Általános INFORMATIKAI Tanszék

Számítógépek, számítógép rendszerek

3. A Unix használata

Dr. Vadász Dénes

Miskolc, 2005. február

TARTALOM

3. A UNIX operációs rendszer használata

A Unix

- nagyon elterjedt,
- multi-tasking, multi processing,
- időosztásos,
- általános célú operációs rendszer.

Használatának megtanulását sok, kiváló könyv segítheti [lásd az ütemterv irodalomajánlatát]. Nyájas bevezetést ad az Orlando Unix iskola. A Unix rövid történetét olvashatjuk itt.

Találunk könyvet a MEK-ben is: Rideg Márton: Unix alapismertek címmel.

Szokásos parancsértelmezős kezelői (kapcsolattartói) a ***burkok*** (shell). Különféle burkokkal kezelhetjük a Unixos gépeket!

Neve	programja	szokásos promptja[1]	előnye
Bourne shell	sh	$	Mindenütt! Shell programozásra!
Bourne again shell	bash	$	Interaktív használatra!
C shell	csh	%	Mindenütt! Interaktív használatra!
TC shell	tcsh	%	Interaktív használatra!
Korn shell	ksh	$	SVR4-ben ez az ajánlott!
Superuser shellje	sh	#	

3.1. A UNIX filozófia

Build on the work of others! Sok-sok kész segédprogram, szűrő létezik, amiből építkezhetünk. Nem írunk mindent újra, hanem használjuk a kész megoldásokat.

Az ezt segítő koncepciók

1.Egy Unixhoz készült program processzként általában a szabványos bementről (stdin) olvas, a szabványos kimenetekre (stdout/stderr) ír. Így szokás programozni. A segédprogramok, szűrők mind ilyenek. A szabványos ki/bementetek általában, alapértelemben a terminál kép-

[1] Megjegyzés: a fejezetben a példákban $ promptot fogok írni, ha hangsúlyozni akarom, hogy Bourne buroknak kell adni a parancsot. A > prompttal jelzem, ha mindegy, milyen burkot használunk.

ernyőjére ill. billentyűzetére vannak hozzárendelve. Nagyon sokszor az első fájlnévként adott argumentum a szabványos bemenet.

Ezt kihasználhatjuk. Ezt a konvenciót használhattuk pl. az ülés megszüntetésére, mikor is az üléshez tartozó burok processznek fájlvég (EOF) jelet adva azt terminálták.

Egy másik példa a következő: a Unix burkaiban tulajdonképpen nincs az MS DOS-ban megszokott *type* parancs (ami egy szövegállományt a képernyőre listáz). Használhatjuk helyette a *cat* nevű, valójában fájlok összefűzésére való (concatenation) parancsot!

Íme a példa:

```
> cat f1 f2 f3  # oszefuzo, stdout-ra irja a fajlokat
.....
> cat f1        # type helyett! Most nincs "összefűzés",
                # csak a stdout-ra íródik az f1 fájl!
....
```

Egy másik példa (itt is a hiányzó type parancsot "helyettesítjük"):

```
> more <file
```

Sőt:

```
> more file
```

2. A standard adatfolyamok átirányíthatók! (Az átirányító operátorokat később összefoglaljuk!)

Ezt persze már megszokhattuk az MS DOS-ban is. Jegyezzük meg azonban, hogy az átirányítást a DOS örökölte, az volt a későbbi! Az alábbi példában a már ismert *cat* segítségével szövegsorokat írunk az *f1* fájlba. Lássuk be, hogy itt a *cat* a szabványos bemenetről olvas egészen a fájlvég jelig, amit a CTRL/D billentyűkombinációval generáltunk. A cat szinte egy kis szövegrögzítőként viselkedi, csak azért nem szövegszerkesztő, mert ha egy sort bevittünk (a billentyűzet bufferből elküldtük), akkor az már nem javítható!

A példa:

```
> cat > f1
első sor
második sor
...
utolsó sor
CTRL/D
```

3. Csővezeték képezhető!

Ez is ismerős lehet. Ezt is tárgyaljuk később részletesebben. Mindenesetre ez a tulajdonság segíti, hogy különböző, már meglévő szűrőket használjunk a feldolgozásokban. Az alábbi példában a már ismert cat kiírná a képernyőre az f1 fájl tartalmát, de azt a grep mintakereső

szűrővel megszűrjük. Ennek eredménye sem kerül a képernyőre, mert tovább szűrjük a wc sor-szó-karakterszámlálóval. Csakis ennek az eredményei fognak kiíródni.

A példa:

```
> cat f1 | grep minta | wc
...
```

4. Az ún. *daemon* processzek szolgáltatásokat biztosítanak!

Például nyomtatási kérelmeket szolgálnak ki, a levelezést segítik stb. Sok démon futhat a használt Unix rendszerben, ezek szolgáltatásait kihasználhatjuk.

3.2. Honnan tanulhatjuk a UNIX használatot?

- Könyvekből.
- Kézikönyvekből, dokumentációkból.
- Segédletekből.
- Az *on-line manual*ból.
- Saját jegyzeteinkből, társainktól.
- A WWW lapjaiból (Orlando iskola, shell összefoglaló stb).

Az on-line kézikönyv, a *man*

A *man* parancs megjeleníti a kézikönyv (on-line manual) lapjait.

A kézikönyvekben tömörítve, formátumozó direktívákkal tárolt dokumentumok vannak. A man parancs több szűrőn (végül a *more* szűrőn) keresztül jeleníti meg a dokumentumokat, a bejegyzéseket.

A *more* legfontosabb parancsai:

- *space*: lapot dob,
- *Return:* sort dob,
- *q*: - kilép (quit).

A *man* hívásának szintaxisa

```
> man [ -opciok] [section] bejegyzes
```

Ha bővebben meg akarunk ismerkedni a man-nal, hívjuk a man-t a man bejegyzésre!

```
> man man
```

Ha pl. a parancsokat, azok szintaktikáját és szemantikáját akarjuk megismerni, miután minden parancsról van bejegyzés, ismerkedhetünk a parancsokkal a *man* segítségével. Sajnos, a bejegyzésekhez a klasszikus *man*-ban nincsenek "dzsókerek"! Ebből következően tudni kell a pontos bejegyzés neveket!

Segít, ahol van:

- az *apropos* adatbázis,
- a *whatis*,
- a *whereis*.

Segít az X11 GUI felületén a szebb formátumú, kezelhetőbb *xman*. Segít a gold-on az X11-es környezetű *info* (hypertext ismertető). Segít SGI-n a GL-es környezetű *insight* (könyvespolc). Segít SUNOS, SOLARIS környezetben az *Answerbook*.

Tanácsok

Miután a kézikönyv lapjai angol szövegek,

- tudni kell angolul olvasni.
- A fontos parancsok nevét pontosan tanuljuk meg. Használjunk parancs-kártyát, készítsünk jegyzeteket,
- A man lapok végén az utalások vannak kapcsolatos lapokra. Nézzük ezeket is!

3.3. Fontos parancsok, csoportosítva

A következőkben felsorolok fontos parancsokat, megemlítve a parancs nevét és rövid feladatát, jellemzőjét. A csoportosítás is segíthet egy-egy parancs pontos nevének, feladatának megtalálásában.

3.3.1. Manipulációk fájlokon, jegyzékeken

1. Editorok, szövegszerkesztők

ed	sororientált
vi (vim)	képernyő-orientált
e (emacs)	képernyő-orientált
pico	egyszerű, sok rendszeren

2. "Kiírók"

cat	concatenál, stdout-ra
pr	printel, stdout-ra
head	fájl első sorait, stdout-ra
tail	fájl utolsó sorait
more	lapokra tördelő szűrő
od	oktális dump (ömlesztés)

3. Jegyzékekkel kapcsolatos parancsok

ls	jegyzék tartalom lista (dir helyett)
mkdir	jegyzék készítés
rmdir	jegyzék törlés
cd	munkajegyzék váltás
pwd	munkajegyzék lekérdezés
chmod	fájl védelmi maszk váltás (nemcsak jegyzékre)
chown	fájl tulajdonos váltás (nemcsak jegyzékre)
file	fájl típus lekérdezés (nemcsak jegyzékre)

4. Másolások, mozgatások

cp	copy, másolás
mv	move, mozgatás (rename helyett is!)
ln (link)	"linkel"
rm (unlink)	"linket" töröl, remove: fájl törlés
find	keres fájlt egy fán és csinál is valamit (bonyolult, de nagyon hasznos!)

3.3.2. Állapotok (státusok), információk lekérdezése, beállítása

ps	processzek listázása
file, ls, pwd	ld. fönn
date	dátum, idő lekérdezés
who, w, rwho, rusers	ki van bejelentkezve?
rup	mely rendszerek élnek?
top, gr_top	erőforrás-használat csúcsok
osview, gr_osview	erőforrás-használat
last	utolsó bejelentkezések
finger	ki kicsoda?
passwd, yppasswd	jelszóállítás
chsh, chfn, ypchpass	név, induló burok stb. beállítás
ypcat	NIS (yellow pages) adatbázis lekérdezés
xhost	X11 munka engedélyezése
set	környezet (environment) lekérdezése
du, df	diszk használat

3.3.3. Processz indítás, vezérlés

sh, bash, csh, ksh, tcsh	shell idítás
exec	processz indítás
kill	processz "megölése", szignálküldés
sleep	processz altatása
wait	processz várakoztatás
at	processz indítása egy adott időpontban
nohup	kilépéskor ne ölje meg
test	kifejezés tesztelése
expr	kifejezés kiértékeltetése
if, case, for, do while	vezérlő szerkezetek
break, continue	vezérlő szerkezetek
echo	argumentumai echoja (meglepően hasznos valami)

3.3.4. Kommunikáció a világgal, felhasználókkal

ssh, telnet, rlogin, rsh	kapcsolatlétesítés,
rwho, rusers, finger	lásd fönt
write	üzenet konzolokra
talk, xtalk	interaktív "beszélgetés"
mail, Mail, pine, zmail	elektronikus levelezés kliense
ftp	fájl átvitel kliense
lynx, netscape, mozilla	WWW böngésző (kliens)

3.3.5. Hasznos szűrők

grep	mintakereső
awk, nawk	mintakereső feldolgozó
wc	sor, szó, karakterszámláló
sed	áradatszerkesztő
head, tail	ld. fönn
cut	mezőkivágó

3.3.6. Parancsok a tanuláshoz

man	laplekérdezés a kézikönyvből
apropos	kézikönyvben kulcsszó

whereis	hol van egy parancs
whatis	man lap leírás
xman	X11-es kézikönyv

stb.

Egy kis segítség a DOS-ból UNIX-ba áttérőknek.

3.4. A Bourne shell (sh)

A shell (burok) szót meghallva, kétféle értelmezésre kell gondolnunk. Hogy melyik értelemben használjuk a burok szót, az a szövegkörnyezetből derül ki.

A burok (shell) egy parancsértelmező processz

Tehát egy futó program. Van azonosítója (pid), ami lekérdezhető. Készíthet gyermek processzeket. A feladata:

- *készenléti jelet* (prompt) ad, ami azt jelzi, a szabványos bemeneti csatornán képes beolvasni parancsot (csövet, listát);
- *parancsot, csövet, listát* elfogad, elemez, esetleg átalakításokat végez, behelyettesít, végrehajt.

A shell egy programnyelv

Mint programnyelv,

- van vezérlési szerkezete;
- vannak (egyszerű) adatszerkezetei, változói.

Szövegszerkesztővel írhatunk ún. burok programokat (shell-szkripteket), később ezeket "odaadhatjuk" egy shell parancsértelmezőnek, hogy azt dolgozza fel.

3.5. Az sh burok, mint parancsértelmező

Tárgyalásához meg kell tanulnunk néhány alapfogalmat.

3.5.1 Alapfogalmak

3.5.1.1. A parancs fogalma

A *parancs* "fehér"[2] karakterekkel határolt szavak sora. A sorban az első szó a parancs neve, a többi szó a parancs argumentumai (általában opciók és módosítók, fájlnevek gazdagép és felhasználó azonosítók stb.).

[2] Fehér karakterek: a szóköz, tabulátor, sorvég karakterek.

A parancsot a burok beolvassa, elemzi, átalakítja és végrehajtja (a parancsnak megfelelően csinál/csináltat valamit). A parancs vagy külön processzben fut (a burok gyermek processzeként, szeparált processzként), vagy végrehajtja maga a burok (ekkor nem készül gyermek processz).

A parancsnak, akár külön processzben fut, akár maga a burok hajtja végre,

- van visszatérési értéke!

  A visszatérési értéke lehet

  - normális (0) visszatérés,
  - nem normális (nem 0) visszatérés.

A visszatérési értéket a burok használhatja a vezérlés menetének szabályozására.

A burok processznek (ami a parancsot végrehajtja) van legalább három *nyitott adatfolyama*.

Leírójuk	Nevük	Szokásos leképzésük
0	stdin	billentyűzet
1	stdout	képernyő, ablak
2	stderr	képernyő, ablak

Láttuk, a parancsban *szavak* vannak. A ***szó*** az, amit *fehér karakter* határol. Idézőjelbe (" " ' ') tett szöveglánc (quótázott szöveglánc) csak egy szónak számít (az idézőjel semlegesíti a fehér karakterek szóhatároló funkcióját).

Ügyelni kell a speciális karakterekre! Ezek szerepe különleges! (Ilyenek a * $ [ ] { } \ . stb. karakterek).

Egy példa parancsra:

```
> find . -name a.c -print
   0  1     2   3    4
```

azaz, a fenti parancs 5 szóból áll.

A parancsokban a parancsértelmező *adatfolyamai* átirányíthatók. Ekkor a parancs – ha nem kellene is - szeparált processzben fut. Miért? Mert az indító shell processz szabványos adatfolyamainak leképzését nem változtatják (hibalehetőségekhez vezetne). Ilyenkor új processzt készítenek, és abban végzik az új leképzést.

Példa adatfolyam átirányításra:

```
> ls  >mylist.txt
```

Ebben a parancsban az átirányítás miatt az *ls* szeparált processzben fut. Az *ls* a burkokban rendszerint belső parancs, nem kellene neki feltétlenül szeparált processz. Kérdezhetnénk, milyen *program* fut ekkor a szeparált processzben? ls program nincs, hiszen az az sh/bash/tcsh/ksh burkok belső parancsa? Nos, a válasz: a gyermek processzben is a burok fut, ennél viszont a szabványos (standard) kimenet a *mylist.txt* fájlba van leképezve, és ez a gyermek burok processz fogja az *ls*-t végrehajtani!

3.5.1.2 A csővezeték (pipe) fogalma

A *csővezeték* parancsok sora | -vel (cső operátorral) szeparálva. A | a csővezeték operátor.

A csővezeték szintaxisa:

```
> parancs_bal  |  parancs_jobb
```

A szemantikája:

Végrehajtódik a *parancs_bal* és szabványos kimenete leképződik az utána végrehajtódó *parancs_jobb* szabványos kimenetére.

A csővezeték parancsai szeparált processzekben futnak Miért? Mert itt is szükséges a szabványos adatfolyamok leképzésének megváltoztatása!

A *csővezetéknek is van visszatérési értéke*: a parancs_jobb visszatérési értéke.

A parancs degenerált csővezeték. Ezentúl, ha valahol csővezetéket írunk, oda parancsot is írhatnánk.

Példa:

```
> cat /etc/passwd  |  grep  valaki
```

Az *cat* itt a számlaszámokat tartalmazó állományt teszi a szabványos kimenetére, a csővezetékbe. Ezt „megszűrjük” a *grep* mintakereső szűrővel, keresve a *valaki* mintát tartalmazó sort. A grep a csővezetékből olvas: arra képzi szabványos bemenetét.

3.5.1.3.A parancslista fogalma

A *parancslista* csővezetékek sora, szeparálva a következő operátorokkal:

```
   &&    ||     # magasabb precedencia
    &   ; \n    # alacsonyabb precedencia
```

A parancslista operátorainak precedenciája alacsonyabb, mint a csővezeték | -jé!

A *parancslista* szintaxisa:

```
> cső́bal  listaoperátor  cső́jobb
```

A szemantika:

;	soros végrehajtása a csöveknek
&	aszinkron végrehajtása a csőbal-nak (ez a háttérben fut, és azonnal indul a csőjobb is, vagy visszatér az indító shell)
||	csak akkor folytatja a listát, ha csőbal nem normális visszatérési értékű
&&	csak akkor folytatja a listát, ha csőbal normális visszatérési értékű

Először látjuk a visszatérési érték értelmét, az valóban megszabhatja a „vezérlés menetét”!

A csővezeték degenerált lista. Ezentúl, ha valahová parancslistát írunk, az lehet csővezeték, sőt parancs is!

A *parancslista* visszatérési értéke az utolsó csővezeték visszatérési értéke. Háttérben futó csővezeték visszatérési értéke külön kezelhető.

3.5.2. Parancs, cső, lista csoportosítás, zárójelezés

A csoportosítás, zárójelezés oka kettős lehet:

- az operátorok precedenciájának átértékelését akarjuk elérni;
- processz szeparálást akarunk elérni.

Lehetséges zárójelek: ( ) { }

A szintaxis:

```
{ lista }   vagy   ( lista )
```

3.5.2.1. Zárójelezés a precedencia átértékelés miatt

Emlékezzünk a csővezeték operátor és a listaoperátorok precedencia sorrendjére. Ezt a precedenciát tudjuk zárójelezéssel átértékeltetni.

Beláthatjuk, hogy az alábbi példákban eltérő eredményeket kapunk! Emlékeztetek arra, hogy a *date* parancs dátumot és időt ír a szabványos kimenetre, a *who* parancs a bejelentkezettek listáját teszi a kimenetre, a *wc* parancs pedig sor-, szó- és karakterszámláló.

Példa:

```
$
$ date ; who | wc        # mast ad ez
...
$ ( date ; who ) | wc    # mint ez
...
```

Házi feladatként magyarázzák meg, miért ad mást a két lista!

3.5.2.2. A processz szeparálás miatti zárójelezés

{ *lista* } zárójelezéssel csoportosított parancsnál, - hacsak más ok miatt (pl. átirányítás van, csővezeték van, külső parancsot kell végrehajtani) nem kell szeparált processzben végrehajtani - ugyanabban a processzben fut a lista.

(*lista*) zárójelezéssel a parancslista mindenképp szeparált processzben fut!

Megpróbálom megmagyarázni példákkal. A megértéshez érteni kellene a *processz környezet* (process environment) fogalmat, amit később részletezünk. Mindenesetre a környezethez tartozó információ a *munkajegyzék* (working directory). A pillanatnyi munkajegyzék lekérdezhető a *pwd* paranccsal, munkajegyzék váltható a *cd* paranccsal. Az *rm* parancs fájltörlésre való. A két példa ugyanabból a kiinduló helyzetből induljon; munkajegyzék a *vhol*, ebben be van jegyezve az *ide* jegyzék, utóbbiban van *junk* fájl.

1. példa:

```
$ pwd                    # hol vagyunk?
vhol
$ cd ide ; rm junk       # törli vhol/ide/junk-ot
$ pwd
vhol/ide                 # most ez a munkajegyzék
$
```

2. példa:

```
$ pwd
vhol                     # hol vagyunk?
$ ( cd ide ; rm junk )   # u.azt torli
$ pwd                    # mivel szeparalt processzben
vhol                     # futott, a cd csak
                         # „ideiglenes” volt.
```

3.5.3. A parancs végrehajtás

Általában az *sh burok* készít új processzt a parancs számára, ebbe betölti a parancshoz tartozó végrehajtható fájlt, átadja az argumentumokat az így készült processznek. Ez az általános szabály, ami alól vannak kivételek.

Nem készül új processz az ún.

- *belső parancsoknak* (special commands, built in commands),
- a *vezérlő parancsoknak* (*for, while, case* stb.),
- a *definiált függvényeknek* (sh makróknak),

de a kivételeknek is vannak kivételei:

- hacsak nem zárójeleztünk ( ) -vel,
- hacsak nincs átirányítás, csővezeték (> >> < << |) .

Biztos készül új processz a *külső parancsoknak*. Ezek lehetnek:

- V*égrehajtható* (compilált, linkelt executable) *fájlok* . (A burok ezeket a fork/exec villával indítja. Az argumentumok itt is átadódnak! Lásd a C-ben a main függvény argumentum-átvételét!)
- *Burok programok* (shell eljárások, shell szkriptek). A burok ekkor is a fork/exec villával indít szeparált processzt, ebbe burkot tölt és ennek adja a burokprogramot feldolgozásra.) (az argumentumok átadódnak!)

Mind a végrehajtható fájlok, mind a burokprogramok futtatására jellemző:

- PATH szerinti keresés,
- kell hozzájuk az x (executable) elérési mód,
- a burokprogramokra kell az r (readable) elérési mód is,
- a gyermek processz örökli a környezetet (environment, lásd később).

Külön érdekesség: vajon milyen végrehajtható fájl fut a szeparált processzben, ha belső parancsot indítunk, de kikényszerítjük (vagy kikényszerül), hogy mégis szeparált processzben fusson? Nos a válasz erre: akkor a gyermek processzben is a burok fut!

3.5.4. Az adatfolyamok átirányítása

Fontos szerepűek az 0/1/2 leírókkal azonosított szabványos adatfolyamok. Ahogy említettük, a parancsok általában az *stdin*-ről olvasnak, az *stdout*/*stderr*-re írnak.

Mielőtt a parancs végrehajtódik, a végrehajtó shell megnézi, van-e átirányítás a parancs sorában.

Ehhez a szavakban

```
<    >    <<-vmi   >>
```

átirányító operátorokat használhatjuk. Ha ilyen operátorokat talál a burok, akkor - szeparált processz(eke)t készítve, azokban leképezve az adatfolyamokat futtatja a parancsot.

Az átirányító operátorok szemantikája:

```
< file          # file legyen az stdin

> file          # file legyen az stdout (rewrite)

>> file         # file legyen az stdout (append)

<<[-]eddig      # here document: beagyazott input
```

A beágyazott inputnál a `-` elmaradhat, ezt jelzi a szintaxishoz nem tartozó [ ] zárójelpár.

Az átirányítás szintaxisát és szemantikáját lásd bővebben az on-line kézikönyv *sh* lapján! Az *append* hozzáfűzést, a *rewrite* újraírást jelent.

Legnehezebb megérteni a *beágyazott input* fogalmat. A burokprogramokban parancsokat, csöveket, listákat szoktunk írni, néha azonban jó lenne a feldolgozandó *adatokat* is oda írni.

Jelezni kellene azonban, hogy ezek nem parancsok, hanem feldolgozandó adatok. A végrehajtó burok ugyanarról a szabványos bemeneti csatornáról (a szkriptből) kell, hogy olvassa ezeket is, mint a parancsokat! Vagyis a bemeneti csatornát akarjuk leképezni magára a burokprogramra, annak a soron következő soraira. Persze, azt is kell jelezni, hogy meddig tartanak az adatok, hol kezdődnek újra a parancsok! Nos, ezt a problémát oldja meg a beágyazott input, adatok beágyazása a burokprogramba.

Egy kis példa a beágyazott inputra, ahol is létezik az *a.script* szövegfájl, (futtatható és olvasható,) a tartalma az alábbi:

```
a.script
--------------------------------
grep  ezt  <<!
elso sorban van ezt
2. sor, ebben nincs
3. sor
!
echo ' na mi van?'
--------------------------------
```

Így indíthatjuk, és az alábbi az eredmény:

```
$ a.script
elso sorban van ezt
 na mi van?
$
```

A fenti példában a ! (felkiáltójel) használatos az adatsorok végének jelzésére. Olyan karakterkombinációt válasszunk, ami nincs az adatsorok között, hiszen ez fogja jelezni, meddig tartottak az adatok, hol kezdődnek újra a parancsok.

3.5.5. Fájlnév kifejtés

A parancsok argumentumai gyakran fájlnevek. Ezekre van "behelyettesítési" lehetőség, alkalmazhatunk *dzsókereket*.

Parancsok argumentumaként, argumentumaiban használhatunk ún. fájlbehelyettesítési dzsóker *karaktereket*. Ilyenek a kérdőjel (?), a csillag (*), a szögletes zárójelek [ ].

Ha ezek előfordulnak a parancs szavaiban (általában ott, ahol a burok fájl nevet várna), akkor a *szót* (amiben szerepelnek), *minta*ként (pattern) veszi a burok! A *minta* a hívó shellben behelyettesítődik (kifejtődik)

alfabetikus sorrendű fájl nevek listájává, olyan nevekre, melyek illeszkednek a fájlnévtérben a mintára

A fájlnév-teret a hierarchikus fájlrendszer ösvénynevei alkotják, beleértve az abszolút és a relatív ösvényneveket is.

Az illeszkedés szabályaiból néhányat felsorolunk:

- A nem dzsóker karakterek önmagukra illeszkednek
- A ? bármely, egyetlen karakterre illeszkedik
- A * tetszőleges számú és tetszőleges karakterre illeszkedik
- A szögletes zárójelbe írt karaktersorozat [...] illeszkedik egy, valamelyik bezárt karakterre (a pontok helyére képzeljünk karaktereket.
- A [!.] illeszkedik bármely, kivéve a ! utáni karakterre.

További érdekes minta szintaktika van! Érdemes utánanézni!!

Példa:

Tegyük fel, az aktuális jegyzékben van 4 fájl, a nevük:

```
a   abc   abc.d   xyz
```

Ekkor (a `->` itt azt jelzi, mivé helyettesítődik az eredeti parancs):

```
$ ls *          -> ls a abc abc.d xyz
$ ls a*         -> ls a abc abc.d
$ ls [a]??      -> ls abc
$ ls [!a]??     -> ls xyz
```

Vegyük észre, hogy a fenti példa soraiban a fájlnév behelyettesítés megtörténik, és csak utána hívódik az *ls* parancs! Vagyis az sh burok nagyban különbözik az MS DOS parancsértelmezőjétől, bár ott is használhatók dzsókerek, de azokat a command.com nem helyettesíti be, hanem átadja a parancsnak, és az, ha tudja, majd behelyettesít.

Fájlnév behelyettesítés történik ott is, ahol tulajdonképpen nem fájlneveket várnánk, pl. az echo argumentumában! Ezért pl. az előző példa aktuális jegyzékét feltételezve a következő parancs

```
$ echo [a]??
abc
$
```

eredményt adja., miután az sh burok előbb fájlneve(ke)t helyettesít be, aztán hívja az *echo*-t.

3.5.6. A metakarakterek semlegesítése, az ún. quótázás

Láttuk a fájl-behelyettesítés dzsóker karaktereit, és tudjuk, hogy további metakarakterek is vannak (a fehér karakterek, a cső és lista operátorok, a változóbehelyettesítés operátora stb.).

```
      ;       &       ( )       ^       < >       $       space       |
stb.
```

Némelyiknek tudjuk a szerepét (pl. szeparátorok, operátorok), némelyiket később tanulhatjuk meg. Látni fogjuk, némelyiknek több szerepe is lehet.

Mindezeket a burok különlegesen kezeli (pl. fájlnév behelyettesítéshez mintaként a *-ot, a space karaktert szóelválasztóként stb..

Ha mégis szükségünk van rájuk: *semlegesítsük* (*quótázzuk) őket*!

- Egyetlen karakter quótázása
  `\spec_karakter`
- Több karakter quótázása:

```
'karaktersorozat' # Minden bezárt karakter quotázott, kivéve '
"karaktersorozat" # Ezen belül a változó/paraméter és
                  # parancsbehelyettesítés megtörténik
                  # (lásd később, most csak jegyezd meg!),
                  # de a fájlnév behelyettesítés nem!
                  # Ha mindenképp kell, a \ quotázással
                  # semlegesítsd a \ ` " $ karaktereket!
```

Példa (előlegezett a *burokváltozó* és a *változóbehelyettesítés* fogalma):

```
# a - sh változó, $a - kifejtése
$ a=abc        # értéket kap az a
$ echo  '$a'   # semlegesítve a $ kifejtő operátor
$a
$ echo  "$a"   # hatásos a $ operátor
abc
$ echo  "$\a"  # a \a-val az a karaktert értjük
$\a
$
```

Próbáljuk megérteni, megmagyarázni a fentieket!

3.6. Burokprogramozás

A *burok program* (shell szkript) szövegszerkesztővel készült fájl. Egy program, ami parancsokat tartalmaz soraiban (esetleg a beágyazott input szerkezetben adatsorokat is). A burok programot a parancsértelmező processz olvassa sorról-sorra, elemezi a sorokat és sorról-sorra végrehajtja/végrehajtatja a program parancsait.

A burokprogramozásnak meglehetősen szigorú a szintaktikus szabályai vannak.

Egy nagyon egyszerű példa, melyben az a.script szövegfájl 2 sort tartalmaz:

```
a.script
---------------
who  > kik
ps  >> kik
---------------
```

A szövegszerkesztővel készített burok-programot végrehajthatóvá és olvashatóvá kell tenni! Utóbbi az szövegszerkesztők (editorok) kimenetének alapértelmezése szokott lenni, előbbit explicite írjuk elő!

```
> chmod  +x  a.script
```

Ezután hívható:

```
> a.script      # magyarazd, mi tortenik
```

Figyelem! Az előadáson részletesebben tárgyaljuk, hogy milyen kivételes esetben elegendő a burokprogram csakis olvashatósági elérése.

Ha a *shell szkript* program, akkor

- vannak (egyszerű) adatszerkezetei (változók, konstansok: szöveglánc jellegűek, de néha numerikus adatként is kezelődnek)
- van (egyszerű) végrehajtási szerkezete (soros, elágazás, hurok);
- kommentározzuk (a # után a sor maradéka kommentár).

3.6.1. A shell változók

- van nevük,
- vehetnek fel értékeket (szövegeket),
- kifejthetők a pillanatnyi értékei.

3.6.2. A shell változók osztályai

3.6.2.1. Pozícionális változók (parancs argumentumok)

A nevük kötött, rendre a 0 1 2 ... 9

Rendre a parancssor 0., 1., 2. stb. aktuális argumentumát veszik fel értékként. A 0 nevű változó mindig a parancs nevét kapja, az 1-es nevű az első szót s.í.t. Annyi pozícionális változó definiálódik, amennyi argumentum van a parancs sorában, maximum persze 9. Ha több mint 9 aktuális argumentummal hívjuk a parancsot, a shift paranccsal a változók „eltolhatók”!

Példa:

```
> script  alfa  beta     # ez a szkript hivasa
```

Ekkor a script-en belül

```
0     ->     script     értékű
1     ->     alfa       értékű
2     ->     beta       értékű
3 – 9                   nincs definiálva.
```

3.6.2.2. Kulcsszós shell változók

a) Felhasználó által definiált kulcsszós változók

A felhasználó általi definícióval a felhasználó választja ki a változó nevét. A definiálás szintaxisa:

```
valtozo=string
```

Vigyázz!

```
valtozo  =  string    # nem jo! Miert?
                      # Mert a space szóelválasztó karakter!
```

A *string* lehet 0 hosszú is! Ekkor a változó ugyan definiált, de 0 hosszúságú.

b) A rendszergazda és a shell által definiált kulcsszós shell változók

b1) Rendszergazda által definiált változók

A rendszergazda által definiált változók rendszerint konvencionális nevűek. Szintén konvenció, hogy ezek nagybetűsek. Ilyenek pl. a

```
PATH
HOME
MAIL    stb., változók.
```

Valahol a rendszergazda "leírta" a definíciót és a változót „exportálta”, azaz láthatóságát kiterjesztette (rendszerint egy startup fájlban): `PATH=string ; export  PATH`

A rendszergazda által definiált változókat a segédprogramok, szűrők stb. használják, nevük ezek miatt konvencionális.

b2) A shell által definiált változók

Ezek neveit a shell programozók választották, nevük ezért kötött, konvencionális. Ilyen nevek pl. a `#`, a `*` (meglepő. a – shell változó is, nemcsak dzsóker!). A változókifejtés alfejezetben további shell által definiált változókat (és pillanatnyi értéküket is) adunk meg.

3.6.3. Hivatkozások shell változókra, kifejtésük

A legegyszerűbb hivatkozás, a parancs-sorba írt

```
$valtozonev
```

Itt a $ a kifejtő operátor. Jegyezzük meg, hogy a nem definiált, vagy 0 hosszú változó hibajelzés nélkül „kifejtődik”, természetesen 0 sztringgé. Később láthatjuk, hogy a definiálás hiánya, vagy a 0 sztring definíció „ellenőrizhető”.

Példákon bemutatunk néhány előre definiált változót (pozícionális változót, shell által definiáltat, rendszergazda által definiáltat) a kifejtésükkel:

```
$0   a parancsnév
$1   az első aktuális argumentum
$9   a kilencedik argumentum kifejtve
```

```
$*   minden definiált argumentum kifejtve
$#   a pozicionális paraméterek száma decimálisan
$?   az utolsó parancs exit státusa (visszatérési értéke)
$$   a processz azonosítója: a pid értéke
$!   az utolsó háttérben futó processz pid-je
$HOME  a bejelentkezési katalógus  stb.
```

A változóbehelyettesítés teljes szintaktikája, szemantikája

`${valt}` szerkezet is egyszerű behelyettesítés. A kapcsos zárójelek hozzátartoznak a szintaktikához. A {valt} bármikor használható, de csak akkor kell feltétlenül, ha az egyértelműséghez a változónév pontos elválasztása szükséges. Ha a változónév „folytatódik” szöveggel, akkor jelentkezhet az egyértelműsítési igény:

```
$ nagy=kis
$echo  ${nagy}kutya
kiskutya
$ echo $nagykutya


# Miert?  Mert nem definialt a nagykutya valtozo!
```

A változókifejtés teljes szabályrendszere:

Az alábbiakban a

valt shell változó
szo szövegkifejezés (pl. szöveg-konstans)
: A kettőspont (colon) önmaga, de elmaradhat.

${*valt*:-*szo*} Ha *valt* definiált és nem 0 string, akkor kifejtődik pillanatnyi értéke, különben kifejtődik a *szo*.

${*valt*:=*szo*} Ha *valt* nem definiált vagy 0 string, akkor felveszi a *szo*-t, különben nem veszi fel. Ezután kifejtődik a *valt*.

${*valt*:?*szo*} Ha a *valt* definiált és nem 0 string, akkor kifejtődik, különben kiíródik a *szo* és exitál a shell. A *szo* hiányozhat, ilyenkor default üzenet íródik ki.

${*valt*:+szo} Ha a *valt* definiált és nem 0 string, akkor behelyettesítődik a *szo* (nem a *valt*!), különben semmi sem fejtődik ki.

Ha a : (colon) hiányzik, csak az ellenőrződik le, vajon a definiált-e a *valt*.

3.6.4. Parancs behelyettesítés

A parancsbehelyettesítés szintaxisa: (vegyük észre, hogy a ` grave accent, más mint a '):

```
`parancs`
```

A szemantikája: végrehajtódik a *parancs*, és amit a szabványos kimenetre (stdout) írna, az oda, ahova a `parancs` szerkezetet írtuk, behelyettesítődik (kifejtődik). Használhatjuk a kifejtett füzért. burokváltozóhoz érték adásra, de más célra is.

Példa:

```
$ valt=`pwd`
$ echo $valt
/home/student/kovacs
$
```

Jegyezzük meg!

Minden adat füzér (string) jellegű. A füzérben lehetnek fehér karakterek is, ilyenkor quázi szavak vannak benne!

Példa:

```
$ szamharmas=`who | wc`
$ echo  $szamharmas
3 15 11
$
```

3.6.5. Változók érvényessége (scope-ja)

A processzeknek van *környezetük* (environment), amit megkülönböztetünk a *process context*-től. (A processz kontextus fogalmat az Operációs rendszerek tárgyban részletezzük.)

A környezet (environment) szerkezete, implementációja

A környezet a processz (itt a shell) kontextusához tartozó *szövegsorokból álló* tábla.

Egy sor ebben

```
 valt=string
```

alakú.

Mikor egy shell indul, végigolvassa a környezetét, és definálja magának azokat a változókat, melyeket a környezetben megtalál, olyan értékkel, amit ott talál. Ugyanennek a shellnek aztán további definíciók is adhatók: sőt, a környezetből az induláskor definiált változók átdefiniálhatók, meg is szüntethetők. Környezeti változó átdefiniálása nemcsak az aktuális shellnek, hanem a környezetnek is szól.

A környezet lekérdezhető a *set* paranccsal.

```
 $ set
...
```

A környezetbe tehető egy változó az export paranccsal. Ezzel tulajdonképpen a leszármazott processzekben (shellekben) is láthatóvá tesszük a változókat.

Szintaxis:

```
$ export valt
```

(Ezzel a technikával "öröklődik" a HOME, MAIL, PATH stb. Ki definiálta és exportálta ezeket? Az ülés létesítés (login) és az ülésben a burok (shell) indítás során végrehajtott *startup shell* programok!)

Kérdés merülhet fel: mi történik, ha még nem definiált változót exportálok? Vajon ekkor definiálttá válik? Válasz: nem! Ha (újra)definiálom, marad exportált? Válasz: igen!

Jegyezzük meg!

1. Exportálással csakis a gyermek (és unoka) processzek öröklik a változókat! A szülő processzek nem látják a gyermekei exportált változóit!

2. Nem exportált, de definiált változó a gyermek processzekben nem látható. Visszatérve arra a burokra, amiben definiálták: újra látható! Miért? Mert a szülő processz átélte a gyermekei életét.

Pozícionális paraméterek láthatósága

A pozicionális változók csak abban a burokban láthatók, ahová adódnak. A gyermek processzeknek új pozícionális paraméterek adódnak át.

3.7. Vezérlési szerkezetek a sh shellben

3.7.1. Szekvenciális szerkezet

Szekvenciális, legegyszerűbb vezérlési szerkezetek a parancslisták.

3.7.2. Elágazás: az if

Szintaxis (a [ ] szögletes zárójel itt nem része a szintaktikának, csak azt jelzi, hogy elmaradhat a bezárt rész):

```
 if list1 then
        list2
[elif list3 then
        list4]
[else list5]
fi
```

Szemantika:

Az if, elif predikátuma a *list1*, ill. *list3* visszatérési értéke. A predikátum igaz, ha a visszatérési érték 0, azaz normális. (Lám, láthatjuk már a visszatérési érték értelmét!)

Értelemszerűen: ha *list1* igaz, akkor végrehajtódik *list2*, különben ha *list3* igaz, akkor *list4*, máskülönben *list5*.

Vegyük észre az új neveket (kulcsszavakat: if, then, elif, else, fi). Külön érdekes a fi lezáró!

3.7.3. Elágazás: a case

Szintaxis:

```
 case  szo  in
        pattern1 ) list1 ;;
       [pattern2 ) list2 ;;]
       ...
esac
```

Ahol:

```
 * )                    akármi, default
pattern | pattern )     alternatíva
[patt patt] )           alternatíva]
stb.,
```

Vagyis a mintaképzés hasonló a fájl-név generáció mintaképzéséhez.

Az alternatívákra adok két példát:

```
 -x|-y          vagy -x, vagy -y
-[xy]           vagy -x, vagy -y
```

Figyelem! A [ ] a szintaxis harmadik sorában annak jele, hogy valamilyen szerkezet elmaradhat, a mintákban a [ ] viszont hozzátartozik a szintaktikához!

Vegyük észre az új neveket, az érdekes case lezárót! Vegyük észre a mintát lezáró) zárójelet és a listákat lezáró ;; jelpárt!

Az értelmezését legáltalánosabban a következő: kifejtődik a *szo* és összevetődik a mintákkal, olyan sorrendben, ahogy azok le vannak írva.. Ha egyezés van, végrehajtódik a mintához tartozó lista, és befejeződik a case. Egy példán keresztül bemutatom ezt. Képzeljük el, hogy van egy *append* nevű shell programunk:

```
 append
----------------------------------------
case  $#  in     # a # a poz. param. szama
       1 ) cat  >> $1 ;;
       2 ) cat  >> $2 ;;
       * ) echo 'usage: append from to'
esac
----------------------------------------
```

Ezt hívhatjuk:

```
 $ append
usage: append from to
$ append f1
...
...
CTRL/D
$ # elkeszult az f1
$ append f1 f2  # f2-hoz fuzodott az f1
```

3.7.4. Ciklus: a for

Szintaxis:

```
 for  valt  [in szolista...]
    do
       lista
    done
```

Ahol *szolista...*: szavak fehér karakterekkel elválasztott listája, ami el is maradhat (jelzi ezt a [ ] szintaktikához nem tartozó zárójelpár).

Hiányzó in *szolista* esetén a pozícionális paraméterek szólistája az alapértelmezés (ugyanaz, mintha in $*-t írtunk volna!).

Új neveket jegyezhetünk meg, köztük a do-done parancszárójel párt!

Szemantika: a *valt* rendre felveszi a *szolista* elemeit, és mindegyik értékkel végrehajtódik a do és done zárójelpár közötti *lista* (azaz annyiszor hajtódik végre a test, ahány eleme van a szólistának).

Példa:

```
 tel
-------------------------------------------------------
for i in $*
      do
        grep  Si ${HOME}/telnotes
      done
-------------------------------------------------------
```

Magyarázzák meg, mi történik, ha így hívom:

```
 $ tel  kiss nagy kovacs
```

3.7.5. Ciklus: a while

Szintaxis:

```
 while  lista1
```

```
do
   lista2
done
```

Szemantika: ha *lista1* exit státusa 0 (normális visszatérési értékű, ami azt jelent, igaz), akkor ismételten végrehajtja do done zárójelpárral közrezárt *lista2*-t, majd újra a *lista1* végrehajtása következik s.í.t.

3.7.6. Az if-hez, while-hoz jó dolog a test

A test parancs szemantikai igaz tesztelésre normális visszatérési értéket ad. Jól, értelemszerűen használható vezérlési szerkezetekben.

Kétféle szintaxisa van (lásd bővebben a kézikönyvben: man test). A másodikhoz szintaktikai formához kellenek a [ ] zárójelek! A

```
test  expression
```

az egyik, a

```
[  expression  ]
```

a másik szintaxis. Ügyeljenek a szóelválasztó helyközökre (space-ekre)!

Szemantika: 0-val (normálisan) tér vissza a test, ha az *expression* igaz. Tudjuk tehát az *if* és a *while* predikátumaként használni.

Az expression lehetőségek

I. csoport: fájlokkal kapcsolatos tesztelő kifejezések. Példák (nem teljes):

```
test  -f  file  # igaz, ha file letezik és "plain file"
                # (nem jegyzék, nem fifo stb.).
test  -r  file  # a file olvasható
test  -w  file  # a file írható
test  -d  file  # a file létezik és jegyzék, stb.
```

II. csoport: shell változók/adatszerkezetek relációi.

Ezt is csak példákkal mutatom be, tehát ez sem teljes! És a másik szintaktikát használom!

```
[ s1 ]             # igaz, ha s1 nem 0 sztring
[ $v  -gt  ertek ] # algebrai nagyobb v. egyenlo
[ $v  -eq  ertek ] # algebrailag egyenlő
[     -z  s1  ]    # az s1 0 hosszú
[     -n  s1  ]    # az s1 nem 0 hosszú
[ $v  =  ertek  ]  # fűzérként egyforma
```

3.7.7. További jó dolog: az expr parancs

Szintaxis:

```
expr  ertek  operator  ertek
```

Szemantika: Kiértékel és az eredményt az *stdout*-ra írja.

Az operátorokat lásd a *man expr*-ben. Mindenesetre: vannak algebrai operátorok, ekkor az *ertek*-ek numerikus stringek kell, hogy legyenek.

Példák:

1. példa:

```
$ expr 1 + 2
3
$
```

2. példa:

```
$ sum=0
$ sum=`expr $sum + 1`
$ echo $sum
1
$
```

3. példa:

```
bell
-------------------------------------------------------
n=${1-1}             # ha nincs arg, akkor is 1 legyen
while [ $n -gt 0 ]
do
        echo  '\07\c'   # quotazas, \c szerepe
        n=`expr $n - 1`
        sleep 1         # alszik 1 sec-ot
done
-------------------------------------------------------
```

Hívása:

```
> bell 3      # harmat sipol
```

3.7.8. A rekurzió lehetősége

A burok programok rekurzívan hívhatók. Legyen a HOME jegyzékünkben a dw burokprogram:

```
dw
```

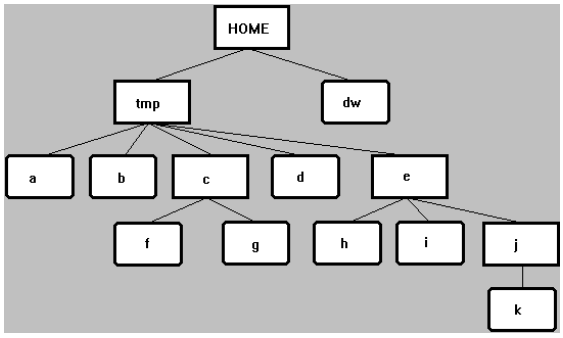
```
-----------------------------------------------------
cd  $1  ;  echo  $1
ls  -l
for  i  in  *
do  if  test  -f  $i
    then  :
    else  $HOME/dw  $i
    fi
done
-----------------------------------------------------
```

Ami itt új, az a : (kettőspont, colon) parancs, a ne csinálj semmit (*do-nothing)* parancs. A : parancsnak mindig 0 (normális, igaz) exit státusa van, bár itt ez nem érdekes.

Értermezzük, magyarázzuk, mit is csinál a dw burok-program! Bevallom, kisebb hibák vannak benne. Az egyik, induláskor nem ellenőrződik, vajon a $1 jegyzék-e (értelmes-e rá a cd parancs). Mielőtt a 6. sorban rekurzívan újra hívódik, már „ellenőrizzük", vajon a $i jegyzék-e. De ez az ellenőrzés a `test -f $i` szerkezettel történik, és itt lehet a másik hiba! A –f a "sima fájl" (plain file) létet ellenőrzi, nem azt, hogy jegyzék-e a fájl! (Utóbbit a test -d file paranccsal tehettük volna.) Sajnos, ha a $i nem sima fájl, még nem biztos, hogy jegyzék! Vannak ui. más típusú fájlok, pl. pipe-ok, speciális fájlok, amikre a rekurziónak már nem szabadna menni. Sebaj, legalább megtanultuk a *do-nothing* parancsot! Szóval, óvatosan ezzel a példával!

Mindenesetre elemezzük, hogy mit csinálna, ha a 3.1. ábrán látható hierarchikus faszerkezetű részfájl rendszer volna és így indítanám:

```
>  dw  tmp
```



3.1. ábra. Példaprogramhoz fájlrendszer

- Belép tmp-be, echózza, hosszú listát készít róla.
- Újrahívja dw c-vel;
  - belép c-be, echózza, listát készít róla;
  - visszalép.
- Újrahívja dw e-vel;
  - belép e-be, echózza, listát készít róla. Újrahívja dw j-vel;
    - belép j-be, echózza, listát készít róla;
    - visszalép.
  - visszalép.
- Visszalép.
- Kilép.

3.7.9. A read parancs

Kulcsszós változók definiálására az értékadáson kívül még jó a *read* parancs is

Szintaxis:

```
read val1 val2 val3 ...
```

Szemantika: beolvas egy sort az stdin-ről, és a első szót a *val1*-be, második szót a *val2*-be teszi. Ha kevesebb szó van a beolvasott sorban, mint ahány változónk van, akkor a „maradék" definiálatlan marad. Ha kevesebb a változók száma, akkor az utolsó változóba a sor maradéka kerül, akár többszavas sztringként is.

A visszatérési érték 0, hacsak *end-of file* nem jött.

Beállítható a szóelválasztó karakter (alapértelmezés: fehér karakterek), az IFS shell változó értékadásával.

Tapasztalatom szerint némely burokban a read parancshoz az stdin nem irányítható át.

3.7.10. Fontos tanácsok

1.) Régebben az iit tartomány felhasználói alapértelmezési interaktív burokként a *tcsh*-t használták. Ez interaktív használatra a z SGI Irix rendszernél igen jó volt. Ha a tcsh-nak adjuk az *sh* szintaxisú parancsokat, gondunk lehet. Javasoljuk áttérni az sh-ra (ez ugyan nem kényelmes interaktív használatra), vagy a bash-ra! Utóbbi felülről kompatibilis az sh-val, de sokkal kényelmesebb az interaktív használata! Szerencsére a *gold ksh*-ja (Korn shellje) felülről kompatibilis az *sh*-val. Szerencsére, a *segédprogramok* (*utility*), *szűrők* bármely shellből ugyanúgy (v. nagyon hasonlóan) hívhatók. Tulajdonképpen csak a shell programozási vezérlési szerkezetek és a változó definíciók a tcsh-ban!

2.) Általános szokás, hogy a shell szkripteket Bourne shellben írják, az *sh*-val dolgoztatják fel. Újabban ksh-s és bash-os szkriptek is előfordulnak.

Kérdés merülhet fel, hogyan szabályozzuk azt, hogy shell szkript futtatásnál mindenképpen az *sh* dolgozza fel a programot?

Megoldások:

- Az *sh* program első sorába tegyük be

```
#! /bin/sh
```

Ez ugyan kommentár a burok processznek, de a rendszer „kitalálja" ebből, hogy az *sh*-val kell feldolgoztatnia a burokprogramot. További lehetőségek:

- Az *sh* szkript első sora semmiképp ne legyen ettől eltérő kommentár. Ha ui. az első sor kommentár, de nem a fenti, akkor a hívó shell típusával azonos shellel fogja feldolgoztatni a kernel a szkriptet. Nem lesz baj, ha a hívó shell is *sh*, de lesz, ha az *tcsh* vagy *csh*.
- A szkript első sora egyáltalán ne legyen kommentár. Ekkor ui. bármilyen shellből hívták, mindenképp az *sh* dolgozza fel. Apró példaprogramokhoz javasolható ez a megoldás, de komoly programokat illik kommentározni, méghozzá fej-kommentárral kezdeni, nem a jó tehát.
- Hívjunk előbb interaktív *sh*-t, bash-ot, ekkor a programértelmezéshez is az *sh* vagy bash dolgozik.

Interaktív Bourne shellből *csh* szkriptek feldolgoztatásához éppen az javasolt, hogy az első sor a következő kommentár legyen:

```
#! /bin/csh
```

De ritka az az emberpéldány, aki *csh* szkripteket ír!

További tanácsok

A különböző shellek indulásakor különböző *startup file*-ok "kézből" végrehajtódnak. Céljuk a globális változók definiálása, egy-két adminisztratív feladat elvégzése. Ráadásul:

- Vannak közös *startup szkriptek*: ezeket a rendszergazda gondozza, írja, felügyeli.
- Vannak a HOME-ban *saját startup szkriptek*. Ezeket mi is módosíthatjuk.
- Egyes *szkriptek* csak a login során indulnak. De egy új terminálemuláció indítás transzparens login lehet, és ekkor ebben is elindul ez a típusú *shell szkript*).
- Más *szkriptek* mindenképp futnak, ha indul egy shell (pl. > sh-ra).

A startup file-ok összefoglalása:

Bourne shell:	/etc/profile	# közös
	$HOME/.profile	# saját
C, TC shell:	/etc/cshrc	# közös
	$HOME/.cshrc	# saját, minden csh-ra
	$HOME/.login	# saját, login-re
Korn shell:	/etc/profile	# közös
	$HOME/.profile	# saját, továbbá a
	$ENV	# saját, ami szokásosan a $HOME/.kshrc-re van definiálva.

Mi, hol található a Unix-ban?

Hol vannak a végrehajtható programok, a dokumentumok, a könyvtárak (nem a jegyzékek, hanem az ar fájlok!), a konfigurációs fájlok stb., ez a kérdés. Jó összefoglaló van a Bartók-Laufer könyv 72. oldalán a konvencionális helyekre! Érdemes ezt tanulmányozni, érdemes a rendszerünkben körülnézni!

A reguláris kifejezések

Az *awk, ed, grep, lex, sed* segédprogramok szövegfájlokat olvasva reguláris kifejezésekkel választanak ki sorokat feldolgozásra. A kiválasztás: minta illesztése a sorra. A minta lehet reguláris kifejezés. Különböztessük meg a fájlnév-behelyettesítés minta fogalmától! A fájlnév-behelyettesítési minta az *sh*-nak szól, a reguláris kifejezés a segédprogramoknak (ezért, ha olyan dzsóker van a reguláris kifejezésben, ami fájlnév behelyettesítő karakter is egyben, quótázni kell, nehogy a burok kifejtse azt!)

A *minta* (pattern): füzérek halmaza (set of strings). A halmaz definíció az alkalmazástól függ. A *minta* valamire illeszkedhet (match). Ha illeszkedik, azt kiválasztottuk.

c a c látható karakter önmagára illeszkedik;
\c itt a c quótázott karakter, önmagára illeszkedik;
. a dot bármely nem új sor karakterre illeszkedik;
[...] füzér bezárva [ ]-be illeszkedik egyetlen karakterre, ami a füzérben van; [^...] negálás;
[a-c] illeszkedik egyetlen, a bezárt tartományba eső karakterre;
[^a-c] nem illeszkedik a tartomány karaktereire;
e* illeszkedik 0, vagy több,
e+ 1, vagy több,
e- 0, vagy 1 előfordulására e füzérnek;
e1e2 két összefűzött reguláris kifejezés illeszkedik az elsőre, majd a másodikra.
^|$ illeszkedik a sor kezdetére|végére.

Csakis az *awk*, *lex* és *grep* esetén továbbá:

e1|e2 akár e1-re, akár e2-re illeszkedik;
(...) illeszkedik a bezárt reguláris kifejezésre.

További alapfogalmak:

sor/rekord	line
mező/szó	field
mezőelválasztó karakter	field separator: fehér karakter, : (colon) stb lehet.

3.8. Az awk mintakereső és feldolgozó

Az awk-t szűrőként szoktuk használni. Alapgondolata: szövegfájl sorokat olvas, minden sorban keres *mintákat* és a mintákhoz tartozó *akciókat* végrehajtja.

A szintaxisa:

```
awk  [-Fc]  [program]  [parameterek]  [file-lista]

    szóelválasztó                        input, ha nincs
      kijelölés                             az stdin
```

A szemantika:

Beolvassa az input sorait. A sorok szavait (szóelválasztó karakter alapértelmezésben a fehér karakterek egyike, megadható a -F opcióval, vagy a FS belső változó értékadásával beállítható) az *awk* 1, 2, 3 stb. nevű *mezőkbe* teszi (ezekre a programban lehet hivatkozni). Minden sorra nézi a *programban* megfogalmazott *minták* (reguláris kifejezés) illeszkedését, és amelyik *minta* illeszkedik a sorra, az ahhoz tartozó *akciót* végrehajtja. Vagyis, a program

```
minta {akciok}
minta {akciok}
...
```

formájú.

A program megjelenhet literális programként:

```
'program'
```

vagy egy fájlban:

```
-f  filename
```

A *paraméterek* segítségével további adatokat vihetünk az *awk*-ba (nézz utána!).

Akkor most egy példa, a *program* literális (egy soros és a minta hiányzik belőle, ami azt jelenti, minden sorra illeszkedik), *szóelválasztó* kijelölés nincs, *paraméterek* nincsenek, *input* az stdin:

```
> who | awk '{print $3,$4,$5,$1,$2}'
```

A példában a *who* kimenetét szűrjük az *awk*-val, a who kimenetének minden sorában a szavakat egy másik sorrendben íratjuk ki.

A program lehet több elemű, minden elem

```
minta {akciok}
```

formájú.

A programban az *akciók*: C-szerű utasítások, utasításblokkok. A fenti példában a print az egyetlen utasítás, hogy C-szerű-e, vagy sem, döntse el az olvasó. Mindenesetre, használhatók az akciók programrészben a C-szerű

if (*feltétel*) *utasítás* [else *utasítás*];
while (*feltétel*) *utasítás;*
for (*kif1; felt_kif2; kif3*) *utasítás;*
break, continue ;
{*utasítás; utasítás*} # összetett utasítás
printf *formátum, kifejezéslista;*

Ezeken kívül vannak jellegzetes *awk* utasítások is

print *kifejezéslista;*
for (*name* in *array*) *utasítás;*
next; # vedd a következő minta {akció} elemet;
exit; # exitálj

stb.

Végül változó definíciós utasítások is lehetségesek;

name értékadó-operátor kifejezés;
name[*kifejezés*] *értékadó-operátor kifejezés;*

A következőkben látunk majd példákat a programok *akciós* részére, ebben mező- és változó hivatkozásokra stb.

Érdekes a programok *minta* része is. Ezek tehát *reguláris kifejezések*, vagy *awk kifejezések* lehetnek. Legegyszerűbb az üres minta: ez minden sorra illeszkedik, a hozzá tartozó akció minden sorra végrehajtódik.

```
{print "Minden sorrra kiirni!"}
```

A következő programban a minta illeszkedik, ha a sorban valahol megtalálható a valami szó:

```
/valami/ {print "Megtalaltam valami-t."}
```

Ha valamely sorban a második mező a valami, akkor ezt jelzi az alábbi program:

```
$2=="valami"{print "Megtalaltam valamit a masodik mezoben"}
```

Vagyis a mintában a szokásos relációkkal is kapcsolhatunk mezőhivatkozásokat, awk belső változó (lásd ezeket később) hivatkozásokat is.

NF > 5 illeszkedik olyan sorokra, melyek mezőszáma nagyobb 5-nél.

$1~/valami|VALAMI/ illeszkedik azokra a sorokra, melyek első szava kis- vagy nagybetűs valami.

$1~/[Ss]treet/ illeszkedik azokra a sorokra, melyek első szava street, vagy Street.

Lehetséges tartományokat is kijelölni! Az alábbi minta illeszkedik a begin és az end szavakat tartalmazó sorok közötti sortartományra, azaz minden sorra a begin-t tartalmazó sortól az end-et tartalmazó sorig végrehajtódik az akció:

```
/begin/,/end/ {akcio}
```

Akkor most néhány további példát, melyek a /etc/*passwd* fájl sorait dolgozzák fel. Ez a fájl mindenki által olvasható, sorokból áll, a sorokban : (comma) elválasztóval hét mező található, rendre: login-név, titkos-jelszó, uid, gid, teljes-név, home-dir és induló-program. (NIS-es rendszerben az ypcat passwd parancs ilyen sorokat generál.)

1. példa:

```
> ypcat passwd | awk -F: '$4==105 {print NR,$4,$1}'
```

Kiírja a 105-ös csoporthoz tartozó számlaszámok sorszámár (NR), csoportszámát ($4) és bejelentkezési nevét ($1). A program literális, a minta a $4==105 formájú.

NR az awk belső változója, felveszi az éppen feldolgozott sor sorszámát.

2. példa, írjuk ki a passwd fájl 5. sorát!

```
> awk 'NR==5 {print $0} /etc/passwd
```

Ami új: a $0 nem mezőre, hanem az egész sorra vonatkozik. Literális a program. Van input, ez az /etc/passwd fájl.

3. példa, minden név (1-es mező) és gid (4-es mező) kiírandó:

```
> awk '{print  $1, $4}' /etc/passwd
```

4. példa, a gid, a név és az induló shell formázva írandó ki:

```
> awk -F: '{printf "%8s %4s %s \n",$1,$4,$7}' /etc/passwd
```

Láthatjuk, a printf hasonlít a C printf-hez (Nézz utána!)

Különleges minták: BEGIN és END

Különleges minta a BEGIN és az END. Ezek nem *utasítás zárójelek*, hanem *minták*:

A BEGIN illeszkedik minden sor előtt, az END a sorok feldolgozása után. Lehet tehát a BEGIN-nel inicializálni, az END-del a feldolgozás végén összegezni.

Az alábbi példákon mindjárt bemutatjuk használatukat, de a példákhoz bemutatjuk az awk további belső változóját, az FS-t (Field Separator). Az FS a mezőelválasztó karakter alapértelmezés szerint fehér karakter, beállítható a -Fc opcióval, illetve az awk programban FS=c értékadással is.

5. példa, számláljuk az input sorait. (Bár ez egyszerűbben is megoldható, most így csináljuk!) A példa bemutatja azt is, hogy a program lehet egy fájlban is:

Legyen a *prog* fájl tartalma:

```
-----------------------------------
BEGIN { s=0}
      { s = s + 1}
END   { print "osszeg: ", s}
-----------------------------------
```

Illetve, legyen *prog1* tartalma:

```
-----------------------------------
BEGIN {FS=":";  s = 0}
      { if ($4 == 105) s = s + 1 }
END   { printf "Osszeg: ", s}
-----------------------------------
```

Akkor

```
> ypcat passwd | awk -f prog
 osszeg: ddd
```

kiadja a számlaszámok számát, míg a

```
> ypcat passwd | awk -f prog1
Osszeg: nnn
```

kiadja a 105-ös csoportba tartozók számát.

A programokhoz még: látható az s = 0 értékadás, ami egyben egy belső változó definíciója is. Mindkét program három *minta {akció}* szerkezetet tartalmaz.

A számlaszámok számának egyszerűbb kiíratása:

```
> ypcat passwd | awk 'END { print NR}'
```

Mielőtt további példákat adnánk, foglaljuk össze az awk legfontosabb **beépített változóit:**

FILENAME	az aktuális input fájl neve, akár meg is változtatható;
FS	mezőelválasztó;
NF	mezőszám egy sorban;
NR	a pillanatnyi sorszám;
RS	input sorelválasztó (default: újsor-karakter);
OFMT	output formátum (default: %g);
OFS	output mezőelválasztó (default: szóköz);
ORS	output sorelválasztó (default: újsor)
stb.	

Az awk operátorok (csökkenő precedencia):

++, --	pre/postfix inkrementáció, dekrementáció;
*, /, %	multiplikatív operátorok;
+, -	additív operátorok;
	karakterlánc összekapcsolás (semmi operátor);
<,<=,>,>=,==, !=, ~, !~	relációs operátorok, ~ az egyezés, !~ a nem egyezés operátora;
!	kifejezés értékének tagadása;
&&	és;
||	vagy;
=, +=, -=, *=, /=, %=	értékadó operátorok.

Az awk-nak van néhány **beépített függvénye** is:

sqrt, log, exp, int matematikai függvények.

length(s)	szöveghosszt adja vissza.
substr(s,m,n)	substring s-ből, m-től, n hosszan.
index(s,t)	s füzérben t első előfordulásának indexe.

Tömbök az awk-ban

Lehetséges az alábbi tömbdefiníció:

```
tomb_name[konst_kifejezes]  ertekado_oerator  kifejezes;
```

6. példa:

prog2

```
---------------------------------
    { line[NR] = $0 }
END { for (i=NR; i > 0; i--)
     print line[i]
    }
---------------------------------
```

Ez a program két komponensű. Első komponensében nincs minta, minden sorra illeszkedik tehát. Az akció részében tömbdefiníció van: line[NR] = $0 formában, szöveglánc elemeket tartalmazó *társtömböket* definiál, amiknek indexe a sorszám!

Hogy hogyan helyezi el őket, az nem érdekes. Mindenesetre lehet minden definiált elemére később hivatkozni!

Mit csinál ez a program? Fordított sorrendben kiírja a sorokat! Vigyázzunk, sok-sok sorból álló inputra veszélyes elereszteni, mert elfogy a memória!

A társtömbök indexei és az awk speciális for-ja

Társtömb index akármilyen konstans kifejezés lehet.

Használható a for (*name* in *tömbname*) ciklus is.

A 7. példán mutatom be a két új koncepciót. A példához tételezzük fel, van egy szövegfájlunk (ez lesz a feldolgozandó input), amiben név-érték párok vannak. A nevek ismétlődhetnek:

```
szovegfile
--------------------------
joe             400
mary            200
joe             200
john            300
susie   500
mary            200
-------------------------
```

Összegezzük az egyes nevekhez tartozó összegeket. A *prog* fájl:

```
prog
-----------------------------------
    { sum[$1] += $2 }
END { for (name in sum)
      print name, sum[name]
    }
-----------------------------------
```

Így hívjuk:

```
> awk -f prog  szovegfile
...
```

Az awk rendre társtömböket definiál

```
sum[joe]
sum[mary]
sum[john]
sum[susie]
```

nevekkel, ezekben összegzi az értékeket.

Figyeljük meg a for ciklust! A for és az in kulcsszó, a *name* általunk választott változónév.

És egy utolsó példa, szószámlálás. Vigyázzunk erre is, sok szóból álló szövegre veszélyes elereszteni. Csak a programot adom meg:

```
    { for (i=1; i<= NF, i++) num[$i]++}
END { for (word in num)
      print word,num[word]
    }
```

Meg tudják magyarázni?