, 1981.

[Gru 90] Grund, F. – Issel, W.: *PL/I programmierung.* Hüthig Verlag, Heidelberg, 1990.

[Guly 99a] Gulyás László – Kozsik Tamás: The Use of Aspect-Oriented Programming in Scientific Simulations. Jaan Penjam (ed.): *Software Technology, Fenno-Ugric Symposium FUSST'99 Proceedings*, Technical Report CS 104/99, Tallin, Estonia, pp. 17–28., August 19–21, 1999.

[Guly 99b] Gulyás László – Tatai Gábor: *Ágensek és multi-ágens rendszerek.* Futó Iván et. al.: Mesterséges intelligencia, pp. 709–754., Aula Kiadó, Budapest, 1999.

[Gun 94] Gunter, C. A. – Mitchell, J. C. (eds.): *Theoretical Aspects of Object-Oriented Programming.* MIT Press, Cambridge, Massachusetts, 1994.

[Han 72] Hansen, P. B.: Structured multiprogramming. *Communications of the ACM,* vol. 15, pp. 574–578., ACM Press, New York, 1972.

[Han 94] Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming,* Elsevier Science Publishing Co., Inc., New York, 1994.

[Han 00] Hanák Péter: *Deklaratív programozás.* Oktatási segédlet, BME, Budapest, 2000.

[Har 98] Hartley, S. J.: *Concurrent Programming: The Java Programming Language.* Oxford University Press, Oxford, 1998.

[Har 01] Harper, R.: *Programming in Standard ML.* Working Draft, Carnegie Mellon University, Spring Semester, 2001. `http://www.cs.cmu.edu/~rwh/smlbook/`

[Heg 02] Hegedűs Hajnalka – Horváth Zoltán – Zsók Viktória: *A Haskell és a Clean nyelv nyelv összehasonlító elemzése.* Informatika a Felsőoktatásban, pp. 1075–1084., Debrecen, 2002.

[Hen 86] Henterick, P. V.: *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Massachusetts, 1986.

[Her 79] Herschel, R. – Pieper, F.: *PASCAL: systematische Darstellung von PASCAL und Concurrent PASCAL für den Anwender.* Oldenbourg Verlag, München-Wien, 1979.

[Hoa 72] Hoare, C. A. R.: Proof of Correctness of Data Representations. *Acta Informatica,* vol. 1, pp. 271–281., 1972.

[Hoa 74a] Hoare, C. – Wirth, N.: An Axiomatic Definition of the Programming Language Pascal. *Acta Informatica,* 3, pp. 135–153., 1974.

[Hoa 74b] Hoare, C. A. R.: Monitors: An Operating System Structuring Concept. *Communications of the ACM,* vol. 17, no. 10., pp. 549–557., 1974.

[Hoa 78] Hoare, C. A. R.: Communicating Sequential Processes. *Communications of the ACM,* vol. 21, no. 8, pp. 666–677., 1978.

[Hoa 85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[Hol 83] Holt, R. C. – Cordy, J. R.: *The TURING language report.* Computer Systems Research Institute, University of Toronto, December 1983. Technical Report CSRG-153.

[Hol 84] Holt, R. C.: *TURING: An inside look at the genesis of a programming language.* Computer World, vol. 18, no. 20, May 1984.

[Hor 84] Horowitz, E.: *Fundamentals of Programming Languages.* 2nd ed., Computer Science Press, Rockville, Maryland, 1984. Magyarul: *Magasszintű programnyelvek.* Műszaki Könyvkiadó, Budapest, 1987.

[Hor 87] Horowitz, E. (ed.): *Programming Languages: A Grand Tour.* 3rd ed., Computer Science Press, Rockville, Maryland, 1987.

[Hor 94] Horváth Zoltán: *A párhuzamos programozás alapjai.* Jegyzet kéziratban, ELTE TTK ITCS, Budapest, 1994–95.

[Hor 97] Horváth Zoltán: *A funkcionális programozási stílus és nyelvi elemei.* Jegyzet kéziratban, ELTE TTK ITCS, Budapest, 1997.

[Hor 99a] Horváth Zoltán – Achten, P. – Kozsik Tamás – Plasmeijer, R.: Verification of the Temporal Properties of Dynamic Clean Processes. *Proceedings of the 11th International workshop on the Implementation of Functional Languages,* IFL'99, Lochem, The Netherlands, pp. 203–218., 1999.

[Hor 99b] Horváth Zoltán – Fóthi Ákos: *A funkcionális programozási stílus nyelvi eszközei.* Informatika a Felsőoktatásban, pp. 570–575., Debrecen, 1999.

[Hor 02a] Horváth Zoltán – Kozsik Tamás: *Certified Proven Property Carrying Code (CPPCC) – Safe Functional Mobile Code.* ECOOP Workshops 1 and 18, Malaga, 2002.

[Hor 02b] Horváth Zoltán – Pásztorné Varga Katalin – Tejfel Máté: *Funkcionális programok helyessége.* Informatika a Felsőoktatásban, pp. 942–946., Debrecen, 2002.

[Hor 00] Horváth Zoltán – Zsók Viktória – Serrarens, P. – Plasmeijer, R.: *Parallel Elementwise Processable Functions in Concurrent Clean.* To appear in Computers & Mathematics with Applications, Elsevier.

[Huc 01] Huch, F. – Norbisrath, U.: *Distributed Programming in Haskell with Ports.* Implementation of Functional Programming Languages, 12th International Workshop, IFL2000, Sep. 4–7, Achen, LNCS 2011, pp. 107–121., Springer-Verlag, Berlin, 2001.
`http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell`

[Hud 89] Hudak, P.: *Conception, Evolution and Application of Functional Programming Languages.* ACM Computing Surveys, vol. 21. no. 3. Sept. 1989. pp. 359–411.

[Hud 99] Hudak, P. – Fasel, J. H. – Peterson J.: *A Gentle Introduction to Haskell 98.* `http://www.haskell.org`, 1999.

[Hut 87] Hutchinson, N. C. et al.: *The Emerald Programming Language Report.* Technical Report 87–1007, Department of Computer Science, University of Washington, Seattle, Oct. 1987.

[IPF 03] *Intentional Programming Frequently Asked Questions.* `http://www.omniscium.com/?page=IntentionalFaq`, 2003.

[Jac 75] Jackson, M. A.: *Principles of Programming Design.* Academic Press, London, 1975.

[Jav 03a] *A Sun Microsystems Java honlapja.* `http://java.sun.com/`, 2003.

[Jav 03b] *The Java Language Specification.* Sun Microsystems, 2003. `http://www.javasoft.com/docs/`

[Jen 74] Jensen, K. – Wirth, N.: *Pascal – User Manual and Report.* Springer-Verlag, New York, 1974.

[Jon 96] Jones, S. P. – Gordon, A. – Finne, S.: Concurrent Haskell. *Proceedings of Int. Conf. on Principles in Programming Languages,* pp. 295–308., 1996.

[Jos 99] Josuttis, N. M.: *The C++ Standard Library. A Tutorial and Reference.* Addison-Wesley, Reading, Mass., 1999.

[Kac 91] Kaci, H. A.: *Warren's Abstract Machine,* MIT Press, Cambridge, Massachusetts, 1991.

[Kee 90] O'Keefe, R. A.: *The Craft of Prolog.* MIT Press, Cambridge, Massachusetts, 1990.

[Ker 89] Kernighan, B. W. – Ritchie, D.: *The C Programming Language.* 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1989.
`http://cm.bell-labs.com/cm/cs/cbook/`
Magyarul: *A C programozási nyelv (az ANSI szerint szabványosított változat)* Műszaki Könyvkiadó, Budapest, 1996.

[Ker 99] Kernighan, B. W. – Pike, R.: *The Practice of Programming.* Addison-Wesley, Reading, Mass., 1999. `http://cm.bell-labs.com/cm/cs/tpop/`

[Kes 96] Kesseler, M. H. G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD Thesis, Catholic University of Nijmegen, 1996.

[Kic 97] Kiczales, G. et al.: *Aspect-Oriented Programming.* ECOOP 1997, Jyväskylä, Finland, Proceedings, Springer-Verlag, Berlin, 1997.

[Kic 01] Kiczales, G. et al.: *An Overview of AspectJ.* ECOOP 2001, Budapest, Proceedings, LNCS 2072, pp. 327–353., Springer-Verlag, Berlin, 2001.

[Kis 93] Kiss Anita: *A funkcionális programozás specifikációt támogató nyelvi eszközei.* Szakdolgozat, ELTE TTK Informatika Tanszékcsoport, 1993.

[Knu 81] Knuth, D. E.: *The Art of Computer Programming.* vol. 1–3, Addison-Wesley, Reading, Mass., 1981. Magyarul: *A számítógép-programozás művészete.* Műszaki Könyvkiadó, 1987–88.

[Kor 02] Korpela, J.: *A Tutorial on Character Code Issues.* `http://www.cs.tut.fi/~jkorpela/chars.html`, 2002.

[Kow 79] Kowalski, B.: *Logic for Problem Solving.* North-Holland, New York, Amsterdam, Oxford, 1979.

[Koz 92a] Kozics Sándor: *Az ALGOL 60, a FORTRAN, a COBOL és a PL/I programozási nyelvek.* Egyetemi jegyzet, ELTE TTK, Budapest, 1992.

[Koz 92b] Kozics Sándor: *A Modula-2 programozási nyelv.* Egyetemi jegyzet, ELTE TTK, Budapest, 1992.

[Koz 93] Kozics Sándor: *Az Ada programozási nyelv.* Egyetemi jegyzet, ELTE TTK, Budapest, 1993.

[Köl 97] Kölling, M. – Rosenberg, J.: *Blue – Language Specification.* Version 1.0, School of Computer Science and Software Engineering, Monash University, Technical Report TR97–13, 81 pages, November 1997. `http://www.mip.sdu.dk/~mik/blue/doc/spec-102.pdf`

[Lak 96] Lakos, J.: *Large-Scale C++ Software Design.* Addison-Wesley, Reading, Mass., 1996.

[Lam 77] Lampson, B. W. – Horning, J. J. – London, R. L. – Mitchell, J. G. – Popek, G. J.: Report on the programming language Euclid. *ACM SIGPLAN Notices* vol. 12, no. 2, February 1977.

[Lam 82] Lamprecht, G.: *Einführung in die Programmiersprache SIMULA.* Vieweg, Braunschweig, 1982.

[Lis 73] Liskov, B.:
Guidelines for the Design and Implementation of Reliable Software Systems.
National Technical Information Service, 1973.

[Lis 81] Liskov, B. et al.: *CLU Reference Manual.*
LNCS 114, Springer-Verlag, New York, 1981.

[Lis 86] Liskov, B. – Guttag, J.:
Abstraction and Specification in Program Development.
MIT Press, Cambridge, Massachusetts, 1986.

[Lis 92] Liskov, B.: A History of CLU.
ACM SIGPLAN Notices, vol. 28, no. 3, pp 133–147, 1993.

[Lis 00] Lischner, R.: *Delphi in a Nutshell.* O'Reilly & Associates, Sebastopol, CA, 2000.

[Llo 87] Lloyd, J. W.: *Foundations of Logic Programming.*
2nd ed., Springer-Verlag, Berlin, 1987.

[Loe 87] Loeckx J. – Sieber, K. – Stansifer, R. D.:
The Foundations of Program Verification.
2nd ed., John Wiley and Sons, New York, 1987.

[Loi 01] Loidl, H. W. – Trinder, P. T.: *A Gentle Introduction to GPH.*
Department of Computing and Electrical Engineering, Heriot-Watt University, July 2001.

[Lop 97] Lopez, C. V.: *D: A Language Framework for Distributed Programming.*
PhD Thesis, Graduate School of College of Computer Science, Northeastern University, Boston, MA, 1997.

[Lőcs 77] Lőcs Gyula – Vigassy József: *A FORTRAN programozási nyelv.*
Műszaki Könyvkiadó, Budapest, 1977.

[Luc 69] Lucas, P. – Walk, K.: On the Formal Description of PL/1.
Ann. Rev. Auto. Prog. 3, pp. 105–182., 1969.

[Mac 99] MacLennan, B. J.:
Principles of Programming Languages: Design, Evaluation, and Implementation.
3rd ed., Oxford University Press, New York, 1999.

[Mad 93] Madsen, O. L. – Moller-Pedersen, B. – Nygaard, K.:
Object-Oriented Programming in the Beta Programming Language.
Addison-Wesley, Reading, Mass., 1993.

[Man 91] Manna, Z. – Pnuelli, A.:
The Temporal Logic of Reactive and Concurrent Systems: Specification.
Springer-Verlag, New York, 1991.

[Mar 87] Marcotty, M. – Ledgard, H.: *The World of Programming Languages.*
Springer-Verlag, New York, 1987.

[Mar 93] Marx György: Kemény János. *Fizikai Szemle*, 1993/5. sz., p. 169.

[Mar 98] Marriot, K. – Stuckey P. J.: *Programming with Constraints.* MIT Press, Cambridge, Massachusetts, 1998.

[McC 85] McCarthy, J. et al.: *Lisp 1.5 Programmer's Manual.* 2nd ed., MIT Press, Cambridge, Massachusetts, 1985.

[McN 00] McNamara, B. – Smaragdakis, Y.: Static interfaces in C++. *First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[MDSC 02] *Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces.* `http://www.research.ibm.com/hyperspace/`, 2002.

[Mey 91] Meyer, B.: *Eiffel: The Language.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Mey 00] Meyer, B.: *Object-Oriented Software Construction.* Prentice Hall, New York, 1997. 2nd ed., 2000.

[Mil 97] Milner, R. – Tofte, M. – Harper, R. – MacQueen, D.: *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.

[Mir 90] Turner, D. A.: *Miranda System Manual.* Research Software Ltd. Canterbury, England, 1990.

[Mit 98] Mitchell, J. C.: *Foundations for Programming Languages.* MIT Press, Cambridge, Massachusetts, 1998.

[Mol 99] Mol, de M. – van Eekelen, M.: *A Proof Tool Dedicated to Clean.* AGTIVE'99, Kerkade.

[Mol 01] Mol, de M. – van Eekelen, M. – Plasmeijer, R.: SPARKLE: A Functional Theorem Prover. Arts, Th. – Mohnen M. (eds.): *Proceedings of the 13th International Workshop on the Implementation of Functional Languages*, IFL 2001, Selected Papers, Älvsjö, Sweden, September 24–26, 2001, Springer-Verlag, Berlin, LNCS 2312, pp. 55–71. `ftp://ftp.cs.kun.nl/pub/Clean/papers/2002/molm2002-TheoremProvingFP.ps.gz`

[Mos 94] Moss, C.: *Prolog++, The Power of Object-Oriented and Logic Programming.* Addison-Wesley, Reading, Mass., 1994.

[MPI 94] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard.* University of Tennessee, Knoxville, Tennessee, 1994.

[MPI 95] Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface.* University of Tennessee, Knoxville, Tennessee, 1995, 1996, 1997.

[Muc 97] Muchnick, S. S.: *Advanced compiler design and implementation.* Morgan Kaufmann, San Mateo, CA, 1997.

[Nel 91] Nelson, G.: *Systems Programming with Modula-3.*
Prentice Hall Series in Innovative Technology, Englewood Cliffs, NJ, 1991.

[Nik 01] Nikhil R. S. – Arvind: *Implicit Parallel Programming in pH.*
Morgan Kaufman, San Francisco, California, May 2001.

[Nil 82] Nilsson N. J.: *Principles of Artificial Intelligence.*
Springer-Verlag, Berlin, 1982.

[Nyé 98] Nyékyné Gaizler Judit (ed.) et al.: *Az Ada95 programozási nyelv.*
ELTE Eötvös Kiadó, Budapest, 1998.

[Nyé 01] Nyékyné Gaizler Judit (ed.) et al.: *Java 2 útikalauz programozóknak: 1.3.*
ELTE TTK Hallgatói Alapítvány, Budapest, 2001.

[Nyé 03a] Nyékyné Gaizler Judit (ed.) et al.: *Programozási nyelvek leírásai.*
ELTE TTK ITCS, 2003.
`http://www.inf.elte.hu/targyak/progterv/p-ny4/`

[Nyé 03b] Nyékyné Gaizler Judit (ed.) et al.: *Az Eiffel programozási nyelv.*
ELTE TTK ITCS, 2003.
`http://www.inf.elte.hu/targyak/progterv/eiffel/`

[Oss 99] Ossher, H. – Tarr, P.:
Multi-Dimensional Separation of Concerns and The Hyperspace Approach.
IBM Research Report 21452, April, 1999. IBM T. J. Watson Research Center.
`http://www.research.ibm.com/hyperspace/Papers/tr21452.ps`

[Pas 89] *Turbo Pascal 5.5 – Object-Oriented Programming Guide.*
Borland, 1989.

[Pâr 92] Pârv, B. – Vancea, A.: *Fundamentele limbajelor de programare.*
Egyetemi jegyzet, vol. II., BBTE, Kolozsvár, 1992.

[Pem 82] Pemberton, S – Daniels, M. C.:
Pascal: The Language and Its Implementation – The P4 System.
Ellis Horwood Limited, Chichester, England, 1982.

[Per 79] Perrott, R. H.: A Language for Array and Vector Processors.
ACM TOPLAS, vol. 1, no. 2, pp. 177–195., Oct. 1979.

[Per 83] Perrott, R. H. – Crookes, D. – Milligan P.:
The programming language ACTUS
Software Practice and Experience, vol. 13, no. 4, pp. 305–322., Apr. 1983.

[Pey 99] Peyton Jones, J. – Hughes J. et al.
Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language. February 1999.

[Pet 81] Peterson, G. L.: Myths about the mutual exclusion problems.
Information Processing Letters, vol. 12., pp. 115–116., 1981.

[Pil 99] Pil, M.: Dynamic Types and Type Dependent Functions.
Hammond, K. et al., eds., *Implementation of Functional Languages, 10th International Workshop, IFL'98*, London, UK, September 1998, LNCS, vol. 1595, pp. 169–185., Springer-Verlag, Berlin, 1999.

[Pin 91] Pinson, L. J. – Wiener, R. S.: *Objective C.* Addison-Wesley, Reading, Mass., 1991.

[Pla 99] Plasmeijer, R. et al.: *Functional Programming in Clean.* July 1999. Draft.
`http://www.cs.kun.nl/~clean/`

[Pla 93] Plasmeijer, R. – van Eekelen, M.:
Functional Programming and Parallel Graph Rewriting.
Addison-Wesley, Reading, Mass., 1993.

[Pla 01] Plasmeijer, R. – van Eekelen, M.: *Concurrent Clean Language Report.*
University of Nijmegen, 2001.

[Pos 83] Post, E.: *Real Programmer's Don't Use Pascal.*
A letter to the editor, Datamation, vol. 29, no. 7, July 1983.
`http://www.pbm.com/~lindahl/real.programmers.html`

[Pra 01] Pratt, T. W.: *Programming languages: design and implementation.*
4th ed., Prentice Hall, Upper Saddle River, N. J., 2001.

[PVM 02] OAK Ridge National Laboratory: *PVM Parallel Virtual Machine.*
`http://www.epm.ornl.gov/pvm/pvm_home.html`, 2002.

[Ray 00] Raymond, E. S.: *The Jargon File.* v4.2.0, 2000.
`http://www.jargonfile.com/jargon/`

[Rák 74] Rákosi, M.: *A PL/I programozási nyelv.*
Műszaki Könyvkiadó, Budapest, 1974.

[RFC 1345] Simonsen, K.: *Character Mnemonics & Character Sets.* RFC 1345, 1992.
`http://www.ietf.org/rfc/rfc1345.txt`

[Rob 65] Robinson J. A.: *A Machine-Oriented Logic Based on the Resolution Principle.*
J. ACM 12, pp. 23–41., January 1965.

[Rog 01] Rogers, W. P.: *Reveal the magic behind subtype polymorphism.* 2001.
`http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-polymorph.html`

[Sam 69] Sammet, J.: *Programming Languages: History and fundamentals.*
Prentice Hall, Englewood Cliffs, NJ, 1969.

[Sch 01] Schwartz, R. L. – Phoenix, T.: *Learning Perl.*
3rd ed., O'Reilly & Associates, Sebastopol, CA, 2001.

[Sch 02] Schildt, H.: *C# The Complete Reference.*
McGraw-Hill Osborne Media, New York, 2002.

[Sco 00] Scott, M. K.: *Programming Language Pragmatics.*
Morgan Kaufmann, San Francisco, California, 2000.

[Sea 02] Searle, S. J.:
A Brief History of Character Codes in North America, Europe, and East Asia.
`http://tronweb.super-nova.co.jp/characcodehist.html`, 2002.

[Seb 02] Sebesta, R. W.: *Concepts of Programming Languages.*
5th ed., Addison-Wesley, Reading, Mass., 2002.

[Ser 99] Serrarens, P. R.: Explicit Message Passing for Concurrent Clean.
Hammond, K. et al., eds., *Implementation of Functional Languages, 10th International Workshop, IFL'98*, London, UK, September 1998, LNCS, Vol. 1595, pp. 229–245., Springer-Verlag, Berlin, 1999.

[Set 96] Sethi, R.: *Programming Lanuages: Concepts and Constructs.*
2nd ed., Addison-Wesley, Reading, Mass., 1996.

[Sha 02] Shankar, A. U.: *Object Oriented Exception Handling in Perl*
`http://www.perl.com/pub/a/2002/11/14/exception.html`, 2002.

[Sik 01] Sike Sándor – Varga László: *Objektum elvű modellalkotás UML-ben.*
ELTE TTK, Informatikai Tanszékcsoport, 2001.

[Sim 99] Simonyi, Ch.: *The Future is Intentional.*
IEEE Computer Magazine, May 1999, Volume 32 Number Five, pp. 56–57.

[Skl 97] Sklenar, J.: *Introduction to OOP in SIMULA.*
`http://staff.um.edu.mt/jskl1/talk.html`, 1997.

[Sma 91] *Smalltalk/V Tutorial.* Digitalk, Inc., 1991.

[Smi 00] Smith, G.: *The Object-Z Specification Language.*
Advances in Formal Methods, Kluwer Academic Publishers, 2000.

[SOP 03] *Subject-Oriented Programming.*
`http://www.research.ibm.com/sop/`, 2003.

[Spi 92] Spivey, J. M.: *The Z Notation: A Reference Manual, Second Edition.*
Prentice Hall International (UK) Ltd., 1992.
Programming Research Group, University of Oxford.

[Spl 02] *SICStus Prolog 3.9 User's Manual.*
Swedish Institute of Computer Science, 2002.
`http://www.sics.se/isl/sicstuswww/site/documentation.html`

[Ste 90] Steele, G. L., Jr.: *Common Lisp: The Language.*
2nd ed., Digital Press, Burlington, 1990.

[Ste 94] Sterling L. – Shapiro E.: *The Art of Prolog (Second Edition).*
MIT Press, Cambridge, Massachusetts, 1994.

[Ste 95] Stepanov, A. – Lee, M.: *The Standard Template Library.*
Hewlett-Packard Laboratories, Palo Alto, CA, February 1995.

[Str 00] Stroustrup, B.: *The C++ Programming Language.*
Third Edition and Special Edition, Addison-Wesley, Reading, Mass., 2000.
`http://www.aw.com/catalog/academic/product/1,4096,0201700735,00.html`

Magyarul: *A C++ programozási nyelv.* Kiskapu Kiadó, 2001.
`http://www.zolix.hu/konyv_hu.html`

[Sze 01] Szeredi Péter – Benkő Tamás:
Deklaratív programozás: Bevezetés a logikai programozásba.
Budapesti Műszaki Egyetem, Budapest, 2002.
`http://www.inf.bme.hu/~dp/documents.html`
`http://www.inf.bme.hu/~szeredi`

[Szi 96] Szili László – Tóth János: *Matematika és Mathematica.*
ELTE Eötvös Kiadó, Budapest, 1996.

[Tan 76] Tanenbaum, A. S.: A Tutorial on Algol 68.
ACM Computing Surveys, vol. 8, no. 2, pp. 155–190., June 1976.

[Tan 99] Tandi Péter – Harmat Kristóf – Horváth Zoltán – Wierich M. – Plasmeijer R.: Web Computing in Clean. *Proceedings of 4th International Conference on Applied Informatics*, Eger, Hungary, pp. 87–93., 1999.

[Tho 90] Thomson, S.: Lawful Functions and Program Verification in Miranda.
Science of Computer Programming, Vol. 13, Num. 2–3, pp. 181–218., May 1990.

[Tho 99] Thompson, S.: *Haskell, The Craft of Functional Programming.*
Addison-Wesley, Reading, Mass., 1999.

[Til 96] Till, D.: *Teach Yourself in Perl 5 in 21 Days.* SAMS Publishing, Indianapolis, IN, 1996.

[Tri 98] Trinder, P. W. – Hammond, K. – Loidl, H. W. – Peyton Jones, S. J.:
Algorithm + Strategy = Parallelism.
Journal of Functional Programming vol. 8 no. 1: pp. 23–60., 1998.

[Tur 86] Turner, D.: An Overview of Miranda.
ACM SIGPLAN Notices, vol. 21, no. 12, pp. 158–166., 1986.

[Tur 03] Turing Programming Language Home Page.
Holt Software Associates Inc, 2003.
`http://www.holtsoft.com/turing/`

[UCS 02] *UCS (ISO 10646, Unicode) character blocks.*
`http://www.cs.tut.fi/~jkorpela/ucs.htm8`, 2002.

[Uni 02] The Unicode Consortium: *The Unicode Standard.*
Version 2.0: Addison-Wesley, Reading, Mass., 1996.
Version 3.0: Addison-Wesley, Reading, Mass., 2000.
`http://www.unicode.org/unicode/standard/standard.html`

[Wad 85] Wadge, W. W. – Ashcroft, E. A.: *Lucid, the Dataflow Programming Language.*
Academic Press, London, 1985.

[Weg 87] Wegner, P.: Dimensions of Object-Based Language Design.
ACM SIGPLAN Notices, vol. 22, no. 12, pp. 168–182., Dec. 1987.

[Weg 89] Wegner, P.: *Concept and Paradigms of Object-Oriented Programming.* Expansion of Oct. 4 OOPSLA-89 Keynote Talk, 1989.

[Wel 79] Welsh, J. et al.: Pascal Plus – Another Language for Modular Multiprogramming. *Software Practice and Experience*, vol. 9, pp. 947–957., 1979.

[Wex 81] Wexelblat, R.: *History of Programming Languages.* Academic Press, New York, 1981.

[Whe 96] Wheeler, D. A.: *Ada 95 The Lovelace Tutorial.* Addison-Wesley, Reading, Mass., 1996.

[Wir 73] Wirth, N.: *Systematic Programming.* Prentice Hall, Englewood Cliffs, NJ, 1973.

[Wir 82] Wirth, N.: *Programming in Modula-2.* Springer-Verlag, Berlin, 1982. 2nd ed., 1983.

[Wir 88] Wirth, N.: The Programming Language Oberon. *Software Practice and Experience*, vol. 18, no. 7, pp. 671–690., 1988.

[Wul 74] Wulf, W. A.: *Alphard: Toward a Language to Support Structured Programs.* Technical Report, Carnegie Mellon University, Pittsburgh, PA, April, 1974.

[Zus 72] Zuse, K.: *Der Plankalkül.* Gesellschaft für Mathematik und Datenverarbeitung, Bonn, 1972.

[Zsó 02] Zsók Viktória – Horváth Zoltán – Tejfel Máté: *Párhuzamos funkcionális programozás.* Informatika a Felsőoktatásban, pp. 1085–1094., Debrecen, 2002.

Tárgymutató

.NET, 67, 227
λ-kalkulus, 588, 589, 591, 592, 596, 599
λ-kifejezés, 237, 593

A, Á

ábrázolás, 121, 123, 125–127, 129, 134, 174, 186, 194, 195, 199, 333–335, 338–340, 342, 347
ábrázolási záradék, 175
ábrázoló függvény, 121, 476
absztrakt, 580
  adattípus, 332–350, 356, 393, 405, 477
  alaposztály, 364
  hely, 632, 633
  megvalósítás, 360
  művelet, 616, 620
  típus, *lásd* típus
Actus nyelv, 506, 507
Ada nyelv, 4, 9, 10, 12, 34, 36, 37, 224, 618, 624
  Ada 83, 12, 149, 251, 699
  Ada 95, 12, 40, 150, 214, 251, 289, 323, 379, 399, 404, 406, 422, 697
    ábrázolás, 174
    direktszorzat típus, 170, 174
    mutató, 154
    rekord típus, 170
  aktivációs rekord, 266
  alprogram fejléce, 216
  alprogram specifikációja, 221
  alprogram törzse, 222
alprogrammal való paraméterezés, 214
alprogramra hivatkozó mutató, 214, 215, 236
altípus, 161, 218, 251
attribútum, 128, 191
blokk utasítás, 73
ciklus, 91
csomag, 323
diszkriminánsos rekord, 185
egész típus, 134, 136
eljárás, 206
eljárás és függvény, 208
előrevetett deklaráció, 233
formális paraméter típusa, 210
főprogram, 221
függvény, 206
függvény visszatérési értéke, 215
függvények mellékhatása, 208
gyerekcsomag, 323
in módú paraméter, 253
inline alprogram, 235
IRONMAN, 9
karakter típus, 138
kifejezés kiértékelési sorrendje, 208
kivételkezelés, 222, 223, 289, 306
konstruktor, 404
külön fordítás, 254, 255
logikai típus, 139
más nyelven írt törzs, 226
modul, 233, 323
mutató, 147, 148

operátorok túlterhelése, 264
optimalizálás, 236
osztály, 399
öröklődés, 422
paraméter-tömbök mérete, 210
paraméterátadás, 248
paraméterek, 221, 229, 250
parametrikus polimorfizmus, 361
privát nézet, 338
programegységek egymásba ágyazása, 256
rekurzív alprogram, 232
return utasítás, 224
sablon, 263, 353, 356, 362, 379
sablon-szerződés modell, 375
skalár típus, 130
STEELMAN, 9
STRAWMAN, 9
származtatott típus, 161
szemétgyűjtés, 112
szintaktikus polimorfizmus, 372
taszk, 511, 514–532
típusattribútum, 128, 191
típusekvivalencia, 168
típusellenőrzés, 217
típuskonverzió, 124, 127
típusparaméter, 376
többdimenziós tömb, 211
tömb típus, 190, 194
túlterhelés, 108, 262, 369
üres paraméterlista, 227
valós típus, 142
védett egység, 208
védett objektum, 530
végrehajtási szál, 272
vektor típus, 188, 190
adaptív programozás, 692
adatabsztrakció, 205, 271, 333, 356, 609, 614, 615, 618
adatfolyam, 623
adatkonstruktor, *lásd* típus
adatrejtés, 406, 423
C++ nyelv, 407
Eiffel nyelv, 409
Java nyelv, 409
Object Pascal nyelv, 408
Smalltalk nyelv, 407
adatszegmens, 254
adatvezérlésű nyelv, 505
advice, *lásd* tanács
ágens, 625, 632, 695
aggregátum
direktszorzat, 172, 199
tömb, 191
Aksit, M., 693
aktivációs rekord, 105, 265
aktuális paraméter, 209
alacsonyszintű programozási nyelv, 50
ALGOL nyelv, 10–12
ALGOL 58, 8
ALGOL 60, 8, 12, 37, 109, 208, 214, 215, 217, 224, 243, 246, 248, 255, 258, 263, 268, 348, 589, 700
ALGOL 68, 12, 37, 155, 156, 160, 163, 180, 183, 188, 189, 199, 214, 215, 217, 225, 236, 237, 248, 272, 509, 697, 700
ALGOL W, 12, 241, 242, 248
alprogram fejléce, 216
alprogram típus, 236
alprogram típusú literál, 237
alprogrammal való paraméterezés, 214
blokkszerkezet, 255
címke paraméter, 215
eljárás, 206
eljárás és függvény, 208
előre definiált túlterhelt alprogram, 263
függvény, 206
függvény visszatérési értéke, 215, 225
paraméterátadás, 248
típusellenőrzés, 217
végrehajtási szál, 272
visszatérés alprogramból, 224
algoritmus, 46
alias, 239
állapot-átmenetek, 61
állapottér, 61
allokátor, 144, 149
Alphard nyelv, 270, 697
alprogram, 203–275, 285, 344, 349
aktív, 209, 264
belépési pontja, 229
definíciója, 209
deklarációja, 233, 254
fejléce, 216

hívása, 45, 209, 227
hívási lánc, 265, 278
inline, 235
interfésze, 216
lokális változója, 265
megvalósítás, 264
mutató, 151
neve, 206
paraméter, *lásd* paraméter
paraméterként, 214, 217, 246, 267
paraméterlista, 227
példány, 264
profil, 221
protokoll, 221
prototípus, 217, 234
rekurzív, 232, 265
közvetetten, 233, 234
végrekurzív, 266
specifikációja, 209, 216, 233, 262, 278
szignatúra, 262
típus, 236
törzse, 209, 222
más nyelven írva, 226
túlterhelése, 261
változó hosszú paraméterlista, 218
végrehajtás, 293
visszatérési értéke, 215, 224, 282, 283
visszatérési értékként, 215, 268
altípus, 161, 185, 191, 251, 262
Analytical Engine, 203
annotáció (párhuzamos kiértékeléshez), 626
ANSI, 10, 31
ANSI C nyelv, *lásd* C nyelv
AOP, *lásd* aspektusorientált programozás
APL nyelv, 10–13, 210, 257
ArgoUML eszköz, 563
argumentum, 591
aritás, 155, 343
aritmetikai elágazás, *lásd* elágazás
ARPANET, 12
ASA FORTRAN nyelv, *lásd* FORTRAN nyelv
ASCII kód, 30, 31, 130, 713
AspectJ nyelv, 686, 689
aspektus, 686, 689
aspektusorientált programozás, 686
assembler, 320
Assembly nyelv, 10, 50, 56, 243
aszinkron kommunikáció, 515
asszociatív, 597, 599, 610
balról, 610
jobbról, 264
AT&T Bell Labs, 701, 702
átdefiniálás
kontravariáns, 429
kovariáns, 429
novariáns, 429
átírás, 591–593
átírási lépés, 591–593
átíró rendszer, 591
gráfátíró rendszer, 591, 596, 703
kifejezésátíró rendszer, 591
konfluens, 591, 592, 597
átlapolás, *lásd* túlterhelés
átlátszatlan struktúra, *lásd* struktúra
attribútum, 128, 143, 191, 278
automatikus változó, *lásd* változó
AWK nyelv, 10, 116
azonosító, 35
azonosítótér, 328

B

Babbage, Ch., 7, 203
Backus, J., 8, 700, 705
balérték, 171, 189, 192
BASIC nyelv, 9, 10, 12, 36, 170, 188, 344, 345, 347–349, 701
paraméter, 209, 210
Tiny BASIC, 9
Visual BASIC, 9, 37, 702
beágyazott osztály, 261
befejezett taszk (komplett), 289, 518
bejáró, 93, 95, 269, 694
bejáró ciklus, *lásd* ciklus
belépési pont, 229, 511, 515, 520, 521, 531
belső dokumentáció, 568
bemenő paraméter, *lásd* paraméter
Bemer, B., 31
BETA nyelv, 178, 215, 511, 701
bevezetés, 689
bináris, 155
bináris szemafor, 511, 530
bitművelet, 134, 139, 195
biztonságos elágazás, 82
BLISS nyelv, 235, 272

blokk, 144, 150, 285, 343
  catch, 282
  finally, 278, 282, 285
  try, 282
  utasítás, 71–75
blokkdiagram, 49
blokkszerkezet, 66, 150, 255, 266, 268, 343, 595
Blue nyelv, 224
Bootstrap módszer, 564
Borland, 704
break utasítás, 78
bug
  EPOCH 0x7fffffff, 113
  Y2K, 113
Burstall, R. M., 596, 710
busy waiting, *lásd* tevékeny várakozás
Byron, Ada, 7, 699

C

C nyelv, 4, 9, 10, 12, 14, 41, 66, 589, 701, 711
  ... jelölés, 219
  alprogram
    fejléce, 216
    rá hivatkozó mutató, 214, 215, 236, 268
    törzse, 222
  altípus, 218
  ANSI C, 201, 217, 249, 255
  aspektusok, 691
  blokk utasítás, 74, 255
  ciklus, 92
  const, 249
  egész típus, 137
  elágazás, 84, 85
  eljárás és függvény, 206
  előfordító, 235
  értékadás, 69
  fejléc fájl, 109, 255, 340
  felsorolási típus, 132
  fordítási egység, 254
  főprogram, 220
  free függvény, 104
  független fordítás, 254
  függvény, 206
  függvény visszatérési értéke, 215, 216, 254
  goto, 77
  karakter típus, 138
  kifejezés kiértékelési sorrendje, 208
  kivételkezelés, 279, 283
  logikai típus, 139
  lokális változó élettartama, 260
  makró, 260
  malloc függvény, 104
  mutató, 146–148, 150, 154
  mutatóaritmetika, 152
  operátorok precedenciája, 160
  paraméterátadás, 248
  printf, 207, 219
  prototípus, 217, 255
  return utasítás, 224
  static módosító, 109
  statikus változó, 260
  típusekvivalencia, 168
  típusellenőrzés, 108, 217
  típuskonverzió, 124, 125
  többdimenziós tömb, 212
  tömbök mérete, 211
  tömböket tartalmazó tömb, 213
  túlterhelés, 108
  unió típus, 180
  üres paraméterlista, 227
  végrekurzió optimalizációja, 267
  vektor típus, 188, 190
  vessző operátor, 207
C* nyelv, 506
C++ nyelv, 4, 9, 10, 12, 34, 36, 41, 692, 693, 702
  absztrakt osztály, 438
  adatrejtés, 407
  alprogram törzse, 222
  alprogramra hivatkozó mutató, 214, 215, 236, 268
  altípusos polimorfizmus, 365
  aspektusok, 691
  auto_ptr sablon, 115
  automatikus változók számontartása, 112
  const, 249
  delete operátor, 104, 114
  destruktor, 111, 114, 224, 402
    azonnali meghívása, 111
    neve, 111
  egész típus, 137

eljárás és függvény, 206
fejléc fájl, 109, 340
felsorolási típus, 132
főprogram, 220
friend, 110
  metódus, 409, 410
  osztály, 409
függvény, 206
gyenge típusosság, 383
halmaz típus, 195
hatókör-operátor, 108
inline alprogram, 235
invariáns altípusosság, 386
iterátor, 271
karakter típus, 138
kivételkezelés, 222, 223, 291, 302
konstruktor, 114, 402
  rá mutató pointer, 111
konstruktor-destruktor-mechanizmus, 114
lusta példányosítás, 382
metódus, 234, 413
mutató, 147, 148, 150, 154
mutatóaritmetika, 152
névtelen névtér, 109
new operátor, 104
operátorok túlterhelése, 264
osztály, 395
osztályadat, 413
osztálymetódus, 413
öröklési mód, 424
öröklődés, 419
összekapcsolás, 445
paraméter alapértelmezett értéke, 228
példányosítás, 382
polimorfizmus, 430
prototípus, 217, 234
referencia típus, 153, 248
sablon, 263, 355
  osztály, 358
  specializáció, 384
standard könyvtár, 115
static módosító, 109
statikus változó, 260
STL, 195
szintaktikus polimorfizmus, 372
tagok láthatósága, 222
  template, *lásd* sablon
  típuskényszerítés, 370
  típuskonverzió, 124, 125
  többdimenziós tömb, 212
  többszörös öröklődés, 442, 447, 448
  túlterhelés, 108, 262, 369
  unió típus, 182
  változók kezdőértéke, 112
  vektor típus, 190
C-sharp, *lásd* C# nyelv
C# nyelv, 34, 36, 38, 66, 206, 220, 222, 252, 253, 271, 702
  absztrakt osztály, 439
  aspektusok, 691
  ciklus, 93
  elágazás, 85
  függvény, 206
  interfész, 453
  kivételkezelés, 297
  metódus, 413
  osztály, 398
  osztályadat, 413
  osztálymetódus, 413
  öröklődés, 421
  polimorfizmus, 433
  szemétgyűjtés, 112
call chain, *lásd* hívási lánc
Caml Light nyelv, 596, 710
Caml nyelv, 596, 710
cast, *lásd* típuskonverzió
Cayenne nyelv, 219
CC++ nyelv, 514
CFD FORTRAN, *lásd* FORTRAN nyelv
CFT FORTRAN, *lásd* FORTRAN nyelv
Church, A., 589
ciklus, 87–98, 267, 269
  bejáró, 87, 93, 95
  biztonságos, 90
  ciklusinvariáns, 479, 488
  ciklusmag, 88
  ciklusváltozó, 87, 91
    értéktartománya, 92
    hatóköre, 93
  ciklusvariáns, 481, 488
  számlálásos, 87
cím, 146
cím szerinti paraméterátadás, 153, 239, 247

címaritmetika, 604
címfüggvény, 194
címke, 75, 215, 285
címzési módok, 55
CISC processzorok, 51
Clean nyelv, 209, 254, 256, 263, 264, 507, 589–594, 596, 600, 603–605, 610, 611, 614, 615, 620, 622, 624–626, 629, 703
  Concurrent Clean, 513, 596, 703
  Object I/O, 623
  polimorfizmus, 356, 361
CLOS nyelv, 694, 707
CLU nyelv, 36, 206, 215, 216, 252, 253, 260, 269, 272, 697, 703
  absztrakt adattípus, 338
  direktszorzat típus, 177
  egész típus, 135
  értékadás, 70
  felsorolási típus, 132
  függvény, 206
  immutable típus, 169
  kivételkezelés, 288
  logikai típus, 139
  mutable típus, 169
  mutató, 147
  skalár típus, 129
  típuskonverzió, 126
  unió típus, 183
  vektor típus, 192
CMM eszköz, 564
COBOL nyelv, 10, 12, 13, 37, 63, 214, 248, 703
  ciklus, 87
  COBOL 65, 12
  COBOL 74, 12
  dinamikus memória használata, 104
  elágazás, 80
  értékadás, 68
  goto, 76
  kivételkezelés, 286
coercion, *lásd* típuskényszerítés
coercion polymorphism, *lásd* típuskényszerítéses polimorfizmus
Colmerauer, A., 637
COmmon Business Oriented Language, *lásd* COBOL nyelv
Common Language Runtime, 227
Common LISP nyelv, *lásd* LISP nyelv
Concurrent Clean, *lásd* Clean nyelv
Concurrent Eiffel nyelv, *lásd* Eiffel nyelv
Concurrent ML nyelv, *lásd* ML nyelv
Concurrent Pascal, *lásd* Pascal nyelv
Concurrent Prolog nyelv, *lásd* Prolog nyelv
continue utasítás, 78
control driven language, *lásd* ellenőrzés vezérelt nyelv
CORBA, 693
CSP nyelv, 507, 532
  CSP/K, 512
  CSP/P, 512
Curry módszer, 230, 594, 600
Curry, Haskell B., 593, 594, 596, 599, 705
Cyber FORTRAN nyelv, *lásd* FORTRAN nyelv

CS

csatorna, 511, 512, 625, 626, 628, 632
  aszinkron, 632
  halmaz, 633
  szinkron, 632
csellengő ELSE, 84
csomag, 323, 328, 342

D

D-diagramok, 48
DAF FORTRAN nyelv, *lásd* FORTRAN nyelv
Dahl, O. J., 710
Data Definition Language, 711
data driven language, *lásd* adatvezérlésű nyelv
Data Manipulation Language, 711
Dataflow nyelv, 507
Dataparallel-C nyelv, 506
DDL, *lásd* Data Definition Language
deadlock, *lásd* holtpont
deallokátor, 145, 149
DEC, *lásd* Digital Equipment Corporation
definíció, 209
definíciós modul, 234
deklaráció, 222, 233, 254, 595
  alprogramé, 233, 254
  deklarációs rész, 74, 109, 289
  előrevetett, 148, 233
  függvényé, 591, 597

hatókör, *lásd* hatókör
lokális, 595, 597, 605, 622
típusé, 594
deklaratív programozási nyelv, 61, 588
delegáció, 693
Delphi eszköz, *lásd* Object Pascal nyelv
Delphi rendszer, 704
demand driven language, *lásd* igényvezérlésű nyelv
Demeter törvénye, 692
Demeter/C++ nyelv, 693
DemeterJ nyelv, 693
dereferencia, 150, 151
destruktor, 111, 112, 224, 292, 342, 400
Digital Equipment Corporation, 707
Digital Research, 10
dinamikus
hatókör, 257, 268
kapcsolódási pont, 689
kiértékelés, 172
kivételkezelés, 287
kötés, 365, 426
lánc, 266
öröklődés, 693
összekapcsolás, 144, 231, 343, 345, 416
predikátum, 670
tárterület, 104, 112, 144, 149, 150, 337, 341
típus, 126, 179, 625, 629
típuskényszerítés, 371
változó, *lásd* változó
direktszorzat, 170, 171, 198, 333
Distributed Eiffel nyelv, *lásd* Eiffel nyelv
Distributed Haskell nyelv, *lásd* Haskell nyelv
diszkrét típus, *lásd* típus
diszkriminánsz, 185, 344, 518, 531
DJ nyelv, 693
DML, *lásd* Data Manipulation Language
dokumentáció, 568

E, É

eager instantiation, *lásd* mohó példányosítás
EBCDIC kód, 30, 31, 716
Eclipse projekt, 690
Eclipse Universal Modeler eszköz, 695
Eden nyelv, 597
egész típus, *lásd* típus
egyenlőségi érvelés, 594
egyenlőségvizsgálat, 151, 173, 176, 196, 198
egyértelműség, 208
egymásba ágyazás, 253, 255, 290
egységbe zárás, 234, 334, 335, 379, 405
egységelem (ML, Haskell), *lásd* rendezett nullás
egyszeresen hivatkozott
belső állapottér, 621
környezet, 596
típusérték, 622
tömb, 604
változó, *lásd* változó
Eiffel nyelv, 10, 36, 37, 178, 189, 200, 224, 225, 253, 704
absztrakt osztály, 439
altípusos polimorfizmus, 366
átnevezés, 444
ciklushelyesség, 487, 488
ciklusinvariáns, 479–481
ciklusvariáns, 481
Concurrent Eiffel, 509
Distributed Eiffel, 509
egész típus, 137
Eiffel//, 509
előfeltétel, 473
felsorolási típus, 132
gyűjtemények, 367
helyesség-ellenőrzés, 460–498
kivételkezelés, 292, 308
konstruktor, 404
logikai típus, 139
metódushelyesség, 487–489
mutató, 147
NONE osztály, 372
osztály, 396
osztályhelyesség, 490
osztályinvariáns, 477, 484, 485
öröklési mód, 425
öröklődés, 386, 420
ismételt, 436
összekapcsolás, 444
polimorfizmus, 386, 433
sablon paramétere, 379
sablon-szerződés modell, 374
select, 449
skalár típus, 129
szelektív láthatóság, 409

típuskonverzió, 126
típusparaméter, 378
többszörös öröklődés, 443, 448, 449
undefine, 446
utófeltétel, 473
Eiffel// nyelv, *lásd* Eiffel nyelv
elágazás, 79–86
aritmetikai, 79
kétirányú, 79
többirányú, 79, 82, 85
elemi típus, *lásd* típus
elemtípus, 188
elérhetetlen objektum, 111
élettartam, 103, 110–116, 144, 260, 265
eljárás, 45, 205, 349, 521
ellenőrzés vezérelt nyelv, 505
ellenőrzött kivétel, 222
előfeltétel, 471, 475
előfordító, 235, 340
előrevetett deklaráció, *lásd* deklaráció
elsőrendű logika, 464
Emacs editor, 645, 690, 695
Emerald nyelv, 507
encapsulation, *lásd* egységbe zárás
encoding, *lásd* kódolás
Enterprise JavaBeans, 688, 693
eredmény szerinti paraméterátadás, 240
Erlang nyelv, 590
erőforrás, 278
kezelése objektumban, 115
külső, 111
erős típusosság, *lásd* típus
érték szerinti paraméterátadás, 237
érték típus, 252
érték/eredmény szerinti paraméterátadás, 241, 248
értékadás, 68–70, 148, 171, 176, 189, 191, 196–198, 200, 344, 591
megsemmisítő, 593, 619, 620
értékadó operátor, 69
értékmegosztás, 372
értelmező, 298
esemény, 62
eseményleíró kifejezés, 689
eseményvezérelt programozás, 62
esetszétválasztás, 597
étkező filozófusok, 533
Euclid nyelv, 225, 235, 240
euklideszi algoritmus, 46, 50
exception, *lásd* kivétel
explicit példányosítás, *lásd* példányosítás
Extended Pascal nyelv, 36

F

fejállomány, *lásd* fejléc fájl
fejléc fájl, 255, 320, 340, 342
felsorolási típus, *lásd* típus
feltétel, 597, 598
elő-, 293
utó-, 293
feltétel nélküli vezérlésátadás, *lásd* goto
feltételes
belépési pont hívás, 526
feltételes fordítás, 79
feltételes hívásfogadás, 523
feltételes kritikus szakasz, 505
felüldefiniálás, 343
fenntartott szó, 37
Feys, R., 593, 599
Fibonacci-sorozat, 105
finally blokk, *lásd* blokk
fixpontos típus, *lásd* típus
folyamat, 501, 512, 632
interaktív, 624
konkurens, 632
példány, 632
vándorlás, 633
adatvezérelt, 634
folyamat (processz), 508
folyamatábra, 47
fordítási egység, 29, 108, 254
fordítóprogram, 51
fordítóvezérlő direktíva, 227, 236
formális paraméter, 209, 591
Forte 4J eszköz, 690
Forth nyelv, 10
FORTRAN nyelv, 8, 10, 12, 13, 35, 37, 38, 162, 163, 170, 198, 200, 333, 347, 589, 705
alprogram fejléce, 216
ASA FORTRAN, 201, 210, 214, 254, 255, 260
belépési pont, 229
blokkszerkezetesség hiánya, 255

CALL utasítás, 227
CFD, 506
CFD FORTRAN, 506
CFT, 506
COMMON direktíva, 108
Cyber FORTRAN, 506
DAF FORTRAN, 506
előfordító, 235
előre definiált túlterhelt alprogram, 263
fordítási egység, 254
FORTAN 0, 8
FORTRAN 66, 12
FORTRAN 77, 12, 194, 212, 214, 217, 229, 247
FORTRAN 90, 12, 194, 210, 217, 233, 255, 264–266
FORTRAN II, 254
FORTRAN IV, 248, 506
független fordítás, 254
függvény visszatérési értéke, 215
kivételkezelés, 285
külön fordítás, 255
modul, 321
operátorok definiálása, 264
paraméter, 210
paraméter-tömbök mérete, 210
paraméterátadás, 247
rekurzív alprogram, 233, 266
statikus változó, 260
típusellenőrzés, 217
többdimenziós tömb, 212
tömb típus, 194
visszatérés alprogramból, 224
VS FORTRAN, 506
forward declaration, *lásd* előrevetett deklaráció
főprogram, 220, 289, 291, 294, 298
főszegmens, 254
funkcionális
absztrakció, 594
I/O modell
adatáramlási, 596, 623
folytatás, 596
I/O művelet, 619, 621, 622
nyelv, 61, 507, 513
program, 588, 591
programozás, 513, 588–634
programozási stílus, 589–634
tisztán, 593, 596, 619, 620, 703
funktor, 610, 615, 618, 620
futásidejű programhiba, 277
független fordítás, 254
függő típus, 219
függvény, 45, 205, 349
alkalmazás, 594, 597, 600
részleges, 594, 600
argumentum, 591–595, 597, 599, 600, 602
definíció, 591–593, 595, 597
deklaráció, 591, 597
elsőrendű, 599
érték, 595, 597, 599
értékkészlet, 597, 599
értelmezési tartomány, 597, 599
kompozíció, 592
lokális, 605
magasabbrendű, 591, 594, 599
objektum, 152, 345
osztály, *lásd* típusosztály
paraméter, 592, 602
aktuális, 595, 597, 598
formális, 591, 593, 595, 597, 606
parciális, 471, 597, 624
rekurzív, 594, 597
reprezentációs, 476
származtatott, 610
szignatúrája, 107
terminális, 481
típusa, 599, 610
totális, 471
többváltozós, 594, 599
törzs, 591, 592, 595, 597, 605

G

garbage collector, *lásd* szemétgyűjtő
generatív programozás, 693
generátor, 270, 602, 608
egymásba ágyazott, 603
párhuzamos, 603, 620
tömb, 605
generic, 353, 354, 356–358, 361, 362, 380, 381
Generic Java (GJ) nyelv, *lásd* Java nyelv
generikus programozás, 694
GJ nyelv, *lásd* Java nyelv

Glasgow Parallel Haskell nyelv, *lásd* Haskell nyelv
globális, 144, 266
Goldberg, A., 710
Gosling, J., 10, 695
goto
  hiánya, 78
  kiszámított, 76
  problémája, 75
  utasítás, 287, 301
goto utasítás, 279
GPH nyelv, *lásd* Haskell nyelv
gráfátíró rendszer, *lásd* átíró rendszer
Green projekt, *lásd* Java nyelv

GY

gyenge kapcsolat, 317
gyermekcsomag, 324

H

halmaz, 187, 195, 201
halmazkifejezés, *lásd* Zermelo-Fraenkel halmazkifejezés
Harper, R., 596, 710
Harrison, W., 692
hashtable, *lásd* hasítótábla
hasítótábla, 187, 196, 201
Haskell nyelv, 215, 227, 589–593, 595, 596, 599, 600, 604, 605, 610, 611, 613–615, 620–625, 705
  Distributed Haskell, 513, 597, 625, 630
  eljárás, 206
  függvény, 206
  függvények mellékhatása, 208
  Glasgow Parallel Haskell (GPH), 507, 597
határolójel, 34
hatékonyság, 564
hatókör, 103, 106–110, 253, 594, 597, 605, 613, 621
  dinamikus, 257, 268
  fordítási egység, 108
  függvény és kódblokk, 109
  globális, 108
  korlátozott, 614
  lokális definíció, 606, 620
  lokális deklaráció, 595, 597, 605, 622
  lokális konstans, 623
  margó szabály, 595, 597, 605, 621
  statikus, 257, 268
  típus, 110
  where, 594, 597, 605
header, *lásd* fejléc fájl
heap, *lásd* dinamikus tárterület
helyesség, 563, 594
helyességbizonyítás, 75, 294
helyességi alspecifikáció, 484
helyességi specifikációs nyelv, 464
hibakezelés, 279
Hindley–Milner típusrendszer, *lásd* típus
hívási lánc, 265, 278
hivatkozás típus, 252, 253
hivatkozás-feloldás, 150, 153, 253, 621
hivatkozási helyfüggetlenség, 593, 602, 604, 619
hivatkozási szint, 143, 150
hivatkozásszámláló, 145, 149, 342
Hoare-formula, 472
holtpont, 503, 520
Hope nyelv, 593, 595, 596, 710
hordozható assembly, *lásd* C nyelv
hordozhatóság, 208, 242, 568
Horowitz, E., 18
HTML, 12
Hudak, P., 596, 705
Hughes, J., 596, 705
HyperJ nyelv, 692

I, Í

IBM, 8, 9, 13, 705
IBM/360 gépcsalád, 9
Ichbiah, Jean, 699
IEEE 754 szabvány, 140
igény szerinti példányosítás, *lásd* példányosítás
igényvezérlésű nyelv, 506
immutable, *lásd* típus
imperatív nyelv, 61
implementációs modul, 234
implikáció, 139
import, *lásd* modul
inclusion polymorphism, *lásd* altípusos polimorfizmus
indexfüggvény, 212
indextartomány, 188, 190–193, 200

indukció, 232
infix, *lásd* művelet
információ-elrejtés, 103, 317
Ingalls, D. H., 710
inline alprogram, 235
instantiation, *lásd* példányosítás
Intel 8080, 9
intencionális programozás, 694
interaktív funkcionális program, 622
interfész, 317, 406, 450
Interlisp nyelv, 272
Internet, 10, 12
intervallum típus, 251
invariáns
  ciklus-, 293
  osztály-, 294
  specifikációs, 120
IRONMAN nyelv, *lásd* Ada nyelv
Irvine Dataflow nyelv, 505
ISO 10646 szabvány, 32, 130, 138
ISO 8859 szabvány, 30, 32
ISO 8859-1 kód, 32, 714
ISO 8859-2 kód, 32, 714
ISO 9001 szabvány, 564
ISWIM nyelv, 596
iterált, 187, 201
iterátor, *lásd* bejáró

J

J# nyelv, 66
Jackpot projekt, 695
Jacobson, I., 695
Java nyelv, 4, 10–12, 34, 36, 37, 41, 511, 692, 693, 695, 706
  absztrakt osztály, 438
  adatrejtés, 409
  alprogram törzse, 222
  aspektusok, 689
  beágyazott osztály, 261
  belső osztály, 110
  ciklus, 92
  csomag, 328
  egész típus, 137
  eljárás és függvény, 206
  felsorolási típus, 133
  finalize, 111
  finally, 115
  fordítási egység, 254
  főprogram, 220
  függvény, 206
  Generic Java (GJ), 362, 378, 382, 385
  Green projekt, 10
  immutable típus, 169
  interfész, 215, 451
  iterátor, 271
  JDBC API, 111
  karakter típus, 138
  kifejezés kiértékelési sorrendje, 208
  kivételkezelés, 222–224, 295, 304
  konstruktor, 403
  más nyelven írt törzs, 227
  metódus, 413
  metódus meghívása, 231
  modul, 328
  monitor, 540
  mutable típus, 169
  mutató, 147
  new operátor, 104
  operátorok, 264
  osztály, 397
  osztályadat, 413
  osztálymetódus, 413
  önelemzés, 694
  öröklődés, 421
  paraméterátadás, 251
  párhuzamosság, 537
  polimorfizmus, 432
  programegységek egymásba ágyazása, 256
  skalár típus, 129
  statikus adattag kezdőértéke, 112
  szálak, 537
  szemétgyűjtés, 112
  szemétgyűjtő algoritmus, 111
  tagok láthatósága, 222
  tagosztály, 454
  típusellenőrzés, 217
  típuskonverzió, 124, 126
  többdimenziós tömb, 211
  tömbök mérete, 211
  túlterhelés, 108, 262
  végrehajtási szál, 272
  vektor típus, 190
  visszatérés függvényből, 226
Java Virtual Machine, 227, 304, 689

JavaScript nyelv, 10, 12, 116, 218, 224, 228
JBuilder eszköz, 690
jel, 508
Jensen's device, 244
jgnat, 227
jobbérték, 192
JoCaml nyelv, 507, 513, 597, 625, 632
join, *lásd* összekapcsolás
join point, *lásd* kapcsolódási pont
JOVIAL nyelv, 10, 12
JVM, *lásd* Java Virtual Machine

K

kapcsolat, 317
kapcsolódási pont, 688
kapu, 630
karakter
  halmaz, 30
  készlet, 29
  kód, 29, 30
  kódolás, 29, 30
karakterisztika, 140
karakterlánc-konverzió, 124, 131
karbantarthatóság, 205, 316, 565
Kay, A., 710
Kemény János, 701
késői összekapcsolás, 231, 245, 253, 365
kétirányú elágazás, *lásd* elágazás
kezdeti kifejezés, *lásd* kifejezés
kicsomagolás, 174, 199
Kiczales, G., 694
kiértékelés, 591, 593, 594, 597
  dinamikus, 172
  elosztott, 625
  kifejezésfa, 157
  lusta, 592–596, 609, 626, 703
  módszer, 591–593, 597
    normalizáló, 593
    összetett, 626
  mohó, 593
  párhuzamos, 597, 625, 626
    annotáció, 626
  sorrend, 208, 264, 592, 594
  spekulatív, 626
kiértékelő jel, 592
kifejezés, 155, 159, 206, 288, 593, 597
  case, 613
  eseményleíró, 689
  kezdeti, 588, 591, 592
  kiértékelés, 157
  szelektor, 79
  szerkezet, 155
  utasítás, 69
kifejezésátíró rendszer, *lásd* átíró rendszer
kifejezésfa, 157
kifejtett típus, 253
kimenő paraméter, 206
kiszámítási modell, 58
kiszámított goto, 76
kiterjesztett, 126
kiterjeszthetőség, 315, 565
kivétel, 112, 114, 207, 222, 277
  beépített, 286, 288, 289
  beépített kivételkezelő, 294
  csoportosítás, 285
  dobása, 278
  ellenőrzött, 222, 295
  előredefiniált, 278, 293
  hierarchia, 285
  kezelés, 287
    dinamikus, 287
    engedélyezése, 287
    felülbírálása, 287
    tiltása, 287
  kivételbiztos, 114
  kivételkezelés, 215, 222–224, 277, 278, 526, 597, 621, 624
  paraméterezés, 285
  specifikálás, 278, 285
  továbbdobás, 278, 285
  utáni programfutás, 285
kliens, 630
kód, 30
  érték, 30
  pozíció, 30
  szám, 30
kódhalmaz-érték, 30
kommunikáció, 502, 625, 626, 632, 633
kompatibilitás, 568
kompozíciós szűrők, 693
konfluens, *lásd* átíró rendszer
konkatenáció, 191, 193
konkretizálás, 571
konkurens folyamat, *lásd* folyamat

konstans, 154, 172, 249
konstruktor, *lásd* típus
koplalás, 503
Korpela, J., 29
korutin, 272, 508, 509
Kowalski, R., 637
kölcsönös kizárás, 503, 511, 512, 527
kötött unió, 179
KRC nyelv, 596
Kristensen, B. B., 701
kritikus szakasz, 503, 531
kulcsszó, 37
kupac, 513
Kurtz, T., 701
külön fordítás, 254, 255
Kylix rendszer, 704

L

label, *lásd* címke
LACE nyelv, 704
lambda-kalkulus, *lásd* λ-kalkulus
lambda-kifejezés, *lásd* λ-kifejezés
láthatóság, 253, 338–340, 342, 347, 613, 617, 618
  félnyilvános, 110
  nyilvános, 110
  privát, 110
  publikus, 110
  védett, 110
láthatósági kör, 103
Latin-1 kód, 32, 138, 714
Latin-2 kód, 32, 714
lebegőpontos típus, *lásd* típus
lengyel forma, 156
lexikális elem, 28
lexikális hatókör, *lásd* statikus hatókör
lezárt, 268
Lieberherr, K., 692
lifespan, *lásd* élettartam
Linda nyelv, 513
Linda Program Builder környezet, 514
Liskov, B., 13, 68, 703
LISP nyelv, 9, 10, 12, 13, 210, 214, 215, 227, 235, 257, 264, 297, 589, 593, 595, 596, 706, 708, 711
  Common LISP, 210, 596, 707
  kivételkezelés, 297
lista, 187, 600, 601, 605, 607, 609, 614, 615, 626
  generátor, 605, 620
  láncolt, 604
  lusta, 593
  végtelen, 595
literál, 39, 172, 199, 237
logikai műveletek, 465
logikai nyelv, 61, 513
logikai processzor, 514
logikai programozás, 636–683, *lásd még* Prolog nyelv
  akkumulátor argumentum, 651
  akkumulátor módszer, 650
  akkumulátor pár, 651
  alagút argumentum, 651
  alkalmazások, 637, 682
  állítás, 636, 638, 640–642, 649
  állításfej, *lásd* szabályfej
  állítástörzs, *lásd* szabálytörzs
  argumentum, 641
  aritás, 638, 641
  az általánosítás módszere, 651
  az eredmény fokozatos közelítése, 650
  bizonyítási fa, *lásd* keresési fa
  cél, 636, 638–645
  célredukció, 642, 643
  célsorozat, 640–644
  definit klóz, 638
  deklaráció, 638
  direktíva, 638
  Horn klóz, 638
  kérdés, *lásd* cél
  keresési fa, 642–644, 649–651, 681
  keresési tér, *lásd* keresési fa
  kiterjesztések, 680
  klóz, *lásd* állítás
  legbal részcélkiválasztási módszer, 642–645
  levezetési fa, *lásd* keresési fa
  lista, 647
  logikai nyelvtanok, 638, 668, 680
  Mercury nyelv, 681
  ötödik generációs számítógépek, 680
  párhuzamosság, 680, 681
  predikátum, 638, 641, 642, 644, 648–651, *lásd* reláció

predikátumhívás, *lásd* cél
Prolog nyelv, *lásd* Prolog nyelv
redukció, *lásd* célredukció
rekurzív keresés, 649
reláció, 638, 640, 641, 644, *lásd* predikátum
rezolúció, 637
szabály, 638, 640–642, 644, 651
szabályfej, 640, 642
szabálytörzs, 640
tény, 638–642, 651
term, 646
lokális, *lásd* hatókör
longjmp(), 77
LP, *lásd* logikai programozás
Lucid nyelv, 507

M

Madsen, O. L., 701
magasszintű programozási nyelv, 50
makró, 235, 242, 243, 258, 345
mantissza, 140
Maple nyelv, 707
margó szabály, *lásd* hatókör
Marvel nyelv, 10
Mascetti, R., 10
Massachusetts Institute of Technology, 9, 67, 703, 706
Mathematica nyelv, 41
McCarthy, J., 9, 596, 706
megbízhatóság, 277, 564
meghatározatlan méretű tömb, 210
megszakítás, 277
megszorítatlan diszkriminánsú rekord típus, 210
megvalósítás elrejtése, 233
megvalósító rész, 109
Meijer, E., 596, 705
mellékhatás, 205–209, 236, 284, 593
mély kötés, 268
memóriagazdálkodás, 144, 164
szemétgyűjtéses, 145, 147, 149, 150, 170, 342
memóriaterület, *lásd* tárterület
Message Passing Interface, *lásd* MPI programkönyvtár
Meta-Object Factory, 695
metaprogramozás, 694
metódus, 45, 231, 392
friend metódus, 409
osztálymetódus, 411
Meyer, B., 2, 37, 67, 441, 444, 446, 489, 498, 704
mező, 171, 175, 176, 184
Microsoft, 9, 10, 66, 67, 694, 702, 706, 711
Microsoft Research, 694
Milner, R., 596, 710
minősített hivatkozás, 171
minősített név, 262
minta, 511, 595, 597, 598, 601
illesztés, 595, 597, 598, 601, 603, 613
típusra, 629, 630
összefogó, 607
rekordminta, 603
Miranda nyelv, 589, 591–594, 596, 599, 600, 605, 613, 614, 624
MIT, *lásd* Massachusetts Institute of Technology
mixin, 692
ML nyelv, 10–12, 123, 215, 264
Concurrent ML, 597
modul, 329
SML nyelv, 12, 589, 591–593, 595, 599–601, 603–605, 607, 610, 611, 613–618, 621, 623, 624, 710
szignatúra, 329
mnemonic, 320
mobil kód, 625, 629, 632, 633
módosíthatóság, 204, 254
modul, 109, 229, 233, 314, 334, 335, 338, 340, 342, 348, 597, 613, 614, 615, 616, 686
definíciós, 615, 616
export lista, 615
implementációs, 615, 616
import, 343, 615, 617
minősített név, 615
Math (SML), 592
nyelv (SML), 610, 616
nyitott, 318, 323
paraméterezett, 618
struktúra (SML), *lásd* struktúra
szignatúra (SML), *lásd* szignatúra
zárt, 318
Modula nyelv, 10, 510

Modula-2, 174, 184, 185, 195, 199, 201, 339, 341, 347, 348, 507, 508, 512, 697, 707
alprogram típus, 236
alprogrammal való paraméterezés, 214, 268
eljárás, 206
eljárás és függvény, 208
fordítási egység, 254
formális paraméter típusa, 210
főprogram, 221
függvény, 206
függvény visszatérési értéke, 215
halmaz típus, 195
korutin, 272
külön fordítás, 255
modul, 233
programegységek egymásba ágyazása, 256
Modula-3, 168, 198, 510, 707
alprogram specifikációja, 221
alprogram típus, 236
alprogrammal való paraméterezés, 214, 268
ciklus, 90
elágazás, 82–84
eljárás, 206
eljárás és függvény, 208
függvény, 206
függvény visszatérési értéke, 215
GOTO, 77
kivételkezelés, 223, 294
modul, 322, 323
paraméter alapértelmezett értéke, 229
READONLY paraméter, 250
típusekvivalencia, 168
végrehajtási szál, 272
moduláris
dekompozíció, 315
érthetőség, 316
folytonosság, 316
kompozíció, 316
tervezés, 315
védelem, 316
modularitás, 334, 335, 594
modularizáció, 348
modulfüggőség, 334, 342, 348
mohóság
jelölés, 593
vizsgálat, 593
Moller-Pedersen, B., 701
monád, 620, 624
alapművelet, 620
I/O, 621, 622, 624
monadikus
művelet, 620
összekapcsolás, 621, 622
osztály, 615, 620, 622
mondatszerű leírás, 46
monitor, 504, 511, 512
monomorf típusrendszer, *lásd* típus
Motorola 68000, 52
MPI programkönyvtár, 514, 543
mutable, *lásd* típus
mutató, *lásd* típus
értékének elrejtése, 112
mutató típus, *lásd* típus
mutatóaritmetika, 152, 190, 213
mutual exclusion, *lásd* kölcsönös kizárás
művelet
absztrakt, 379
infix, 156, 158, 264, 344, 600, 609, 610
precedencia, *lásd* precedencia
prefix, 600
műveletosztály, 378
Myhrhaug, B., 710

N

namespace, *lásd* névtér
NaN, *lásd* nem szám
Naur, P., 700
nem szám, 141
nem teljesen meghatározott típus, 210
nem-lokális, 266
NetBeans eszköz, 695
Netscape, 706
név
ekvivalencia, 168
előre definiált, 38
névkötés, 381
szerinti megfeleltetés, 191
szerinti paraméter-megfeleltetés, 221, 229
szerinti paraméterátadás, 243, 246, 258, 268

névtér, 328
névvel jelölt forma, 221, 229
normál forma, 140, 592–594, 597, 626
Nygaard, K., 701, 710

NY

nyelvtervezés alapelvei, 13

O, Ó

Object Management Group, 695
Object Pascal nyelv, 511, 704
  absztrakt osztály, 438
  adatrejtés, 408
  Delphi eszköz, 704
  destruktor, 403
  interfész, 452
  kivételkezelés, 296
  konstruktor, 402
  közös ős, 441
  logikai típus, 139
  metódus, 413
  osztály, 395
  osztályadat, 413
  osztálymetódus, 413
  öröklődés, 420
  polimorfizmus, 432
Object-Z nyelv, 464
Objective Caml nyelv, 596, 710
Objective-C nyelv, 216, 232, 234, 708
  protocol, 450
  többszörös öröklődés, 451
objektum, 391, 621
  belső állapota, 392
  létrejötte, 110
  megsemmisítése, 111
  objektumdiagram, 400
objektumorientált párhuzamos nyelv, 507, 513
objektumorientált programozás, 391–457
Occam nyelv, 272, 507, 512, 534
oktet, 29
olvashatóság, 204, 208, 263
on-demand instantiation, *lásd* igény szerinti példányosítás
opció, 577
operációs rendszer, 62, 67, 277
operátor, 128, 135, 137, 142, 143, 152, 155, 264, 337
  asszociativitás, 156, 160
  balasszociatív, 160
  dereferencia, 150
  hivatkozás-feloldás, 190
  infix, 156
  jobbasszociatív, 160
  kiértékelési sorrend, 208, 264
  konverziós, 123, 129–131, 137, 142, 161, 339
  logikai, 139, 176
  lusta, 139, 163
  mixfix, 156
  mohó, 139, 163
  postfix, 156
  precedencia, 156, 158, 208, 264
  prefix, 156
  referenciaképzés, 149
  relációs, 130
  szabad, 155, 344, 349
  túlterhelés, 96, 155, 172, 190, 198, 264, 271, 343, 349
optimalizáló fordítóprogram, 51
Oracle Designer eszköz, 563
ortogonalitás, 14
Ossher, H., 692
osztály, 234, 391, 415
  absztrakt osztály, 437
  beágyazott osztály, 454
  friend osztály, 409
  helyesség, 491
  invariáns, 477, 484, 485
  környezet, *lásd* típus
  leszármazott, 378
  méret
    belső, 574
    közvetlen, 573
    külső, 574
    növekményes, 573
    sima, 574
    teljes, 574
  osztályadat, 411
  osztálydiagram, 400, 414, 423
  osztálymetódus, 411
  tagosztály, 454
  végleges (final), 421, 579
osztály típus, *lásd* típus
osztályozás típus, *lásd* típus

osztott
memória, 502, 507
objektum, 527
változó, 507, 508, 511, 512
overloading, *lásd* túlterhelés

Ö, Ő

önelemzés, 694
önhívás, *lásd* rekurzió
önmódosító programok, 76
őrfeltétel, 523, 530
öröklődés, 182, 345, 415, 423, 483
Ada 95 nyelv, 422
C++ nyelv, 419
C# nyelv, 421
dinamikus, 693
dinamikus kötés, 426
Eiffel nyelv, 420
interfész, 450
ismételt
Eiffel nyelv, 436
Java nyelv, 421
közös ős, 440
Object Pascal nyelv, 420
öröklési módok, 424
összekapcsolás, 444
polimorfizmus, 426
rombusz-öröklődés, 447
SIMULA 67 nyelv, 416
Smalltalk nyelv, 418
többszörös, 441
virtuális művelet, 428
összefogó minta, *lásd* minta
összekapcsolás, 444
összeszerkesztés, 320
összetett típus, *lásd* típus
ötletszerű kötés, 268

P

package, *lásd* csomag
Par C nyelv, 512
Paradigm+ eszköz, 563
paragrafus, 65
ParAlf nyelv, 513
Parallel Virtual Machine, *lásd* PVM programkönyvtár
paraméter, 237, 257, 265, 344, 521, 577
adatmozgató, 246
alapértelmezett érték, 219, 228, 263
alprogram, 214, 246, 267
alprogram típusa, 217
átadási módja, 237
be- és kimenő, 237
bemenő, 206, 237
cím szerinti, 153, 239, 284
címke, 215
eredmény szerinti, 240
érték, 209
érték szerinti, 237
érték/eredmény szerinti, 241
igény szerinti, 253
kimenő, 237, 301
konstans, 249
lista, 209, 227
változó hosszú, 218
másolat szerinti, 242
megfeleltetés, 209
meghatározatlan méretű tömb, 210
megosztás szerinti, 251
módja, 250
név szerinti, 221, 243, 258, 268
név szerinti megfeleltetése, 229
parancssori, 220
pozicionális megfeleltetése, 221, 229
száma, 218
szövegszerű helyettesítése, 242, 258
típus, 215, 217, 344, 345
többdimenziós tömb, 211
parciális függvény, 471
párhuzamos, 62
generátor, 603, 620
kiértékelés, 597, 625, 626
klóz, 509
kompozíció, 633
környezet, 284, 285, 513
látszólagos párhuzamosság, 272, 502
programozás, 501
programozási nyelv, 505
szálak, 510
tömbfeldolgozás, 507
valódi párhuzamosság, 502
Parlog nyelv, 506, 513
Pascal nyelv, 11–13, 34, 37, 339, 347, 348, 507, 512, 708

alprogram fejléce, 216
alprogram törzse, 222
alprogrammal való paraméterezés, 214
altípus, 251
blokk utasítás, 72
ciklus, 88
Concurrent Pascal, 507, 512
egész típus, 135
elágazás, 81, 82
értékadás, 69
felsorolási típus, 131
fordítási egység, 254
formális paraméter típusa, 210
főprogram, 221
függvény visszatérési értéke, 215
goto, 76
halmaz típus, 195
intervallum típus, 251
karakter típus, 138
logikai típus, 139
paraméter-tömbök mérete, 210
paraméterátadás, 248
Pascal P4, 65
Pascal Plus, 507
Standard Pascal, 210, 214, 217, 236
Super Pascal, 512
típusellenőrzés, 217
Turbo Pascal, 12, 704
unió típus, 184
valós típus, 142
variáns rekord, 184
vektor típus, 188
visszatérés alprogramból, 224
példány, 392
példányosítás, 345, 381–385, 610, 614, 618, 620
explicit, 381–383
igény szerinti, 381, 382
lusta, 382
mohó, 382
specializáció, 384
Perl nyelv, 10, 12, 36, 116, 162, 198, 200, 201, 210, 257, 348, 708
aspektusok, 691
hasítótábla, 196
kivételkezelés, 299
Peterson, J., 596, 705
Peyton, J. S., 596, 705
pH nyelv, 507
PHP nyelv, 709
Piranha környezet, 514
PL/I nyelv, 9, 10, 12, 13, 138, 511, 709
alprogram típus, 236
alprogrammal való paraméterezés, 214
belépési pont, 229
CALL utasítás, 227
címke paraméter, 215
egész típus, 138
eljárás, 206
eljárás és függvény, 208
formális paraméter típusa, 210
függvény, 206
kivételkezelés, 286
külön fordítás, 254
rekurzív alprogram, 233, 266
statikus változó, 260
üres paraméterlista, 227
PL/SQL nyelv, 221, 223, 711
Plankalkül nyelv, 8
Plasmeijer, R., 596, 703
pointcut, *lásd* eseményleíró kifejezés
pointer, *lásd* mutató
polimorfizmus, 111, 123, 144, 218, 246, 263, 345, 353, 415, 426, 594, 609–611, 694
ad hoc, 360, 368–371, 610
altípusos, 360, 364
C++ nyelv, 430
C# nyelv, 433
egyszerű, 609
Eiffel nyelv, 433
Java nyelv, 432
Object Pascal nyelv, 432
parametrikus, 360, 364
szintaktikus, 362, 372
típuskényszerítéses, 360, 369, 370
túlterheléses, 360, 368
univerzális, 359–368
pontosság
dupla, 140, 141
egyszeres, 140
pontosvessző szerepe, 74
Pool-T nyelv, 507, 513
Portal nyelv, 509, 512

postfix, 156
Postscript nyelv, 10, 12
pozíció szerinti megfeleltetés, 191, 229
precedencia, 208, 264, 343, 597, 610
precompiler, *lásd* előfordító
prefix, 156, 264, 344
primitív művelet, 335
prioritás, *lásd* precedencia
privát, 339
procedurális programozás, 61
processz, *lásd* folyamat
profil, 221
program
  állapota, 61
  helyesség, 2, 487, 491
  karbantarthatóság, 2
  kompatibilitás, 3
  konstrukció, 591
  könyvtárak, 561–586
    karbantartás, 586
    megvalósítás, 586
    tervezés, 585
  megbízhatóság, 2
  összeegyeztethetőség, 3
  programegység, 209
  újrafelhasználhatóság, 2
  újrahasznosíthatóság, 2
Prolog nyelv, 10, 12, 636, *lásd* logikai programozás
  alaplista, 648
  alapterm, 647
  alkalmazások, 637, 679, 680
  állandó, 646
  állítás, 645, 652, 654, 656, 658, 660, 668–672, 678
  állításfej, 653, 655, 656, 668, 682
  állításlista, 652, 656, 657
  állítástörzs, 656, 657
  aritás, 653, 665, 675
  aritmetika, 662
  aritmetikai argumentum, 662
  aritmetikai kifejezés, 663
  aritmetikai predikátum, 662
  atom, *lásd* név
  beolvasás és kiírás, 668, 669
  cél, 645, 646, 652, 656
  célok megoldásainak összegyűjtése, 672
  célredukció, 652–654
  célsorozat, 645, 652–654, 656
  Concurrent Prolog, 506, 513
  cut, *lásd* vágó utasítás
  DCG nyelvtanok, 680
  deklaráció, 638, 670, 675, 677
  determinisztikus cél, 656
  dinamikus predikátum deklarációja, 670
  dinamikus predikátum létrehozása, 671
  dinamikus predikátumok, 670
  direktíva, 638, 645, 655, 667, 669, 675
  Edinborough Prolog, 637
  egyesítés, 654–656
  eljárás, *lásd* predikátum
  eljárás alapú modulrendszer, 675
  előtag műveleti jel, 667
  feltételes cél, 638, 658, 668
  feltételes szerkezet, *lásd* feltételes cél
  feltételes utasítás, *lásd* feltételes cél
  forrásmodul megváltoztatása, 677
  funktor, 647
  függvényszimbólum, 647
  illesztés, 653, 654, 656
  importálás, 675
  indeterminisztikus cél, 656
  ISO szabvány, 637, 646, 653, 667, 673, 674, 679
  keresési fa, 645, 652, 654, 656, 658, 661, 662, 668, 672, 683
  kiterjesztések, 680
  kivétel elkapása, 673
  kivétel kiváltása, 673
  kivételkezelés, 673
  környezet kezelő predikátumok, 668
  közbevetett műveleti jel, 667
  legbal részcélkiválasztási módszer, 645
  lista, 647, 648
  logikai változó, 646, 660, 663
  logikán kívüli predikátumok, 668
  Marseille-i Prolog, 637
  mellékhatásos predikátumok, 668
  meta-argummentum, 661
  metacél, 661
  metalogikai predikátumok, 662
  metapredikátum, 661, 677, 678
  metapredikátum-deklaráció, 677, 678, 683

modul exportlistája, 675
modul lokális predikátumai, 675
modul nyilvános predikátumai, 675
moduldeklaráció, 675
modulnév-kiterjesztés, 677, 678
modulrendszerek, 675
műveleti jelek, 666
előre definiált készlete, 667
felüldefiniálása, 667
létrehozása, 667
prioritása, 667
törlése, 667
zárójelezése, 667
negáció, 660, 668
név, 646
név alapú modulrendszerek, 675
nonvar term, 664
nyilvános predikátum, 676
összetett term, 647
predikátum, 645, 646, 648, 652, 654–666, 668–675, 677, 678, 681, 683
predikátum alapú modulrendszer, 675
programbetöltés, 668, 669, 678
programmódosító predikátumok, 668, 670, 671, 678
reláció, 645
részleges lista, 648
SICStus Prolog, 637, 639, 645, 654, 675, 677–679, 681
statikus predikátumok, 670
struktúra, *lásd* összetett term
szabály, 645, 652, 661
szám, 646
tagadás, 660, 668
tény, 652, 653, 658, 675
term, 646, 664
termek összehasonlítása, 664
termek szabványos rendezése, 664, 672
termek típusa, 664
termműveletek, 665
tiszta Prolog, 652, 658, 673
utolsó hívás optimalizáció, 657
utótag műveleti jel, 667
vágó utasítás, 638
valódi lista, 647, 648
végrekurzió-optimalizáció, 657
visszalépés, 653
protokoll, 221
prototípus, 217, 234
PVM programkönyvtár, 514, 553
Python nyelv, 709
aspektusok, 691
egész típus, 138

R

randevú, 515, 520, 521, 525
RATFOR előfordító, 235
Rational Rose eszköz, 563, 695
Rational Unified Process, 564, 695
redex, 592
legbaloldalibb legbelső, 593
legbaloldalibb legkülső, 593
redukció, *lásd* átírás
referencia, 125, 129, 143, 145, 150, 153, 170, 189, 190, 192, 342
képzés, 149, 151, 153
típus, 143, 248
reflektív program, 694
regiszter, 52, 265
rekord, 170, 174, 176, 283, 603, 613
diszkriminánsos, 185
kivéve művelet, 604
mező, 603, 613
minta, 603
variáns, 184, 613
rekurzió, 45, 98, 232, 265, 597
kölcsönösen rekurzív, 594, 607
közvetett, 233, 234
rekurzív algoritmusok, 99
rekurzív függvény, 594
rekurzív hívás helyettesítése, 609
végrekurzió optimalizációja, 266
rendezett *n*-es, 600, 601, 603
rendezett nullás, 601, 622
reprezentáció, *lásd* ábrázolás
reprezentációs függvény, *lásd* ábrázoló függvény
résztömb-képzés, 191
rezolúció, 637
RISC processzorok, 12, 51
Ritchie, D., 9, 701
van Rossum, G., 709
rövidre zárt, 246
Ruby nyelv

aspektusok, 691
runtime error, *lásd* futásidejű programhiba
RUP, *lásd* Rational Unified Process

S

sablon, 214, 335, 345, 353, 618
paraméter
aktuális, 381
alprogram, 378
formális, 381
modul, 379
objektum, 379
típus, 376
sablon-szerződés modell, 373–377, 383
SASL nyelv, 596
Sather nyelv, 215, 230, 270
Scalable Vector Graphics, 695
Scheme nyelv, 12, 596
Schönfinkel, M., 599
scope, *lásd* láthatósági kör
Sebesta, R., 203
sed nyelv, 10
sekély kötés, 268
Sethi, R., 203
SICStus Prolog, *lásd* Prolog nyelv
Simonyi, Ch., 694
SIMULA 67 nyelv, 12, 214, 215, 248, 272, 509, 697, 710
absztrakt osztály, 438
osztály, 393
öröklődés, 416
szemétgyűjtés, 112
Sina nyelv, 693
skalár típus, *lásd* típus
Smalltalk nyelv, 10, 12, 67, 123, 129, 162, 178, 220, 264, 694, 710
absztrakt osztály, 438
adatrejtés, 407
aspektusok, 691
blokk utasítás, 74
ciklus, 97
elágazás, 86
értékadás, 70
osztály, 394
osztályadat, 412
osztálymetódus, 412
öröklődés, 418
skalár típus, 129
Smalltalk-80, 12
SML nyelv, *lásd* ML nyelv
SNOBOL nyelv, 10, 12, 257, 268
SNOBOL4 nyelv, 11, 12
Software Through Pictures, 563
sor, 187
sorozat, 600, 601
eleme, 608
generátor, *lásd* generátor
számtani, 603
végtelen, 603, 609
sorszám, 75
SP/K nyelv, 512
specifikáció, 61, 209, 216, 233, 262
SPICE eszköz, 564
SQL nyelv, 333, 711
SR nyelv, 225
Standard ML, *lásd* ML nyelv
Standard ML nyelv, *lásd* SML nyelv
Standard Pascal nyelv, *lásd* Pascal nyelv
Standard Template Library, 271, 564, 694
statikus, 108
élettartam, 112, 260, 265
hatókör, 257, 268
lánc, 266
összekapcsolás, 343
predikátum, 670
puffer, 113
static módosító, 109
tárterület, 104, 144, 265
típus, 629
típuskényszerítés, 371
változó, 144, 149, 164
STEELMAN nyelv, *lásd* Ada nyelv
STL, *lásd* Standard Template Library
Strand nyelv, 514
STRAWMAN nyelv, *lásd* Ada nyelv
strong typing, *lásd* erős típusosság
Stroustrup, B., 9, 564, 700, 702
struktogram, 50
struktúra, 329, 610, 616, *lásd még* rekord
átlátszatlan, 618
beágyazott, 618
import, 617
kifejezés, 616
megfelel, 616, 617

nyitott, 617
strukturális ekvivalencia, 168
strukturált programozás, 65, 66
Sun Microsystems, 10
SunOne Studio eszköz, 690
Super Pascal nyelv, *lásd* Pascal nyelv

SZ

szabad előfordulás, *lásd* változó
szabad lista, 144, 149
szabad unió, 179
szál, 105, 114, 510, 511, 626, 627
számábrázolás
BCD, 138
fixpontos, 130, 140, 142
kettes komplemens, 134, 135
lebegőpontos, 130, 140, 142
számítási modell, 588, 589
számlálásos ciklus, *lásd* ciklus
számliterál, 39
szándékorientált programozás, 694
származtatott típus, 161
szegmens, 321
szekció, 65
szekvencia, 71–75
szekvenciális vezérlés, 45
szelekció
direktszorzat, 171
hasítótábla, 197
unió, 179
vektor, 189
szelektív várakozás, 522
szelektor, 79, 170, 171, 174, 177, 179, 180, 182, 184, 189, 192, 194, 196, 198
függvény, 601, 604
szeletelés, 191
szemafor, 504, 508, 509, 527, 531
szemét, 113
szemétgyűjtő, 111, 112, 145, 149
szerver, 630
szignatúra, 262, 329, 610, 611, 616, 618
absztrakt, 610
elsődleges, 616, 617
finomítás, 617
kifejezés, 616
kiterjesztés, 617
öröklődés, 617
szimmetrikus vezérlési modell, 272
szimuláció, 272
szinkron
kommunikáció, 506
nyelv, 507
üzenetküldés, 512
szinkronizáció, 502
minta, 633
szinkronizációs specifikáció, 62
szintaktikus cukor, 135, 157, 178
szkriptnyelv, 218
szökőkút tervezési módszer, 205
szubjektumorientált programozás, 692
szűrő, 602

T

tanács, 689
Tarr, P., 691
tárterület
allokálása, 110, 111
automatikus, 105
dinamikus, 104, 112
statikus, 104
típusai, 104
taszk, 511, 514, 515
attribútum, 519
lezárása (terminálása), 519
objektum, 517
típus, 517
végrehajtása, 518
Tcl nyelv, 10, 12, 711
template, 355, 358, 359, 384, 385
template metaprogramozás, 694
termelő–fogyasztó feladat, 528
terminálás, 519, 522, 524
terminálló függvény, 481, 488
tevékeny várakozás, 503, 526
TEX nyelv, 10
thread, *lásd* szál
Tiny BASIC, *lásd* BASIC nyelv
típus, 119–165, 588, 591, 609
absztrakt, 127, 332–350, 379, 393, 580, 594, 609, 614, 617, 624
algebrai, 616
adatkonstruktor, 595, 598, 601, 612, 614, 624
alaptípus, 611

algebrai, 594, 595, 598, 609, 612, 613, 623
absztrakt, 616
alkalmazás, 611
állandó, 611
alprogram, 236
általánosabb, 611
altípusosság, 218, 372
invariáns, 385
kovariáns, 385
átlátszatlan, 335
csomópont, 581
definíció, 119
deklaráció, 594
delta, 142
dinamikus, 126, 625, 629
direktszorzat, 171
diszkrét, 127, 131, 195
egész, 127, 133, 163
ekvivalencia, 168, 198
elemi, 127
ellenőrzés, 217, 254
felsorolási, 127, 128, 131, 163, 612
fixpontos, 127
Hindley–Milner típusrendszer, 594, 611
immutable, 169, 177, 183, 188, 192, 198, 253
invariáns, 121, 176
karakter, 138, 164
kényszerítés, 263, 372
dinamikus, 371
implicit, 370
statikus, 371
kifejezés, 611
kiterjesztett, 129
kiterjeszthető, 624
konkrét, 339, 578
konstrukció, 131, 167, 333, 348, 600, 620, 622
alaptípus, 612
direktszorzat, 170
fájl, 187
halmaz, 187, 195
hasítótábla, 187, 196
iterált, 187
lista, 187, *lásd* lista
osztály, 612, 615
rekord, *lásd* rekord
rendezett n-es, *lásd* rendezett n-es
sor, 187
sorozat, *lásd* sorozat
tömb, 187, 193, *lásd* tömb
unió, 178
vektor, 187, 188
konstruktor, 110, 284, 339, 342, 344, 400, 612, 614, 615, 624
változó, 615
konverzió, 123, 125, 154, 218, 339
bővítő, 123
szűkítő, 123
környezet, 611
lebegőpontos, 127
legáltalánosabb, 594, 611
levezetés, 594, 611
logikai, 139, 163
magasabbrendű, 609
megvalósítás, 121
mintaillesztés, 630
mutable, 169, 177, 183, 188, 192, 253
mutató, 127, 143, 145, 146, 148–152, 154, 190, 191, 284, 337, 341, 344, 345, 347, 621
művelet, 120, 122, 130, 134, 141, 148, 171, 179, 189, 196, 333, 335, 337, 345
nem teljesen meghatározott, 375
numerikus, 599
osztály, 128, 162, 588, 591, 599, 609–611
környezet, 611
osztályozás, 127
összetett, 127
paraméter, 344, 354, 363, 376–381
explicit, 361
implicit, 361
levezetése, 384
megfeleltetés, 383
megszorítása, 377, 378
típusosztály, 378
referencia, 143, 189
rendszer, 611
jobbára monomorf, 372
monomorf, 371
polimorf, 371
szigorúan monomorf, 371

skalár, 127, 129, 197
small érték, 142
specifikáció, 120, 334, 339, 340, 348, 393, 618
statikus, 600, 629
származtatott, 161, 612, 614
szinoníma, 612, 614, 616
teljesen meghatározott, 375
típusértékhalmaz, 119, 120, 130, 134, 136, 140, 146, 170, 178, 185, 188, 193, 195
típuskikövetkeztető rendszer, 123
típusosság, 122, 146, 162
erős, 123, 175, 383
gyenge, 123, 383
statikus, 123, 594
szigorú, 594
törlés, 382
valós, 127, 130, 140, 164
változó, 361, 609–612, 614
tisztán funkcionális, *lásd* funkcionális
Tofte, M., 596, 710
Together eszköz, 563
totális függvény, 471
többalakúság, *lásd* polimorfizmus
többdimenziós tömb, 211
többirányú elágazás, *lásd* elágazás
többszörös értékadás, 69, 70
tömb, 152, 187, 188, 190, 192, 193, 201, 278, 604, 620
dobozolatlan, 605
dobozolt, 605
egyszeresen hivatkozott, 604
eleme, 605
frissíthető, 621
generátor, 605
határozatlan indexhatárú, 189, 190
index, 605
meghatározatlan méretű, 210
többdimenziós, 211
törzs, 209, 222
tranzakció alapú vezérlés, 100
Trellis nyelv, 270, 514
túlterhelés, 107, 261, 343, 349, 372, 610
operátor, 155, 172, 190, 198, 343, 349
Turbo Pascal nyelv, *lásd* Pascal nyelv
Turing nyelv, 224
Turing, A., 589
Turing-gép modell, 61, 589
Turner, D., 596

U, Ú

UCSD P-System, 65
újrafelhasználhatóság, 204, 254, 316, 318, 566, 568
UML, 685, 695
unáris, 155, 157
Unicode, 30, 138
Unified Process, 695
unió, 178, 199
címkézett, 132, 183
kötött, 179, 199
szabad, 179, 199
UNIX, 279, 701
Epoch, 113
shell, 116
utasítás, 206, 222, 285
aritmetikai, 286
elérhetősége, 78
törzs, 74
utófeltétel, 472, 475

Ü, Ű

üres típus, 206
üres utasítás, 71
ütemezés, 62
üzenetküldés, 67, 502, 507, 512, *lásd még* kommunikáció

V

VAL nyelv, 505, 507, 513
valós idejű programozás, 509
valós idejű vezérlés, 100
valós típus, *lásd* típus
változó, 591, 593, 597, 599
automatikus, 112, 144, 149, 164
dinamikus, 144, 149
egyszeresen hivatkozott, 619, 620
frissíthető, 621
globális, 144, 284
lokális, 108, 109, 292
statikus, 144, 149, 164
szabad előfordulás, 594
Value Algorithmic Language, *lásd* VAL nyelv

várakozási sor, 520, 522
védett, 339
    adat, 530
    egység, 208
    objektum, 514, 530, 531
    típus, 531
végrehajtási
    megközelítés, 62
    szál, 62
    verem, 144, 264
végrekurzió optimalizációja, 266
végtelen adatszerkezet, 595
vektor, 187, 188, 193, 196, 199
vektorizálás, 506
verem, 105
vessző operátor, 207
vezérlésátadás, 45, 75
vezérléscsere, 508
vezérlési absztrakció, 204, 216, 269, 353
vezérlési megközelítések, 61
virtuális osztály, 365
Visual BASIC, *lásd* BASIC nyelv
visszatérési érték, 206, 215, 224
    több, 215
vízesés tervezési módszer, 205
vonatkozások szétválasztása, 686, 691
VS FORTRAN, *lásd* FORTRAN nyelv

W

Wadler, P., 596, 705
Wall, L., 708
Warren, D. H. D., 637
WayPointer rendszer, 695
where, *lásd* hatókör
while-programok, 58, 59
Wijngaarden-nyelvtan, 700
Wirth, N., 65, 82, 89, 322, 707, 708
World Wide Web, 10
World Wide Web Consortium, 695

X

Xerox Palo Alto Research Center, 710
XML, 12
XML Metadata Interchange, 695

Z

Z nyelv, 464, 704
Zermelo-Fraenkel halmazkifejezés, 595
Zuse, K., 8

A kiadvány átfogó ismertetést nyújt az informatikusok számára kulcsfontosságú eszközkészletről, a programozási nyelvekről. Bemutatja az informatika fél évszázados fejlődése során létrejött nyelvi eszközöket, részletesen elemzi kialakulásuk hátterét, a tervezők céljait. Áttekinti a problémamegoldásban leghatékonyabbnak bizonyult módszereket. Részletesen összehasonlítja a különböző programozási nyelvekben hasonló céllal létrehozott eszközöket. Az ismertetésben nagy hangsúlyt kap mind a nyelvek, mind az eszközök mögött lévő szemléletmód bemutatása. Tárgyalja az imperatív, objektumorientált, funkcionális, logikai, párhuzamos, eseményvezérelt és az aspektusorientált megközelítéseket. Ismerteti a nyelvek tervezésének fontosabb elveit, gyakorlati példákkal segítve a bemutatott megoldások megértését és értékelését.

Néhány a tárgyalt témakörök közül:

- Programozási nyelvek tervezésének kérdései
- Történeti áttekintés a nyelvekről
- Vezérlési szerkezetek, utasítások
- Adattípusok
- Procedurális programozás
- Objektumorientált programozás
- Az absztrakt adattípusok, mint a modellezés eszközei
- Párhuzamos programozást támogató nyelvi eszközök
- Programkönyvtárak tervezési szempontjai
- A funkcionális programozás elemei
- Logikai programozási nyelvek
- Aspektusorientált programozás

A könyvet elsősorban a felsőfokú informatikai képzésben résztvevő hallgatóknak és gyakorló informatikusoknak ajánljuk. Főiskolai vagy egyetemi szintű képzésben lehet értékes, amikor a hallgatók ismernek már egy-két programozási nyelvet és a közös vonásokat kihasználva bővíteni szeretnék nyelvi ismereteiket.

A könyv szerzőinek csapata az Eötvös Loránd Tudományegyetem programozási nyelvek témakörével foglalkozó oktatóiból és kutatóiból, valamint Magyarország néhány meghatározó informatikai fejlesztési központjának szakembereiből alakult. Együttes munkájuk eredményeként egy olyan könyvet tarthat kezében az Olvasó, amelyben a szerzők több évtizedes oktatói és fejlesztői munkájuk során szerzett tapasztalataikat rendszerezve, érthetően, olvasmányos formában adják át.

OM Oktatási Minisztérium

A kiadvány az Oktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolított felsőoktatási tankönyvtámogatási program keretében jelent meg.

KISKAPU
www.kiskapu.hu

Kategória: programozás
Felhasználói szint: haladó

ISBN szám: 963 9301 47 7
Ára: 5600 Ft

ISBN: 963 9301 47 7
9 789639 301474