Shell vagy héjprogramozás

Szabó László Zsolt

Shell vagy héjprogramozás

Szabó László Zsolt

Megjelent: 2011

Lektorálta: Budai László, Informatikai tanácsadó, Acceleris Gmbh

A tananyag a TÁMOP-4.1.2-08/1/A-2009-0046 számú Kelet-magyarországi Informatika Tananyag Tárház projekt keretében készült.

A tananyagfejlesztés az Európai Unió támogatásával és az Európai Szociális Alap társfinanszírozásával valósult meg.

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu 06 40 638-638

Verziótörténet
Verzió 1.1 2012 November 8.
Korrekciók
Verzió 1.0 2011 Október 19.
TAMOP Kelet-magyarországi Informatika Tananyag Tárház első változat

Tartalom

A táblázatok listája

1. fejezet - Bevezető

A shell vagy héjprogramozás hátteréről

Ez a tananyag a UNIX alapú operációs rendszerek egyik fontos komponensének, a shellnek vagy héjnak az alapszintű használatát mutatja be.

Nem célunk a UNIX alapú rendszerek meglehetősen bonyolult történetét bemutatni, azonban néhány részletet meg kell említenünk, hogy utaljunk a héjak jelenlegi használati lehetőségeire.

A UNIX operációs rendszert az AT&T, távközlésben működő cég Bell nevű laboratóriumában kezdték tervezni, 1969-1972 között, tervezői Ken Thompson, Dennis Ritchie, Doug McIlroy voltak (UNIX1970). A cél többfelhasználós multitaszking rendszer fejlesztése volt, amelyik valós idejű funkcionalitásokkal is rendelkezik.

Ezzel egy időben alakul ki a C nyelv standard könyvtára, és a klasszikus operációs rendszer függvények sora, így kialakulása összefonódik a C nyelv tervezésével (Dennis Ritchie és Brian Kernighan). A UNIX az első operációs rendszer amelynek kódja igen kis résztől eltekintve C-ben íródott. Első változatát PDP számítógépeken fejlesztették, ekkor alakult ki a klasszikus fájlrendszer struktúrája.

Az 1970-80-as évek jelentették a UNIX felhasználásának kezdeti időszakát. Mivel az AT&T nem forgalmazta a rendszert mint operációs rendszert önállóan, tervezési elvei és forráskódja sok fejlesztő laborba eljutott, elsősorban egyetemekre. Így több helyen is folytattak kiegészítő tervezői-fejlesztői tevékenységet. Több fontos fejlesztési vonulat alakult ki, ezek közül a legfontosabbak a System V illetve a Berkeley-i fejlesztésekre épülő BSD rendszerek. A UNIX szerzői joga átkerült más cégekhez, és egy meglehetősen bonyolult történetté alakult.

Jelenleg sok cég fejleszt saját UNIX-ot amelyek licencelt fejlesztések és az eredeti UNIX kódra épülnek: IBM, Hewlett-Packard, SCO, Novell, Oracle (a Sun Microsystems fejlesztését vette meg). Ezek vállalati kiszolgálók piacára fejlesztett rendszerek. Az Internet, telefonhálózatok stb. sok vonatkozásban UNIX-okon épültek ki, így ezek mai fejlettségéhez a UNIX-on használt elvek lényegesen hozzájárultak. A UNIX használatának egyik nagy előnyét kiváló hálózati és valós idejű szolgáltatásai jelentik.

A sok UNIX változat következményeként több standardizálási folyamat alakult ki (1984, X/Open Portability Guide: X/Open Company; 1988-2004, POSIX (Portable Operating System Interface): IEEE 1003-asnak is nevezett standard; 1995, Open Group: Single UNIX Specification). Ezek a standardok a UNIX programozási interfészeire illetve parancsnyelvére határoznak meg közös irányonalakat. A UNIX nevet, amely védett név (tulajdonosa jelenleg az Open Group nevű konzorcium) általában azokra a rendszerekre alkalmazzuk, amelyek követik az eredeti UNIX rendszerek felépítését, és meg is felelnek bizonyos szabványoknak (mint az Open Group Single UNIX Specification szabványa).

Unix alapú vagy Unix-szerű rendszereknek nevezzük azokat az operációs rendszereket, amelyek úgy viselkednek mintha UNIX rendszerek lennének. Ez az elnevezés vitatható, de célunknak megfelel. Ezek a rendszerek nem valósítják meg szükségszerűen minden vonatkozásban a UNIX standardokat. Ide tartozik több nyílt forráskódú operációs rendszer is,

mint a Linux kernelre épülő rendszerek vagy a Free BSD. Ezek a rendszerek a 90-es években lettek népszerűek meg különböző nyílt forráskódú fejlesztések eredményeként.

A Linux egy UNIX-hoz hasonlóan viselkedő rendszer ("Unix-like", Unix-szerű rendszer, vagy klón): egész pontosan csak a rendszermag, a kernel klónja. A klón fogalma azt jelenti, hogy nem tartalmaz UNIX kódot, de más szoftver megoldásokat használva megvalósítja ugyanazt, mint egy eredeti UNIX kernel és hasonlóan is használható. Eredeti alkotója Linus Torvalds.

A Linux kernel köré fel lehet építeni egy operációs rendszert, ami Unix-szerűen viselkedik, ha a UNIX-hoz hasonlóan klón felhasználói programokkal vesszük körül. Egy ilyen összeállítást, ahol jelen van a Linux kernel és egy csomag felhasználói illetve rendszer program, Linux disztribúciónak nevezzük.

A shell vagy héj tulajdonképpen egy parancsértelmező, a felhasználó által parancssori szöveggel leírt parancsokat hajtja végre, így interfészt biztosít a felhasználó és az operációs rendszer között. Ebben az értelemben parancsnak nevezünk bármilyen futtatható programot amelyik része egy Unix alapú operációs rendszernek.

Az első időkben a héj (1970-es évek) egyszerűbb parancs nyelvet biztosított az akkoriban kialakuló UNIX rendszerek számára. Az évek haladásával ez a nyelv egyre bonyolultabb lett, több változata alakult ki, ezek saját fejlődési útjaikat járták, ettől kezdve már programozási nyelvi lehetőségeket is biztosítottak, ezért talán megengedhető az is, hogy a héjat programozási nyelvnek is nevezzük.

A UNIX parancsértelmező angol neve *shell*, a magyar nyelvű dokumentumokban ez helyenként *burok*, helyenként *héj* megnevezéssel fordul elő, a mindennapi nyelvben tapasztalatunk szerint gyakrabban használjuk magyarul is a shell megnevezést. Így ebben a tananyagban a héj és shell megnevezés egyaránt előfordul. A parancsértelmező neve arra utal, hogy burokként veszi körül a UNIX operációs rendszer legfontosabb komponensét, a rendszer magját vagy kernelt, és ennek szolgáltatásaihoz csak a parancsértelmezőn keresztül lehet hozzáférni, amikor egy felhasználó használni akarja a rendszert.

A héj változatai

A Unix alapú rendszerek történelmi fejlődése folyamán több héjat terveztek, a legtöbb esetben megtartva az eredeti rendszerben használt héj elveit és kiegészítették különböző, a programozást könnyítő szerkezetekkel.

A Bourne shell volt az első parancsértelmező amelyiket kimondottan arra fejlesztették, hogy szkript nyelv is legyen egyben. 1977-ben kezdték fejleszteni az AT&T-nél (első fejlesztő Stephen Bourne). Ezzel vezettek be sok új lehetőséget először, mint a parancssor helyettesítés. A későbbi években is az AT&T-nél fejlesztették. Jelenlegi Linux disztribúciókon **sh** program név alatt található meg, és legtöbbször nem külön programként, hanem szimbólikus linkként, amely a rendszeren található Bash-re mutat (ez emulálja a Bourne shellt).

A héjak történetében egyik erős hatást kifejtő fejlesztés A Korn héj. Ezt a változatot ugyancsak az AT&T-nél fejlesztették (David Korn), és a legfontosabb fejlesztési cél a shell programozási nyelvvé való átalakítása volt. Visszafelé kompatibilis maradt a Bourne héjjal. Sok új lehetőséget vezettek be, így az asszociatív tömböket, az egész és valós számokkal való új számítási szerkezeteket. A POSIX standard "Shell Language Standard" nevű előírásait pontosan betartja,

ezért gyakran ezt használják ipari standard héjként a UNIX variánsokban. Eleinte az AT&T tulajdonában volt a fejlesztése, 2000-ben nyílt forráskódúvá tették és jelenleg Common Public Licence licenc alatt fejlesztik (lásd http://www.kornshell.com/). Annak ellenére, hogy a mindennapi munkában sokan ennek modernebb változatait vagy a Bash-t használjuk, UNIX karbantartó vagy installáló szkriptekben legtöbben ma is a Bourne vagy a Korn változatait futtatják.

A C héjat (C shell) a Berkeley rendszerek fejlesztés során fejlesztették ki (fejlesztője Bill Joy, a későbbi Sun Microsystems alapító). Amint nevéből kiderül, szerkezetei hasonlítanak a C-re. Ebben fejlesztették tovább a feladatok (jobs) felügyeletét biztosító szerkezeteket. Nem annyira használt mint a Bourne-ra épülő változatok. Jelenleg **tcsh** nevű modern változatát használják, és leginkább a BSD alapú rendszerekkel.

A Bash (Bourne Again Shell) változatot eleve nyílt forráskódúnak kezdték fejleszteni, a cél a GNU operációs rendszer héjának fejlesztése volt, így lett a Linux kernelt használó rendszerek alapértelmezett héja. A kompatibilitás szempontjából Bourne és Korn héjat követi, és a legtöbb ezekre írt szkript lefut módosítás nélkül a Bash-en is. Mindhárom előzőleg említett shell-ből vett át megoldásokat, de fejlesztettek sok saját nyelvi lehetőségeket is. A Bash első fejlesztője Brian Fox volt, jelenlegi karbantartója és fejlesztője Chet Ramey.

Modern Linux-okon általában ez az alapértelmezett shell, és a leggazdagabb programozási lehetőségeket is ez nyújtja.

A felsoroltakon kívül több más fejlesztés is létezik, mint pl. az igen népszerű Z-Shell.

A héjak közti különbségek tárgyalása meghaladja ennek a tananyagnak a kereteit. Így feltételezzük, hogy a diák valamilyen Linux disztribúcióval dolgozik, és a Bash héj legalább 3.0 verzióját használja. A tananyagban nagyon kis kivétellel klasszikus szerkezeteket mutatunk be, amelyek az említett három héjból származnak és működnek a Bash-ben.

A standard UNIX rendszereken és a Linuxon kívül más Unix alapú operációs rendszerek esetében is használható a Bash, mint a BSD rendszereken vagy az Apple gépeken futó OS X alatt (Darwin).

Windows rendszereken is használható, amennyiben telepítjük a Cygwin eszközöket (http://www.cygwin.com), bár kezdeti tanuláshoz nem ajánljuk ezt, mert ilyen környezetben különbségek adódnak a használatában.

A héj legfontosabb feladatai: Unix parancsok indítása, összefűzése, felügyelete. Így a tananyag részben a fontosabb Unix parancsok bemutatásával is foglalkozik. Mivel nem tehetjük meg ezt kimerítően, a parancsok leggyakoribb használati módját és opcióit mutatjuk be. Ezek kimerítő tanulmányozásához a parancsok kézikönyv (vagy **man**) lapjaihoz, esetenként **info** lapjaihoz kell fordulni.

A parancsok használatának esetében is feltételezzük a Linux disztribúciók használatát. UNIX illetve más Unix szerű rendszerek felhasználói esetenként más vagy másként használható opciókkal találkozhatnak a saját rendszereiken.

A héj használata számtalan irányba ágazik, a tananyag ebben a vonatkozásban a leggyakrabban használt sémákat mutatja be. További és folyamatos tanuláshoz a Bibliográfiában [184]

említett könyveket, illetve Interneten a Bash Reference Manual (BashRef2010 [184]) és Advanced Bash-Scripting Guide (Mendel2011 [184]) munkákat, valamint magyar nyelven Büki András: UNIX/Linux héjprogramozás című könyvét (Buki2002 [184]) ajánljuk.

A tananyag szerkezete

A tananyag folyamatosan vezeti be a héj szerkezeteit.

A második fejezet a Unix terminál használatához ad néhány támpontot, a héjhoz kötődő fogalmaink itt még nem egészen pontosak, a fejezet célja bevezetni az olvasót a Unix parancsok gyakorlati használatához szükséges részletekbe.

A harmadik fejezet a Unix fájlrendszeréhez kötődő fontosabb fogalmakat és gyakran használt fájlkezelő parancsokat ismerteti.

A negyedik fejezet a Unix parancssor kezelés és parancs végrehajtás fogalmaiba nyújt bevezetőt.

Az ötödik fejezetben a héj változóival és azok egyszerű használatával foglalkozunk.

A hatodik fejezet a héj vezérlő szerkezeteit tárgyalja.

Mivel a reguláris kifejezések sok Unix parancs alapvető használatában jelen vannak, a hetedik fejezetet ennek a kérdéskörnek szánjuk.

A nyolcadik fejezetben további szerkezeteket mutatunk be, amelyek nem egy bizonyos témakörbe illeszkednek, de igen hasznosak a héj használata folyamán. Így ez a fejezet vegyes tematikájú.

A kilencedik és tizedik fejezet a Unix két ismert és gyakran használt parancsának, a **sed**-nek és **awk**-nak ismertetésére szánjuk.

A tizenegyedik fejezetben a Unix folyamatok kezelésével foglalkozunk

A tananyagot első illetve másod éves műszaki informatika vagy informatika szakos hallgatóknak szánjuk, egy bevezető UNIX/Linux tananyag egyik részeként, így feltételezzük, hogy az olvasó ismeri például a C nyelvet. Ettől függetlenül használható az anyag más kontextusban is.

2. fejezet - A UNIX terminál használata

Történelemi háttér

Jelenleg bármely Unix vagy Linux rendszer grafikus, ablakkezelő interfésszel is rendelkezik, ha egy személyi számítógépen vagy munkaállomáson használjuk. A legtöbb PC-re telepített Linux rendszer ezzel indul el. Az ablakkezelő rendszerben viszonylag könnyű eligazodni - már ami a programok egyszerű elindítását illeti. A héjhoz kötődő munkát viszont általában egyszerű szöveggel leírt parancsokkal végezzük, ezeket pedig parancssoron gépeljük be. Így elegendő egy jóval egyszerűbb interfészt használni mint a grafikusat. Amennyiben grafikus interfész van előttünk, akkor ennek az interfésznek az elérésére egy kis programot indítunk el, amely általában egy terminál emuláló program. A program egyszerű beviteli eszközt utánoz, amelybe csak szövegsorokat lehet begépelni, illetve válaszként szövegsorokat ír ki. Ez a parancs beviteli mód a UNIX rendszerek használatának történelméhez kötődik, de ma is aktuális és hatékony.

A 70-es évek UNIX-ához ember-gép interfészként egy külső hardver eszköz csatlakozott, amelyet *teletype*-nek neveztek (ez távirati rendszerekben használt terminál utóda volt, így lett a rendszerben a terminált csatlakoztató speciális fájlok neve tty). A kommunikáció egy soros kommunikációs vonalon keresztül zajlott, a bemenet írógépszerű billentyűzet volt, a gép által visszaírt szöveget papírra nyomtatta (ezért zajlott a kommunikáció mindkét irányban szövegsorokon keresztül). Később (1978 körül) képernyős terminálok jelentek meg (CRT - Cathode Ray Tube terminálok). Legismertebbek a DEC cég VT sorozata, pl. a VT100. Ezeken a kurzort mozgatni lehetett, ezt a kiíró szoftver vezérlő, un. escape szekvenciákkal tette. Az első időszakban ezek nem tudtak grafikus képet megjeleníteni. A szöveget 80 oszlop, 23 vagy 25 sor méretben tudták kiírni. Utánuk a grafikus terminálok következtek, ezek egy programot futtattak (X server) amely a UNIX-ot futtató gép grafikus alkalmazásai jelenítette meg. Ezek voltak az X terminálok.

Később személyi számítógépeken a hagyományos terminált egy program jelenítette meg a grafikus képernyőn, bemeneti eszközként a gép billentyűzetét, kimenetként pedig a képernyőt használták. Ezek a rendszerhez egy programon keresztül csatlakoztak, ugyanis itt már nem volt szükség egy külső hardver kapcsolatra. Például a UNIX rendszerekben használt X Window grafikus felület esetében ez a program (X terminál) kirajzolt egy terminált, ami úgy nézett ki, mint egy terminál képernyő. A géptől jövő információ (karakterek és kurzorvezérlés továbbra is úgy érkezett mint a CRT terminálóknál, de a szoftver átalakította grafikus képpé). Mivel a program pont úgy viselkedett, mint pl. egy igazi VT-100 terminál, azt mondjuk, hogy a szoftver a VT-100 terminált emulálta.

Így pl. a PC-k esetében a parancssor beütése ugyanúgy történik, mintha egy külső eszközt használnánk terminálként. A terminált emuláló programnak ezúttal nem volt szüksége egy konkrét hardware eszközre (pl. a soros port), hanem az operációs rendszer nyitott a számára 2 fájlt (pontosabban karakter eszközt) amin keresztül a kommunikáció zajlott. Ezért ezt az eszközt pseudo terminálnak nevezik.

A terminállal való munkát segítő billentyűk

A terminál billentyűjén beütött karakterek eljutnak egy programhoz, az pedig a karakterekre adott választ visszaírja a terminál képernyőjére. A programmal való kapcsolat két irányú, ezt *full duplex* működési módnak nevezzük. Ha a billentyűn leütött karaktert a vezérelt program fogadta, visszaírja a terminál képernyőjére – ekkor az megjelenik, ezt nevezzük visszhangnak (*echo*).

Az alábbi képernyőkimenet parancs beviteleket és a rendszer által kiírt válaszokat tartalmaz:

```
$ ls
a.out  echoarg.c  echo.c
$ ls -l -t
total 16
-rwxrwxr-x 1 lszabo lszabo 4847 Feb  7 10:44 a.out
-rw-rw-r-- 1 lszabo lszabo  198 Feb  7 10:43 echoarg.c
-rw-rw-r-- 1 lszabo lszabo  232 Feb  7 10:43 echo.c
$ ls -l echo.c
-rw-rw-r-- 1 lszabo lszabo  232 Feb  7 10:43 echo.c
$
```

A kimeneten a **$** karakter és az utána következő szóköz a héj készenléti jele (*prompt*), az **ls [23]** a begépelt parancs (ez fájlok nevét és tulajdonságát listázza), amelyet egymás után háromszor hívunk meg: első alkalommal opciók [11] nélkül, második alkalommal a `-l` és `-t` opcióval (ami részletes listát és módosítás szerinti lista sorrendet kér,a harmadik alkalommal csak az `echo.c` nevű programot listázzuk részletesen. Látható, hogy a héjjal való kommunikáció folyamatos gépelést jelent, ezért maga a Unix terminál kezelő alrendszere, illetve a használt programok is biztosítanak olyan szerkesztési műveleteket amelyek felgyorsítják ezeknek a soroknak a gyors gépelését. Ezek a műveletek általában kontroll karakterekhez vannak rendelve.

Készenléti jel vagy prompt

A könyv terminál ablak példáiban készenléti jelként a `$` jelet és az utána következő szóközt fogjuk használni, amely csak akkor vehető észre ha van valami a parancsoron, üres parancssoron nem látható:

```
$
```

Esetenként ez a különböző rendszereken más és más karaktersor lehet, általában egy praktikusabb üzenet, például a munkakönyvtár neve. Ezzel kapcsolatban lásd a PS1 [51] környezeti változót.

A terminál beállításait az **stty** paranccsal lehet megnézni illetve módosítani. Pl. a `-a` opciója kilistázza a terminál beállításait, ennek a kontroll billentyűkre vonatkozó része így néz ki:

```
$ stty -a
```

```
speed 38400 baud; rows 38; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z;
rprnt = ^R;werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
$
```

Az alábbiakban a kontroll és valamilyen más billentyű, pl. a C egyidejű lenyomását Ctrl-C -vel, illetve az Alt és C egyidejű lenyomását Alt-C -vel fogjuk jelölni.

A paraméterek terminál jellemzőket tartalmaznak, témánk szempontjából az első néhány sor tartalmazza azokat a kontroll karakter - terminál művelet hozzárendeléseket, amelyek a kernel terminál alrendszere kezel. Például a Ctrl-C karakterre a terminál az `intr` (*interrupt*) műveletet hajtja végre, azaz megszakítja a terminálon elindított, futó programot (pontosabban jelzést küld, lásd a Jelzések [159] című alfejezetet). Az alábbi táblázat tartalmazza a fontosabb műveleteket:

2.1. táblázat - A parancssor szerkesztési billentyűi

Művelet neve	Billentyű (kód)	Mit végez a terminál
intr	Ctrl-C	interrupt: a megszakító billentyű, a futó programot szakítja meg
quit	Ctrl-\	szintén megállítja a futó programot, az un. QUIT jelzéssel (lásd jelzéseket tárgyaló fejezet)
eof	Ctrl-D	fájl vége jel
erase	Ctrl-H vagy BS	a törlésre használt karakter (a modern terminálokon backspace)
werase	Ctrl-W	egy szót töröl (word erase)
kill	Ctrl-U	egy teljes parancssort töröl
suspend	Ctrl-Z	felfüggeszti az éppen futó folyamatot (a folyamat ideiglenesen leáll: a felfüggesztett folyamat fogalmát lásd később)
stop	Ctrl-S	megállítja a terminálra való írást
start	Ctrl-Q	újraindítja az írást
clear	Ctrl-L	törli a terminál képernyőjét, vagy más alkalmazások esetében újrarajzolja a képernyőt

Az **stty** segítségével ezek a hozzárendelések módosíthatóak, pl.:

```
$ stty intr Ctrl-X
```

megváltoztatja a megszakító kombinációt, Ctrl-C helyett Ctrl-X-re.

A fenti kombinációk minimális szerkesztési lehetőséget jelentenek, ezért a futó programok ennél jobb szerkesztési lehetőségeket építhetnek be a parancssor kezelésébe. A Bash héj a GNU Readline nevű sor szerkesztést biztosító könyvtárát használja, ez elegáns szerkesztési opciókat biztosít.

Az alábbi táblázat a legfontosabbakat tartalmazza a szerkesztési parancsok közül.

2.2. táblázat - A Readline szerkesztési műveletei a parancssoron

Billentyű (kód)	Szerkesztési művelet
Ctrl-B	back - balra lép egy karaktert
Ctrl-F	forward - jobbra egy karaktert
DEL vagy BS	a kurzor bal oldalán levő karaktert törli
Ctrl-A	a sor elejére ugrik
Ctrl-E	end - a sor végére ugrik
Alt-F	egy szót ugrik előre
Alt-B	egy szót ugrik vissza
Ctrl-U	törli a teljes sort
Ctrl-K	a kurzortól a sor végéig töröl előre
Ctrl-W	a kurzortól visszafelé töröl az első szóközig (gyakorlatilag egy szót töröl)
Ctrl-Y	visszamásol (yank): a törölt szöveget visszailleszti a kurzor pozíciójától (kivéve a teljes sor törlését)
Ctrl-T	kicseréli a kurzor alatti és az előtte levő karaktereket (ha rossz sorrendben gépeltük őket)

Amennyiben a terminálra hosszabb szöveget írunk és az kifut a rendelkezésre álló sorokból, az alábbi parancsokkal forgathatjuk illetve törölhetjük a szöveget:

2.3. táblázat - Képernyőt kezelő parancsok

Billentyű (kód)	Szerkesztési művelet
Shift-PgUp	lapozza a képernyő tartalmát felfelé
Shift-PgDown	lapozza a képernyő tartalmát lefelé
Ctrl-L	törli a képernyőt

A szerkesztési opciókon kívül, a gépelés felgyorsítható, ha a parancssort fogadó program rendelkezik szavakat kiegészítő opcióval (*word completion*). Szavakon a parancsnyelv esetében karakterek sorozatát fogjuk érteni. Ez azt jelenti, hogy a begépelt szavak kezdőbetűi után a Readline felismeri, hogy mit akarunk beírni, és kiegészítheti a szavakat. A Readline a parancsneveket, fájlneveket és változó neveket is képes kiegészíteni. Az időtrabló általában a fájlnevek beírása, próbálkozzunk először ennek gyakorlásával. A szavak kiegészítésére a tabulátor karaktert használjuk (Tab billentyű).

Ha a fájlnév első betűje (pl. legyen ez **a**) után egy tabulátor karaktert ütünk le , akkor a Readline kiegészíti automatikusan a fájl nevét - amennyiben egy olyan fájl vagy könyvtár van a munka könyvtárban amely a betűvel kezdődik.

Ha nincs ilyen fájl, a terminál egy csipogó hangot hallat (legelábbis általában így van beállítva, de természetesen a csipogás kikapcsolható).

Ha több ilyen fájl van, akkor ismét csipog, ilyenkor egy második tabulátor leütésével ki lehet íratni ezeket a fájlneveket, és folytatni lehet a parancssort a fájl második, harmadik, stb. betűjével, addig amíg a sorozat egyedi lesz: utána ismét leütve a Tab billentyűt kiegészül a név. A Tab leütésekor, amennyiben karakter egyezés van a nevek elején, a kiegészítés a parancssoron megtörténik addig a karaktermintáig amely közös a nevek első részében.

Ugyanez alkalmazható a parancsok nevére illetve héj változók [44] nevére is (a szavak parancssori helyének kontextusától függően fogja ezeket kiegészíteni).

Pl. amennyiben az alábbi parancsot akarjuk leírni, és az előző példában használt két fájl van a könyvtárban (`echoarg.c` és `echo.c`):

```
$ gcc echoarg.c
```

1. leírjuk: `gcc e`
2. lenyomjuk a Tab billentyűt, az eredmény: `gcc echo`
3. leütjük kétszer a Tab-ot, a shell kiírja a parancssor alá: `echoarg.c echo.c` , így látható, mivel lehet folytatni
4. beírunk még egy karaktert a névből `gcc echoa` és ismét leütjük a Tab-ot
5. a shell behelyettesíti a teljes nevet: `gcc echoarg.c` .

A fenti példában nem volt látványos a nyereség, de vannak 30-40 karakteres fájlneveink, amikor szinte lehetetlen a kiegészítő opció nélkül dolgozni.

Mozgás a parancssorok listájában

A héjjal való munka gyakran feltételezi parancssorok ismétlését, gyakran kis módosításokat végezve egy előzőleg már használt soron.

A shell rögzíti a leütött parancsokat, a pontos mechanizmus változik a különböző héjaknál (ha beírjuk a **history** parancsot, ez láthatóvá válik). A bash esetében az előzőleg leütött parancsok a legkönnyebben szintén a Readline lehetőségek segítségével kereshetőek vissza. Az alábbi billentyű kombinációkkal tesszük ezt:

2.4. táblázat - A parancssor ismétlésére használt billentyűkódok

Billentyű (kód)	Művelet
Ctrl-P vagy ↑	kiválassza az előző parancssort
Ctrl-N vagy ↓	kiválassza a következő parancssort
Ctrl-R	elindítja a keresést visszafele az eddig leütött sorok között

A parancssorok közti keresés során egy bizonyos sztringet tartalmazó parancssort lehet gyorsan visszakeresni. A keresést a Ctrl-R kombináció leütésével kell kezdeni. Utána, ahogy a keresett sztringet gépeljük megjelenik visszafele kereséssel az első olyan sor amelyben megtalálható.

Az alábbi példában a Ctrl-R lenyomása után egy 2-est írunk be, ami előhozza a utolsó, 2-es karaktert tartalmazó sort:

```
$ ls 2.txt
2.txt
$ ls 1.txt
1.txt
(reverse-i-search)`2': ls 2.txt
```

Minél több karaktert gépelünk be a keresett sztringből, a találat annál pontosabb lesz, és dinamikusan változok a gépelt karakterek hatására. Ha az első találat nem megfelelő, és további gépeléssel nem lehet szűkíteni, egy második Ctrl-R lenyomásával a második találatra lehet ugrani, és ezt lehet ismételni, amíg előkerül a keresett sor. Ekkor a keresést az Enter (Ctrl-J) vagy ESC billentyűk egyikével lehet leállítani. A keresett sor felkerül a parancssorra, meg lehet szerkeszteni és ismét el lehet indítani.

A munkaszesszió

A bejelentkezés egy asztali PC-n található rendszer esetében, amelyen el van indítva a grafikus felhasználói környezet egy grafikus ablak által megvalósított név/jelszó űrlap kitöltéséből áll. Amennyiben a felhasználót azonosítja a rendszer, elindítja mindazokat a szolgáltatásokat amelyre a felhasználónak szüksége van, többek közt a grafikus felületet. Linux rendszereken a leggyakoribb a Gnome vagy KDE (K Desktop Environment) elnevezésű grafikus felület.

Amennyiben a gépen nincs elindítva grafikus környezet, akkor a felhasználói interfész gyakorlatilag egy egyszerű szöveges terminál interfész. Asztali Linux rendszeren könnyen előhívhatunk egy ilyent, még akkor is ha a rendszer elindítja a grafikus interfészt. A Ctrl-Alt-F1, Ctrl-Alt-F2, ...,Ctrl-Alt-F6 billentyű kombinációk hatására egy virtuális terminált hív meg a rendszer, a gép billentyűzetet és és képernyőjét használva szöveges módban. Linuxon ezt a rendszerben elindított **mingetty** nevű folyamatok biztosítják, amelyek indításkor automatikusan indulnak (a grafikus felületet futtató terminálhoz általában a Ctrl-Alt-F7 kombinációval jutunk vissza).

A bejelentkezés ebben az esetben is név/jelszó pár, és ami ilyenkor történik az a klasszikus Unix belépés a rendszerbe (ugyanezt fogjuk tapasztalni, ha egy már megnyitott terminálról belépünk hálózaton keresztül egy távoli gépre). Linux rendszeren a felhasználó nevet átvevő **mingetty** folyamat egy **login** nevű programot indít el: ez bekéri a jelszót. Ha sikerült azonosítania a felhasználót elindítja a héjat, ezt a példányt *login shell*-nek nevezzük. A héjnak parancsokat írhatunk be, majd ha munkánkat befejeztük akkor a **logout** vagy **exit** parancsokkal léphetünk ki.

A belépés után a rendszer elindítja számunkra a héjat, amelyik a következőket végzi:

1. Beolvas néhány konfigurációs fájlt és végrehajtja azokat.

2. Kiír egy un. készenléti jelet, szöveget (*prompt string* vagy *prompter*) (ez általában a gép neve, a felhasználó neve vagy munkakönyvtár), a szöveg végén egy karaktert: ez egy `$`, `#` vagy > jel. Ebben a tananyagban egy $ és egy utána következő szóköz a készenléti jel.

3. Mindezek után parancsokat vár tőlünk, mindaddig amíg beütjük az **exit** vagy **logout** parancsokat vagy a fájl vége (Ctrl-D) karaktert.

A parancsok formátuma

Egy parancs szintaxisa a Unix rendszereknél:

parancsnév [opciók] [argumentumok]

A parancs név valamilyen futtatható fájl neve, az opciók (vagy más néven kapcsolók, *switch*-ek) egy adott parancs lefutási módját befolyásolják. Az opciók és argumentumok is lehetnek "opcionálisak": nem kell feltétlenül megadni őket ahhoz, hogy végre tudjuk hajtani a parancsot. Ezt jelzi a `[]` zárójel a parancsok szintaxisában. Opciót kétféleképpen adhatunk meg, rövid (egy - jel az opció előtt, ilyenkor az opciót egy karakter jelöli) illetve hosszú formában (két - jel az opció előtt, az opció egy szó). Az első esetben a kapcsoló általában egy karakter, a másodikban egy egész szó. Az **ls** esetében a `-l` opció részletes listát kér, a `--color` pedig a színes listázást tiltja le:

```
$ ls -l
total 16
-rwxrwxr-x 1 lszabo lszabo 4847 Feb  7 16:34 a.out
-rw-rw-r-- 1 lszabo lszabo  198 Feb  7 10:43 echoarg.c
-rw-rw-r-- 1 lszabo lszabo  232 Feb  7 10:43 echo.c
$ ls --color=no
a.out  echoarg.c  echo.c
$
```

Több opciót megadhatunk csoportosítva is:

```
$ ls -1tr
echo.c
echoarg.c
a.out
$
```

A parancs nevekben, opciókban a kis-nagy betűk számítanak, a Unix parancsai általában kisbetűsek. Néhány példa az ls parancs használatára különböző kapcsolókkal (a # karakter a shell számára a megjegyzés kezdetét jelenti, a '*$* ' pedig az alábbi esteben a shell készníléti jele.

```
$ #hosszú lista
$ ls -l
total 8
-rw-r--r-- 1 lszabo lszabo  0 Sep 19 14:26 1.txt
-rw-r--r-- 1 lszabo lszabo  8 Sep 19 14:26 2.txt
-rw-r--r-- 1 lszabo lszabo 54 Sep 19 14:27 3.txt
$ #időrendben listáz, legelől a legutóbb létrehozott fájl
$ ls -t
3.txt  2.txt  1.txt
$ #fordított sorrendben listáz
$ ls -rt
1.txt  2.txt  3.txt
$ #fordított sorrendben, időrendben és részletesen
$ ls -ltr
```

```
-rw-r--r-- 1 lszabo lszabo  0 Sep 19 14:26 1.txt
-rw-r--r-- 1 lszabo lszabo  8 Sep 19 14:26 2.txt
-rw-r--r-- 1 lszabo lszabo 54 Sep 19 14:27 3.txt
$ #ezt így is be lehet ütni:
$ ls -l -t -r
-rw-r--r-- 1 lszabo lszabo  0 Sep 19 14:26 1.txt
-rw-r--r-- 1 lszabo lszabo  8 Sep 19 14:26 2.txt
-rw-r--r-- 1 lszabo lszabo 54 Sep 19 14:27 3.txt
$ #a file1 és file2 tulajdonságait listázza
$ ls -l 1.txt 2.txt
-rw-r--r-- 1 lszabo lszabo 0 Sep 19 14:26 1.txt
-rw-r--r-- 1 lszabo lszabo 8 Sep 19 14:26 2.txt
$
```

Ha egy parancs szövege túl hosszú a terminál parancssorán, az folytatható a következő sorban.

Példaként vegyük az **echo** parancsot, amelyik a parancssori argumentumait írja ki:

```
$ echo ez egy szöveg
ez egy szöveg
$
```

A szöveg folytatását egy második sorban meg lehet oldani úgy, hogy a szövegben idézőjelet nyitunk és leütjük a RETURN billentyűt, így az a következő sorban lesz folytatható. Ott zárhatjuk az idézőjelet, és egy újabb RETURN-el indíthatjuk a parancsot:

```
$ echo "ez egy hosszú szöveg első része
> ez pedig a második"
ez egy hosszú szöveg első része
ez pedig a második
$
```

Látható, hogy a `$` helyett egy másik készenléti jel, a > jelenik meg ilyenkor.

Ennél fontosabb, a sorvégi vissza-per jel használata, amelyikkel bármilyen sort le lehet zárni, és folytatni a következőben (nem idézett szöveget):

```
$ echo ez egy hosszú szöveg első része \
> ez pedig a második
ez egy hosszú szöveg első része ez pedig a második
$
```

Látható a különbség is: az első esetben az újsor bekerül a parancssorba, a másodikban pedig nem, a módszer arra jó, hogy egyszerűen folytassuk a sort.

Ezt a megoldást héjprogramokban is alkalmazhatjuk, bármikor. Ennek az anyagnak a példáiban is többször alkalmazásra kerül azért, hogy a hosszú sor ne nyúljon túl az oldal szélességén. Természetesen munka közben a terminálnak lehet hosszabb sorokat is beírni mint az ablak szélessége.

Fontos: a sor végét lezáró \ után egyetlen karakter sem jöhet, csak az újsor.

3. fejezet - A fájlrendszer

Bevezető

Az állományok vagy fájlok névvel elérhető adathalmazok a számítógép tárolóin. Szervezésüket az operációs rendszer oldja meg, és a héjprogramozás számára az egyik fontos "adattípust" is jelentik. A héjprogramok egyik gyakori feladata a fájlok kezelése (fájlokat mozgatnak el, törölnek, keresnek, stb.), ezért a Unix rendszerek fájl szervezését ismerni fontos.

Aki tisztában van a Unix rendszerek fájlrendszerének struktúrájával és a leggyakrabban használt fájlkezelő parancsokkal, átlapozhatja a fejezetet.

A Unix egy általános fájl fogalommal dolgozik, amelyet egyszerű 8 bites karakter (byte) folyamnak tekint (*byte stream*). Az operációs rendszer szintjén nincs különösebb jelentése a fájlok tartalmának. A rendszer egyik szellemes megoldása, hogy nem csak a különböző adathordozókon található adatokat tekinti fájlnak, hanem a hardver eszközöket is. Ebben a fejezetben elsősorban a szabályos, adathordozókon található fájlokkal foglalkozunk.

A fájlnevek

A fájl nevek karakterekből állnak, hosszuk rendszerenként változik, de számíthatunk arra, hogy egy modern rendszeren akár 255 karakter hosszúak is lehetnek. Bármilyen karaktert tartalmazhatnak kivéve `/` és `\0` (a C programozásban a sztring végi null) karaktereket. Ennek ellenére fájl nevekben nem ajánlatos speciális karaktereket használni, mert az ilyenek zavarhatják a feldolgozásukat. Az ajánlott karakterek a betűkön és számokon kívül a `_` , - és a `.` . Kis és nagybetűk közti különbség számít, tehát a `naplo.txt` és `Naplo.txt` nevek nem ugyanarra a fájlra mutatnak.

A UNIX fájlrendszer felépítése, fontos könyvtárak

A könyvtár (*directory*) is egy fájl, amely könyvtári bemeneteket tartalmaz. Ezek adják meg a könyvtár alá szervezett fájlok, illetve más könyvtárak listáját és elérhetőségét.

A Unix fájlrendszere hierarchikus, fa szerű struktúra, amely első megközelítésben két szerkezetből, a könyvtárakból illetve fájlokból épül fel (amint azt látni fogjuk, tulajdonképpen a könyvtárak is fájlok, és a tartalmazott fájlok több típusúak lehetnek). A könyvtárszerkezet egy gyökérnek nevezett könyvtárral indul amelynek a neve egy karakter, a per jel: `/` (a gyökér könyvtár angol neve: *root directory*).

A gyökér alatt az alábbiakat láthatjuk:

- fa-szerűen szervezve könyvtárak, illetve fájlok találhatóak
- ez a struktúra megismétlődhet minden könyvtár esetében

- minden könyvtárban található 2 speciális könyvtár, a `.` és a `..` (az első önmagára, a második közvetlenül a felső szintre mutató könyvtár).

A Linux **tree** parancsa a fájlrendszer első szintjét így jeleníti meg (valamennyi megjelenő név könyvtár, az `-L` opció a listázott könyvtár szintet adja meg):

```
$ tree  -L 1 /
/
|-- bin
|-- boot
|-- data
|-- dev
|-- etc
|-- home
|-- lib
|-- lost+found
|-- mediac
|-- mnt
|-- proc
|-- root
|-- sbin
|-- selinux
|-- sys
|-- tmp
|-- usr
`-- var
$
```

Legfelül a gyökér könyvtár található, nyilván ez nem tartalmaz .. mutatót. Abban az esetben, ha mégis használná valaki, itt alapértelmezetten a `..` is a `/` -re mutat, tehát: `/../bin` vagy `/../../bin` az ugyanazt a `/bin`-t jelöl.

A könyvtárrendszer első szintjét úgy határozták meg, hogy jellegzetességeik szerint csoportosították a fájlokat, így adódtak az általánosan használt első szintű könyvtárnevek. Így a felhasználó gyorsan megtalál bármilyen fájlt, mert nagy valószínűséggel tudja, melyik csoportban található. Ez a felosztás nem teljesen egyforma a különböző UNIX rendszerek illetve Linux disztribúciók esetében, de a felosztási elvek nagyjából ugyanazok (Az alábbi példák a Red Hat cég Fedora disztribúciójában találhatóak így).

3.1. táblázat - A fájlrendszer fontosabb könyvtárai

Könyvtár neve	Mit tartalmaz
`/boot`	A rendszer indításához szükséges fájlokat tartalmazza, itt van a kernelt tartalmazó fájl is.
`/home`	A felhasználók saját könyvtárai, az un. "home" könyvtárak vannak itt. Például: `/home/lszabo` .
`/dev`	Az eszközökre mutató fájlok a `/dev` könyvtárban találhatóak.
`/bin`	Bináris, futtatható fájlok helye. Itt vannak a parancsok.

Könyvtár neve	Mit tartalmaz
`/sbin`	Csak a superuser (root) által végrehajtható fájlok.
`/usr`	A felhasználók és rendszer által használt fontos fájlok. A /usr alatt találhatóak például: `/usr/include` C header fájlok, `/usr/bin` futtatható fájlok, `/usr/src` forrás fájlok , `/usr/lib` könyvtárak.
`/tmp`	Mindenki által írható ideiglenes fájlok helye.
`/var`	Terjeszkedő, rendszer által írt fájlok (pl. naplózó fájlok).
`/proc`	Ez egy kernel által létrehozott virtuális fájlrendszer amely a futó folyamatokat és egyéb rendszerparamétereket írja le.

A Unix rendszerek használata mindig felhasználói jogosultságokhoz kötődik. A felhasználók egy bejelentkezési nevet kapnak, a bejelentkezés a rendszerbe általában jelszóval történik. Amikor a rendszerrel dolgozunk, az alábbi felhasználóhoz kötődő helyek adottak a fájlrendszerben:

- A felhasználó saját, vagy home könyvtára, pl: `/home/lszabo` .
- A munkakönyvtár (*working directory*): minden egyes futó programnak (vagy folyamatnak) van egy munkakönyvtára. Implicit ebből olvas fájlokat, illetve ebbe ír. Ehhez a könyvtárhoz viszonyítjuk a relatív elérési utakat (lásd alább). A munkakönytár váltása a **cd** paranccsal történik.
- a `.` és `..` könyvtárak amelyek az aktuális és a fölötte levő könyvtárat címzik

A programozásnál a fájlneveket kezelni a `basename` (egy abszolut elérési út fájlnevét írja ki) és a `dirname` (egy abszolut elérési út könyvtárnevét írja ki) segít:

```
$ basename /usr/include/readline/keymaps.h
keymaps.h
$ dirname /usr/include/readline/keymaps.h
/usr/include/readline
```

A fájl elnevezési konvenciók a UNIX alatt nem kötelezőek, bármilyen nevet és kiterjesztést adhatunk a különböző tartalmú fájloknak. Pl. bármilyen nevű vagy végződésű fájl lehet futtatható. Elnevezési konvenció azonban létezik, és többnyire használjuk is, hogy különbséget tegyünk a fájlok közt (gyakran használt végződések: .c, .o, .h, .txt, .gz, .ps, .tgz, .tar, .sh, .tex, stb.).

A UNIX könyvtárakat létrehozó parancsa a **mkdir** [24], könyvtárat törlő parancsa a **rmdir** [25]. A könyvtár fa struktúrájában a munkakönyvtárat a cd paranccsal lehet váltani (rövid bemutatásukat lásd a fejezet végén).Egy egyszerű könyvtárstruktúrát a **tree** paranccsal írhatunk ki.

Közönséges fájlok

A közönséges fájlok többnyire mágneses lemezeken vagy más adathordozókon találhatóak. Megkülönböztetünk szöveges ("szemmel" is olvasható, *text*) illetve bináris fájlokat.

Információt a fájlokról az **ls** [23] (listázás), **file** (típus kiíratás) és **stat** (fájl rendszer adatai) nyerünk.

A szöveges fájlok igen fontosak a Unix rendszerek alatt, a legtöbb program szöveges formában tartja konfigurációs adatait, valamennyi Unix parancs szövegeket ír ki a kimenetre és szövegeket olvas a bemeneten. A szöveges fájlok a használt karakterkészlet olvasható karaktereiből épülnek fel. A karakterek szövegsorokat alkotnak melyeknek végén egy újsor karakter található (ASCII kódja 10).

A fájlok lehetnek 0 byte hosszúak (üres file) is. Maximális hosszukat egy előjeles egész számmal tárolt fájlhossz korlátozza. Régebbi rendszereken ez 32 bites előjeles szám, így a maximális hossz kb. 2 GB. Modern rendszerek képesek a fájlhosszat 64 bites számban tárolni, ez elméletileg 8 milliárd Gigabyte-os fájlokat enged meg.

A hozzáférési jogok

Minden felhasználónak (*user*) bejelentkezési neve van (*login name*) és egy csoporthoz (*group*) tartozik. A felhasználó jelszóval jelentkezik be. A felhasználók illetve csoportok neveit a rendszer a nyilvános password fájlban, illetve a csoportokat tartalmazó fájlokban tartja (`/etc/passwd`, `/etc/group`).

Modern rendszerekre való bejelentkezés történhet valamilyen hálózaton keresztüli azonosítással, amikor a felhasználót azonosító adatokat nem az a gép tárolja amelyre bejelentkezünk.

A rendszerben egyetlen felhasználónak van jogosultsága bármilyen műveletet elvégezni, ezt a felhasználót a Unix rendszerek **root**-nak nevezik (ő a rendszergazda vagy *superuser*). Az többi felhasználó jogai korlátozottak, és általában a tulajdonában álló objektumok (pl. fájlok) vonatkozásában korlátlanok (ezekkel bármit tehet), a más felhasználó tulajdonában álló objektumok vonatkozásában korlátozottak.

A felhasználókat a rendszer a felhasználó neve és jelszó szerint azonosítja, a rendszerben viszont egész számokkal tartja nyilván (`user id` vagy `uid`, `group id` vagy `gid`). A bejelentkezett felhasználó az **id** paranccsal listázhatja ezeket ki (illetve megtalálja őket a `passwd` fájlban):

```
$ id
uid=500(lszabo) gid=500(lszabo) groups=500(lszabo)
$
```

A fájlokat a felhasználó által futtatott programok hozzák létre. A Unix alapú rendszerekben kétféle tulajdonjog létezik a fájlokon: felhasználói (*user*) és csoport (*group*). Így minden fájl valamelyik felhasználó illetve valamelyik csoport tulajdona. Létrehozásnál egy új fájl mindig a felhasználó és annak csoportja tulajdonaként jön létre, egy implicit elérési joggal (lásd alább). Ezek a jogosultságok a **chmod** [25] paranccsal változtathatóak, a fájlhoz rendelt (tulajdonos) felhasználó és csoport pedig a **chown** és **chgrp** parancsokkal.

A fájlok nevének és tulajdonságainak listázására az **ls** [23] parancsot használjuk.

A **chmod** parancs négy csoportra osztja a felhasználókat az elérési jogok megállapítása kapcsán (a **chmod** parancs egy-egy betűt használ a csoportok megjelölésére):

- a felhasználó (*user*) jogai (a **chmod** parancsban szimbóluma: `u`) ,
- a csoport (*group*) jogai, szimbóluma: `g` ,
- a többiek (*others*) jogai: `o` (azok akik a tulajdonos szempontjából sem a felhasználót, sem annak csoportját nem jelentik) ,
- mindenki jogai (*all*), szimbólum: `a` .

Az írás, olvasás és végrehajtás jogok közönséges fájlokon

A jogokat egy 3 jegyű oktál számmal ábrázolja a rendszer, mert 3 jog állítható be minden fájlon 3 kategória számára (user, group, others). Ezek a jogok, valamint betűjelük:

- olvasás: `r`
- írás: `w`
- végrehajtás: `x`

Tehát egy fájlon beállított, tulajdonos-csoport-mások számára beállított jog így néz ki, ha ezeket a szimbólumokat egymás mellé írjuk:

rwxr--r--

A három oktál szám `ugo`, mindegyiknek 3 bitje lehet: `rwx`, amennyiben egy bit nincs beállítva, helyette kötőjelet ír ki például az **ls** parancs. A fenti példa jelentése: a tulajdonos írhatja, olvashatja, végrehajthatja míg a csoport tagjai illetve mások csak olvashatják (oktálban a kombinációt így fejezzük ki: 744). Az alábbi jog kombináció esetében a tulajdonos bármit tehet az állománnyal, a többiek olvashatják és futtathatják de nem módosíthatják (oktál 755):

rwxr-xr-x

Az írás, olvasás és végrehajtás jogok könyvtárakon

A könyvtárak tartalmazzák a fájlok neveit: a Unix rendszerek fizikai fájlszervezésében a fájl nevek nincsenek hozzákötve a fizikai tárolás leírását tartalmazó struktúrákhoz. Ezeket a struktúrákat általában egy azonosító egész számon keresztül érjük el, a strktúra neve *i-node* (a névnek történelmi gyökerei vannak, az első UNIX tervezéséhez kötődik). A könyvtárak valósítják meg a név és i-node-ok közti kapcsolatot, így ezek listákat tárolnak, melyben a név és i-node egymáshoz rendelések találhatóak.

A könyvtárak esetében az `r` jog jelentése a könyvtár fájl "olvasása": ez gyakorlatilag azt jelenti, hogy a benne levő név lista leolvasható és listázható (tehát nem a könyvtár tartalma, hanem a könyvtár fájl tartalmára vonatkozik az olvashatóság).

A `w` jog a könyvtár fájl "írhatóságát" jelenti: tehát a könyvtárban létrehozható új fájl, törölhetőek a meglevőek.

Az `x` végrehajtási jognak megfelelő bit viszont itt egészen mást jelent: a könyvtár "átjárható", azaz amennyiben egy felhasználó a fájlrendszer fastruktúrájában keres, keresheti az illető könyvtár alatti fájlokat, illetve könyvtárakat, és leolvashatja (írhatja) azokat amennyiben joga van rá. Ugyanakkor **cd** paranccsal beléphet az ilyen könyvtárakba, és haladhat lefelé a könyvtárstruktúrán.

Az `x` jog megvonása mindezen jogok elvesztését jelenti.

A root felhasználó mindent leolvashat illetve írhat a rendszerben, és azokat a fájlokat amelyeken legalább egy `x` bit be van állítva (bárkinek) végrehajthatja.

A hozzáférési jogok állítása a **chmod**, a tulajdonos beállítása a **chown** parancsokkal történik, listázásuk az **ls** parancs `-l` opciójával. A chown parancs végrehajtása közönséges felhasználó számára nem megengedett a setuid mechanizmus veszélyei miatt (lásd alább).

A **chmod** [25] bemutatását lásd a fejezet végén [25] itt néhány példát adunk a használatára.

A `1.txt` fájlra az `rwxr--r--` jogot az alábbi parancs segítségével állítjuk:

```
$ chmod 744 1.txt
$
```

az oktál számok segítségével. Ha ezt módosítani akarjuk a `rwxr-xr--` jogra (a csoportomnak végrehajtási jogot adok) akkor az alábbit tehetjük, ha szimbólumokkal akarjuk megadni a jogokat a **chown** parancsnak:

```
$ chmod g+x 1.txt
$
```

Az implicit jogokat amelyeket a rendszer akkor használ ha egy bizonyos felhasználó egy új fájlt hoz létre az **umask [19]** paranccsal állítjuk be (lásd alább).

Az említett jogokon kívül egy negyedik oktál szám speciális információkat tartalmaz a jogosultsági bitek közt:

Jog bit neve	Milyen fájl?	Jelentése
Sticky bit	Futtatható fájl vagy könyvtár	Ha fájl: Állandóan a memóriában marad (ha több felhasználó futtatja ugyanazt a programot). A régi UNIX rendszerek használták, ma már nincs szükség erre pl. Linuxon. Amennyiben könyvtárra van megadva, ha többen írnak ugyanabba a könyvtárba, a fájlt csak a tulajdonos módosíthatja pl. ilyen a `/tmp` beállítása. Oktál értéke : 1 jele az **ls**-ben: t, mások jogait jelző x helyett áll az **ls** listájában.

Jog bit neve	Milyen fájl?	Jelentése
Set Group Id bit	fájl vagy könyvtár	Ha a Set Group Id bit könyvtáron van beállítva, akkor az alatta létrehozott könyvtárak a felső szintű könyvtár tulajdonosáé lesznek, és nem a létrehozóé. Fájlon ugyanaz érvényes mint alább a Set User Id bit-re. Oktál értéke: 2, jele: s vagy S (s = S+x) az **ls** listájában.
Set User Id Bit	Futtatható fájl	Ha egy programon be van állítva valamelyik a bitek közül, akkor bárki indítja el a programot, a program a tulajdonos, és nem a futtató jogosultságával fog futni. Pl.: a **passwd** parancs tulajdonosa root, így root-ként fut, bárki indítja el.

Oktálban a **chmod** parancs egy negyedik számjegyet használ a fenti bitek beállításakor, ennek értéke: 4: setuid, 2: setgid, 1:sticky bit. Például a `/tmp` könyvtár jogainak beállításakor az alábbi parancsot kellene kiadni ha betűkombinációt használunk:

```
$ chmod a+rwx /tmp
$ chmod +t /tmp
$
```

és az alábbit ha oktál számot:

```
$ chmod  1777  /tmp
$
```

Ezek után érthetőek az **ls [23]** parancs által kiadott listák. Az `rwx` jogok kijelzése előtt van meg egy karakter, amely a file típusát jelzi. Reguláris file esetében egy kötőjel áll ott, más speciális esetekben az alábbi karakterek: d - könyvtár, l - szimbolikus link, b - blokk eszköz, c - karakter eszköz (lásd eszközök [21]).

```
$ ls -l
total 4
-rw-rw-r-- 1 lszabo lszabo    0 Oct  3 19:35 1.txt
lrwxrwxrwx 1 lszabo lszabo    5 Oct  3 19:36 2.txt -> 1.txt
drwxrwxr-x 2 lszabo lszabo 4096 Oct  3 19:36 piros
$
```

A GNU **ls** esetében a különböző típusú fájlokat más-más színnel színezi az **ls** ha interaktív módban használjuk.

Az umask és az implicit jogok

Az újonnan létrehozott fájlok számára a héj kap a rendszertől egy beállítást, ami meghatározza milyen jogosultságokkal jön létre egy új fájl. A beállítást egy maszk tartalmazza, amelyet *umask*-nak neveznek, és információként azt tartalmazza, milyen jogosultságot ne állítson be a shell amikor létrehoz egy új fájlt. A maszk beállítása feltételezi, hogy a végrehajtási jogot

amúgy sem kell implicit beállítani (ezt mindig tudatosan állítja be egy program ha szükséges), a többi jogra pedig egy 3 oktál számból álló maszkot ad meg. Ennek eredeti értéke 0002 vagy 0022 (a szám 4 jegyű, az `rwx` biteken kívül a kiegészítő biteket is tartalmazza). A 0002 jelentése a létrehozandó fájl `ugo` csoportjainak `rwx` bitjeire vonatkozik, és az egyetlen beállított bit azt jelenti, hogy egy újonnan létrehozott fájl nem lehet mások által írható (a 0022 esetében a csoport által sem).

Az implicit jogokat egy új fájl létrehozásakor az **umask** paranccsal állíthatjuk be. A parancs egy argumentumot fogad el, amivel átállíthatjuk a maszkot. Például az alábbi parancsok kiadása után:

```
$ umask 0002
$ touch test.txt
$ ls -l test.txt
-rw-rw-r-- 1 lszabo lszabo 0 Mar  3 15:28 test.txt
$ umask 0022
$ touch test1.txt
$ ls -l test1.txt
-rw-r--r-- 1 lszabo lszabo 0 Mar  3 15:28 test1.txt
```

az újonnan létrehozott `test1.txt` fájl sem mások, sem a csoport által nem lesz írható – az előző `test.txt` pedig írható a csoport által.

A fájlok időbélyegei

Minden fájl 3 időbélyeget tartalmaz, ezek a fájlt leíró adatstruktúrában, az i-node-ban vannak tárolva.

- az utolsó hozzáférés ideje: atime (*access time*), ezt **ls -ltu** mutatja
- az utolsó módosítás ideje: mtime (*modification time*), ezt **ls -lt** mutatja
- az utolsó állapotváltozás ideje (tk. az i-node-ban tárolt tulajdonságok változása, jogok, tulajdonos, linkek száma): ctime (*status change time*), ezt **ls -lct** mutatja

A fenti **ls** parancsok, amennyiben a `-t` kapcsoló nélkül használjuk őket kijelzik a megfelelő időbélyeget és név szerint rendeznek. A `-t` kapcsolóval a megfelelő időbélyeg szerint lesznek rendezve

```
$ ls -lt
total 16
-rwxrwxr-x 1 lszabo lszabo 4847 Feb  7 16:34 a.out
-rw-rw-r-- 1 lszabo lszabo  198 Feb  7 10:43 echoarg.c
-rw-rw-r-- 1 lszabo lszabo  232 Feb  7 10:43 echo.c
$ ls -ltr
total 16
-rw-rw-r-- 1 lszabo lszabo  232 Feb  7 10:43 echo.c
-rw-rw-r-- 1 lszabo lszabo  198 Feb  7 10:43 echoarg.c
-rwxrwxr-x 1 lszabo lszabo 4847 Feb  7 16:34 a.out
$
```

Az időbélyegek egy egész számot tartalmaznak a fájlt leíró adatstruktúrában, amelynek típusa UNIX időbélyeg (UNIX timestamp). Ez az 1970 január 1 éjfél óta eltelt szekundumok számát jelenti.

Fájl típusok, eszközök

Mivel a legtöbb program bemenetet olvas és kimenetre ír, ezek pedig nem feltétlenül igazi fájlok, hanem sokszor különböző bemeneti eszközök, ezért ezeket illetve más speciális adatot is a rendszer fájlként kezel. Így pl. a könyvtárakat is fájloknak tekinti. A rendszer az alábbi fájltípusokat különíti el:

- közönséges fájl (regular file)
- könyvtár (directory)
- eszköz file (special device file), kétféle van: karakter és blokk eszköz
- hivatkozások vagy linkek
- socket-ek - a hálózati kapcsolatokat oldják meg

Megjegyzés: a lista azokat a típusokat tartalmazza, amelyekkel a leggyakrabban találkozunk, a különböző Unix változatokon más típusok is vannak, mint a *pipe* vagy *door*.

Az eszközillesztést csatlakoztató fájlok, a `/dev` könyvtárban vannak. Jellegüket tekintve háromféle eszköz van: blokk eszköz (pl. merevlemez partíció), karakter eszköz (pl. terminál) illetve pseudo eszközök (ilyen pl. a `/dev/null` vagy a `/dev/zero`).

Linux rendszereknél itt találhatóak például a merevlemezekre mutató eszközfájlok, pl. `/dev/hda1` az első lemez első partíciója, `/dev/hda2` az első lemez második partíciója, `/dev/sda1` az első SCSI vagy soros IDE lemez első partíciója.

Ezekkel a speciális fájlokkal általában rendszergazdai szerepkörben találkozunk, a mindennapi interaktív munkában viszont gyakran használunk hivatkozásokat vagy linkeket.

Hivatkozások (linkek)

A UNIX fájlrendszerben gyakran dolgozunk ún. hivatkozásokkal: ez azt jelenti, hogy egy fájl vagy könyvtárnak kettő vagy akár több nevet is adunk, többnyire azért hogy kényelmesebben dolgozzunk a rendszerrel, vagy elkerüljük a szükségtelen kettőzéseket. Kétféle hivatkozást alkalmaz a rendszer: a kemény vagy hard linket és a szimbolikus linket.

A kemény linkek esetében két névvel hivatkozunk egy fájlra, mindkét név egyenértékű. Ilyenkor ha az egyiket megszüntetjük (töröljük a fájlt), a fájl megmarad a másik név alatt (mindegy, hogy melyik névvel volt létrehozva).

A szimbolikus linkek esetében a második név csak egy mutató az elsőre. Ilyenkor, ha az első név szerint töröljük a fájlt, a fájl is törlődik, és a második név a semmibe mutat. Akárhány linket létrehozhatunk egy fájlra. Kemény (hard) hivatkozásokat csak ugyanazon a fájlrendszeren található fájlnevek közt hozhatunk létre, a szimbolikus linkek átmutathatnak fájlrendszerek

fölött (fájlrendszer alatt itt például egy merevlemez partíción található fájlrendszert értünk, amelyet felcsatolunk a könyvtárstruktúrába).

Könyvtárakra nem hozhatóak létre kemény hivatkozások.

Létrehozásuk az **ln** paranccsal történik, használata:

ln eredeti_név új_név

A szimbolikus linkek esetében ezen kívül a `-s` kapcsolót kell alkalmazni, tehát:

```
$ ln 1.txt 2.txt
$
```

parancs hard linket hoz létre, a

```
$ ln -s 1.txt 3.txt
```

pedig szimbólikusat. Az alábbi listán látszik, hogy a `3.txt` szimbolikus link (kis `l` betű a sor elején). A GNU **ls** parancsa a név mellett egy `->` jellel is jelzi a hivatkozást. A lista második oszlopa pedig a fájlra mutató kemény hivatkozások számát jelzi.

```
$ ls -l
-rw-rw-r-- 2 lszabo lszabo 5 Sep 21 11:37 1.txt
-rw-rw-r-- 2 lszabo lszabo 5 Sep 21 11:37 2.txt
lrwxrwxrwx 1 lszabo lszabo 5 Sep 21 11:38 3.txt -> 1.txt
$
```

A fájl nevekben használható metakarakterek

A parancssoron speciális karaktereket használhatunk akkor, amikor fájlnevekre hivatkozunk. Ezek megengedik az általánosabb neveket, és egyszerre a több fájlra való hivatkozást. Az alábbi táblázatban található jelöléseket lehet használni. Amint látni fogjuk, ezeknek a kezelését a héj végzi.

Bizonyos karaktereknek a héj számára más jelentéssel bírnak, mint eredeti jelentésük. Mielőtt a shell végrehajtaná a parancsainkat, átírja (behelyettesíti) ezeket, ezért *metakarakteresnek* nevezzük őket. Jellegzetes példa a `*`, amely a shell számára a munkakönyvtárban található fájlok listáját jelenti.

3.2. táblázat - A fájlnevekben használható metakarakterek

Metakarakterek	Jelentés
*	A megadott könyvtárban minden állomány nevet jelöl ha egyedül áll: `*` , de használható más karakterekkel kombinálva is: `*.txt` minden `.txt` típusú fájlt jelent.
?	Az állománynévben egy karaktert helyettesít: `1?.txt` (minden `.txt` típusú állomány amelynek neve 1-el kezdődik, és utána bármilyen karakter áll a pont előtt).

Metakarakterek	Jelentés
[halmaz]	Karakterhalmazt definiál: a két szögletes zárójel közé azok a karakterek kerülnek amelyek részei a halmaznak. A halmazzal egy karakterre hivatkozunk. `[abx]` - jelenti az `a`, `b` vagy `x` karaktereket; `[a-z]` - jelenti a kisbetűket Például: `1[0-9].txt` (minden `.txt` típusú állomány amely neve 1 -el kezdődik, és utána számjegy áll a pont előtt).

{felsorolás}	Bár nem egyetlen metakarakterről van szó, itt beszélünk a héjnak erről a megoldásáról, mert fájlnevekben használható. Sztringek felsorolását adjuk meg. A shell azokat a fájl neveket válassza ki, amelyekben a felsorolt sztringek alternatív szerepelnek: `{ab,cd}` akár az ab, akár a cd sztringet jelöli:
	
	`$ ls {a,b}.txt`
	`a.txt  b.txt`
	`$`

A `*` és `?` nem illeszkedik ha a fájlnévben az első karakter a pont: `.` . Ezek a rendszerben az un. láthatatlan fájlok, például: `.bash_profile` .

Ha a fájl nevekben a metakarakterek literálisan fordulnak elő, vissza-per jelöléssel használhatjuk őket, akárcsak az elválasztó (szóköz) karaktert. Pl: "`piros alma.txt`" fájl nevet megadhatjuk így is: `piros\ alma.txt`. Ennek ellenére, sem a metakatraktereket sem az elválasztó karaktereket nem célszerű fájlnevekben használni.

A fejezetben használt parancsok

Itt megadjuk a fejezetben használt parancsok rövid leírását. Részletes információkért a parancsok kézikönyv vagy info lapjait kell átolvasni.

Az ls parancs

Kilistázza egy vagy több könyvtár tartalmát.

Használat:

`ls` [-lhdirSt] *`fájl(ok)`*

Fontosabb opciók:

`-1` egy oszlopos kimenetet generál, minden fájl neve új sorba kerül.

`-a` Kilistázza a rejtet fájlokat is (all). Rejtett fájlok azok, amelynek neve egy ponttal kezdődik (pl. `.bash_profile`).

`-d` Könyvtárak tulajdonságait listázza.

`-h` Emberi szemmel könnyen olvashatóian írja ki a fájl hosszakat(K, M, G – kiló, mega, gigabyte).

`-i` Az inode számokat listázza.

`-l` Hosszú lista: minden fontos fájl tulajdonságot kiír.

`-S` A fájlok hossza szerint rendez.

`-t` A fájlok módosítási ideje szerint rendez.

`-r` Megfordítja az éppen alkalmazott rendezési listát.

Leírás:

Az **ls** egy vagy több könyvtárban található fájl nevét írja ki. Argumentuma hiányozhat, ekkor a munkakönyvtár tartalmát listázza. Ha van argumentum, akkor az a cél fájl vagy könyvtár. Ha több információra van szükségünk, akkor azt kapcsolóval kell kérni. Implicit a fájlok neve szerint rendezi a listát, ez módosítható pl. a `-t` vagy `-S` kapcsolókkal.

Például a `-l` opcióval részletes listát ad:

```
$ ls
1.txt  a.txt  test.txt  x.txt
$ ls -l
total 4
-rw-rw-r-- 1 lszabo lszabo  0 Oct  2 21:30 1.txt
-rw-rw-r-- 1 lszabo lszabo  0 Oct  2 21:30 a.txt
-rw-rw-r-- 1 lszabo lszabo 55 Oct  2 21:30 test.txt
-rw-rw-r-- 1 lszabo lszabo  0 Oct  2 21:30 x.txt
$
```

A `-t` opcióval módosítási idő szerint rendez.

```
$ ls -lt
total 4
-rw-rw-r-- 1 lszabo lszabo  0 Oct  2 21:30 1.txt
-rw-rw-r-- 1 lszabo lszabo  0 Oct  2 21:30 a.txt
-rw-rw-r-- 1 lszabo lszabo 55 Oct  2 21:30 test.txt
-rw-rw-r-- 1 lszabo lszabo  0 Oct  2 21:30 x.txt
$
```

A mkdir parancs

Létrehozza az argumentumában megadott könyvtárakat. .

Használat:

`mkdir` [-pv] *`könytár(ak)`*

Fontosabb opciók:

`-p` parent: megengedi több szint létrehozását egy paranccsal

`-v` verbose: kiírja amit végez

Leírás:

Létrehozza az argumentumában megadott könyvtárakat. Akár többet is megadhatunk egyszerre. Például:

```
$ mkdir alpha
$ mkdir -p gamma/beta
$
```

Az rmdir parancs

Könyvtárat vagy könyvtárakat töröl.

Használat:

`rmdir` [-vp] *`könyvtár(ak)`*

Fontosabb opciók:

`-p` parent: megengedi több szint törlését

`-v` verbose: kiírja amit végez

Leírás:

Törli az argumentumában megadott könyvtárakat. Csak akkor törli őket, ha a könyvtárak üresek.

Például:

```
$ rmdir elso
$ rmdir masodik harmadik
$ rmdir -p negyedik/otodik
$
```

A chmod parancs

Jogokat állít be a fájlrendszerben.

Használat:

`chmod` [-fR] *`mód fájl(ok)`*

Fontosabb opciók:

`-f` Erőlteti a jog változtatást és nem ad ki figyelmeztető üzenetet (force).

`-R` A változtatásokat elvégzi rekurzívan lefelé a könyvtárstruktúrán.

Leírás:

A parancs megváltoztatja a *mód* argumentumban megadott jogokat a paraméterként megadott fájlon vagy fájlokon. A *mód* argumentum kötelező. Kétféleképpen adható meg: oktál formában vagy szimbólikus jelöléssel. Oktál jelöléssel például:

```
$ chmod 755 a.txt
$
```

A szimbólikus jelölésnél az `ugoa` szimbólumokat használva adjuk meg a paramétert (`u`: user, `g`: group, `o`: others, `a`: all). A + jel hozzáadást, a - megvonást jelent, a hozzárendelést az = jellel jelöljük. Pl.:

```
$ chmod u+x elso.sh
$
```

végrehajtási jogot állítunk be;

```
$ chmod  u-x, o-w 1.txt
$
```

a felhasználótól a végrehajtási jogot, a többiektől az írásjogot vonjuk meg.

A rekurzív használat veszélyes (egyszerre állíthat be könyvtárakon és fájlokon jogokat, ami nem várt eredményhez vezethet).

A cp parancs

Fájlokat másol.

Használat:

`cp` [-viprlf] *`forrás cél`*

`cp` [-viprlf] *`forrás(ok)... könyvtár`*

`cp` [-viprlf] -t *`könyvtár forrás(ok)...`*

Fontosabb opciók:

`-i` interaktívan fut (kérdez, ha felülírás esete áll fenn)

`-v` verbose: kiírja amit végez

`-r` rekurzívan másol egy könyvtárstruktúrát lefele

`-p` a cél fájl felveszi a forrás jogait

`-l` másolás helyett hivatkozást készít

`-f` force: ha olyan fájlt talál, amire nem tud másolni, megpróbálja törölni és megismételni a műveletet

Leírás:

Állományokat másol. Az első alakban egy forrás fájlt egy célba, a második alakban akár több fájlt egy cél könyvtár alá. A harmadik alakban, ha a `-t` kapcsolót használjuk, a célkönyvtár a `-t` utáni első paraméter. Ha ugyanolyan nevű a cél fájl mint a forrás, szó nélkül felülírja: ezt a `-i` kapcsolóval kerülhetjük el.

```
$ mkdir elso
$ touch 1.txt
$ #fájlt fájlba
$ cp 1.txt 2.txt
$ #egy fájlt két különböző helyre
$ #több állomnyt ugyanabba a könyvtárba
$ cp 1.txt 2.txt elso/
$ mkdir masodik
$ cp -t masodik/ 1.txt 2.txt
$ tree
.
|-- 1.txt
|-- 2.txt
|-- elso
|   |-- 1.txt
|   `-- 2.txt
`-- masodik
    |-- 1.txt
    `-- 2.txt

2 directories, 6 files
$ # másolás jóváhagyással
$ cp -i 1.txt 2.txt
cp: overwrite `2.txt'? y
$
```

A **cp** implicit csak fájlokat másol. Könyvtárakat csak a `-r` kapcsolóval másolhatunk: így nem csak a forráskönyvtárat, hanem annak teljes tartalmát átmásolja Az alábbi példában az `elso` és `masodik` is könyvtár:

```
$ ls
elso  masodik
$ cp elso harmadik
cp: omitting directory `elso'
$ cp -r elso harmadik
$
```

A rm parancs

Fájlokat vagy könyvtárakat töröl.

Használat:

`rm` [-viprlf] *`fájl(ok)`*...

Fontosabb opciók:

`-i` *interactive*: kérdez minden törlés előtt

`-v` kiírja amit végez

`-f` *force*: akkor is elvégzi a törlést ha az állomány írásvédett (de a felhasználóé)

`-r` rekurzívan töröl: ilyenkor könyvtárakat is töröl

Leírás:

Törli a megadott fájlokat. Akár többet is megadhatunk egyszerre. Csak fájlokat töröl, de ha megadjuk a `-r` kapcsolót akkor könyvtárat is, az alatta található teljes tartalommal.

```
$ # egy fájl törlése
$ rm 1.txt
$ # több fájl törlése
$ rm 2.txt 3.txt 4.txt
$ #egy könyvtár törlése az alatta levő tartalommal együtt
$ rm -rf elso
$
```

A parancssoron kiadott törlés a Unix alatt végleges: nincs módszer az állomány visszaállítására törlés után, ezért a műveletet, különösen ha a -r opciót is használjuk, kétszer meg kell gondolni, ajánlott előtte listázni a parancssoron megadott állományokat ugyanazzal a névvel ellenőrzés végett. Az:

```
$ rm *
$
```

alakú parancstól óvakodni kell, ajánlott a `-i` kapcsolóval használni:

```
$ rm -i *
$
```

vagy előtte ellenőrizni, hogy **ls *** mit listáz.

Az mv parancs

Fájlokat nevez át (mozgat el).

Használat:

`mv` [-vif] *`forrás cél`*

`mv` [-vif] *`forrás(ok)... cél`*

Fontosabb opciók:

`-i` interactive: kérdez, ha felülírás esete áll fenn

`-f` nem jelzi, ha felülír egy állományt ami írásvédett

`-v` verbose: kiírja amit végez

Leírás:

Az első alak elköltöztet egy állomány más név alá. Ha a cél ugyanazon a fájlrendszeren található (pl. merevlemezek esetében ugyanazon a lemezpartíción), akkor tulajdonképpen csak átnevez. Ha másikon, akkor át is kell másolnia a tartalmát. Az eredeti állomány ilyenkor törlődik. Állományokat és könyvtárakat egyaránt elmozgat. Ha több fájon végezzük egyszerre (második alak), akkor a cél könyvtár, és a fájlokat a könyvtár alá viszi.

```
$ #létrehozunk 2 könyvtárat és 2 fájlt
$ mkdir elso masodik
$ touch 1.txt 2.txt
$ #az 1.txt átneveződik 3.txt-vé
$ mv 1.txt 3.txt
$ #minden .txt típusú fájlt az elso könyvtár alá visz
$ mv *.txt elso/
$ #az elso könyvtár a második alá kerül
$ mv elso/ masodik
$ tree
.
`-- masodik
    `-- elso
        |-- 2.txt
        `-- 3.txt

2 directories, 2 files

$
```

4. fejezet - Alapfogalmak a héj (shell) használatához

Héjprogramok

Egy héjprogram vagy shell szkript egyszerű szöveges fájl, amelyet a héj soronként értelmez. Minden egyes sort külön parancsnak értelmez (*mintha* a parancssorról ütöttük volna be), és egyenként végrehajtja őket. A parancsok indításának szintaxisát láttuk már a terminál használatakor: a parancs nevéből, opciókból (kapcsolók) és argumentumokból áll.

A héj számára a megjegyzést deklaráló karakter a # (kettős kereszt vagy diez, angol: *hash mark*). Azokat a sorokat amelyekben megjelenik, a kettős kereszt karaktertől a sor végéig megjegyzésnek tekinti a héj, akár interaktív módban adtuk meg, akár egy héj programban (az alábbi példában a héj készenléti jele a `$` és a **cat** valamint a **sort** programokat indítjuk el egy kis szöveges állománnyal).

```
$ #fájl tartalmának listázása
$ cat lista.txt
delta
omega
alpha
beta
$ #itt rendezzük a fájlt
$ sort lista.txt
alpha
beta
delta
omega
```

A **cat** parancs a standard kimenetre írja a parancssorán megadott szöveges fájlt vagy fájlokat, összefűzi őket (*concatenate*), használata:

`cat` [*`fájl`*...]

A **sort** az argumentumaként megadott szöveges fájl sorait rendezi ebben az egyszerű használati módban, rendezés után kiírja őket a terminálra.

A héj programok szövegének első sora speciális. Itt adunk meg információt arról, hogy a szöveges fájlt milyen értelmezővel szeretnénk végrehajtani. A sor egy olyan megjegyzést tartalmaz, amelyben a kettős keresztet azonnal egy felkiáltójel követ (tehát: #!) , utána pedig a kívánt értelmező teljes elérési útja áll.

Így egy egyszerű szkript így néz ki:

```
#!/bin/bash
echo helló
echo vége
```

A végrehajtás azt jelenti, hogy elindítjuk a parancsértelmezőt (és ez egy új héj lesz – új folyamat - nem ugyanaz ami alól indítottuk) úgy, hogy argumentumként megadjuk a programot tartalmazó fájlt. A két **echo** helyén bármilyen a héj számára értelmes program szöveg állhat, egymás utáni parancssorok amelyeket a héj egymás után: mindig az előző lefutását megvárva hajt végre. Ugyanez történik, ha a parancsokat tartalmazó szöveges fájlnak végrehajtási jogot adunk, és azt indítjuk, mintha egy parancs lenne:

```
$ chmod u+x hello.sh
$ ./hello.sh
helló
vége
```

Ebben az esetben a héj a fentebb bemutatott első program sor szerint válassza ki a parancsértelmezőt, és ugyanúgy új folyamatot indít. Az indításnál szükségünk van a `./` relatív útvonal megadására, mert alaphelyzetben a munkakönyvtár - ahol a programunk van - nincs a héj PATH változójának listájában.

Amennyiben a **bash**-t megjelölő sor hiányzik, a héj megpróbálja a szöveget a felhasználó számára beállított alapértelmezett héjjal végrehajtani (ennek nevét a `SHELL` környezeti változó [34] tartalmazza):

```
$ echo $SHELL
/bin/bash
$
```

A jelölés más szkript nyelvek esetében is érvényes, általánosan, ha a shell egy szöveget értelmező programot akar elindítani, pl. egy Perl szkript esetében a sor:

```
#!/bin/perl
```

Az héj elérésének útvonala rendszerenként változhat, ellenőrizni kell, ezt a **which** paranccsal tehetjük meg, amely kiírja egy végrehajtható, és a shell által megtalálható (lásd később a `PATH` változót) program elérési útját:

```
$ which bash
/bin/bash
```

A szkript másik végrehajtási módja azzal a héjjal való végrehajtás amellyel dolgozunk, ezt a **source [56]** paranccsal tesszük (ugyanezt a műveletet végzi a **.** - **dot** [56] parancs is):

```
$ source hello.sh
helló
vége
```

A fenti programban használtunk ékezetes karaktereket is, régi rendszerek általában csak ASCII kódolással tudták a karakteret kiírni a terminálra. A jelenlegi rendszerek nagy része, köztük valamennyi Linux disztribúció implicit UTF-8 kódolást használ, tehát elvileg lehet ékezetes karaktereket is használni. A shell `LANG` vagy `LC_ALL` változói tartalmazzák a beállított nyelvet

és kódolást. Ezt, valamint ha távoli terminálról lépünk a rendszerre, akkor a terminál kódolását is ellenőrizni kell.

```
$ echo $LANG
en_US.UTF-8
$
```

A héj által végrehajtott parancsok

Amikor a héj megkap egy parancsnevet, az alábbi lehetőségek közt kezdi keresni a nevet a végrehajtás elvégzéséhez:

- **Alias**-ok: helyettesítő nevek vagy pszeudo parancsok (lásd alább [32]).
- **Függvények** (*function*) - a héj nyelvén függvényeket [105] lehet meghatározni az olyan feladatokra amelyeket többször hívunk meg. Később tárgyaljuk őket.
- **Beépített parancsok** (*builtin commands*)- a nagyon gyakran használt parancsokat beépítik a héjba. Így ezeket sokkal gyorsabban lehet végrehajtani. Ilyen például az **echo**, **pwd**, **test** (hamarosan használni fogjuk ez utóbbiakat). Ugyanakkor ezek léteznek önálló parancsként is (bináris programként). A beépítésnek két oka lehet: a sebesség (nem kell új folyamatot indítani) vagy a héj belső változóihoz való hozzáférés.
- **Külső parancsok** - azok a parancsok amelyek programok formájában valahol a fájlrendszerben találhatóak.

Ugyanolyan név esetén a prioritási sorrend alias, függvény, beépített majd külső parancs.

A Bash héjba épített parancsokra leírást a Linux rendszerekben a **help** paranccsal kapunk. A **help** argumentum nélkül a beépített parancsokat, argumentummal egy parancs segédletét listázza. Tehát a **man 1 echo** a Linux külső **echo**-jának, a **help echo** pedig a belső **echo** segédletét írja ki. Bash alatt az ugyanolyan nevű beépített parancs fut a külső helyett (ha nem hivatkozunk a külsőre speciálisan).

A külső parancsok programokat jelentenek, amelyeket a héj meg tud keresni a fájlrendszerben, és a héjat futtató felhasználónak végrehajtási joga van reájuk. Ezeket, ha csak parancsnevükkel adjuk meg a `PATH` környezeti változóban leírt útvonalakon keresi a héj. Ennek tartalma egy elérési út lista, az elemek kettősponttal vannak elválasztva:

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/home/lszabo/bin:
```

Ezekben a könyvtárakban fogja a héj a végrehajtható parancsokat keresni.

Alias-ok vagy helyettesítő nevek

Az alias-ok vagy helyettesítő nevek arra szolgálnak, hogy a gyakran használt, több opcióval meghívott programoknak egy helyettesítő nevet adjunk, és gyorsabban, egyszerűbben hivatkozzunk rájuk. Jellegzetes példa a Linuxon az **ls** parancsot helyettesítő alias, amely színes kiírással hívja meg az **ls**-t. Létrehozása az **alias** paranccsal történik:

```
$ alias ls='ls --color=tty'
```

Amint látható, az alias létrehozás tulajdonképpen a névnek egy karakterláncot feleltet meg, és meghívásakor ez helyettesítődik a parancssorra. A helyettesítő neveknek elsőbbségük van az ugyanolyan nevű parancsokhoz képest amikor meghívjuk őket. Törlésük az **unalias** , a már definiált alias-ok listázása egyszerűen az **alias** paranccsal történik.

A parancsvégrehajtás

Hogyan fut le egy egyszerű parancsvégrehajtás? Beépített parancsok esetében a héj végzi ezt. Külső parancsok, azaz programok futtatása során, pl. ha beírjuk parancsként az **ls** parancsot, az alábbi történik:

- a héj elindítja az **ls**-t (új fiú folyamatot indít, és abban fut az **ls**)
- "háttérbe vonul" (vár amíg az **ls** lefut)
- ez alatt amennyiben az **ls** ír, a szöveg a terminálra kerül, ha olvasna, a terminálról olvas
- ha az **ls** lefutott és kilépett a héj kiírja a készenléti jelet.

Egy parancssoron (az új sor elválasztó karakterig tartó karakterlánc) általában egy programot indítunk el. Megtehetjük, hogy többet indítsunk el egyszerre, ha köztük a **;**-t használjuk mint elválasztó jelet.

```
$ echo "rendezés indul" ; sort szoveg.txt;  echo "vége"
rendezés indul
vége
$
```

Ilyenkor a második parancs megvárja az elsőt, tehát nem párhuzamosan hajtódnak végre, ugyanaz történik mintha 2 parancssort írtunk volna.

A parancssoron elindított programról azt mondjuk, hogy "előtérben fut"; mivel a terminálra ír, és ilyenkor a héj nem adja vissza a készenléti jelet (*prompt*): azt jelzi ezzel, hogy nem fogad el új parancsot az előzőleg elindított befejezéséig.

Háttérben való végrehajtás

Megtehetjük, hogy a parancsot "*háttérben*" futtassunk; ilyenkor a héj visszaadja a készenléti jelet, és új parancsot üthetünk be. A háttérben futtatást kétféleképpen válthatjuk ki:

1. a parancssort a parancs indításakor az **&** jellel zárjuk, ilyenkor a folyamat azonnal a háttérben indul
2. az előtérben futó folyamatot felfüggesztjük (*suspend*) a Ctrl-Z billentyű kombinációval a terminálról , majd a **bg** (*background*) paranccsal háttérben indítjuk

A folyamatnak a `Ctrl-Z` kombinációval *jelzést [159]* küldünk: a felfüggesztés tipikus példa erre. Lenyomunk egy billentyűt amelyet egy megszakítás közvetítésére használunk, és azt a héj rendszert vezérlő kernelen keresztül eljuttatja a folyamathoz.

Mindkét esetben a héj 2 számot, egy feladat [153] (*job*) azonosítót illetve egy folyamat azonosítót ír ki számunkra.

```
$ sort rendezetlen.txt > rendezett.txt &
[1] 4467
$
```

Ezek segítségével hivatkozhatunk később az elindított munkára, például az előtérbe hívhatjuk az **fg** (*foreground*) paranccsal.

A munkák illetve folyamatok kezelésével, valamint ezek különböző indítási módjával később, a Folyamatok kezelése [151] c. fejezetben foglalkozunk, itt röviden megadtuk a használathoz szükséges szintaxist.

A környezeti változók

A héj, hasonlóan más programnyelvekhez az információkat, adatokat névvel látja el és változóknak nevezi őket. Részletesen a következő fejezetben foglalkozunk ezekkel. Ebben a fejezetben kimondottan a parancshasználattal kapcsolatos környezeti változók miatt említjük őket.

Egy folyamat a környezetéről (az őt futtató operációs rendszerről, gépről, beállításokról) információkat tudhat meg. Ezeket az információkat változókon keresztül kapja (név-érték párok). Nagy betűs változó nevekkel adjuk meg ezeket, különbséget téve így (a programozó számára) a saját, illetve az operációs rendszertől kapott változók közt. ilyen változót már láttunk: például a végrehajtható fájlok elérési útja, a `PATH` változó.

Néhány környezeti változót az alábbi listában láthatunk. a beállított változókat a **printenv** vagy **env** parancsokkal lehet kilistázni. Például az alábbi változók szinte minden gépen be vannak állítva:

```
TERM=xterm
SHELL=/bin/bash
USER=lszabo
MAIL=/var/spool/mail/lszabo
PATH=/bin:/usr/local/bin:/bin:/usr/bin:/home/lszabo/bin
PWD=/home/lszabo
EDITOR=/usr/bin/vim
LANG=en_US.UTF-8
HOME=/home/lszabo
```

A héj indítása

Indításkor a héj végigolvas néhány konfigurációs fájlt, ezek változókat állíthatnak vagy programokat indíthatnak, amelyek meghatározzák azt a környezetet amelyben a héj indul.

A konfigurációs fájlok különböznek, attól függően, hogy a héj interaktívan fut vagy szkriptet hajt végre. Az interaktívan futtatott héj esetében a bejelentkezéskor induló héj (*login shell*) is külön fájlokat olvas. Interaktív héjnak nevezzük azokat a héjakat amelyek direkt a terminállal dolgoznak - onnan kapják a parancsokat. Egy adott szkriptet futtató héj általában nem interaktív.

Más shell

Az alábbi konfigurációs fájl nevek a Bash-re érvényesek, más héjak esetenként más indító fájlokat használhatnak.

Egyes konfigurációs fájlok közösek, tehát bármely felhasználó belépésekor értelmeződnek, és a rendszer `/etc`, konfigurációs fájlokat tartalmazó könyvtárában találhatóak (pl.: `/etc/profile`, `/etc/bashrc`). Ezen kívül minden felhasználónak saját fájlai vannak a home könyvtárában (`.profile`, `.bash_profile`, `.bashrc`). Ezeket szabadon szerkesztheti, és amennyiben indításkor automata módon akar változókat, alias definíciókat, stb. beállítani, azt innen teheti meg. A `.profile` és `.bash_profile` fájlok a bejelentkezési interaktív héj indításnál értelmeződnek (a megadott sorrendben).

A `.bashrc` fájl is tartalmaz beállításokat, ezek más interaktívan indított héj indításakor értelmeződnek.

A nem interaktív héjaknak külön indító állomány jelölhetünk ki a BASH_ENV változó segítségével.

Ha állandó jellegű beállításokat akarunk végezni, akkor ezeket a fájlokat kell megváltoztatni. Például, ha gyorsabb interaktív munka miatt létre akarunk hozni egy **alias**-t, akkor ezt a `.bash_profile` fájl végére írjuk. A **locate** program fájlnevek gyors keresésére szolgál, használata:

`locate` *`sztring`*

, kilistázza azokat a rendszeren található fájlneveket amelyekben megtalálható az adott sztring. Ha rövidítve szeretnénk használni, pl. csak:

```
$ l sztring
```

, és azt szeretnénk, hogy ezt minden shell indításkor megtehessük, ezt kell írnunk a `.bash_profile` végére:

```
alias l='locate'
```

Nyelvi karakterkészlet beállítása

A héj karakterkészletének beállítása fontos, amennyiben például a magyar nyelv karaktereit akarjuk használni. Ezt a `LANG` illetve az `LC_` kezdőbetűket tartalmazó héj változókkal tehetjük. Két paramétert kell ilyenkor beállítani: a karakterkódolást (*encoding*) és a nyelvet, esetleg országot amelyben az illető nyelvet beszélik. Az `LC_` nevű változókkal (`LC_ALL`, `LC_TIME`, `LC_NUMERIC`) a nyelven kívül különböző helyi beállításokat lehet megadni (idő, számok, stb. formázása).

A nyelvi beállítást, amennyiben nincsenek `LC_` változók megadva, elsődlegesen a `LANG` változó adja meg, ennek lehetséges értékeit (ami függ attól, hogy fel vannak-e telepítve a szükséges szoftverek a rendszerre), a **locale -a** paranccsal lehet kilistázni.

Az UTF-8 kódolás használata ma már jelentősen megkönnyíti ezeket a beállításokat.

A standard bemenet és kimenet

A UNIX egyik alapelve, hogy minden eszközt, hardvert, stb.-it fájlként kezel a virtuális fájlrendszere. Így az eszközökre való írás mindig egyszerű fájlba való írásra vezethető vissza a programok szintjén, ami nagyon leegyszerűsíti kezelésüket. Gyakorlatilag minden rendszer komponens, amelyre írni illetve ahonnan olvasni lehet fájlként viselkedik (a közönséges fájltól kezdve a terminál billentyűzetét keresztül a merevlemezig).

Megjegyzés: a "minden eszköz fájl" állítás (bár általában igaz, és jól használható) alól vannak kivételek, mint pl. Linux alatt a hálózati kártyák.

Egy C nyelven írt program mindig (legkevesebb) 3 fájlt talál nyitva indításkor. Ezeket a programozónak nem kell külön előkészítenie vagy megnyitnia. Ezek: a standard bemenet, kimenet és hibakimenet. Mindhármat fájlként kezeli a héj (akárcsak a C nyelv). Jelölésükre egész számot tartalmazó azonosítókat használ a C nyelv akkor, amikor első szintű fájlkezelő függvényeket használ.

- standard bemenet, azonosító: 0
- standard kimenet, azonosító: 1
- standard hibakimenet, azonosító: 2

Ezeket az azonosítókat a héj is használja, és a héj által elindított programok gyakran használják ezt a 3 fájlt.

Amennyiben a **cat** parancsot argumentum nélkül indítjuk, a bemeneti fájl helyett a standard bemenetet fogja olvasni és értelemszerűen a standard kimenetre ír. Így gyakorlatilag a standard bemenetet (ami a terminál használata esetében a billentyűzet) átmásolja terminál kimenetére, ha sorokat kezdünk beírni:

```
$ cat
alpha
alpha
beta
beta
gamma
gamma
$
```

Több Unix szöveget feldolgozó program működik ezzel a logikával, általában valamilyen műveletet is végeznek a beolvasott szövegsoron, így szűrőknek nevezzük őket. A **cat** a fenti működési módban egyszerűen átmásolja a standard bemenetet a kimenetre.

Átirányítások

Átirányításnak nevezzük az a műveletet, amikor egy adott kimenetre (vagy bemenetről) érkező adatsort egy, az eredetitől különböző kimenetre küldünk (illetve bemenetről fogadunk). Például, egy standard bemenetről olvasó programnak egy fájlból érkező szöveget vagy adatsort küldünk, úgy, hogy a programnak nem lesz tudomása, honnan érkezik a szöveg. A héj néhány metakaraktert használ a standard bemenet és kimenet illetve hibakimenet átirányítására, ezeket átirányítási operátoroknak nevezzük.

A standard bemenet átirányítása (<)

A **sort**, akárcsak a **cat**, argumentum nélkül indítva a standard bemenetet olvassa, és az ott érkező sorokat rendezi. Az alábbi példában a **sort** program bemenete nem a terminál által generált standard bemenetről jön, ahogy az a **sort** egyszerű futtatásánál tenné, hanem a `lista.txt` nevű fájlból. A < jel (operátor) megváltoztatja a **sort** standard bemenetének forrását, és a jel jobb oldalán álló fájl tartalmát irányítja a program bemenetére. A **sort** nem tud arról, hogy pontosan honnan kapja a bemenetet: rendezi a standard bemenetre érkező szöveget.

A standard kimenet fájlba irányítása (>)

Az alábbi példában a **cat** program nem a terminálra, hanem a `lista.txt` nevű fájlba ír, és a standard bemenetről olvas:

```
#a cat-tel a standard bemenetről olvasunk és standard
#kimenetét egy fájlba küldjük
$ cat > lista.txt
delta
omega
alpha
beta
#a terminálra listázzuk a fájlt
$ cat lista.txt
delta
omega
alpha
beta
$
```

A > jel (fájlba irányítás operátor) végrehajtás előtt törli a régi fájlt ha ilyen nevű már létezik. A szöveg bevitelét a Ctrl-D karakterrel zárjuk (fájl vége jel). A parancs így alkalmas egy kis szöveges fájl létrehozására, amelyet a fejezet példáiban használtunk.

A két típusú átirányítást egyszerre is használhatjuk a parancssoron:

```
$ sort < lista.txt > rendezett.txt
```

a **sort** a `lista.txt` fájlból olvas és a `rendezett.txt` fájlba ír.

A fenti átirányítások tulajdonképpen az alábbit jelentik:

```
$ sort < lista.txt  1> rendezett.txt
$
```

de a standard bemenet és kimenetet jelentő fájl azonosítókat nem kell kiírni.

A kimenet hozzáfűzése egy fájlhoz (>>)

Ezzel az átirányítással hozzáfűzés módban írunk a kimeneti fájlba, tehát a fájl tartalma amennyiben létezik megmarad, és az új sorok a fájl végére kerülnek.

```
$ echo elso > f.txt
$ echo masodik >> f.txt
$ echo harmadik >> ff.txt
$ cat f.txt
elso
masodik
harmadik
$
```

Csővezetékek

A csővezetékek kommunikációs csatornák, amelyeken keresztül egy folyamat kimenete egy másik bemenetére irányítható egy rendkívül egyszerű szintaxissal a parancssoron. A deklaráláshoz használt jel a `|` elválasztójel. Az alábbi példában a **cat** kimenetét a **sort** standard bemenetére irányítjuk:

```
$ cat lista.txt | sort
alpha
beta
delta
omega
$
```

Az átvitel szinkronizációról az operációs rendszer (a kernel) gondoskodik. Az említett átirányítások többször is alkalmazhatóak ugyanazon a parancssoron. Az alábbi parancssoron a `lista.txt` fájl a **cat**-ből a **sort**-ra, onnan pedig a **head** parancs bemenetére kerül:

```
$ cat lista.txt | sort | head -1
alpha
$
```

A **head** szintén szövegsorokat szűrő parancs: a bemenetére kerülő szöveges fájl vagy fájlok első sorait listázza, mégpedig annyi sort, amennyit egész számot tartalmazó opciója megad. Így a fenti parancs kimenete csak a rendezett lista első sora. A **head** használata:

`head` [-n] {*`fájl`*...}

A listázandó sorok számát megadó kapcsoló opcionális, amennyiben nem adjuk meg, implicit értéke 10.

Itt említjük meg a **tee** parancsot is, amely sorozatos feldolgozás esetén biztosíthatja egy köztes kimenet elmentését, ha arra később szükség lehet. A **tee** használata az alábbi:

`tee` [-a] {*`fájl`*}

A parancs a bemenetet a kimenetre és az megadott nevű fájlba írja (a `-a` kapcsolóval hozzáfűzi, nélküle új fájlt nyit). Az alábbi esetben végrehajtódik ugyanaz a feldolgozás mint a fenti példában, de ezúttal a **sort** által előállított teljes rendezett lista is elmenésre kerül a `rendezett.txt` fájlba:

```
$ cat lista.txt | sort | tee rendezett.txt | head -1
alpha
$
```

Speciális eszközök

A fájlokon kívül eszközöket is használhatunk a ki-és bemenetek átirányításakor. Ilyen például a `/dev/null` eszköz, amely végtelen sok karaktert teljesen elnyelő kimenetként viselkedik. Erre lehet minden olyan kimenetet irányítani, amelyre nincs szükség: jelenléte felesleges vagy zavaró.

A standard hiba kimenet azonosítója a 2-es, erre íródnak a parancsok hibaüzenetei. Ez lehetővé teszi a parancsok hasznos kimenetének és a hibaüzeneteknek a kettéválasztását. Interaktívan dolgozva a terminálon, a parancsok által az 1-es és 2-es kimenetekre írt szövegek együtt kerülnek a terminál kimenetére. Ha a hibaüzenetek nem érdekelnek, csak a hasznos kimenet át lehet irányítani őket a null eszközre. Ha az `1.txt` és `2.txt` fájlokról kérünk egy listát az **ls**-el, amennyiben valamelyik nem létezik az **ls** hibaüzenete összekeveredik a kimeneten a hasznos információval. Az alábbi példában a **wc** (*word count*) paranccsal próbálunk információt nyerni szöveges fájlokról.

```
$ wc 1.txt
2 2 8 1.txt
```

A **wc** az argumentumában megadott szöveges fájl sorainak, szavainak és karaktereinek számát írja ki, a három szám után a fájl nevével. Használata

`wc` [-l] [-w] [-c] [*`file`*...]

Opciói megengedik külön a sorok (`-l`), szavak (`-w`) vagy karakterek (`-c`) listázását. Több fájlt is megadhatunk a parancssorán:

```
$ wc 1.txt 2.txt
 2  2  8 1.txt
 1  1 11 2.txt
 3  3 19 total
$
```

Ellenben ha például a `2.txt` nem létezik, ezt kapjuk:

```
$ wc 1.txt 2.txt
2 2 8 1.txt
wc: 2.txt: No such file or directory
2 2 8 total
$
```

Ha a 2-es kimenetet átirányítjuk a null perifériára, megszabadulunk a hibaüzenet kimenetbe való beszúrásától:

```
$ wc 1.txt 2.txt 2>/dev/null
2 2 8 1.txt
2 2 8 total
$
```

A standard kimenet és hibakimenet egyszerre történő átirányítása

Vannak esetek, amikor mindkét standard kimenetet (kimenet és hibakimenet) külön-külön fájlba vagy ugyanabba a fájlba akarjuk irányítani. A külön-külön fájlba való irányításhoz mindkét kimenetnek meg kell adni az átirányítást:

```
$ sort lista.txt  > rendezett.txt  2> hibak.txt
$
```

A parancs lefutásának eredménye és hibái külön fájlokban lesznek rögzítve.

A két kimenet ugyanabba a fájlba való irányítására a megoldás a következő: a kimenetet egy fájlba irányítjuk és utána a hiba kimenetet a kimenetre (a két átirányítás sorrendje számít):

```
$ sort lista.txt  >  rendezett.txt  2>&1
$
```

Ilyenkor nyilván semmit sem látunk a terminálon, minden kiírt szöveg (akár eredmény, akár hiba) a `naplo.txt` nevű fájlba kerül, és onnan olvashatjuk el később. A parancs lefutását pedig meg kell várnunk (nincs készenléti jel csak ha a **sort** lefutott). Ilyenkor megtehetjük, hogy a feladatot a háttérben futtatjuk (hiszen amúgy sem látunk semmit), az alábbi indítással:

```
$ sort rendezetlen.txt > rendezett.txt 2>&1  &
[1] 5039
$
```

Így azonnal új paranccsal foglalkozhatunk, nem kell a **sort** végére várnunk.

Van olyan eset, amikor a két kimenetet ugyanabba a csővezetékbe szeretnénk irányítani, ezt az alábbi megoldással érjük el:

```
$ sort rendezetlen.txt 2>&1  |  cat  > rendezett.txt
```

```
$
```

Először a hibakimenetet a standard kimenetre irányítjuk, majd a standard kimenetet vezetjük egy csővezetékbe. Utána a csővezetékéket egy másik program, jelen esetben a **cat** bemenetére.

Egyszerű szövegszűrés

A Unix alapú rendszerek a rendszerrel kapcsolatos információkat, konfigurációs beállításokat szöveges állományokban tárolják. Ennek klasszikus példája az `/etc` könyvtárban található `passwd` nevű fájl, amelyik a rendszeren regisztrált felhasználók adatait tárolja. A fájl egy sora így néz ki:

```
lszabo:x:500:500:Laszlo Szabo:/home/lszabo:/bin/bash
```

1. a felhasználó rendszeren használt neve
2. jelenleg nem használt mező (a felhasználó jelszavának hash kódja állt itt régi rendszereken)
3. a felhasználó azonosítója (egy egész szám)
4. a felhasználó csoportjának azonosítója (ez is egy egész szám)
5. a felhasználó valódi neve
6. a saját könyvtára
7. a shell neve ami elindul, ha a felhasználó belép a rendszerbe

A `passwd` állományhoz hasonló, szöveges mezőket tartalmazó állományokból egy bizonyos mezőt (pl. ebben az esetben a felhasználók nevét) kivágni soronként például a **cut** nevű paranccsal lehet, bemutatását lásd a fejezet végén [42].

A `passwd` állományra alkalmazva, a felhasználók listáját így lehet kivágni:

```
$ cut -f 1 -d : /etc/passwd
root
bin
daemon
adm
lp
sync
 ...
```

Ha szükségünk lenne például az első 5 felhasználónévre rendezett sorrendben, azt így kapnánk meg:

```
$ cut -f 1 -d : /etc/passwd | sort | head -5
adm
apache
avahi
```

```
beaglidx
bin
```

A fejezetben használt parancsok

A cut parancs

Szövegmezőket vagy karaktereket vág ki szövegsorokból.

Használat:

`cut` -c *`lista`*... [*`fájl(ok)`*]

`cut` -f *`lista`*... [-d karakter] [*`fájl(ok)`*]

Fontosabb opciók:

`-c lista` Megadjuk azoknak a karaktereknek a sorszámát a bemeneti sorokból, amelyeket ki akarunk választani. Pl. -c 6 vagy -c 6-8 vagy -c 6,9-12,10-22 .

`-d kar` Megadjuk azt az elválasztó karaktert, amelyiket a -f opcióval együtt, szövegmezők kiemelésére használunk. Ha nem adunk meg semmit, az alapértelmezett elválasztó a tabulátor karakter.

`-f lista` Megadjuk azoknak az elválasztó karakterrel elválasztott mezőknek a listáját amelyet a kimenetre akarunk írni. Pl.: -f 2 (második mező) vagy -f 7,8 a hetedik és nyolcadik mező.

Leírás:

A cut a bemenet minden sorából ugyanazokat a mezőket vagy karaktereket vágja ki és azokat soronként küldi a kimenetre. Alapvetően két feladatot oldhatunk meg vele: karaktereket vágunk ki a bemeneti sorokból vagy mezőket, ez utóbbiak esetében meg kell adnunk egy elválasztó karaktert. Bemenetként a paraméterként megadott állományokat vagy a standard bemenetet használja ha nem adunk meg állományt. Pl.:

```
$ echo a b cd | cut -f 2 -d " "
b
$ echo abcdef | cut -c 2,4
bd
$ echo ab cd ef | cut -f2 -d " "
cd
$
```

Megjegyzés:

POSIX standardot megvalósító rendszereken kezeli a több byte-os karaktereket is (pl. UTF-8). Ilyenek a modern Linux disztribúciók, ahol pl.:

```
$ echo á ó ű | cut -f 2 -d " "
```

```
ó
$
```

az ó karaktert írja ki.

5. fejezet - A héj változói

A héj változóinak típusa

A héj az adatok, információk tárolására változókat használ. A héj változói három kategóriába sorolhatóak:

1. saját változók, amelyekben működéséhez szükséges információkat tárol, ezek egy héjra jellemzőek, és a felhasználó szempontjából automatikusan jönnek létre,

2. környezeti változók, amelyek például az operációs rendszerről adnak információt, és minden indított új héj megkap,

3. a felhasználó vagy futó szkript által létrehozott változók.

Először áttekintjük a változók létrehozásának módját a felhasználó által. Ezeket paramétereknek is nevezik, megkülönböztetve a héj saját illetve környezeti változóitól.

Változók létrehozása

Mivel a héj parancssorai szövegekből épülnek fel, a változók nevét és értékét is egyaránt sztringekben tárolja. A név-érték hozzárendelést az = jel segítségével adjuk meg. Például a:

```
$ szin=piros
$
```

sor végrehajtása során létrejön az a `szin` változó `piros` értékkel. A változónevek az angol Abc betűiből (kis és nagybetű közti eltérés számít) valamint számjegyekből és az _ aláhúzásjel karakterből állhatnak, de nem kezdődhetnek számjeggyel.

Az = jel közvetlenül a változó neve után kell elhelyezkedjen, az érték pedig közvetlenül az = jel után, így hibás az alábbi hozzárendelés:

```
$ szin= piros
-bash: piros: command not found
$
```

mert így közvetlenül az = jel után a szóköz áll.

A változó létrehozása dinamikus: akkor jön létre amikor értéket adunk neki, nincs szükség előzetes deklarációra. Amikor a létrejött változót később héj parancsok argumentumaiban használjuk, egy `$` jellel és a változó nevével hivatkozunk rá (a héj készenléti jeleként továbbra is a `$` karaktert használjuk az alábbi példákban, ne tévesszük ezt össze a nevek előtti $ karakterrel). Tehát `$szin` lesz a hivatkozott változó, és legkönnyebben az **echo** paranccsal (részletes bemutatását lásd a következő pontban) tudjuk kiírni:

```
$ echo $szin
```

```
piros
$
```

A változók tehát sztringeket tartalmaznak, így megtörténhet az is, hogy üres sztringet tartalmaznak, tehát nincs értékük. A Bash alap beállításban a nem definiált változóneveket is üres sztringként értékeli ki (ez a viselkedés megváltoztatható Bash opciók beállításával, illetve tesztelhető, hogy létezik-e a változónév, lásd alább [48]).

```
$ echo $NEMLETEZO

$
```

A változók értékét idézőjelek közé is tehetjük, ha a változó értéke olyan sztring amelyik elválasztó karaktert is tartalmaz, akkor ezt kell tennünk. Például, ha egy változó értéke a `piros alma` karaktersor, akkor ezt így adjuk meg:

```
$ valt="piros alma"
$
```

Egyébként a héj csak a piros sztringet rendelné a változóhoz és úgy értelmezné, hogy az alma egy következő parancs.

Megjegyzés: a másik megoldás a vissza-per szekvencia használata:

```
$ valt=piros\ alma
$ echo $valt
piros alma
$
```

Változó nevet a héjból az **unset** paranccsal törölhetünk:

```
$ szin=piros
$ echo $szin
piros
$ unset szin
$ echo $szin

$
```

Az idézőjelek használata

Háromféle idézőjelet használhatunk a héj programokban: `"` (idézőjel, macskaköröm angolul: *double quote*), `'` (aposztróf, *single quote* vagy *forward tick*) és `` ` `` (fordított idézőjel, *backtick*).

Az idézést az első kettővel valósítjuk meg, ezzel a művelettel levédjük a létrehozott sztringet, így a speciális karakterek egy részének és a kulcsszavaknak a jelentése megszűnik.

A fordított idézőjel a héj parancs helyettesítő műveletét vezeti be régi jelölés szerint, részletesen lásd A parancs helyettesítés [53] alpontot alább. Ezzel tulajdonképpen nem "idézünk".

A kettős idézőjel

A " macskakörmöt vagy kettős idéző jelet sztringek deklarálására használjuk:

```
$ uzenet="Hosszabb szöveg"
$ echo $uzenet
Hosszabb szöveg
$
```

Ezeket a sztringeket, a héj használat előtt még átírja, amennyiben változókra hivatkozunk a `$` jel segítségével:

```
$ szin="piros"
$ echo "Hosszabb $szin szöveg"
Hosszabb piros szöveg
$
```

A `$` karakterrel bevezetett átírást változó vagy paraméter kifejtésnek nevezzük (*variable or parameter expansion*).

Az átírás nem csak a `$` karakter megjelenésekor történik meg, hanem a következő esetekben is:

- `` ` `` (fordított idézőjel) Ezzel a karakterrel *parancs helyettesítést* vezetünk be (lásd alább), amellyel egy sztring tartalmába be lehet szúrni egy parancs kimenetét (mindazt amit a parancs a standard kimenetre ír). Ugyanez történik a `$(` karakter kombináció megjelenésekor - tehát a `$` karakter is vezethet be parancs helyettesítést. Megjegyezzük azt is, hogy `$((` szekvencia, aritmetikai kiértékelést vezet be (lásd Aritmetikai kiértékelés ... [111] című alfejezetet).

- `!` (felkiáltójel) A felkiáltójellel előzőleg végrehajtott parancsokra hivatkozhatunk, ez a héj un. *history expansion* opciója, amely alapszinten be van kapcsolva ha a héj interaktívan fut. Pl. `!!` sorozat az utolsó parancssort helyettesíti be, `!-5` az öt paranccsal megelőzőt. Tehát csak akkor történik ez meg, ha ez az opció aktív. Ha a héj egy szkriptet futtat, akkor általában ez az opció nincs bekapcsolva, tehát az átírás nem történik meg. Ebben a tananyagban nem foglalkozunk ezzel az opcióval (részletekért lásd BashRef2010 [184] vagy Newham2005 [184] , 2. fejezet), de ne feledjük, hogy az alábbi egyszerű parancs:

  ```
  echo "Szia!"
  ```

  interaktív munka folyamán hibát ad, míg végrehajtott szkriptben simán lefut - ilyenkor ez az opció nem aktív. Ha interaktív munka folyamán szükségünk van a felkiáltójelre egy sztringben, akkor külön aposztróffal megadható:

  ```
  $ echo "Szia"'!'
  Szia!
  $
  ```

- `\` (vissza-per jel) Ha a per jel után `$` , `` ` `` , `"` , `\` vagy újsor karakterek vannak, akkor ezek jelennek meg literálisan, és a `\` jel elmarad. A többi karakterek esetében, pl. (`\a` vagy `\.`) a per jel megmarad a sztringben. Pl. a:

```
echo "\$"
```

parancs literális `$` jelet ír ki, a

```
echo "\."
```

pedig a `\.` szekvenciát.

Az aposztróf

Az `'` aposztróf segítségével olyan sztringeket deklarálunk, amelyekben minden egyes karakter saját magát jelenti, és betű szerint lesz értelmezve, tehát a héj semmiféle átírást nem végez. Így az alábbi kiíráskor:

```
$ $ echo 'ez itt egy $jel'
ez itt egy $jel
$
```

valódi `$` karakter kerül a kimenetre. A héj nem ír át semmit a sztring belsejében, úgy dolgozik vele ahogy leírtuk.

Ha egy sztringben idézőjel vagy aposztróf fordul elő, megadhatjuk azokat vissza-per jelöléssel, mint:

```
$ echo \"piros\"
"piros"
$ echo \'piros\'
'piros'
$
```

Egy kicsit bonyolultabb a helyzet, ha idézőjelek vagy aposztrófok fordulnak elő olyan szövegben, amelyet egyébként is idézőjelbe kell tenni a változók kiértékelése miatt. Ha pl. ki akarjuk írni a következő szöveget:

```
a piros a "kiválasztott" szín
```

akkor megtehetjük többféleképpen:

```
$ echo "a $szin a \"kiválasztott\" szín"
a piros a "kiválasztott" szín
$ echo "a $szin a "'"'kiválasztott'"'" szín"
a piros a "kiválasztott" szín
$
```

A második változatban több sztringet ragasztunk össze, és kihasználjuk azt, hogy az idézőjel és aposztróf mint egy karakteres sztring megadható egymás "idézésével":

```
$ echo "'"
'
$ echo '"'
"
$
```

Előfordulnak esetek, amikor egy parancssori sztringet darabokból kell összeragasztani.

A {} jelölés

A sztringekben a héj nem tudja mindig a változó nevét meghatározni a változó nevek egyszerű használatával, például alábbi esetben, ha adott a `szin` változó, amelynek értéke "piros", a "pirosalma" szöveget akarjuk kiírni:

```
$ echo "$szinalma"
```

meddig tart a változónév? a héj úgy fogja értelmezni, hogy egy új változó nevet használunk, a `szinalma` nevűt, ezért használunk egy jelölési módot ami ezt egyértelművé teszi. Ilyenkor a változó nevét jelentő karakterláncot `{ }` kapcsos zárójelekkel vesszük körül. Így már egyértelmű a változó nevének megadása:

```
$ echo "${szin}alma"
```

Ez a változónevek megadásának általános szintaxisa a parancsoron, ha hivatkozunk a változó értékére. A `{}` zárójelek elhagyhatóak, ha a sztringben a változó nevét nem betű, számjegy vagy alulvonás jel követi.

A példa azt is szemlélteti, hogy két sztring összefűzését úgy oldjuk meg, hogy egyszerűen egymás után írjuk őket a parancsorra, és a héj elvégzi a műveletet akkor, amikor kiértékeli a parancssort.

A `{}` jelölés, amely a változók kifejtésének egyik módja, további műveleteket is lehetővé tesz a kapcsos zárójelek szerkezet között különböző operátorokat használva. Így lehetőséget ad a változó kezdeti értékének tesztelésére és beállítására:

5.1. táblázat - Változók kezdeti értékét kezelő operátorok

Operátor	Jelentése
`${valtozo:-piros}`	Ha a változó nem létezik, nem ad értéket a változónak, de visszatéríti a "piros"-at.
`${valtozo:=piros}`	Ha a változó nem létezik, értéket a változónak és visszatéríti a "piros"-at.
`${valtozo:? hibaüzenet}`	Ha a változó nem létezik a héj program hibaüzenettel leáll, ha a héj nem interaktív (végrehajtott szkriptek esetében ez az implicit helyzet).
`${valtozo:+érték }`	A változó létezésének tesztelésére használjuk: a kifejezés visszatéríti az "érték" sztringet, ha a változónév létezik és

Operátor	Jelentése
	nem üres sztringet tartalmaz, egyébként az üres sztringet kapjuk vissza.

Az alábbi kis programban (`szin.sh`) nem adunk értéket a `szin` változónak, így első használatkor automatikusan felveszi azt:

```
#!/bin/bash
echo ${szin:=piros}
echo $szin
```

, elindítva a program kétszer fogja a "piros" sztringet kiírni:

```
$ bash szin.sh
piros
piros
$
```

A {} jelölés megengedi más, igen hasznos sztring műveletek elvégzését is a Bash és Korn héjakban, az alábbiakat igen gyakran használjuk:

5.2. táblázat - Sztring operátorok I.

Operátor	Jelentése
`${#valtozo}`	A változóban tárolt sztring hosszát adja vissza.
`${valtozo:n:m}`	A változó rész sztringjét adja vissza az n.-dik karaktertől kezdődő m hosszú sztringet; az első karakter indexe 0. Ha a második paraméter elmarad, akkor a sztring végéig tartó karaktersort adja vissza.

Például:

```
$ valtozo=abcdef
$ echo ${#valtozo}
6
$ echo ${valtozo:2:3}
cde
$
$ echo ${valtozo:3}
def
$
```

A {} jelöléssel kapcsolatban további műveleteket találhatunk a Műveletek karakterláncokkal [114] fejezetben.

Az echo parancs

Változók és sztringek kiírására többnyire az **echo** parancsot használjuk, amely kiírja saját argumentumainak listáját és utána automatikusan egy új sor karaktert. Használata:

`echo` [-ne] *`sztring`*...

Például:

```
$ echo ez egy üzenet
ez egy üzenet
$ echo ez egy          üzenet
ez egy üzenet
$
```

Látható, hogy a paraméterként kapott sztringeket írja ki. Két kapcsolója van:

`-n` Nem ír új sor karaktert.

`-e` Engedélyezi a következő speciális karakterek értelmezését a sztringekben:

\a riadó
\b egy karakter törlése visszafelé
\f lapdobás
\n új sor
\r kocsi vissza
\t vízszintes tabulátor
\\ backslash
\nnn a karakter ASCII kódja nnn (oktálisan)
\xnn a karakterkódot hexadecimálisan lehet megadni, pl. \x32

Az **echo** a leggyakrabban használt parancs a héjprogramokból való kiírásra.

A Bash héj rendelkezik egy **printf** paranccsal is, amelyet ritkábban, abban az esetben használunk, ha bonyolultabban kell formázni a kimenetet. Meghívása az alábbi:

`printf` *`formázósztring sztring(ek)`*...

A formázó sztring nagyjából megegyezik a C nyelvből ismert printf szintaxisával, a kiírt argumentumokat pedig a formátum után, egy listában kell felsorolni, elválasztójelek nélkül:

```
$ printf "%s %.2f  %s %d\n" "Az eredmény:" "0.733"  "kerekítve:"  7
Az eredmény: 0.73  kerekítve: 7
$
```

Pontos használatával kapcsolatban lásd: `man 1 printf` .

A héj saját változói

A héj igen nagy számú belső változót tartalmaz, amelyek megadják a rendszer által biztosított környezet tulajdonságait, illetve konfigurációs lehetőségeket biztosítanak a felhasználónak.

Amint említettük, ezek 2 kategóriába sorolhatóak, a saját és környezeti változók kategóriájába. Nevük konvenció szerint nagybetűs. A saját változókat a **set** , a környezeti változókat a **printenv** paranccsal listázhatjuk.

Ezek közt vannak klasszikusak, amelyek szinte minden héjban megtalálhatóak ugyanazzal a névvel, és vannak olyanok, amelyek csak bizonyos héjakra, pl. a Bash-re jellemzőek. Az interaktív munka szempontjából megemlítünk néhányat a fontosabbak közül.

A héj készenléti jelének vagy *prompt* sztringjének kiírását egy `PS1` nevű változó vezérli. Ennek értékét írja ki a héj, de úgy, hogy a benne levő vezérlő karaktersorozatokat átírja. A készenléti jel jelezheti, hogy éppen melyik a munka könyvtárunk, mi a gép neve amin dolgozunk, vagy mi a felhasználó nevünk. Ezeket az információkat vezérlő karakter szekvenciákkal írhatjuk be a `PS1` változóba. Például a `\h` a gép rövid nevét jelenteni, `\u` a felhasználónevet, `\w` pedig a munkakönyvtár nevét. Így a`PS1="\u@\h $"` értékadás után a prompt a felhasználó és a gép nevét írja ki, köztük egy @ jellel. Az összes használható átírást a Bash esetében megtaláljuk a kézikönyv PROMPTING című fejezetében.

A `PS2` a másodlagos készenléti jel, ez általában egy > karaktert ír ki. Akkor jelenik meg, ha egy befejezetlen sort írunk a parancssorra, például megnyitunk egy idézőjelet és nem zárjuk azt:

```
$ msg="Ez egy két sorban
> beírt üzenet"
$ echo $msg
Ez egy két sorban beírt üzenet
$
```

A `PS3` és `PS4`-el később fogunk találkozni ebben a könyvben.

Ugyanilyen belső változó a már említett `PATH`, a végrehajtható programok elérési út listáját tartalmazó változó vagy az `EDITOR`, amely a preferált szövegszerkesztő nevét tartalmazza.

Az környezeti változók és az export parancs

A héj saját változói közül egyesek csak a futó héj számára lesznek láthatóak, mások átadódnak a héjból induló parancsoknak vagy új indított héj (lefuttatott szkript) számára is, ezeket környezeti változóknak nevezzük. A már említett `PATH` változó átadódik minden új indított programnak illetve héjnak, a `PS1` pedig nem.

Az **export** parancs beírja a héjváltozót a környezeti változók listájába. Használata:

`export` [-nfp] *`VÁLTOZÓ`* [*`=ÉRTÉK`*]

Így lehet például a `PATH` változó értékét megváltoztatni. Amennyiben azt szeretnénk, hogy a munka könyvtárunk (a **.** könyvtár) is legyen a keresett könyvtárak közt a parancsok elindításánál, az alábbi parancsokat kell végrehajtani:

```
PATH=$PATH:.
export PATH
```

vagy egyszerűbben:

```
export PATH=$PATH:.
```

Ha minden bejelentkezéskor szükség van erre, akkor a parancsot be kell írni a `.bash_profile` vagy `.bashrc` fájlba (ha a Bash-el dolgozunk).

Az első sor átállítja (újra deklarálja) a `PATH` változót: a régi mögé odailleszti az elválasztó karaktert (:) és utána a pillanatnyi munkakönyvtár nevét, majd az export héj utasítással beírja a környezeti változók közé, úgy, hogy a héj által indított folyamatok is örököljék.

Munkakönyvtár a `PATH` változóba

A `.` könyvtárat root felhasználó esetében nem ajánlott a `PATH` listába betenni biztonsági okok miatt.

Az **export** `-f` kapcsolóval függvényneveket ír az export listába, `-n` kapcsolóval törli a beírt neveket. Az `-p` opcióval kilistázza azokat a neveket, amelyeket a futó héj exportál.

A héj speciális változói vagy paraméterei

A héj több speciális változót vagy paramétert tartalmaz, amelyek automatikusan jönnek létre. Többnyire olyan információkat fognak tartalmazni amelyek minden programra jellemzőek és gyakran kerülnek felhasználásra. Néhányat megemlítünk itt, a többit pedig az anyaggal való haladás közben.

Ezeknek a változóknak nem tudunk felhasználóként értéket adni, azok is automatikusan állnak elő.

Nevük általában egy speciális karakterből áll, így nagyon gyorsan be lehet írni őket. Például a parancssor argumentumait az `1`, `2`, `3`, ..., `9` nevű változókkal lehet elérni, így `$1`, `$2`, ... `$9` -ként hivatkozunk reájuk. A parancssoron található argumentumok számát pedig a `#` változó tartalmazza. Ezek a kapcsos zárójeles szintaxissal is elérhetőek. Az alábbi szkript (`arg.sh`) például kiírja első és harmadik argumentumát, valamint az argumentumok számát:

```
#!/bin/bash
echo $1
echo ${3}
echo a parancssor $# argumentumot tartalmaz
```

, végrehajtva:

```
$ bash arg.sh elso masodik harmadik
elso
harmadik
a parancssor 3 argumentumot tartalmaz
$
```

Az következő szkript (`valtozok.sh`) az első paraméter hosszát és első karakterét írja ki a {} jelölés operátorait használva:

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

```
#ellenőrzés
str=${1:? nincs paraméter}

echo a paraméter hossza ${#str}
echo első karaktere: ${str:0:1}
```

, lefuttatva:

```
$ bash valtozok.sh
valtozok.sh: line 4: 1:  nincs paraméter
$ bash valtozok.sh piros
a paraméter hossza 5
első karaktere: p
$
```

Még két speciális változót említünk itt meg: az `@` és a `*` változókat. Mindkettő a parancssor összes argumentumát tartalmazza, azzal a különbséggel, hogy az `@`-ban minden argumentum külön sztringént jelenik meg, a `*` -ban pedig egy sztringként, a héj implicit elválasztó karakterével elválasztva (ez a szóköz, tabulátor vagy újsor). Ha a parancssor argumentumai például az `alpha`, `beta` és `gamma` sztringek:

```
bash args.sh alpha beta gamma
```

akkor a "$@" jelölés az: `"alpha" "beta" "gamma"` sztringet tartalmazza, a "$*" pedig a: `"alpha beta gamma"` sztringet. Ha egy szkriptben **echo**-val kiírjuk ezeket, nem látunk semmi különbséget, de abban az esetben, ha paraméterként továbbadjuk őket egy másik struktúrának, vannak különbségek, amint azt látni fogjuk.

A parancs helyettesítés

Parancs helyettesítésnek azt a műveletet nevezzük, amely során a héj lefuttat egy parancsot, és a parancs kimenetét (a standard kimenetet) mint egyetlen karaktersort egy változóhoz rendel.

Két jelölést használunk erre, egy hagyományosat és egy modernet. Hagyományos jelölés:

```
változó=`parancssor`
```

, míg a modern:

```
változó=$( parancssor )
```

Mindkét jelölés ugyanazt az eredményt idézi elő. Amint azt tapasztalni fogjuk, a `$()` jelölés jobban olvasható, ezért ezt ajánlott használni, de a gyakorlatban mindkettővel ugyanolyan mértékben találkozhatunk. A POSIX standard az utóbbi, `$()` jelölés alkalmazását javasolja.

Az alábbi példában az első parancssor **echo** kimenete nem a kimenetre kerül, hanem az `uzenet` változóba:

```
$ uzenet=` echo "ez a kiírni való" `
$ echo $uzenet
ez a kiírni való
$
```

Ugyanazt érjük el az alábbi jelöléssel:

```
$ uzenet=$( echo "ez a kiírni való" )
$ echo $uzenet
ez a kiírni való
$
```

A fenti példában az **echo** kimenete egy 1 soros karakterlánc. Amennyiben nem így van, és a kimenet több sorból áll, a héj átírja az új sor karakter elválasztót egy szóköz karakterré. Ilyenkor egy karakterláncokból álló, szóközzel ellátott listát rendel a változóhoz. Például az **ls -1** (úgy listáz, hogy minden egyes fájlnév külön sorba kerül) parancs kimenetét változóba írva egy fájlnév listát kapunk:

```
$ ls -1
alma.sh
hello.sh
phone.sh
$ lista=$( ls -1 )
$ echo $lista
alma.sh hello.sh phone.sh
$
```

A parancssor átírása

Amint azt az idézőjel használatánál láttuk, a héj a parancssort végrehajtás előtt "átírja": ha vannak benne olyan szerkezetek vagy metakarakterek amelyeket a végrehajtás előtt helyettesíthet, akkor megteszi azt.

A metakarakterek olyan karakterek, amelyek mást jelentenek a héj számára mint betű szerinti jelentésük.

Az alábbiakkal találkoztunk eddig:

Metakarakter	Jelentése
*	A munkakönyvtárban levő fájl neveket írja be helyette.
~	A saját könyvtár elérési útját jelenti (pl. `/home/lszabo` -t ír be a hullámvonal helyett).
$	A közvetlenül utána következő karakterláncot változónévnek tekinti.

A metakaraktereket lehet literális értelemben is használni a parancssoron, de ekkor vissza-per jelöléssel kell megadni vagy aposztróf közé kell zárni őket. Egy `*`-ot így írunk ki a terminálra:

```
$ echo \*
*
```

```
$ #vagy:
$ echo '*'
*
$
```

A héj bonyolultabb folyamatok közti összefüggéseket is létrehoz amikor a parancssort kiértékeli, erre később még visszatérünk.

A here document átirányítás ("itt dokumentum")

Ha tudjuk, hogyan dolgozik a héj a változókkal, megismerhetünk még egy átirányítási műveletet. Ez nem egy parancssorra, hanem egymás után következő szöveg sorokra vonatkozik, ezért leginkább héjprogramokban használjuk. Segítségével a héj program bemenete átirányítható arra a fájlra amelyben a héj program található. Használati módját az alábbi minta adja meg:

```
parancs  <<  VEGE
itt bármilyen
       szöveg   állhat
VEGE
```

A parancs végrehajtása után (ez bármilyen Unix parancs lehet amelyik a standard bemenetről olvas), a héj a parancs bemenetét a saját programszövegére irányítja (az "itt bármilyen szöveg állhat" sorra vagy sorokra), és a parancs addig olvas innen, amíg egy sor elején ugyanarra a karakterláncra akad, amely a << jelek után áll. Itt abbahagyja az olvasást, és a héj végrehajtja a következő soron levő parancsot. Az alábbi héj program egy sorozatlevélhez generál szöveget:

```
#!/bin/bash

cimzett=${1:? hiányzik az elő paraméter, a címzett neve}

cat << VEGE

Könyvtári figyelmeztető

Kedves ${cimzett}

kérjük hozza vissza a kölcsönkért könyveket.
Üdvözlettel,
Könyvtáros

VEGE
```

Végrehajtásakor az alábbit látjuk:

```
$ bash mintalevel.sh Péter

Könyvtári figyelmeztető
```

```
Kedves Péter

kérjük hozza vissza a kölcsönkért könyveket.
Üdvözlettel,
Könyvtáros
$
```

Miközben a szöveg átkerül az őt olvasó programhoz (a **cat**-hez ebben a példában), a héj ugyanúgy átírja, mintha macskakörömmel körülvett sorok lennének: tehát amennyiben a szövegben változók is vannak, azok átírásra kerülnek. A *here document* átirányítás rendkívül hasznos, ha több példányban kell olyan szövegeket generálni, amelyekben változó mezők vannak.

Fontos: a `VEGE` szót tartalmazó lezáró sorban akármilyen szócska állhat, de a sornak csak ezt az egyetlen szót kell tartalmaznia a sor elején.

Megemlítjük, hogy a shell rendelkezik egy *here string* nevű átirányítással is, amellyel egyetlen sztringet lehet egy parancs standard bemenetére küldeni. Az átadás előtt a shell a sztringet átírja. A használt átirányító jel a `<<<` , tehát így vágjuk ki a `b` karaktert az `"a b c"` sztringből::

```
$ cut -f 2 -d " " <<< "a b c"
b
$
```

A . és a source parancs

A **.** (*dot*) parancs beolvassa és azonnal végrehajtja azokat a shell parancsokat amelyek az argumentumában megadott állományban vannak. Mindezt az éppen futó héjban végzi el, nem indít új héjat, ezért a külső állományból beolvasott változók és definíciók (pl. a függvények [105]) az aktuális héjban maradnak.

Tekintsük a `def.sh` fájlt, amelynek tartalma:

```
a=2
export LANG=hu_HU.UTF-8
```

Ha végrehajtjuk a szkriptet egy új héjjal, a definíciók az új héjban jönnek létre, és lefutás után nem lesznek jelen a hívó héjban:

```
$ bash def.sh
$ echo $a

$
```

Ez történik minden normál szkript futtatásakor.

Ezzel szemben, a **.** paranccsal végrehajtva:

```
$ unset a
$ echo $a

$ . def.sh
$ echo $a
2
$
```

Az `a` változó az aktuális héjban jön létre. Ugyanezt a feladatot végzi a **source** parancs is:

```
$ source def.sh
$ echo $a
2
$
```

Példaprogramok

Az alábbi szkriptek egész egyszerű programok, gyakorlatilag csak változókat használunk benne ismétlődő szerkezetek nélkül.

Írjunk egy szkriptet amelyik kiírja egy állomány N.-dik sorát!

Az alábbi szkript (`printline.sh`) a feladatot a **head** és **tail** parancsokkal oldja meg, kilistázva a fájl első N sorát a **head** segítségével, utána ennek a listának az utolsó sorát. A paramétereket a `${}` szerkezettel saját változóba írjuk át használat előtt. Ennek általában az az előnye, hogy sokkal olvashatóbb szkriptet kapunk amikor felhasználjuk őket.

A **tail** parancs a fájl végén levő sorokat listázza. A leggyakrabban használt opciója egy szám, amely a fájl végéről listázott sorok számát tartalmazza, ennek implicit értéke 10.

Az alábbi kis szöveget felhasználva:

1 elso
2 masodik
3 harmadik
4 negyedik
5 otodik
6 hatodik
7 hetedik

így listázhatjuk az utolsó 3 sort:

```
$ tail -3 lista.txt
5 otodik
6 hatodik
7 hetedik
$
```

A szkript paramétereként kapott fájlneveket mindig idézőjelbe tesszük, mert tartalmazhatnak elválasztó karaktereket, pl. szóközt:

```
#!/bin/bash
# kilistázza egy fájl n.-dik sorát
#
# paraméterek:
#   $1 - fájl név
#   $2 - egész szám
# paraméterek átvétele
file=${1?: hiányzik a fájlnév}
N=${2?: hiányzik a sorszám}
head -$N "$file" | tail -1
```

a végrehajtás (első alkalommal hibásan):

```
$ bash printline.sh
printline.sh: line 9: 1: : hiányzik a fájlnév
$ bash printline.sh lista.txt 6
6 hatodik
$
```

A program csak helyes paraméterekkel működik, nem ellenőrizzük, hogy az N egész szám.

Megjegyzés: a feladatot sokkal egyszerűbben meg lehet oldani a **sed** paranccsal (lásd 9. fejezet [118]):

```
$ sed -n 6p lista.txt
6 hatodik
$
```

Egy másik kis feladat: szúrjunk be egy szöveges fájl első sorának végére egy sztringet, és mentsük el a fájlt más név alatt!

A szkriptben ugyancsak a **head [38]** és **tail** parancsokat használjuk.

A fájl végi sorok listázása megadható a **tail** `-n` opciójával is:

```
$ tail -n 3 lista.txt
5 otodik
6 hatodik
7 hetedik
$
```

Ha az opció paramétere előtt `+` jel van, akkor jelentése: az n.-dik sortól a fájl végéig, ezt fogjuk felhasználni szkriptünkben:

```
$ tail -n +6 lista.txt
6 hatodik
7 hetedik
```

Az eddig bemutatott egyszerű eszközökkel így oldható meg a feladat:

```
#!/bin/bash
# beszúr egy sztringet egy szöveges fájl első sorának végére
# Paraméterek:
#   $1 - eredeti fájl
#   $2 - új fájl
#   $3 - szó
#

#paraméterek átvétele:

f1=${1:? első fajlnév hiányzik}
f2=${2:? második fajlnév hiányzik}
szo=${3:? hiányzik a szó}

#az első fájl első sorát egy változóba írjuk
elso=$( head -1 "$f1")

#hozzáadjuk a szó paramétert
elso="${elso}${szo}"

#létrehozzuk az új fájl első sorát
echo "$elso" > "$f2"

#f2 -be másoljuk a többi sort f1-ből
tail -n +2 "$f1" >> "$f2"
```

Végrehajtva:

```
$ bash insfirstline.sh lista.txt lista1.txt " vege"
$ cat lista1.txt
1 elso vege
2 masodik
3 harmadik
4 negyedik
5 otodik
6 hatodik
7 hetedik
```

Ez a feladat is egyszerűen fejezhető ki a **sed [118]** paranccsal:

```
$ sed '1s/$/ vege/' lista.txt
1 elso vege
2 masodik
3 harmadik
4 negyedik
5 otodik
6 hatodik
7 hetedik
$
```

6. fejezet - Vezérlő szerkezetek

Vezérlő szerkezetek

A héjprogramozásban - a klasszikus programnyelvekhez hasonlítva – egy "végrehajtási egységnek" egy kiadott parancs, illetve parancssor felel meg. Ha parancssorról beszélünk láttuk, hogy ennek szerkezete egy parancsnál több is lehet, például tartalmazhat akár 2 parancsot is csővezetékkel összekötve. Ezt a vezérlő szerkezetek megadásánál **parancssor**-nak fogjuk jelölni.

A parancssori szerkezetek végrehajtásának sorrendjét vezérelhetjük. Erre a héj különböző, úgynevezett *vezérlő szerkezetet* használ.

A strukturált programozás elméletéből tudjuk, hogy egy program végrehajtásához egy programozási nyelvnek a változók kezelésén kívül legalább a döntéshelyzeteket (if szerkezetek) illetve a ciklusok kezelését kell megoldania. Ezzel a kettővel bármilyen parancs végrehajtási sorrend megszervezhető. A héj vezérlő szerkezeteit eredetileg az ALGOL68 nyelvet követve alakították ki.

Az igaz/hamis feltétel a héjprogramozásban

A szerkezetek kialakítása miatt szükségünk van az igaz illetve hamis feltétel fogalmára. Mivel a héj alatt programok, illetve parancsok futnak, a feltételt ezek lefutása után nyert, programok által szolgáltatott **kilépési érték vagy állapot** (*program exit status*) jelenti. Ez a C programok kilépése után, az `int main()` függvény által visszaadott egész szám: ennek értékét a C-ben meghívott `void exit(int n)` függvény paraméterében adjuk meg.

Ez az érték közvetlenül a parancs befejezése után, a parancs által futtatott utolsó program kilépési értékéből áll, amely bekerül a héj speciális `?` nevű változójába (és `$?`-ként érhető el). Ezt az értéket az **echo** paranccsal tesztelni is lehet:

```
$ touch 1.txt
$ ls -l 1.txt
-rw-rw-r-- 1 lszabo lszabo 0 Oct 12 10:00 1.txt
$ #végrehajtás után kiírhatom a ? változót amit az ls állított be
$ echo $?
0
$ #ha elvégzünk egy hibát adó ls-t
$ ls -l nemletezik
ls: nemletezik: No such file or directory
$ echo $?
2
$
```

Látható, hogy jó lefutás esetén a visszatérített érték 0, míg hiba esetén eltér 0-tól, jelen esetben 2, de minden esetben egész szám (a számnak általában van pontos jelentése, és azt hogy milyen hibára utal az illető parancs kézikönyv lapján lehet megnézni. Például az ls kézikönyve erről ennyit ír: "*Exit status is 0 if OK, 1 if minor problems, 2 if serious trouble*"). Tehát a tesztek

számértéke pont a fordított lesz a C, Java nyelveknél megszokottnál: itt az "igaz" feltételt a `?` változó `0` értéke adja.

A POSIX szabvány standardizálja a programok által visszaadott értékeket, így ajánlatos ezeket akár saját C programjaink írásánál betartani. Ez az érték általában egy 255-nél kisebb előjel nélküli egész szám.

6.1. táblázat - Az exit kilépési értékei

? változó értéke	**Mit jelent**
0	Sikeres parancsvégrehajtás.
>0	Hiba történt a parancssor kiértékelésénél (akár az átirányításokban, behelyettesítésekben is lehet ez). Az eseteket lásd alább:
1 - 125	A program lefutott, de hibával lépett ki. A kód a hibát jelenti, minden programnál más lehet a jelentése.
126	A program létezik, de nem lehetett végrehajtani (pl. nincs jogunk hozzá, vagy nem egy végrehajtható bináris kód).
127	A parancs nem létezik (nem lehetett a `PATH` elérési útjai szerint megtalálni)
> 128	A program jelzéssel állt le (pl. ha a **cat** futása közben lenyomjuk a Ctrl-C - t, a program 130-at térít vissza). Megjegyzés: a 128-as kód jelentése nincs standardizálva.

Az && , || és ! szerkezetek

Ezek a szerkezetek rendkívül egyszerűen, egyetlen parancssoron alkalmazhatóak a meghívott parancsok között. Amint az sejthető, az `&&` egy *és*, míg a `||` egy *vagy* típusú feltételt fog megoldani az egymásutáni parancsvégrehajtásban.

Az `&&`, `||` , illetve `!` szerkezeteket rövidre záró operátoroknak is nevezzük (*short circuit operators*), mert a programozási nyelvekhez hasonlóan, a kiértékelésük leáll, ha egy logikai szerkezet végeredménye valamelyik (pl. az első) parancs végrehajtása után előre tudható.

Az alábbi módon használhatóak:

```
$ touch 1.txt
$ #ha hibával fut az ls parancs, kiírok egy üzenetet
$ ls 1.txt && echo 'van ilyen fájl'
1.txt
van ilyen fájl
$ #ha nem sikerül az ls-t lefuttatni, kiírok egy üzenetet
$ ls 1.txt || echo 'uzenet'
1.txt
$
```

Az `&&` szerkezet végrehajtja a parancssoron következő parancsot, amennyiben az előző lefutott parancs "jól" futott le, tehát igaz visszatérített értékkel zárult. Tulajdonképpen úgy is

tekinthetjük, hogy az `&&` kiértékeli a két lefutott parancs által visszatérített értéket. Logikailag a másodikat akkor értelmes végrehajtani, ha az első parancs jól lefutott. A `||` esetében viszont a második parancsot nincs értelme végrehajtani, ha az első jól futott le, amint az a második példából (`ls 1.txt || echo 'uzenet'`) látszik.

Ezzel a két egyszerű szerkezettel akár if-else struktúra is kialakítható egyetlen parancssoron két parancs között:

```
$ ls 1.txt && echo 'van ilyen fájl' || echo 'nincs ilyen fájl'
1.txt
van ilyen fájl
$ ls nemletezik && echo 'van ilyen fájl' || echo 'nincs ilyen fájl'
ls: nemletezik: No such file or directory
nincs ilyen fájl
$
```

A `!` operátornak egy jobb oldalon található parancs operandusa kell legyen, segítségével a visszaadott `?` változó értéke tagadható:

```
$ ! ls *.txt
1.txt  2.txt  felhasznalok.txt
$ echo $?
1
$
```

A lexikális egységekre figyelni kell

A szintaxis itt és az alábbiakban végig eltér a C, Java stb. nyelvekben megszokottaktól: nem írhatunk **!ls**-t, a parancs értelmezőnek külön lexikális egységként kell látnia a `!` illetve az **ls** sztringeket.

A test parancs

Az parancsok kilépési értékén kívül, gyakran teszteljük az alábbiakat:

- fájlok tulajdonságait (létezik-e, írható-e, olvasható-e, stb.)
- karakterláncokat (vagy változókat, amelyek karakterláncot tartalmaznak: létezik-e, mennyi az értéke)
- számszerű értékeket (számokat tartalmazó karakterláncokat)

A három típusú teszt elvégzését a régi héjakban egy parancsban valósították meg, és ezt ma is használjuk. A parancs argumentumként megkapja a tesztelendő fájlnevet, karakterláncot, számot tartalmazó változót, lefut és kilépési értékében közli a teszt igaz vagy hamis voltát, a már említett fordított logikával. A parancs neve **test** , és megtalálható külső parancs illetve a héjba épített parancs formájában is (mivel igen gyakran használjuk). A **test** hívásakor az argumentumok szerkezete különbözik, ha fájlnévről, karakterláncról illetve számról van szó. Általánosan ezt így adjuk meg:

`test kifejezés`

de az alábbi szintaxissal is hívható, ami jobban kifejezi a feltétel kiértékelést:

`[ kifejezés ]`

A test fontosabb kifejezései az alábbiak (a teljes lista megtalálható a kézikönyben, `man 1 test`, az alábbi opciók megadása a kézikönyv magyar fordítását [184] követi):

Fájlok tesztelése:

`-d file` Igaz ha a file létezik és könyvtár.

`-e file` Igaz ha a file létezik.

`-f file` Igaz ha a file létezik és szabályos fájl.

`-L file vagy -h file` Igaz ha a file létezik és szimbolikus hivatkozás (szimbolikus link).

`-r file` Igaz ha a file létezik és olvasható.

`-s file` Igaz ha a file létezik és 0-nál nagyobb méretű.

`-w file` Igaz ha a file létezik és írható.

`-x file` Igaz ha a file létezik és végrehajtható.

`-O file` Igaz ha a file létezik és az aktuális felhasználó tulajdonában van.

`-G file` Igaz ha a file létezik és az aktuális csoport tulajdonában van.

`file1 -nt file2` Igaz ha file1 újabb (a módosítási időt tekintve), mint file2.

`file1 -ot file2` Igaz ha file1 régebbi, mint file2.

`file1 -ef file2` Igaz ha file1 és file2 -nek azonos eszköz- és i-node száma van. Tulajdonképpen ez azt jelenti, hogy hard linkek.

Sztringek tesztelése:

`-z string` Igaz ha a string 0 hosszúságú.

`-n string` Igaz ha a sztring nem 0 hosszúságú (kétféleképpen lehet tesztelni, a `sztring` kifejezés ugyanazt jelenti).

`string1 = string2` Igaz ha a stringek megegyeznek.

`string1 != string2` Igaz ha a sztringek nem egyeznek meg.

Logikai tesztek két test kifejezés között:

`! expr` Igaz ha expr hamis.

`expr1 -a expr2` Igaz ha expr1 és expr2 is igaz

`expr1 -o expr2` Igaz ha expr1 vagy expr2 igaz

Számokat tartalmazó sztringek összehasonlítása:

`arg1 OP arg2` az OP operátor valamelyik a következőkből: `-eq`, `-ne`, `-lt`, `-le`, `-gt`, `-ge` . A rövidítések a következő angol kifejezésekből származnak: *equal*, *not equal*, *less then*, *less or equal*, *greater then*, *greater or equal* stb. műveleteket jelentik. Ezek az aritmetikai operátorok igaz értéket adnak, ha arg1 rendre egyenlő, nem egyenlő, kisebb mint, kisebb vagy egyenlő, nagyobb mint, nagyobb vagy egyenlő mint arg2. arg1 és arg2 pozitív vagy negatív egész kell legyen

Példák (a test lefuttatása után azonnal meghívjuk az **echo $?** parancsot, hogy lássuk a teszt eredményét):

```
$ touch 1.txt
$ touch 2.txt
$ chmod a-w 2.txt
$
$ # létezik-e a file
$ test -f 1.txt ; echo $?
0
$ # könyvtár-e
$ test -d 1.txt ; echo $?
1
$ # írható-e
$ test -w 1.txt ; echo $?
0
$ # létrehozok 2 sztringet
$ a='abc'
$ b='def'
$ # a $a hossza 0 ?
$ test -z $a ; echo $?
1
$ # $a egyenlő-e $b-vel
$ test $a = $b ; echo $?
1
$ # létrehozok 2 sztringet amelyek számot tartalmaznak
$ x=2
$ y=3
$ # $x értéke kisebb mint $y ?
$ test "$x" -lt "$y" ; echo $?
0
$ # $x értéke kisebb mint 1 ?
$ test "$x" -lt "1" ; echo $?
1
$
```

Valamennyi esetben a `[ ]` szintaxissal is hívható a **test**, tehát az utolsó esetre például írhatjuk:

```
$  [ "$x" -eq "$y" ]  ; echo $?
1
$
```

A [[ ]] szerkezet

A hagyományos **test** vagy **[]** parancs helyett lehet ennek modern változatát, a **[[ ]]** szerkezetet használni újabb héjakban. Ezt a Bash esetében bemutatjuk a A [[ ]] szerkezet [115] című fejezetben.

Az expr parancs

Az **expr** parancsot egyszerű kis műveletek elvégzésére alakították ki héj változók között, mint például két számértéket tartalmazó változó összeadása (amire gyakran van szükség vezérelt végrehajtás során). Ma már ritkábban használják, mert van jobb megoldás a műveletek elvégzésére és ezt a Bash is átveszi (a `(( ))` vagy a **let** szerkezetek, lásd az Aritmetikai kiértékelés ... [111] című fejezetben). Ennek ellenére, használata ma is előfordul, és kezdetnek, két szám közti művelet elvégzésére megteszi. Az **expr** tud karakterláncokkal is műveleteket végezni. Használata:

`expr kifejezés`

Az **expr** a kimenetre írja ki a kifejezés eredményét, és parancssor helyettesítéssel lehet azt egy másik változóba átvenni. Az **expr** által elfogadott aritmetikai és logikai kifejezések az alábbiak:

`arg1 | arg2` arg1 ha az nem 0 vagy null sztring, egyébként arg2

`arg1 & arg2` arg1 ha egyik argumentum sem 0 vagy null sztring, másképp 0

`arg1 < arg2` arg1 kisebb mint arg2

`arg1 <= arg2` arg1 kisebb vagy egyenlő arg2

`arg1 = arg2` arg1 egyenlő arg2

`arg1 != arg2` arg1 nem egyenlő arg2

`arg1 >= arg2` arg1 nagyobb vagy egyenlő arg2

`arg1 > arg2` arg1 nagyobb arg2

`arg1 + arg2` aritmetikai összeg: arg1 + arg2

`arg1 - arg2` aritmetikai különbség arg1 - arg2

`arg1 * arg2` aritmetikai szorzat arg1 * arg2

`arg1 / arg2` aritmetikai osztás arg1 / arg2

`arg1 % arg2` aritmetikai maradéka az arg1 / arg2-nek

Az **expr** sztring operátorai pedig az alábbiak:

`match string regexp` mintaillesztés a string első karakterétől

`substr string pos length` visszaadja a `string`-ből a `length` hosszúságú alsztringet `pos`-tól kezdve

`index string chars` visszaadja azt az indexet a sztringből ahol bármelyik `chars` halmazban levő karakter található

`length string` a string hosszát adja vissza

Például:

```
$ expr 2 + 3
5
$ a=$(expr 2 + 3)
$ echo $a
5
$
```

A második példából látható, hogyan kell átvenni változóba a kiszámított értéket. Az **expr** használatánál vigyázni kell az olyan műveleti jelekre, amelyek a héj metakarakerei is. Ezeket vissza-per jelöléssel kell használni, mint például a szorzást, mert először a héj értelmezi, és nyilván átírja őket, ha nem használjuk a \-t :

```
$ expr 3 \* 4
12
$
```

Ugyanebbe a kategóriába esnek a < , > , & , | jelek.

Feltételes végrehajtás if szerkezettel

A feltételes végrehajtást az **if** szerkezettel vezéreljük, ez biztosítja, hogy egy bizonyos feltételtől függően egyik vagy másik parancssor legyen végrehajtva. Ez a szerkezet természetesen egy végrehajtott parancs kilépési értékétől függően dönt a további végrehajtásról. Használata:

```
if parancssor
then
    parancssor
    . . .
else
    parancssor
    . . .
```

```
fi
```

, vagy többszörös feltétel esetén:

```
if  parancssor
then
    parancssor
    . . .
elif  parancssor
then
    parancssor
    . . .
else
    parancssor
    . . .
fi
```

A feltételt jelentő parancssorban bármilyen parancs végrehajtható, akár egyedülálló, akár csővezetékkel összekötött parancs lehet. Leggyakrabban a **test** parancsot használjuk itt a döntések megvalósítására, de lehet ott bármilyen más parancs is.

Az alábbi szkript egy sort próbál beírni egy fájl végére. A **test** `-f` opciója a fájl szabályosságát, `-w` opciója az írhatóságát teszteli.

```
#!/bin/bash

szoveg="Ezt írjuk a fájl végére."
file="1.txt"


#teszteljük, hogy a fájl nem szabályos fájl
if ! [ -f "$file" ]
then
    echo "$file" nem létezik vagy nem szabályos fájl
    exit 1
fi

#teszteljük, hogy a fájl írható-e
if [ -w "$file" ]
then
    echo "$szoveg" >> "$file"   #szöveg a fájl végére
else
    echo "$file" nem írható
fi
```

A [ ] szerkezet szintaxisa

A test zárójelezésénél a [ ] zárójelek mindkét oldalán egy elválasztó szóközt kell hagyni. A `[` és `]` lexikális szerkezetek:

```
if [ -w "$filenev" ]
```

```
   # ^ ^             ^ ^  a nyilak fölött szóköz van !!!
```

Használhatunk feltételnek egy egyszerű parancsot is, például: belépek az `elso` könyvtárba, ha az létezik, egyébként hibát írok ki :

```
#!/bin/bash

#megpróbálok belépni az elso könyvtárba

if cd elso 2>/dev/null
then
    echo "sikerült a könyvtárba lépni."
else
    echo "nincs elso könyvtár."
fi
```

Ezekben az esetekben nincs szükségünk a feltételként használt parancs kimenetére vagy hibakimenetére, így ezeket a `null` eszközre vagy napló fájlba irányítjuk. példa (`pipes.sh`) csővezeték használatára a feltételben:

```
#!/bin/bash

#a feltételben egy hosszabb parancssort hívunk meg

if cat /etc/passwd | cut -f 1 -d ":" | sort > felhasznalok.txt
then
    echo 'megvan a felhasználók listája'
else
    echo 'nem sikerült előállítani a listát'
fi
```

Csővezeték mint feltétel

Az **if** szerkezet a csővezeték utolsó parancsa által beállított `?` változót kapja meg. A parancssor lefuthat úgy is, hogy valamelyik előző program hibásan fut, de az utolsó mégis igaz kimenetet ad. Ha a fenti programban elírjuk a `passwd` fájl elérési útját, például `/etc/passw`-t írunk helyette, a szkript lefut, de az első csővezeték üres bemenetet kap: ennek ellenére az üres `felhasznalok.txt` létrejön. A Bash ennek kivédésére az utolsó lefuttatott csővezeték összes parancsának kimeneti értékét egy `PIPESTATUS` nevű környezeti változóban adja vissza, amely egy tömb. Ez a 0 indextől kezdve tartalmazza a csővezeték első, második, stb. parancsának kilépési értékét. Fontos szkripteknél ezt tesztelni lehet. Ebben a tananyagban nem foglalkozunk a Bash tömbökkel, a részletek erről a Bash kézikönyvében (BashRef2010 [184], 6.-dik fejezet).

Ugyanakkor az említett `&&` , `||` illetve `!` szerkezeteket is használhatjuk a feltétel parancssorán bonyolultabb döntések kivitelezéséhez. Az alábbi szkript törli az `1.txt` fájlt ha az írható és a mérete 0 byte:

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

```
#az első argumentum a fájlnév

file=${1:? hiányzik a fájlnév}

#teszteljük, hogy írható-e a fájl és hossza 0 -e?
#a hossz tesztelésénél megnézzük,
#hogy mérete nagyobb-e mint 0 byte és tagadjuk ezt

if [ -w "$file" ]  &&  ! [ -s "$file" ]
then
    rm "$file"
    echo "$file" törolve
else
    echo "$file" hossza nem 0 vagy nem írható
fi
```

(a `-s` operátor igazat ad vissza, ha a fájl hossza nagyobb mint 0 byte).

A fenti példában két teszt parancs kimenetén végzünk logikai műveletet.

Az **if** vagy **else** ág a feltételes végrehajtásnál nem lehet üres (valamit végre kell ott hajtani). Amennyiben a programozás során mégis üres ágat akarunk használni, a héj üres utasítását kell használnunk, a melyet a `:` szimbólummal hívunk meg. Ez semmit nem végez, de egy igaz értéket hagy a `?` változóban:

```
if [ -w "$file" ]  && ! [ -s "$file" ]
then
    rm "$file"
    echo $file törolve
else
    :
fi
```

A for ciklusok

A héj hagyományos **for** ciklusa egy listán végez iterációt. A lista elemei sztringek. Használata:

```
for  i  in  lista
do
    parancssor
done
```

A parancssor-ban az `i` változó használható, és sorra felveszi a lista elemeinek értékét. A lista megadása opcionális: amennyiben elhagyjuk, használata így alakul:

```
for  i
do
    parancssor
done
```

ekkor az `i` változó a parancssor paraméterein iterál, tehát a `@` változóban található sztringeket járja végig.

Példák:

Egy megadott lista végigjárása (`for1.sh`):

```
#!/bin/bash

for i in abc def ghi
do
    echo $i
done
```

sorban az abc, def, ghi karakterláncokat fogja kiírni. Futtatásnál ezt látjuk:

```
$ bash for1.sh
abc
def
ghi
$
```

Ha pl. számokat akarunk végigjárni, akkor számokat ábrázoló sztringeket a **seq** paranccsal generálhatunk (`for2.sh`):

```
#!/bin/bash
for i in $(seq 1 5)
do
    echo $i
done
```

futtatva:

```
$ bash for2.sh
1
2
3
4
5
$
```

A **seq** parancs kiír egy számokból álló szekvenciát, a korlátokat és a lépésközt az argumentumokban lehet megadni. Az implicit lépés az 1. Pl. a **seq 1 10** kiírja a terminálra 1-től 10-ig a számokat, a **seq 1 2 10** hívás csak minden második számot írja ki.

Egy könyvtárban található fájlneveket az alábbi ciklussal járjuk végig (`for3.sh`):

```
#!/bin/bash

for f in $( ls )
```

```
do
    echo $f
done
```

lefuttatva:

```
$ bash for3.sh
1.txt
2.txt
$
```

A Bash itt az **ls** parancsot és nem az **ls** aliast hívja meg, ezért a visszaadott listába nem kerülnek színes kijelzést vezérlő karakterek, ami általában jellemző az **ls** alias-ra, ha az definiálva van.

Az alábbi példa (`for4.sh`) a következőket végzi:

-a parancssor argumentumait járja végig, feltételezzük, hogy ezek fájlnevek
-megpróbálja a fájlokat a home könyvtárban levő backup könyvtárba másolni
-ha ez nem sikerül (pl. valamelyik nem igazi fájl hanem könyvtár) egy sztring listába írja a fájl nevét
-a program végén a listát egy napló fájl végére írja

A programban használjuk a **date** parancsot, ez egyszerű hívásnál a pillanatnyi dátumot és időt írja ki. Az **exit** kilépési parancs paramétere a kilépési kód (lásd alább [76]). Ez 0, ha a program jól futott le, és 0-tól különbözik, ha hibával lépünk ki.

A **date** kimenete az alábbi:

```
$ date
Mon Mar 14 23:55:45 EET 2011
$
```

A szkript:

```
#!/bin/bash

#a napló fájl a home könyvtárban van
NAPLO="$HOME/naplo.log"     #ez a napló
BACKUP="$HOME/backup"       #ide másolunk
PN="for4.sh:"               #ezt  programnevet írjuk a naplóba

#létezik-e a backup könyvtár
if ! [ -d "$BACKUP" ]
then
    echo "A $BACKUP könyvtár nem létezik"
    exit 1
fi

datum=$(date)  #a date parancs a futás időpillanatát
               #rögzíti egy változóban

#ha a programot argumentumok nélkül indították
```

```
if [ $# -eq 0 ]
then
    echo $PN $datum "argumentum nélküli végrehajtás" >> "$NAPLO"
    exit 1
fi

#ez a lista kerül a program végén a naplóba
lista=""

#a $@ parancsori változókat tartalmazó listán iterál
for file
do
    if cp "$file" $BACKUP 2>/dev/null  #hibaüzenet eldobva
    then
        echo "$file" elmentve
    else
        #ha nem sikerül, akkor beírom a listába
        #például a fenti masolást könyvtárra nem lehet
        #végrehajtani
        lista="$lista"" $file"
    fi
done


#a napló végére írjuk ha van valami a listában
if [ -n "$lista" ]  #ha nem üres sztring
then
    echo $PN $datum "Nincs elmentve:" "$lista" >> "$NAPLO"
fi

exit 0
```

A while és until ciklusok

A **while** illetve **until** ciklusok feltételét ugyanúgy állítjuk elő, mint az, **if** szerkezetét, tehát a feltétel valamilyen parancssor, amely beállítja a `?` változót. A **while** addig folytatja a ciklust míg feltétele igaz értékkel lép ki, az **until** pedig amíg a feltétel hamis. Használatuk:

```
while parancssor
do
    parancssor
    . . .
done
```

valamint:

```
until parancssor
do
    parancssor
    . . .
done
```

Megfigyelhető, hogy a héjban minkét ciklusnak az elején van a teszt. A **while** ciklust sokkal gyakrabban használjuk, ebben a tananyagban szinte minden esetben a **while**-t alkalmazzuk.

A **while** ciklus használható valamilyen rendszerbeli feltételtől függő ismétlésre. Ha a ciklusban nem végzünk olyan tevékenységet, amely késleltetne, és a ciklus elszabadulhat nagyon gyors, hosszú idejű ismétlésre, ajánlott egy késleltető parancsot használni, amelyik rövid időre felfüggeszti a végrehajtást. A **sleep** parancs (részletesen lásd a fejezet végi feladatokban [85]) egy megadott ideig függeszti fel a szkript futását, pl. 1 szekundumot várakoztat a **sleep 1** hívás.

Az alábbi ciklus arra vár, hogy a `teszt.txt` fájl megjelenjen. Ezt 1 másodpercenként ellenőrzi.

```
#!/bin/bash

while ! [ -f teszt.txt ]   # teszt.txt reguláris fájl (létezik?)
do
    sleep 1                # vár 1 másodpercet
done

echo teszt.txt megjelent!
```

Az alábbi pedig egy klasszikus, egész számmal vezérelt ciklus:

```
#!/bin/bash

n=5                      #n számértéke kezdetben 5
echo $n

while [ $n != 0 ]
do
    sleep 1

    n=$( expr $n - 1 )   # n=n-1

    echo $n
done
```

Az **expr** kissé nehézkessé teszi a számláló csökkentését, a `(( ))` szerkezettel megoldható ugyanez elegánsabban (lásd 8. fejezet [111]).

Az alábbi példában addig olvasunk a terminálról, amíg az "end" szót gépeljük be. A példában a héj **read** parancsát használjuk (használatának részletes ismertetését lásd a fejezet végén [80]). A

```
read line
```

parancs hívás egy sort olvas a terminálról a `line` változóba.

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

```
end="end"

#addig olvas sorokat a terminálról, amíg begépeljük
#az "end" sztringet

while [ "$line" != "$end" ]
do
    read line
done
```

A **read** feltételként is kényelmesen használható, mert hamisat térít vissza ha a beolvasás nem sikerül (pl. Ctrl-D fájl vége karaktert kapott bemenetként). Az alábbi kis program például egy nagyon egyszerű szűrőként működik. Beolvas sorokat a bemenetről (ami átirányítás esetén akár egy fájl is lehet), feldolgozza és kiírja őket.

```
#!/bin/bash
while read line
do
    #sor feldolgozasát itt lehet elvégezni
    # ...
    echo $line              # sor kiírása
done
```

Megjegyzendő, hogy a fájlok soronkénti feldolgozására alkalmasabb olyan parancsokat használni, amelyeket eleve erre írtak (**wc**, **uniq**, **tail**, **cut**, **egrep**, stb.), mert azok sokkal gyorsabban teszik ezt mint egy **while** ciklus a héjban. Mégis, amennyiben nem áll rendelkezésünkre megfelelő parancs, ezzel a módszerrel dolgozhatunk fel egy sorokból álló bemenetet.

Az **until** ciklus akkor hasznos, ha valamilyen eseményt kell figyelni, és van egy változó amelyik a ciklus végrehajtása közben kap értéket. Ennek ellenére, nagyon ritkán használjuk. Az példában (`until.sh`) akkor lépünk ki a ciklusból, ha a `line` változó végleges értéket kap:

```
#!/bin/bash
#
#az until szerkezet ciklus negatív logikaval

#a teszt azt figyeli, hogy mekkora a $line változó hossza

#az első  ciklusban a line változó nem létezik
#így a test biztos hamis
until [ $line ]
do
    echo 'A $line változó értéke: ' "$line"
    echo "Írja be pontosan az \"abc\" sztringet"

    read line

    #ha nem abc-t ír be, akkor törlöm a változót
    #a shell unset parancsával

    if [ "$line" != "abc" ]; then
```

```
        unset line                  #változó törlése
    fi
done
```

A héj **unset** parancsa törli a megadott változót.

A break és continue használata

A két héj szerkezet hasonlóan működik a C-ből megszokott ciklus módosító utasításokkal. A **break** megszakítja a ciklust ott ahol megjelenik (és a ciklus utáni parancson folytatja), a **continue** nem fejezi be a kurrens ciklust hanem a következő iterációra lép. Megjegyzendő, hogy a **break** használható többszintű ciklusból való teljes kilépésre, pl. a **break 2** elhagy egy 2 szintes ciklust.

Az alábbi példában a **break** egy feltétel hatására elhagy egy végtelen ciklust. A végtelen ciklust a héj **true** parancsával vezéreljük, amelyik mindig igazat térit vissza (létezik az ellenkezője is, a **false** parancs, amelyik hamisat térít vissza a `?` változóba). A **read** parancs `-p` kapcsolója egy készenléti jelet megadó karakterláncot ír ki.

```
#!/bin/bash

#végtelen ciklus

while true
do
    read -p "Írjon be egy sort:" line
    #ha nem üres a beolvasott sor kilépek a ciklusból
    if [ -n "$line" ]
    then
        break
    fi
done
```

A shift parancs

A **shift** "elmozgatja" a parancssor argumentumait egy argumentummal balra, kiejtvén a listából az első argumentumot. Ugyanakkor az argumentumok számát tartalmazó `#` változó értéke is változik, egyel csökken. Az alábbi példában a szkript kiír minden parancssori argumentumot, de mindig az `1`-es változót írja ki. A **shift** minden ciklusban a átnevezi a `1`,. . .,`9` speciális változókat, úgy hogy a megfelelő parancssori értékeknek eggyel kisebb számjegy nevű változót rendel. A módosítás amit végez egy elmozgatás: pl. ha a `2`-t eltolja `1`-be és ha pl. csak 2 paraméter van, a `2` üres változó marad.

```
#!/bin/bash

while [ $# -gt 0 ]          #amíg van még argumentum
do
    echo  A '$#' értéke: $# , a '$1' értéke: $1
    shift
done
```

```
$ ./shift.sh a b c d
A $# értéke: 4 , a $1 értéke: a
A $# értéke: 3 , a $1 értéke: b
A $# értéke: 2 , a $1 értéke: c
A $# értéke: 1 , a $1 értéke: d
$
```

A **shift**-et paraméterrel is meg lehet hívni, olyankor a paraméternek megfelelő számmal forgatja el a parancssor argumentumait, tehát a **shift 2** hívás kettővel.

Kilépés a héj programból: az exit

A héj programokból az **exit** paranccsal lépünk ki, melynek egy egész számból álló paramétere van, ez lesz a szkript kilépési értéke. Ha jól, hiba nélkül fut le a programunk, ez az érték 0 kell legyen. Az alábbi példa leteszteli, hogy a programunknak van-e legalább egy paramétere. Ha nincs, hibával lép ki.

```
#!/bin/bash

#tesztelem, hogy a parancssor első paraméterének 0 -e a hossza

if [ -z "$1" ]
then
    echo legalább egy paraméter szükséges !
    exit 1           #hibával lép ki a szkript
fi

# feldolgozás itt

exit 0               #jól futott le a szkript
```

Ha elhagyjuk az **exit 0** -t, akkor az utolsó végrehajtott parancs kilépési értéke marad a `?` változóban.

A case szerkezet

A **case** vezérlő struktúra hasonló a programozási nyelvek "*switch*" vezérléséhez: alternatív végrehajtási ágakra bontja a program parancsait. Az alternatívát egy karakterlánc mintákra való illesztése szerint hajtja végre. A következőképpen adjuk meg általános használati formáját:

```
case valtozó in
    minta1 ) parancsok ;;
    minta2 ) parancsok ;;
    .  .  .
    mintaN ) parancsok ;;
esac
```

A szerkezetet **case** és **esac** kulcsszavak határolják. A megadott változóra illeszti a következő sorok elején található mintákat. Ahol a minta talál, ott végrehajtja a jobb zárójel utáni

parancssorozatot amelyet kettős **;** karakter zár le. Csak az első találatnak megfelelő alternatívát hajtja végre.

A minta típusa shell minta (*shell pattern*): karaktereket illetve az állománynevek megadásánál használatos metakaraktereket lehet használni: `* ? []`, valamint két minta "vagy" logikával való összekapcsolására a `|` -at. Példák:

`abc`	pontosan abc-re illeszkedik
`*`	bármire illeszkedik
`?bc`	bármi az első karakter pozíción, és utána pontosan bc
`ab|cd`	ab vagy cd láncokra illeszkedik

Az alábbi kis példa különböző kiválasztásokat végez a parancssor első argumentumát használva:

```
#!/bin/bash

valtozo=${1:? nincs paraméter}

case "$valtozo" in
         a) echo kis a betű ;;
    [0-9] ) echo egy számjegy ;;
  *[0-9]* ) echo van benne számjegy ;;
   *.txt  ) echo ".txt" sztringgel végződik ;;
        * ) echo erre nem írtunk pontos kiválasztást ;;
esac
```

Futtatva:

```
$ bash case1.sh 2
egy számjegy
$ bash case1.sh a2
van benne számjegy
$ bash case1.sh piros
erre nem írtunk pontos kiválasztást
```

Az alábbi kis program (`case3.sh`) a parancssor opcióit elemzi. Ha betartjuk azt a feltételt, hogy egy program hívásakor először a kapcsolókat, utána pedig az argumentumokat adjuk meg (és opciókat nem csoportosítunk) akkor ez a szerkezet alkalmas kis programok opcióinak kezelésére.

```
#!/bin/bash

#vegigjárja a paramétereket
#a while végén levő shift gondoskodik erről
while [ -n "$1" ]
do
    case $1 in
        -f ) echo f opció ;;
        -c ) #a c kapcsolónak van argumentuma
             #tehát elvégzünk még egy shift-et
```

```
            shift
            arg="$1"
            #itt ellenőrizhetnénk, hogy $1 nem üres sztring
            #de ettől most eltekintünk
            echo c opció, argumentuma "$arg" ;;
        -* ) echo ismeretlen opció: $1
            #ez hiba
            exit 1 ;;
           #ha elérkeztünk az első argumentumhoz,
           #amelyik nem kapcsoló:
         * ) break;;
    esac
    shift  #eltoljuk a paramétereket, $1 -ben a következő paraméter
done
```

Utolsó mintának a kiválasztásai sorozatban általában érdemes a `*`-ot használni, ami akkor hajtódik végre, ha az előző kiválasztások közül egyikre sincs találat: így előállíthatjuk a C programozás `switch` szerkezetének `default` ágához hasonló szelekciót. A fenti példában ez az ág azokat az paramétereket kezeli amelyek nem kezdődnek - jellel.

A program a `-c` és `-f` kapcsolót ismeri fel, ha futtatjuk:

```
$ bash case3.sh -f -c 33 22
f opció
c opció, argumentuma 33
az elso opciók utáni paraméter: 22
```

A `-c` opció paraméterének átvétele előtt még egy **shift** parancsot kell végrehajtani. Természetesen a paraméter értékét és létezését ellenőrizni kell egy teljesebb szkriptben (megtörténhet, hogy a **shift** után üres lesz az 1-es változó).

A `getopts` függvény

A héjnak van jobb megoldása az argumentumok kezelésére, a `getopts` függvény, rövid bemutatását lásd a Függelékek A getops függvény [182] című fejezetében vagy a Bash kézikönyvében. Egyszerű programoknál a fenti megoldás is megteszi.

Az alábbi szkript (`case2.sh`) pedig különböző típusú állományokat szortíroz a típusnak megfelelő könyvtárba:

```
#!/bin/bash

#pdf, mp3, avi, jpg, gif típusú állományokat
#külön könyvtárakba válogatja egy forráskönyvtárból
#egy argumentumot vár: a könyvtárnevet ahol az
#állományok vannak

#a célkönyvtárakat az alábbi változókban adom meg
#ezeket is meg lehetne adni paraméterként a parancssoron
PDF=pdf
MP3=mp3
AVI=avi
```

```
KEPEK=kepek

#pontosan 1 paramétert vár
if [ -z "$1" ];then
    echo használat: case2.sh könyvtárnév
    exit 1
fi

#a paraméter tartalmát egy jobban olvasható nevű
#változóba tesszük
forras="$1"

#megnézzük, hogy a célkönyvtárak léteznek-e,
#ha nem létrehozzuk őket
for konyvtar in "$PDF" "$MP3" "$AVI" "$KEPEK"
do
    #ha például van pdf könyvtár, akkor az első feltétel
    #után abbahagyja a tesztet
    if ! [ -d "$konyvtar" ]  &&  ! mkdir "$konyvtar" 2>/dev/null
    then
        echo nem lehet a $konyvtar könyvtárat létrehozni
        exit 1
    fi
done

#végigjárjuk a forrás  könyvtár állományneveit
for f in $( ls "$forras" )
do
    #a case-el választjuk és másoljuk különböző helyekre
    #ha egy fájl egyik kategoriába sem esik akkor a *-al
    #jelölt ágra fut
    case "$f" in
        *.mp3 ) mv "$forras/$f" "$MP3" ;;
        *.avi ) mv "$forras/$f" "$AVI" ;;
        *.pdf ) mv "$forras/$f" "$PDF" ;;
  *.jpg|*.gif ) mv "$forras/$f" "$KEPEK" ;;
        * ) echo a "$f" nem tartozik egyik kategoriaba sem. ;;
    esac
done
```

A select szerkezet

A **select** szerkezetet kis interaktív programok esetében használjuk. Használati módja:

```
select valtozo in lista
do
    parancsok
done
```

A **select** kulcsszót tartalmazó sor végrehajtásakor a parancs egy kis menüt ír ki a `lista` (ez egy sztring lista) segítségével. Ezek után készenléti jelet (*prompt*) ír ki (ezt a `PS3` héj változóból veszi). Válaszként a menü egyik sorát kell kiválasztani úgy, hogy a sorrendjének megfelelő számot ütjük le. Ezek után a **select** belsejében a `valtozo` értéke felveszi a listából kiválasztott

karakterlánc értékét, és ezt fel lehet használni parancs végrehajtásra. Ugyanakkor a héj `REPLY` változója felveszi a beütött szám vagy más sztring értékét.

A szerkezet addig ismétli a menü kiírását és a végrehajtást amíg bemenetként állomány vége jelet kap (Ctrl-D) vagy a parancsok végrehajtási részében egy **break** parancs hajtódik végre. Ha olyan bemeneti számot adunk meg ami nincs a menüben, a változó üres sztringre lesz állítva (a `REPLY` viszont megkapja a beütött értéket).

Működését az alábbi kis példával illusztráljuk:

```
#!/bin/bash
#a PS3 értékét beállítjuk, ez lesz a prompt
#csereljük

PS3='>>'

select valtozo in Elso Masodik Harmadik Vege
do
    echo $valtozo
    echo a '$REPLY' valtozo erteke ilyenkor: "$REPLY"

    #itt barmilyen parancsot vegre lehet hajtani, felhasznalva
    #a $valtozo es $REPLY ertekeit

    if [ "$valtozo" = 'Vege' ];then
            echo "...és jön a kilépés a select-ből"
        break
    fi
done
```

Működése az alábbi:

```
$ bash select.sh
1) Elso
2) Masodik
3) Harmadik
4) Vege
>>1
Elso
a $REPLY valtozo erteke ilyenkor: 1
>>2
Masodik
a $REPLY valtozo erteke ilyenkor: 2
>>
$
```

Addig ismétli a ciklust, amíg az állomány vége jelet (Ctrl-D) ütjük le vagy kiválasztjuk a 'Vége' opciót.

A read parancs

Sorokat olvas be.

Használat:

`read` [-p prompt] [-s] [-t timeout] [-n nkar] [-d kar] *`változó`*...

Fontosabb opciók:

`-p prompt` kiír egy készenléti jel karakterláncot olvasás előtt

`-s` silent: nem használ visszhangot, tehát nem látjuk a leütött betűket: jelszó beolvasásra használjuk

`-t timeout` vár timeout másodpercig, ha az alatt nem írunk be semmit, visszatér

`-n nkar` csak nkar darab karaktert olvas, utána visszatér

`-d kar` delimiter: más elválasztót keres a sorok végén mint az újsor karaktert

Leírás:

A **read** parancs beolvas egy sort a parancssorról egy változóba vagy változó listába. Ha egy változót adunk meg a parancssorán, akkor a teljes sort abba olvassa, ha többet akkor feldarabolja a sort a héj implicit elválasztói szerint (szóköz és tabulátor) és a változók felveszik a kapott sztringek értékeit. A

```
read line
```

tehát az egész sort a `line` valtozóba téríti vissza, a

```
read szo1 szo2 szo3
```

pedig a beírt szavakat a `szo1`, `szo2` stb.-be írja. Elválasztónak a héj implicit elválasztóit használja.

Ha több szó jön a bemenetre mint ahány változót megadunk, az utolsó változóba kerül a fennmaradó szöveg. Pl. ha csak az első szóra van szükségünk, akkor a:

```
read hasznos szemet
```

olvasás a `hasznos` változóba olvassa az első szót, a `szemet`-be a többit.

Ha elhagyjuk a változónevet, akkor a héj `REPLY` nevű beépített változójába kerül a bemeneti sor : ez praktikus akkor, ha a bemeneti változót csak a bemenet átvételére használjuk.

A **read** igaz értékkel tér vissza a `?` változóban ha sikerült beolvasnia. Fájl vége (Ctrl-D) esetén hamissal tér vissza, így lehet egy hosszabb beolvasási sorozatot leállítani. Az alábbi **read** kiír egy rövid szöveget, és siker esetén a sort visszatéríti a `line` változóba:

```
$ read -p "Írd be a neved:" line
```

```
Írd be a neved:
```

Az alábbi csak 3 másodpercig vár, és csak egy karaktert olvas be az `igen_nem` változóba, utána visszatér:

```
$ read -p "Igen vagy nem [I/N]?:" -n 1 -t 3 igen_nem
Igen vagy nem [I/N]?:
```

Ciklusban a while szerkezetben lehet használni mint feltétel:

```
while read line
do
    #a beolvasott sor feldolgozása
done
```

Akkor hagyja el a ciklust, ha nem tud olvasni: pl. a bemenet végéhez ér.

Fájlok olvasása a read-el

A **read** -el csak a standard bemenetről fogunk olvasni ebben a könyvben, ez az implicit működési módja. A parancs tud bármilyen fájlból olvasni, ha megadjuk a fájl azonosítóját. Ezt a `-u` kapcsolóval lehet elérni. A fájlt ezt megelőzően az **exec** paranccsal meg kell nyitni és használat után le kell zárni egy bemenet lezáró operátorral. Erre adunk most egy példát, de nem fogjuk így használni a **read**-et.

```
exec 6< "1.txt"   #bemeneti fájl megnyitása 6-os azonosítóval
read -u 6 line       #egy sor beolvasása
echo $line
exec 6<&-     #fájl lezárása
```

Részletek, valamint az összes Bash által a be és kimeneti műveleteknél használt operátor megtalálható a Bash kézikönyvének [184] 3. fejezetében.

Példaprogramok

Az alábbi kis szkriptekben már kihasználjuk a ciklus szerkezetek által nyújtott lehetőségeket. A példák fájlrendszerhez kötődő kis feladatok.

Fájl másolás

Írjunk egy szkriptet, amelyik egy könyvtárból átmásolja a fájlokat egy másik könyvtárba, ha a fájlok teljesítenek egy feltételt!

Az alábbi sémát alkalmazhatjuk. Legyen a feltétel például a fájlban levő sorok száma. Másoljunk, ha a sorok száma nagyobb mint egy N szám. Feltételezzük, hogy a forráskönyvtárban csak igazi fájlok vannak, és minden állomány szöveges (ezt a két feltételt is lehet tesztelni, de első lépésben ettől eltekintünk). Attól azonban nem tekintünk el, hogy a könyvtárként megadott nevek létezzenek és valóban könyvtárak legyenek. Ha nagyon pontosak akarunk lenni, azt is le kell tesztelnünk, hogy a forráskönyvtáron az r és x , a célkönyvtáron

pedig az rwx jogok be vannak-e állítva. Új fájl létrehozásának feltétele a w jog a könyvtáron. Az alábbi szkriptben csak néhány fontosabb tesztet végzünk el ezek közül.

Egy megoldás (`fcopy.sh`):

```
#!/bin/bash
#átmásol fajlokat egy könyvtárból egy másikba,
#ha a fájlban legalább N szövegsor van
#ha a célkönyvtár nem létezik, létrehozza
#
#Paraméterek:
#   $1 - forráskönyvtár
#   $2 - célkönyvtár
#   $3 - sorok száma
#
#ha nincs pont 3 paraméter, kilép:

if [ $# -ne 3 ]
then
    echo használat: fcopy.sh forrás cél sorszám
    exit 1
fi

#parameterek átvétele
forras="$1"
cel="$2"
sorszam="$3"

#teszteljük a könyvtárakat
if ! [ -d "$forras" ]
then
    echo "$forras" könyvtár nem létezik
    exit 1
fi

#ha a célkönyvtár nem létezik létrehozzuk
if ! [ -d "$cel" ]
then
    if ! mkdir "$cel" 2>/dev/null
    then
        echo nem lehet létrehozni a cél könyvtárat
        exit 1
    else
        #ki is írjuk ha sikerült
        echo "$cel" új könyvtár
    fi

fi

#ellenőrizzük, hogy a célkönytárban tudunk e létrehozni
#fájlokat, van-e w jogunk
if ! [ -w "$cel" ]
then
    echo a "$cel" -ben nem lehet fájlokat létrehozni
```

```
    exit 1
fi

#megszámoljuk hány fájlt másolunk
szam=0

#végigjárjuk a forráskönyvtár fájljait
#feltételezzük, hogy csak szöveges fájlok vannak benne
#es ezek elérhetőek
for f in $( ls "$forras" )
do

    #a fájlban levő sorok száma
    N=$( wc -l "$forras/$f" | cut -f 1 -d " " )

    #itt teszteljük a másolás feltételét
    if [ $N -gt $sorszam ]
    then
        #ha teljesül a feltétel másolunk

        #kiírjuk, hány sor van a fájlban, de nem írunk újsort
        #így ugyanabban a sorban fog megjelenni
        #a cp által kiírt szöveg is

        echo -n "$N soros fájl : "

        #használjuk a -v opciót, hogy lássuk mi történik:

        if ! cp -v "$forras/$f" "$cel/$f"
        then
            echo másolás nem sikerült: "$forras/$f" "->" "$cel/$f"
            exit 1
        fi

        #számláljuk a fájlokat
        szam=$( expr $szam + 1 )

    fi
done

echo $szam fájl másolva
```

Végrehajtva a `teszt1` könyvtárra:

```
$ bash fcopy.sh teszt teszt1 2
teszt1 új könyvtár
4 soros fájl : `teszt/1.txt' -> `teszt1/1.txt'
3 soros fájl : `teszt/2.txt' -> `teszt1/2.txt'
2 fájl másolva
$
```

Könyvtár fájljainak követése

Írjunk egy szkriptet amelyik a parancssorán megadott könyvtár állományait figyeli 2 másodpercenként, és amennyiben ezek száma meghalad egy adott N számot, letörli a legrégebbi állományt!

Egy megoldás az alábbi szkript (`watch_file.sh`), az 5 másodpercenkénti várakozást a **sleep** paranccsal oldjuk meg, melynek argumentumként a várakozás idejét kell megadni másodpercben. A **sleep** Linux rendszereken elfogad tizedes számokat is paraméterként, illetve a paraméter után az implicit másodperc mértéket módosíthatjuk percre vagy órára az `m` vagy `h` jelző karakterekkel: `sleep 3` ; `sleep 0.5` ; `sleep 2m` ; `sleep 2h` helyes parancsok. A szkriptünk:

```
#!/bin/bash
# a parancssorán megadott könyvtárat figyeli 2 másodpercenként
# ha az ott levő fájlok száma meghalad egy N számot
# letörli az időben legrégebben módosítottat
# Paraméterek:
#   $1 - a könyvtár neve
#   $2 - az N egész szám
#ha nincs pont 2 paraméter, kilép
if [ $# -ne 2 ]
then
    echo használat: watch_file.sh dir N
    exit 1
fi

#paraméterek átvétele
d="$1"              #könyvtárnév
N="$2"              #fájlok száma

#ha d nem könyvtár kilép
if ! [ -d "$d" ]
then
    echo "$d" nem könyvtár
    exit 1
fi

#számoljuk, hány ciklust végzünk
ciklus=0

#a ciklust a true paranccsal vezéreljük, ez mindig igaz
#így a ciklus végtelen, kilépni csak a szkriptet megszakítva lehet
while true
do
    #megnöveljük a számlálót
    ciklus=$( expr $ciklus + 1 )

    #hány fájl van a könyvtárban
    fileno=$( ls -1 "$d" | wc -l )

    #kifecsegjük a ciklus és fájlok számát
    echo $ciklus ciklus, $fileno fájl van
```

```
    #túl sok fájl van?
    if [ $fileno -gt $N ]
    then
        #melyik a legrégebben módosított
        f=$( ls -1tr teszt/ | head -1 )
        #megpróbáljuk letörölni
        if ! rm -f "$d/$f" 2>/dev/null
        then
            #ha ez nem sikerült, kilépünk
            echo vége: nem lehet letörölni a "$d/$f" fájlt
            exit 1
        else
            #kiírjuk a letörölt fájl nevét
            echo "$d/$f" törölve
        fi
    fi

    #várunk 5 másodpercet
    sleep 5

done
```

Példakönyvtárként az alábbi teszt könyvtár fájljait figyeljük:

```
$ ls -l teszt
total 0
-rw-rw-r-- 1 lszabo lszabo 0 Aug  2 04:00 1.txt
-rw-rw-r-- 1 lszabo lszabo 0 Feb  2  2011 2.txt
-rw-rw-r-- 1 lszabo lszabo 0 Jan  2  2011 3.txt
-rw-rw-r-- 1 lszabo lszabo 0 Sep 22 17:28 4.txt
-rw-rw-r-- 1 lszabo lszabo 0 Sep 22 17:57 5.txt
$
```

Elindítjuk a programot, és futás közben a harmadik ciklus után egy második terminálról létrehozunk egy ötödik fájlt, majd Ctrl-C -vel leállítjuk a programot:

```
$ bash watch_file.sh teszt 4
1 ciklus, 4 fájl van
2 ciklus, 4 fájl van
3 ciklus, 4 fájl van
4 ciklus, 5 fájl van
teszt/3.txt törölve
5 ciklus, 4 fájl van

$
```

Az ötödik fájlt így hozhatjuk létre, egy második terminálról:

```
$ touch teszt/5.txt
```

Megjegyzések:

Az **ls -1** soronként egy fájl nevet ír ki, `-t` opció a módosítási idő szerint listáz, `-r` opció megfordítja a lista sorrendjét, az első a legrégebben módosított fájl lesz.

A fájlok számát megkapjuk, ha a **wc**-al megszámoljuk az **ls -1** kimenetét. A szám csak akkor lesz pontos, ha a könyvtárban csak reguláris fájlok vannak. Ha könyvtárak vagy más típusok is vannak, akkor ugyanezt megtehetjük a Unix **find** parancsával, amely csak az igazi fájlokat fogja keresni (opció: `-type f`) és csak egy könyvtárnyi mélységben (opció: `-maxdepth 1`):

```
$ find teszt -maxdepth 1 -type f
teszt/5.txt
teszt/4.txt
teszt/2.txt
teszt/1.txt
$
```

Ezek időbeli sorrendbe való rendezését el lehet végezni ha ezt a listát átadjuk az **ls**-nek az **xargs** parancs segítségével:

```
$ find teszt -maxdepth 1 -type f -print | xargs ls -1tr
teszt/2.txt
teszt/1.txt
teszt/4.txt
teszt/5.txt
$
```

A **find** és **xargs** bonyolultabb parancsok, használatukat lásd kézikönyveikben.

Szöveges fájl sorainak végigjárása

Amint már említettük [73], ha egy szöveges fájl sorait kell egyenként feldolgozni, arra egyik legjobb módszer a `while` ciklus és a **read** parancs kombinációja úgy, hogy a `while` szerkezet bemenetére irányítjuk a feldolgozott szöveget. A ciklus belsejében bármilyen feladatot elvégezhetünk a sorokkal.

A példaprogramban használni fogunk egy népszerű szövegszűrő parancsot, a **tr**-t.

A **tr** (*translate*) parancs a bemenetére érkező szöveget a kimenetre írja, de kiírás előtt karaktereket cserél vagy töröl, tehát szövegszűrőként működik. Használata:

`tr` [opciók] karakter-lista1 [*`karakter-lista2`*]

Opciók nélkül egyszerűen az első, `karakter-lista1`-ben megadott karaktereket kicseréli a `karakter-lista2`-ben megadott karakterekkel. A karakter lista itt karakter felsorolást jelent, tehát elvileg minden karakternek az első listából megfelel egy a másodikból. Ha a második lista nem elég hosszú, akkor az utolsó karakterét ismétli, amíg kiegészíti. Például az alábbi parancs kicseréli a bemeneti sztring kisbetűit nagybetűkre:

```
$ echo bemenet | tr a-z A-Z
BEMENET
$
```

A két karakterlistát ajánlatos aposztrófok közé zárni ha speciális karakterek jelennek meg a listákban.

A `tr -d` opcióval csak egy karakterhalmazt kér argumentumnak, ezeket a karaktereket törli a bemeneti sztringből:

```
$ echo abcdef | tr -d ae
bcdf
$
```

A `-s` (*squeeze*) kapcsolójával való használata eltünteti az ismétlődő karaktereket a bemenetből:

```
$ echo abbbbbcddddddddddde | tr -s bd
abcde
$
```

, a `-c` (*complement*) opciója pedig a a megadott karakterlista komplementum halmazával (tehát azokkal a karakterekkel amelyek nincsenek a listában) végzi el a törlés vagy ismétlés szűrés. műveleteket (tehát a `-s` vagy `-d` -vel együtt használjuk.

Az alábbi szkript (`sorok.sh`) végigjárja a szöveges állomány sorait, és kiírja őket úgy, hogy a sor elé írja a sorszámot és a benne levő karakterek számát, a sort pedig nagybetűssé alakítja. Az sor nagybetűssé alakítását a **tr** paranccsal végezzük:

```
#!/bin/bash
#kiírja a egy szöveges fájl sorait nagybetűkkel,
#minden sor elé írja a sor számát és a sorban levő
#karakterek számat
#Paraméterek:
#   $1 - a fájl neve
#

#csak egy paraméter
if [ $# -ne 1 ];then
    echo hasznalat: sorok.sh fájlnév
    exit 1
fi

#változóba kerül a név
file="$1"

#ha a paraméter nem egy közönséges állomány, kilépek
if ! [ -f "$file" ];then
    echo $file nem közönséges fájl
    exit 1
fi

#kezdeti érték
sorszam=0

#a sorokat úgy olvassuk, hogy a while
#ciklusba irányítjuk a bemeneti állományt
```

```
#lásd a done kulcsszó utáni átirányítást

while read line
do
    # a sorszám növelése
    sorszam=$( expr $sorszam + 1 )

    #az egy sorban levő karakterek számának kiszámítása a wc-vel
    karakterek=$( echo $line | wc -c )

    #a sorok végén van az újsor karakter is,
    #így kivonunk 1-et a kapott számból
    karakterek=$( expr $karakterek - 1 )

    #a line változó tartalmát nagybetűssé alakítjuk
    #a tr paranccsal

    line=$( echo "$line" | tr a-z A-Z )

    #kiírom
    echo $sorszam ':' $karakterek ':' $line

done < "$file"  #itt adjuk meg az átirányítást
```

Végrehajtás:

```
$ cat lista1.txt
elso
masodik
harmadik
negyedik
otodik
hatodik
hetedik
$ bash sorok.sh lista1.txt
1 : 4 : ELSO
2 : 7 : MASODIK
3 : 8 : HARMADIK
4 : 8 : NEGYEDIK
5 : 6 : OTODIK
6 : 7 : HATODIK
7 : 7 : HETEDIK
$
```

Megjegyzés

A **tr** az átalakításokat, amennyiben felsorolunk karaktereket az un. POSIX karakterkészletekkel végzi, tehát az `a-z` listában nincsenek benne a magyar ékezetes karakterek. Ilyen halmazokat külön fel kell építeni, pl: `áéíóú` .

A programozási hibák felderítése

Ha egy héjprogramban változók, vezérlő szerkezetek is vannak, a jó működés követéséhez szükség lesz néhány hibakereső módszerre. Az egyszerű, gyorsan használható hibakeresés

módszereit a Hibakeresés héjprogramokban [176] című függelékben mutatjuk be. Ezek kipróbálhatóak gyakorlatképpen az előző programokon.

7. fejezet - Reguláris vagy szabályos kifejezések alkalmazása

Bevezető

A reguláris kifejezések szövegmintákat leíró nyelvet (tulajdonképpen formális nyelvet) jelentenek, segítségével rendkívül könnyen oldhatóak meg keresés, helyettesítés, általában szövegek feldolgozáshoz kötődő feladatok.

Bár konkrétan, mindennapi gyakorlatként először a UNIX-on kezdték használni őket szövegszerkesztőkben (Ken Thompson), jelentőségük túlmutat a UNIX eszközein: ma már minden modern programnyelvbe beépítették őket nyelvi vagy könyvtár szintjen. Ha egy szoftver szövegek feldolgozását is el kell végezze, akkor szinte egyértelműen ezeket használjuk. Így általánosan programfejlesztés közben, adatbányászatban, címkéket használó nyelvekkel való munkában (XML, HTML) vagy a genetikai adatok feldolgozásában nemcsak használatosak, hanem sokszor az első számú eszközt jelentik.

A reguláris kifejezésekkel a UNIX parancsaiban legalább három változatban találkozunk:

- alapszintű (*basic*, *BRE*)
- POSIX bővített (*extended*, *ERE*)
- Perl kompatibilis - a Perl programozási nyelvben használatos kifejezések, mai változata a Perl 5-ös verziójában jelent meg.

Az alapszintű kifejezéseket a régi programokkal való kompatibilitás miatt használjuk. Ilynek például az alábbi Linux/Unix alatt használatos parancsok: **expr**, **grep**, **sed**.

A bővített kifejezéseket az alábbi esetben használjuk: **egrep** vagy **grep** `-E` kapcsolóval, **sed** `-r` kapcsolóval, **awk**, illetve a különböző programozási nyelvekből, ahol a függvények ilyen kifejezést használnak.

A kifejezések variánsai a Unix parancsokban eléggé szerteágazóak, így általában egy új parancs használatánál át kell nézni, hogy milyen típust használ az illető program. Például a **vi** vagy **emacs** szerkesztők saját dialektusaikat használják, amelyek eltérnek az említettektől. Ebben a tananyagban a POSIX bővített kifejezéseket használjuk, de megemlítjük a hagyományos BRE kifejezéseket is. A Perl illetve más programozási nyelvekben használatos kifejezések általában bonyolultabbak ezeknél, és szintén nyelvspecifikusak lehetnek.

További tanuláshoz *Jeffrey Friedl: Reguláris kifejezések mesterfokon* című könyve (Friedl2004 [184]) részletesen tárgyalja a reguláris kifejezéseket, beleértve a különböző programozási nyelvekben (Perl, Java vagy .NET-ben) használt variánsokat.

A fejezet elsődleges célja az **egrep** parancs használatának elsajátítása lesz: a megszerzett tapasztalatot utána más parancsoknál is lehet hasznosítani. Az **egrep** alapértelmezésben szövegmintákat keres egy szöveges állomány soraiban.

A bővített kifejezések (extended)

Ebben a fejezetben a reguláris kifejezések jelölésénél a következő megjelenítést használjuk: `abc[0-9]` . Ha szükséges, a szóközt • , tabulátort → jellel jelöljük. A kifejezésre történő illesztést így emeljük ki: **abc7**xyz .

A reguláris kifejezések tehát egy olyan nyelvet jelentenek amellyel karakterláncokban megtalálható mintákat írunk le. A *minták* alatt az egymásutáni karakterek egy jellegzetes sorozatát értjük. Így mintáról beszélünk, ha azt mondjuk, hogy három egymás utáni kis a betű és utána egy kettes, de akkor is, ha általánosabban fogalmazunk, mint pl.: három egymás utáni kis betűt egy számjegy követ.

A mintákat karaktersorozatokban fogjuk keresni, és első megközelítésben csak az angol nyelv karakterkészletével fogunk dolgozni (gépi nyelvekben tulajdonképpen ezek fordulnak elő). Nem foglalkozunk a más nyelven írt szövegek kapcsán használt mintákkal, ezek használata a programoktól függően eltérhet az angol nyelvben használtaktól.

Ha a minta megtalálható egy szövegrészben, akkor azt mondjuk, hogy a minta *illeszkedik* a szövegre. Az illesztés (*match*) fogalmával tulajdonképpen egy keresés eredményére utalunk. Pl. a fent említett minták 2 helyen is illeszkednek a **aaa2**xyx**aaa2**klm sorozatban. Ilyen kereséskor az első illesztésnek jelentősebb szerepe lehet: sokszor csak az a cél, hogy az elsőt megtaláljuk.

Ha a keresett kifejezés többször fordul elő a feldolgozott sorban, akkor a használt programtól vagy annak futtatási opcióitól függ, hogy keresi-e a második előfordulást, vagy leáll az elsőnél. Az **egrep** például implicit megkeres minden találatot egy sorban: de vannak olyan futtatási opciói, amikor megelégszik az első találattal (pl. ha csak azt kell megállapítania, hogy egy bizonyos állományban megvan-e a minta, és nem azt, hogy hol).

Bár ezzel nem fogunk foglalkozni, megemlítjük, hogy a keresést véges automatákat használó karakterlánc keresés algoritmusokkal történik. Ezeket egy olyan szoftver komponens hajtja végre a leírt minták alapján amelyet reguláris kifejezés motornak nevezünk (*regular expression engine*).

A reguláris kifejezésben *karakterek* és *metakarakterek* találhatóak: ezek közösen határozzák meg a keresett mintát.

Metekaraktereknek nevezzük azokat a karaktereket amelyek egy reguláris kifejezésben más jelentéssel bírnak, mint a karakter valódi jelenése. Például a `^` karakter amennyiben egy kifejezésben használjuk arra utal, hogy a mintának azt a pontját ahol megjelenik csak a feldolgozott karakterlánc elejére lehet illeszteni.

A minta illesztése egy karakterláncra úgy történik, hogy a motor balról jobbra végigjárja a karakterláncot, és megpróbálja illeszteni a mintát. Egy ilyen feldolgozott karakterláncban külön pozíciót jelentenek a karakterek, de mellettük a karakterek közti üres karakterek is. Így a karakterlánc legelejét nem az első karakter határozza meg, hanem az első karakter előtti üres karakter, és azt is mondjuk `ab` karakterek között van egy üres karakter.

A következőkben a feldolgozott karakterláncról feltételezzük, hogy az egy egysoros szövegre terjed ki. Így a karakterláncunk végét mindig az újsor karakter előtti üres karakter jelenti.

Olyan feldolgozásokról, amelyeknél egyszerre több sorban keresünk (*multiline* vagy *singleline* keresés) ebben a tankönyvben nem beszélünk. Azt fogjuk mondani, hogy egy szövegsorban keresünk. Ez az alapszintű shell szkriptekhez elegendő.

Egyedi karakterekre való illesztések

A `c` kifejezés a **c** karakterre illeszkedik, ha c nem metakarakter, a `\c` kifejezés **c** karakterre illeszkedik ha c metakarakter.

Így például `abc` olyan minta amely az **abc** sorozatra illeszthető, és a következő láncban az illesztés a következő: xyz**abc**xyz**abc** . Az `a` minta bármilyen láncra illeszkedik ha található benne egy `a` karakter.

Ha a `^` egy metakarakter, akkor jelenlétét az `a` betű előtt ezzel a mintával fogjuk keresni: `\^a` ami illeszthető az következő sor egy részsorozatára: abc^**a**bc .

A . metakarakter

A pont bármely karakterre illeszkedik. A mintának az a karaktere ahol előfordul bármilyen karakterre illeszthető. A `.` illeszkedik akár az `a`, akár a `b` karakterekre és egy hosszabb láncban bármely karakterre egyenként. A `..` minta az **ab** illetve **xy** -ra is illeszkedik, az `a.c` minta pedig azokra ahol az `a` és `c` között pontosan egy karakter áll, mint **abc** , **axc** , **a•c** .

A karakter halmaz és a karakter osztály

A karakter halmaz egy alternatív karakter előfordulást feltételez: például ha a mintában arra szeretnénk utalni, hogy egy bizonyos helyen előfordulhat az `a`, `b` vagy `c` betű (bármelyik a három közül) akkor a karakterhalmazt jelölő metakaraktereket használjuk. Ez egy karakterlista, amely szögletes zárójelben adunk meg: például `[abc]` . Rövidíteni karaktersorozatot a − jellel lehet, például `[a-z]` a kisbetűk felsorolását jelenti Így a − jel a szögletes zárójel belsejében speciális jelentésű lesz, de amennyiben listában van első vagy utolsó karakternek kell tenni, a `]`-t pedig elsőnek, mert ez is speciális jelentésű: ő a halmaz záró.

Például:

`[abc]` az `a` vagy `b` vagy `c` karaktert jelenti,
`[a-z]` egy kisbetűt jelent,
`[0-9]` egy számjegyet jelent,
`[-a]` az a betűt és a kötőjelet jelenti, mert az itt a kötőjel az első helyen áll.

Ha a lista `^` -el kezdődik, akkor a komplementer karakterhalmazt definiáljuk, `[^a-z]` jelentése: *nem kisbetű* (ha a halmazban `^` is van, akkor azt bárhová lehet írni, kivéve az első pozíciót).

A metakarakterek is saját magukat jelentik egy karakterhalmazban, nem kell a `\` vissza-per jelölést használni. Így `[a.]` a valódi pontot vagy az `a` karaktereket keresi.

Az `ab[0-9][^xyz]` minta jelentése: az `ab` karakterek után számjegy jön, utána pedig nem következik sem `x`, sem `y`, sem `z`. Például w**ab6c**zyz sorozat egy részére illeszkedik, de a wab6xzyz -ra nem.

A POSIX standard tulajdonképpen "szögletes zárójel kifejezés"-nek (*bracket expression*) nevezi a karakter halmazt, és a felsoroláson kívül használható még az alábbi jelölés, un. karakter osztályok (*character class*) megadására.

A szintaxis `[: :]` jelek közé zárt halmaz név, ezek a nevek a C nyelvből ismert osztályok: `alnum` - alfanumérikus karakter; `digit` - számjegy; `punct` - punktuációs karakter; `alpha` - alphabetikus – csak betűk; `space` - szóköz; `blank` - üres karakterek: szóköz, sorköz, tabulátor; `lower` - kisbetűk; `upper` - nagybetűk; `cntrl` - kontrol karakterek; `print` nyomtathatóak.

Például:

`[[:cntrl:]]` egy kontrol karaktert jelent,
`[[:digit:]]` egy számjegyet,
`[[:lower:]][[:upper:]]` egy kisbetű után egy nagybetűt.

Csoportosítás és alternálás: ( ) és |

A mintában a karakterek egymás után következnek, balról jobbra, az egymás után következő karakter sorozatokat szekvenciának nevezzük. A *szekvenciákon* belüli al-sorozatokat csoportosítani lehet a `( )` metakarakterekkel. Ilyenkor a csoportosított rész egy összefüggő entitást fog jelenteni. Így a `x(def)y` minta továbbra is a látható `x`,`d`,`e`,`f`,`y` karakterek sorozatát jelenti, de a kiemelt `(def)` részre külön hivatkozhatunk bizonyos programokban.

A zárójellel csoportosított kifejezést, akárcsak egy egyedi karaktert *atomnak* nevezzük.

Amennyiben egy mintában alternatív szekvenciákat akarunk definiálni, tehát vagy az egyik vagy a másik illesztését várjuk, akkor a `| metakaraktert` használjuk az alternatívák között.

`ab|cd` jelentése: vagy az **ab** sorozat, vagy a **cd** állhat azon a helyen, a motor először az `ab`-t, utána a `cd`-et próbálja illeszteni.

Például ha egy dátumban az október hónap az October, Oct. vagy 10. szövegekkel szerepelhet, akkor abban a kifejezésben ami bármelyikre illeszkedhet ezt írom: `October|Oct\.|10\.` , természetesen a teljes dátumra illeszkedő kifejezésben ez majd csoportosítva szerepel: `(October|Okt\.|10\.)` .

Ismétlés, intervallum

Ismétlődő karaktereket (vagy atomokat) az alábbi metakarakterekkel határozhatunk meg: `*`, `+`, `?` amelyeket az ismétlődő karakter után írunk a kifejezésben. Jelentésük az alábbi:

`*` az előtte álló karakter nulla vagy akárhányszor ismétlődhet,
`+` az előtte álló karakter legalább egyszer vagy akárhányszor jelenik meg,
`?` az előtte álló karakter opcionálisan, tehát egyszer sem vagy pontosan egyszer jelenik meg.

Ezeket a metakaraktereket *kvantoroknak* is nevezzük. Látható, hogy nem pontos számú ismétlődést, határoznak meg. Az a `*` minta olyan karakterláncokra illeszkedik amelyekben "akárhányszor" fordul elő az a karakter: tehát nulla, egy, kettő stb. Így illeszkedik az **a** , **aa** , **aaa**, **bac** karakterláncokra, de a **b** , **c** , **x** karakterláncokra is, mert az a ezekben is "nullászor" megvan, vagy értelmezhetjük úgy is, hogy megvan a karakterek előtt álló üres sztringben.

Egy fontos észrevétel a reguláris kifejezés motor működésével kapcsolatban: az `a*` minta a következő láncra így illeszkedik: **aaaaaaa**xyz , tehát az illesztés nem a második a karakteren, hanem az elsőtől lehető legtávolabbi a karakteren áll le. Ezért a `* kvantort` mohó kvantornak nevezzük.

A `.*` minta olyan láncot jelöl, amiben bármely karakter akárhányszor előfordulhat: tehát az üres láncra és a nagyon hosszú, bármit tartalmazóra is illeszkedik. A `*` mohósága miatt óvatosan kell használni: `a.*a` például az első a-tól a legmesszebb levőig illeszkedik.

Egy idézőjelben levő szöveg kikeresése egy nagy szövegből jellemző példa arra, amikor a mohó kvantort az első lehetséges zárulás pontján le akarjuk állítani: a "abc•def"•"xyz•ghi" szövegben csak akkor tudunk az első idézőjel párra és a benne levő szövegre illeszteni, ha az alábbi mintát használjuk: `"[^"]*"` : ez olyan karakterekre alkalmazza a `*` ismétlést amelyek "nem idézőjelek". Ha a `".*"` mintát használnánk, az a második idézett szöveg végénél állna meg.

A `+` metakarakter előtt álló karakternek legalább egyszer vagy akárhányszor kell előfordulni. Akárcsak a `*` , ez is mohó: a legtávolabbi lehetséges illesztést keresi. Az `a+` minta illeszkedik az **a** , **aa** , **aaa** , **aaaa** karaktersorokra, de olyanokra amelyekben nem fordul elő az a nem.

A `?` előtt álló karakter opcionálisan fordul elő: a mintának az `a?` helyén állhat vagy nem `a` karakter. Például ha az Anna nevet keressük reguláris kifejezéssel, és gyanítjuk, hogy román helyességgel Ana -nak is írhatták, akkor a `Ann?a` kifejezést próbáljuk illeszteni. Ez mindkettőre illeszkedik: **Anna** vagy **Ana** .

Az ismétlődést jelölő metakarakterek `()` -el csoportosított szekvenciákra is alkalmazhatóak (azok is atomok). Így például a `([0-9][a-z])+` kifejezés jelentése: egy számjegy és utána egy kisbetű következik, és ez ismétlődhet egymás után: de a sorozatnak legalább egyszer ott kell lennie, pl.: **9a** vagy **9a5b7c** (ez utóbbira a + miatt egyszer illeszkedik a kifejezés) .

Pontos ismétlődést (intervallumot) a `{ }` metakarakterekkel határozzunk meg. Az alábbi módon használjuk:

`{n}` az előtte álló karakter pontosan n-szer fordul elő (n egész szám),
`{n,}` az előtte álló karakter legalább n-szer de akárhányszor előfordulhat,i
`{n,m}` az előtte álló karakter legalább n-szer de maximum m-szer fordul elő.

Így a `[0-9]{7}` kifejezés pontosan 7 egymásutáni számjegyre illeszkedik, a `[a-z]{2,3}` pedig két vagy három egymásutáni kisbetűre.

Horgonyok

A horgonyok (*anchor*) segítségével meghatározhatjuk, hogy a minta a szövegnek csak bizonyos helyére illeszkedjen. A `^` metakarakter a sor elejére utal, a `$` pedig a sor végére. Pontosabban:

`^` a sor elején,
`$` a sor végén

található *üres karakterláncot* jelentik. A `^abc` minta olyan sorokra illeszkedik amelyeknek elején abc lánc áll, a `\.$` azokra amelyeknek végén egy pont van. Az `^[a-z]{3}$` sorban pontosan 3 kisbetű van és semmi egyéb. Ez utóbbi módszert gyakran használjuk sztringek szigorú ellenőrzésére.

A `^$` kifejezés az üres sort jelenti (a sor elején és végén levő üres karakterlánc egymás mellett van).

A visszautalás

Sokszor olyan mintákat keresünk, amelyeknél egy előforduló karakter szekvencia megismétlődik a keresett mintában. Ilyenkor az első előfordulás helyét megjelöljük, erre a `()` -el való csoportosítást használjuk, és a `\n` (n egész szám, tehát `\1`, `\2` , stb.) jelöléssel utalunk rá vissza a kifejezésben. Az `([0-9])cd\1` jelentése: egy számjegy, utána `cd` majd ugyanaz a számjegy még egyszer.

Vagy: "a sor végén két ugyanolyan kisbetű mint a sor elején levő kettő, köztük pedig akármi" mintát így írjuk le: `^([a-z]{2}).*\1$` .

A `\1`, `\2`, `\3` , ... jelölés a zárójelezett részek számára utal balról jobbra: a következő: `([a-z])([a-z])\2\1` mintai a következő láncokra illeszkedik: **abbc** , **xyyx** , **cddc** .

További vissza-per szekvenciák

A reguláris kifejezések terminológiájában "szavakat alkotó" karakterek azok, amelyekből változónevek, azonosítók épülhetnek fel a C vagy más programozási nyelvekben. Ez pontosan az alábbi halmazt jelenti: `[a-zA-Z0-9_]` (betűk, számjegyek és a _). Ezeket fogjuk a továbbiakban szavakat alkotó (*word*: a gépi nyelvekben használt szavakról van szó) karaktereknek nevezni. Az alábbi metakarakter szekvenciák azt segítik elő, hogy azonosítókat, kulcsszavakat keressünk ki könnyen egy szövegből.

Így a `\` az **egrep** által használt reguláris kifejezésekben, ha utána az alábbi karakterek vannak, a következő jelentésekkel bír:

`\b` szóhatár (*boundary*): egy word karakter és egy nem word karakter közti üres lánc
`\B` nem szóhatár: két word karakter közti üres lánc
`\>` üres karakterlánc a szóvégén
`\<` üres karakterlánc a szó elején
`\w` szó alkotó karakter: ugyanaz mint: `[a-zA-Z0-9_]` vagy `[[:alnum:]]`
`\W` nem szó alkotó karakter : ugyanaz mint `[^a-zA-Z0-9_]` vagy `[^[:alnum:]]`

Az következő szövegben: Alkalmas•alma•hatalma a `\Balma\B` kifejezés az első alma-ra, `\balma\b` a másodikra, `\Balma\b` pedig a harmadikra illeszkedik: Alk**almas**•**alma**•hat**alma** .

Összefoglaló tábázat

Összefoglalva egy táblázatban a bővített (*extended*) reguláris kifejezések metakarakterei az alábbiak:

7.1. táblázat - A bővített reguláris kifejezések metakarakterei

Megnevezés	Metakarakter	Jelentése
Bármely karakter	.	bármilyen karakterre illeszkedik

Megnevezés	Metakarakter	Jelentése
Kvantorok	*	az előtte levő atom ismétlődik akárhányszor (lehet 0 is)
	?	az előtte levő atom egyszer vagy egyszer sem fordul elő
	+	legalább egyszer, de akárhányszor legalább egyszer vagy akárhányszor ismétlődik
Intervallum	{n}	az előtte levő atom pont n-szer ismétlődik
	{n,}	az előtte levő atom legalább n-szer de akárhányszor ismétlődik
	{n,m}	az előtte levő atom legalább n-szer de nem több mint m-szer ismétlődik
Horgonyok	^	a sor eleje előtt levő üres sztringre illeszkedik
	$	a sor végén levő üres sztringre illeszkedik
Csoportosító	( )	csoportosítja egy alkifejezés elemeit
Alternálás	|	alternálás, vagy a jobb, vagy a bal oldalon levő kifejezés illeszkedik
Karakter halmaz és osztály	[ ]	karakter osztály vagy halmaz kijelölő
Visszautalás	\n	visszautal egy ()-lel csoportosított sorozatra: \1 az elsőre, \2 a másodikra, stb.
Vissza-per szekvenciák	\	Az egrep esetében ezeken kívül még használhatóak az előző fejezetben [96] említett vissza-per sorozatok.

Alap szintű (basic) reguláris kifejezések

Néhány hagyományos program esetében használjuk őket: **expr**, **grep**, **sed** (a sed és grep esetében lehet bővítetteket is használni).

Ezeknél a kifejezéseknél a `?`, `+`, `{`, `|`, `(`, és `)` metakaraktereket vissza-per jelöléssel kell használni tehát: `\?`, `\+`, `\{`, `\|`, `\(`, és `\)` -t írunk.

Így pl:. a "legalább egy a betű" minta így fog kinézni: `a\+`, az "a vagy pontosan 3 b" pedig `a\|b\{3\}` ; az a minta amelyben megjelöljük visszautalás miatt a 3 egymás utáni kisbetűt pedig így: `\([a-z]\{3\}\)` .

Ezekben a kifejezésekben nem használhatóak a vissza-per szóhatár szekvenciák.

Ebben a tananyagban nem használunk ilyen kifejezéseket, de a gyakorlatban régi UNIX parancsok szintaxisában előfordulhatnak, például az **expr** és **grep** esetében.

A grep és egrep használata

A **grep** parancs kikeresi a bementi szöveges állományokból azokat a sorokat amelyekre a megadott minta illeszkedik, és kilistázza őket különböző opciókkal. Az egyik leggyakrabban

használt parancs a Unix rendszereken. **egrep** nevű változata (vagy `-E` kapcsolóval indított **grep**) használja a bővített reguláris kifejezéseket, mi is így fogjuk használni.

Alább a Linux rendszereken általánosan használt GNU **egrep** változat kapcsolóit ismertetjük (klasszikus UNIX rendszerek **egrep** parancsaiban nem mindegyik működik ezek közül).

Parancsor formátuma:

`egrep` [opciók] *`minta fájl(ok)`*...

Interaktív munkához a `-color=auto` opcióval indítva a sorok listázásakor színessel írja ki az illesztés helyét, ha a terminál ezt támogatja. Ezt beállíthatjuk egy környezeti változóval is, ehhez írjuk az alábbi parancsot a `.bashrc` állományunk végére:

```
export GREP_OPTIONS='--color=auto'
```

7.2. táblázat - A grep és egrep fontosabb opciói

Opció	Jelentés
`-c`	Kiírja azon sorok számát, amelyben találat van (ilyenkor nem írja ki a találatokat). Találat lehet több is, mert egy sorban lehet pl. kettő vagy több.
`-E`	A **grep** bővített kifejezésekkel dolgozik (az **egrep** esetében nem kell megadni).
`-e minta`	A mintát így is meg lehet adni: feltétlenül így kell megadni, ha a minta – jellel kezdődik, pl.: `egrep -e '-q' file`
`-P`	Perl kifejezésekkel dolgozik (csak a **grep** esetében használható).
`-f fájl`	A mintát egy fájl egymás utáni soraiból veszi. Minden minta találatát keresi.
`-i`	Kis és nagybetű közti különbség nem számít.
`-l`	Ha több állományban keres, kiírja azok neveit amelyben megvan a minta.
`-L`	Ha több állományban keres, kiírja azok neveit amelyben nincs meg a minta.
`-m szám`	*szám* darab találat után leáll, a szám egész értékű.
`-n`	A találat elé kiírja annak sorszámát.
`-o`	Csak az illesztést vagy illesztéseket írja ki, több illesztés esetén külön sorba.
`-q`	Nem ír ki semmit. A kilépési érték jelzi csak, hogy talált vagy nem.
`-r vagy -R`	Rekurzívan végigjár minden könyvtárat és állományt a megadott könyvtár alatt.
`-s`	Nem ír ki hiba üzeneteket azokról az állományokról amelyeket nem tud elolvasni.

Opció	Jelentés
`-v`	Inverz kiírás: azokat a sorokat írja ki amelyekben nem volt illesztés.
`-w`	Csak azokat a karakterláncokat tekinti találatoknak amelyek teljes szót alkotnak (szó elválasztó van a két szélükön, és a szó számokból, betűkből és _ karakterekből épül fel.
`-x`	Csak azokat az illesztéseket tekinti találatnak amelyek teljes sort alkotnak.

Az **egrep** igaz értéket ad vissza a `?` változóban ha talált legalább egy illesztést, és hamisat ha nem talált: ez felhasználható if szerkezetekben, ha feltételként egy **egrep**-et futtatunk.

Példák az opciók használatára a következő alfejezetben.

Példák reguláris kifejezések használatára az egrep-el

Keresések

A leggyakrabban egyszerű szövegkereséshez használjuk a reguláris kifejezéseket az **egrep**-el.

Például a meg szeretnénk nézni, hogy a `get_puff()` nevű C nyelvben írt függvény milyen sorokban fordul elő a `circbuff.c` fájlban:

```
$ egrep  get_puff circbuff.c
char get_puff ( struct puffer * p );
printf ("fiu: -1 -> buffer,  c = %c\n", get_puff ( puff ) );
char get_puff ( struct puffer * p )
$
```

Ha több fájlban akarunk keresni, akkor a keresett fájlneveket metakarakterekkel adhatjuk meg:

```
$ egrep   get_puff *.c
circbuff.c:char get_puff ( struct puffer * p );
circbuff.c:printf ("fiu: -1 -> buffer,  c = %c\n", get_puf  ( puff ) );
circbuff.c:char get_puff ( struct puffer * p )
pufferfun.c:char get_puff ( struct puffer * p )
puffertest.c:    printf ("%c\n", get_puff(puff));
$
```

Látható, hogy ebben az esetben a fájlneveket is kiírja a találatok elé. Ha csak a fájlnevekre van szükség, a `-l` opcióval csak ezeket adja vissza:

```
$ egrep -l  get_puff *.c
circbuff.c
pufferfun.c
puffertest.c
$
```

Ez jól használható arra, hogy a fájllistát változóba tegyük:

```
$ get_puff=$(egrep -l  get_puff *.c)
$ echo $get_puff
circbuff.c pufferfun.c puffertest.c
$
```

és átadjuk pl. a **vim** szövegszerkesztő parancssorának, azt 3 fájlt fogja megnyitni, amelyben használjuk ezt a függvényt:

```
$ vim $(egrep -l  get_puff *.c)
```

Találat számlálás

A GNU **egrep** kíválóan alkalmas egy szövegkeresés találatainak megszámlálására vagy a szöveg feldarabolására valamilyen szempont szerint.

Angol nyelvű példát használva, tekintsük az alábbi kis szöveget:

```
$ cat szoveg.txt
i'm tired of being what you want me to be
feeling so faithless
lost under the surface
i don't know what you're expecting of me
put under the pressure
of walking in your shoes
[caught in the undertow / just caught in the undertow]
every step that i take is another mistake to you
$
```

Hányszor fordul elő a szövegben a *under* karaktersorozat:

```
$ egrep under szoveg.txt
lost under the surface
put under the pressure
[caught in the undertow / just caught in the undertow]
$
```

Ha nem karaktersorozatot, hanem kisbetűs szót keresünk amelynél szóhatár van a szó két szélen, akkor így keresünk:

```
$ egrep '\bunder\b' szoveg.txt
lost under the surface
put under the pressure
$
```

Ugyanezt megkapjuk ha a GNU **egrep** `-w` opcióját használjuk:

```
$ egrep -w under szoveg.txt
```

```
lost under the surface
put under the pressure
$
```

Ha az érdekel, hogy hányszor fordul elő ez a karaktersorozat (*under*), megszámoljuk a találatokat:

```
$ egrep -o '\bunder\b' szoveg.txt
under
under
$ egrep -o '\bunder\b' szoveg.txt | wc -l
2
$
```

Ezt már könnyű változóba írni:

```
$ egrep -o '\bunder\b' szoveg.txt | wc -l
2
$ under=$( egrep -o '\bunder\b' szoveg.txt | wc -l )
$ echo Az under szónak ${under} előfordulása van a szövegben
Az under szónak 2 előfordulása van a szövegben
$
```

Ha az érdekelne, melyek a kisbetűs szavak a szövegben, az alábbi keresést használhatjuk (csak az első 20 találatot írjuk ki a **head**-el):

```
$ egrep -io '\b[a-z]+\b' szoveg.txt | head -20
i
m
tired
of
being
what
you
want
me
to
be
feeling
so
faithless
lost
under
the
surface
i
don
$
```

Mivel ebben a listában minden szó benne van, és egyes szavak többször fordulnak elő, ha az érdeklne, hogy hány szó fordul elő (tehát a többszörös előfordulástól meg kell szabadulni), a **sort** és **uniq** parancsok használatával ezt megtehetjük. A **sort** rendezi a találatokat, az azonos

szavak így egymás után kerülnek, ebből a **uniq** kiszűri az egymás utáni azonos sorokat, csak egy példányt hagy meg, így a listában a szavak rendezetten és egy példányban fordulnak elő:

```
$ egrep -io '\b[a-z]+\b' szoveg.txt | sort | uniq | head -15
another
be
being
caught
don
every
expecting
faithless
feeling
i
in
is
just
know
lost
$
```

Így már könnyű megszámolni és egy változóba írni, hány kisbetűs szó fordul elő ebben a szövegben:

```
$ egrep -io '\b[a-z]+\b' szoveg.txt | sort | uniq | wc -l
39
$ szam=$( egrep -io '\b[a-z]+\b' szoveg.txt | sort | uniq | wc -l )
$ echo $szam
39
$
```

Néhány egyszerű művelettel statisztikát is készíthetünk szövegeinkről.

A GNU **egrep** szépen működik UTF-8 magyar karakterekkel is, ami nem minden rendszerről mondható el:

```
$ export LANG=hu_HU.UTF-8
$ echo 'áéíóú' | egrep 'éí'
áéíóú
$
```

Szövegek vágása

Szövegfájlok szűrése, bizonyos feltétel szerinti rész-szövegek kiírása egy szövegből is szépen megoldható az **egrep**-pel. Pl. az alábbi listából, amelyben a mező elválasztó karakterek szóközök:

```
$ cat lista.csv
Tipus  db   Ar  raktar
A   2   22.50    i
```

```
B   1   17.80    i
F   2   17.10    n
G   3   1.12    i
H   2   12.10   i
O   1   12.00    n
$
```

így vághatjuk ki az első mezőt:

```
$ egrep -o '^[A-Z]' lista.csv
T
A
B
F
G
H
O
$
```

Erre a feladatra az **awk** parancs lesz a megfelelőbb (lásd az Awk nyelvet [133] leíró fejezetet).

Ha például automatikusan ki szeretnénk szedni a hivatkozásokat egy weben levő weblapból az alábbi tehetjük (a **head**-el helyszűke miatt csak az első 5 találatot írjuk ki):

```
$ wget -O - http://ubuntu.hu 2>/dev/null \
                          | egrep -o 'href="[^"]+"' | head -5
href="http://ubuntu.hu/node"
href="http://ubuntu.hu/rss.xml"
href="/sites/default/files/udtheme_favicon.ico"
href="/modules/aggregator/aggregator.css?w"
href="/modules/book/book.css?w"
$
```

A **wget** parancs Internetes protokollokat használva fájlokat tölt le, a `-O -` kapcsolóval a letöltött fajlt a standard kimenetre írja (lásd a **wget** kézikönyvét). A `href` elem után olyan sorozatokat keresünk amelyek 2 idézőjel közt idézőjel nélküli sorozatokat tartalmaznak (a `2>/dev/null` átirányítás a wget standard hibakimentre írt üzeneteit szűri ki a terminálról). Az első szűrés után már nem lesz probléma kivágni magát az URL-t.

Megjegyzés: a feladat nem ennyire egyszerű ha weblapokban keresünk, mivel a weblapok szerkezete nem mindig egyetlen fájlra korlátozódik. Ugyanakkor az **egrep** keresése mindig egy sorra korlátozódik a feldolgozott fájlban. A HTML szövegeknél az elemek átnyúlhatnak egyik sorból a másikba. A teljes megoldáshoz Perl típusú, egyszerre több sorban is kereső kifejezéseket kell használni.

Sztringek tesztelése

Amint azt már említettük, héjprogramokban egyszerű sztringek tesztelését végezhetjük el az egrep-el. A `-q` (*quiet*) opciót használva az **egrep** semmit sem fog kiírni, de találat esetén igazra állítja a `?` változót, így azt kihasználhatjuk vezérlő szerkezetekben. A tesztelt sztringet a parancs bemenetére küldjük csővezetéken.

Több programunkban használtunk egész számot bemeneti paraméterként. Ennek alakját könnyen tesztelhetjük, és azt a korlátot állítjuk, hogy sor eleje és vége közt ne legyenek csak számjegyek:

```
$ echo 112 | egrep '^[0-9]+$'
112
$
```

amennyiben előjellel is jöhet a szám:

```
$ echo +12345 | egrep '^[+-]?[0-9]+$'
+12345
$
```

A szkriptekben a tesztet így végezhetjük el, ha pl. az első paraméterben várunk egész számot:

```
if ! echo "$1" | egrep -q '^[0-9]+$'
then
 echo Az első paraméter egész szám!
 exit 1
fi
```

ugyanezt modern shell változatokban (pl. Bash 2.05 utáni verziók) elvégezhetjük a beépített `[[ ]]` szerkezettel.

A gyakori tesztekre függvényeket írhatunk, amelyekben pl. egész szám teszt esetén nem csak a számalakot, hanem a határokat is tesztelhetjük más módszerekkel.

8. fejezet - Különböző héj szerkezetek

Bevezető

A héj igen változatos szerkezeteket biztosít feladataink kifejezéséhez. Ebben a fejezetben néhányat emelünk ki ezek közül, olyanokat amelyeket gyakran használunk mindennapi feladataink megoldásában. Így a fejezet nem csoportosul egyetlen téma köré, változatos nyelvi szerkezeteket járunk be.

Függvények

Definíció

A függvények általában névvel ellátott kódszekvenciát jelentenek a programozási nyelvekben. A shell esetében is így van, mivel a nyelv interpretált, a függvény nevét és kódját felhasználás előtt, a program szövegében kell definiálni. Ez történhet a szkript elején vagy egy külön állományban, amelyet függvényhívás előtt beolvasunk.

A függvény kód szekvenciáját kapcsos zárójelek közt definiáljuk, a `{ }`-ek közti részt a héj kód blokknak (*code block*) nevezi.

Ha külön állományban definiáljuk a függvényeket, akkor a **.** (*dot*) paranccsal [56] kell őket az főprogramot tartalmazó állományba illeszteni.

A függvények úgy viselkednek mint egy-egy külön kis shell program. Argumentumaikat is ugyanúgy kapják (nem kell argumentumlistát definiálni) és a függvény testében az `1`, `2`, stb. változókkal érhetők el (`$1`, `$2`).

A függvény definíció szintaxisa:

```
function nev ()
{
  #ide kerül a függvény kódja
}
```

A nyitó kapcsos zárójel a név sorában is szerepelhet:

```
function nev () {
  #ide kerül a függvény kódja
}
```

Ugyanakkor a `function` kulcsszó akár el is hagyható, tehát a definíció ilyen is lehet:

```
nev () {
  #ide kerül a függvény kódja
}
```

Például:

```
function equal()
{
    if [ "$1" = "$2" ];then
        return 0
    else
        return 1
    fi
}
```

Fontos: a **return** parancs a függvény kilépési értékét küldi vissza, ez hívó héj `?` változóját állítja be, ennek értéke lesz a **return** által visszaadott kód, és azt a `$?` hivatkozással érhetjük el a hívó szekvenciában, tehát nem eredmény visszatérítésre használjuk!

Így a függvény használata:

```
if equal "$a" "$b"
then
    echo egyenloek
fi
```

A függvények megosztják változóikat a hívó héjjal, vigyázni kell tehát a kilépési értékekre illetve a változómódosításra.

Ezek a megosztott változók un. *globális változók*. Ennek következtében a függvényben láthatóak a hívó szkript változói, és a a függvényben létrehozott változók láthatóak lesznek a hívó szkriptben.

Ha az `equal` függvényt egy `equal.sh` állományba írjuk, akkor a **.** paranccsal az alábbi módon használhatjuk az aktív héjban:

```
$. equal.sh
$equal "a" "b"
$echo $?
1
```

A **.** beolvassa a definíciót, és utána lehet használni ebből a héjból. Függvénydefiníciót törölni az **unset** parancsot használva lehet:

```
$unset equal
$equal "a" "b"
-bash: equal: command not found
$
```

Az **unset** két opciót fogad el, `-v` ha csak a megadott változóneveket, `-f` ha csak megadott függvényneveket akarunk törölni. Opció nélkül mindkettőt próbálja törölni.

A függvénydefiníciót megadhatjuk egyetlen parancssorban:

```
$ function hello () { echo "Helló" ; return 0; }
$ hello
Helló
```

Ilyenkor az utolsó parancs után is meg kell jelennie a `;` elválasztónak.

Bárhogyan adnánk meg a definíciót, annak meg kell jelennie a kódban a függvény használata előtt (a definíció bárhol lehet, nem kötelező a szkript elejére tenni)!

A függvények nem lehetnek üresek, tehát legalább egy parancsot kell tartalmazzanak.

Függvények érték visszatérítése

Az eredmények visszaadását a hívó szkriptnek az alábbi módszerekkel oldjuk meg:

1. globális változó beállítása

2. nyomtatás a függvényben és parancssor helyettesítés a hívásnál

Az alábbi függvényt a `getfname.sh` állományban definiáljuk:

```
#visszaadja az állománynevet, levágva az extenziót
#egy bemeneti paramétere van
#ha több pont van a névben, az utolsótól a végéig vágja le
#a karaktereket
#ha nincs . , akkor a teljes nevet adja vissza

function getfname()
{

    #elhagyjuk az utolsó ponttól
    #a név végéig terjedő szubsztringet

    echo "$1" | sed -r 's/(.+)(\.[^.]*)$/\1/'

    return
}
```

A **sed** parancs használatát lásd a parancsot bemutató fejezetben [118].

A függvény használata:

```
#!/bin/bash

. getfname.sh             #függvény definíció betöltése

fname="file.txt"

name=$(getfname $fname)

echo $name
```

Az alábbi esetben viszont a függvényben létrehozott `line` változón keresztül tér vissza az érték (a függvénydefiníció a `getline.sh` fájlban található):

```
#egy sort olvas be és azt a line változóban adja vissza
#a főprogramnak vagy más függvénynek
#ha a felhasználó  Ctrl-D-t üt be, akkor kilép

function getline
{
    line=""
    if ! read -p "Kérek egy sztringet:" line
    then
        echo "Olvasás hiba";
        return 1
    fi
    return 0
}
```

Használata:

```
$.  getline.sh
$ if getline ; then echo $line ; fi
Kérek egy sztringet:piros
piros
$
```

Környezettel kapcsolatos kérdések

A függvény megkapja a hívó héj munka könyvtárát, környezeti változóit. A függvényben létrehozott változók láthatóak lesznek a hívó héjban, és a függvény is látja az ott előzőleg létrehozott változókat.

Lokális változók használata

A Bash megengedi lokális változók használatát is. Ezeket a `local` kulcsszóval adjuk meg:

```
function teszt ()
{
    local valtozo=3

    local eredmeny=$(( valtozo * $1 ))

    echo $eredmeny

}
```

Ebben az esetben a `valtozo` és `eredmeny` változók nem lesz láthatóak a hívó héjban, még csak nem is írják felül az ott található ugyanolyan nevű változókat:

```
$ . teszt.sh
```

```
$ eredmeny=33
$ teszt 8
24
$ echo $eredmeny
33
$ echo $valtozo

$
```

Rekurzív hívások

Rekurzív hívás lehetséges. Azonban ha a rekurzív feldolgozás számításokat használ, amelyek paraméterként kell átadódjanak egyik szintről a másikra, meglehetősen nyakatekert kódot kapunk. Az alábbi függvény faktoriálist számol, és az eredményt egy `fact` nevű változóban téríti vissza:

```
#egy paraméterrel kell meghívni
#n! -t számol
function calc_fact1()
{
    if (( $1 == 0 )) ; then
        fact=1                    #ha n == 0 , akkor fact=1
        return
    else
        calc_fact1 $(( $1 - 1 ))   # (n-1)! marad a
        # fact-ban változoban
        fact=$((  fact  *   $1  )) # fact n! lesz
    fi
    return
}
```

Ugyanez megoldható úgy is, ha az eredményt a `?` változóban adjuk vissza, persze ezzel a `$?` -t nem rendeltetésének megfelelően használjuk. Az alábbi példában az `n` változót lokálisként használjuk (rekurzív hívások esetében ez a Bash tervezőinek ajánlata), így minden hívás esetében ismét létrejön:

```
function calc_fact()
{
    local n=$1       #n lokális

    if (( n == 0 )) ; then
        return 1
    else
        calc_fact $(( n - 1 ))    # (n-1)!
        return $(( $? * n ))      #  n!
    fi
}
```

Megjegyzendő, hogy a `?` változóban a Bash esetében csak 255-nél kisebb egész számot lehet visszatéríteni (tehát annál nagyobb számokra a fenti függvény nem használható).

A rekurzió a héjban lassú, erőforrás igényes és ezért nem javasolt a használata.

Kód blokkok használata

A `{ }` elhatárolást függvényeken kívül is lehet használni kódszekvenciák elhatárolására. A `{ }` zárójelek közti szekvencia úgy fog viselkedni, mint egy névtelen függvény.

A `{ }`-ek közti parancsok közös standard bemenet és kimenetet használnak, így alkalmasak közös átirányítási feladatok megoldására: a blokkban található parancsok kimenetét egy állományba lehet irányítani, illetve ugyanazt az állományt több paranccsal lehet végigolvasni.

Amennyiben adott az alábbi számokat soronként tartalmazó `be.txt` állomány:

```
2
4
1
2
3
4
```

és ebből az első két számot egy számítás paramétereiként akarjuk használni, a többit pedig a fájl végéig összeadva egy harmadik paraméterként, az alábbi szkripttel megtehetjük:

```
#!/bin/bash

{
    read szam1                      #első számjegy
    read szam2                      #Második számjegy
    s=0
    while read szam                 #ciklusban összeadjuk a többit
    do
        s=$(( s + szam ))
    done

    echo $(( szam1 * s / szam2 ))   #a számítás elvégzése
                        #a Bash-ben ez csak egész számokkal működik

} < be.txt
```

Mivel a `{ }` közötti parancsok közös bemenetet használnak, és erre egy fájlt irányítottunk, egymás után olvassák a parancsok a számokat. Ugyanakkor ha szükséges, a szerkezet kimenetét is egy közösen használt állományba lehet irányítani.

A kód blokkban láthatóak a szkript változói, és ha újakat hozunk létre azok láthatóak lesznek a blokkon kívül is. Lokális változók nem használhatók kód blokkban és a héj nem indít külön folyamatot ennek végrehajtására.

A határokat jelző `{ }` kapcsos zárójelek kulcsszónak számítanak (tehát előttük és utánuk elválasztó - implicit szóköz - áll, az utolsó parancs után pedig kötelező a záró `;` pontosvessző használata, ha nincs újsor. A fenti szkript egy parancssorban így írható:

```
$ { read szam1; read szam2; s=0; while  read szam ; \
  do  s=$(( s + szam )); done; \
  echo $(( szam1 * s / szam2 )) ; } < be.txt
```

Aritmetikai kiértékelés: számítások a (( )) szerkezettel

Amint láttuk, az **expr** segítségéve elég kényelmetlen számolni. A 90-es évek fejlesztése során egyes héj értelmezők tartalmaznak néhány olyan kiterjesztést, amely könnyűvé teszi az egész számokkal való számítást (a POSIX szabvány is tartalmazza ezt, így a Korn héj is, valamint 2.04 verziótól kezdve a Bash is). Jelenlétére tehát nem lehet mindig számítani és elvileg hordozható programok esetén kerülni kell.

Ezen kiterjesztések közül a két zárójellel határolt aritmetikai kiértékelést emeljük ki. Egy másik megoldás ugyanerre a feladatra a héjak **let** parancsa, amely ezzel ekvivalens.

A ((…)) artimetikai kiértékelés szerkezet használata:

```
valtozo=$(( aritmetikai műveletek ))
```

vagy, használat közben a terminálon:

```
$ x=$(( 22 + 6 ))
$ echo $x
28
$
```

A két határoló közti területen aritmetikai műveleteket lehet végezni, ugyanolyan szintaxissal mint a C nyelven történő programozásnál.

A szerkezetet a teszteknél is lehet használni egész számokkal végzett tesztek esetében:

```
i=4
while (( i > 2 ))
do
    echo $i
    (( i-- ))
done
```

Szabályok a (( )) használatára:

- Az operátorok a C nyelvből valóak, minden operátor használható.
- A Bash csak egész számokkal dolgozik. Kiértékeléskor a Korn héj dupla pontosságú valós számokat használ (a Korn héj tud valós változókkal dolgozni).
- A `$` jelet nem kell változók előtt használni, de lehet. Kivételt képeznek a héj speciális változói, azok előtt viszont kell. Tehát:

```
b=$(( a + 2 ))
```

, de:

```
b=$(( $a + 2 ))
```

is helyes. A

```
b=$(( $1 + 2 ))
```

összeadást viszont már csak a `$` jel segítségével tudjuk elvégezni ha a parancssor első argumentumát akarjuk használni.

- A C nyelv hozzárendelő operátorai is használhatóak, például:

```
(( x+=2 ))
```

ebben az esetben viszont a `$` jel nem használható a változó neve előtt:

```
(( $x+=2 ))  #helytelen !!!
```

- Zárójelezés használható.
- A relációs operátorok esetében az igaz értéke 1 – de csak a (( )) belsejében. A tesztek esetében a (( )) -ből való kilépés után a héj a `?` változót fordított értékre állítja. Tehát:

```
$ (( 2 > 1 ))
$ echo $?
0
```

így programban az alábbi módon használható:

```
i=4
while (( i > 2 ))
do
    echo $i
    (( i-- ))
done
```

- Nem kell feltétlenül ügyelni a szigorú, héjnál megszokott szóköz elválasztóra operátorok és változók között, tehát írhatunk `((2+2))` -t vagy `(( 2 + 2 ))` -t is.
- A hatványozás operátora a `**` , ebben az esetben a két `*` karakternek egymás mellett kell állnia.

Az alábbi szkript néhány példát tartalmaz:

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

```
#példák számításokra és kifejezés kiértékelésre a (( )) -tel

#néhány segédváltozó

a=1
b=2
z=0

#használhatók a C nyelv operátorai, a változónevek
#előtt nem kell a $ szimbólumot használni

echo $(( a + 1))

#az sem számít, ha nem tartjuk be feltétlenül a héj által
#kért szóközöket

echo $((a+1))
echo $(( a++ ))

#van egy eset, amikor kell $ szimbólumot
#használni a nevek előtt:
#ha a héj speciális változóit használjuk

if [ -n "$1" ];then
    echo $(( $1 + 2 ))
fi

#a hatvány operátor a **

echo $(( a**2 ))

#feltételekben lehet használni

if (( 1 < b ))
then
    echo '1 < b feltétel igaz'
fi
```

A C stílusú for ciklus

A `(( ))` zárójeles szerkezet for ciklusok előállítására is használható, éspedig a C nyelv for utasításához hasonlóan. A szintaxis az alábbi:

```
for (( i=0; i<10; i++ ))
do
    héj parancsok   #ezekben használható a $i változó
    . . .
done
```

Egy egészen egyszerű `for` ciklust tehát így írhatunk:

```
#!/bin/bash

for (( i=1; i<=10; i++ ))
do
    echo $i
done
```

Látható, hogy egész számok végigjárására épülő ciklusokban ez kényelmesebben alkalmazható, mint az előző változatú, sztring lista végégjárására épülő `for` .

Műveletek karakterláncokkal

A karakterláncok kezelésénél néha igen hasznos gyorsan elvégezni kis műveleteket, mint például: levágni egy karakterlánc elejét/végét vagy átírni egy részt a karakterláncban. Ezért amikor egy változót kiválasztunk a `${valtozo}` szintaxissal, akkor a két `{}` zárójel közt operátorokat használhatunk erre. Az operátorok az alábbiak:

8.1. táblázat - Sztring operátorok II.

Operátor	Mit végez
`${valtozo#minta}`	Ha a minta illeszkedik a változóban levő karakterlánc elejére, akkor letörli a legkisebb illeszkedést és úgy téríti vissza a karakterláncot.
`${valtozo##minta}`	Ha a minta illeszkedik a változóban levő karakterlánc elejére, akkor letörli a legnagyobb illeszkedést és úgy téríti vissza a karakterláncot.
`${valtozo%minta}`	Ha a minta illeszkedik a változóban levő karakterlánc végére, akkor letörli a legkisebb illeszkedést és úgy téríti vissza a karakterláncot.
`${valtozo%%minta}`	Ha a minta illeszkedik a változóban levő karakterlánc végére, akkor letörli a legnagyobb illeszkedést és úgy téríti vissza a karakterláncot.
`${valtozo/minta/behelyetesít}`	Behelyettesíti a változóban a mintának megfelelő részté; ha a minta #-el kezdődik a karakterlánc elején, ha % -al akkor a karakterlánc végén.

Példák:

```
$ a=abcxyzabc
$ echo ${a#abc}
xyzabc
$ echo ${a%abc}
abcxyz
$ echo ${a/xyz/abc}
abcabcabc
$
```

A duplázott %% és ## használata akkor indokolt, ha olyan mintát adunk meg, amelybe metakarakterek (`*`, `?`) is vannak, amelyek használhatóak. Ezek egy karakterlánc esetében

rövidebb vagy hosszabb illesztést is meghatározhatnak. Például, az: `abcxyzabcxyz` karakterlánc elejére a `*abc` minta (bármi és utána abc) rövidebben: `abc` (hiszen az üres karaktereklánc is bármi) vagy hosszabban: `abcxyzabc` is illeszkedhet. Így a `#` operátor a legrövidebb illesztést, a `##` pedig a leghosszabbat fogja levágni:

```
$ valtozo=abcxyzabcxyz
$ echo ${valtozo#*abc}
xyzabcxyz
$ echo ${valtozo##*abc}
xyz
$
```

Az operátorok könnyebb memorizálása miatt, a következőt tartsuk szem előtt: a # jelet az angolok mindig valamilyen számosság jelölésére használják, és a szám elé teszik, pl. #10 , a % jelet pedig mindig a szám után tesszük, pl. 100% (innen a lánc elejére illetve végére utalás).

Ha van egy változónk amiben egy fájlnév van, könnyen le tudjuk vágni a fájl típusát tartalmazó végződést, és megtartani a nevet (ha pl. más típusra kell átnevezni egy fájl):

```
$ file="szoveg.txt"
$ echo ${file%.*}
szoveg
$ nev=$( echo ${file%.*} )   #változóba írjuk a nevet
$ echo $nev
szoveg
$
```

Vagy egy hosszú fájlrendszerbeli útvonal részeit lehet gyorsan kivágni:

```
$ mypath="/elso/masodik/harmadik/szoveg.txt"
$ echo ${mypath#/*/}
masodik/harmadik/szoveg.txt
$ # vagy:
$ echo ${mypath##*/}
szoveg.txt
```

A [[ ]] szerkezet

A `[ ]` szerkezet jelenti a klasszikus **test** parancs végrehajtását, ezt a Bourne shell változatok vezették be. A Korn shell egy új teszt szerkezetet definiált 1986-ban, majd a 1993-as változatban véglegesítette ezt. A szerkezetet a Bash is átvette a 2.05 verziótól kezdve.

A `[[ ]]` a klasszikus teszthez hasonlóan használható. A belsejében a shell nem hajtja végre a shell minták kiterjesztését (`*`, `?`, stb.). Változó helyettesítést, parancs helyettesítést, ~ helyettesítést azonban végez, az idézőjeleket is ugyanúgy használhatjuk mint a parancssoron.

A klasszikus teszt `[ ]` -nek minden operátorát lehet benne használni:

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

```
#az egész számokra vonatkozó tesztek ugyanúgy használhatók
if [[ $# -ne 1 ]]          # ha $# nem egyenlő 1-el
then
    echo nincs paraméter
    exit 1
fi

file="$1"

#az állományok tulajdonságait lekérő tesztek ugyanúgy
#használhatóak

if [[ -w "$file" ]]        #írható-e a fájl?
then
    rm "$file"
else
    echo nincs ilyen fájl
fi
```

A `[ ]` teszt operátorain kívül a `[[ ]]` szerkezetben az alábbiak használhatók:

Sztring teszthez: a `-n` , `-z` , = és `!=` operátorokon kívül használható a < és > operátorok is amelyek karakter alapú összehasonlítást végeznek :

```
$ [[ a < b ]]; echo $?
0
$ [[ a > b ]]; echo $?
1
```

Az == és `!=` operátorok esetében ha az operátor jobb oldalán levő sztringet nem tesszük macskakörömbe, használhatunk shell mintákat is benne. Tehát:

```
$ [[ "ab" == a? ]] ; echo $?
0
```

igazat ad vissza, mert a jobb oldali sztring shell minta,

```
$ [[ "ab" == "a?" ]] ; echo $?
1
```

viszont hamisat, mert macskakörömben egyszerű sztring a jobb oldali.

A Bash shellben a =~ operátor használható bővített szabályos kifejezésekkel való egyeztetésre . A teszt igazat ad vissza, ha a minta illeszkedik a tesztelt sztringre. Például:

```
$ [[ "ab" =~ '^a' ]] ; echo $?
0
```

igazat ad vissza, mert a bal oldal a -val kezdődik.

A `[[ ]]` szerkezeten belül használható a zárójel, ha kifejezést akarunk elhatárolni, illetve az `&&` és `||` kifejezések közti operátorokként valamint a `!` kifejezés tagadására. Pl. azt, hogy az `f` változóban levő fájlnévhez létezik szabályos állomány és az állománynév `.txt`-ben végződik így teszteljük:

```
$ f=a.txt
$ echo a > a.txt
$ [[ -f "$f" && "$f" =~ '.txt$' ]]
$ echo $?
0
$
```

9. fejezet - A sed folyamszerkesztő

Bevezető

A szövegfeldolgozással kapcsolatos feladatok legfontosabb követelménye a szövegek automatikus, programból vezérelt szerkesztése. Ezt a követelményt biztosítja a **sed** nevű szerkesztő, amelyik "műfajában" a leginkább használt Unix parancs.

A **sed** egy speciális szövegszerkesztő, un. folyamszerkesztő program (*stream editor*). Ez azt jelenti, hogy nem interaktív módban szerkesztünk vele szöveget, hanem a bemenetére vezetjük, a program futás közben végigolvassa azt szövegsoronként és közben szerkesztési műveleteket végez rajta.

Az ilyen folyamot szerkesztő programok rendelkeznek egy parancsnyelvvel, amellyel meg lehet adni nekik a szerkesztési műveleteket. A nyelv segítségével a feldolgozás vezérlésére kis szkripteket írhatunk, ezeket a **sed** esetében *sed szkriptnek* fogjuk nevezni.

A szerkesztett szöveg fájlokba vagy egy kimeneti folyamba kerül további feldolgozásra. Így szöveges fájlok szerkesztésére vagy szűrőként valamilyen szövegfeldolgozás köztes állomásaként használjuk.

A sed parancssora

A **sed** meghívásának általános alakja:

```
sed  [opciók] 'szkript' [bemeneti_fájlok]
```

A **sed** a vezérlő szkriptet a `-e` vagy `-f` opciókkal argumentumaként kaphatja meg, illetve egy karaktersor argumentumként opciók nélkül. A 'szkript' paraméternek akkor kell szerepelnie, ha sem a `-e` sem a `-f` opciókkal nem adunk meg más szkriptet.

A feldolgozott szöveg vagy a standard bemenetre jön, vagy fájlnév argumentumként: itt akár több fájl is szerepelhet, a **sed** sorban fogja feldolgozni őket.

Az alábbi hívások tehát mind helyesek:

a. Az `elso.txt` nevű fájl kerül a bemenetre, a **sed** szkript a `-e` opció után jön. A parancssoron megadott szkript az alábbi példában egyetlen `p` nevű parancs aposztrófok közt. A `p` parancs listázza a bementre érkező sorokat, a **sed** maga is listáz, így a kimeneten (ami a standard kimenet) duplázva fogjuk látni az `elso.txt` sorait). A parancssor:

```
$ sed -e 'p' elso.txt
```

b. Ha nincs `-e` opció, a **sed** az első paraméterként megadott sztringet tekinti szkriptnek:

```
$ sed 'p' elso.txt
```

A fenti példákban a szerkesztett bemeneti fájl az `elso.txt`.

c. A **sed** a standard bemenetet olvassa, és a végrehajtó szkriptet az egyik előbb említett módon adjuk meg:

```
$ cat elso.txt | sed 'p'
```

d. A **sed** szkript egy fájlban van, ilyenkor a fájl neve elé a `-f` opció kerül, a szkript fájl pedig egy `.sed` típusú fájl (lehetne bármilyen nevű szöveges fájl, helyes `sed` parancsokkal):

```
$ sed -f szerkeszt.sed elso.txt
```

Fontosabb opciók:

9.1. táblázat - A sed opciói

Opció	A sed működése
`-e szkript`	ezt a szkriptet is hozzáfűzi a végrehajtandókhoz
`-f szkript_file`	a szkriptet fájlból veszi
`-n`	nem ír a kimenetre a mintatér (lásd következő szekció) feldolgozása után
`-r`	bővített reguláris kifejezéseket használ
`-i`	*in place* – magát a fájlt szerkeszti, tehát a bemeneti fájl módosul a merevlemezen
`-s`	*separate*: ha több fájlt adunk meg, alapértelmezésként a fájlokat egymás után, egy folyamnak tekinti: `-s` esetében viszont külön kezeli őket. A `-i`-vel a `-s` opciót is kell használni ha több fájlt akarunk egyszerre szerkeszteni.
`-g`	minden helyettesítési parancsnál alkalmazza a `g` (global) opciót (lásd alább)

Ha külön fájlba írjuk a **sed** programot, annak első sorában a **sed**-et kell bejelölni mint végrehajtó program, tehát a forrásnak így kell kinézni:

```
#!/bin/sed -f
#itt soronként jönnek sed parancsok:
1,2 p
3,$ d
```

A `-f` opció a végrehajtó programnak azt jelzi, hogy ezt a fájlt kell szkriptként értelmezni.

A sed működése

A **sed** működés közben követi a vezérlő szkriptben található parancsokat. Egy **sed** szkript egy vagy több egymás utáni parancsból áll. A sed szkriptben található parancsok formátuma:

```
[cím] parancs [opciók]
```

így ezek egy címet, egy szerkesztő utasítást és az utasításhoz tartozó opciókat tartalmaznak, kis feladatokat fogalmaznak meg: "cím : ha az első sor jön a folyamból, parancs: töröld", "cím : a második és harmadik sor közötti részben, parancs: helyettesítsd az a karaktert b -re".

A **sed** soronként dolgozza fel a bemeneti szöveget. Két pufferel dolgozik, az egyiket mintatérnek nevezi (*pattern space*), a másikat egy másodlagos tároló puffernek (*hold space*). A következő algoritmust ismétli:

1. Beolvas egy szövegsort levágja a végéről az újsor karaktert és a sort a mintatérbe teszi.

2. Ezután lefuttatja a szkriptben levő parancsokat a mintatéren: minden egyes parancsot ellenőriz (egymás után), és csak azokat hajtja végre, amelyeknek a címe *illeszkedik* a mintatérben levő sor címére (azért használhatjuk itt az illeszkedés fogalmat, mert a címeket legtöbbször nem sorszámokkal adjuk meg, mint pl. első és második sor, hanem pl. így: azok a sorok amelyek számjeggyel kezdődnek. Ezt pedig a legkönnyebben reguláris kifejezésekkel lehet kifejezni).

3. Ha ezzel elkészül, és nem adtunk meg `-n` opciót, a mintatér tartalmát kiírja a kimenetre, a sor után kiírja a levágott újsor karaktert.

4. Ha nem használunk valamilyen speciális parancsot, ezek után törli a mintateret.

5. Veszi a következő sort és végrehajtja ugyanazt.

A **sed** egyszer olvassa végig a bemeneti szöveget, ha áthaladt egy soron, oda már nem tér vissza

A hold tér arra szolgál, hogy a mintatér törlése előtt speciális parancsokkal sorokat mozgassunk át oda bonyolultabb feldolgozások végett. Így tehát pufferelt feldolgozást is végezhetünk: ezzel nem foglalkozunk, a hold tér használata feltételezi a **sed** egyszerű működésének jó elsajátítását, ugyanakkor kis héjprogramokban nincs rá feltétlenül szükség.

A sed parancsai

A címek megadása

A cím meghatározhat egy szöveg sor címet, egy tartományt, de kifejezhet valamilyen szövegben található mintát is. A feldolgozás során ezzel válasszuk ki "mintateret", azokat a sorokat amelyeket feldolgozunk. Így a címeket az alábbiak szerint lehet megadni:

Szintaxis	Mit címez
`nincs cím a parancs előtt`	Ha nincs a parancs előtt cím, akkor azt minden soron végig kell hajtani.
`n`	Egy szám, amely a feldolgozott szöveg bizonyos sorát címzi, pl.: `22`

Szintaxis	Mit címez
	a 22.-dik sort címzi. A sorokat 1-től számozzuk.
`$`	A bemenet utolsó sorát címzi.
`n~m`	Az n.-dik sortól m lépésként címzi a sorokat, pl: `1~3` minden harmadik sort címez.

`/regex/`	Azokat a sorokat címzi, amelyekben a `regex` reguláris kifejezésre illeszthető karakterláncot talál. Reguláris kifejezés opció nélkül a sed alapszintű reguláris kifejezéseket használ, a `-r` opcióval pedig bővítetteket. Mi többnyire így fogjuk használni. A kifejezést két per jellel határoljuk, pl. a:
	
	`/^a/`
	
	cím azokra a sorokra vonatkozok amelyek kis a karakterrel kezdődnek,
	
	`/[0-9]/`
	
	pedig azokra amelyekben van számjegy.
	
	Határolójelként a per jel helyett más karaktert is lehet használni, például a % jelet, ez akkor hasznos ha a reguláris kifejezésben van / . Ilyenkor az illető karakter első előfordulását egy vissza-per jellel kell jelezni a GNU **sed**-ben, pl. az alábbi cím kiválassza azokat a sorokat amelyek per jellel kezdődnek:
	
	`\%^/%`

`/regexp/I`	Az reguláris kifejezés I módosítója kis-nagybetű nem számít típusú illesztést tesz lehetővé

`cím1,cím2`	A két cím közti tartományt címzi, a 10. és 15.-dik sor közti tartományt
	
	`10,15`
	
	a 10. és az utána következő első olyan sor közti részt amely a-val kezdődik
	
	`10,/^a/`
	
	Figyelem: amennyiben a második cím kisebb mint az első vagy nem található meg a szerkesztett fájlban, akkor a sed az első címtől a fájl végéig szerkeszt.

A címek mindig teljes sorra vonatkoznak, és amennyiben címhatárokra vonatkoznak, a határok is beleesnek a címzett tartományba, tehát a `6,8` cím 3 sort jelent.

Ha reguláris kifejezésekkel megadott tartományokkal dolgozunk, azok akár több címteret is kiválaszthatnak. Pl. az `/^a/,/^b/` tartomány az első a betűvel kezdődő sortól a következő b

betűvel kezdődő sorig válassza ki a címteret, és azon hajtja végre a neki megfelelő parancsot - ez többször előfordulhat a bemeneti sorokban.

Ha a **sed** nem kapja meg a tartományt záró sort (pl. az előbbi példában nincs b-vel kezdődő sor), akkor a bemenet végéig mindent kiválaszt attól a sortól kezdve, amelyik a-val kezdődik.

Mindez azért van így, mert a sed működése közben nincs "előretekintés", egyszer járja végig a szöveget (a végrehajtási sebesség miatt).

A reguláris kifejezések használatáról

Ebben a tananyagban minden reguláris kifejezést bővített – *extended* – kifejezéssel adunk meg, tehát csak akkor működnek helyesen a sed-el való feldolgozásban, ha a **sed**-et `-r` opcióval indítjuk.

A címek utáni `!` jel esetében a parancsok a megadott cím komplementer tartományára vonatkoznak. Tehát a "nyomtass mindent kivéve a 3.-dik sort" parancs az alábbi:

```
$ sed -n '3!p' teszt.txt
```

Gyakran használt sed parancsok

A **sed** parancsai egymás után következnek külön szövegsorokban, amennyiben a sed szkript fájlban van, vagy ; jellel elválasztva, ha a szkript egy karakterláncban van a parancssoron. Lássuk tehát a **sed** leghasználtabb parancsait: egy táblázatban adjuk meg őket, minden parancs leírása után példákon keresztül.

Parancs	Mit végez
`#`	Megjegyzés

`p`	Nyomtatás (*print*)
	
	Kinyomtatja a kimenetre mintatér tartalmát. A **sed** normál módban minden feldolgozási ciklus után automatikusan nyomtat. Ha `-n` opcióval indítjuk, akkor nem nyomtat, és ilyenkor azokat a sorokat amelyeket mégis nyomtatni akarunk ezzel a paranccsal küldjük a kimenetre. Pl. az alábbi parancs csak a harmadik sort fogja nyomtatni:
	
	`$ sed -n '3p'  teszt.txt`
	
	az alábbi pedig mindent sort, de a harmadikat kétszer:
	
	`$ sed  '3p'  teszt.txt`

`d`	Törlés (*delete*) Törli a mintatérbe került sort vagy sorokat és új ciklusba kezd. Pl.: `3d     #törli a harmadik sort`

Parancs	Mit végez
	`3!d  #töröl minden sort, kivéve a harmadikat`
`q`	Kilépés (*quit*) Kilép a sed, ha eléri azt a sort, amelyre ez a parancs vonatkozik. Ilyenkor a feldolgozás véget ér. Az alábbi parancs a `szoveg.txt` fájl első 3 sorát listázza, utána kilép: `$ sed -n 'p;3q' szoveg.txt`

`s/regexp/csere/k`	Helyettesítés, csere (substitute)
	
	A címzett sorban a regexp -re illeszkedő karakterláncot a csere láncra cseréli. Például az alábbi parancs a sorban található első ab szekvenciát xy-ra cseréli:
	
	`s/ab/xy/`
	
	vagy az alábbi az utolsó sor (címe a `$` jel) elején levő kis a karaktert b-re cseréli
	
	`$/s/^a/b/`
	
	A parancs külön metakarakterekkel rendelkezik, amelyeket a `csere` sztringben használhatunk, illetve a második `/` jel után módosító opciókat adhatunk meg.
	
	A `csere` sztringben az alábbi metakaraktereket lehet használni:
	
	`\n` a reguláris kifejezés n.-dik `()` -el kiválasztott részláncát helyettesíti. Pl: `\1 \2`
	`&` a teljes regexp által illesztett láncot helyettesíti (így a konkrét `&` karakter beírására csere karakterláncba `\&` -et kell használni)
	
	A `k` opció helyén az alábbiak használhatók:
	
	`g` a regexp minden előfordulását helyettesíti a feldolgozott sorban
	`n` egy szám: csak az n.-dik előfordulást helyettesíti
	`I` kis-nagybetű nem számít a `regexp` illesztésénél
	`w fájl` a csere után az illető sort a fájl végére írja. Induláskor a sed nullára állítja a fájl hosszát. Ez arra jó, ha csak bizonyos szerkesztett sorokat akarunk egy fájlba kiírni.
	
	Példák:
	
	`s/(ab).*/\1/` attól a ponttól kezdve, ahogy ab-t talál a sorban, az egész sort ab-re cseréli.
	
	`s/ab/xy/2` a sorban található második ab-t cseréli xy-ra.

Parancs	Mit végez
	`s/^.*$/x&x/` beilleszt a kiválasztott sor elejére és végére egy kis x karaktert: a kifejezés: `^.*$` a teljes sort kiválasztja: a csere sztring pedig egy kis x, utána a kifejezéssel kiválasztott rész (az & így a teljes sort jelenti) utána ismét egy kis x. `s/[0-9]/x/2` a szerkesztett sorban előforduló 2.-dik számjegyet kis x-el helyettesíti. `/[0-9]/-/g` a szerkesztett sorban előforduló összes számjegyet kötőjellel helyettesíti. `1,3s/^[0-9]//w o.txt` az elsőtől a 3.-dik sorig töröl egy számjegyet a sor elejéről, és a szerkesztett sort (amellett, hogy elvégzi a **sed** beállított kimenetre írását) kiírja az `o.txt` fájlba - ezt a **sed** indításkor nullázza ha létezik.

`cím {parancs1; parancs2}`	Kapcsos zárójelet használunk a **sed** parancsok csoportosítására: egy címtartományhoz több parancsot tudunk így rendelni, amelyek egymás után kerülnek meghívásra. Az alábbi sed szkript veszi a `mas` -al kezdődő sort, elvégez benne egy helyettesítést és kinyomtatja:
	
	`$ sed -n '/^mas/{ s/a/b/; p }' szoveg.txt`
	`$`

További példák:

1. Töröljük egy szövegből az üres sorokat:

```
$ sed -r '/^$/d' teszt.txt
$
```

2. Cseréljük ki a szövegben mindenhol a `get` szót `put`-ra:

```
$ sed -r 's/get/put/g' teszt.txt
$
```

   A fenti művelet a get sorozat minden előfordulását lecseréli: ha azt akarjuk, hogy valóban teljes szavakat cseréljen ki, akkor így kell használjuk (megadjuk szó két végén a szóhatárt):

```
$ sed -r 's/\bget\b/put/g' teszt.txt
```

3. Cseréljük ki a `California Hotel` szópárt `Hotel California` -ára :

```
$ echo 'Hotel California' | sed -r 's/(Hotel) (California)/\2 \1/'
California Hotel
$
```

   Vagy általánosabban, minden sorban cseréljük fel az első két szót:

```
$ echo 'elso masodik harmadik' | sed -r 's/^(\w+) (\w+)/\2 \1/'
masodik elso harmadik
$
```

4. Minden C fájlban cseréljük ki az `elso_fuggveny` azonosítót `masodik_fuggveny`-re:

```
$ sed -is -e 's/\belso_fuggveny\b/masodik_fuggveny/g' *.c
$
```

   Ebben példában használtuk `-i` (*in place*) és `-s` (*separate*) opciókat: ezek a fájlt a lemezen szerkesztik meg (ott ahol van) és úgy, hogy minden fájlt külön (separate). Egyébként a **sed** az összes megcélzott fájlt egyetlen bemeneti folyamnak tekintené.

5. Nyomtassuk ki az első olyan sort amelyik kis a betűvel kezdődik, és utána lépjünk ki:

```
$ sed -rn '/^a/{p;q}' teszt.txt
$
```

6. Helyettesítsünk minden tabulátor karaktert 4 szóközzel:

```
$ #teszteljük, hogy az echo valóban tabulátorokat ír
$ #a cat -A opcióval kiírja a kontrol karaktereket,
$ #a tabulátor Ctrl-I
$ echo -e '\t\t\t'   | cat -A
^I^I^I$
$ #átírjuk a tabulátorokat szóközzé és teszteljük,
$ #hogy valóban így van
$ echo -e '\t\t\t'   | sed 's/\t/    /g'  |cat -A
$
$ #az előző sorban már nincsenek Ctrl-I karakterek
```

7. Szóközzel lválasztott mezők kezelése, pl.: három közül hagyjuk meg csak a második mezőt:

```
$ cat a.txt
aaa bbb ccc
111 222 333
$ cat a.txt | sed -r 's/([^ ]+) ([^ ]+) ([^ ]+)/\2/'
bbb
222
$
```

8. A sed megjegyzi a legutóbb használt reguláris kifejezés az s parancs számára, és amennyiben üres reguláris kifejezést adunk meg, akkor a megjegyzettet használja:

```
$ cat > b.txt
12
34
$ # a fenti két szám van a b.txt két sorában
```

```
$ # elvégzünk 2 helyettesítést, de a másodiknál
$ # nem adjuk meg a reguláris kifejezést
$ sed -r 's/^([0-9])/x/;n;s//y/;' b.txt
x2
y4
$
```

Helyettesíti a sor első számjegyét egy x-el, utána másik sort vesz, és ugyanazt a pozíciót helyettesíti y-al.

A sed ritkábban használt parancsai

A sed rendelkezik még néhány paranccsal, amelyek segítségével összetett műveletekre is képes.

Parancs	Mit végez
`y/halmaz1/ halmaz2/`	Karaktercsere (az eredmény azonos a **tr** programéval). A két halmazban ugyanannyi karakternek kell lennie. Nem használhatóak karakterfelsorolások – -el (mint a tr programnál). `y/ax/AX/` cserél minden a és x kisbetűt nagyra

`a\`
`szoveg`
	Hozzáfűzés (*append*)
	
	Csak egy cím használható a parancs előtt. A szoveg -et a mintatér tartalma után fűzi. Több sor beszúrása esetén mindig sorvégi `\` -el kell jelezni, hogy ismét egy sor következik.
	
	`a\`
	`Elso sor\`
	`masodik sor`
	
	Az `a`, `i`, `c` parancsokkal beszúrt szövegek nem kerülnek további feldolgozásra.

`i\` `szoveg`	Beszúrás (*insert*) A mintatér elé fűzi a szöveget
`c\` `szoveg`	Változtat (change) Kicseréli a mintatér tartalmát a szöveggel.
`=`	Kiírja a címzett sor számát a kimenetre. Például az alábbi sed szkript a **wc** `-l` parancsot helyettesítheti: `$=` Az utolsó sornál kiírja a sor sorszámát, használata:

Parancs	Mit végez

	`$ cat teszt.txt | sed -n '$='`
	`3`
	`$`

`r file`	Beolvassa a `file` fájlt. Csak egy cím szerepelhet előtte. Ha `file` nem olvasható, szó nélkül dolgozik tovább. A beolvasott szöveg a címzett sor után kerül a kimenetre.
`w file`	Kiírja a mintateret egy fájlba. Az induláskor létrehozza, vagy nulla hosszra viszi a fájlt ha az létezik. Ha a `file` név több `w` vagy `s` parancs utáni `w` opcióban szerepel, ezeket ugyanabba a fájlba írja egymás után.
`n`	Következő sor (*next*) Új sort vesz a feldolgozandó szövegből, a nélkül, hogy új ciklust kezdene. Ez előtt kiírja a mintateret (amennyiben ez nincs letiltva a `-n` opcióval). Ha nincs több sor, a **sed** kilép. Az alábbi szkript a bemenet minden második sorát listázza: `$ sed -n 'n ; p' teszt.txt`

`e shell_parancs`	Az `e` parancs hatására a sed végrehajt egy parancsot a shell alatt, és a kimenetét a mintatérbe írja az éppen beolvasott sor elé. A következő szkript a bemenet második sora elé szúrja az **id** parancs kimenetét:
	
	`$ id -u`
	`500`
	`$ echo -e "a felhasználó azonosítóm\n-as"`
	`a felhasználó azonosítóm`
	`-as`
	`$ echo -e "a felhasználó azonosítóm\n-as" | \`
	`  sed '2e id -u'`
	`a felhasználó azonosítóm`
	`500`
	`-as`

A hold pufferre vonatkozó parancsok

Érdekességként megemlítjük a hold pufferre vonatkozó parancsokat is. A hold puffer szövegsorok tárolását teszi lehetővé feldolgozás közben. Az alábbi műveleteket lehet végezni:

Parancs	Mit végez
`N`	Új sort vesz a bemenetről, egy újsor karaktert ír a mintatér végére, és utána illeszti a sort. Ha ez az utolsó bementi sor, a **sed** kilép.
`P`	Kiírja a mintatér tartalmát az első újsorig.
`h`	A mintateret átírja a hold térbe.
`H`	A mintateret a hold tér végére írja egy újsor karakterrel együtt.

Parancs	Mit végez
g	Átveszi a hold tér tartalmát a mintatérbe.
G	A hold tér tartalmát a mintatér végére illeszti, elé egy újsor karaktert ír.
x	Kicseréli a hold és a mintatér tartalmát.
D	Kitöröl egy sort a mintatérből: az első újsorig. Ha marad még szöveg a mintatérben, akkor új ciklust kezd, anélkül, hogy új sort olvasna.

Íme néhány jól ismert példa a használatukra:

1. Beszúr egy üres sort minden sor után:

```
$ sed 'G' teszt.txt
$
```

2. "Megfordítja" a bemeneti fájlt, az utolsó sort elsőnek listázza:

```
$ sed -n 'G;h; $p' teszt.txt
$
```

3. Két egymás utáni sorból egye sort állít elő:

```
#!/bin/sed -nf
#a program összeragaszt két egymás utáni sort
        #mielőtt az N végrehajtódik, már van egy sor beolvasva
N       #következő sor a mintatérbe
s/\n//  #átírom az újsor karaktert a 2 sor között üresre
        #ezután a sed automatikusan kiírja a mintateret
```

4. A **tac** parancs szimulálása (a **tac** a **cat** fordított műveletét végzi, fordítva listázza a fájlt, elsőnek az utolsó sort):

```
#!/bin/sed -nf
#fordítva listáz ki egy állományt, mint a tac
#használat:
# sed -nf reverse.sed file.txt

1!G  #visszakérjük a teljes hold puffert,
     #így a beolvasott lesz az első sor
     #az első sor esetében nem kell ezt csinálni, ezért 1!
$p   #az utolsó sornál nyomtatunk, minden eddigi sort egyszerre

h    #az összes eddig bekerül a hold pufferbe
     #minden ciklus végén, fordított sorrendben
```

Ugyanennek a feladatnak egy másik megoldása az alábbi:

```
$ sed -n -r '1h;1!{x;H};${g;p}' teszt.txt
$
```

Példák a sed használatára héjprogramokban

A sed használható parancssorról, futtathat hosszabb sed szkriptet külön fájlból és használható héjprogramokban kis műveletek elvégzésére. A sed akár több fájlon is elvégez parancssoron kiadott műveleteket ha felsoroljuk azokat argumentumában. Így meghívhatjuk egy-egy önálló feladat elvégzésére szkriptekből is

Szkriptekben a **sed** parancsot egész kis műveletek kapcsán is használjuk, igen gyakran a helyettesítő `s` parancsát, ha egy sztringet valamilyen okból szerkeszteni kell. Ilyen pl. a függvényeket bemutató szekcióban az érték visszatérítő példa [107].

Gyakran használjuk szűrőként, szöveges fájlok előfeldolgozásához. Az alábbi példában adott CSV fájlból például a 3. oszlopot szeretnénk használni, ugyanakkor az első sortól meg kellene szabadulni:

```
$ cat lista.csv
Tipus  db   Ar  raktar
A   2   22.50    i
B   1   17.80    i
F   2   17.10    n
G   3   1.12    i
H   2   12.10   i
O   1   12.00    n
$
```

Az alábbi szűrés elvégzi a feladatot (a teljes sorra illesztünk reguláris kifejezést, és helyette csak a kiválasztott mezőt írjuk vissza, miután meggyőződünk, hogy az elválasztók valóban szóközök, és nem tabulátorok):

```
$ cat lista.csv | sed -r '1d;s/^[A-Z] + [0-9] +([^ ]+).*/\1/'
22.50
17.80
17.10
1.12
12.10
12.00
$
```

Megjegyzés: a fenti feladatra, különösen ha számolni is kell a kiválasztott számsorral alkalmasabb az awk (lásd Az awk fejezetet [133]).

Szkriptből történő fájl szerkesztésnél viszont inkább a **sed**-et használjuk.

Például ha egy program konfigurációs állománya gyakran változik, paramétereket kell átírni benne, érdemes annak egy sablont készíteni, és a sablont az aktuális változókra átírni egy szkripttel.

Legyen a kis példa sablonunk (`ip.conf.tpl`) az alábbi:

```
#tartalom

#IP beállítás
IP = $MY_IP

#tartalom
```

A `$MY_IP` helyett akarunk egy IP címet beszúrni a szerkesztésnél, és az `ip.conf.tpl` fájlból egy `ip.conf` nevűt létrehozni. Az alábbi kis szkript elvégzi ezt, paramétere az IP szám:

```
#!/bin/bash
#változó helyett egy IP számot ír egy konfigurációs
#állomány sablonjába
#Paraméter:
#   $1 - IP szám

#ellenőrizzük, hogy valóban IP szám

TEMPLATE=ip.conf.tpl
CONF=ip.conf

if ! echo "$1" | egrep -q '^([0-9]{1,3}\.){3}[0-9]{1,3}$'
then
    echo make_conf: Nem IP szám: "$1"
    exit 1
fi

sed -r '/^IP /s/\$MY_IP/'"$1"'/' "$TEMPLATE" > "$CONF"
```

Ellenőrizzük az IP szám formátumát (feltételezzük, hogy a legegyszerűbb alakban jön, és bármilyen számot elfogadunk), majd a **sed**-et szerkesztőként használva létrehozzuk az új fájlt a kimeneten. Látható, hogy a **sed** parancshívást több sztringből raktuk össze, mert az aposztrófok közti sed parancssor nem terjesztette volna ki a `$1` változó referenciát. A `$` metakarakter, így `\$`-t írunk ahhoz, hogy literálisan jelenjen meg a reguláris kifejezésben. Végrehajtva először hibásan majd helyesen:

```
$  bash make_conf.sh 192.168.1
make_conf: Nem IP szám: 192.168.1
$  bash make_conf.sh 192.168.1.1
$ cat ip.conf

#tartalom

#IP beállítás
IP = 192.168.1.1

#tartalom

$
```

Az következő példában (`elines_clean.sh`) kitöröljük az üres sorokat egy könyvtár szöveges állományaiból, de csak azokban, amelyekben több üres sor van mint egy adott szám. A törlési műveletet egyetlen **sed** paranccsal végre lehet hajtani, viszont a kiválasztási feltételt egyszerűbb egy szkripttel megoldani.

```
#!/bin/bash
#törli az üres sorokat azokból a szöveges állományokból
#amelyekben az üres sorok száma meghalad egy N számot
#Paraméterek:
#   $1 - célkönyvtár neve
#   $2 - sorok száma
#

#paraméter átvétel
d=${1:? az első paraméter a könyvtárnév}

#éles szkripteknél ajánlatos ellenőrizni, hogy N egész szám
#ezt itt nem tesszük meg
N=${2:? a második paraméter a sorok számának határa}

for f in $( ls "$d" )
do
    #előállítom a célfájl teljes elérési útját egy változóban
    target="$d/$f";

    #ellenőrzöm, hogy a fájl valóban szöveges
    if ! file "$target" | egrep -q 'text'
    then
        echo elines_clean: a "$target" nem szöveges vagy üres
   #ha nem szöveges, a új ciklus
       continue
    fi

    #hány üres sor van a kiválasztott fájlban
    empty=$( egrep '^$' "$target" | wc -l )

    if (( empty > N ))
    then
      sed -i.bak -r '/^$/d' "$target"
      echo elines_clean: "$target" szerkesztve
    fi
done
```

Végrehajtva a `teszt_sed` könyvtáron, amelyet a végrehajtás előtt és után is listázunk:

```
$ ls teszt_sed
1.txt  2.txt  3.txt  4.txt  5.txt
$ bash elines_clean.sh teszt_sed 3
elines_clean: teszt_sed/1.txt szerkesztve
elines_clean: teszt_sed/3.txt szerkesztve
elines_clean: a teszt_sed/4.txt nem szöveges vagy üres
$ ls teszt_sed
1.txt  1.txt.bak  2.txt  3.txt  3.txt.bak  4.txt  5.txt
```

```
$
```

Megjegyzések

1. Mivel a sed stílusú szerkesztés megváltoztatja a fájlt, és csak szöveges fájlokon van értelme, ezt ellenőrizzük.

   A **file** parancs kiírja egy fájl típusát:

   ```
   $ file teszt_sed/1.txt
   teszt_sed/1.txt: ASCII text
   $
   ```

   ezt a kimenetet ellenőrizzük az **egrep**-el.

2. A **sed -i** opcióval végzi a helyben való szerkesztést (tehát ilyenkor nem a kimenetre írja a feldolgozott fájlt, hanem meg is szerkeszti). Pl.:

   ```
   $ sed -i -r '/^$/d' teszt_sed/1.txt
   $
   ```

   A művelet veszélyes, mert ilyenkor csak a megszerkesztett fájl marad az `1.txt` név alatt, így lehetőség van arra, hogy háttér másolatot is készítsünk. A másolat nevének kiterjesztését meg lehet adni közvetlenül az `-i` opció után, pl. ha azt karjuk, hogy egy `.bak` nevű kiterjesztésű másolat keletkezzen, a feladatban is használt megoldás:

   ```
   $ sed -i.bak -r '/^$/d' teszt_sed/1.txt
   $
   ```

   ## A sed -i és -s opciói

   Fontos: ha egyszerre több fájlt adunk meg a parancssoron, a kimenet szempontjából a **sed** egyetlen kimeneti folyamot készít belőlük. Így `-i` *inline* szerkesztés esetén meg kell adni, hogy egyenként szerkessze a fájlokat, ezt a `-s` opcióval tesszük, és így meg lehet adni neki egy több fájlt kiválasztó mintát, mint a `*.txt`. Külön-külön fogja őket szerkeszteni:

   ```
   $ sed -s -i.bak -r '/^$/d' teszt_sed/*.txt
   $
   ```

Bővített reguláris kifejezéseket használunk, ezért minden esetben alkalmazzuk a `-r` opciót.

10. fejezet - Az awk

Bevezető

Az Awk egy programozási nyelv, szerzői "*Awk, a pattern scanning and processing language*": minta elemző és feldolgozó nyelv nevet adták neki. A nyelv tulajdonképpen bármilyen felépítésű szövegből minták alapján mezőket tud kiemelni, és azokkal számítási vagy szerkesztési műveleteket tud végezni. Így akár a Unix programok kimenetét (pl. az **ls** `-l` parancs kimenetét) akár adatokat tartalmazó, szövegmezőkre bontható fájlok (pl. a CSV típusú fájlok) soraiból ki lehet vele hámozni egy feladat megoldásához szükséges mezőket. A nyelv programozási szerkezetei a C nyelv utasításaira épülnek. Nevét a szerzők nevének kezdőbetűiből állították össze: Aho – Weinberger – Kernighan. Az Awk rendkívüli módon megkönnyíti a szöveges fájlok feldolgozását. A mérnöki munkában nyers adatok előfeldolgozását végezhetjük vele könnyen.

Mivel szkript nyelv, kis mindennapi feladatok megoldását végezhetjük el vele gyorsan. A nyelv szerzői nem ajánlják a nyelvet nagy és bonyolult feladatok megoldásához: ennek ellenére, elég sok típusú feladatot lehet vele megoldani, különösen azokat, amelyek rendszerprogramozáshoz közeliek, és valós számokkal is kell számolni.

Ez a fejezet az Awk fontosabb elemeit mutatja be. Kimerítő tanulmányozásához a felhasználói kézikönyv is szükséges, amelynek van egy magyar fordítása (Iványi2005 [184]). Ebben a rövid bemutatóban a hangsúly a nyelv elveinek megértésén és héjprogramokban való használatán van.

Feltételezzük, hogy az olvasó ismeri a C programozási nyelvet - az Awk nyelv több szintaktikai elemet a C-hez hasonlóan használ, így kis feladatokhoz könnyen elsajátítható néhány egyszerű használati mód azok számára akik programoztak már C-ben. Ha ez nincs így, ajánljuk az előző bekezdésben említett Iványi Péter fordításnak tanulmányozását (vagy az aktuális angol nyelvű kézikönyvet).

Az Awk-nak több változata van, az alábbiak a GNU Awk-ra érvényesek Linux alatt.

Az awk parancs használata

A nyelvet értelmező **awk** parancs indítása hasonló a **sed**-éhez, a program végigolvassa a bemenetét (a standard input jelenti az implicit bemenetet), feldolgozza azt és a kimenetre ír. A `-f` kapcsoló ugyanúgy a szkript fájlra hivatkozik, de a szkript megjelenhet az első paraméterben is idézőjelek között.

Néhány fontos, általunk is használt opciója:

`-F c` *Field separator*: megállapítja a mezőelválasztó karaktert. Ha nem adjuk meg, akkor az implicit elválasztó a szóköz és a TAB, illetve ezek ismétlődése (tulajdonképpen egy reguláris kifejezés minta: `[ \t]+`.
`-f fájl` a szkript fájl megadása.
`-v var=val` a var változónak val értéket ad indulás előtt.

A leggyakrabban használt parancssori indítások az alábbiak:

`awk` *`program-szöveg`* [*`fájl(ok)`* ...]

`awk -f` *`program-fájl`* [*`fájl(ok)`*...]

Az első esetben az Awk szkript szövege a parancssoron van, általában aposztrófok között,pl.:

```
$ awk '{print}' teszt.txt
```

A `{print}` szkript egyszerűen kinyomtatja a bemenetre érkező sorokat.

A második esetben a szkript egy fájlban van:

```
$ awk -f doit.awk teszt.txt
```

A bementi fájlt megadhatjuk a standard bemenetre is:

```
$ cat teszt.txt | awk '{print}'
```

Az Awk programokat

```
#!/bin/awk -f
```

sorral kezdődően külön fájlba írjuk, melynek az `.awk` kiterjesztést adjuk, hogy megkülönböztessük őket más szkriptektől (az elérési utat ellenőrizzük a **which awk** paranccsal saját rendszerünkön).

Megjegyzés: a `.awk` kiterjesztés nem kötelező, a fájl bármilyen nevű lehet, helyes **awk** parancsokkal.

Működési elv

Az Awk a következő feldolgozási folyamatot alkalmazza a bemeneti sorokra: úgy tekinti a sorokat, mint egy elválasztó karakterrel (vagy karakter szekvenciával) határolt, *mezőkből* álló sor. Egy sornak a neve *rekord* (ez a feldolgozási egység). Pl. az alábbi (zeneszámok szövegeit tartalmazó katalógus egy sora) esetében:

112 LinkinPark Numb numb.txt 35

sornak 5 mezője van, az elválasztó egy vagy több szóköz. A sor feldolgozása közben a nyelv a mezőkre a `$1`, `$2`, ... változókkal hivatkozik. Pl. a `$3` értéke `Numb`. A teljes sorra a `$0` változóval lehet hivatkozni.

A mezőkre az alábbi szerkezettel is lehet hivatkozni:

```
k=3
print $k
```

ahol k egy változó (lásd alább hogyan definiáljuk a változókat). Az Awk sor a harmadik mező értékét fogja kinyomtatni .

A sorok számát az Awk az `NR` (*Number of Rows* - az összes eddig beolvasott sor) és `FNR` (az aktuális bemeneti fájl, ha több van) változókban követi.

A mezők száma az `FN` változóban van, amely az utolsó mező címzésére is használható: `$FN`. Az alábbi példa:

```
print $FN
```

az utolsó mezőt nyomtatja, függetlenül attól, hogy hány van. A mezők címzésére kifejezések is használhatóak:

```
print $(2+2)
```

a negyedik mezőt fogja nyomtatni.

Használat közben az Awk által előállított mezőket tartalmazó változóknak meg lehet változtatni az értékét:

```
$echo 'a b' | awk '{$1="b"; print $0}'
b b
```

Az Awk programozási nyelv

Az Awk programsorok szabályokból állanak: egy szabály egy mintából (*pattern*) és a hozzá tartozó tevékenységből (*action*) vagy utasításokból (*statement*) áll, amely `{}` közé zárt blokkban van:

```
minta  {utasítások}
```

A `{}` blokkban utasítások követik egymást. A nyelvben függvények vagy nyelvi szerkezetek hívhatóak különböző műveletek elvégzésére. Függvényei közül a legegyszerűbb a print : kinyomtatja a `$0` változó tartalmát.

```
{print}
```

Ha argumentumokkal használjuk, a print függvény a felsorolt változókat kiírja a kimenetre egy kimeneti elválasztóval (ez implicit egy szóköz):

```
$ echo '112 LinkinPark Numb numb.txt 35' | awk '{print $1, $4}'
112 numb.txt
```

Normális menetben az Awk nyelv értelmezője egy-egy sort olvas a `$0` változóba, feldolgozza azt: azaz alkalmazza reá szabályait ha a szabályhoz rendelt minta kiválassza a sort. Ezt a műveletet ismétli amíg a bemenetről elfogynak a sorok.

Mintái bővített reguláris kifejezéseket használnak, pl. a: `/^a/` azokat a sorokat fogja kiválasztani amelyek `a` -val kezdődnek.

Az Awk szabályban a mintának vagy tevékenységnek meg kell jelennie. Ha a minta elmarad, a tevékenységet minden bemeneti sorra alkalmazza az Awk, ha a tevékenység marad el, akkor az implicit tevékenység a `print $0`, azaz a bemeneti sor nyomtatása. Így:

```
$ echo piros | awk '{print $0}'
piros
```

kinyomtatja az egyetlen bemeneti sort, mert a minta illeszkedik a sorra,

```
$ echo piros | awk '/^p/'
piros
```

ugyanazt teszi, mert elhagytuk a tevékenységet de használtunk mintát. Ugyanakkor:

```
$ echo piros | awk '{print $0}'
piros
```

is kinyomtatja a bemeneti sort, mert nincs minta, tehát minden sorra illeszkedik.

Az Awk minta

A minta felépítése bonyolultabb mint a **sed** által használté. Az alábbi táblázat tartalmazza a lehetséges eseteket:

10.1. táblázat - Az Awk mintái

Minták	Kiválasztás
`BEGIN`	A `BEGIN` és `END` minták speciálisak: az utánuk következő utasításokat a sorok feldolgozása előtt (`BEGIN`) illetve minden sor feldolgozása után (`END`) hajtja végre az Awk, egyetlen egyszer. Ezek a program előkészítő utasításai (pl. változók kezdeti értékének megadása) illetve befejező utasításai (pl. végső eredmény nyomtatása). A program futtatható bemenet nélkül is, ha csak a BEGIN mintát használjuk. Ilyenkor végrehajtja utasításait és utána kilép.
`END`	
`/regex/`	Reguláris kifejezés: a Awk bővített (*extended*) kifejezéseket használ, ugyanazokat mint az egrep. Nincs tehát szükség külön opcióra, hogy ezt használjuk.
`minta && minta`	2 minta és, vagy illetve egy minta tagadásának logikai relációjából képzett feltétel. Az `&&` esetében pl. mindkét mintának illeszkednie

Minták	Kiválasztás
`minta || minta`	kell a feldolgozandó sorra, az `||` esetében csak az egyiknek. A `!` -al jelölt minták esetében a blokk csak akkor hajtódik végre, ha a minta nem válassza ki a sort.
`! minta`	
`minta1, minta2`	Sorok tartományának címzése két mintával, az első minta illeszkedésétől a másodikig.
`minta ? minta1 : minta2`	A C nyelv `?:` operátorához hasonló feltételes teszt. Ha a `minta` talál, akkor a `minta1` teszt lesz döntő, ha nem akkor a `minta2`.

`Awk relációs kifejezés`	Az Awk változóira alkalmazott C-szerű relációs kifejezés, pl. `$1 > 2` tehát ha az első mező numerikusan, számmá alakítva nagyobb mint 2, akkor arra a sorra végrehajtódik az utasítássorozat. A kifejezésben a C nyelv relációs kifejezéseit lehet használni (>, `&`, `&=`, >=, ==, `!=`).
	
	A C nyelv operátorain kívül az Awk nyelvnek van két operátora, a `~` és a `~ !` (*match* és *no match*). Használatuk így történik:
	
	`$1 ~ /^a/`
	
	jelentése: ha az első mező kis a betűvel kezdődik, azaz illeszkedik a megadott reguláris kifejezésre.
	
	A kifejezés akkor választja ki a tevékenységet, ha az értéke 0-tól különbözik amennyiben valós számra lehet konvertálni, vagy nem az üres sztring amennyiben sztringre.
	
	Tehát a kifejezéseket sztringekre is lehet alkalmazni: ilyenkor az Awk karakterenként hasonlítja össze a sztringeket. ha két sztring megegyező karakterei ugyanazok, akkor a hosszabbik tekintendő nagyobbnak. Pl. "10" kisebb mint "9" , mert első karaktereik összehasonlításakor az "1" kisebb mint "9". De 10 nagyobb mint 9: az Awk az adattípustól függően végzi el az összehasonlítást.
	
	Így az alábbi példákban az első hívás nem listázza a bemeneti sort, a második viszont igen:
	
	`$ echo "10 szöveg" | awk ' $1 < 9  {print $2}'`
	`$ echo "10 szöveg" | awk ' $1 < "9" {print $2}'`
	`szöveg`
	`$`

Az utasítások

Az Awk nyelv elemei `{ }` zárójelek közé kerülnek és szabályokat határoznak meg. A C nyelvhez hasonlóan, változó értékadást, kifejezéseket, vezérlő szerkezeteket tartalmaznak. Az operátorok, utasítások a C-hez hasonlóak (a C nyelvet vették mintának a tervezésüknél). Pl.:

```
{a=2; print a}
```

Az utasítások közt az elválasztó az újsor karakter illetve, amennyiben egy sorba több utasítás kerül(pl. a parancssoron) akkor a `;` .

Az Awk program 3 részt tartalmazhat, egy bevezető (BEGIN), egy szövegsoronként végrehajtható és egy záró (END) programrészt, tehát általánosan így néz ki:

```
BEGIN {bevezető utasítások}
minta { a kiválasztott sorra végrehajtott utasítások}
...
END   {záró utasítások }
```

A minta `{utasítasok}` szerkezet egymás után többször is előfordulhat, és ez a rész ismétlődik minden egyes feldolgozott sorra.

```
$ echo '1 2'| awk 'BEGIN{a=1} {a=a+$1+$2} END{print a}'
4
$
```

A `#` jel magyarázat beillesztését jelenti a sor végéig, akár a shell esetében.

A változók

Az Awk változóneveket ugyanúgy jelöljük mint a C nyelv változóit. A deklaráció pillanatában jönnek létre és típusukat nem kell explicit deklarálni (tehát nem szükséges őket még csak a `BEGIN` részben sem deklarálni), és a hozzájuk rendelt értéktől vagy kontextustól függően lesznek valósak vagy karakterláncok. A szkript nyelvekre jellemzően ezek dinamikus változók.

A nyelv két adat típussal rendelkezik: dupla pontosságú valós illetve sztring.

```
s="piros"
```

egy sztringet hoz létre,

```
x=2.2
```

pedig egy valós változót.

```
y=$10
```

Az `y` a `$10` mezőtől függően lesz valós vagy karakterlánc.

Ha nem inicializált változóra hivatkozunk, sztring esetében az üres sztringet, valós szám esetében a 0-át adja vissza a nyelv, hibát nem jelez. Pl.:

```
$awk ' BEGIN {  print s+2; } '
2
```

Ilyenkor a típus a kontextus szerint alakul: a fenti példában az `s` változó egy valós kifejezésben (összeadás) vesz részt, így a nyelv 0-ra inicializálja. Egyszerű változókon kívül tömbök is használhatók, ezek asszociatív tömbök (lásd az **awk** kézikönyvének megfelelő fejezetében).

Belső változók

A program tartalmaz belső változókat, ezek mindig létrejönnek, nem kell őket deklarálni. Közülük az alábbiak fontosabbak, ezeket gyakran használjuk.

10.2. táblázat - Az Awk belső változói

Változó	Angol neve	Mit tartalmaz	Implicit érték
`ARGC`	*Command line argument count*	Hány argumentum van az Awk parancssorán.	
`FILENAME`	*Filename*	A bemeneti fájl neve.	
`FS`	*Input Field Separator*	A bementi mezőelválasztó karakter vagy karakterek. Az FS reguláris kifejezés is lehet, tulajdonképpen az implicit érték is az, pontosan: `[\t ]+` , tehát szóköz vagy TAB legalább egyszer. Ha az FS értéke az üres sztring (FS="") akkor a bemenet minden egyes karaktere külön mezőnek számít.	`[\t ]+`
`RS`	*Record separator*	Ez a rekord (azaz bemeneti sor) elválasztó. Ez is reguláris kifejezés, megváltoztatható.	`\n` egy újsor karakter
`NF`	*Fields in the current input record*	Hány mező van a bemeneti rekordban.	
`NR`	*Number of records seen so far*	Hányadik sornál tart a feldolgozás.	
`OFMT`	*The output format for numbers*	Hogyan formatálja a valós számokat	`"%.6g"`

Változó	Angol neve	Mit tartalmaz	Implicit érték
		ha a kimenetre írjuk azokat. Látható az implicit értékből, hogy a formátumokat a C nyelv `printf` függvényének formázó sztringjeivel állítja elő.	
`OFS`	*Output field separator*	A kimenti mezőelválasztó karakter.	szóköz
`ORS`	*Output record separator*	A kimenti rekord (azaz sor) elválasztó karakter. Implicit újsor, akár más is lehet, amennyiben pedig az üres sztring, akkor a kimeneten nem lesznek szétválasztva a sorok.	`\n` újsor
`IGNORECASE`	*Ignore case*	Ha nem zéró, akkor a reguláris kifejezéseknél kis nagybetű nem számít.	0

Az `ARGV` illetve `ENVIRON` változók is hasznosak lehetnek akár kis programokban is, ezek a parancssor argumentumait illetve a környezeti változókat tartalmazzák. Ezek a változók Awk tömbök. Használatukat megtaláljuk az **awk** kézikönyvében (lásd Ivanyi2005 [184], Tömbök az awk-ban című fejezet).

Konstansok

A sztring konstansok kettős idézőjellel körülvett sztringek. A sztringeken belül használhatóak a C nyelv vissza-per szekvenciával megadott speciális karakterei, például az újsor (`\n`) vagy tabulátor (`\t`) . A `\c` literálisan a `c` karaktert jelenti.

```
$ awk 'BEGIN {s="abc"; t="def\nitt újsor jön"} END{print s,t}'
abc def
itt újsor jön
```

A tízes, nyolcas és tizenhatos számrendszerben megadott konstansokat ugyanúgy kezeli mint a C nyelv. Pl:

```
awk 'BEGIN {print 10.2,011,0x22}'
```

```
10.2 9 34
```

Operátorok

Az Awk a C nyelv operátorait használja, ezek változóira és konstansaira alkalmazhatók.

Az operátorok precedenciája és asszociativitása ugyanaz mint a C nyelvnél. Különbségek:

A sztringek összefűzésének operátora a szóköz karakter, tehát:

```
{print "a" "b"}
```

`ab`-t fog kiírni, két sztringet összefűzni így lehet:

```
$ awk 'BEGIN {  x="a" "b"; print x  } '
ab
```

A nyelv nem rendelkezik a vessző operátorral, tehát nem használhatunk ilyen szerkezeteket mint: `a=2, b=5` .

Rendelkezik viszont egy sajátos operátorral, amely nem található meg a C nyelvben, a reguláris kifejezés tesztelése (illesztés, *match*) operátorral. Ezt a ~ jellel adjuk meg, az ellenkező műveletet, azaz a *nem illeszkedik* tesztet pedig a !~ jelekkel. Az első operandus egy változó, a második pedig egy reguláris kifejezés. Pl.:

```
if ( s ~ /^a/ ) print "Az s változó tartalma kis a betűvel kezdődik"
```

Példák az operátorok használatára:

```
$ #inkrementálás
$ echo '' | awk '{i=0; i++; print i}'
1
$
$ #összeadás
$echo '1.1 2.2' | awk '{ s=$1+$2; print s  }'
3.3

$ #reguláris kifejezés illesztés
$ echo '112' | awk '{if ($1 ~ /^[0-9]+$/)\
  {print "az első mező számokból áll"}}'
az első mező számokból áll
$
$ echo 'abc' | awk '{if ($1 !~ /^[0-9]+$/) \
  {print "az első mező nem áll számokból"}}'
az első mező nem áll számokból
$
$ #pelda sztring összefűzésére
$ echo '1 2 3 abc def ' | awk '{print $4 $5}'
abcdef
```

```
$
```

Vezérlő kifejezések

A programok döntés és ciklus szerkezeteinek megoldására un. *vezérlő kifejezéseket* használ a nyelv. A struktúrákat a C nyelvből kölcsönzi az Awk, az alábbiak használhatóak:

if (feltétel) utasítás [else utasítás]

while (feltétel) utasítás

do utasítás while (feltétel)

for (expr1; expr2; expr3) utasítás

break

continue

exit [kifejezés]

{ utasítások }

Pl. az if használata:

```
$ echo '2.2'| awk '{ if ($1==1) {print "igaz"} else {print "hamis"}}'
hamis
```

Az Awk nyelv vezérlési utasításnak tekinti a `next` utasítást (tulajdonképpen függvényt) is.

A `next` végrehajtásakor az Awk abbahagyja a kurrens rekord feldolgozását, újat vesz a bemenetről és újraindítja vele az Awk programot. Pl. ha arra számítunk, hogy a bemenet minden sorában 4 mező van, és nem szeretnénk hibás feldolgozást végezni, akkor olyan sorok érkezésekor amelyekben nincs 4 mező könnyen kikerülhetjük ezeket:

```
#!/bin/awk -f
{
    if (NF != 4) {
        print "Rossz mezőszám, sorszám: " NR " : " $0
        next
    }
}
#... feldolgozások itt kezdődnek
```

Az Awk függvények

Az Awk nyelvben használható függvényeknek 2 kategóriája van, a beépített és a felhasználó saját függvényei. A beépített függvények fontosabb csoportjai a numerikus, a sztring és a be/kimenetet vezérlő függvények.

A fontosabb numerikus függvények

Mindazok a függvények amelyek egy kis tudományos kézi számológépben megtalálhatóak, megtalálhatóak az Awk-ban is. A függvényeket a C nyelvhez hasonló szintaxissal lehet meghívni. Ezek ismertek a C nyelv standard könyvtárából, ugyanúgy kell meghívni őket mint a C nyelvben. Az `x` a zárójelben egy valós kifejezés is lehet. Ezek az alábbiak:

`atan2(y, x)` - az y/x arkusz tangensét számolja radiánban, a paraméterek előjele meghatározza, hogy az eredmény melyik körnegyedbe esik
`sin(x),cos(expr)` - sinus és cosinus, bemenet radiánban
`exp(x)` - visszaadja az x természetes alapú hatványát
`int(x)` - egész részt ad vissza
`log(x)` - visszaadja az x természetes alapú logaritmusát, ha x pozitív; egyébként hibát jelez
`sqrt(x)` - x gyökét adja vissza
`rand()` - egyenletes eloszlású véletlen szám generálása 0 és 1 között
`srand ( [ szám ] )` - véletlen szám generálásának kezdőpontját állítja be - más véletlen számot kapunk, ha ezzel adunk kezdőértéket a generátornak.

Például a szinusz függvényt így hívjuk meg:

```
$ echo '' | awk '{print sin(1)}'
0.841471
```

Fontosabb sztring függvények

Az alábbi egyszerű kis függvények állnak rendelkezésre:

10.3. táblázat - Az Awk sztring függvényei

Függvény	Mit végez el
`index(hol, mit)`	Visszatéríti a `mit` sztring pozícióját az `hol` sztringben, 0-t ha nincs benne. A sztring indexeket 1-től számolja a nyelv.
`length (s)`	A sztring hosszát téríti vissza.
`substr(s, i [, n])`	Szubsztringet térít vissza az i-edik pozíciótól kezdve, legtöbb n karaktert.
`tolower(str)`	Kisbetűssé konvertál.
`toupper(str)`	Nagybetűssé konvertál.
`sprintf (format, kifejezés-lista)`	A C sprintf -jének megfelelő formázó függvény.
`sub(regex, mivel, [miben])`	Az `miben` sztringben behelyettesíti a `regex`-re illeszkedő részt a mivel sztringgel. Ha a miben paraméter hiányzik, akkor a `$0`-ban helyettesít. A `sub` egyszer helyettesít, a gsub minden találatot átír. Ha a miben paramétert nem adom meg, akkor a `$0` -n dolgozik. Sikertelen átírás után 0-val térnek vissza, sikeres után a `sub` 1-el, a `gsub` a cserék számával. Itt is használható az & karakter a találatra való visszautalásként, akár a **sed** s parancsában.
`gsub (regex, mivel, [miben])`	

Függvény	Mit végez el
`gensub(regex, mivel,hogyan, [miben])`	Ez az általános helyettesítő függvény. Ugyanúgy működik mint a `sub`, de a hogyan mezőben meg lehet adni ugyanazokat a kapcsolókat mint a **sed** helyettesítésénél (n: hányadik találatot illetve g). Ugyanakkor a mivel sztringen használhatóak a `\1`, `\2` stb. visszautalások. Ez a függvény a Gawk kiterjesztése.
`match(s,regexp)`	Illeszti a `regexp`-et a sztringre: visszatérít egy számot, ami 0 ha nincs illesztés illetve egy pozitív egész szám ha van: ez pontosan az illesztés indexe a sztringben.

Például:

```
$ echo 'első második' | awk '{s=substr($2,5,3); \
s1=sprintf("%s%s", $1,s); s2=substr($2,0,4); print s1,s2}'
elsődik máso
```

Saját függvények

Az Awk függvények, mint más nyelvben, számításokat csoportosítanak, és ezek név szerinti meghívását teszik lehetővé. Szintaktikailag rendelkeznek néhány furcsasággal, amire oda kell figyelni.

Saját függvényeket az alábbi szintaxissal hozhatunk létre:

```
function név (arg1, arg2, . . .)
{
    utasítások;
}
```

A létrehozás helye az Awk szabályokon kívül kell történjen a programban (programszervezési szempontból mindegy, hogy hol vannak a szabályokon kívül: nem szükséges őket használatuk előtt definiálni). Ajánlott a program végére tenni őket.

Fontos: saját függvény hívásakor ne írjunk szóközt a függvény neve és a zárójel közé, a:

```
fuggvenynev (
```

szerkezetben a szóközt a kontextustól függően a nyelv értékelheti úgy is, mint egy sztring összefűzés operátort, ezért jó megszokni, hogy a név és zárójel közé ne tegyünk szóközt.

Az argumentumok lokális változók lesznek a függvény testében. Ezzel ellentétben, ha a függvény belsejében változókat hozunk létre, azok globálisan viselkednek, tehát alkalmasak a főprogrammal való kommunikációra.

Azért, hogy ezt (a függvényben létrehozott változók láthatósága a főprogramból) elkerüljük, a függvényben használt belső változókat is felsorolhatjuk az argumentumlistában, de értéküket nem adjuk meg híváskor, így üres változókká inicializálódnak. Pl.:

```
function hatvany(i,j,hat) {
    hat=i**j
    return hat
}
```

Híváskor így használjuk:

```
x=hatvany(2,3)
```

és akkor a `hat` változó nem lesz látható a főprogramból.

Tehát az Awk programból az alábbiak szerint használjuk a függvényeket:

`név(kifejezés1, kifejezés2, … )`

vagy:

`változó=név(kifejezés1, kifejezés2, … )`

mennyiben a függvény `return kifejezés ;` utasítással lép ki, és egy értéket rendel hozzá.

A függvényeket lehet rekurzív módon használni.

Az alábbi program számok négyzetét listázza. Itt, bár nem használtuk fel, a `negy` változó is létrejön és látható lesz a főprogramban mivel nevét nem adtuk meg a függvény argumentumai között.

```
#!/usr/bin/awk -f

BEGIN {
    for (i=0; i < 10; i++) {
        printf "%4d\t%4d\n", i,negyzet(i)
    }
    print "\na főprogramból látható negy:" negy
}

function negyzet(i) {
    negy=i*i
    return negy
}
```

A program futtatásakor ezt látjuk:

```
$ awk -f negyzet.awk
   0       0
   1       1
   2       4
   3       9
   4      16
```

```
   5      25
   6      36
   7      49
   8      64
   9      81

a főprogramból látható negy:81
$
```

Mivel a szkript csak a `BEGIN` mintát használja, lehet bemenet nélkül is futtatni. A négyzetet a függvény példa kedvéért számoltuk, a nyelvnek van hatványozó operátora, a `**` .

Az awk használata héjprogramokban

Az **awk** parancsot is használhatjuk nagyon egyszerű, kis műveletekre vagy önálló, nagyobb programok készítésére, amelyeket héjprogramokból futtatunk és részfeladatokat oldunk meg velük.

Az **awk** használatára lemezpartíciók lefoglaltságának követését adjuk példaként.

A Unix rendszerek felcsatolt lemezpartícióinak méretét és elfoglaltságát a legegyszerűbben a **df** (*disk free*) paranccsal tudjuk kiírni, a méretek implicit 1 kilobyte-os egységekben vannak megadva:

```
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             25798712   3428312  21059900  14% /
udev                    896720       104    896616   1% /dev
/dev/sda1               108883     29450     73811  29% /boot
/dev/sda5            185778200  21664256 154676964  13% /home
/dev/sda6             24525480   8870072  14409572  39% /usr
$
```

A lista az eszköz fájl nevét, méretét valamint az elfoglalt és szabad 1 kilobyte-os blokkok számát írja ki. Az utolsó oszlop a fájlrendszerbe való csatolás helyét adja meg. A `-h` opciójával szemmel könnyebben olvasható listát készít:

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda2              25G  3.3G   21G  14% /
udev                  876M  104K  876M   1% /dev
/dev/sda1             107M   29M   73M  29% /boot
/dev/sda5             178G   21G  148G  13% /home
/dev/sda6              24G  8.5G   14G  39% /usr
```

A lista mezői szóközökkel vannak elválasztva, ezek száma nem ugyanannyi minden mező közt, így az **awk** a legalkalmasabb arra, hogy az ehhez hasonló kimenetekből információt vágjon ki. Pl. ha szükségünk van a `/home` könyvtár alatti partíció méretére, akkor az az alábbi paranccsal kaphatjuk meg:

```
$ df | awk '/home/{print $2}'
185778200
$
```

, vagy a lefoglaltságra százalékban:

```
$ df | awk '/home/{print substr($5,1,length($5)-1)}'
13
$
```

Ezeket a számokat változóba lehet írni, és felhasználni számításokra:

```
$ used=$( df | awk '/home/{print substr($5,1,length($5)-1)}' )
$ echo A /home partíció $used százaléka foglalt
A /home partíció 13 százaléka foglalt
$
```

Az alábbi parancs összeadja a rendszeren létező partíciók méretét (1 kilobyte-os blokkokban):

```
$ df | awk 'NR!=1{s=s+$2}END {print s}'
237107995
$
```

A **df** opcióit, lehetőségeit lásd a kézikönyvében.

A következő szkript a partíciók lefoglaltságát ellenőrzi. A fájlrendszer partíciói az alábbi állapotban vannak:

```
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             25798712   3428400  21059812  15% /
udev                    896720       104    896616   1% /dev
/dev/sda1               108883     29450     73811  29% /boot
/dev/sda5            185778200  21664260 154676960  13% /home
/dev/sda6             24525480   8870072  14409572  39% /usr
$
```

A szkriptnek (`dmonitor.sh`) paraméterként megadjuk a partíciók csatolási pontját, és azt a százalékban maximális lefedettséget amit már problémának tekintünk. Ezt egy egyszerű felsorolásban adjuk meg a parancssoron. Ha a partíciók nincsenek megtelve, a szkript nem ír ki semmit, ha valamelyik meghaladja az kijelölt határt, a szkript üzenetet ír ki. Tehát:

```
$ bash dmonitor.sh /home 80
$ bash dmonitor.sh /home 80 / 60
$
```

nem kapunk üzenetet, mert sem a `/home`, sem a `/` nem haladta meg a 80% illetve 60% lefoglaltságot (a `/home` esetében a 80%-ot, `/` esetében a 60%-ot ellenőrizzük). Ha kis méreteket adunk meg az ellenőrzött százalékoknak (teszt miatt):

```
$ bash dmonitor.sh /home 12 / 60
Baj van a /home partícióval: lefoglaltság: 13 %

$ bash dmonitor.sh /home 12 / 12
Baj van a /home partícióval: lefoglaltság: 13 %
Baj van a / partícióval: lefoglaltság: 15 %

$
```

A szkript az alábbi:

```
#!/bin/bash
Az argumentumában magadott könyvtárakra csatolt
#lemez partíciók lefoglaltságát figyeli
#ha az lefoglaltság meghaladja a tesztelendő maximális
#lefoglaltságot, egy üzenetet ír ki, egyébként nem ír ki semmit
#Paraméterek:
#   $1 - csatolási pont neve, pl.: /home
#   $2 - maximális lefoglaltság, pl.: 90 (egész szám, jelentése 90%)
#
#   utána $3,$4 hasonló párt jelent, $5, $6 szintén, stb.
#

#ideiglenes fájl a df kimenetének felfogására
TEMPFILE="/tmp/df.tmp"

#a df kimenetét az ideiglenes fájlba írjuk
#hogy ne legyen szükséges többször meghívni
#előtte levágjuk az első sorát amire nincs szükségünk
df | tail -n +2 > $TEMPFILE

msg=""              #ebbe a változóba gyűjtjük a hibaüzeneteket

#kell legalabb 2 paraméter
if (( $# < 2 )) ;then
    echo dmonitor: hibás indítás, kevés paraméter
    exit 1
fi
#minden egyes könyvtár és százalék párosra elvégezzük az ellenőrzést
#amíg van még parancssori paraméter
while (( $# != 0 ))
do
    part="$1"       #első paraméter a partíció
    max="$2"        #második paraméter a maximális lefoglaltság
    shift 2         #eltoljuk két pozícióval a parancssort
                    #a következő ciklust készítjük elő ezzel

    #végzünk néhány ellenőrzést
    if [ -z "$part" ] ; then
        echo dmonitor: hibás bemenet, partíció helyett üres sztring!
        exit 1
    fi
    #a méret egy vagy kétjegyű egész szám?
```

```
    if ! [[ $max =~ ^[0-9]{1,2}$ ]] ; then
        echo dmonitor: a max=$max méret nem kétjegyű egész szám!
        exit 1
    fi

    #kiszűrjük a partíciónak megfelelő sort
    used=$( cat $TEMPFILE | \
        awk -v part="$part" '{if ($6==part) \
        print substr($5,1,length($5)-1)}' )

    if (( used > max )) ;then
        msg="$msg""Baj van a $part partícióval: $used % \n"
    fi

done

#ha van üzenet kiírjuk a kimenetre
#az echo -e kapcsolója a \n szekvenciát újsorrá alakítja
if [ -n "$msg" ] ; then
    echo -e "$msg"
fi
```

Megjegyzések:

1. A **df** kimenetét ideiglenes állományba tároljuk, hogy ne hajtsuk többször végre. A `/tmp` könyvtárat használjuk erre, és egy egyszerű fájlnevet, feltételezzük, hogy a szkript egy példányban fut. Ha fennállna annak a lehetősége, hogy egyszerre több példány fut, a fájl nevébe be kellene illeszteni egy egyedi számot. Ez lehet a folyamatazonosító (részletesen lásd a következő fejezetben [152]), amelynek értéke a `$` változóban van:

   ```
   TEMPFILE="/tmp/df_$$.tmp"
   ```

   vagy a **mktemp** [165] paranccsal létrehozott fájlnév.

2. Az egész szám ellenőrzését a `[[ ]]` [115] szerkezettel végezzük. Ebben a reguláris kifejezés csak bizonyos új héjakban működik. Ha nem biztos, hogy ilyen áll rendelkezésünkre, vagy hordozhatóbbá akarjuk tenni ezt, akkor helyette az **egrep**-et alkalmazhatjuk:

   ```
       #a méret kétjegyű egész szám?
       if  !  egrep -q '^[0-9]{1,2}$' ; then
           echo dmonitor: a max=$max méret nem kétjegyű egész szám!
           exit 1
       fi
   ```

3. A szűrőként használt nevekben előfordul a `/` karakter is. A nevet külső változóként adjuk át az **awk**-nak a `-v` opcióval [133]. Ezt egyszerűen lehet az == operátorral ellenőrizni. Így a `6`-os változóval egyeztetünk, és csak akkor írjuk ki a százalékos értéket ha egyezés van. Elvileg ennek egyetlen sorban kell megtörténnie a bemeneti fájlban. Utána a fenti kis példában már alkalmazott szűrést végezzük az **awk**-ban.

4. Az ehhez hasonló rendszer ellenőrző szkripteket időzítve futtatjuk egy felügyelő szkripttel, ez elküldi levélben a szkript kimenetét a rendszergazdának, ha probléma van.

Ez a szkript egyszerű megoldás, igazi élesben futó rendszereken általánosabb megoldást kell tervezni.

11. fejezet - Folyamatok kezelése

Folyamatok

Folyamatnak (*process*) nevezzük az operációs rendszerben a futó program egy példányát.

Amikor egy operációs rendszer egy programot futtat, akkor a program kódján kívül egyéb információkat is nyilvántart erről. Így például a programhoz rendelt memóriaterületet, megnyitott állományokat, környezeti változóit és még sok egyebet. Ezeket is kezelnie kell, így a futó program példánya több információt jelent mint a programkód. A Unix rendszerek egyik legfontosabb feladata a folyamatok kezelése. Interaktív munka során a héj szintjén a felhasználónak több lehetősége van befolyásolni a futó folyamatokat.

A Unix rendszerek folyamatai "párhuzamosan" futnak, legalábbis ez így tűnik a felhasználó számára. A folyamatok párhuzamos végrehajtását a kernel egyik komponense, az ütemező (*scheduler*) vezérli, így a folyamatok felváltva jutnak rövid futási időhöz a processzoron (a kis időintervallum neve *time slice*), amennyiben egy processzoros rendszeren futnak. Ha a rendszer több processzoros, a kernel megpróbálja ezt is kihasználni a párhuzamosítás érdekében. Ugyanakkor ezeken a rendszereken több felhasználó dolgozhat egyidejűleg. A folyamatokat az ütemező bizonyos prioritások szerint kezeli, így a fontosabbak gyakrabban jutnak futási időhöz

Interaktív munka közben a programokat (pl. a Unix parancsokat) a héjból indítjuk, magát az indítás folyamatát mindig a kernel végzi. A kernel adatstruktúrákat hoz létre, amiben a folyamat dinamikus leíró információit tartja. A folyamatok egyenként rendelkeznek az alábbi tulajdonságokkal:

- Minden folyamat rendelkezik egy saját védett memória zónával. Ez azt jelenti, hogy ezt csak a folyamat használja, és mások nem írnak/olvasnak belőle.

- Ha a parancsot interaktív módban indítottuk, akkor rendelkezik legalább 3, a folyamat indulásakor megnyitott állománnyal: standard ki és bemenet, valamint hibakimenet. Ugyanakkor ezeken az implicit bemenetet és kimenetet az interaktív terminál jelenti, ezt ilyenkor a folyamatot felügyelő terminálnak (*controlling terminal*) nevezzük.

- A folyamat megkapja a környezetét leíró változókat, név érték párok formájában, pl. `LANG='hu_HU.UTF-8'`. Ez lehetőséget ad arra, hogy különböző információkat juttassunk el a folyamathoz, a nélkül, hogy a a parancssort használnánk. Természetesen elsősorban az operációs rendszer által nyújtott futási környezetről tartalmaznak ezek a változók információt.

- Megkapja az indításakor beírt parancssorát. Amennyiben a folyamat bináris program képe, és pl. C nyelven íródott, a megfelelő sztringeket a `main()` függvény paramétereiként kapja meg. A héj feldolgozva adja oda a programnak, így pl. a * helyett be van írva a munkakönyvtárban található állományok listája. A héj parancssora összetettebb parancsot is tartalmazhat, amely egyszerre több folyamat indítását is tartalmazhatja, ennek szétválasztását az indító héj kezeli.

- A kernel egy prioritást rendel hozzá: erre azért van szükség, mert a fontosabb folyamatok nagyobb eséllyel kell futási időhöz jussanak, mint a kevésbé fontosak.

A folyamatlétrehozás gyorsan, kevés erőforrást használva történik a Unix rendszerek alatt. Ezért működik jól a Unixban az a stratégia, hogy a nagyobb feladatokat kis programok összekapcsolásával kell megoldani.

A folyamatok futásuk közben több állapotot vesznek fel. Erre azért van szükség, mert a kernel a folyamatokat felváltva engedi futni, hol megállítja és várakoztatja ("blokkolja") őket hol pedig egy adott időre teljesen felfüggeszti őket (pl. ha egy **sleep** parancsot hajtunk végre).

A folyamatokat a Unix **ps** nevű parancsának segítségével listázzuk (használatát lásd alább [156]). Mivel a folyamatok ütemezése nagyon kis időintervallumok alatt történik (pl. az ütemezési idő 10 miliszekundum körüli is lehet), nem látható minden köztes állapot a terminállal történő listázás folyamán (ennek nincs is értelme, a terminálon megjelenő szöveg kiírása több ideig tarthat, mint jó néhány folyamat egymás utáni ütemezése). A folyamatok fontosabb, **ps** által listázható állapotai a UNIX/Linux rendszerek alatt az alábbiak (Linuxon, a GNU **ps**-t használva):

11.1. táblázat - A ps által megjelenített folyamat állapotok

Az állapot neve	Az állapot betűjele	Leírás
Fut (*Running*)	R	A folyamat fut. Parancssoros munka esetében ide soroljuk azt az esetet is amikor a folyamat még nem fut, de futásra kész. Ilyenkor a folyamat arra vár, hogy processzoridőhöz jusson.
Várakozik (*Interruptible Sleep*)	S	A folyamat várakozik valamilyen eseményre vagy erőforrásra és megszakítható egy jelzés által. Például egy számlálóra (**sleep**) vagy valamilyen Be/Ki műveletre.
Eseményre vár (*Uninterruptible Sleep*)	D	A folyamat várakozik valamilyen eseményre vagy erőforrásra és **nem** szakítható meg jelzés által. Általában ez az állapot valamilyen be/kimeneti eszközre való várakozást jelent.
Fel van függesztve (*Stopped, Traced*)	T	Fel van függesztve és áll. Ide kerül egy folyamat ha a terminálról futtatva lenyomjuk a Ctrl-Z billentyűt.
Apátlan (*Zombie*)	Z	Ha egy folyamat kilép, az apa folyamatnak át kell vennie a fiú kilépési állapot adatait. Amíg ezt nem teszi meg, a fiú un. apátlan állapotba kerül, bár már nem fut. Ha az apa ezt nem teszi meg, az `init` fogja átvenni helyette.

A folyamatazonosító

A folyamatok azonosítására a rendszer egy folyamatazonosítónak nevezett egész számot használ(*process id* vagy *pid*). Ezen keresztül érjük el a gyakorlatilag folyamatot illetve annak tulajdonságait. Azt mondjuk, hogy az 1562 számú, vagy csak 1562-es folyamat fut, függesztődik fel, kilép, stb.

Amikor a rendszer indul, és a kernel elkezdi felépíteni a rendszert, egy `init` nevű folyamat az első amit elindít. Ezt az ütemező indítja, amelyiket a legtöbb rendszeren 0 számú folyamatként tartunk számon (neve *sched* , *swapper* vagy egyszerűen csak *kernel* a különböző rendszereken,

a **ps** parancs nem listázza ki). Az első valódi folyamat az `init` , amelynek azonosítója 1. A utána indított folyamatok mind tőle származnak, így a folyamatok összefüggésükben egy fa szerű struktúrát alkotnak. Ezt megjeleníthetjük a **pstree** paranccsal Linuxon: alább ennek kis részlete látszik (a zárójelben levő számok a folyamat azonosítót jelentik):

```
$ pstree -p
init(1)─┬─acpid(2991)
        ├─atd(3273)
        ├─auditd(2368)─┬─audispd(2370)───{audispd}(2371)
        │              └─{auditd}(2369)
        ├─automount(2964)─┬─{automount}(2965)
        │                 ├─{automount}(2966)
        │                 ├─{automount}(2969)
        │                 └─{automount}(2972)
...
```

A héj a saját folyamatazonosítóját a `$` speciális változójábanban tartalmazza, értékét a `$$` hivatkozással érjük el. Az interaktív héj folyamatazonosítóját tehát így listázhatjuk:

```
$ echo $$
16239
$
```

A Unix feladat (job) fogalma

A terminállal való munka során egy parancssorral egyszerre akár több folyamatot indíthatunk. Pl. az alábbi parancssor:

```
$ head -20 data.txt | sort | uniq
```

3 folyamatot indít és köztük csővezetékeket hoz létre. Ezért a terminállal történő munka számára szükséges egy másik fogalom is, ami jellemzi az egy parancssorral indított folyamatokat.

Egy parancssor által kiadott "feladatot" nevezi a UNIX "job" -nak. Ezeket külön követni lehet a Unix **jobs** paranccsával.

A fontosabb fogalmakkal amelyek a feladatok kezeléséhez tartoznak már találkoztunk:

- az előtérben és háttérben futtatás
- az **fg** és **bg** parancsok
- az `&` jel jelentése a parancssoron.

Az előtérben futtatott folyamatokat meg lehet szakítani a terminálról beütött megszakító karakterekkel, amelyek tulajdonképpen jelzéseket küldenek (lásd alább): `Ctrl-C` - megszakítás, `Ctrl-Z` - felfüggesztés, `Ctrl-\` kilépés program memóriaképének kiírásával (*core dump*).

Ha a parancssoron elindítunk egy folyamatot a háttérben, az utolsó indított folyamat azonosítóját a `!` speciális változóban találjuk és `$!` hivatkozással érjük el.

A folyamatok kezelésének illusztrálásához egy kis szkriptet fogunk használni, amelyik semmit sem végez, de alkalmas arra, hogy ciklusban hosszú ideig fusson periodikus várakozásokkal, így lesz időnk megfigyelni a történteket. A szkriptnek indításkor megadjuk egy paraméterben, hány másodpercig végezze a "semmit".

A `lassu.sh` szkriptünk kódja:

```
#!/bin/bash

if [ -z "$1" ];then
    echo használat: lassu.sh  másodperc
    exit 1
fi

n=$1                    #hány ciklust végez

while (( n-- ))
do
    sleep 1
done
```

Az alábbi példában a háttérben indítjuk a `lassu.sh`-t (pontosabban egy második **bash** értelmezőt, amelyik a `lassu.sh`-t értelmezi). Utána kilistázzuk az apa és fiú folyamat azonosítóit. Majd a **pstree**-vel az apa-fiú viszonyt (a **pstree** `-p` opciója a folyamat azonosítókat, a `-a` a folyamatot indító parancssort listázza). Ha a **pstree**-nek argumentumként egy folyamatazonosítót adunk, csak a kiválasztott folyamatot és a fiait listázza. Látható, hogy a **sleep** mint új folyamat jelenik meg a második **bash** alatt.

```
$ bash lassu.sh 200 &
[1] 12606
$ echo $!
12606
$ echo $$
23571
$ pstree -ap 23571
bash,23571
  ├─bash,12606 lassu.sh 200
  │    └─sleep,12735 1
  └─pstree,12737 -ap 23571
$
[1]+  Done                    bash lassu.sh 200
$
```

A szkriptet háttérben indítottuk: a []-ben található számot, amelyet a shell ír vissza "*job*" vagy feladat számnak nevezzük. A második szám (12606) a szkriptet futtató héj folyamatazonosítója. Észrevehető, hogy az indító héj `!` változója ezt tartalmazza. Ugyanakkor a `$` változó értéke egy másik azonosító, ez az általunk használt interaktív héjé. Ha 200 másodperc múlva egy újsort ütünk be, a héj kiírja a *Done* szót tartalmazó sorban, hogy véget ért a háttérben futó folyamat (az újsor leütése a kimeneti pufferben levő szöveget írja ki a terminálra). Az első háttérben indított feladat száma 1, ha több feladatot indítunk, ezek kis, növekvő egész számok lesznek (2,3,...).

A **jobs** parancs biztosítja a terminálról az elindított feladatok felügyeletét. Fontosabb opciói:

- `-p` csak a folyamatazonosítót listázza
- `-l` mindent listáz, a feladat folyamatait is külön-külön
- `-r` csak a futó folyamatokat listázza
- `-s` csak a leállított feladatokat (stopped) listázza

Az alábbi példában a `lassu.sh` szkript egymás után indított két példányán keresztül illusztráljuk használatát:

```
$ ./lassu.sh 500 &
[1] 18002
$ #a masodik az előterben, majd Ctrl-Z-vel felfüggesszük
$ ./lassu.sh 500

[2]+  Stopped                 ./lassu.sh 500
$ #kilistázzuk a futó feladatokat
$ jobs -l
[1]- 18002 Running                 ./lassu.sh 500 &
[2]+ 18108 Stopped                 ./lassu.sh 500
$ #kilistázzuk a futó feladatokat
$ #elindítjuk háttérben a másodikat is
$ bg 2
[2]+ ./lassu.sh 500 &
$ jobs -l
[1]- 18002 Running                 ./lassu.sh 500 &
[2]+ 18108 Running                 ./lassu.sh 500 &
$ #eloterbe hozzuk a kettest
$ fg 2
./lassu.sh 500
$
```

A "+" jel a job szám mellett az utolsó, a "-" jel az utolsó előtti indított folyamatot jelzi.

A wait parancs

Ha egy szkriptből háttérben indítunk egy másikat, és az előtérben fut, a szkript következő parancsa csak akkor indul, ha az előző szkript véget ér, pl. az alábbi `start.sh` szkriptben:

```
#!/bin/bash
./lassu.sh 100
echo vége
exit 0
```

, de ha a `lassu.sh`-t háttérben indítjuk, az indító szkript hamarabb ér véget, mint pl. az alábbiban:

```
#!/bin/bash
./lassu.sh 100 &
echo vége
exit 0
```

A Unix rendszerekben futó folyamatok általában egy futó folyamat fiaként kell létezzenek, így az indító folyamatnak meg kell várnia fiát, ezt a **wait** paranccsal oldhatjuk meg. Használata:

```
wait                #minden fiú folyamatra vár
wait n              #az n azonosítójú folyamatra vár
wait %n             #az n.-dik job folyamataira vár
```

Az opcionális **n** szám folyamatazonosítót jelent, ha `%` jel van előtte akkor *job* (feladat) azonosítót.

A **wait** parancs az argumentumában megadott job számhoz (`%n`) tartozó folyamatokra (mindegyikre) vagy egy bizonyos folyamatra (`n`) vár, utána visszaadja annak kilépési kódját. Ha hiányzik a folyamatazonosító, akkor minden fiú folyamatra vár.

`n` helyett szerepelhet egy lista is: `n1 n2 n3` … formában.

A **wait** végrehajtása alatt a shell nem indít új parancsot és nem olvassa a terminált sem.

Az alábbi programban kétszer indítjuk a `lassu.sh` szkriptet utána bevárjuk végrehajtásukat. A `lassu.sh` annyi másodpercig fut, amennyit kérünk tőle a parancssoron. Így az elsőnek indított példány hamarabb ér véget. A **wait** végrehajtásakor a program mindkettőt bevárja:

```
#!/bin/bash

./lassu.sh 3 &
./lassu.sh 10 &

wait

echo vége
exit 0
```

Folyamatok követése a ps paranccsal

A **ps** (*process status*) parancs a folyamatok követésére használt legfontosabb parancs. Gyakorlatilag minden, a kernel által nyilvántartott információt ki lehet vele listázni a folyamatokról.

Opciói System V (- jel az opció előtt) vagy BSD (- jel nélkül) stílusúak is lehetnek; a **ps** komplex parancs és sok opcióval rendelkezik. A GNU változata több UNIX rendszer **ps** parancshasználati módját is tudja emulálni, lásd ezzel kapcsolatban a **ps** kézikönyv oldalát. Ezért csak néhány egyszerű használati módot fogunk átnézni.

A **ps** egy táblázatszerű kimenetben írja ki a folyamatok információit. Első sora jelzi az oszlopokban kiírt információkat. Kimenetének legfontosabb oszlopai:

11.2. táblázat - A ps parancs kimenetének fontosabb oszlopai

Oszlop neve	Mit tartalmaz
PID	A folyamat folyamatazonosítója.

Oszlop neve	Mit tartalmaz
PPID	parent pid: az apa folyamat azonosítója.
%CPU	A processzornak mennyi idejét használja százalékban.
TIME	Mennyi időt futott eddig: processzor idő + rendszer idő.
COMMAND	A parancssora.
USER	A tulajdonos neve.
UID	A tulajdonos azonosítója.
C	A folyamat időhasználata: processzor idő / valódi idő . A szám az osztás egész részét tartalmazza. A %cpu kijelzés ugyanezt írja ki százalékokban.
PRI, RTPRIO, NI	A folyamat prioritásának értéke a PRI mezőben kerül kijelzésre. Ez az érték belső kernel változó, nem lehet kívülről módosítani (kisebb érték nagyobb prioritást jelent). Az RTPRIO (real time priority) olyan alkalmazások kapnak magasabb értéket amelyek valós időben működnek (pl. egy videó lejátszó). A folyamat nice száma az NI oszlopban jelenik meg: implicit programindításnál 0, értékeket 1-től 20-ig adhatunk minél nagyobb ez a szám, annál kisebb prioritással fut a folyamat. A PRI érték függ az RTPRIO és NICE számoktól, de az összefüggést a kernel számolja ki.
WCHAN	Annak a kernel függvénynek a címe amelyben a folyamat "alszik" (várakozik, wait chain).
TTY	Melyik a folyamatot kontrolláló terminál.
CMD	Milyen paranccsal indították.

Az alábbi táblázatban pedig megadjuk a ps néhány gyakran használt módját, a jobb oldali oszlopnban a ps ilyenkor használt opciói találhatóak.

11.3. táblázat - A ps parancs néhány gyakran használt parancssori opciója

Opciók	Mit listáz a ps

opció nélkül	A **ps** egyszerűen, opciók nélkül a folyamatazonosítót, terminált, elhasznált processzoridőt és a program nevét listázza, de csak azokat amelyek abban a terminálban lettek elindítva amelyet éppen használunk:
	
	`$ ps`
	`  PID TTY          TIME CMD`
	`15304 pts/0    00:00:00 bash`
	`15366 pts/0    00:00:00 cat`
	`15715 pts/0    00:00:00 bash`
	`15716 pts/0    00:00:00 sleep`
	`16874 pts/0    00:00:00 ps`

`-u nev`	A nev nevű felhasználó folyamatait listázza.
`-a`	Az összes terminál ablakban a felhasználó által elindított folyamatot listázza.
`-f`	full format: több információt listáz, megjelenik az indítási időpont, az apa folyamat azonosítója is.

Opciók	Mit listáz a ps
`-efl`	-e , every: minden folyamatot listáz, -l: long format részletes lista.
`uax`	Minden folyamatot listáz felhasználóbarát formában.

`-C lista`	A listában megadott nevű parancsok folyamatait listázza:
	
	`$ ps -C bash,lassu.sh`
	`  PID TTY          TIME CMD`
	`1373 pts/0    00:00:00 bash`
	`2976 pts/1    00:00:00 bash`
	`3009 pts/2    00:00:00 bash`
	`6167 pts/0    00:00:00 lassu.sh`

`-H`	A folyamatok hierarchiáját mutatja:
	
	`$ ps -H`
	`  PID TTY          TIME CMD`
	`15304 pts/0    00:00:00 bash`
	`15366 pts/0    00:00:00   cat`
	`15715 pts/0    00:00:00   bash`
	`15716 pts/0    00:00:00     sleep`
	`17582 pts/0    00:00:00   ps`

`ps -o formátum`	A `-o` opció után megadhatjuk, hogy milyen mezőket tartalmazzon a ps listája, pl.: `cmd` a folyamatnak megfelelő program neve, `pid` a folyamatazonosító, `ppid` az apa folyamatazonosítója, `start_time` mikor indították, `user` felhasználó neve, (a lehetőségeket lásd részletesen a ps kézikönyvében). Például:
	
	`$ ps -o pid,comm`
	`  PID COMMAND`
	`1373 bash`
	`6167 silent.sh`
	`7721 sleep`
	`7722 ps`

`-p szám`	Kérhetünk a ps-től listát csak egy bizonyos folyamatra:
	
	`$ cat > a.txt &`
	`[4] 9616`
	`$ ps  -p 9616`
	`  PID TTY          TIME CMD`
	`9616 pts/0    00:00:00 cat`

A top parancs

Dinamikusan a **top** paranccsal listázhatjuk a folyamatokat. A **ps** csak egy adott időpillanatban létező állapotot mutat. A **top** alkalmas a kiugróan sok erőforrást használó folyamatok listázására (processzoridő, memória). A **top** parancsai:

11.4. táblázat - A top fontosabb interaktív parancsai

Parancs	Mit végez
?	segédletet listáz
k	kill parancsot közvetít
r	renice: prioritást változtat
u	csak egy bizonyos felhasználó folyamatait mutatja
q	kilép
o	order: meg lehet adni interaktívan melyik oszlop szerint rendezzen

A **top** több parancssori argumentummal is rendelkezik, részletesen lásd a parancs kézikönyv oldalait.

Folyamatok futási prioritását parancssori indításkor a paranccsal módosíthatjuk, illetve a **renice**-al változtathatjuk. A folyamatok egy un. *nice* számmal rendelkeznek: ez nem a pontos prioritást adja meg, de információt jelent az ütemezőnek, és ez figyelembe is veszi. A folyamatok *nice* száma -20 -tól (legnagyobb prioritás) egészen 19-ig (legkisebb) terjed. A **nice** `-n` paranccsal n-et adunk az implicit *nice* számhoz. Argumentum nélkül a **nice** kilistázza az implicit *nice* számot. Az alábbi program pl. a legkisebb *nice* számmal fog futni:

```
$ nice -n 19 job.sh
```

Jelzések

A folyamatok normális esetben lefutnak és ha elvégezték feladataikat és befejeződnek, amennyiben shell szkriptek az **exit** paranccsal kilépnek (a C programozási nyelven írt programok a C `exit()` függvényével lépnek ki). Futás közben többféleképpen kommunikálnak egymással: vagy adatokat küldenek át egyik folyamatól a másikhoz (pl. állományokon vagy csővezetéken keresztül) vagy egyszerűen csak egymás feladatának összehangolása végett szinkronizációs információkat küldenek.

Az folyamatok közti kommunikáció egyik formája az aszinkron kommunikáció. Aszinkron üzenetnek azokat az üzeneteket nevezzük, amelyek bármely pillanatban elküldhetőek egy folyamathoz, függetlenül attól, hogy az mit végez: vár-e éppen erre, vagy nem. Az aszinkron üzenetek egyik formája a jelzések általi kommunikáció.

A jelzés (*signal*) egy aszinkron értesítés amelyet egyik folyamat küld a másiknak (vagy a kernel egy folyamatnak) valamilyen esemény bekövetkeztéről. A folyamatok jelzéseket kiszolgáló függvényeket, un. jelzés kezelőket (*signal handler*) definiálhatnak. Amikor a folyamathoz egy jelzés érkezik, akkor normális futása megszakad, és a jelzéskezelő indul el. Amennyiben a folyamat nem definiált kezelő függvényt, a jelzés implicit kezelője szolgálja ki azt.

A jelzésekre egy rövid nagybetűs névvel (pl. `INT`) vagy egy egész számmal (pl. 2 -es jelzés) hivatkozunk. Bizonyos esetekben, pl. a C nyelv fejléc állományaiban vagy a **kill** `-l` parancs listázásakor a név elé kerül a `SIG` szócska, tehát így találkozunk velük: `SIGINT`.

A legtöbb jelzésnek van egy implicit kezelési sémája. Van két olyan jelzés, amelyre a folyamatok nem definiálhatnak kiszolgáló függvényt (`KILL` és `TSTP`, lásd alább).

A Unix rendszerek nem egyformán definiálják a rendszerben lehetséges jelzéseket, illetve a hozzájuk rendelt azonosítókat. Az alábbi jelzések minden rendszeren megvannak, és a héj programozás példáihoz elegendőek. A felsorolt jelzések kiszolgálójánál nem kötelező az alábbi előírásokat implementálni. Ezek inkább ajánlások, és a legtöbb program ezeket implementálja. De elvileg, a jelzés megérkezése után a jelzést kiszolgáló függvény bármit implementálhat.

11.5. táblázat - A fontosabb jelzések

Jelzés neve	Száma	Jelentés
Terminálról küldött jelzések:		
TSTP	20	Felfüggeszti a folyamatot a terminálról. A Ctrl-Z karakter leütése után ez hajtódik végre.
INT	2	Megszakítja és azonnal leállítja a folyamatot a terminálról , a Ctrl-C karakter leütése váltja ki.
QUIT	3	Leállítás: a folyamat lezárhatja állományait, takaríthat de kilép (Ctrl-\ váltja ki a terminálról).
Folyamatokból küldött jelzések (parancssorról a **kill** paranccsal):		
KILL	9	Feltétel nélkül azonnal leállítja a futó folyamatot, mindig egy folyamat küldi folyamatnak. A KILL jelzést nem lehet kezelni.
ABRT	6	Abort - leállítás, de előtte un. *core dump* (program memóriaképe) nyomtatása.
HUP	1	*Hang up*: ha a folyamat interaktívan fut, és megszakad a kapcsolat a felügyelő terminállal, akkor ezt a jelzést kapja. De gyakori egy folyamat újraindítására való használata is (konfigurációs állományok újraolvasása, inicializálás).
TERM	15	Szintén leállítást kér (*terminate*), egy másik folyamat küldi. Akárcsak a QUIT esetén, ajánlott, hogy leállás előtt a folyamat elvégezze takarítási feladatait. A legtöbb jelzés küldő parancs ezt küldi implicit jelzésként.
USR1, USR2	10,12	*User*: tetszés szerinti művelet programozására használt jelzések.
FPE	8	Lebegőpontos művelet hiba, implicit kezelése a leállás.
STOP	19	Program által küldött jelzés felfüggesztésre.Program által küldött jelzés felfüggesztésre.
CONT	18	Újraindítja a STOP-al leállított programot.
CHLD	17	Ezt a jelzést akkor kapja a folyamat ha egyik fiú folyamata kilép.

Látható, hogy a legtöbb jelzés valamilyen rendkívüli, bármikor előforduló eseményhez kapcsolódik, és implicit következményük a folyamat leállítása. Hirtelen leálláskor, ha a folyamat pl. ideiglenes állományokkal dolgozik, fontos lehet a "takarítás", pl. a már nem szüksége állományok törlése. Sok jelzés hagy erre időt, tehát írható jelzéskezelő, amely elvégzi ezt a feladatot.

A kill parancs

Jelzést a **kill** paranccsal küldhetünk. Használata:

`kill` [-n] *`folyamatazonosító(k)`*...

ahol az n egy egész szám, a jelzés száma. Helyette használhatjuk a jelzés nevét is:

`kill` [-JELZÉSNÉV] *`folyamatazonosító(k)`*...

Használatos még:

`kill` -s JELZÉSNÉV *`folyamatazonosító(k)`*...

alakban. Mindhárom alak a megadott jelzést küldi a folyamatazonosítóval rendelkező folyamatnak vagy folyamatoknak. A fenti alakokon kívül, a folyamatazonosító speciális értékeire (0, -1, -n) folyamat csoportoknak is küldhető jelzés (lásd a **kill** kézikönyvét).

A jelzések neveit és azonosítóit a **kill** `-l` paranccsal listázhatjuk ki. A `-p` opciója csak kilistázza azokat a folyamatokat amelyeknek jelzést küldene, de nem küldi el a jelzést (tesztelésre használjuk).

Jelzést csak azoknak a folyamatoknak küldhetünk, amelyekre az engedélyezve van (jogrendszer). Futást módosító jelzéseket csak saját nevünk alatt futó programnak küldhetünk (kivétel természetesen `root`, ő minden programnak küldhet).

A **kill** igaz értékkel tér vissza ha a jelzést sikerült elküldeni, egyébként hamissal (1).

Terminálról billentyűkkel 3 fontosabb jelzést küldhetünk a folyamatoknak: az `INT` azonnal megszakítja a folyamatot, a `TSTP` (*terminal stop*) felfüggeszti és a `QUIT` szintén leállásra szólítja fel, időt hagyva "tisztogatás"-ra. Mindhárom jelzés kezelhető a folyamat által (tehát a folyamat futtathat más kódot is mint az implicit viselkedés).

Folyamatokat folyamatokból négy jelzéssel állíthatunk le: `HUP` (*hangup*), `TERM` (*terminate*) és `KILL` illetve `ABRT` (*abort*). A `TERM` és `ABRT` esetében a folyamat kilépés előtt "tisztogatást" végezhet: lezárhatja állományait, pl. letörölheti ideiglenes állományait. Az `ABRT` lementheti lemezre a program memóriaképét (*core dump*).

A `HUP` jelzés is leállást kér, jelentése: megszakadt a kapcsolat a kontrolláló terminállal. Több démon folyamat esetében csak újra indítást, vagy egyszerűen a program konfigurációs állományának beolvasását kéri.

A `KILL` azonnal leállítja a folyamatot, a `STOP` pedig felfüggeszti azt (*suspend*).

Mivel a `KILL` esetében a programnak nem hagyunk időt a tisztogatásra, így ideiglenes vagy egyéb állományok maradhatnak utána az állományrendszerben.

Ha egy program `sleep` állapotban van (*Uninterruptible Sleep*, valamilyen Be/Ki eszközre vár pl.), akkor jelzés által nem megszakítható csak akkor, ha kilép ebből az állapotból. Bizonyos

jelzéseket (pl. USR1) a sleep parancs felfüggesztése alatt sem kap meg a szkript: ilyenkor a megszakító jelzéseket (pl. TERM megkapja).

Az alábbi példában a lassu.sh-t indítjuk. Miután a **ps**-el megállapítjuk a folyamatazonosítót, a TERM jelzéssel állítjuk le.

```
$ bash lassu.sh 200 &
[1] 14835
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   500 14835 23571  0  75   0 -  1117 wait   pts/4    00:00:00 bash
0 S   500 14859 14835  0  75   0 -   926 -      pts/4    00:00:00 sleep
0 R   500 14861 23571  0  77   0 -  1050 -      pts/4    00:00:00 ps
0 S   500 23571 23570  0  75   0 -  1163 wait   pts/4    00:00:00 bash
$ ps -O stat
  PID STAT S TTY          TIME COMMAND
14835 S    S pts/4    00:00:00 bash lassu.sh 200
14909 S    S pts/4    00:00:00 sleep 1
14910 R+   R pts/4    00:00:00 ps -O stat
23571 Ss   S pts/4    00:00:00 -bash
$ kill -TERM 14835
$
[1]+  Terminated              bash lassu.sh 200
$
```

Egy adott rendszeren létező jelzések nevét, jelentését és számát a **kill** -l , jelentésüket a man 7 signal paranccsal listázhatjuk ki.

A killall parancs

Jelzéseket a **killall** paranccsal is küldhetünk. Ez a végrehajtható program neve szerint vagy akár a névre illesztett reguláris kifejezésekkel is megtalálja a futó folyamatokat. Ezért használata néha kényelmesebb mint a **kill**-é.

A **killall** legegyszerűbb használata az alábbi:

```
$ killall firefox
```

ez TERM jelzést küld a firefox böngésző összes futó példányának (a TERM az implicit jelzés, ha nem adjuk meg mit küldünk). KILL jelzést így küldhetünk:

```
$ killall -s KILL firefox
```

A cél folyamatokat reguláris kifejezésekkel is megadhatjuk. Az illesztést bővített kifejezésekkel végzi, és kérhetünk a folyamat parancs nevére való teljes vagy részleges illesztést.

```
$ ./lassu.sh 100 &
[1] 29301
$ killall -r '^la'
```

```
[1]+  Terminated              ./lassu.sh 100
```

Tehát abban az esetben, ha reguláris kifejezéssel keressük a folyamatot, a `-r` opciót kell használni.

A **killall** a `-i` opcióval megerősítést kér minden jelzés küldése előtt, a `-v` opcióval üzenetet ír ki minden elküldött jelzés után. Több opciója tud szelektíven kiválasztani folyamatokat (lásd az angol nyelvű aktuális kézikönyv lapot).

A pgrep és pkill parancsok

Modern Unix rendszereken vannak olyan parancsok, amelyek megengedik, hogy a folyamatok azonosítójára programnevükön keresztül hivatkozzunk. Pontosabban, a programnévre illeszkedő reguláris kifejezést megadva, megkapjuk az azonosítót. A **`pgrep`** `név` kilistázza a név program folyamatazonosítóját, a **`pkill`** `-JELZÉS név` pedig név szerint küld jelzést.

A **pgrep** is bővített reguláris kifejezéseket használ:

```
$ cat > a.txt &
[4] 9616
$ # a cat a háttérben fut, megkeressük:
$ ps  -p $( pgrep '^cat$')
  PID TTY          TIME CMD
 9616 pts/0    00:00:00 cat
$
```

A név szerint való kiválasztáson kívül más szűkítésekre is alkalmas:

`-u felhasználó` csak egy bizonyos felhasználó folyamatait keresi,
`-f` (*full*) a teljes parancsorra illeszt, nem csak a parancsnévre,
`-o` (*oldest*) a legrégebb indított folyamatot válassza ki,
`-n` (*newest*) a legutóbb indítottat válassza a találatok közül,
`-P ppid` csak azokat válassza amelyeknek apa folyamat azonosítója `ppid` (*parent pid*),

Más opciók megtekinthetők a `man pgrep` kézikönyv lapon.

A **pgrep** is tud jelzést küldeni `-jelzés` opcióval egyes rendszereken (a GNU pgrep a Linuxon nem tud), de erre általában a parancs párját a **pkill**-t használjuk. A **pkill** ugyanazokat az opciókat fogadja, mint **pgrep**, részletekért lásd a parancs kézikönyv lapját, alább pedig lássunk egy példát:

```
$ cat > 1.txt &
[1] 29851
[1]+  Stopped                 cat > 1.txt
$ pgrep '^cat$'
29851
$ pkill -KILL  '^cat$'
[1]+  Killed                  cat > 1.txt
$
```

Jelzések kezelése shell alatt: a trap parancs

A jelzéseket shell alatt a **trap** (shell-be épített) parancs segítségével lehet elkapni: a **trap** tulajdonképpen egy jelzéskezelő függvényt állít be. Használata az alábbi:

`trap` tevékenység jelzéslista

A tevékenység egy `;` -el elválasztott és két idézőjel közti parancssorozatot vagy egy shell függvényhívást jelent, a jelzéslista pedig a kezelt jelzés vagy jelzések nevét vagy számát.

A **trap** `-l` opcióval kilistázza a létező jelzéseket és a hozzájuk rendelt azonosítókat, a `-p` opcióval pedig a jelzésekhez rendelt **trap** parancsokat.

Pl. ha egy shell programban meghívjuk az alábbi parancsot:

```
trap "echo Ez egy Ctrl-C "  INT
```

a program bemenetére kerülő `Ctrl-C` nyomán a program nem áll le, hanem kiírja a látható szöveget.

A **trap** parancsot mindig a program elejére kell elhelyezni. Egy jelzéskezelő akár több jelzést is fogadhat:

```
trap "echo Ez egy megszakító jelzés !"  INT TERM QUIT
```

A **trap**-el nem lehet kezelni a `KILL` jelzést.

Speciális, programkövetésre használt jelzések (DEBUG, EXIT, ERR)

Néhány speciális jelzés kimondottan a héjjal való programkövetési munkához opciódik.

A `DEBUG` jelzés minden program sor végrehajtása után lesz meghívva (bizonyos héjaknál végrehajtás előtt: ellenőrizzük le).

Az `EXIT` jelzés a szkriptből való kilépésnél hívódik meg, és ha több jelzésre írunk kezelőt, ez hívódik meg utoljára.

Az alábbi program folyamatosan, minden sor végrehajtása után listázza az `a` változó értékét:

```
#!/bin/bash

function debuginfo
{
    echo "debug: az a értéke,  a=$a"
}
```

```
trap "debuginfo" DEBUG

a=2

a=3

a=4
```

Végrehajtva:

```
$ bash debug.sh
debug: az a értéke,  a=
debug: az a értéke,  a=2
debug: az a értéke,  a=3
$
```

Az alábbi program pedig az `EXIT` jelzés hatására kilépéskor törli a használt ideiglenes állományt. Ideiglenes állományok nevében gyakran használjuk a `$` változót, azaz a folyamatazonosítót, hiszen ez a szám egyedi, és így a program akár két példányban is futhat, az egyik nem fogja zavarni a másikat.

```
#!/bin/bash

#takarító függvény
function takaritas
{
    #ha létezik a fájl
    if [ -f $$.txt ]
    then
        rm $$.txt  #törli az ideiglenes fájlt
    fi
}

#a terminálról jövő megszakító jelzéseken
#kívül az exit hívásokra is ez a kezelő aktív

trap "takaritas" INT TERM QUIT EXIT

#ideiglenes fájl előállítása
echo első >> $$.txt
echo második >> $$.txt
echo harmadik >> $$.txt

cat $$.txt | sort > eredmeny.txt

exit 0
```

A mktemp parancs

Ideiglenes állománynevek előállítására a **mktemp** nevű parancs alkalmasabb. Ez random számokból álló nevet generál, és a számok generálásakor felhasználja

a folyamatazonosítót is. A generálandó név sablonját meg kell neki adni egy állománynéven keresztül, melyben a random szám részt X karakterekkel helyettesítjük a név végén. A parancs létrehozza az állományt (csak a felhasználó írás és olvasás jogával) és ugyanakkor kilistázza a kimenetre is, hogy a név átvehető legyen. Olyan alkalmazásokban kell használni, ahol egy generált név nem kell kitalálható legyen. Használatára alább adunk egy példát:

```
$ mktemp "fileXXXXXXX"
filecO30382
$ ls -l
total 0
-rw------- 1 lszabo lszabo 0 2010-12-13 20:50 filecO30382
$
```

Héjprogramokban így használjuk:

```
TEMPFILE=$( mktemp "fileXXXXXXXXX" )
echo "titkos" > $TEMPFILE
```

Egyes héjakban, pl. a Bash-ben is működik egy `ERR` nevű jelzés is: ez akkor hívódik meg, ha a héj valamilyen súlyos hiba következtében kilép, és hibát jelez.

A nohup parancs

Háttérben való futtatáshoz a **nohup** parancsot használjuk abban az esetben ha ki akarunk lépni a rendszerből, és azt szeretnénk, hogy az indított programunk tovább fusson. Az így futtatott parancs vagy szkript nem válaszol a HUP jelzésre. Ugyanakkor a futtatott parancs kimenete egy napló állományba lesz vezérelve (a GNU rendszereken ez a `nohup.out` nevű fájl. Használata:

```
$ nohup bash hosszu_feladat.sh  &
[1] 32280
$
```

Ha a kimenetet átirányítjuk egy állományba, a `nohup.out` nem jön létre.

Alhéjak (subshell)

Az alfejezet anyaga kizárólag a Bash-re érvényes.

Amint láttuk, minden folyamat tulajdonképpen fiú folyamat, az `init` kivételével mindegyiknek van "apja", és ha a héj alatt egy szkriptet vagy parancsot futtatunk, akkor azok fiú folyamatban fognak futni (amennyiben a parancs nem beépített parancsa a héjnak és nem az **exec** parancs). Fiú folyamatot egy szkripten belül, akár néhány szerkezet végrehajtására is indíthatunk, ezt szintaktikailag a szekvencia `( )` zárójel operátorok közé helyezésével adjuk meg a héjnak.

Tekintsük az alábbi szkriptet (`sub.sh`):

```
#!/bin/bash

#ezt a változót az külső shellben hozzuk létre
valtozo=1

#innen kezdve új shellben fut a szkript
(

#a külső shell változója

if [ -n "$valtozo" ] ; then
    echo a 'változo értéke:' "$valtozo"
fi

while read -p "kérek egy sztringet:" line
do

    if [ "$line" == "ok" ] ;then
        break
    fi

done

echo "Az utolsó beolvasott sor a subshell-ben: $line"

)
#a subshell itt zárul, ez után ismét a hívó shell fut

echo "Az utolsó beolvasott sor a hívó shell-ben: $line"
```

A `( )`-ekkel határolt parancsok un. subshellben fognak futni, ez látható, ha a szkriptet futtatva:

```
$ ./sub.sh
a változo értéke: 1
kérek egy sztringet:
```

és a **read** végrehajtásánál várakoztatva egy másik terminálból megnézzük a futó parancsainkat:

```
$ ps -a -o pid,ppid,command
  PID  PPID COMMAND
27314 21534 /bin/bash ./sub.sh
27315 27314 /bin/bash ./sub.sh
27863 23472 ps -a -o pid,ppid,command
$
```

A `sub.sh` parancs két folyamatazonosítóval fut, és a másodiknak az első az apja.

Ha beírjuk a **read**-nek az "ok" sztringet:

```
$ ./sub.sh
a változo értéke: 1
```

```
kérek egy sztringet:ok
Az utolsó beolvasott sor a subshell-ben: ok
Az utolsó beolvasott sor a hívó shell-ben:
```

akkor az is látható lesz, hogy a subshellben létrehozott változó nem lesz látható az apa shellben (`line` változó), ellenben az apa változói a subshellben globálisan viselkednek, akár a függvények esetében (`valtozo` változó). A végrehajtási módból az is kiderül, hogy a subshell előtérben fut, és az apa megvárja annak kilépését.

Tekintsük az alábbi, módosított szkriptet (`sub1.sh`):

```
#!/bin/bash

#ezt a változót az külső shellben hozzuk létre
valtozo=1

#innen kezdve új shellben fut a szkript

./loop.sh

#a subshell itt zárul, ez után ismét a hívó shell fut

echo "Az utolsó beolvasott sor a hívó shell-ben: $line"
```

amelyben a kis beolvasó ciklust külön szkriptbe írtuk át (`loop.sh`):

```
#!/bin/bash
#a külső shell változója

if [ -n "$valtozo" ] ; then
    echo a 'változo értéke:' "$valtozo"
fi

while read -p "kérek egy sztringet:" line
do

    if [ "$line" == "ok" ] ;then
        break
    fi

done

echo "Az utolsó beolvasott sor a subshell-ben: $line"
```

Nyilván ebben az esetben is a `loop.sh` szkript külön shellben fut, a két végrehajtási mód azonban nem egyezik teljesen.

Ha megismételjük az előző tesztet, látható lesz, hogy ezúttal a külső shell változói nem lesznek láthatóak a belső shellben, és a belsőben létrehozott változók sem a külsőben.

Amikor parancsokat futtat, a Bash un. végrehajtási környezetekkel (*execution environment*) dolgozik, amennyiben a futtatott parancs külső parancs (pl. a külső programként létező **cat**,

ls vagy egy új shell amelyik egy adott szkriptet hajt végre) ez a környezet az alábbiakat tartalmazza:

az indító shell nyitott állományait amelyeket átirányítások módosíthatnak,
a munkakönyvtárat,
az `umask` örökölt értékét (fájl létrehozási környezet öröklődik),
az **export** paranccsal a környezeti változók közé írt változókat.

Ha a parancs belső parancs (pl. a belső **echo**) vagy a `( )` operátor között fut, akkor subshellben fog futni, ilyenkor a shell duplikálja saját végrehajtási környezetét: ebbe az említetteken kívül beletartoznak:

a shell előzőleg létrehozott változói,
a futás alatt előzőleg definiált függvények,
a **trap** paranccsal beállított jelzéskezelők,
az **alias** paranccsal beállított nevek,
a **shopt** és **set** parancs beállított opciói.

Így érthetővé válik az első példa globális változóinak viselkedése. Sem a subshell, sem az új végrehajtási környezet nem módosíthatja a hívó shell futási környezetét.

Befejezésként még két megjegyzés a fejezethez:

A subshell `( )` és a `{ }` kód blokk [110], bár átirányítási mechanizmusként hasonlónak tűnik, és interaktív végrehajtás esetén nem érzékelhető a különbség: merőben más végrehajtási struktúra, mert a subshell esetében új folyamat jön létre.

A subshell is alkalmas párhuzamos futtatásra, de komplikált mellékhatásokkal járhat, ezért nagyon óvatosan kell használni, ha változónevek, fájlnevek átfedődnek.

Példaprogramok

A rendszerben futó folyamatok felügyelete (indítása, leállítása, követése és statisztikák készítése) általában héjprogramok feladata. Az alábbiakban néhány egyszerű példát adunk ebből a témakörből.

A szkripteknek azonosítani kell a futó folyamatokat, ez rendszerint azok folyamatazonosítójának megszerzésével jár. A felhasználó a folyamat nevével hivatkozik ezekre, így szükséges a névhez rendelt azonosító megszerzése. Erre több módszer is van.

Ha a követett folyamat héjprogram, ajánlatos névvel indítani és nem argumentumként használni a héj parancssorában, tehát:

```
$ bash teszt.sh
$
```

helyett:

```
$ ./teszt.sh
```

```
$
```

vagy:

```
$ /home/lszabo/teszt.sh
$
```

, így a szkript fájlra végrehajtási jogot kell adni.

Meg lehet keresni a nevet a parancssorban is, de általában egyszerűbb, ha a parancssori programnév egyezik a szkript nevével.

A klasszikus módszer a **ps** parancs és az **egrep** vagy **grep** használata. A **ps** paranccsal minden folyamatot listázunk részletes listával. Pl. ha elindítjuk háttérben már használt `lassu.sh` szkriptet:

```
$ ./lassu.sh 10000 &
[1] 23611
$
```

A **ps -f** opcióval az alábbi listát kapjuk:

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
lszabo   23016 23015  0 17:56 pts/0    00:00:00 -bash
lszabo   23611 23016  0 18:17 pts/0    00:00:00 /bin/bash ./lassu.sh 10000
lszabo   23891 23611  0 18:20 pts/0    00:00:00 sleep 1
lszabo   23892 23016  0 18:20 pts/0    00:00:00 ps -f
$
```

Ilyenkor a **ps** csak a terminálhoz rendelt folyamatokat listázza. Ha rendszer szinten akarunk keresni, a `-e` (every) opciót is használva, egy hosszabb listát kapunk amelyből az **egrep**-el szűrhetünk a parancs neve után:

```
$ ps -ef | egrep 'lassu\.sh'
lszabo   23611 23016  0 18:17 pts/0    00:00:00 /bin/bash ./lassu.sh 10000
$
```

A listában az azonosító a második mező, ezt az **awk**-val szűrjük:

```
$ ps -ef | egrep 'lassu\.sh' | awk '{print $2}'
23611
$
```

és írjuk változóba:

```
$ pid=$( ps -ef | egrep 'lassu\.sh' | awk '{print $2}' )
$ echo $pid
23611
$
```

Ha a folyamat több példányban fut, ennek megfelelő számú sort kapunk a **ps** kimenetének szűrése után (még 2 `lassu.sh`-t indítunk):

```
$ ./lassu.sh 100 &
[2] 25119
$ ./lassu.sh 100 &
[3] 25125
$ ps -ef | egrep 'lassu\.sh' | awk '{print $2}'
23611
25119
25125
$
```

, a sorok számát a **wc**-al megszámolva:

```
$ ps -ef | egrep 'lassu\.sh' | awk '{print $2}' | wc -l
3
$
```

megkapjuk, hány példányban fut a `lassu.sh`. A példányszám ismeretében pedig a felügyelő szkriptek dönthetnek további tevékenységről (pl. ha túl sok példány fut, leállíthatnak egyes példányokat).

Ha a folyamat nevében nincs pont karakter, az **egrep** vagy **grep** becsaphat, ti ilyenkor saját magát tartalmazó **ps** sort is listázhatja. Ha átnevezzük a `lassu.sh`-t `lassu`-ra, és úgy indítjuk:

```
$ cp lassu.sh lassu
$ ./lassu 200 &
[2] 25986
$ ps -ef | egrep lassu
lszabo   25986 23016  0 18:42 pts/0    00:00:00 /bin/bash ./lassu 200
lszabo   26081 23016  0 18:42 pts/0    00:00:00 egrep lassu
$
```

Látható, hogy találat lett az **egrep lassu** parancs is. Ezt az **egrep** `-v` opciójával szűrhetjük, az **egrep**-re keresve (a `-v` a "*nem találat*" sorokat listázza):

```
$ ps -ef | egrep lassu | egrep -v egrep
lszabo   26319 23016  0 18:45 pts/0    00:00:00 /bin/bash ./lassu 200
$
```

Ettől kezdve ugyanabban a helyzetben vagyunk, mint az első esetben.

Modern rendszereken egyszerűbben kereshetünk a már említett **pgrep** paranccsal, ha megadunk egy névre illeszkedő reguláris kifejezést:

```
$ ./lassu.sh 200 &
[3] 27090
$ pgrep 'lassu\.sh'
27090
```

```
$
```

Az alábbi szkript ezt a módszert használja, és a következőket végzi: vár amíg a követett nevű folyamat megjelenik, egy napló fájlba írja az adott névvel futó folyamatok példányszámát; utána már csak azt naplózza, ha változik a futó példányok száma; ha már egy példány se fut kilép.

```
#!/bin/bash
# A paraméterként névvel megadott folyamatokat követi és naplózza
# a. ha nem indult el egy példány sem, vár amíg elindul
# b. utána követi hány példány fut
# c. INT, TERM, QUIT, HUP jelzésre kilép
# d. ha már egy példány se fut követett programból, kilép
# e. az eseményeket naplózza
#Paraméterek:
#  $1 - a program vagy szkript neve
#  $2 - opcionális, hány másodpercet vár 2 ciklus között
#       implicit 1 másodperc
#
#használat:
#   follow_process.sh  programnév [ másodperc ]
#

#napló fájl neve
LOGFILE="follow_process.log"

#jelzés hatására kilép
function exit_script ()
{
    echo follow_process: jelzés miatt leállt
    exit 1
}

#jelzéskezelő beállítása
trap exit_script INT QUIT TERM HUP

#napló állományba írja az első paramétert, elé teszi a
#datumot és időpontot
function log()
{
    #időpont lekérdezése
    d=$( date +%Y-%m-%d:%T )
    echo follow_process: "$d" "$1"  >> $LOGFILE
}

#ha nem létezne a napló, létrehozzuk
touch $LOGFILE

#paraméterek megfelelőek-e
if (( $# == 0 || $# > 2 )); then
    echo "használat:  follow_process.sh programnév [ másodperc ]"
    exit 1
fi

prog="$1"   #a program neve az első paraméter
```

```
#a várakozási másodpercek száma
sec=1   #az implicit érték
if (( $# == 2 ))
then
    sec="$2"
    #leteszteljük, hgy egész szám-e
    if ! echo "$sec" | egrep -q '^[0-9]+$'
    then
        echo follow_process: a második paraméter egész szám!
        exit 1
    fi
fi

#a program nevében csak a . lehet reguláris kifejezésekben
#használt metakarakter, feltételezzük, hogy ez így van
#atírjuk a . -ot \. -ra
prog_re=$( echo "$prog" | sed -r 's%\.%\\.%;' )

#addig megy végtelen ciklusban, amíg megjelenik a program
while ! pgrep "$prog_re" >/dev/null
do
    sleep $sec
done

#hány példány fut
N=$(  pgrep "$prog_re" | wc -l )
log "$N $prog példány fut"

#követjük
while true
do
    #hány példány
    NA=$( pgrep "$prog_re" | wc -l )
    #ha egy sem kilép a ciklusból
    if (( NA == 0 )) ;then
        break
    fi
    #követő változó átírása és naplózás
    if (( NA != N )) ;then
        N=$NA
        log "$N $prog példány fut"
    fi
    #késleltetés
    sleep $sec
done

log "minden $prog leállt"

echo follow_process: kilép

exit 0
```

Ha a szkript nevében ott van a `.` karakter , átírjuk `\.` -ra, előállítva a szükséges bővített reguláris kifejezést. Ha nagyon pontosak szeretnénk lenni, a két határoló horgonyt is beírhatjuk:

```
prog_re=$( echo "$prog" | sed -r 's%\.%\\.%;s%(.*)%^\1$%' )
```

A **pgrep** mindenképpen a program nevére fog illeszteni, függetlenül attól, hogy teljes elérési úttal, `./` munkakönyvtár címmel a név előtt vagy csak a nélkül indítottuk.

A napló sztringben dátum és időpont előállítására a rendszer **date** parancsát használtuk. Paraméter nélkül saját formátumú dátumot, időpontot ír ki:

```
$ date
Mon Sep 26 00:22:28 EEST 2011
$
```

Formázó sztring paraméter hatására a formátumot módosíthatjuk. A formázó sztringet a + jellel vezetjük be, az egyes formátum mezőket a `%` jellel:

```
$ date +%Y-%m-%d:%T
2011-09-26:00:21:35
$
```

A részletekért, lehetőségekért lásd a **date** kézikönyv lapját.

A programot futtatva:

```
$ ./follow_process.sh lassu.sh 1
```

és egy másik terminálról `lassu.sh` szkripteket több példányban indítva, leállítva ez kerül a `follow_process.log` fájlba:

```
$ cat follow_process.log
follow_process: 2011-09-26:00:03:55 1 lassu.sh példány fut
follow_process: 2011-09-26:00:03:56 2 lassu.sh példány fut
follow_process: 2011-09-26:00:04:05 1 lassu.sh példány fut
follow_process: 2011-09-26:00:04:16 2 lassu.sh példány fut
follow_process: 2011-09-26:00:04:21 1 lassu.sh példány fut
follow_process: 2011-09-26:00:04:22 2 lassu.sh példány fut
follow_process: 2011-09-26:00:04:25 3 lassu.sh példány fut
follow_process: 2011-09-26:00:04:27 2 lassu.sh példány fut
follow_process: 2011-09-26:00:04:30 1 lassu.sh példány fut
follow_process: 2011-09-26:00:04:46 minden lassu.sh leállt
$
```

Ha a feladat az lenne, hogy a szkript a felügyelet mellett ne engedjen csak egy bizonyos számú `lassu.sh` szkriptet futni, akkor a `while` ciklust a következőképpen módosíthatjuk:

```
#ez a változat leállítja a legrégebb futó lassu.sh-t
#ha a példányszám meghalad egy maximális értéket
MAX=3

#követjük
```

```
while true
do
    #hány példány
    NA=$( pgrep "$prog_re" | wc -l )
    #ha egy sem kilép a ciklusból
    if (( NA == 0 )) ;then
        break
    fi

    #követő változó átírása és naplózás
    if (( NA != N )) ;then
        #ha nőtt a példányszám
        if (( NA > N ))
        then
            #meghaladta a maximumot
            if (( NA > MAX ))
            then
                #a legrégebb indult folyamatszáma
                oldest=$( pgrep -o "$prog_re" )
                log "$prog száma $NA, TERM elküldve"
                #TERM jelzéssel álítjuk le, ez az implicit
                if ! kill  $oldest > /dev/null
                then
                    log "nem lehet TERM jelzést küldeni, pid=$oldest"
               echo follow_process: TERM hiba, kilépés
                    exit 1
                fi
                #vár, hogy biztos kilépjen a folyamat
                sleep 0.5
                #az N megváltozott, újra lekérjük
                N=$( pgrep "$prog_re" | wc -l )
            else
                N=$NA
            fi
        else
            N=$NA
        fi
        log "$N $prog példány fut"
    fi

    #késleltetés
    sleep $sec
done
```

A **pgrep** `-o` opcióval a legrégebben futó folyamat azonosítóját adja vissza, ennek küldünk `TERM` jelzést. Sikertelen jelzésküldés esetén kilépünk, bár ilyenkor még megpróbáljuk a `KILL` jelzés küldését.

Folyamatok automatikus leállítása kényes kérdés, nagyon óvatosan kell vele bánni egy éles rendszerben, akárcsak a többi folyamatkezelő feladat, különösen ha rendszergazdaként futtatjuk a programokat. A bibliográfiában említett Michael2003 [184] munka részletesen mutatja be ezeket a kérdéseket.

A. függelék - Hibakeresés héjprogramokban

A héjprogramok fejlesztéséhez nem jellemzőek a C++, Java, stb. fejlesztésre jellemző keretrendszereket, amelyekkel kényelmesen követhető a programok fejlesztése, a szerkesztés mellett többek közt a tesztelés és hibakeresés.

Szövegszerkesztőként bármilyen Unix alatt futó szövegszerkesztőt használhatunk, mindegyik támogatja a shell szkriptek szerkesztését, akár terminálon szerkesztünk (vim, emacs) vagy grafikus környezetben (Kate, Gedit, vagy valamelyik C++, Java stb. fejlesztő környezet szövegszerkesztője).

A programok követésére léteznek hibakereső alkalmazások, mint a **bashdb** a Bash esetében (http://bashdb.sourceforge.net) vagy a **kshdb** (https://github.com/rocky/kshdb). Használatuknak akkor van értelme, ha programjainkat már nem tudjuk egyszerű, magába a héjba épített opciókkal megoldani. A Bash esetében lásd a **bashdb** kézikönyvét: *Debugging with the BASH debugger* (Bashdb [184]).

Kis programok esetében egyszerű eszközökkel is megoldhatjuk a hibakeresést.

Változók kiírása

A legegyszerűbb eszköz a változók kiírása az **echo** paranccsal, persze ez maga után vonja a program szövegének folyamatos szerkesztését, a már nem szükséges kiírások kommentelését illetve újak beszúrását:

```
#!/bin/bash

a=2;

echo a="$a"

b=$( expr $a + 1 )

echo b="$b"
```

Ez zavaró hosszabb programok, sok változó esetében, ennek ellenére használjuk.

A héj opcióinak használata és a set parancs

Mint minden Unix parancs, a héj is elindítható különböző opciókkal, amelyek befolyásolják futását. Ezek megadhatóak a parancssoron, de futás közben is beállíthatóak vagy átkapcsolhatók.

A Bash futási módját befolyásolhatják a környezeti változók és a parancssorán megadott opciók, amelyeknek száma igen nagy. Ezeket futás közben a **set** és **shopt** parancsokkal tudjuk módosítani.

A **set** régebbi, a **shopt** a Bash 2.0 után bevezetett beépített parancs a héj működésének befolyásolására. Mindkettő komplikált, részletes tanulmányozáshoz lásd a Bash kézikönyvének 4. fejezetét (BashRef2010 [184]).

A **set** parancs ki és bekapcsolja a különböző futási opciókat. Használata:

`set` -o opciónév

`set` +o opciónév

A szintaxis furcsa: a `-o` változat bekapcsolja, a `+o` kikapcsolja a megnevezett opciót. Pl. `noglob` opció beállítása letiltja a `*` karakter átírását a parancssoron:

```
$ ls
1.txt  2.txt  4.txt  5.txt
$ echo *
1.txt 2.txt 4.txt 5.txt
$ set -o noglob
$ echo *
*
$ set +o noglob
$ echo *
1.txt 2.txt 4.txt 5.txt
$
```

Megjegyzés: az opció beállítására általában használható egy karakteres megoldás is, az előbbi `set -o noglob` -ra például a `set -f` , a kikapcsolására a `set +f` .

A **shopt** parancs kiterjeszti ezeket a beállításokat, használatát lásd az említett kézikönyv [184] fejezetben.

A hibakövetésre 3 opciója használható a héjnak, ezeket megadhatjuk parancssoron indításkor, illetve a set paranccsal futás közben.

A.1. táblázat - A Bash hibakövetésnél használható opciói

A set parancsban használt neve	Parancssori opció	A héj viselkedése
`noexec`	`-n`	A héj nem fogja futtatni a bemenetére küldött programot, csak szintaxis ellenőrzést végez.
`verbose`	`-v`	A héj futás közben ki fogja írni az éppen futtatott parancsot indítás előtt.
`xtrace`	`-x`	A héj ki fogja írni a behelyettesített futtatott parancsot miután feldolgozta a parancssort. Így a parancssor feldolgozásának lépései is láthatóak lesznek.

Tekintsük az alábbi kis szkriptet (`debug1.sh`):

```
#!/bin/bash

d=${1:? nincs könyvtár}

sum=0

#minden fájlt végigjárva
for f in $( ls "$d" )
do
    #hány sor van benne
    n=$( cat "$d/$f" | wc -l )
    #összeg
    sum=$( expr $n + $sum)
done

#összeget egy fájlba
echo $sum > sum.txt
```

Az első két opció követése egyszerű. A `noexec` csak szintaxis ellenőrzést végez, ha nincs hiba:

```
$ bash -n debug1.sh
$
```

, ha beszúrunk egy hibát:

```
$ bash -n debug1.sh
debug1.sh: line 11: unexpected EOF while looking for matching `)'
debug1.sh: line 19: syntax error: unexpected end of file
$
```

Az alábbi, két fájlt tartalmazó könyvtáron fogjuk tesztelni a `-v` és `-x` opciókat:

```
$ ls -l teszt
total 8
-rw-rw-r-- 1 lszabo lszabo 8 Sep 22 19:31 1.txt
-rw-rw-r-- 1 lszabo lszabo 7 Sep 22 19:31 2.txt
$
```

Ha a `-v` opciót használjuk:

```
$ bash -v debug1.sh teszt
#!/bin/bash

d=${1:? nincs könyvtár}

sum=0

#minden fájlt végigjárva
for f in $( ls "$d" )
do
    #hány sor van benne
    n=$( cat "$d/$f" | wc -l )
```

```
    #összeg
    sum=$( expr $n + $sum)
done
 ls "$d"
 cat "$d/$f" | wc -l
 expr $n + $sum
 cat "$d/$f" | wc -l
 expr $n + $sum

#összeget egy fájlba
echo $sum > sum.txt

$
```

Tehát látható minden egyes sor végrehajtás előtt. Így ha pl. a második ciklusban akadna ki a program, az jól lokalizálható lesz.

Az `xtrace` opció pontos listát ad a változók alakulásáról:

```
$ bash -x debug1.sh teszt
+ d=teszt
+ sum=0
++ ls teszt
+ for f in '$( ls "$d" )'
++ cat teszt/1.txt
++ wc -l
+ n=4
++ expr 4 + 0
+ sum=4
+ for f in '$( ls "$d" )'
++ cat teszt/2.txt
++ wc -l
+ n=4
++ expr 4 + 4
+ sum=8
+ echo 8
$
```

Az `xtrace` minden sort egy + jellel kezd, ez tulajdonképpen a parancssor értelmezési szintjét jelöli. A `for f in $( ls "$d" )` sor végrehajtása előtt el kell végeznie a zárójelben álló `ls "$d"` kiterjesztését, így ez második szintnek felel meg, ezért a kiterjesztett ls teszt sor két + jellel jelenik meg a `for f in ...` sor előtt.

A két opciót (`-v` és `-x`) együtt is használhatjuk. Ugyanakkor megtehetjük azt, hogy az opciókat csak futás közben állítjuk be, és csak ott, ahol ténylegesen látni akarjuk mi történik. A forrás így módosul, ha csak a `for` ciklusban történtek érdekelnek:

```
#!/bin/bash

d=${1:? nincs könyvtár}

sum=0
```

```
#minden fájlt végigjárva
for f in $( ls "$d" )
do
    set -o xtrace
    #hány sor van benne
    n=$( cat "$d/$f" | wc -l )
    #összeg
    sum=$( expr $n + $sum)
    set +o xtrace
done

#összeget egy fájlba
echo $sum > sum.txt
```

a futás pedig:

```
$ bash  debug1.sh teszt
++ cat teszt/1.txt
++ wc -l
+ n=4
++ expr 4 + 0
+ sum=4
+ set +o xtrace
++ cat teszt/2.txt
++ wc -l
+ n=4
++ expr 4 + 4
+ sum=8
+ set +o xtrace
$
```

Amint már említettük, használhatunk rövid szintaxist is, `set -o xtrace` helyett `set -x` , a `set +o xtrace` helyett `set +x` .

A kiírt + jel tulajdonképpen a héj beépített `PS4` nevű változója:

```
$ echo $PS4
+
$
```

Ha úgy gondoljuk, hogy helyette írhatunk ki valami értelmesebbet, ami segít a hibakeresésben, azt megtehetjük a `PS4` tartalmának megváltoztatásával. Egyik gyakran használt megoldás erre a héj `LINENO` változója, amely mindig a végrehajtott bemeneti fájl sorszámát tartalmazza. Ezzel a változtatással az előbbi futtatás így néz ki (a **bash** hívása előtt a környezeti változok közé írjuk a megváltoztatott `PS4`-et az **export** paranccsal):

```
$ export PS4='sor:$LINENO '
$
$ bash  debug1.sh teszt
ssor:12 cat teszt/1.txt
ssor:12 wc -l
```

```
sor:12 n=4
ssor:14 expr 4 + 0
sor:14 sum=4
sor:15 set +o xtrace
ssor:12 cat teszt/2.txt
ssor:12 wc -l
sor:12 n=4
ssor:14 expr 4 + 4
sor:14 sum=8
sor:15 set +o xtrace
$
```

Ha gyakran akarjuk ezt a működési módot választani, a `PS4` átírásának sorát beírhatjuk a `.bash_profile` konfigurációs állományba, és akkor az minden bejelentkezésnél átállítódik.

Jelzések használata programkövetésre

Az említett egyszerű eszközökön kívül programkövetése használhatunk jelzéseket is. Ezeket a `DEBUG`, `EXIT`, `ERR` nevű jelzéseket és azok használatát a Jelzések című fejezet, "Speciális, programkövetésre használt jelzések [164]" alfejezetében mutatjuk be.

B. függelék - A getopts függvény

A shell `getopts` függvénye a parancssori argumentumok átvételét oldja meg, az alábbi módon használjuk:

```
getopts  opciósztring  változó
```

Az opció sztringben felsoroljuk a parancssoron megjelenő összes lehetséges opciót. Azoknál a opcióknál amelyek argumentumot is várnak, egy kettőspont jelet is teszünk az opció betűjele után, például ha az opció `-a` , akkor `a:` -ot írunk.

A függvényt többször hívjuk (legegyszerűbb ezt ciklusból végezni), végighalad a parancssoron és igaz értékkel tér vissza amíg talál a parancssoron opciót. Az argumentumában megadott változót beállítja arra az opció névre amit megtalált. Ezen kívül még két változót állít automatikusan:

- `OPTIND` - annak a parancssori paraméternek a száma ameddig eljutott
- `OPTARG` - az opcióhoz rendelt argumentumot tartalmazza

Ha az opciósztring első karaktere a kettőspont, a `getopts` nem ír ki hibákat, és ismeretlen opció esetén `?`-et tesz a visszatérített változóba. Ha nincs kettőspont az opció sztring elején, hibákat ír ki (hibás kapcsoló, elmaradt paraméter). Elmaradt paraméter esetén nem az opciót adja vissza a változóban, hanem a `?` karaktert.

Ugyanakkor figyeli a shell `OPTERR` nevű változóját (ennek implicit értéke 1) és ha ez 0, akkor semmiképpen sem fog hibákat kiírni (még akkor sem, ha az opció sztring elején nincs kettőspont).

Tipikus használata az alábbi `while-case` kombináció:

```
#!/bin/bash
#példa program a getopts függvény használatára
#a getopts opció sztringjének első karaktere a :
#ez azt jelenti, hogy nem fog hibákat kiírni
#ezt ki lehet törölni, akkor a getopts hibákat ír ki
#
#az alábbi hívás a következő opciókat várja:
#   -a opció paraméterrel
#   -b opció paraméterrel
#   -c és -d opciók paraméter nélkül
#
#a következő megtalált opciót a opcio változóban
#téríti vissza

#ha az OPTERR változót 0-ra állítjuk, semmiképp nem ír ki hibát
# OPTERR=0
#ez ebben a példában felesleges, mert van : az opció sztring
#elején
```

```
#az első : -t ki lehet törölni, akkor a getopts kiírja a hibákat
while getopts ":a:b:cd" opcio
do
    case $opcio in
        a) echo a kapcsoló, argumentum: $OPTARG ;;
        b) echo b kapcsoló, argumentum: $OPTARG ;;
        c) echo c kapcsoló;;
        d) echo d kapcsoló;;
        ?) echo ismeretlen kapcsoló: $OPTARG
           echo használat: getopts.sh \
               [ -a N ] [ -b N ] [ -c ] [ -d ] paraméterek ...
           exit 1
           ;;
    esac
done

echo vége a getopts hívásoknak, az OPTIND index értéke = $OPTIND

#el kell mozgatni a parancssort balra $OPTIND-1 pozícióval
#ha még vannak paraméterek, amelyek nem opciók
#azok a $1, $2 stb.-be kerülnek

shift $(($OPTIND-1))

if [ -n "$1" ];then
    echo "$1" a megmaradt első parameter
else
    echo nincsenek további paraméterek
fi
```

Bibliográfia

[Manhu] ***, *Magyar Linux Dokumentációs Projekt, Kézikönyv oldalak [http://tldp.fsf.hu/man.html]*, (nincs karbantartva)

[UNIX1970] ***, *The Creation of the UNIX* Operating System [http://www.bell-labs.com/history/unix/]*, 2002

[Albing2007] Carl Albing and JP Vossen and Cameron Newham, *bash Cookbook*, O'Reilly, 2007.

[Buki2002] Büki András, *UNIX/Linux héjprogramozás*, Kiskapu, 2002.

[Bashdb] Rocky Bernstein, *Debugging with the BASH debugger [http://bashdb.sourceforge.net/bashdb.html]*, 2010

[Mendel2011] Mendel Cooper, *Advanced Bash-Scripting Guide [http://tldp.org/LDP/abs/html/]*, 2011

[Friedl2004] Jeffrey Friedl, *Reguláris kifejezések mesterfokon*, Kossuth Kiadó, 2004.

[Ivanyi2005] Péter Iványi, *A GAWK felhasználói kézikönyve magyarul [http://hexahedron.hu/personal/peteri/gawk/index.html]*, 1998-2005

[Michael2003] Randal K. Michael, *Mastering UNIX shell scripting*, Wiley, 2003.

[BashHogyan] Mike Mikkey, *Bevezetés a bash programozásba HOGYAN [http://tldp.fsf.hu/HOWTO/Bash-Prog-Intro-HOWTO-hu/Bash-Prog-Intro-HOWTO-hu.html]*, 2000

[Newham2005] Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, O'Reilly, 2005.

[BashRef2010] Chet Ramey, *Bash Reference Manual [http://www.gnu.org/software/bash/manual/html_node/index.html]*, 2009

[Readline] Chet Ramey, *The GNU Readline Library [http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html]*, 2009

[Robbins2005] Arnold Robbins and Nelson Beebe, *Classic Shell Scripting*, O'Reilly, 2005.

[Rosenblatt2002] Bill Rosenblatt and Arnold Robbins, *Learning the Korn Shell*, O'Reilly, 2002.

Tárgymutató

U

W