Operációs rendszerek alapjai

3. gyakorlat

Mai gyakorlat

- Vezérlési szerkezetek
- Argumentumok kezelése
- Függvények

Pattern matching a terminálban

- Gyakran van arra szükség, hogy regex-t használjunk
  - adott kiterjesztésű fájlok törlése, listázása, tömörítése, átnevezése stb
  - pattern listás műveletekhez az `extglob` shell beállítás kell (`shopt -s extglob`)

`*`	bármilyen karakterből bármennyi
`?`	bármilyen karakterből pontosan egy darab
`[characters]`	felsorolt karakterek közül pontosan egy darab
`[[:class:]]`	megadott osztályból pontosan egy darab (`alnum`, `alpha`, `digit`, `lower`, `upper`)
`pattern-list`	`pattern1|pattern2|...`

`?(pattern-list)`	nulla vagy egy találat
`*(pattern-list)`	bármennyi találat
`+(pattern-list)`	legalább egy találat
`@(pattern-list)`	negálás
`!(pattern-list)`	negálás

`test` parancs - I.

- Visszatérési érték a `$?` változóban (`0`: true, `1`: false)
- `test flag file`
  - `-a`, `-e` kapcsoló: fájl létezik-e
  - `-f`, `-d`, `-h`, `-p`, `-S`, `-b`, `-c` kapcsoló: fájl létezik és sima fájl, mappa, szimbolikus link, pipe, socket, blokk eszköz, karakter eszköz
  - `-r`, `-w`, `-x` kapcsoló: fájl létezik és olvasható, írható, futtatható
- `test string1 operator string2`
  - `==` (`=`), `!=`, `<`, `>`
- `test operand1 operator operand2`
  - `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge` operátor: egyenlő, nem egyenlő, kisebb, kisebb egyenlő, nagyobb, nagyobb egyenlő
- `!`, `-a`, `-o` kapcsoló: kifejezések negálás, konjunkciója, diszjunkciója

`test` parancs - II.

- Használható terminálban, illetve elágazásokban
- Helyettesíthető a `[]` blokkal
  - `test flag file` helyett `[ flag file ]`
- Bash-ben
  - `[[]]` blokk
    - `test flag file` helyett `[[ flag file ]]`
    - `[[ string =~ regex ]]`
    - `-a` és `-o` helyett `&&` és `||`
  - `(())` blokk
    - egész számokra kapcsolók helyett szokásos operátorok
    - `-eq` helyett `==`
    - `-a` és `-o` helyett `&&` és `||`

Vezérlési szerkezetek - `if`

```
if <condition>; then
     <commands>
elif <condition>; then
     <commands>
else
     <commands>
fi
```

G03F01

- Olvass be két string-t!
- Regex segítségével döntsd el, hogy egész számok-e!
  - nemnegatív egész szám: `$string =~ ^[0-9]+$`
  - egész szám: `$string =~ ^(-)?[0-9]+$`
- Ha egész számok, akkor írd ki az összegüket!
- Ha nem egész számok, akkor jelezd a felhasználókat, hogy egészeket kell megadni!

Vezérlési szerkezetek - `case`

```
case <variable> in
     <pattern>)
          <commands>
          ;;
     *)
          <default commands>
          ;;
esac
```

G03F02

- Olvass be egy string-t!
- Ha egész számot adott meg a felhasználó, akkor írd ki a négyzetét!
  - regex `if`-hez: `^-?[0-9]+$`
  - pattern `case`-hez: `?(-)+([[:digit:]])`
- Ha nem egész valós szám, akkor írd ki az egészrészét!
  - regex `if`-hez: `^-?[0-9]+\.?[0-9]+$`
  - pattern `case`-hez: `?(-)+([[:digit:]])[.]+([[:digit:]])`
  - egészrész utolsó indexe (.-al együtt!) megkeresése: `expr match $number '.*[.]'`
  - ne felejtsd el, hogy például `[5.6]=5`, de `[-5.6]=6`!
- Ha nem szám, akkor jelezd a felhasználókat, hogy számot kell megadni!
- Ne felejtsd el beállítani az extended globbing-ot a `shopt -s extglob`paranccsal!

Ciklusok - `while`

```
while <condition>; do
    <commands>
done
```

- Amíg igaz a feltétel, addig végrehajtja a parancsokat
- Van végtelen ciklus
  - `while true; do`
  - `while :; do`
- `break`, `continue` szokásosan

Ciklusok - `until`

```
until <condition>; do
    <commands>
done
```

- Amíg hamis a feltétel, addig végrehajtja a parancsokat
- Van végtelen ciklus
  - `until false; do`
- `break`, `continue` szokásosan

G03F03

- Olvass be egy string-t!
- Ha nemnegatív egész számot adott meg a felhasználó, akkor írd ki az osztóit!
- Ha nem nemnegatív egész számot adott meg a felhasználó, akkor ezt jelezd, majd lépjen ki a script az `exit` paranccsal!
- Próbáld a ciklus feltételét megadni [[]] blokkal és (()) blokkal is!
- Próbáld ki while és until ciklussal is!

Ciklusok - `for`

```
for <variable> in <words>; do
     <commands>
done

for (( expr1; expr2; expr3 )); do
     <commands>
done
```

- Utóbbi ciklusban `expr1` inicializál és a ciklus addig fut, amíg `expr2` nem nulla, ekkor `expr3` kiértékelődik és `<commands>` parancsok futtatásra kerülnek (hiányzó kifejezés értéke 1)
- `break`, `continue` szokásosan

Brace expansion

- String(tömb) generálására alkalmas, `for` ciklus ezt fel tudja dolgozni
- `{string1,string2,...}`
  - megadott string-k kifejtése
  - tud konkatenálni, illetve a disztributív szabály is érvényes
    - `{a,b}{c,d}` eredménye `ac ad bc bd`
- `{x..y[..incr]}`
  - `x` és `y` egész szám, vagy karakter
  - `incr` egész szám
  - `x`-től `y`-ig generál incr ugrással
    - `{0..5}` eredménye `0 1 2 3 4 5`
    - `{a..z..5}` eredménye `a f k p u z`

G03F04

- Olvass be egy string-t!
- Ha nemnegatív egész számot adott meg a felhasználó, akkor írd ki az osztóit!
- Ha nem nemnegatív egész számot adott meg a felhasználó, akkor ezt jelezd, majd lépjen ki a script az `exit` paranccsal!
- Próbáld ki hagyományos és C-szerű ciklussal is!

Fájlkezelés I.

- Fájl nevét a mappa neve nélkül a `basename` paranccsal kaphatjuk meg
  - ebből tudjuk a kiterjesztést megtudni

```
name=$(basename file)
ext=${name##*.}
name_no_ext=${name%.*}
```

- `read` paranccsal olvashatunk standard inputról, fájl deskriptorból és fájlból
  - `-p` kapcsoló: üzenet az olvasáshoz
  - `-t` kapcsoló: timeout beállítása
  - `-s` kapcsoló: nem látszik a terminálon gépelés közben (jelszavakhoz)
  - `-r` kapcsoló: backslash-t nem escape karakterként értelmezi
  - `-d` kapcsoló: elválasztó beállítása (alapértelmezetten új sor)

Fájlkezelés II.

- Fájl beolvasása soronként

```
while read -r line; do
    echo "$line"
done < filename
```

- Fájl beolvasása tetszőleges karakterenként (például .csv estén vesszőnként)
  - `-d` kapcsoló addig olvas, amíg van elválasztó
  - `field1,field2` struktúra esetén csak az első mezőt olvassa be
  - IFS-t (Internal Field Separator) kell használni

```
while IFS=, read -r f1 f2; do
    echo $f1 $f2
done < filename
```

Fájlkezelés III.

- .csv fájl soronkénti beolvasása tömbbe

```
while IFS=, read -a array; do
    echo ${array[*]}
done < filename
```

- Fájl egészének beolvasása tömbbe (minden elem egy egész sor)

```
readarray array < filename
```

Argumentumok

- Meghívott scriptnek adhatunk át paramétereket
  - `int main(int argc, char* argv[])` megfelelője
  - `$0` - script neve, ahogy meghívtuk (például `./myscript.sh`)
  - `$#` - argumentumok száma
  - `$n` - n-edik argumentum
  - `$@` - argumentumok egy tömbben
- Indirect parameter expansion
  - `${!var}` esetén a var változó értékével jelölt változót érhetjük el
  - például ha `var` értéke 3, akkor a harmadik argumentumot kapjuk vissza

G03F05

- Írd ki tömbként a script argumentumait!
- Módosítsd a scriptet, hogy `for` ciklust használjon!
- Próbáld ki hagyományos és C-szerű ciklussal is!
    - utóbbihoz szükség lesz indirect parameter expansion-re

Kapcsoló kezelés

- Flag-eket lehetne argumentumként is feldolgozni, de elég kényelmetlen lenne
- `getopts` parancs segítségével jól kezelhetőek a kapcsolók és az argumentumok
- `getopts <flag_structure> flag`
  - beolvas egy kapcsolót a flag változóba a megadott struktúra szerint
    - egyszerűen fel kell sorolni a kapcsolókat
    - `:` karakter jelzi, hogy az adott kapcsoló argumentumot vár, ez elérhető az `$OPTARG` változóban
    - például `ab:` azt jelenti, hogy az `a` kapcsoló nem vár argumentumot, a `b` igen
  - kiírja, ha nem létező argumentumot próbál alkalmazni a felhasználó
  - kiírja, ha nem adott argumentumot adott kapcsolónak
  - `while` ciklusban érdemes használni, `case` elágazással pedig feldolgozni

G03F06

- Írj egy scriptet, ami tetszőleges számú paramétert fogad a következő struktúrával!
  - `-a` kapcsoló: a kapcsoló után megadott parancsra apropos lefuttatása
  - `-m` kapcsoló: a kapcsoló után megadott parancsra man lefuttatása
  - `-w` kapcsoló: a kapcsoló után megadott parancsra whatis lefuttatása
  - `-v` kapcsoló: apropos, man és whatis parancsok verziójának kiírása

Függvények

```
<function_name> () {
    <commands>
}

function <function_name> {
    <commands>
}
```

- A két szintaxis között nincs különbség
- Argumentumok nincsenek jelölve a függvény szignatúrában
- Argumentumok átadása és elérése script-hez hasonlóan
  - `function_name arg1 arg2 …`
  - `$0` - függvény neve
  - `$#` - argumentumok száma
  - `$n` - n-edik argumentum
  - `$@` - argumentumok egy tömbben

Visszatérési értékek, láthatóságok

- Nincs hagyományos visszatérési érték
  - `return` kulcsszóval nemnegatív egész számot beállíthatunk visszatérési status-ként
    - 0-255 között
  - ez a `$?` változóban érhető el
  - ez függvények és script-ek esetén is működik
- Szokás egyszerűen `echo`-val visszaadni az eredményt
- Alapértelmezetten minden változó globális
  - ezzel könnyen lehet visszatérési értéket szimulálni, ugyanis függvényben létrehozott változó elérhető a függvény lefutása után
  - `local` kulcsszóval létrehozhatunk lokális változókat
- Függvények elfedik a parancsokat
  - például ha definiálok egy `ls` nevű függvényt, akkor az `ls` parancs `command ls` néven érhető el

G03F07

- Olvass be egy string-t!
- Ha nemnegatív egész számot adott meg a felhasználó, akkor írd ki a faktoriálisát!
- Próbáld ki az eredmény átadását globális és lokális változóval is!
  - figyeld meg a `return` viselkedését nagy számok esetén
- Próbáld ki iteratívan és rekurzívan is!
  - rekurzióval interpretált nyelv lévén nem túl hatékony, nem ajánlott a használata (de jó gyakorlás)

Következő gyakorlat

- Archívumkezelés, tömörítések
- Alapvető programnyelvek fordítása terminálból
  - C, C++
    - röviden make és cmake is
  - Python
  - LaTeX

Érdekességek, olvasnivalók

- forkbomb
- Bash Reference Manual
- Luke Smith: Bash script without if statements