1. Alapok

1.1. A programfájlok szerkezete

A bash programok tulajdonképpen egyszerű szöveges fájlok, amelyeket bármely szövegszerkesztő programmal megírhatunk. Alapvetően ugyanazokat a parancsokat használhatjuk bennük, amelyeket parancssorból is kiadhatunk, csak ezekhez társul még néhány programvezérlési szerkezet. A héjprogramokat általában a

```
#!/bin/bash
```

sorral kezdjük. Ez jelzi a parancsértelmezőnek, hogy egy bash programról van szó, és ez alapján kezeli az állományt. A programok végére érdemes odaírni az

```
exit 0
```

parancsot. Bár a programfájl végén a program egyébként is visszatér a parancssorhoz, csak az `exit` parancs használatával tudunk pontosan egy értéket visszaadni a programból. A programban megjegyzéseket a `#` jel után helyezhetünk el.

1.2. A programok futtatása

A programok futtatására két lehetőségünk van. Az első az, hogy az `sh` parancs után beírjuk a fájl nevét az alábbi formában:

`sh` *fájlnév*

Futtathatjuk azonban a programokat a teljes elérési útjuknak a megadásával is (ez a gyakoribb). A programnak az első használat előtt futtatási jogot kell adnunk a `chmod` parancs segítségével.

2. Változók

2.1. Változók létrehozása

A bash nyelv gyengén típusos, ami azt jelenti, hogy a változókat nem kell deklarálnunk, hanem azok az első használatkor létrejönnek, típusuk pedig a használat módjának megfelelően alakul. Tulajdonképpen minden változó egyszerű karakterlánc. Ha egy már meglévő változót szeretnénk használni, akkor arra úgy kell hivatkoznunk, hogy egy `$` jelet írunk a neve elé:

`$`*név*

A `$` jel használata minden esetben kötelező, ha a változót már korábban létrehoztuk, és nem akarunk neki új értéket adni.

2.2. Értékadás a változóknak

Egy változónak értéket programon belül az egyenlőségjel használatával adhatunk:

név=érték

Ha a változó értékét a felhasználótól szeretnénk megkapni, akkor a `read` parancsra van szükségünk, amelyet az alábbi formában használhatunk:

`read` *név*

Ekkor a program megfelelő helyén a felhasználónak kell megadni a változó értékét. A parancs alapértelmezésben a bevitel lezárására az ENTER billentyű lenyomását várja. Ezen módosíthatunk a parancs kapcsolóival.

A parancs néhány kapcsolója:

`-s`	megtiltja, hogy a szöveg megjelenjen a képernyőn
`-n`*`szám`*	megadja, hogy pontosan hány karaktert vár, a karakterek számát a *szám* helyére kell írni
`-p` *`karakterlánc`*	megadja a kiírandó szöveget, a szöveget a *karakterlánc* helyére kell írni (kettős idézőjelek (″) közé zárva)

Ha már értéket adtunk egy változónak, akkor a kiíratása az `echo` paranccsal történik. Ez a paraméterül kapott változót, vagy karakterláncot a szabványos kimenetre írja:

```
echo $név
```

Ha az `echo` parancsnak olyan karakterláncot szeretnénk adni, amely szóközt is tartalmaz, akkor kettős idézőjelek (″) közé kell azt tennünk.

2.3. Belső változók

A nyelvnek vannak belső változói is. Ezeknek a nyelv automatikusan értéket ad, így azokat már nem kell a programban külön bevezetni.

A legfontosabb belső változók:

`$#`	a paranccsori paraméterek száma
`$?`	a legutoljára végrehajtott parancs visszatérési értéke
`$$`	a futó program folyamatazonosítója
`$!`	a háttérben utoljára végrehajtott parancs folyamatazonosítója
`$n`	az `n`-edik parancssori paraméter értéke, ahol `n` értéke legfeljebb 9 lehet
`$*`	minden parancssori paraméter egyben, egyetlen karakterláncként
`$@`	minden parancssori paraméter egyben, egyenként idézőjelbe téve

2.3. Előre definiált változók

A nyelvnek vannak előre definiált változói is, amelyeket nem csak a programokban, hanem a parancssorban is használhatunk. Ezek listáját a `set` paranccsal kérdezhetjük le.

Néhány előre definiált változó:

`HOME`	a felhasználói könyvtár elérési útját tartalmazza
`USER`	a felhasználó login nevét tartalmazza
`MAIL`	annak a mappának a nevét tartalmazza, ahová e-mailjeink érkeznek
`OSTYPE`	az operációs rendszerünk típusát tartalmazza
`PATH`	azokat az elérési útvonalakat tartalmazza, ahol a parancsértelmező futtatható állományokat keres, ha több ilyen könyvtár van, elérési útjaik közé kettőspontot kell tenni
`PWD`	az aktuális könyvtár elérési útját tartalmazza
`SHELL`	a jelenleg futó parancsértelmező elérési útvonalát tartalmazza
`TERM`	a terminál típusát tartalmazza
`COLUMNS`	a képernyő oszlopainak számát tartalmazza
`LINES`	a képernyő sorainak számát tartalmazza
`SSH_CLIENT`	az ssh kliens adatait tartalmazza

Az előre definiált változók értékét meg is változtathatjuk, egyszerű értékadással.

3.Kifejezések kiértékelése

3.1. Matematikai kifejezések

A matematikai műveletek elvégzésére az `expr` parancs áll rendelkezésre. Ez tudja az alapvető műveleteket, de csak egész számokat tud kezelni. A parancsnak paraméterként kell megadni az elvégzendő műveleteket, szóközökkel elválasztva egymástól a számokat és a műveleti jeleket. Ha egy `expr` parancs által kiszámolt értéket akarunk értékül adni egy változónak, a műveletet balra dőlő idézőjelek (`) közé kell zárni:

név=`expr *kifejezés*`

A balra dőlő idézőjel minden esetben a kifejezést kiértékeli, és a kimenő adatával helyettesíti azt. Ha a kifejezésben speciális karaktereket használunk, akkor azokat le kell védenünk a \ jellel.

A parancs által ismert műveletek:

`+`	összeadás
`-`	kivonás
`*`	szorzás
`/`	osztás
`%`	maradékképzés

3.2. Logikai kifejezések

A logikai műveletek elvégzésére a `test` parancs áll rendelkezésre. Igaz értékkel tér vissza, ha a kifejezés teljesül, hamissal, ha nem teljesül. A parancs egyszerűbb alakban úgy használható, hogy a kifejezést szögletes zárójelek közé írjuk:

[*kifejezés*]

A parancs fájlokra vonatkozó kapcsolói:

`-r`	értéke igaz, ha a fájl létezik, és olvasható
`-w`	értéke igaz, ha a fájl létezik, és írható
`-x`	értéke igaz, ha a fájl létezik, és futtatható
`-f`	értéke igaz, ha a fájl létezik, és közönséges fájl
`-d`	értéke igaz, ha a bejegyzés létezik, és könyvtár
`-h`	értéke igaz, ha a bejegyzés létezik, és közvetett hivatkozás
`-c`	értéke igaz, ha a bejegyzés létezik, és karaktereszköz-meghajtó
`-b`	értéke igaz, ha a bejegyzés létezik, és blokkeszköz-meghajtó
`-p`	értéke igaz, ha a bejegyzés létezik, és nevesített csővezeték
`-u`	értéke igaz, ha a fájl létezik, és setuid bitje be van állítva
`-g`	értéke igaz, ha a fájl létezik, és setgid bitje be van állítva
`-k`	értéke igaz, ha a fájl létezik, és sticky bitje be van állítva
`-s`	értéke igaz, ha a fájl létezik, és hossza nem nulla

A parancs karakterláncokra vonatkozó kapcsolói:

`-z` *karakterlánc*	értéke igaz, ha a karakterlánc hossza nulla
`-n` *karakterlánc*	értéke igaz, ha a karakterlánc hossza nem nulla
karakterlánc1 `=` *karakterlánc2*	értéke igaz, ha a két karakterlánc azonos
karakterlánc1 `!=` *karakterlánc2*	értéke igaz, ha a két karakterlánc nem azonos
karakterlánc	igaz értéket ad vissza

A parancs egész számokra vonatkozó operátorai:

n1 `-eq` *n2*	értéke igaz, ha *n1* és *n2* egyenlők
n1 `-ne` *n2*	értéke igaz, ha *n1* és *n2* nem egyenlők
n1 `-gt` *n2*	értéke igaz, ha *n1* nagyobb, mint *n2*
n1 `-ge` *n2*	értéke igaz, ha *n1* nagyobb, vagy egyenlő, mint *n2*
n1 `-lt` *n2*	értéke igaz, ha *n1* kisebb, mint *n2*
n1 `-le` *n2*	értéke igaz, ha *n1* kisebb, vagy egyenlő, mint *n2*

A parancs logikai operátorai:

`!`	tagadás
`-a`	logikai és
`-o`	logikai vagy

4. Programvezérlési szerkezetek

4.1. Elágazás

Kétféleképpen hozhatunk létre elágazást. Az első az `if`-es szerkezet, amely így néz ki:

```
if [ feltétel ]
then
  parancsok1
else
  parancsok2
fi
```

Értelemszerűen, ha a *feltétel* teljesül, akkor a *parancsok1* ág, ha nem teljesül, akkor a *parancsok2* ág fut le. Ha több ágú elágazást szeretnénk írni, akkor beszúrhatunk még `elif` ágakat.

A másik lehetőségünk elágazás létrehozására a `case`-es szerkezet, amely így néz ki:

```
case változó in
  érték1)
    parancsok1;;
  érték2)
    parancsok2;;
  ...
  *)
    parancsok;;
esac
```

Ez a szerkezet egy változó értékétől teszi függővé, hogy az elágazás melyik ága fusson le. A `*` ág akkor fut le, ha a változó az egyik *érték*kel sem egyezik meg.

4.2. Ciklus

Ciklusszervezésre három lehetőségünk van. Ezek közül az első a for ciklus, amelyet az alábbi módon hozhatunk létre:

```
for változó in lista
do
  parancsok
done
```

Ebben az esetben a *változó* felveszi az összes listabeli értéket, és mindegyikre végrehajtja a *parancsok*at. A *listá*t megadhatjuk egyszerű felsorolással is, de megadhatjuk egy olyan programmal is, amelynek kimenete egy lista (például `` `ls` ``).

A második lehetőségünk az úgynevezett `while` ciklus, amely így néz ki:

```
while [ feltétel ]
do
  parancsok
done
```

Ez mindaddig végrehajtja a *parancsok*at, ameddig a *feltétel* igaz. Ezzel szemben az `until` ciklus működése teljesen ellentétes:

```
until [ feltétel ]
do
  parancsok
done
```

Ez addig hajtja végre a *parancsok*at, amíg a *feltétel* nem igaz.

5. Függvények

5.1. Függvények létrehozása

Programunkon belül létrehozhatunk függvényeket is, majd azokat a programból bárhonnan meghívhatjuk. A függvények létrehozása úgy történik, hogy megadjuk a függvény nevét, majd utána kapcsos zárójelek között a függvény által végrehajtandó *parancsok*at:

```
név()
{
  parancsok
}
```

Ha létrehoztunk egy függvényt, akkor azt a programból a következőképpen hívhatjuk meg:

```
név paraméterek
```

Fontos, hogy a paraméterekre ugyanazokat a belső változókat használhatjuk, amelyeket a programban, de függvényen belül ezek a függvény paramétereit jelentik. A függvényeknek lehet visszatérési értékük is, ezt a `return` paranccsal tudjuk megadni.

7. Példaprogramok

7.1. Feladat

Döntsük el, hogy a programnak adott paraméter pozitív, negatív, vagy nulla, az eredményt pedig írjuk ki a képernyőre:

```
#!/bin/bash
if [ $1 -lt 0 ]
then
    echo "A megadott parameter negativ"
    exit 0
elif [ $1 -gt 0 ]
then
```

```
    echo "A megadott parameter pozitiv"
    exit 0
else
    echo "A megadott parameter nulla"
    exit 0
fi
exit 0
```

7.2. Feladat

Adjuk össze a természetes számokat 1-től a megadott paraméterig (az ismert képlet használata nélkül), az eredményt pedig írjuk ki a képernyőre:

```
#!/bin/bash
osszeg=0
i=1
while [ $i -le $1 ]
do
  osszeg=`expr $osszeg + $i`
  i=`expr $i + 1`
done
echo $osszeg
exit 0
```

7.3. Feladat

Adjuk össze a természetes számokat 1-től a megadott paraméterig (az ismert képlettel), az eredményt pedig írjuk ki a képernyőre:

```
#!/bin/bash
osszeg=`expr \( $1 \* \( $1 + 1 \) \) / 2 `
echo $osszeg
exit 0
```

7.4. Feladat

Vizsgáljuk meg a programunk által kapott paramétereket. Írjunk ki hibaüzenetet, ha a paraméterek száma nem pontosan egy, továbbá vizsgáljuk meg, hogy a megadott paraméter numerikus-e, és ellenkező esetben írjunk ki hibaüzenetet:

```
#!/bin/bash
if [ $# -lt 1 ]
then
  echo "HIBA: Keves parameter"
  exit 1
elif [ $# -gt 1 ]
then
  echo "HIBA: Sok parameter"
  exit 1
fi
case $1 in
  *[^0-9]*)
    echo "HIBA: A parameter nem szam"
    exit 1
esac
echo "Minden rendben"
exit 0
```

7.5. Feladat

Írjuk ki a képenyőre a megadott paraméter összes osztóját:

```
#!/bin/bash
i=1
while [ $i -le $1 ]
do
  if [ `expr $1 % $i` -eq 0 ]
  then
    echo $i
  fi
  i=`expr $i + 1`
done
exit 0
```

7.6. Feladat

Döntsük el a megadott paraméterről, hogy az prímszám-e, az eredményt pedig írjuk ki a képernyőre:

```
#!/bin/bash
if [ $1 -le 1 ]
then
  echo "A parameter nem primszam"
elif [ $1 -eq 2 ]
then
  echo "A parameter primszam"
else
  i=2
  while [ $i -lt $1 ]
  do
    if [ `expr $1 % $i` -eq 0 ]
    then
      echo "A parameter nem primszam"
      exit 0
    fi
    echo "A parameter primszam"
    exit 0
  done
fi
exit 0
```