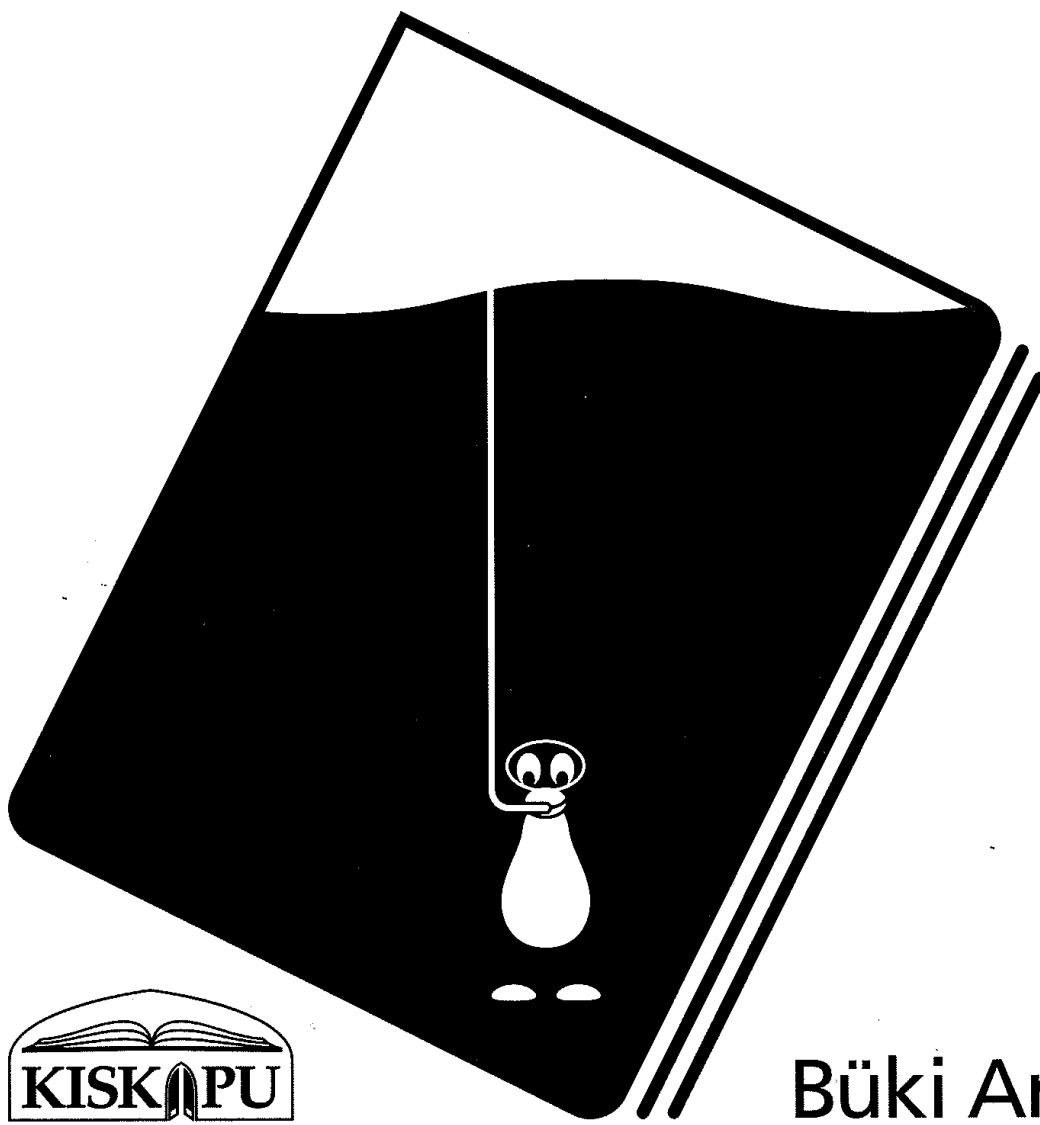


UNIX / Linux

héjprogramozás



Büki András

Tartalomjegyzék

1. fejezet Alapelemek

Helló világ: első héjprogramunk	3
Változók használata	5
Idézőjelek használata, parancsbehelyettesítés	8
Parancssori paraméterek	8
A szabványos be- és kimenet és ezek átirányítása	9
Csövek	11
A csatornák számozása és a hibacsatorna átirányítása	13
Matematikai kifejezések kiértékelése : az expr parancs	15
Parancsvégrehajtás : az eval parancs	18
A beágyazott dokumentum (here document)	18

2. fejezet Programvezérlési szerkezetek

Feltételes utasítás : if, test, && és 	23
Többszörös elágaztatás: a case szerkezet	26
Ciklusszervezés : for, while és until	28
Függvények	33

3. fejezet Keresés, szűrés, szövegfeldolgozás, avagy a szabályos kifejezések lélektana

Mire valók a szabályos kifejezések?	39
A szabályos kifejezések alapelemei	42
„c” (tetszőleges karakter)	42
\c	42
^	43
\$	43
.	43
[karakterek]	43
[^...]	44
[c1-c2]	44
Jelentésmódosító jelek	44
*	44
+	44
(...)	45
{x} vagy {x,y} vagy {x,y}	45
szabályos kifejezés 1. szabályos kifejezés 2.	46

A szabályos kifejezések és a héj	47
A szabályos kifejezések és a mosogép	49
Avagy hogyan írunk szabályos kifejezéseket tartalmazó programot	49

4. fejezet A sed használata héjprogramokban

A sed működési elve	55
A sed alapvető parancsai	57
p (Print)	57
d (Delete)	58
s (Substitute)	58
a (Append), i (Insert) és c (Change)	62
y (Transform)	63
w (Write), r (Read)	64
n (Next), q (Quit)	65
Tizedespont, tizedesvessző	65
Készítünk telefonkönyvet	68

5. fejezet Az AWK használata héjprogramokban

Az awk működésének alapelvei	77
A programok szerkezete és indítása	77
Mezők	78
Kiíratás (print)	79
Felhasználó által megadott változók	80
Kapcsolattartás a héjprogram és az awk program között	80
Belső változók	81
A mezőelválasztó karakterek kezelése	82
Az AWK nyelvi elemei	83
A BEGIN és END blokkok	83
Matematikai műveletek	84
Feltételes utasítás	86
Beépített függvények	87
Ciklusok	87
Fájlok kezelése	88
Összetett gyakorlatok	89
Telefonkönyv	89
Kettes számrendszer	90
Átlagolás	92
Névsor	94
Címlista és telefonkönyv összefésülése	95
Egy sor, több sor...	96

6. fejezet A héjprogramok alapvető építőelemei

A parancssori paraméterek és kapcsolók kezelése	101
Parancssori paraméterek hiányának felismerése	101
Kapcsolók felismerése és fájl létezésének vizsgálata	102
Kapcsolók egybeírása	103
A getopt parancs	106
Fájl típusának vizsgálata	107
A szabványos bemenet olvasása: szűrőként működő héjprogram	108
Írás a képernyőre, olvasás a billentyűzetről	110
Várakozás billentyű leütésére	113
Átmeneti fájlok kezelése	113
Jelek elfogása és kezelése	117
Zárolás	121
Időzített végrehajtás, várakozás	122
Önhívó parancsvégrehajtás teljes könyvtárszerkezetben	125
Az xargs parancs	127

7. fejezet Gyakorlatok I.

Egyszerű feladatok	133
Változatok egy témára: az angol ábécé betűi	133
Feladat	133
Első megoldás	134
Ötletek	134
A megoldás	135
Második megoldás	137
Ötletek	137
A megoldás	137
Harmadik megoldás	138
Ötletek	138
A megoldás	139
Negyedik megoldás	139
Ötletek	139
A megoldás	140
A UNIX segédprogramok „magyartudásának” ellenőrzése	141
Feladat	141
Ötletek	141
A megoldás	141
Tac	142
Feladat	142

Ötletek	143
A megoldás	143
Rev	145
Feladat	145
Ötletek	145
A megoldás	145
Titkosítás	148
Feladat	148
Ötletek	148
A megoldás	148
Betűk megszámlálása	150
Feladat	150
Ötletek	150
A megoldás	151
Digitális számok	153
Feladat	153
Ötletek	153
A megoldás	154
Szavak keresése	156
Feladat	156
Ötletek	156
A megoldás	156
Számábrázolási pontosság	159
Feladat	159
Ötletek	159
A megoldás	160
Csomagoljunk	160
Feladat	160
Ötletek	161
A megoldás	161

8. fejezet Gyakorlatok II.

Segédprogramok	165
Igen vagy Nem?	165
Feladat	165
Ötletek	165
A megoldás	165
Egyszerű menürendszer	167
Feladat	167
Ötletek	168

A megoldás	168
Interaktív parancsértelmező	170
Feladat	170
Ötletek	170
A megoldás	171
Számból szöveg	174
Feladat	174
Ötletek	174
A megoldás	175
Telefonköltség kiszámítása	180
Feladat	180
Ötletek	180
A megoldás	181
Felhasználók bejelentkezésének figyelése	187
Feladat	187
Ötletek	187
A megoldás	187
Felhasználók lemezfoglalásának figyelése	190
Feladat	190
Ötletek	191
A megoldás	191

9. fejezet Tippek, trükkök

Kiegészítés Bash-felhasználóknak	201
Változó „nem meghatározottá” tétele	201
Parancsbe helyettesítések egymásba ágyazása	203
A let parancs	204
Közvetett változóhasználat	205
Az üres parancs	206
C stílusú megoldások	207
Gyermekhéj és névtelen függvény használata	209
Tömbök használata	211

Függelékek

I. Függelék	215
II. Függelék	223
Tárgymutató	229

Bevezető

Köszönet...

Itt szeretném köszönetemet kifejezni mindenazonknak, akiknek a UNIX operációs rendszer használatát valaha is tanítottam. Nagyon sokat tanultam tőlük és őszintén remélem, hogy ez a könyvön is meglátszik.

Köszönöm továbbá Prof. H. V. Kuksinak a könyv megírása során tanúsított türelmét, és a felmerült gondok feletti bölcs hallgatást.

Kikhez és miről szól ez a könyv?

Ezt a könyvet elsősorban azok a UNIX operációs rendszerrel még csak ismerkedő számítógép-használók forgathatják haszonnal, akik a UNIX okozta „első traumán” már túlvannak. Értem ez alatt a parancsok néha kissé fura formáját, a más operációs rendszerektől eltérő gondolatvilágot, no meg a sokat emlegetett szűkszavúságot.

Ez a könyv tehát bevezető jellegű, de nem azokhoz szól, akik még semmit nem tudnak a UNIX operációs rendszerről. Ennek megfelelően nem, vagy csak érintőlegesen tárgyalja például a UNIX alapfogalmait. Kivételt képeznek azok a téma-körök, amelyek valamilyen módon szervesen kapcsolódnak a héjprogramozás elméletéhez vagy gyakorlatához.

Ahhoz tehát, hogy az itt leírtakat megértse, az Olvasónak már eleve rendelkeznie kell számos alapismerettel. Legalább érintőlegesen ismernie kell például a jogosultságok rendszerét, bizonyos alapvető UNIX parancsokat, alapszinten értenie kell, hogy mi az a szabványos be- és kimenet, és természetesen hatékonyan kell tudnia használni legalább egy szövegszerkesztőt.

Összefoglalva tehát ez a könyv azoknak a középhaladóknak vagy haladóknak íródott, akik a tanulás folyamatában eljutottak arra a pontra, ahonnan a UNIX rendszert már szeretnék saját munkájukkal kapcsolatos tényleges feladatok megoldására is használni.

Shell, burok, héj...

Nyilván senkinek nem mondok újat, ha azt állítom, hogy napjainkban a számítástechnika rohamléptekben fejlődik. Ezt az ütemet a magyar nyelv fejlődése nem mindig képes követni, így állandóan akadnak olyan szakkifejezések, amelyeket értünk, használunk, de valahogy nincs igazán jól hangzó magyar megfelelőjük.

A UNIX ugyan nem nevezhető fiatal operációs rendszernek, a vele kapcsolatos magyar terminológia azonban a mai napig hiányos. Olyannyira az, hogy ennek a könyvnek már a címét is nehéz volt magyarul leírni. Az első ötlet természetesen a „Shell-programozás” volt, hiszen a UNIX parancsértelmezőjét a legtöbb szakember „shellnek” titulálja. Nyilván az Olvasó is érzi, hogy ez a szó nincs igazán összhangban a magyar nyelvvel. Sokan használják a „burok”, illetve a „héj” megnevezést is. Ezek már valódi magyar szavak, így értelemszerűen sokkal jobban belesimulnak a magyar szövegkörnyezetbe. Hosszas töprengés és vita után a Kiadóval végül a „héj”, illetve „héjprogramozás” kifejezések használata mellett döntöttünk. Tettük ezt a magyar számítástechnikai szaknyelv fejlesztéséért, abban a reményben, hogy sikerül hagyományt teremtenünk.

Miért tanuljuk meg a héjprogramozást?

Nyilván számos olyan olvasója lesz ennek a könyvnek, aki a DOS-on vagy esetleg valamelyik Windows operációs rendszeren „nevelkedett”. Feltehetőleg sem nekik, sem a számítástechnikával hosszabb ideje foglalkozóknak nem kell magyarázni, mennyire hasznos lehet bizonyos helyzetekben egy olyan programozási nyelv, amellyel könnyen és gyorsan szervezhetjük rendszerre az operációs rendszer nyújtotta szolgáltatásokat és segédprogramokat.

Különösen igaz ez a UNIX operációs rendszerekre, ezeknek ugyanis már a gondolkodásmódja is azt feltételezi, hogy a felhasználó egyrészt tud programozni, másrészt a rendelkezésére bocsátott segédprogramokból maga fogja összeállítani a saját adataihoz illeszkedő feldolgozási sorokat. Míg tehát az „egyéb” operációs rendszerek általában adott típusú feladatokra adnak kész megoldásokat, a UNIX-ok inkább egy szerszámoslására hasonlítanak.

A UNIX segédprogramok első látásra „butának”, vagy jobb esetben „szakbarbárnak” tűnnek, mivel mindegyik csak egy bizonyos dolgot tud megvalósítani. Azt viszont nagyon! Van például egy unióq nevű segédprogram, amely képes egy szövegből kiszűri a többször előforduló sorokat, de csak akkor, ha azok betűrend szerint következnek egymás után. Ha ez a feltétel az általunk feldolgozni kívánt szöveget nem

teljesül, a uniq „tehetetlen”, sőt hibásan működik. Van azonban egy sort nevű parancs is, ami éppen a sorba rendezésben „profi”. Termézetesen hasonlóan előző testvérehez gyakorlatilag semmi egyébre nem képes. Mármost ha van egy megoldás, amivel ezt a kettőt „sorba tudjuk kötni”, akkor akármilyen szöveget feldolgozhatunk velük. A UNIX gondolkodásmódjának éppen ez a lényege: a „soros kapcsolás”.

A UNIX-hoz a felhasználó rengeteg különleges tudású segédprogramot kap, valamint az úgynevezett csövek (pipes) formájában egy olyan rendszerszolgáltatást, amely lehetővé teszi, hogy ezekből szinte tetszőlegesen bonyolult feldolgozási hálózatot alkossan saját igényeinek megfelelően. Tulajdonképpen e hálózatok felépítését nevezzük összefoglaló néven héjprogramozásnak.

Az alcímben feltett kérdésre tehát röviden a következő válasz adható: a héjprogramozást azért érdemes (sőt kell!) megtanulnunk, mert a UNIX operációs rendszer képességeit igazán csak ezen ismeret birtokában tudjuk majd kihasználni.

Hogyan születnek a héjprogramok?

Csőhálózatot termézetesen a parancssorban is lehet építeni, de ha ehhez több sort kell begépelni, a dolog már kényelmetlenevé válik. Ilyenkor nyilván célszerűbb a megfelelő parancssorokat egy szövegfájlba írni, majd „rávenni” a parancsértelemzőt arra, hogy olvassa el azt, és hajtsa végre a benne leírtakat. Az ilyen „recepteket” nevezzük összefoglaló néven héjprogramoknak.

Mivel a héjprogramok tulajdonképpen közönséges szövegfájlok, tetszőleges szövegszerkesztővel létrehozhatjuk őket. „Elméleti síkon” a UNIX héjprogramok tulajdonképpen a DOS-os kötegfájlok (batch programok) megfelelői, bár segítségükkel nagyságrendekkel összetettebb feladatokat oldhatunk meg. (Ez az összehasonlítás ebben a sorrendben termézetesen nem igazán helyénvaló, hiszen a DOS és a UNIX közül az utóbbi volt előbb...)

Mikor érdemes héjprogramot írni, és mikor nem?

Programot általában olyan feladatra érdemes írni, amit várhatóan sokszor kell majd végrehajtanunk. Előfordulhat termézetesen az is, hogy egész életünk során csak egyszer lesz valamire szükségünk, de a végrehajtandó műveletsor túlságosan összetett ahhoz, hogy egyszerűen a parancssorba begépeljük.

Az, hogy milyen feladatra írunk héjprogramot, és minek a megoldásához használunk inkább valamilyen „hagyományos” programnyelvet (például a C-t), első közelítésben „ízlés dolga”. Ugyanakkor a magasszintű programnyelvek alapvetően nem támogatják a műveleti szempontból egyszerű, de gépi szinten összetett műveletek kényelmes végrehajtását. Egy ilyen nyelvben magunknak kell gondoskodnunk a fájlok megnyitásáról és lezárásáról, a tárfoglalásról, a jogosultságok beállításáról és számos egyéb dologról akkor is, ha csupán az a célunk, hogy egy szövegfájlban kicséréljünk egy megadott szót egy másikra. Ez rengeteg többletmunkát és számos hibalehetőséget rejt magában. Ráadásul a UNIX valóban lenyűgöző eszközökkel bocsát a rendelkezésünkre az ilyen „apró” feladatok megoldására. Ezeket a programokat teljesen értelmetlen lenne újra megírni.

Vannak ugyanakkor olyan feladatok is, amelyeket kifejezetten célszerűtlen héjprogram segítségével megoldani. E kategória jellegzetes képviselői a nagy számításigényű eljárások (például differenciálegyenletek numerikus megoldása), illetve a kifejezetten bonyolult algoritmusok. Vannak a terjedelemmel kapcsolatos „józan megfontolások” is. Egy héjprogram gyakorlatilag akármilyen hosszú lehet, viszont ha valamit nem tudunk néhány száz sorban megoldani, azt feltehetőleg nem is érdemes ezzel a módszerrel.

A könyv vázlatos felépítése és jelölései

Ebben a könyvben elsősorban a Bourne héj, illetve a vele rokon parancsértelmezők programozásáról lesz szó. Ennek legfőbb oka, hogy a másik nagy csoportot alkotó „C típusú” héjak programozása utasításformájában és néhol logikájában is eltérő. Egyrészt nem akartam az Olvasót azzal terhelni, hogy egyszerre két formát mutatok be minden egyes héjprogramból, másrészt a gyakorlat azt mutatja, hogy a UNIX világában vidáman elboldogulhatunk anélkül, hogy egyszer is rákényszerülnénk a csh vagy tcsh használatára.

A könyv első felében csaknem lexikonszerűen ismertetem a Bourne típusú héjak által nyújtott lehetőségeket, a héjprogramokban használható változókat, parancsokat és szerkezeteket. Előre kell bocsátanom, hogy ez meglehetősen száraz adattömeg lesz. Hosszú leírásokat és egészen egyszerű példaprogramokat fog tartalmazni, amelyeknek a való élethez legtöbbször vajmi kevés köze lesz.

Ez a szakasz természetesen nem is azt a célt szolgálja, hogy az Olvasó ebből tanulja meg a héjprogramozás rejtelmait. Sokkal inkább referenciának szántam, amelyből először megtudhatjuk, hogy mit nem tudunk. Később valószínűleg majd sokszor kell az Olvasónak ide visszalapoznia, hogy kiderítse azon parancsok helyes formáját, amelyek a gyakorlás során valamiért csak nem akarnak működni.

A második – jóval hosszabb – részben életszerű feladatokat és azokra írt héjprogramokat fogok bemutatni, a szükséges helyeken soronként elmagyarázva, pontosan mi is történik. Mindkét részben kiemeltem a kezdő (vagy talán nem is annyira kezdő) felhasználóra leselkedő veszélyeket, buktatókat, illetve a „trükkös megoldások” lehetőségeit. Ezekre a szakaszokra a megfelelő piktogramok utalnak.

Buktató

Az ilyen szövegrészkek általában olyan – helyenként egyáltalán nem nyilvánvaló – hibalehetőségekre hívják fel a figyelmet, amelyekbe a legtöbb felhasználó „beleszalad”.

Megjegyzés

Néha az ilyen szövegrészeken is lehetséges hibákról esik szó, de ezeken a helyeken inkább a témahez kapcsolódó kiegészítő adatokat fogunk találni.

TIPP

Az így kiemelt bekezdésekben a névnek megfelelően a szokványostól kissé eltérő megoldásokat mutatok be. Fontos hangsúlyozni, hogy ezek nem olyan „elvetemült trükkök”, amelyek egy-egy rendszer sajátosságait használják ki, hanem logikus rövidítési vagy egyszerűsítési lehetőségek, amelyekkel a gyakorlottabb felhasználó áttekinthetőbbé teheti programjait.



Elvétve ugyan, de van néhány olyan hibalehetőség, amelyekkel az óvatlan felhasználó kárt tehet az adataiban. (Esetleg a működésképtelen programja fölötti gondolkodás közben önmagában.) Ezeket a mérgező anyagok csomagolásán használatos piktogrammal, szürke háttér előtt jelezük. Feltétlenül olvassuk el az ilyen megjegyzéseket!

A könyv részletes felépítése

Az első fejezetben a héjprogramozás legalapvetőbb elemeit, illetve a UNIX alapvető szolgáltatásait vesszük sorra. Szó esik a héjprogramok futtatásáról, a változók és parancssori paraméterek használatáról, a csatornák átirányításáról és a csövekről, a héj – meglehetősen korlátozott – matematikai képességeiről, valamint a beágyazott dokumentumokról.

A második fejezet az alapvető programszervezési elemeket mutatja be. Itt tudhatjuk meg, hogyan kell ciklust szervezni, logikai kifejezéseket kiértékelni, kétszeres és többszörös elágazásokat megvalósítani, valamint függvényeket írni és használni.

És ezután következik a dolgok „sűrűje”... A harmadik fejezet tulajdonképpen a következő kettő előkészítése. A szabályos kifejezések alapelemeit és használatát mutatja be lépésről lépésre haladva. Már itt szeretném előrebocsátani, hogy kezdők számára ez meglehetősen nehéz téma. Aki viszont „túlélte” ezt és a következő két fejezetet, az az elkövetkező akadályokat általában már könnyen veszi.

A negyedik fejezet a sed, az ötödik az awk használatát tárgyalja. Ez a két segédeszköz egyrészt bonyolultsága, másrészt sokrétű felhasználhatósága miatt kapott ekkorra szeletet a teljes anyagból.

Minden programozási nyelvnek és környezetnek vannak bizonyos általános építőelemei, amelyeket – néha öntudatlanul – újra és újra felhasználunk. A héjprogramokban a legtöbbször azonos módon kezeljük például a parancssori paramétereket és kapcsolókat, azonos módon hozzuk létre az átmeneti fájlokat és azonos, vagy legalábbis nagyon hasonló módon valósítunk meg önhívó (rekurzív) algoritmusokat. A hatodik fejezetben ezekről az „újrahasznosítható” elemekről lesz szó.

A hetedik és nyolcadik fejezetben már életszerű feladatokat oldunk meg. Először csak „elméleti próbálkozásokat” hajtunk végre, de a végén írunk néhány egészen komoly programot is. Itt célszerű, ha az Olvasó először önállóan próbálkozik meg a feladatok megoldásával, és a könyvbeli kódot csak ez után nézi meg. (Kicsit lesni természetesen szabad, ha megakadtunk valahol.) Az önálló munkát e két fejezet szerkezete is elősegíti, mivel minden szakasz tartalmaz a feladat pontos megfogalmazása és megoldása között egy „Ötletek” című részt is. Ezt átgondolva az Olvasónak elvileg képesnek kell lennie arra, hogy megszerzett ismeretei alapján egyedül írja meg a programot.

A könyv megírása során végig törekedtem arra, hogy kerüljem a rendszerfüggő megvalósításokat. A bemutatott példákknak és megoldásoknak elvileg bármely UNIX rendszeren működniük kell. Szükségesnek tartottam azonban azt is, hogy zárásképpen bemutassak néhány olyan – többségében modern – megoldást, programszerkezetet, amely nem feltétlenül hordozható. Ezekről és a szinte bármely rendszeren futtatható Bash héjról szól a kilencedik fejezet.

Az I. függelék a héjprogramokban leggyakrabban használt UNIX parancsok betűrendes felsorolását és rövid leírását tartalmazza. Bővebb (használhatóbb) leíráshoz ezekről természetesen csak a megfelelő dokumentáció átböngészésével juthatunk, néha viszont az is nagy segítség, ha tudjuk, mit kell keresnünk.

A II. függeléknek a hibakeresés során vehetjük hasznát. Hibajelenségek felsorolását és azok leggyakoribb okait tartalmazza. Ha az Olvasó a gyakorlás során úgy érzi, hogy „végleg” elakadt egy programhiba miatt, ez legyen az első hely, ahol a megoldást keresni kezdi.

A könyv számos példaprogramot tartalmaz. Ezek némelyike egészen rövid, akadnak azonban kifejezetten hosszúak is. Programozást tanulni leghatékonyabban úgy lehet, ha egy számítógép előtt ülünk és rögtön ki is próbáljuk az olvasottakat. Az Olvasó munkáját megkönnyítendő a könyvben szereplő és számmal rendelkező valamennyi programlista letölthető a <http://www.kiskapu.hu/konyvek> címről.

Budapest, 2002. május
Büki András

1

Alapelemek

Helló világ: első héjprogramunk

A héjprogramok tulajdonképpen közönséges szövegfájlok, így tetszőleges szövegszerkesztővel megírhatjuk őket. A program első közelítésben ugyanazokat a parancsokat tartalmazhatja, amelyeket a parancssorból is kiadhatnánk. Ezekhez társul még néhány hamarosan ismertetendő vezérlési szerkezet, valamint egyéb szolgáltatás, amelyek a program szervezését teszik lehetővé.

A kész programot többféleképpen futtathatjuk. A legegyszerűbb – ám legritkábban használt – módszer az, hogy a parancsértelmező neve után parancssori paraméterként megadjuk a programot tartalmazó szövegfájl nevét. Itt érdemes kihangsúlyozni, hogy minden héjprogramot külön parancsértelmező hajt végre, vagyis a programmal együtt mindig elindul (legalább) egy új héj is. Ez a módszer tehát a következőképpen fest:

```
sh héjprogram_neve
```

A UNIX rendszerekben az sh mindenkor a Bourne héjat vagy annak valamilyen változatát jelenti. Előfordul az is, hogy az sh név tulajdonképpen egy másik, a Bourne héj szolgáltatásait is nyújtó parancsértelmezőre mutató hivatkozás (link).

Ez a módszer természetesen kissé kényelmetlen, ezért sokkal gyakoribb, hogy magában a héjprogramban adjuk meg a megfelelő parancsértelmező nevét. Ez az angolul „sha-bang”-nak nevezett jelölés a program első sorában szerepel és a következőképpen néz ki:

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

A „#!” karakterkettős tulajdonképpen egy „varázsszám” (magic number), ami azt jelzi a rendszernek, hogy héjprogramról van szó. Ha bármit futtatni próbálunk egy UNIX rendszeren, az első művelet mindenkor ennek a típusjelzésnek a kiolvasása. A rendszermag ez alapján dönti el, hogy pontosan mit is kell tennie az adott futtható állománnyal. (Részletesebben lásd a `magic` címszóhoz tartozó leírást!) A parancsértelmező nevét mindenkor teljes elérési úttal együtt kell megadnunk.

Ahhoz azonban, hogy a fenti megoldás működhessen, a héjprogramnak végrehajtási jogot mindenkor adnunk a `chmod` parancs segítségével. (Tulajdonképpen ez az egyetlen dolog, ami egy héjprogramot és egy közönséges szövegfájlt megkülönböztet.) Ettől kezdve héjprogramunkat ugyanúgy kezelhetjük, mint a rendszer bármelyik „hivatalos” összetevőjét, például az `ls` parancsot.

Ennyi tudással felvértezve meg is írhatjuk életünk első héjprogramját. Ha az Olvasó tanult már bármilyen nyelven programozni, valószínűleg nem fog meglepődni azon, hogy én is – mint annyi más szerző – a „Hello világ!” szöveg kiíratásával kezdem a példák sorát.

Nyissuk meg tehát kedvenc szövegszerkesztőnket, és gépeljük be a következő szöveget:

1.1. lista

```
1:  #/bin/sh
2:  # Ez egy megjegyzés
3:  echo "Hello világ!"
4:  exit 0
```

A könyvben szereplő valamennyi példaprogramban a fent látható módon beszámoltam a sorokat. Ez a számozás *nem* része a forráskódnak, csak azt a célt szolgálja, hogy a magyarázatok során könnyebben tudjak hivatkozni a program egyes helyeire. Röviden tehát: a számokat nem kell begépelni!

Az első sort most már értjük: itt áruljuk el a rendszernek, hogy ez egy héjprogram, és hogy a végrehajtásához az sh parancsértelmezőt kell elindítani, ami a /bin könyvtárban található.

Buktató

Ha elgépeljük az első sort, számos UNIX rendszer a "Command not found" („A parancs nem található”) üzenettel válaszol. Ilyenkor a kezdő felhasználók a legritkább esetben keresik a hiba okát éppen az első sorban. Ennek nyilván pszichológiai oka van: ők hibás *parancsot* keresnek, a hiba viszont a *parancsértelmező* megadásában van. A Bash fejlesztői már rájöttek ennek az üzenetnek a megtévesztő voltára, és kijavították a "bad interpreter" („Rossz parancsértelmező”) szöveget.

A második sor egy megjegyzést tartalmaz. A héj a # karakterből tudja, hogy az ebben a sorban levő szöveget nem kell értelmeznie. Megjegyzés nem csak a sor elején, hanem bárhol kezdődhet, akár egy parancssor után is. Ugyanakkor a # karakter után végrehajtandó parancs az adott sorban már nem szerepelhet.

A harmadik sor sem különösebben meglepő. Ha parancssorból kellene kiíratnunk ezt a szöveget, akkor is pontosan így járnánk el. A negyedik sor megadása tulajdonképpen nem kötelező. Ha elfogytak a végrehajtható sorok, a programnak ak-

kor is vége szakad, ha ezt nem jelezzük külön az exit parancsal. Az esetek túlnyomó többségében azonban célszerű ezt megtenni, mivel az exit segítségével adhatunk vissza egy (és csakis egy) *visszatérési értéket* a programot indító parancsértelmezőnek (szülőhéjnak). Ennek a hasznáról később még részletesen lesz szó.

Változók használata

Mint minden programnyelvben, a UNIX héjprogramokban is használhatunk változókat. Ezeket nem kell sehol külön bevezetni (deklarálni), abban a pillanatban, amikor először használjuk őket, a héj létrehozza őket. Az értékadás az egyenlőségjellel, mint operátorral történik, szintén sok más nyelvhez hasonlóan. A következő néhány példa egy-egy változót hoz létre, illetve ad nekik értéket:

```
szam=43  
szoveg1="Micimacko"  
szoveg2="Ez itt egy szöveges változó"
```

A fenti példa azt sugallja, hogy a héj változói (akkárcsak a többi nyelvben) tartalmuk szerint alapvetően kétfélék lehetnek: numerikusak (számok) vagy szövegesek (karakterláncok). Ez a különbségtétel azonban itt egyáltalán nem éles, sőt tulajdonképpen a héj valamennyi változója egyszerű karakterlánc. A programozás során így egy „numerikus” változót is kezelhetünk szövegként, és „szöveges” változóból is előállíthatunk számot. A változók „típusát” az adott környezet, illetve felhasználási mód dönti el.

Buktató

Ügyeljünk rá, hogy értékadásnál se az egyenlőségjel előtt, se mögötte ne legyen szóköz! A szam = 3 utasítás hatására a héj futtatni próbál egy „szam” nevű programot, és mivel nem találja, hibaüzenettel leáll.

Buktató

Szöveges tartalmú változók esetén értékadásnál mindenkorban idézőjelet kell használnunk, ha a karakterlánc szóközt vagy egyéb nem megjeleníthető karaktert is tartalmaz. Ennek oka éppen a héjváltozók önműködő volta. Ha idézőjelek nélkül egyszerűen leírunk egy szót, a héj azt változónak fogja tekinteni, aminek értéke számára nem meghatározott, ha eddig még semmilyen összefüggésben nem szerepelt.

A már megadott (definiált) változókra érték szerint úgy hivatkozhatunk, hogy közvetlenül nevük előtt egy „\$” jelet írunk. A fenti példánál maradva, ha ki akarjuk íratni a képernyőre a „szam” nevű változó értékét, a következőképpen tehetjük meg:

```
echo $szam
```

A \$ jel használata akkor is kötelező, ha egy már megadott változó tartalmát akarjuk értékül adni egy másiknak!

1.2. lista

```

1: #!/bin/sh
2: Hello=55
3: szoveg1=Hello
4: szoveg2="Hello világ!"
5: echo $Hello
6: echo $szoveg1
7: echo $szoveg2

```

A fenti példában a szoveg1 változó tartalma nem 55, hanem a „Hello” szöveg lesz. A héj ugyanis a \$ jel hiányában szövegnek hiszi a „Hello” szót, és ekként is használja azt. Láthatólag az sem zavarja ebben, hogy az előző sorban létrehoztunk egy ugyanilyen nevű változót és értéket is adtunk neki.

A héjnak vannak bizonyos belső, „közhasznú” változói is. Ilyenben tárolódik például a parancssori paraméterek száma vagy magának a programnak a neve. Ezek értékét a parancsértelmező magától beállítja, így valamennyi programban külön bevezetés (deklaráció) nélkül hozzáférhetők. A héj legfontosabb belső változóiról a 1.1. táblázat ad összefoglalót.

1.1. táblázat A héj névvel nem rendelkező belső változói

VÁLTOZÓ	JELENTÉS
\$#	A parancssori paraméterek száma
\$-	A héjprogramot végrehajtó héjnak átadott kapcsolók
\$?	A legutoljára végrehajtott parancs visszatérési értéke
\$\$	A futó program folyamataazonosítója
\$!	A háttérben utoljára végrehajtott parancs folyamataazonosítója
\$n	Az n-edik parancssori paraméter értéke (n értéke legfeljebb 9 lehet)
\$0	A pillanatnyi héj vagy héjprogram neve
\$*	Valamennyi parancssori paraméter egyben, egyetlen karakterlánc-ként ("\$1 \$2 ... \$9...")
\$@	Valamennyi parancssori paraméter egyben, egyenként idézőjelbe téve ("\$1" "\$2" ... "\$9...")

Buktató**Az „üres” és a „nem meghatározott” fogalma**

Ügyeljünk arra, hogy számos héj nem „nullázza” magától a változókat. Ha tehát egy olyan változóra igyekszünk érték szerint hivatkozni, amit még soha, semmilyen összefüggésben nem használtunk, vagy formálisan használtuk ugyan (leírtuk a nevét), de ténylegesen nem került bele semmi, akkor számos rendszeren titokzatos hibaüzenetet kaphattunk. (Az ilyen hibákat nagyobb programokban ráadásul igen körülmenyes felderíteni.) A héj számára tehát az üres és a nem meghatározott (definiálatlan) változó nem azonos fogalmak, annak ellenére, hogy a változók bevezetésére nincs szükség. Egy későbbi példaprogram kapcsán még visszatérek erre a kérdéskörre.

TIPP**A változók kifinomultabb használata**

Ha egy szövegben úgy kell egy héjváltozóra hivatkoznunk, hogy azt közvetlenül (szóköz nélkül) további szöveg követi, akkor a változó nevét zárójelbe tehetjük, jelezve ezzel a név határát:
`echo "$ (változónév) további szöveg közvetlenül a név után"`

Mint említettem, a héj változói önműködőek, vagyis csak első használatuk pillanatában jönnek létre. Ez egyes helyzetekben gondot okozhat, mivel bizonyos változók tartalma esetleg csak futásidőben dől el. Lehetőségünk van azonban arra, hogy egy korábban nem meghatározott változóra érték szerint hivatkozzunk, ami „normális” körülmények között hibaüzenetet vált ki. A nem meghatározottság állapotának kezelésére három lehetőségünk van (eltekintve a változó értékének vizsgálatától). A

`$ (változó-alapértelmezett érték)`

kifejezés értéke a változó értéke, ha az meghatározott, ellenkező esetben a megadott alapértelmezett érték. Ilyenkor a változó továbbra is nem meghatározott marad. A

`$ (változó=alapértelmezett érték)`

ettől csak annyiban tér el, hogy ha a változó nem meghatározott, akkor értéket is ad neki. Végül a

`$ (változó?üzenet)`

típusú hivatkozás az üzenetben megadott szöveget írja ki és egyben megszakítja a futást, ha a változonak korábban nem adtunk értéket. Ha Bash parancsértelmezőt használunk, a fenti szerkezetekben a „-”, „=” és „?” karaktereket kettőspontnak kell megelőznie.

Idézőjelek használata, parancsbehelyettesítés

A héjprogramokban háromféle idézőjelet használhatunk a karakterláncok megadása során, attól függően, hogy milyen hatást akarunk elérni, illetve hogy a karakterlánc tartalmát milyen mértékig akarjuk betű szerint (literálisan) értelmezni.

Az egyszeres idézőjelek (' . . . ') használata teljes elzárást jelent. Az ilyen karakterláncokban a héj semmiféle értelmezést nem végez, azok tartalmát betű szerint kezeli (például kiíratásnál).

A kettős idézőjelek (" . . . ") részleges elszigetelést jelölnek, ami azt jelenti, hogy a héj értelmezi és behelyettesíti a benne található, számára értelmes karaktereket. Ha például egy * karaktert talál, akkor azt a munkakönyvtárban levő fájlok neveinek listájával fogja helyettesíteni. Ha \$ karaktert „lát”, a közvetlenül utána következő szót változónévnek vagy parancssori paraméternek tekinti, és a megfelelő értéket behelyettesíti.

Megeshet természetesen, hogy valóban egy olyan karaktert akarunk beírni egy ilyen karakterláncba, amely a héj számára is értelmes, viszont valamilyen okból kifolyólag nem használhatjuk a kellő védelmet nyújtó egyszeres idézőjelet. Ilyenkor az értelmezést a kérdéses karaktert megelőző \ jellel akadályozhatjuk meg.

A „visszafelé hajló” egyszeres idézőjelek (` . . . `) kifejezett parancsvégrehajtást jelölnek. Az ilyen szöveget a héj parancssornak tekinti, végrehajtja, és az eredménnyel helyettesíti. Ez a parancsbehelyettesítésnek nevezett eljárás lehetővé teszi, hogy egy változónak azonnal átadjuk egy esetleg meglehetősen bonyolult parancssor kimenetét.

Parancssori paraméterek

A héjprogramnak meghívásakor a parancssorban átadhatunk egy vagy több paramétert. Több paraméter esetén azokat egy vagy több szóköznek kell elválasztania. Ha az átadandó paraméter maga is tartalmaz szóközt, kettős idézőjelbe kell tenni.

A parancssori paraméterek értékére a \$1, \$2 stb. szimbólumokkal hivatkozhatunk. A szám a kérdéses paraméter sorszáma. A 0 sorszámú paraméter minden esetben maga a meghívott program neve. Ez akkor is igaz, ha egyébként egyáltalán nem adtunk meg parancssori paramétereket.

A sorszám legnagyobb értéke a régebben fejlesztett UNIX rendszereken 9. Ha en-nél több paramétert kell kezelnünk, a shift parancsal csúszthatjuk egy helyen balra a paraméterlistát. Ilyenkor az éppen 1-es sorszámú paraméter elvész, tehát ha a későbbiekben szükség lehet rá, megőrzéséről a programnak kell gondoskodnia. A shift parancs a \$0 paramétert nem érinti, a \$# nevű belső változóban tárolt paraméterszámot azonban csökkenti.

A szabványos be- és kimenet és ezek átirányítása

Ha egy tetszőleges operációs rendszeren futtatunk valamilyen programot, annak az esetek túlnyomó többségében valamilyen módon kapcsolatot kell tartania a külvilággal. A külvilág lehet maga a felhasználó, az operációs rendszer egyes szolgáltatásai, vagy a háttértárakon található adatok.

Valamennyi UNIX alatt futó program rendelkezik négy alapértelmezett kapcsolat-tartási lehetőséggel. Ezeket szabványos bemenetnek, szabványos kimenetnek, szabványos hibacsatornának és visszatérési értéknek nevezzük.

A UNIX szemlélete szerint egy program afféle háromágú átfolyó csatornaként működik. A szabványos bemenetére érkező adatokat fogadja, alapértelmezett szerepének, illetve a megadott módosító kapcsolóknak megfelelően feldolgozza azokat, majd az eredményt a szabványos kimenetére küldi. Ha közben hiba keletkezett, vagy valami egyéb közlendője van, akkor azt a harmadik ágon, a szabványos hibacsatornán adja ki. Hogy ezek a csatornák ténylegesen mit jelentenek (hová vezetnek), az az adott körülményektől függ.

Ha egy UNIX parancsnak nem adunk meg külön szabványos bemenetet, akkor a billentyűzetről várja a feldolgozandó adatokat. Ha nincs kifejezetten megadott kimenet, akkor a képernyőre írja az eredményt. Lássuk példaként a cat program működését. Ez semmi egyebet nem csinál, mint a bemenetére érkező adatokat hűségesen elismétli a kimenetén. A

```
cat telefon.txt
```

parancs bemenetét parancssori paraméterként adtuk meg, a kimenetéről viszont semmilyen módon nem rendelkeztünk. Így a szöveg a képernyőn fog megjelenni. (Az adatok most valójában nem a szabványos bemenetről érkeznek, mivel fájlnevet adtunk meg. A bővebb magyarázatot a következő szakasz tartalmazza.)

Ha a cat parancsot egymagában, mindenféle parancssori paraméter nélkül adjuk ki, akkor sem a bemenetet, sem a kimenetet nem adtuk meg. Ezért aztán nem tehet mászt, mint kitesz egy > alakú készenléti jelet (másodlagos prompt) és várja, hogy begépeljük az adatokat. A bemenet végét a CTRL+D billentyűkombinációval jelezhetjük. Ekkor a cat nekilát a „feldolgozásnak”, vagyis elismétli a képernyőn a begépelt szöveget.

A kimenetet természetesen át is irányíthatjuk egy fájlba. Erre szolgál a > műveleti jel. A

```
cat > lista.txt
```

parancs szabványos bemenete mégint nem meghatározott, vagyis a billentyűzetről fogja várni azt. A kimenetét viszont a lista.txt nevű fájlba irányítottuk, így most a gépelést lezáró CTRL+D után nem a képernyőn fogja elismételni a szöveget, hanem létrehozza a megadott nevű fájlt, és abba teszi.

Ha a fájl már létezett, felülírja. Természetesen az is megoldható, hogy egy már létező fájlhoz fűzzünk hozzá újabb szövegrészt. Erre szolgál a >> műveleti jel. A

```
cat >> lista.txt
```

parancs tehát ugyanúgy működik, mint az előbb, azzal az eltéréssel, hogy ha a lista.txt még nem létezik, akkor létrehozza, ha pedig már létezik, akkor hozzáfűzi a begépelt szöveget.

A kimenethez hasonlóan a parancsok bemenete is átirányítható a < műveleti jel segítségével. A

```
cat < lista.txt
```

parancs tulajdonképpen ugyanazt teszi, mint a legelső példa, vagyis kiírja a képernyőre a fájl tartalmát. A különbség ebben az esetben „elméleti”, mégis lényeges. Az első esetben a cat program parancssori paraméterként kapta meg a fájl nevét. Ezt úgy értelmezte, hogy meg kell azt nyitnia, és ami benne van, azt kell bemenetnek tekintenie.

A második esetben a fájl nevét nem a cat parancs, hanem a parancsértelmező (héj) kapja meg. Ez nyitja meg, és a tartalmát a cat bemenetére küldi. Magának a cat-nek tehát ebben az esetben „fogalma sincs róla”, hogy mi a fájl neve, Ő a bemenet készen kapja a héjtól.

Mármost megkérdezheti valaki, hogy ha a különbség ennyire árnyalatnyi, akkor tulajdonképpen mire jó a < operátor. A válasz egyszerű. Egy UNIX alatt futó programnak nem kötelessége, hogy tudjon parancssori paraméterként fájlnevet fogadni, vagy hogy egyáltalán meg tudjon nyitni egy fájlt és képes legyen abból olvasni. Az viszont igen, hogy legyen szabványos bemenete. Azt hogy a cat (és természetesen még sok más UNIX program) elég értelmes ahhoz, hogy minden formában elfogadja az utasítást, tekintsük ajándéknak!

Csövek

A > és >> műveleti jelek mindig egy láncolat végét jelentik azzal, hogy fájlba írják át a kimenetet. Lehetőség van azonban több program összeláncolására is az úgynvezett csövek segítségével. A csővezeték (pipe) a héj egy különleges és rendkívül hasznos szolgáltatása. Tulajdonképpen ez az, ami a UNIX-ot igazán hatékonyá teszi azzal, hogy programozás nélkül képes összekapcsolni az egyedi szolgáltatásokat nyújtó parancsokat.

A cső tulajdonképpen egy átmeneti tároló elem, amely fogadja egy program kimenetét, és azonnal egy másik bemenetére kapcsolja, anélkül, hogy a kérdéses tartalom bárhol felbukkanna fájl, képernyőre írt szöveg, vagy bármilyen más formájában. A csövet jelképező műveleti jel valamennyi héjban a | karakter.

Előző példánknál maradva tételezzük fel, hogy telefonkönyvünk bejegyzéseit ábécésorrendbe akarjuk rendezni. A UNIX-ban természetesen a sorba rendezésre is van parancs, amit nem túl meglepő módon sort-nak hívnak. A sort program a cat-hez hasonlóan egy egészen egyszerű feladatot hajt végre: a bemenetére érkező sorokat az angol ABC szerint sorba rendezi és visszaadja a kimenetén. A kívánt hatás ezek után a következő „soros kapcsolással” érhető el:

```
cat telefon.txt | sort
```

Itt a cat a paraméterként megadott fájlból mint bemenetről elolvassa a sorokat, majd a sort bemenetére küldi azokat. Ez sorba rendezi a szöveget; majd kifejezetten megadott kimenet hiányában a képernyőre írja azt. Természetesen az sem gond, ha a rendezett változatot meg akarjuk őrizni. Mindössze a sort kimenetét kell egy fájlba irányítanunk a már ismert módon:

```
cat telefon.txt | sort > rendezett.txt
```

Buktató**Dugulás a csőben**

Gondolkodjunk el, mi történik a következő parancssor hatására:

```
cat telefon.txt | ls -l | sort
```

Sokan arra számítanak, hogy a telefon.txt fájl tartalma és az ls parancs kimenete egyaránt a sort bemenetére kerül, amely aztán betűrendbe szedve jeleníti meg azokat. A valóság azonban az, hogy csak a könyvtárlistát fogjuk viszontlátni. Ennek az az oka, hogy az ls nem vár semmit a szabványos bemenetén, és ennek megfelelően nem is kezeli azt. Az első csőbe beküldött adatokat a rendszer egyszerűen „lenyeli”, azok örökre, nyomtalanul eltűnnek. A UNIX csöveiben – akárcsak a valódi csövekben – is előidézhetünk tehát dugulást, ha nem megfelelően használjuk azokat.

A csövekkel kapcsolatban még egy hasznos segédprogramról kell említést tenni. A tee parancs képes egy csövet „elágaztatni”. Ezt úgy éri el, hogy a szabványos bemenetére érkező adatokat két helyre küldi: egyrészt a szabványos kimenetére, biztosítva ezzel a cső folyamatosságát, másrészt a paraméterként megnevezett fájlba. Így az adott ponton „megcsapolja” az adatfolyamot, anélkül, hogy a feldolgozásba bármi módon beavatkozna.

A -a kapcsoló hatására a paraméterként megadott kimeneti fájlt nem írja felül, hanem hozzáfüzi az újabb adatokat. A tee parancs általában akkor hasznos, ha egy hosszabb héjprogramban vagy parancsláncolatban keresünk hibát, és szükség van az egyes pontokon átáramló adatok ismeretére.

Buktató

Kezdő felhasználók viszonylag gyakran próbálkoznak a következő „trükkös” megoldásokkal:

```
...
szam=3
$szam | másik_program
...
vagy
...
szoveg=" "
cat telefon.txt > szoveg
...
```

Bár a maga módján mindenki megoldás logikus, természetesen mindenki helytelen. Az első példában egy „változó kimenetét” igyekszünk egy

program bemenetére irányítani. A gond ezzel csak az, hogy egy változónak értéke van, „kimenete” azonban nincs. Csőbe irányítható kimenete csak programnak lehet, tehát a helyes megoldás a következő:

```
...
szam=3
echo $szam | másik_program
...
```

A második példa is egy hasonló fogalomzavaron alapul. Itt egy változóban igyekszünk elhelyezni egy program teljes kimenetét. Ez természetesen lehetséges, de nem a > műveleti jel segítségével. Ezzel ugyanis kizárolag fájlba lehet kimenetet átirányítani. A megoldás a korábban említett parancsbehelyettesítés:

```
...
szoveg=`cat telefon.txt`
...
```

A csatornák számozása és a hibacsatorna átirányítása

A UNIX a programok be- és kimeneti, valamint hibacsatornájához egy-egy számot is rendel. A szabványos bemenet (`stdin`) numerikus megfelelője a 0, a kimeneté (`stdout`) az 1, a szabványos hibáé (`stderr`) pedig a 2. A csatornáakra átirányításnál ezekkel a számokkal is hivatkozhatunk. Ennek igazából a hibacsatorna kezelésénél van jelentősége.

A `cat` példájánál maradva a hibát a következő módon irányíthatjuk egy fájlba:

```
cat telefon.txt 2>hiba.txt
```

Ha most a parancs végrehajtása közben bármilyen hiba lép fel, a `cat` az ezzel kapcsolatos üzenetet nem a képernyőre, hanem a `hiba.txt` fájlba fogja küldeni. (Ha nem rendelkezünk külön a szabványos hibacsatornáról, az alapértelmezett kimenet természetesen itt is a képernyő lesz.)

Rögtön itt hívom fel a figyelmet arra, hogy a szabványos hiba átirányítása a legkiválogatottabb módja a nehezen felderíthető, „misztikus” hibák előállításának. Egy bonyolultabb héjprogram esetében ugyanis így közvetlenül nem látjuk semmi jelét annak, hogy bármi gond lett volna a feldolgozás közben.

A fejlesztési szakaszban így sokkal célszerűbb a hibát a szabványos kimenetre „rákeverni”. Erre szolgál a & műveleti jel. A

```
program 1>kimenet.txt 2>&1
```

parancssor a program kimenetét (1) a kimenet.txt fájlba irányítja, az esetlegesen keletkező hibaüzeneteket pedig „összefésüli” vele. Így végül ezek is a kimeneti fájlba érkeznek.

Buktató**Az átirányítások sorrendje**

A fenti példában az átirányításokat feltétlenül ebben a sorrendben kell megadni. Ha ugyanis a hibacsatorna átirányításáról korábban rendelkezünk, mint a kimenetről, formai (szintaktikai) hibát nem követünk el ugyan, de a dolog biztosan nem úgy fog működni, ahogy elterveztük. Jó esetben a hibaüzenetek továbbra is a képernyőre kerülnek, rossz esetben egyszerűen „lenyeli” őket a rendszer. A végkifejlet rendszerfüggő, de mindig rossz.

TIPP**Saját hibaüzenet küldése**

A hibacsatorna fenti átirányításának a fordítottja is értelmes. Képzeljük el, hogy egy héjprogramból hibaüzenetet akarunk küldeni. A legegyszerűbb megoldás természetesen az
echo "Hibaüzenet..."

Ennek azonban van egy komoly szépséghibája: a szöveg a szabványos kimeneten fog megjelenni, nem a szabványos hibacsatornán. Ha viszont az

```
echo "Hibaüzenet..." 1>&2
```

megoldással élünk, akkor minden úgy történik, ahogy az a UNIX világában szokásos. Ha a felhasználó másként nem rendelkezett, akkor a szöveg természetesen továbbra is a képernyőn jelenik meg, így viszont megvan a lehetősége arra, hogy ezen változtatasson.

A UNIX operációs rendszerekkel kapcsolatban – különösen kezdő felhasználók – gyakran említik hibáként azok szűkszavúságát. Ez a „vád” tulajdonképpen teljesen jogos: UNIX-on annak a legbiztosabb jele, hogy minden rendben van, az, ha az ENTER leütése után a képernyőn nem jelenik meg semmi. Ez a „csend” a kezdők számára kezdetben kétségekívül őrjítő lehet. Gondoljunk azonban bele, hogyan viselkedne egy szószátyár UNIX!

Ha a sikeres végrehajtásról is érkeznének üzenetek, azoknak szükségszerűen a szabványos hibacsatornán kellene megjelenniük. Ez pedig szerfölött kellemetlen lenne, hiszen a lelkendező értesítéseket akkor is „gyomlálnunk” kellene az adatfolyamból, ha amúgy minden a legnagyobb rendben folyik.

Matematikai kifejezések kiértékelése : az expr parancs

A UNIX parancsértelmezők általában meglehetősen korlátozott aritmetikai képességekkel rendelkeznek. Ugyanakkor számos esetben lehet szükségünk egyszerű műveletek programból való elvégeztetésére. (A legegyszerűbb és leggyakoribb példa erre egy ciklusváltozó növelése.) Erre szolgál az expr parancs. Az expr gyakorlatilag egy négy alapműveletes számológép, de kizártlag egész számokkal boldogul. Kapcsolóiról és a lehetséges műveletekről az 1.2. táblázatok adnak összefoglalást.

1.2A. táblázat Az expr logikai operátorai

OPERÁTOR	JELENTÉS
	Logikai VAGY operátor. Visszatérési értéke az első paraméter, ha az nem nulla, vagy nem üres karakterlánc, ellenkező esetben a második.
&	Logikai ÉS operátor. Visszatérési értéke az első paraméter, ha egyik argumentuma sem nulla vagy üres karakterlánc. Ellenkező esetben nullát ad vissza.

1.2B. táblázat Az expr által ismert relációs jelek

JEL	JELENTÉS
<	Kisebb
<=	Kisebb vagy egyenlő
=	Egyenlő
==	Egyenlő
!=	Nem egyenlő
>=	Nagyobb vagy egyenlő
>	Nagyobb

1.2C. táblázat Az expr által ismert aritmetikai műveletek

MŰVELETI JEL	JELENTÉS
+	Összeadás
-	Kivonás
*	Szorzás
/	Egészosztás
%	Maradékképzés (modulo)

Relációs jelek használatakor az expr 1-et ad vissza, ha az összehasonlítás igaz és nullát, ha hamis. Az összehasonlítás elvégzése előtt megkísérli számokká alakítani a megadott paramétereket. Ha ez sikeres, aritmetikai összehasonlítást végez. Ha bármelyik paramétert nem tudja így átalakítani, az összehasonlítás betűrend szerinti (lexikografikus) lesz. Ilyenkor az a paraméter számít nagyobbnak, amelyikben előbb következik magasabb ASCII kódú karakter.

Aritmetikai műveletek természetesen csak számokon hajthatók végre. Ha valamelyik paraméter nem alakítható számmá, hiba keletkezik.

A táblázatok sorrendje egyben az expr műveleteinek rangsorát is tükrözi. Mindazonáltal a műveleteket kerek zárójelek alkalmazásával csoportosíthatjuk, így felülbírálhatjuk a szokásos kiértékelési sorrendet (precedenciát).

Buktató

Ügyeljünk rá, hogy az expr „felfogóképessége” viszonylag szegényes. Az egyszerű műveleti jeleket is csak akkor hajlandó megérteni, ha azokat szóközök választják el a tényezőktől (argumentumoktól). A

```
szam=`expr 3+2`
```

forma tehát – bármilyen meglepő – helytelen. A helyes írásmód:

```
szam=`expr 3 + 2`
```

Arról sem szabad megfeledkeznünk, hogy az expr egyes műveleti jelei a héj számára is értelmesek. Ha ezeket elfelejtjük levédeni a \ karakterrel, annak igen furcsa mellékhatásai lehetnek. (A gondot okozó jelekkel kapcsolatban lásd a következő példaprogramot.)

Az expr parancs képességeit és a héj számára is értelmes műveleti jelek levédését a következő példaprogram szemlélteti.

1.3. lista

```
1: #!/bin/sh
2: # Az expr képességei ...
3:
4: # Logikai műveletek
5:
6: # Logikai VAGY
7: string1="hhh"
8: string2="vvv"
9: ertek=`expr $string1 \| $string2`
10: echo $ertek
11:
12: # Logikai ÉS
13: ertek=`expr $string2 \& $string1`
14: echo $ertek
15:
16: # Aritmetikai összehasonlítás
17:
18: szam1=5
19: szam2=12
20:
21: # Kisebb
22: ertek=`expr $szam1 \<; $szam2`
23: echo $ertek
24:
25: # Kisebb vagy egyenlő
26: ertek=`expr $szam1 \<= $szam2`
27: echo $ertek
28:
29: # Egyenlő 1.
30: ertek=`expr $szam1 = $szam2`
31: echo $ertek
32:
33: # Egyenlő 2.
34: ertek=`expr $szam1 == $szam2`
35: echo $ertek
36:
37: # Nem egyenlő
38: ertek=`expr $szam1 != $szam2`
39: echo $ertek
40:
41: # Nagyobb
42: ertek=`expr $szam1 \> $szam2`
43: echo $ertek
44:
```

```

45: # Nagyobb vagy egyenlő
46: ertek=`expr $szam1 \>= $szam2`
47: echo $ertek
48:
49: # Aritmetikai műveletek
50:
51: osszeg=`expr $szam1 + $szam2`
52: kulonbseg=`expr $szam1 - $szam2`
53: szorzat=`expr $szam1 \* $szam2`
54: hanyados=`expr $szam2 / $szam1`
55: maradek=`expr $szam2 % $szam1`
56:
57: echo "$szam1 + $szam2 =" $osszeg
58: echo "$szam1 - $szam2 =" $kulonbseg
59: echo "$szam1 * $szam2 =" $szorzat
60: echo "$szam2 / $szam1 =" $hanyados
61: echo "$szam2 mod $szam1 =" $maradek

```

Parancsvéghajtás : az eval parancs

Vannak esetek, amikor egy bizonyos műveletsor elemei attól függően alakulnak, hogy a program előző szakaszában milyen eredmények születtek, illetve milyen válaszok érkeztek a felhasználótól. Ez azt jelenti, hogy olyan műveletsort kell majd végrehajtanunk, amely csak a program futása közben áll össze. Ilyenkor magukat a műveleteket szöveg formájában, egy változóban tárolhatjuk, és a teljes parancssor elkészültekor az

`eval $változó`

parancssal hajthatjuk végre. Az eval kimenete megegyezik a végrehajtott parancssor kimenetével, így az eredmény elhelyezhető egy változóban, illetve fájlba vagy csőhálózatba irányítható.

A beágyazott dokumentum (here document)

A beágyazott dokumentum használatának módszere alapvetően hosszú szövegrések egyszerű kiíratására, illetve egyéb feldolgozására szolgál. Lényege, hogy a szöveg elejét és végét megjelöljük egy tetszőleges jelsorozattal, ami bármilyen karakterlánc lehet (általában egy értelmes szót érdemes választani). Ha egyszerűen a szöveg kiíratása a cél, a következő módon járhatunk el:

```
cat << határoló_jel
...
KIÍRANDÓ SZÖVEG
...
határoló_jel
```

A szöveg tartalmazhat már meghatározott változókra történő érték szerinti hivatkozást is. Megjelenítésekor a héj ezeket ugyanúgy be fogja helyettesíteni, mint az egyéb környezetekben. Ha valamiért kifejezetten le akarjuk tiltani a héjnak ezt az értelmező szerepét, a határoló jelet kettős idézőjelek között kell megadni.

A beágyazott dokumentum használatára jó példa lehet a számlák nyomtatása. A számla alapszövege nyilván minden esetben azonos, csak az áru megnevezése, a vevő és az ár változik. Ezt a három dolgot három változóban tárolhatjuk, amelyeket a program futás közben bekér a felhasználótól, vagy még célszerűbben kiolvasva azokat egy adatbázisból. Az egyéb, soha nem változó szövegrészket egy beágyazott dokumentum formájában maga a program tartalmazza. Íme egy hipotetikus példa, ahol a vevő adatait eleve megadtuk a megfelelő változókban:

1.4. lista

```
1: #!/bin/sh
2:  # Példa beágyazott dokumentum használatára: számla nyomtatása
3:
4:  vevo="Kovács János"
5:  vevo_cime="Budapest, Vadvirág u. 2."
6:  arucikk="Micimackó kuckója (könyv)"
7:  ar=812
8:
9:  cat << szamla-vege
10:                         Számla
11:                         -----
12:  Eladó: Kapu Könyvesbolt           Vevő: $vevo
13:          Budapest                   $vevo_cime
14:          Mammut Üzletközpont
15:  Áru megnevezése: $arucikk
16:  Fizetendő: $ar Ft
17:
18:                         Köszönjük, hogy nálunk vásárolt!
19:  szamla-vege
20:  exit 0
```

Ha lefuttatjuk, így fog kinézni a kimenet:

Számla

Eladó: Kapu Könyvesbolt Vevő: Kovács János
 Budapest, Vadvirág u. 2.
 Mammut Üzletközpont
Áru megnevezése: Micimackó kuckója (könyv)
Fizetendő: 812 Ft

Köszönjük, hogy nálunk vásárolt!

Buktató**Beágyazott dokumentumok formázása**

Egyes – elsősorban régebbi – UNIX rendszereken a Bourne parancsértelmező „rosszul tűri” azokat a szövegformázásokat, amelyeket a fenti példában láthatunk. Ezek csak akkor hajlandóak működni, ha a beágyazott dokumentum valamennyi sora az első pozícióon kezdődik. (Sor közben már lehet tetszőleges számú szóköz, csak az elején nem.) Ilyenkor nem nagyon van mit tenni. Vagy megbékélünk a hibával és kerülgetjük, amíg bírjuk, vagy telepítjük a Bash (Bourne Again Shell) parancsértelmezőt, amely nyílt forráskódú szabad program. Ez biztosan működni fog.

2

Programvezérlési szerkezetek

Minden programnyelvben szükség van olyan elemekre, amelyekkel elágazásokat, döntéseket, ugrásokat tudunk megvalósítani, illetve az egyes részek szerepe szerint tagolhatjuk a forráskódot. Ebben a fejezetben a UNIX parancsértelmezők ilyen szolgáltatásaival fogunk megismerkedni.

Feltételes utasítás : if, test, && és ||

A feltételes elágaztató utasítás formája a következő:

```
if [ logikai kifejezés1 ]
then
    ...parancsok...
elif [ logikai kifejezés2 ]
then
    ...parancsok...
else
    ...parancsok...
fi
```

Végrehajtás előtt a héj kiértékeli kifejezés1-et. Ha ez igaz értéket ad vissza, végrehajtja az ehhez tartozó utasításblokkot, a többöt pedig átlépi. Az elif kulcsszó után további logikai feltételek is megadhatók. Ezek sorrendben értékelődnek ki, és az első igaz értékhez tartozó programág hajtódik majd végre.

A többi feltételt ilyenkor ki sem értékeli a héj, tehát az ezekhez tartozó parancsok akkor sem hajtódnak végre, ha a feltétel igaz. Az else ágra akkor kerül a vezérlés, ha a parancsértelmező egyik kifejezést sem találta igaznak. Sem az elif, sem az else ág megadása nem kötelező.

A „logikai kifejezés” legtöbbször egy test parancs. A test parancs a héjprogramozáshoz idomított logikai kifejezések kiértékelésére szolgál (tehát nemcsak a matematikai logikából ismert műveleteket ismeri). Kétféle utasításformája van:

```
test logikai_kifejezés
```

vagy

```
[ logikai_kifejezés ]
```

Sokak szerint az utóbbi írásmód javítja a nagyobb programok olvashatóságát. Jelen téset tekintve a két megoldás természetesen teljesen azonos.

Buktató

Ha a második írásmód mellett döntünk, ügyeljünk rá, hogy számos rendszer parancsértelmezője megköveteli a szögletes zárójeleket a logikai kifejezéstől elválasztó szóközök meglétét. Vannak ugyan kevésbé finnyás változatok is, de ha hordozható programot akarunk írni, „jobb a békesség” alapon tegyük inkább ki azt a két szóközt. Ez biztosan működni fog mindenütt.

A test parancs kapcsolóról a 2.1. táblázatok adnak összefoglalást.

2.1A. táblázat A test parancs fájlokra alkalmazható kapcsolói

KAPCSOLÓ	JELENTÉS
-r	Értéke igaz, ha a fájl létezik és olvasható.
-w	Értéke igaz, ha a fájl létezik és írható.
-x	Értéke igaz, ha a fájl létezik és végrehajtható.
-f	Értéke igaz, ha a fájl létezik és közönséges fájl.
-d	Értéke igaz, ha a bejegyzés létezik és könyvtár.
-h (-L)	Értéke igaz, ha a bejegyzés létezik és közvetett hivatkozás.
-c	Értéke igaz, ha a bejegyzés létezik, és karakteresköz-meghajtó.
-b	Értéke igaz, ha a bejegyzés létezik és blokkeszköz-meghajtó.
-p	Értéke igaz, ha a bejegyzés létezik és nevesített csővezeték (FIFO).
-u	Értéke igaz, ha a fájl létezik, és setuid bitje be van állítva.
-g	Értéke igaz, ha a fájl létezik, és setgid bitje be van állítva.
-k	Értéke igaz, ha a fájl létezik, és sticky bitje be van állítva.
-s	Értéke igaz, ha a fájl létezik, és hossza nem nulla.

2.1B. táblázat A test parancs karakterláncokra alkalmazható kapcsolói

KAPCSOLÓ	JELENTÉS
-z karakterlánc	Értéke igaz, ha a karakterlánc hossza nulla.
-n karakterlánc	Értéke igaz, ha a karakterlánc hossza nem nulla.
karakterlánc1 = karakterlánc2	Értéke igaz, ha a két karakterlánc azonos.
karakterlánc1 != karakterlánc2	Értéke igaz, ha a két karakterlánc nem azonos.
karakterlánc	Igaz értéket ad vissza, a karakterlánc bájtösszege nem nulla, vagyis tartalmaz nullától különböző bájtot.

2.1C. táblázat A test parancs egész számokra alkalmazható operátorai

OPERÁTOR	JELENTÉS
n1 -eq n2	Értéke igaz, ha n1 és n2 egyenlők.
n1 -ne n2	Értéke igaz, ha n1 és n2 nem egyenlők.
n1 -gt n2	Értéke igaz, ha n1 nagyobb, mint n2.
n1 -ge n2	Értéke igaz, ha n1 nagyobb vagy egyenlő, mint n2.
n1 -lt n2	Értéke igaz, ha n1 kisebb, mint n2.
n1 -le n2	Értéke igaz, ha n1 kisebb vagy egyenlő, mint n2.

2.1D. táblázat A test parancs logikai operátorai

OPERÁTOR	JELENTÉS
!	Tagadás (NOT) (egytényezős)
-a	Logikai ÉS (AND) (kéttényezős)
-o	Logikai VAGY (OR) (kéttényezős)
(...)	Kiértékelési sorrend. A zárójelbe tett logikai kifejezések együttesen értékelődnek ki, és egyetlen eredményt szolgáltatnak. (Ügyeljünk rá, hogy a zárójeleket a héj is értelmezi!)

Megjegyzés

A test aritmetikai összehasonlítást végző operátorai meglehetősen furcsák, különösen így, a XXI. században. Ennek elsősorban „történeti okai vannak”. (Valahányszor valami „csökevény” létfogosultságát kell bizonyítani a számítástechnikában, igen jól jön a „történeti okok” felemlegetése.) Aki írt már Fortran nyelven programot, annak természetesen ezek a kapcsolók ismerősek lesznek. A gond csak az, hogy józan emberi számítás szerint már a Fortrannak sem szabadna léteznie úgy jó húsz éve. Jó hír viszont, hogy a Bash – bár együttműködési okok miatt továbbra is érti a régi jelöléseket – támogatja a szokásos relációs jelek használatát is.

Ha csupán kettős elágazást akarunk létrehozni, arra létezik egy egyszerűsített megoldás:

```
logikai kifejezés && parancs1 || parancs2
```

A „logikai kifejezés” itt egy művelet vagy műveletsor lehet. Ha ennek végre-hajtása sikeres, vagyis visszatérési értéke 0, a továbbiakban a parancs1 hajtódik végre, nullától különböző visszatérési érték esetén pedig a parancs2.

Általában is igaz, hogy „logikai kifejezést” nem csak a `test` parancs segítségével lehet megfogalmazni. A héj az `if`, illetve `elif` után megadott parancs vagy parancssor visszatérési értéke alapján dönt a kifejezés igaz vagy hamis voltáról, bármi is legyen az illető utasítás. Tulajdonképpen maga a `test` program is ezt a módszert használja, vagyis nullát ad vissza, ha a megadott logikai kifejezés igaz, és 1-et, ha nem.

A fentieknek megfelelően ha például el akarjuk dönteni, hogy egy könyvtár, ahová be akarunk lépni a `cd` parancssal, egyáltalán létezik-e már, a következő kódrészletet használhatjuk:

2.1. lista

```

1: #!/bin/sh
2: # Van-e ilyen könyvtár?
3: if cd mappa 2>/dev/null
4: then
5:   echo "A könyvtár létezik."
6:   exit 0
7: else
8:   echo "A könyvtár nem létezik."
9:   exit 1
10: fi

```

A 3. sorban az `if` után megadott `cd` parancsot a rendszer megkísérli végrehajtani. Ha a végrehajtás sikeres, a parancs visszatérési értéke 0 lesz, így az igaz ágra kerül a vezérlés. Bármilyen hiba esetén a visszatérési érték nullától különböző érték lesz, így a hamis ág hajtódik végre. Látható, hogy a 3. sorban a `cd` parancs hibacsatornáját (2-es csatorna) átirányítottam a `/dev/null`-ba. (Ez UNIX-on a „fekete lyuk”: bármi, ami ide kerül, örökre eltűnik.) Erre azért van szükség, hogy a `cd` saját hibaüzenete ne kerüljön a képernyőre, csak a programé.

Többszörös elágaztatás : a `case` szerkezet

A többszörös elágaztatás utasításformája a következő:

```

case string0 in
  string1) parancsok;;
  string2) parancsok;;
...
*) parancsok;;
esac

```

A végrehajtás mikéntjét itt az dönti el, hogy a case szerkezet fejlécében megadott karakterlánc (`string0`) melyik másik karakterláncra (`string1, string2...`) illeszkedik. A felsorolt lehetőségek nemcsak közönséges szöveges változók vagy állandók, hanem szabályos kifejezések is lehetnek, ami nagyban növeli a case szerkezetek rugalmasságát. Ha `string0` egyetlen másikra sem illeszkedik, a * karakterrel jelzett ág hajtódnak végre, de ennek megadása nem kötelező. Ha nincs ilyen ág, a héj egyszerűen átugorja a teljes case szerkezetet. Figyeljük meg, hogy valamennyi case blokkot kettős pontosvessző zárja le. Ezek a blokkok természetesen több utasítást is tartalmazhatnak. Ezeket külön sorokban vagy egyszeres pontosvesszővel elválasztva kell megadni.

Egyszerű példaként írunk egy héjprogramot, amely az egyjegyű számokat képes betűkkel megjeleníteni. Íme a forráskód:

2.2. lista

```

1: #!/bin/sh
2: # Egyjegyű szám kiíratása betűkkel
3:
4: # Megvizsgáljuk, hogy van-e parancssori paraméter
5: if [ $# -eq 0 ]
6: then
7:   echo "Adjon meg parancssori paraméterként egy egyjegyű számot!"
8:   exit 1
9: fi
10:
11: echo "Ön a következő számot adta meg: "
12: case $1 in
13:   1) echo "Egy";;
14:   2) echo "Kettő";;
15:   3) echo "Három";;
16:   4) echo "Négy";;
17:   5) echo "Öt";;;
18:   6) echo "Hat";;;
19:   7) echo "Hét";;;
20:   8) echo "Nyolc";;;
21:   9) echo "Kilenc";;;
22:   *) echo "EZ nem egyjegyű szám!"; exit 2 ;;
23: esac
24: exit 0

```

Programunk parancssori paraméterként várja a kiírandó számot, így először azt kell megvizsgálnunk, hogy a felhasználó egyáltalán adott-e meg valamit. A parancssori paraméterek számát a `$#` belső változó tartalmazza. Az 5. sorban ezt vizsgáljuk

meg egy „álcázott” test parancsal. Ha a paraméterek száma nulla, a program tájékoztató szöveget ír ki, majd leáll, hiszen nincs feldolgozandó adat. A hívó programnak az 1-es visszatérési értékkel jelzi a bekövetkezett hiba természetét.

A case szerkezet csak az első parancssori paraméter illeszkedését vizsgálja a megadott mintákra. Ha nincs illeszkedés, hibaüzenetet ír ki, és kettes visszatérési értékkel leáll. minden egyéb esetben nulla lesz a visszatérési érték, ami a szabályos leállás egyezményes jele.

Ciklusszervezés : for, while és until

A ciklusszervezésre egyéb nyelvekhez hasonlóan több lehetőségünk is van. A for ciklus utasításformája a következő:

```
for ciklusváltozó in lista
do
    parancsok
done
```

Végrehajtáskor a ciklusváltozó értéke sorra felveszi a listában megadott értékeket, és minden egyes értékkel végrehajtódik a ciklusmag. Fontos megjegyezni, hogy a ciklusváltozó lehetséges értékeit csak lista formájában adhatjuk meg. Ez jelentősen eltér az egyéb programnyelvek ciklusszervezési „szokásaitól”, viszont jól idomul azokhoz a feladatokhoz, amelyekre héjprogramot szokás írni. Arra ugyanis nincs semmilyen megkötés, hogy az említett listának honnan kell származnia. Nem kell feltétlenül kézzel begépelni, előállíthatjuk egy másik programrészlettel, segédprogrammal vagy parancsbehelyettesítéssel is.

Ha például a ciklusmag utasításait a munkakönyvtárban található valamennyi fájlra alkalmazni akarjuk, a következő megoldással élhetünk:

```
for i in `ls`
do
    parancsok
done
```

A héj végre fogja hajtani az egyszerű ls parancsot, a kimenetét pedig a parancsbehelyettesítésnek megfelelően a for ciklus listájának fogja tekinteni.

A lista elemeit alapértelmezés szerint egy vagy több szóköz választja el egymástól. Azokat az elemeket, amelyek maguk is tartalmaznak szóközt, kettős idézőjelek közt kell zárni:

```
for i in a b "c d" e
do
    echo $i
done
```

Ennek a programnak a kimenete így fest:

```
a
b
c d
e
```

Buktató

Általában véve igaz az a fent megfogalmazott állítás, miszerint a `for` ciklus listája bárhonnan származhat. Ugyanakkor ha az elemek szóközt is tartalmaznak, van egy meglehetősen szigorú megszorítás: ilyen elemeket csak „kézzel” lehet megadni, vagyis úgy, hogy azokat magának a programnak a szövege tartalmazza (mint az előző példában). Ha a listát változó vagy parancssor behelyettesítésével adjuk meg, hiába helyezzük el a megfelelő helyen a megfelelő számú kettős idézőjelet, a `for` kizárolag a szóközöket fogja figyelembe venni, olyannyira, hogy az idézőjeleket is a lista elemeinek tekinti:

2.3. lista

```
1: #!/bin/sh
2: # A for ciklus rendhagyó viselkedése
3:
4: lista="a b \"c d\" e"
5: echo $lista
6:
7: for i in $lista
8: do
9:     echo $i
10: done
```

A várakozásokkal ellentében tehát ennek a programnak a kimenete

```
a
b
"c
d"
e
```

lesz, annak ellenére, hogy az 5. sorban kiadott echo parancs helyesen jeleníti meg a listát.

A bemutatott példákból talán érezhető, hogy a for ciklusok ciklusváltozója a héjprogramokban általában nem számokon halad végig. Annak sincs azonban semmi akadálya, hogy az egyéb nyelvre jellemző „numerikus” for ciklust megvalósítunk. A UNIX egyik segédprogramja, a seq ugyanis képes tetszőleges, állandó lépésközű számsorozatot előállítani. Nincs más dolgunk, mint ennek a kimenetét parancsbehelyettesítéssel a for ciklus listájában elhelyezni.

A másik két ciklusszervezési lehetőség szerkezetileg nagyon hasonló az előzőhöz, egymástól is csupán döntési módszerükben térnek el egymástól.

```
while [ logikai kifejezés ]
do
    parancsok
done

until [ logikai kifejezés ]
do
    parancsok
done
```

A while-lal szervezett ciklus addig fut, amíg a feltételeként szabott kifejezés igaz, az until-lal megvalósított viszont addig, amíg a kifejezés igazzá nem válik (vagyis amíg hamis). A „logikai kifejezés” és a szögletes zárójelek használatára vonatkozó szabályok azonosak a test parancsnál elmondottakkal.

TIPP

Ciklusok be- és kimenete

A ciklusok a héjprogramokban bizonyos mértékig önálló blokkokat képeznek. Lehet például saját be- és kimenetük. Ha például azt szeretnénk, hogy egy ciklus magjában levő valamennyi parancs kimenete ne a képernyőn, hanem egy fájlban jelenjen meg, a következő tehetjük:

```

while [ logikai kifejezés ]
do
    parancsok
done > kimeneti_fájl

```

Természetesen ugyanígy irányíthatjuk a kimenetet egy csőbe is a „|” műveleti jel segítségével.

Egyszerű példaként írassuk ki az egész számokat 1-től 10-ig két különböző ciklus-sal (for és while), úgy, hogy az egyik ciklus kimenete a képernyőn és egy fájlban is megjelenjen. Íme a forráskód:

2.4. lista

```

1: #!/bin/sh
2: # Ciklusok működésének bemutatása
3:
4: for i in `seq 10`
5: do
6:     echo $i
7: done
8:
9: i=1
10: while [ $i -le 10 ]
11: do
12:     echo $i
13:     i=`expr $i + 1`
14: done | tee szamok.txt

```

Látható, hogy a for ciklus listáját a seq segítségével állítottuk elő. A while ciklus kimenetét egy tee parancs segítségével kettéágaztattuk. A számsorozat kiíródik a képernyőre, a tee azonban elhelyezi a szamok.txt nevű fájlban is. Említést érdemel még a 13. sorban alkalmazott megoldás. Ez héjprogramok esetében a numerikus ciklusváltozó növelésének legsokványosabb módja. (C-ben ezt úgy írtuk volna hogy `i+=1;`) Vegyük észre, hogy a héj nem „téveszti el” a műveletek sorrendjét, vagyis előbb kiolvassa az i változó tartalmát, az expr segítségével egygel növeli és csak ezután helyezi vissza ugyanoda az eredményt, ahonnan kiolvasta az eredeti értéket. Parancsbehelyettesítés esetén az ilyen „visszacsatolás” mindenkor működik, van-nak azonban olyan UNIX parancsok is, amelyek nem türik el, hogy be- és kimene-tük azonos legyen. Erre később még visszatérünk.

TIPP**Végtelen ciklusok**

Végtelen ciklus felbukkanása az esetek túlnyomó többségében inkább programozási hiba, vannak azonban helyzetek, amikor akár hasznos is lehet. Egy `while` ciklus „végtelenítéséhez” nyilván csak arról kell gondoskodnunk, hogy a ciklusfeltétel mindenkor igaz legyen. Ezt megoldhatjuk például egy eleve igaz összehasonlítással (egy egyenlő eggyel), de van két elegánsabb megoldás is:

```
while true
do
...
done
```

illetve

```
while :
do
...
done
```

Mindkét szimbólum jelentése logikai IGAZ, de egyes rendszereken a kettő közül csak az egyik működik. A valóban végtelen ciklus természetesen értelmetlen, vagyis egyszer a végtelenített ciklusokból is ki kell lépni valahogyan. Erre két lehetőségünk van. Az egyik a már ismert `exit` parancs, ami nemcsak a ciklust, de magát a programot is befejezi. A másik a csak a ciklust megszakító `break` parancs. A `break` a ciklusmag végrehajtását azon a ponton szakítja meg, ahol meghívjuk, a program végrehajtása pedig a ciklus utáni első parancsal folytatódik:

2.5. lista

```
1: #!/bin/sh
2: # Végtelen ciklus
3:
4: i=0
5: while :
6: do
7:   i=`expr $i + 1`
8:   if [ $i -eq 10 ]
9:   then
10:    break
11:  fi
12: done
13: echo $i
```

Ha a 10. sorban a break parancsot kicseréljük exit-re, a 13. sor soha nem hajtódik végre, így az „i” változó értéke nem jelenik meg a képernyőn.

Függvények

Ha egy kódrészletet gyakran használunk, célszerű azt függvényként elkülöníteni a főprogramtól, és a megfelelő helyen a függvényhívást beilleszteni. Függvényt a következő formában adhatunk meg:

```
függvénynév()  
{  
    parancsok  
}
```

A függvények a program bármely pontjáról meghívhatók, ehhez csupán a nevüket kell leírni.

Buktató

Aki tud C nyelven programozni, valószínűleg sokszor fogja majd a

```
függvénynév()
```

formában megkísérelni egy-egy függvény meghívását. A héj számára azonban a kerek zárójelek a függvény meghatározásának (definíciójának) kezdetét jelentik.

A függvények hozzáférnek valamennyi, a főprogramban meghatározott változóhoz, és maguk is létrehozhatnak újakat. Utóbbiakat a főprogram is látni fogja, azok (más nyelvektől eltérően) nem helyiek (lokálisak) a függvényre nézve. Lássunk egy szerű példát:

2.6. lista

```
1: #!/bin/sh  
2:  # Függvények változókezelésének bemutatása  
3:  
4:  fuggveny()  
5:  {  
6:      szam1=5  
7:      szam2=7  
8:      echo $szam1, $szam2  
9:  }  
10:
```

```

11: szam1=3
12: fuggveny
13: echo $szam1, $szam2

```

Ha lefutatjuk ezt a programot, kétszer fogja kiírni az „5, 7” számpárt. Hiába adtuk szam1-nek kezdőértékül a hármat, a függvény átírta azt, mert hozzáférte ehhez a változóhoz. Az sem gond, hogy a főprogramban a szam2 sehol nem kap értéket. A függvény létrehozta azt, így a 13. sorban már „ismert” a program számára.

Buktató

Ügyeljünk arra, hogy a függvény belséjében meghatározott változók csak a függvény első meghívásakor jönnek létre. Ha tehát a fenti programban megcsereljük a 12. és 13. sort, az echo a szam2 változó helyén semmit nem fog megjeleníteni, mert ez a változó még nem létezik. Hiába előzi meg a függvény leírása a főprogramot, amíg meg nem hívjuk legalább egyszer, a héj nem tud erről a változóról.

Más szempontból a függvények a héjprogramok önálló részei. Gyakorlatilag programok a programban, hiszen önálló be- és kimenettel, visszatérési értékkel, valamint saját parancssori paraméterekkel rendelkezhetnek. A fenti egyszerű példánál maradva a szam1 és szam2 változók értékét akár a függvény parancssori paramétereként is megadhatjuk:

2.7. lista

```

1:#!/bin/sh
2:# Függvények paramétereinek bemutatása
3:
4:fuggveny()
5:{ 
6:    szam1=$1
7:    szam2=$2
8:    echo $szam1, $szam2
9:}
10:
11:szam1=$2
12:fuggveny 5 $1
13:echo $szam1, $szam2
14:echo $1, $2

```

Ha ezt a programot lefuttatjuk a következő módon

```
fv.sh 88 99
```

a kimenet így fest majd:

```
5, 88
5, 88
88, 99
```

Figyeljük meg, hogy a főprogram és a függvény számára \$1 és \$2 teljesen másat jelent! A főprogramban \$1 értéke 88, \$2-é pedig 99. Ez utóbbit felhasználjuk ugyan a 11. sorban, de a függvénynek nem adjuk át. A függvényben \$1 értéke már 5, \$2 értéke pedig 88. Ez utóbbi a főprogram \$1 paraméterével azonos, hiszen ezt adtuk át a függvény második paramétereiként a 12. sorban. A 14. sorban ismét a főprogram parancssori paraméterei jelennek meg, tehát világosan látható, hogy a függvény saját paraméterkészlettel rendelkezik, amely semmilyen módon nem befolyásolja a főprogram működését.

A függvények a programokhoz hasonlóan rendelkeznek visszatérési értékkel. Erre pontosan ugyanazok a szabályok érvényesek, mint amelyeket már megismertünk. Az egyetlen eltérés az érték visszaadásának módjában mutatkozik. A visszatérési értéket a programok és függvények egyaránt működésük végén adják át a szülőfolyamatnak. Programok esetében ez egyben a főprogram végét is jelenti, függvénynél viszont csupán a főprogramhoz való visszatérést. Ennek megfelelően a program az exit, a függvény viszont a return parancssal tudja visszaadni visszatérési értékét. A függvényen belül kiadott exit parancs továbbra is a teljes program befejeződését jelenti.

Amint azt a feltételes utasítások és ciklusok kapcsán már említettem, ezek valamennyien a logikai kifejezésben megadott program visszatérési értékét használják fel a döntés során. Ezek után nem túl meglepő, hogy logikai kifejezésben akár függvényhívások is szerepelhetnek. Lássunk egy egyszerű példát:

2.8. lista

```
1: #!/bin/sh
2:  # Függvények visszatérési értéke
3:
4:  fuggveny()
5:  {
6:      case $1 in
7:          "i") return 0;;
```

```
8:         "n") return 1;;
9:         *) exit 2;;
10:        esac
11:    }
12:
13:    if fuggveny $1
14:    then
15:      echo "Igaz!"
16:    else
17:      echo "Hamis!"
18:    fi
19:  exit 0
```

Ha ennek a programnak parancssori paraméterként egy „i” betűt adunk át, az „Igaz！”, ha egy „n” betűt, a „Hamis！” szöveget fogja kiírni. minden egyéb esetben egyszerűen leáll. A 13. sorban az if utasítás feltételeként egy függvényhívást adunk meg, amelynek gyakorlatilag kétféle visszatérési értéke lehet: 0 vagy 1. A 0 a héj számára a logikai IGAZ érték, az 1 (és minden más érték) a logikai HAMIS. A 7. és 8. sorokban megadott return utasítások a függvény visszatérési értékét határozzák meg, és csupán a függvény futásának végét jelentik. A 9. sorban szereplő exit ezzel szemben a teljes program végét jelenti 2-es visszatérési értékkel.

3

Keresés, szűrés,
szövegfeldolgozás, avagy
a szabályos kifejezések
lélektana

Mire valók a szabályos kifejezések?

Akármilyen operációs rendszerrel is legyen dolgunk, gyakran lesz szükségünk arra, hogy fájlokban egyes szavakat, vagy szövegrészleteket keressünk illetve módosítunk, néha ráadásul meglehetősen bonyolult feltételek alapján.

Az „egyéb” operációs rendszerek a keresés/helyettesítés megoldására általában kész megoldással szolgálnak: begépeljük a keresendő szót a megfelelő szövegmezőbe, leütjük az ENTER-t, és már is fut a keresés. Igen ám, de mi történik akkor, ha nem egyszerűen egy adott szót akarunk keresni, hanem mondjuk annak az összes megadott módon ragozott alakját, vagy olyan elgépeléseket, amiket gyakran követünk el. A hagyományos módszerek ezen a ponton csődöt mondanak. Nem így a UNIX.

A bonyolult esetek megoldása során nyilvánvalóan nélkülözhetetlen, hogy a keresési feltételek tömör, szabatos megfogalmazására az operációs rendszer, illetve segédprogramjai rendelkezzenek egy saját nyelvvel. A UNIX világában az ezen a nyelven megfogalmazott feltételeket nevezik szabályos kifejezéseknek (regular expression; angol rövidítése „regexp”).

A szabályos kifejezések bizonyos egészen egyszerű jellemzőket megtestesítő karakterekből állnak, amelyek gyakorlatilag tetszés szerint kombinálhatók, így tetszőlegesen bonyolult feltételrendszer adható meg velük.

Bár maga a rendszer és az azt használó segédprogramok rendkívül hatékonyak, a kezdő felhasználó számára egy szabályos kifejezéseket tartalmazó parancssor egyszerűen rémálom. Nézzünk rögtön egy frappáns példát:

```
'\(^|\ )\((*(36|06))*[- ]1[- ]*\)[0-9]*([-]*[0-9]\{3\}\{2\}\( \|$\)'
```

Nos igen... Próbaképpen talán mutassuk meg szeretteink közül néhánynak ezt a sort, és kérdezzük meg őket, mit látnak. A legtöbb valószínűleg egyszerű zagyvaléknak hiszik majd. Természetesen, ha az illető kissé humán beállítottságú és van némi képzelőereje, esetlen egy óarab siratóének „forrásnyelvű” változatának is gondolhatja, ami a megfelelő betűtípusok hiánya miatt így jelent meg a képernyőn. (Ha abszolút hallása van, talán még el is énekli.)

Kétségekivül a szabályos kifejezések jelentik a UNIX egyik legnehezebben megtanulható részét. Használatuk elsajátítása azonban elengedhetetlen, mivel az operáci-

ós rendszer nyújtotta lehetőségeket csak így tudjuk valóban hatékonyan kiaknázni. Ezért ebben a fejezetben gyakorlatilag kizárolag a szabályos kifejezésekkel esik majd szó.

Mindenekelőtt azt hiszem, el kellene árulnom a fenti „egyszerű” példa jelentését. Ez a szabályos kifejezés egy tetszőleges formátumban megadott tetszőleges budapesti telefonszám formális leírása. Hogy mire jó ez nekünk?

Képzeljük el, hogy cégünk új helyre költözött és a telefonszáma ezzel megváltozott. Képzeljük el továbbá, hogy számítógépeinken különféle iratok és iratsablonok (számlaminta, fejléces papír, névjegy, termékismertető, reklámanyag stb.) százai vagy talán ezrei találhatók, amelyek tartalmazzák a régi számot, és most természetesen módosítani kell őket.

A feladat első látásra még így sem tűnik olyan nagynak, mint amilyen valójában. Végül is – gondolja a gyanúltan rendszergazda – semmi egyebet nem kell tennünk, mint „rákeresni” a régi számra, és mindenütt helyettesíteni az újjal. Igen ám, csak hogy egy telefonszámot meglehetősen sok formában lehet megadni. Ha a régi szám mondjuk 1234567 volt, ezt a különböző iratok – attól függően, hogy ki gépelte azokat, illetve hogy milyen nyelvűek – a következő formában tartalmazhatják:

1234567
1 234 567
1-234-567
36-1-1234567
36-1-1 234 567
36-1-1-234-567
36 1 1234567
36 1 1 234 567
36 1 1-234-567
(36)-1-1234567
(36)-1-1 234 567
(36)-1-1-234-567
(36) 1 1234567
(36) 1 1 234 567
(36) 1 1-234-567
06-1-1234567
06-1-1 234 567
06-1-1-234-567
06 1 1234567
06 1 1 234 567
06 1 1-234-567
(06)-1-1234567
(06)-1-1 234 567

(06) -1-1-234-567
(06) 1 1234567
(06) 1 1 234 567
(06) 1 1-234-567

Lássuk be, így azért már nehezebb a feladat. Egy hagyományos operációs rendszer-nél esélyünk sincs rá, hogy ezeket a formákat egyetlen művelettel megkeressük. Valószínűleg egyenként kellene a kereséseket végrehajtanunk, és soha nem tudnánk, hogy éppen hol tartunk. Ha pedig ezt több könyvtárban vagy több gépen kell végigcsinálni, az akár napokig is eltarthat.

Hasonló feladattal találhatjuk szembe magunkat, ha egy helyi telefonközpont adatait akarjuk valamilyen szempont alapján feldolgozni. Ha például valamiért meg akarjuk állapítani, hogy egy adott mellékről hányszor hívtak budapesti számot, igen jó szolgálatot tesz majd a fenti példában szereplő szabályos kifejezés.

A szabályos kifejezésekre a UNIX három alapvető segédprogramja támaszkodik. Ezek a grep, a sed és az awk.

A grep egy egyszerű keresőprogram. A bemenetére érkező sorok közül csak azokat küldi ki a kimenetére, amelyek megfelelnek a megadott szabályos kifejezésnek. (Segítségével kiválogathatjuk például egy szöveg minden olyan sorát, amiben szerepel budapesti telefonszám.) Utasításformája kétféle lehet:

... | grep 'szabályos_kifejezés'

vagy

grep 'szabályos_kifejezés' feldolgozandó_fájlok_nevei

A sed ugyanúgy szabályos kifejezésekkel vezérelhető, de a bemenetét már nem csak válogatni tudja, hanem a megadott parancsok szerint módosítani is. A leggyakoribb felhasználási módja valószínűleg a különböző cserék megvalósítása (ezzel cserélhetjük le a régi telefonszámot az újra). A sed továbbra is sor alapú, de a teljes bemenetet egyben kezeli. Ezen kívül a grep-hez hasonlóan fájlból és a szabványos bemenetről érkező adatokat egyaránt képes feldolgozni. Innen ered az angol neve is: stream editor. (Magyarul „folyamszerkesztőnek” szokás hívni.)

Végül az awk egy, a C-hez igen sok tekintetben hasonló teljes programozási nyelv. Alapvetően szintén szövegfeldolgozásra fejlesztették ki, de nem hiányoznak belőle a matematikai képességek sem. A bemenetet a grep-hez hasonlóan soronként kezeli, kimenetét pedig a sed-hez hasonlóan módosítani tudja. Lényeges eltérés

ugyanakkor, hogy az awk a sorokon belül mezőket is megkülönböztet, és ezeket képes külön is kezelní. A sed egybetűs utasításaival szemben az awk programok már „értelmes” parancsokat tartalmaznak, ami a programok írását és olvasását egyaránt nagyban megkönnyíti.

E programok használatáról hamarosan még részletesen szólok. Most azonban lásduk, hogyan áll össze egy, az igényeinknek megfelelő szabályos kifejezés.

A szabályos kifejezések alapelemei

A következőkben jelentésükkel és néhány egyszerű példával együtt felsorolom a szabályos kifejezésekben használható legalapvetőbb elemeket.

„c” (tetszőleges karakter)

Ha „c” egy közönséges karakter, akkor illeszkedik önmagára. „Közönséges” karakterek számít minden, ami nem *, ., [, \,], ^ vagy \$, ez utóbbiak ugyanis a szabályos kifejezésekben különleges jelentéssel bírnak. A telefonos példánkban szereplő kifejezésben tehát a "36" egy egymás után következő hármasra és hatosra illeszkedik. Ha van egy telefon.txt nevű fájlunk, és ki akarjuk belőle válogatni az összes nemzetközi formátumban megadott magyar telefonszámot, a következőt tehetjük:

```
cat telefon.txt | grep '36'
```

(Ez a megoldás valójában még messze nem tökéletes, hiszen egy telefonszámban bárhol előfordulhat a "36" karakterpáros.)

\c

Ha a „c” karakternek valamilyen különleges jelentése van a héj vagy egy program számára, akkor a \ jel kikapcsolja ennek értelmezését, vagyis c-t „közönséges” (literálisan értelmezett) karakterré alakítja. Ha tehát valamiben mondjuk * karaktereket akarunk keresni, akkor a '*' szabályos kifejezést kell használnunk.

^

A mintát a sor elejére igazítja. Csak azok a sorok illeszkednek rá, amelyek a „^”jel utáni szabályos kifejezésnek megfelelnek, de úgy, hogy az illeszkedő rész éppen a sor elején kezdődik. (Szóköz se lehet előtte!)

\$

Jelentése hasonló a ^ karakteréhez, de ez a mintát a sor végére igazítja.

Az újsor kivételével minden karakter illeszkedik rá. Egyszerűen fogalmazva így írják UNIX-ul azt, hogy „bármi”. Egy pont csak egy karakterre illeszkedik! Ha tehát három tetszőleges karaktert akarunk keresni, akkor három pontot kell írnunk. A

```
cat fájlnév.txt | grep '...'
```

tehát a fájl minden olyan sorát megjeleníti, amely legalább három karakterből áll (nem számítva az újsort). Ha viszont olyan sorokat akarunk keresni, amelyekben három ponttal végződő mondatok vannak, azt így tehetjük meg:

```
cat fájlnév.txt | grep '\.\.\.'
```

[karakterek]

A szöglletes zárójelek közé zárt karakterek bármelyike illeszkedik rá. A szabályos kifejezésben ez a forma is csak egy karakternek felel meg, amely a zárójelben megadott értékek közül bármelyiket felveheti. A

```
cat fájlnév.txt | grep '^*[abc]'
```

parancs például azokat a sorokat szűri ki, amelyek első betűje a vagy b vagy c.

[^...]

Ez az előző forma tagadása, vagyis a szögletes zárójelek közé zárt karakterek kivételével bármelyik karakter illeszkedik rá.

[c1-c2]

A megadott tartományon belül eső karakterek bármelyike illeszkedik rá. A tartományt az angol ABC betűsorrendje szerint kell érteni. A '[0-9]' például csak számokra illeszkedik, a '[a-z]' csak kisbetűkre, a '[a-zA-Z]' pedig csak betűkre.

Két egymás után írt szabályos kifejezés szintén szabályos kifejezés. Például a '[^0-9][0-9]' sablon egy nem numerikus karaktert követő számkarakterre illeszkedik.

Jelentésmódosító jelek

A most következő elemek csak az előző szakaszban felsorolt alapelemekkel együtt használhatók, és azok jelentésének módosítására szolgálnak.

*

Ez az ismétlő karakter. A csillag előtt álló karakter akárhány előfordulása (nulla is!!!) illeszkedik rá. Ha például egy szövegből ki akarjuk válogatni azokat a sorokat, amelyekben van 1-essel kezdődő szám, a következő szabályos kifejezést használhatjuk:

```
cat fájlnév.txt | grep '1.*'
```

Emberi nyelvre lefordítva a keresési feltétel a következőképpen hangzik: „keressük azokat a sorokat, amelyekben van olyan 1-es, amit tetszőleges karakterből (.) tetszőleges számú (*) követ, vagy esetleg egy sem.”

+

Ez is ismétlő karakter, de megköveteli, hogy az őt megelőző karakter vagy kifejezés legalább egyszer előforduljon. (Emlékezzünk rá, hogy a * a nulla előfordulást is

megengedi.) Ha az olyan sorokat akarjuk kiválogatni egy szövegből, amelyek csupa egyesből álló számokat tartalmaznak, így tehetjük meg:

```
cat fájlnév | grep '1\+'
```

Hogy mit keres ott az a „\” karakter?! Olvassuk el a következő BUKTATÓ-t!

(...)

A kerek zárójelek segítségével csoportokat képezhetünk a szabályos kifejezések-ből. A programok a zárójelbe tett csoportokat egy egységeként kezelik, és a jelentésmódosító jelek is a teljes csoportra érvényesek. Ha például olyan sorokat keresünk egy szövegben, amelyekben az "ab" betűpáros legalább egyszer előfordul, a következő szabályos kifejezést kell használnunk:

```
cat fájlnév.txt | grep '\(ab\)\+'
```

Itt már egészen tekintélyes mennyiségű fölöslegesnek tűnő „\” szerepel (lásd a BUKTATÓ-t).

{x} vagy {x,} vagy {x,y}

Ezzel az operátorral az illeszkedések pontos számát adhatjuk meg. Ha a kapcsos zárójelek között csak egy szám szerepel, akkor *pontosan* ennyiszeres illeszkedést várunk. Ha a szám után egy vessző is van, akkor *legalább* ennyiszeres az illeszkedés. Végül ha két számot adunk meg vesszővel elválasztva, akkor legalább az első, de legfeljebb a második által meghatározott többszörözés érvényes. Íme néhány példa:

```
cat fájlnév.txt | grep '\.\{3\}'
```

Pontosan három egymást követő pontot tartalmazó sorok válogatása (befejezetlen mondatok).

```
cat fájlnév | grep '\.\{4,\}'
```

Legalább négy, egymást követő pontot tartalmazó sorok válogatása. (Úrlapon kitöltenődő mezők.)

```
cat fájlnév.txt | grep '[0-9]\{3,7\}'
```

Legalább három, de legfeljebb héjtígyű számokat tartalmazó sorok kiválogatása.

Megjegyzés

Vegyük észre, hogy a '\{0,\}' jelentése azonos a '*' jelentésmódosító jelével (nulla vagy több illeszkedés), a '\{1,\}' pedig a '+' megfelelője (egy vagy több illeszkedés). A '\{1\}' jelölésnek (pontosan egy illeszkedés) nincs ilyen egyszerű párja.

szabályos kifejezés 1. | szabályos kifejezés 2.

Ez a szabályos kifejezésekre vonatkozó logikai VAGY művelet. Bármire illeszkedik, ami vagy az egyik, vagy a másik mintának megfelel. Természetesen kettőnél több minta is megadható. Ha mondjuk az "asztal" szó néhány ragozott alakját akarjuk kikeresni egy szövegből, a következőképpen járhatunk el:

```
cat fájlnév.txt | grep 'asztal\ (on\|ban\|hoz\|nak\)'
```

Szabályos kifejezésekben természetesen nincs a szó klasszikus értelmében vett logikai ÉS művelet, hiszen két különböző minta nem illeszkedhet ugyanarra a dologra.



Klikapcs, BEkapcs ...

A kezdeti UNIX-felhasználók már az első órán megtanulják, hogy a * a héj számára különleges karakter. A könyvtárunkban található összes fájl listáját jelenti, amit a héj végrehajtáskor behelyettesít oda, ahová a csillagot írtuk.

Aztán megtanulják, hogy az echo parancsal tetszőleges szöveget megjeleníthetünk a képernyőn. Itt már akad egy kis galiba, mert ha valamilyen különleges karaktert – mondjuk egy csillagot – is tartalmaz az a szöveg, a héj itt is elvégzi a behelyettesítést. De semmi gond! Annyi csak a dolgunk, hogy írunk előre egy \ karaktert, és ezzel Klikapcsoljuk a * különleges jelentését. A Nyájas Felhasználó tehát megtanulta a leckét: a \ karakter Klikapcsolásra való.

Aztán elkövetkezik a második óra, amikor is a szabályos kifejezések-ről esik szó. Itt is vannak különleges karakterek, egészen pontosan a következők: \, ^, \$, *, [és]. Előfordulhat az is, hogy mi juszt is * karaktereket akarunk valamiben keresni a grep segítségével, de ismét semmi gond! Elég a már ismert módszert alkalmazni: '*'. A felhasználó boldog, mert látja, hogy bár bolond beszéd ez, de van benne rendszer. A \ a szabályos kifejezésekben ugyanazt jelenti, mint a héj számára: Klikapcsolja a különleges jelentést.

És akkor jön a meglepetés.

Felhasználónk elhatározza, hogy egy szövegből kikeresi az összes olyan sort, amiben legalább három egymást követő egyes van. Az elhatározást tett követi és a következő szabályos kifejezés fogalmazódik meg:

```
cat fájlnév.txt | grep '1{3,}'
```

És a képernyőn nem jelenik meg semmi... Még hibaüzenet sem... Már most Felhasználó pontosan tudja, hogy van abban a szövegben több olyan sor is, ami megfelel a megadott keresési feltételnek. Akkor viszont valahol valami hibának kell lenni.

És valóban. A { , } , (,) , | és + karakterek (tehát a * kivételével az összes jelentésmódosító jel) a szabályos kifejezésekben ugyan különleges jelentéssel bírnak, de ha egyszerűen leírjuk őket, akkor a grep alapértelmezésben betű szerinti (literális) formában értelmezi valamennyit.

Felhasználónk tehát az előbb azért nem járt sikерrel, mert a grep szó szerint az "1{3,}" karakterSORozatot kereste a szövegben, és természetesen nem talált belőle egyet sem. (Elvégre „normális” ember ritkán ír ilyeneket). Amúgy a parancs formailag teljesen helyes volt, tehát hibaüzenet sem jelent meg.

A jelentésmódosító karakterek esetében (a * kivételével) a különleges jelentést BEkapcsolni kell és nem KI. És most jön az igazi meglepetés: a BEkapcsolásra is a \ karakter szolgál. A szabályos kifejezés helyes formája tehát:

```
cat fájlnév.txt | grep '1\{3,\}'
```

Aztán ha megtanultuk, hogy melyik különleges karaktert kell be- és melyiket kikapcsolni, akkor már csak arra kell figyelnünk, hogy az elmondottak csak a grep és a sed parancsokra érvényesek. Az awk az összes különleges karaktert eredeti, különleges jelentése szerint értelmezi, ott tehát a \ tényleg csak kikapcsolásra szolgál.

A szabályos kifejezések és a héj

Ha már a buktatóknál tartunk, ne felejtük el azt sem, hogy a szabályos kifejezésekben használt karakterek némelyikét a héj is értelmezi (ráadásul teljesen máshogyan). Így héjprogramokban a kívánt hatást megfelelő idézőjelek használatával érhetjük el. Ezek segítségével a szükséges mértékig elrejthetjük a héj elől a számára is „kedves” karaktereket.

Ha a szabályos kifejezést egyszeres idézőjelek (' . . . ') közé zárjuk, a héj változatlanul adja át azt annak a programnak, amelyiknek szántuk. Ilyenkor természetesen arra sincs lehetőségünk, hogy egy héjváltozó értékét felhasználjuk a szabályos kifejezés részeként.

Ha kétszeres idézőjeleket (" . . . ") használunk, a különbség csupán annyi, hogy a héj először megnézi, van-e hivatkozás valahol héjváltozóra, illetve van-e olyan le nem védett különleges karakter, amely behelyettesítést kíván. Ha talál ilyet, kiértékeli azt és így adja át a szabályos kifejezést a programnak.

Végül ha visszafelé döntött egyszeres idézőjelet (` . . . `), vagyis parancsbehelyettesítést használunk, a héj parancsként értelmezi a karaktersorozatot, lefuttatja azt, és az eredményt teszi vissza a helyére. A héj számára értelmes karaktereket természetesen ilyenkor is le kell védenünk.

Példaként lássuk, hogyan kereshetjük ki egy telefonszámokat tartalmazó listából a nemzetközi formátumban megadott magyar számokat, vagyis azokat, amelyek "36"-tal, vagy "(36)"-tal kezdődnek:

3.1. lista

```

1: #!/bin/sh
2: # Nemzetközi számok kikeresése
3:
4: szam=36
5:
6: # Teljesen védett forma
7: cat telefon.txt | grep '(*36)*'
8:
9: # Részleges védelem (változó behelyettesítése)
10: cat telefon.txt | grep "(*$szam)*"
11:
12: # Idézőjelek nélkül
13: cat telefon.txt | grep \(\*36\)\*
14:
15: # Parancsbehelyettesítéssel
16: cat telefon.txt | grep `echo \(\*36\)\*`
```

Figyeljük meg, hogy parancsbehelyettesítésnél (16. sor) pontosan ugyanolyan védelmet kell alkalmaznunk, mintha egyáltalán nem írtunk volna semmilyen idézőjel (13. sor).

A szabályos kifejezések és a mosogép

Avagy hogyan írunk szabályos kifejezéseket tartalmazó programot

Most, hogy már ismerjük a szabályos kifejezések építőelemeit, ideje kipróbálni tudásunkat. Első gyakorlatként talán kíséreljük meg „visszafejteni” a fejezet elején bemutatott „elrettentő példát”:

```
'\(^|\ )\((*\|(36\|06\))*[- ]1[- ]\)*[0-9]\([- ]*[0-9]\{3\}\)\{2\}\(\ \|$\)'
```

Lássuk csak, hogy is állt össze ez az első látásra szerfölött zavarosnak tűnő szerkezet.

Bár a szabályos kifejezések írása a héjprogramok fejlesztése során csupán eszköz, tulajdonképpen ez is programozás. Olyan, mint amikor beállítunk egy automata mosogépet. Vannak bizonyos alapelemek, amelyekből építkezhetünk (víz beszívása, melegítés, dob forgatása, centrifugálás) és van egy kódrendszer, amivel az egyes műveletek sorrendjét megadhatjuk (tárcsa tekergetése, gombok nyomogatása).

Mármost bármilyen programot kezdünk is el fejleszteni, először pontosan tudnunk kell, hogy mit akarunk megvalósítani. A szabályos kifejezések megírása előtt azzal kell tisztában lennünk, hogy egészen pontosan hogy néz ki az a dolog, amit a szabályos kifejezés segítségével azonosítani akarunk. Első lépésként foglaljuk tehát össze, milyen elemekből tevődik, illetve tevődhet össze egy budapesti telefonszám, és milyen logikai szabályszerűségek figyelhetők meg a szerkezetében.

- Egy budapesti telefonszám biztosan tartalmaz legalább hét számjegyet. Ezek értékére, illetve sorrendjére nézve semmiféle további támpontunk nincs.
- A hét számjegy kétféleképpen csoportosítható: megadhatóak folytonosan, illetve 1+2x3 formában. A többi formát az egyszerűség kedvéért nem nézzük.
- Tagolás esetén a csoportokat szóköz vagy kötőjel választhatja el.
- A telefonszám megadható nemzetközi formátumban is. Ilyenkor szerepel benne az országkód, amely Magyarország esetében 36, és a körzetszám, amely Budapest esetében 1.
- Az országkód és a körzetszám csak együtt fordulhat elő, de megadásuk nem kötelező.
- A telefonszám megadható „vidéki” formátumban is. Ilyenkor szerepel benne a távolsági hívás kódja, amely Magyarországon 06, és a körzetszám, amely Budapest esetében 1. Ezek ismét csak együtt fordulhatnak elő, de megadásuk nem kötelező.

- A távolsági hívás kódját, illetve az országkódot egyesek zárójelbe szokták tenni, de ez sem kötelező.
- A telefonszámot megelőző két területi kódot szóköz vagy kötőjel választja el egymástól.
- Telefonszámot nem szokás semmi mással egybe írni, így előtte és utána is van legalább egy szóköz vagy írásjel.

Ha kezünkben a fenti lista, neki is láthatunk a szabályos kifejezés megfogalmazásának. Héjváltozót - legalábbis egyelőre - nem akarunk benne használni, ezért a félreértések elkerülése végett az lesz a legegyszerűbb, ha az egész mintát egyszeres idézőjelek közé zárjuk.

Kezdjük magával a telefonszámmal. Ez biztosan számjeggyel kezdődik, tehát a következőt írhatjuk.

' [0-9] '

Az első számjegyet szóköz vagy kötőjel követheti:

' [0-9] [-] '

Vannak azonban esetek, amikor a szám folytonos, tehát a második karakterpozíció rögtön a második számjegy következik. A szóköz vagy kötőjel keresését tehát „nem kötelezővé” kell tennünk, vagyis meg kell engednünk, hogy „nulla darab” legyen belőlük. A szabályos kifejezésekben éppen erre szolgál a * karakter:

' [0-9] [-] * '

Ez tehát azt jelenti, hogy keresünk egy számjegyet, amit egy szóköz vagy egy kötőjel követ, vagy a kettő közül egyik sem.

Ezután akármilyen formában szerepel is a telefonszám, biztosan három számjegy következik:

' [0-9] [-] * [0-9] \{3\} '

Ne felejtsük el bekapcsolni a kapcsos zárójelek különleges jelentését!

Ezután ismét következik egy „nem kötelező” szóköz vagy kötőjel, aztán megint három számjegy. Megtehetnénk tehát, hogy a két utóbbi lépést egyszerűen megismétljük és még egyszer leírjuk ezt a mintát:

' [0-9] [-] * [0-9] \{3\} [-] * [0-9] \{3\} '

Ez jelen esetben különösebben nem kényelmetlen, viszont van rá egy általánosan használható és sokkal elegánsabb megoldás is. Ha egy ilyen összetett minta több előfordulását keressük, a kerek zárójelek segítségével képezhetünk az alapmintából egy csoportot, majd megadhatjuk az ismétlődések számát:

```
'[0-9]\([-]*[0-9]\{3\})\{2\}'
```

Ismét nem szabad megfeledkeznünk a kerek zárójelek különleges jelentésének bekapcsolásáról.

A most elkészült minta tehát *legalább* hét számjegyre illeszkedik, amelyek között a telefonszámok szokásos tagolásának megfelelően két szóköz vagy két kötőjel is lehet. A hangsúly itt a „legalább” szón van, ez a minta ugyanis a szabályos kifejezésekre vonatkozó „játékszabályok” szerint illeszkedik minden hétnél több jegyet tartalmazó számra is. (Ezek is „tartalmaznak” ugyanis egy hétjegyű számot.)

A feladat megoldása viszonylag egyszerű: meg kell követelnünk, hogy a hetedik számjegy után szóköz következzen:

```
'[0-9]\([-]*[0-9]\{3\})\{2\}'
```

Ez a forma már majdnem tökéletes, de van még egy-két buktató. Ha a hétjegyű telefonszám éppen a sor végére került, akkor nincs utána szóköz. Ha pedig nincs szóköz, akkor a minta nem illeszkedik rá, hiszen feltétel nélkül megköveteltük egy darab szóköz meglétét az utolsó számjegy után.

A gond orvoslására az első ötletünk nyilván az lesz, hogy a * segítségével nem kötelezővé tegyük a szóköz jelenlétét:

```
'[0-9]\([-]*[0-9]\{3\})\{2\}*' 
```

Ez sajnos nem jó, így ugyanis azt is megengedjük, hogy tetszőleges helyzetben (tehát nem csak a sor végén) ne legyen szóköz, így a minta megint illeszkedni fog a hétnél több jegyű számokra is. Az igazi megoldás az, ha szóközt VAGY sorvéget keresünk. Utóbbi azonosítására a szabályos kifejezésekben a \$ jel szolgál:

```
'[0-9]\([-]*[0-9]\{3\})\{2\}\(\|\$\)' 
```

Látható, hogy megint be kellett kapcsolnunk a csoportosítást végző kerek zárójelek és a logikai VAGY művelet jelének különleges jelentését. A \$ jel viszont magában szerepelhet, mivel ez alapértelmezés szerint is különlegesnek minősül.

Most lássuk külön az esetlegesen előforduló előtag formális leírását. Ez vagy 36-tal, vagy 06-tal kezdődik, tehát ismét minták csoportját kell meghatároznunk:

```
' \ (36 \| 06 \| ) '
```

Ez a kód vagy kerek zárójelben szerepel, vagy nem, tehát előtte és utána meg kell adnunk egy nem kötelező nyitó és egy szintén nem kötelező záró kerek zárójelet:

```
' (* \ (36 \| 06 \| )) * '
```

Itt nincs \, hiszen valóban kerek zárójelről, mint keresendő karakterről van szó. Az ország- vagy területi kódot – ha szerepelt egyáltalán – szóköz vagy kötőjel követi. Ezután mindenkorban egy egyes következik, majd ismét egy olyan szóköz vagy kötőjel, amely csak akkor szerepel, ha szerepelt az előző egyes. A következő minta csak részben felel meg ennek a meglehetősen összetett feltételrendszernek:

```
' (* \ (36 \| 06 \| )) * [- ] 1 [- ] '
```

Azért csak részben, mert *feltétlenül* megköveteli a "06 1" vagy a "36 1" előtag valamilyen formában való meglétét. A mi célunk azonban a teljes előtag nem kötelezővé tétele, vagyis az, hogy a „nulla darab” előfordulást is megengedjük. Ezt úgy tehetjük meg, ha a teljes előtagból a kerek zárójelek segítségével egy csoportot készünk, és erre alkalmazzuk a * operátort:

```
' \(( * \ (36 \| 06 \| )) * [- ] 1 [- ] \) * '
```

Mivel két egymás mellé írt szabályos kifejezés maga is szabályos kifejezés, nincs más dolgunk, mint egyesíteni az előtag és a telefonszám leírására megadott mintákat:

```
' \(( * \ (36 \| 06 \| )) * [- ] 1 [- ] \) * [0-9] \([- ] *  
[0-9] \{3\} \) \{2\} \(\ \| \$\)' 
```

Előzetes felsorolásunkban szerepel az a kitétel, hogy telefonszámot nem írunk egybe semmi mással. Ez azt jelenti, hogy a fenti mintát legalább egy szóköz előzi meg. Amint azonban azt a szám utáni szóköz megadásánál már láttuk, itt is előfordulhat, hogy a sor éppen magával a telefonszámmal kezdődik. Ilyenkor nincs szóköz, de a ^ jellel azonosítható a sor kezdete. A végső forma tehát így fest:

```
' \(^ \| \ ) \(( * \ (36 \| 06 \| )) * [- ] 1 [- ] \) * [0-9] \([- ] *  
[0-9] \{3\} \) \{2\} \(\ \| \$\)' 
```

És kész is a mosási program!

4

A sed használata héjprogramokban

Ebben a fejezetben a sed-del végezhető alapvető műveletekről esik szó. Nem fogom ugyan érinteni az összes létező nyelvi elemet, de a legtöbbször használható műveleteket példákkal és a kezdő felhasználóra leselkedő hibalehetőségekkel együtt bemutatom.

A sed működési elve

A sed tulajdonképpen egy programozható szövegszerkesztő, amely egybetűs parancsok és szabályos kifejezések segítségével vezérelhető. Bemenetét veheti fájlból, de igazi erőssége és haszna abban a képességében rejlik, hogy a felhasználó által megadott program alapján a szabványos bemenetére érkező szöveget is képes feldolgozni. Ez lehetővé teszi, hogy a sed-et egy bonyolult adatfeldolgozási sor egyik elemeként használjuk, anélkül, hogy a kimenetét akár csak átmenetileg fájlban kellene tárolnunk. Röviden szólva tehát a sed egy olyan programozható szövegfeldolgozó, amely az adatokat képes „röptében” kezelni.

A fentieknek megfelelően a sed-et kétféle formában hívhatjuk meg:

... | sed 'program'

vagy

sed 'program' feldolgozandó_fájlok_nevei



Számos esetben a sed segítségével feldolgozott fájlra már nincs is szükségünk, csak a kimenetre. Ilyenkor csábító a gondolat, hogy bemenetként és kimenetként ugyanazt a fájlnevet adjuk meg, valahogy így:

sed 'program' szoveg.txt > szoveg.txt

A gyanútlan felhasználó nyilvánvalóan arra számít, hogy ezzel a módszerrel közvetlenül visszairányíthatja az eredményt oda, ahonnan a bemenet jött. Ez az a csábítás azonban, aminek SOHA nem szabad engednünk. Ha ugyanis megtesszük, az eredmény – minden várakozásunkkal ellentétben – egy nulla bájt hosszúságú fájl lesz. A magyarázat egyszerű: átirányításkor a héj első dolga az, hogy nullára állítja a kimeneti fájl hosszát. És mivel esetünkben ez egyben a bemenet is, a sed-nek már nem lesz mit kiolvasni.

A dolog amúgy formailag helyes, tehát hibaüzenetet nem kapunk. Viszont ha egy ciklussal sikerül végrehajtanunk egy ilyen trükkös feldolgozást egy könyvtár teljes tartalmára, akkor kész a katasztrófa.

A feldolgozó programot nem kell feltétlenül a parancssorban megadni, a sed veheti azt egy fájlból is. Ilyenkor a fájl nevét a -f kapcsoló után kell megadni:

```
... | sed -f programfájl
```

A sed a grep-hez hasonlóan sor alapú, vagyis a bemenetét újsor karaktertől újsor karakterig olvassa, és a megadott műveleteket ezeken a darabokon hajtja végre.

Az éppen beolvasott és feldolgozás alatt álló sort saját belső tárában, az úgynevezett mintatérben tárolja. A mintatér tehát (az esetek túlnyomó többségében) egyetlen sornyi szöveget tartalmaz a bemenetből.

A feldolgozási programot egybetűs parancsok és szabályos kifejezések formájában adhatjuk meg. Egy program több különböző utasítást is tartalmazhat külön sorban. Ezeket a sed a megadott sorrendben hajtja végre a mintatér tartalmán, majd ha végzett, az eredményt a szabványos kimenetére küldi.

A szöveg sorai alapértelmezés szerint akkor is a szabványos kimenetre kerülnek, ha a program semmilyen változtatást nem végzett rajtuk. A sed tehát amolyan értelmes „átfolyóként” viselkedik: ha kell, elvégzi a megadott módosításokat, ha pedig erre nincs utasítása, egyszerűen továbbküldi a beérkezett anyagot.

Egy sed program egy sora teljes általánosságban a következőképpen fest:

```
<cím1>,<cím2> parancs
```

<cím1> és <cím2> egy-egy szám vagy szabályos kifejezés lehet. A programnak ez a sora a szövegnek csak azokra a soraira hajtódiik végre, amelyekre illeszkedik a megadott szabályos kifejezés. A

```
/ [0-9] / parancs
```

programsor például csak azokat a szövegsorokat fogja érinteni, amelyekben van legalább egy számjegy, míg a

```
25 parancs
```

csak a szöveg 25. sorát érinti. Figyeljük meg, hogy a szabályos kifejezések két / karakter között kell megadni, a számokat azonban nem.

TIPP

A fájl utolsó sorát a \$ szimbólummal címezhetjük meg egyszerűen:
cat fájlnév | sed -n '\$p'

a fájl utolsó sorát jeleníti meg.

Megadhatunk vesszővel elválasztva két címet is. A sed ilyenkor a két címnek megfelelő sorok közti tartományon hajtja végre a programot. Ha sorszámokat adtunk meg, akkor a művelet egyértelmű. Ha azonban szabályos kifejezéseket használtunk, a sed a beérkező szöveg összes olyan tartományán végre fogja hajtani a műveletsort, amire a megadott kifejezések illeszkednek.

Buktató

Mint azt már említettem, a feldolgozandó szövegnek nem feltétlenül kell a szabványos bemeneten át érkeznie. Megadhatunk a parancssorban is egy vagy több fájlnevet. Arra azonban ügyeljünk, hogy több fájl feldolgozásakor a sed belső sorszámlálója nem nullázódik, amikor a következő fájl feldolgozásába kezd. Ha tehát három százsoros fájlt akarunk egyszerre feldolgozni, a második szöveg első sora nem az 1-es, hanem a 101-es számú lesz.

Egy cím vagy címtartomány egynél több parancsra (vagyis egy egész programblokkra) is vonatkozhat. Ilyenkor az összetartozó programsorokat kapcsos zárójelekkel kell megadnunk:

```
<cím1>,<cím2> {
parancs1
parancs2
parancs3 ...
}
```

A sed alapvető parancsai

A következőkben a sed alapvető parancsait és a jelentésmódosító kapcsolókat ismertetem néhány egyszerű példával együtt.

p (Print)

E parancs hatására a sed kiküldi a szabványos kimenetére a mintatér pillanatnyi tartalmát. Mivel a feldolgozási program végrehajtása után ezt magától is megtenné, ha nem bíráljuk felül az alapértelmezett viselkedést, minden sor kétszer fog megjelenni, esetleg két különbözőképpen feldolgozott formában. A szabványos kimenetre írást a -n kapcsolóval tilthatjuk le.

Az alábbi program hatására a telefon.txt fájl minden sora megkettőzve íródik ki a képernyőre:

```
cat telefon.txt | sed 'p'
```

A következő minden sort csak egyszer jelenít meg, az ötödiket azonban kétszer:

```
cat telefon.txt | sed '5p'
```

Ha csak az ötödik sort akarjuk megjeleníteni, a következőképpen tehetjük meg:

```
cat telefon.txt | sed -n '5p'
```

A fájl sorainak egy tetszőleges tartományát is kiírhatjuk, ha két címet adunk meg:

```
cat telefon.txt | sed '1,10 p'
```

Ez a parancs az első tíz sort jeleníti meg, akárcsak a UNIX head nevű segédprogramja.

d (Delete)

Ez a parancs törli a mintatér tartalmát. Alapvetően ugyanazok a szabályok érvényesek rá, mint a p parancsra. Ha például a telefonlista 3-7. sorait akarjuk törölni, így tehetjük meg:

```
cat telefon.txt | sed '3,7 d'
```

s (Substitute)

Valószínűleg ez a leggyakrabban használt sed parancs. Hatására a mintatér tartalmának egy szabályos kifejezésen keresztül megadott része egy másik karakterláncra cserélődik. Utasításformája a következő:

```
s/mit_cserélünk/mire_cseréljük/
```

Mint minden sed parancsnál, természetesen itt is használható egy vagy két szabályos kifejezés a megfelelő sor vagy sorok címzésére, illetve megadhatunk néhány, a működést befolyásoló kiegészítő parancsot is.

A fenti példában a „mit_cserélünk” és a „mire_cseréljük” rész két tetszőleges szabályos kifejezés lehet. A parancs hatására a sed az összes sorra „rápróbálja” az első szabályos kifejezést, és illeszkedés esetén a mintának megfelelő részt (de csak azt!) helyettesíti a második kifejezéssel. Ha egy sorban több illeszkedő rész is lenne, akkor is csak az első ilyet cseréli le. Ha a minta összes előfordulását helyettesíteni akarjuk, a g módosító kapcsolót kell használnunk:

```
s/mit_cserélünk/mire_cseréljük/g
```

Ez a parancs már valóban a teljes bemeneten elvégzi a megadott helyettesítést, akkor is, ha soronként több megfelelő hely van.

Előfordulhat, hogy a keresett mintának csak egy adott előfordulását akarjuk lecserélni valami másra. Ilyenkor szintén módosító jelzőként megadhatjuk, hogy hányadik előfordulást akarjuk cserélni:

```
s/mit_cserélünk/mire_cseréljük/27
```

Ez a parancs a keresett mintának csak a huszonhetedik előfordulásánál végzi el a cserét. A többszöri előfordulás természetesen itt is egy soron belül értendő! A sed legfeljebb 512-ig követi az előfordulásokat, de ez józan emberi számítás szerint nem jelenthet akadályt senkinek. Az s parancs törlésre is használható. Ha törölni akarjuk a szabályos kifejezésnek megfelelő részt vagy részeket, akkor „semmi-vel” kell helyettesítenünk azokat:

```
echo "1234567abcde" | sed 's/[a-z]//'
```

Buktató

Mi hány, mi mennyi...?

Gyakran előfordul, hogy egy sornak több olyan átfedő részlete is van, amelyekre egy szabályos kifejezés illeszkedhet. Nézzünk egy egyszerű példát:

```
echo "1234567abcde" | sed 's/[a-z]\+/X/g'
```

Itt egy olyan részletet keresünk, amely egy vagy több kisbetű tartalmaz (\+). Ennek a feltételnek elvileg egyetlen kisbetű is megfelel, tehát azt várnánk, hogy a sed öt illeszkedést talál majd, és a g módosító hatására minden ötször elvégzi a helyettesítést. A várt eredmény tehát az

```
"1234567XXXX"
```

karakterlánc lenne.

Ha azonban kipróbáljuk a fenti parancsot, láthatjuk, hogy csak egyetlen „X” jelent meg a számsor végén, vagyis csak egy illeszkedés volt. Ez pedig azt jelenti, hogy a sed a lehetséges illeszkedések közül mindenig a „legtágabbat” választja ki. Ezt az alapszabályt, amely egyébként valamennyi, a szabályos kifejezésekre támaszkodó programra érvényes, nem árt észben tartani a kifejezések megfogalmazásakor.

Ha már itt tartunk, próbáljuk ki a „nulla vagy több” (*) operátor hatását is:

```
echo "1234567abcde" | sed 's/[a-z]*$/X/g'
```

Az eredmény első látásra több mint meglepő:

```
X1X2X3X4X5X6X7X
```

Ezt a hatást a g kapcsoló és a * operátor együttesen váltják ki. A * a „nullaszor” való illeszkedést (magyarul a nem illeszkedést) is megengedi. A sed a sort karakterenként elkezdi végignézni. Rögtön a sor elején (tehát még az 1-es észlelése előtt) azt látja, hogy nem lát semmit, ami pontosan nulla darab illeszkedése a megadott szabályos kifejezésnek. A feltétel teljesült, tehát boldogan behelyettesíti az X-et.

Ezután hét darab számjegy következik. A sed egyenként végigolvassa őket, minden egyik után megáll, és eltöpreng azon, hogy amit látott, az nulla darab betű volt-e. Természetesen minden nyiszor úgy fogja találni, hogy ami szám, az nem betű, tehát nulla darab betűt látott. Mivel a g kapcsolót használtuk, minden nyiszor el kell végeznie a behelyettesítést, valahányszor igaznak találja a feltételt.

Végül eljut a betűkig. Itt már a „legtágabb illeszkedő tartomány” elve érvényesül, vagyis az öt betű helyett egyetlen X-et fog kinyomtatni.

Összefoglalva tehát a szabályos kifejezések „illesztésével” kapcsolatban a következő általános szabályok fogalmazhatók meg:

1. A szabályos kifejezés a bemenetnek mindenkor a lehető legtágabb tartományára illeszkedik.
2. A „legtágabb illeszkedés elve” a logikai NEM művelettel előállított szabályos kifejezésekre is érvényes („a legtágabb nem illeszkedés elve”). A ' [^0-9]' kifejezés például a bemenetnek a legtágabb, csak betűket tartalmazó részére illeszkedik.
3. Ha a „nem illeszkedést” a * operátor segítségével engedjük meg (pl. ' [0-9]*'), akkor a legtágabb illeszkedés (Pontosabban „nem illeszkedés”) elve nem érvényesül.

Bizonyos helyettesítési műveletek során szükségünk van arra a szövegrészre, amelyre a helyettesítést vezérlő szabályos kifejezés illeszkedett. A legegyszerűbb példa erre talán az, amikor egy sor előtt vagy mögött akarunk elhelyezni valamelyen karakterláncot:

```
echo "1234567abcde" | sed 's/.*/XXX&YYY/'
```

A kimenet itt az

```
XXX1234567abcdeYYY
```

karakterlánc lesz, az „&” műveleti jel ugyanis a vezérlő szabályos kifejezés legutóbbi illeszkedését jelenti. Mivel itt „bármilyen karakterből bármennyit” (".*") kerestük, a „legtágabb illeszkedés” elve alapján a teljes sort kijelöltük.

Az „&” művelet működésével kapcsolatban igen fontos, hogy mindenkor a *legutóbbi* illeszkedő szövegrésznek felel meg. Nézzünk egy egyszerű, de szemléletes példát:

```
echo "abc1234def" | sed 's/[a-z]\+/X&Y/'
```

Itt a kimenet az

```
XabcY1234def
```

karakterlánc lesz, mivel a g módosító kapcsoló hiányában az s parancs csak a szabályos kifejezés első illeszkedésére vonatkozott. Ha azonban megadjuk a g kapcsolót:

```
echo "abc1234def" | sed 's/[a-z]\+/X&Y/g'
```

akkor a kimenet már az

```
XabcY1234XdefY
```

karakterlánc lesz, vagyis az „&” művelet a második illeszkedésnél is tette a dolgát.

Amint azt az előző fejezetben már láttuk, a szabályos kifejezések kerek zárójelek használatával tetszőlegesen tagolhatók. Így egy kifejezést tulajdonképpen több kisebb egységből állíthatunk össze, ami első közelítésben azért hasznos, mert a jelen tésmódosító jelek (pl. * vagy +) ezekre egyben vonatkoznak. A sed arra is lehetősséget ad, hogy a helyettesítések során a vezérléshez használt szabályos kifejezés egyes alegységeire egyenként hivatkozzunk. Tegyük fel például, hogy az előbbiekn

ben már többször használt számjegyekből és betűkből álló szövegben meg akarjuk cserélni a két rész sorrendjét. Kihasználva, hogy a megcserélni kívánt részek jól el-különülnek, a feladatot a következőképpen oldhatjuk meg:

```
echo "1234567abcdef" | sed 's/\([0-9]\+\)\(([a-z]\+\)/\2\1/'
```

A kimenet valóban

```
abcdef1234567
```

lesz, ugyanis \1 azt a szövegrészt jelenti, amelyre az első, \2 pedig azt, amelyikre a második alkifejezés illeszkedett. A számozásban egészen kilencig mehetünk ezzel a módszerrel, arra azonban ügyelnünk kell, hogy a kerek zárójelek különleges jelentését a \ segítségével mindenkorú bekapcsoljuk.

a (Append), i (Insert) és c (Change)

Az a parancssal egy vagy több sornyi szöveget fűzhetünk a mintatér tartalmához. Az i parancs ugyanezt teszi, de a sorokat a mintatér tartalma elő szűrja be.

Nagyon fontos azonban megemlíteni, hogy ezek a parancsok valójában nem módosítják magát a mintateret, hatásuk csupán a kimeneten jelentkezik! Amit az a vagy i parancssal adtunk meg, arra a szövegrészre a program további parancsai nem vonatkoznak. Formájuk a következő:

```
<cím> a\
szöveg
```

vagy

```
<cím> i\
szöveg
```

Szintén lényeges, hogy a hozzáadott szövegek a programon belül mindenkorúban új sorban kell kezdődni, és a kimeneten is új sorban fog szerepelni. Az a és i parancsok előtt csak egyetlen cím adható meg, vagyis ezek egész címtartományokra nem alkalmazhatók. Nézzünk egy egyszerű példát:

```
echo "1234567abcd" | sed 'i\
Csak számok:
s/[a-z]\+//'
```

Ennek a parancsnak a kimenete

Csak számok:
1234567

lesz, tehát az i parancsal beszúrt szöveget a későbbi törlés ném érintette.

A c parancs a mintatér tartalmát a megadott karakterláncra cseréli. Utasításformája teljesen azonos az a és i parancsokéval, és az is igaz rá, hogy neve ellenére a mintateret ez sem változtatja meg. A program későbbi parancsai nincsenek hatással az így megadott szövegre, az a kimeneten mindenkorábban változatlan formában fog megjelenni:

```
echo "mindegy mi ez" | sed 'c\  
1234567abcde  
s/[0-9]\+//'
```

A kimenet itt tehát az

```
1234567abcde
```

karakterlánc lesz, annak ellenére, hogy az s parancsnak elvileg törölnie kellett volna az összes számjegyet.

y (Transform)

Ez a parancs karaktereket cserél más karakterekre két megadott minta alapján. Utasításformája a következő:

```
<cím> y/helyettesítendő lista/helyettesítő lista/
```

Lássunk rögtön egy példát:

```
echo "abcdabcdXYZ" | sed 'y/abcd/xyzk/'
```

A kimenet az

```
xyzkxyzkXYZ
```

karakterlánc lesz. Látható tehát, hogy az y parancs a teljes sorra végrehajtódik, nem csak a karakterek első előfordulásaira. A helyettesítendő és a helyettesítő listának szigorúan azonos hosszúságúnak kell lennie, ellenkező esetben ugyanis hibaüzenetet kapunk.

A sed y parancsa tulajdonképpen pontosan ugyanazt teszi, mint a tr nevű UNIX ségédprogram. Ugyanakkor lényeges eltérés, hogy az y listáinak megadásakor nem használhatunk karaktertartományokat. Míg a tr-nél a szöveg nagybetűssé alakítása például megoldható az [a-z] és [A-Z] listák megadásával, a sed y parancsa ezeket szó szerint értelmezi, tehát csak az „a” és „z” karaktereket fogja nagybetűre cserélni.

w (Write), r (Read)

A mintatér tartalmát a megadott fájlba írja. A program befejezése után a sorok ettől függetlenül a szabványos kimeneten is megjelennek, tehát ha az eredményt valóban csak az adott fájlban akarjuk viszontláttni, akkor itt is használnunk kell a -n kapcsolót:

```
cat telefon.txt | sed -n '3,7 w kivonat.txt'
```

Ez a parancs a telefon.txt fájl 3-7. sorait a kivonat.txt fájlban helyezi el, miközben a képernyőn semmi nem jelenik meg.

A w a már létező kimeneti fájlt felülírja, a még nem létezőt pedig létrehozza. Ugyanakkor ha a w parancsot egy sed programon belül többször használjuk, valamennyi kimenet ugyanabba a fájlba kerül. A felülírásról mondottak tehát egy programon belül nem érvényesek.

Az r parancccsal egy fájl teljes tartalmát olvashatjuk be a mintatérbe (akár több sort is egyszerre). Ez a művelet akkor lehet hasznos, ha egy adott szöveget akarunk beszűni egy vagy több dokumentumba egy meghatározott helyre.

Buktató

A sed fájlírási és olvasási műveletei némely UNIX rendszeren meglehetősen szigorúak a fájlnév tekintetében. Az r vagy w parancsot elvileg egy szóköz követheti (ez viszont kötelező!), utána pedig a sed minden karaktert a névhez tartozónak tekint, beleértve az esetleg véletlenül odaírt szóközöket is. A GNU sed kevésbé szigorú: a név előtt tetszőlegesen sok szóközt megenged, utána azonban egyet sem.

n (Next), q (Quit)

Az n parancs a kimenetre küldi a mintatér tartalmát, majd a bemenet következő sorát olvassa be oda. A program elejére ugyanakkor nem ugrik vissza, tehát az újonnan beolvasott sor feldolgozása a következő programsorral kezdődik.

A q parancs hatására a program, és vele együtt a bemenet feldolgozása azonnal befejeződik.

A fejezet hátralevő részében két példán mutatom be a sed használatát, illetve a sed programok fejlesztésének „útvesztőit”.

Tizedespont, tizedesvessző

Mint azt nyilván az Olvasó is tudja, ha egy magyar nyelvű szövegben tizedestörteket írunk le, akkor a nyugati szokásuktól eltérően tizedesvesszőt és nem tizedespontot használunk. Ugyanakkor számos olyan helyzet képzelhető el, amikor a szövegbe bekerülő számok nem ebben a formában keletkeznek (például egy program írta ki azokat valamilyen matematikai művelet eredményeként). Első feladatunk legyen egy olyan sed program megírása, amely az ilyen dokumentumokban képes felismerni a tizedespontokat, és vesszőre cserélni azokat.

A feladat első látásra egyszerűnek tűnik. Nem kell egyebet tennünk, mint a sed y (Transform) parancsával minden pontot vesszőre cserálni. Aztán ha kicsit eltöprengünk a dolgon, rájöhettünk, hogy ez a módszer csak olyan dokumentumoknál fog működni, amelyek kizárolag számokat tartalmaznak. Ha azonban ez a feltétel nem teljesül, akkor ügyelnünk kell arra, hogy a mondat végi pontok és a befejezetlen mondatokat záró három pont „épségben megússza” az átalakítást. Ha pedig netán űrlapokat dolgozunk fel, amelyeken a kitöltendő részeket kipontozták, még érdekesebb a helyzet. Ha ilyen és ehhez hasonló általános célú programokat fejlesztünk, a szabályos kifejezések szabatos megfogalmazásában sokat segíthet, ha van egy próbafájlunk, amely az összes lehetséges változatot tartalmazza. Ha ezzel működik a programunk, akkor jó esély van rá, hogy a gyakorlatban sem vallunk vele szégyent.

Esetünkben a következő szöveg tartalmazza az összes olyan környezetet, amelyben tizedestört előfordulhat:

Pi értéke 3.1415926 , E értéke 2.718282.
Kettő négyzetgyöke (1.41421) irrationális szám.
Ebben a sorban csak szöveg van...

1.2345 Ebben a sorban van tizedestört is...

A legjobb érzés a pH 5.5.

45.6789

234.567

Első ötletünket (y parancs) ezen a fájlon nyilván kipróbálni sem érdemes, magát a parancssort azonban érdemes megmutatni:

```
cat tizedestort.txt | sed 'y/.//,'
```

Figyeljük meg, hogy itt nem kellett levédeni a pontot (\.) ahhoz, hogy literális értelmet kapjon. Ennek oka, hogy az y parancs után eleve csak karakterlánc következhet, szabályos kifejezés nem. A pontnak viszont csak az utóbbiban lenne különleges jelentése. Ha ugyanezt az s parancssal próbáljuk megvalósítani, ott természetesen már kell a \ a pont elő:

```
cat tizedestort.txt | sed 's/\.\./,/g'
```

E két megoldás nem elég általános, tehát rögtön próbáljuk meg pontosabban körülírni, hogy miről ismerszik meg egy tizedespont. A válasz egyszerű: két számjegy határolja úgy, hogy a három karakter közvetlenül követi egymást. Ennek a formátumnak a "[0-9]\.[0-9]" szabályos kifejezés felel meg, amivel rögtön megfogalmazhatunk egy karaktercserét végrehajtó programot:

```
cat tizedestort.txt | sed 's/[0-9]\.[0-9]//,/g'
```

A kimenet a következő lesz:

Pi értéke ,415926 , E értéke ,18282.

Kettő négyzetgyöke (,1421) irrationális szám.

Ebben a sorban csak szöveg van...

,345 Ebben a sorban van tizedestört is...

A legjobb érzés a pH ,.

4,789

23,67

A hiba szembeötlő, sőt az okára sem nehéz rájönni: a csere nem csak a tizedespontra, hanem az azt megelőző és követő számjegyre is kiterjedt, így ezek elvesztek. A megoldás nyilván az, hogy a fent megfogalmazott szabályos kifejezést nem helyettesítési mintaként, hanem a parancs címzéseként kell használni:

```
cat tizedestort.txt | sed '/[0-9]\.[0-9]/ s/\.\./,/g'
```

Így a helyettesítés csak azokban a sorokban megy végbe, amelyek tartalmazzák a megadott mintát. A kimenet most így fest:

```
Pi értéke 3,1415926 , E értéke 2,718282,
Kettő négyzetgyöke (1,41421) irrationális szám,
Ebben a sorban csak szöveg van...
1,2345 Ebben a sorban van tizedestört is.,
A legjobb érzés a pH 5,5,
45,6789
234,567
```

Ez már lényegesen jobb, de az írásjelként használt pontok túlnyomó többsége most is áldozatául esett a cserének. Egy kivétel van csak: az a sor, amelyikben nem volt tizedestört. Itt megmaradt a mondat végén a három pont, mivel a címként használt szabályos kifejezés erre a sorra nem illeszkedett.

Látható tehát, hogy – bár jó helyen keressélünk – a szabályos kifejezésnek címként való megadása nem elég szigorú feltétel. A végső megoldás bonyolultabb feltételerendszer megfogalmazását követeli:

1. Keressük meg a két számjegy által határolt pontokat.
2. Keressük meg a pont két oldalán a leghosszabb, csak számjegyekből álló karakterláncot. (A bal oldali nyilván a tizedestört egészrészé, a jobb oldali pedig a törtrésze lesz.)
3. E két utóbbi numerikus karakterláncot helyettesítsük önmagával, míg a közöttük levő pontot cseréljük vesszőre.

Ennek az algoritmusnak a következő sed program felel meg:

```
cat tizedestort.txt | sed 's/\\([0-9][0-9]*\\)\\.\\([0-9]
[0-9]*\\)/\\1\\,\\2/g'
```

A "[0-9][0-9]*" kifejezés legalább egy számjegy előfordulását megköveteli, tehát nem fog „bedőlni” annak a helyzetnek, amikor egy mondat éppen egy számmal végződik („pH 5.5.”). A kerek zárójelek használatára azért van szükség, hogy a tartalmukra illeszkedő szövegrészekre hivatkozhassunk a behelyettesítés során.

A fentivel teljesen azonos, de valamivel rövidebb megoldás a következő:

```
cat tizedestort.txt | sed 's/\\([0-9]\\+\\)\\.\\([0-9]\\+\\)/\\1\\,\\2/g'
```

Itt csupán annyi az eltérés, hogy a „legalább egy számjegy” illeszkedését a + jellet írtuk elő.

Akármelyik megoldást alkalmazzuk, a kimenet most már éppen az, amit vártunk:

```
Pi értéke 3,1415926 , E értéke 2,718282.  
Kettő négyzetgyöke (1,41421) irrationális szám.  
Ebben a sorban csak szöveg van...  
1,2345 Ebben a sorban van tizedestört is...  
A legjobb érzés a pH 5,5.  
45,6789  
234,567
```

Készítsünk telefonkönyvet

Tegyük fel, hogy van egy neveket és telefonszámokat tartalmazó listánk, amely soronként egy-egy bejegyzést tartalmaz. Az első adat minden sorban a telefonszám, a második a név, valahogy így:

```
cat bubo.txt

1333444 Dr. Bubó Bubó
3141592 Ele Fáni
1428571 Csőr Mester
1111111 Teknőc Ernő
7692307 Prof. Boa Konstrektor
```

(Az egyszerűség kedvéért tételezzük fel, hogy a telefonszámok hétjegyűek és semmiféle tagolást nem tartalmaznak.) A feladat ezek után az, hogy ebből a rendezetlen listából előállítsunk egy betűrendbe szedett telefonkönyvet.

Aki legalább alapszinten ismeri a UNIX operációs rendszert (az Olvasóról ezt eleve feltételeztük), az egy ilyen feladattól nem ijed meg, hiszen emlékszik rá, hogy létezik egy sort nevű segédprogram, amit pontosan erre találtak ki.

Első látásra is van azonban egy kis gond ezzel a listával: előbb szerepel benne a telefonszám, csak aztán jön a név. Márpedig ha ezt egyszerűen átadjuk a sort-nak, az a telefonszámok alapján fogja növekvő sorba rendezni a listát, ami egy telefonkönyvben kissé furcsán festene. A megoldás kézenfekvő: meg kell cserélnünk minden sorban a két bejegyzést, aztán ezt a listát kell átadni a sort-nak.

A szabályos kifejezések lélektanát és a sed lehetőségeit ismerve először a következő megoldás születik meg:

```
cat bubo.txt | sed 's/\(([0-9]\{7\})\)\(([a-zA-Z \.]*\))/\2 \1/'
```

Az s (Substitute) parancsot vezérlő részben két, kerek zárójelek közé zárt szabályos kifejezést látunk. Az első pontosan hét darab számot (a telefonszám), a második kis- és nagybetűkből, valamint szóközökből és pontokból (Dr.) álló karakterláncot (a név) azonosít. A helyettesítés során a két illeszkedő részt megcserélve (\2 \1) íratjuk ki, egy szóköz közbeiktatásával. Reményeinkkel szemben a kimenet sajnos így fest:

```
Dr. Bub 13334446 Bubó
Ursula N 17778886vér
Ele F 3141592áni
Cs 1428571őr Mester
Tekn 11111116c Ernő
Prof. Boa Konstrektor 7692307
```

Azon kívül, hogy nem pont ezt vártuk, az eredmény még egy dolog miatt furcsa: a Prof. Boa Konstrektor adatait tartalmazó sorral pontosan az történt, amit tervezünk. Az összes többivel viszont nem...

Talán már az Olvasó is rájött, hogy a hiba az ékezetes betűkkel függ össze. minden telefonszám az első ékezetes karakter elő került be, ami azt jelenti, hogy a sed a jelek szerint ezeket nem tekintette betűnek...

4.1. táblázat A szabályos kifejezésekben használható POSIX karakterosztályok

Osztály	Leírás
[:alnum:]	Nyomtatható karakterek (szóköz is)
[:alpha:]	Az ABC-ben előforduló karakterek
[:blank:]	Szóköz és tabulátor
[:cntrl:]	Vezérlőkarakterek
[:digit:]	Numerikus karakterek (számok)
[:graph:]	Nyomtatható és látható (nem szóköz) karakterek
[:lower:]	Kisbetűk
[:print:]	Nyomtatható karakterek (szóköz is)
[:punct:]	Írásjelek
[:space:]	Üres helynek megfelelő karakterek
[:upper:]	Nagybetűk
[:xdigit:]	Hexadecimális számok

Buktató**Magyarnak lenni...**

Kezdetben volt az angol ábécé... Aztán megérkeztünk mi magyarok – egyesek szerint a Marsról – és azóta valahogy senki se érti, mit beszélünk. A UNIX megalkotói az angol nyelvet beszélték, tehát valamelyest természetes, hogy a fentihez hasonló gondokba időnként belefutunk. Viszont az is sejthető, hogy megoldásnak is léteznie kell, másikról a UNIX-ok amolyan félkarú óriások lennének azokon a nyelvterületeken, ahol a nemzeti karakterkészlet 26-nál valamivel több betűt tartalmaz.

A fenti példa azt mutatja, hogy a magyar ékezetes magánhangzók nem elemei az [a-z] illetve [A-Z] tartománynak, vagyis a UNIX segédprogramjai ezeket – első közelítésben – egyszerűen nem tekintik betűknek. Azt viszont megtehetjük, hogy az ilyen karakterlisták megadásakor egyenként kiírjuk a számunkra oly kedves ékezetes betűket:

[a-zAÉÍÓÖŰÜŰ]

illetve

[A-ZÁÉÍÓÖŰÜŰ]

Ezek már pontosan úgy fognak működni, ahogyan azt vártuk.

A POSIX szabvány szintén leír „különleges” karakterosztályokat (lásd a fenti táblázatot), amelyekkel tömörebben fogalmazhatunk meg szabályos kifejezéseket. A magyar ékezetes betűk példájánál maradva az [:alpha:] karakterosztály például elvileg ezekre is illeszkedik. Elvileg. Ha ugyanis kipróbáljuk ezt a módszert a GNU segédprogramokkal, azt fogjuk tapasztalni, hogy ezzel a játékszabállyal csak a grep és az awk van tisztában. A sed továbbra is ragaszkodik hozzá, hogy amin ékezet van, az nem betű. Ugyanakkor nem is szám, és ezt néha kiválóan kihasználhatjuk...

Ha GNU sed-et (Linuxot) használunk, a fentiek alapján a hiba orvoslására két lehetőségünk van: vagy kiírjuk az ékezetes betűket, vagy kihasználjuk, hogy az ékezetes betű „nem betű” ugyan, de nem is szám:

```
cat bubo.txt | sed 's/\\([0-9]\\{7\\}\\)\\([a-zAÉÍÓÖŰÜŰÁÉÍÓÖŰÜŰ \\.]*\\)/\\2 \\1/'
```

vagy

```
cat bubo.txt | sed 's/\\([0-9]\\{7\\}\\)\\(([^\0-9]*\\))/\\2 \\1/'
```

Egyszerűsége miatt a továbbiakban az utóbbi – kerülő – megoldást fogom használni. A kimenet tehát jelen pillanatban így néz ki:

```
Dr. Bubó Bubó 1333444
Ursula Nővér 1777888
Ele Fáni 3141592
Csőr Mester 1428571
Teknőc Ernő 1111111
Prof. Boa Konstrektor 7692307
```

Ez már majdnem tökéletes, de azért van még egy kis baj a tudományos fokozatokkal. Bubó Dokinak például nyilván a B betűnél kellene következnie a telefonkönyvben, de ha így hagyjuk a listát, akkor a D-hez kerül. Ezt a kérdést a valódi telefonkönyvekben is úgy oldják meg, hogy a címeket a név után írják. A következő feladatunk tehát a doktori cím felismerése és a név után illesztése.

A sed leírását (man sed) olvasgatva sok ember első ötlete valószínűleg az lenne, hogy az „s” parancsal töröljük a „Dr.” karakterláncot (helyettesítsük egy szóközzel), majd az „a” (Append) parancs segítségével illesszük be ugyanezt a sor végére. A megoldás valahogyan így festene:

```
cat bubo.txt | \
sed -n 's/ Dr\.. / / \
a\
Dr.
p'
```

Az első sor végén szereplő \ karakter a sortörés jele, amit itt csak a könnyebb olvashatóság érdekében használtam. Az Olvasó az előzőek alapján már sejtheti, hogy nem ez a helyes megoldás. És valóban, a kimenet így fest:

```
1333444 Bubó Bubó
Dr.
1777888 Ursula Nővér
Dr.
3141592 Ele Fáni
Dr.
1428571 Csőr Mester
Dr.
1111111 Teknőc Ernő
Dr.
7692307 Prof. Boa Konstrektor
Dr.
```

Itt bizony több baj is van. Egyszerű ünnepélyesen mindenkit doktorrá avattunk, másrészről a doktori címek a nevek utáni sorba és nem közvetlenül a név után kerültek. Az utóbbi gondon nem is lehet segíteni, mivel az a parancs mindenkorban új sorba írja a beillesztett szöveget.

A doktori címek tömeges osztogatását viszont világos módon az okozza, hogy a hozzáfűzési műveletet cím nélkül adtuk meg, s így a sed – az alapértelmezésnek megfelelően – valamennyi sorra végrehajtotta azt.

A fentieknek megfelelően második próbálkozásunk így fest:

```
cat bubo.txt | \
sed -n '/ Dr\. / {
s/ Dr\. / /
a\
Dr.
}
p'
```

Az eltérés mondhatni árnyalatnyi, de nagyon fontos: a "/ Dr\. /" szabályos kifejezést itt már nem csak a helyettesítésben használjuk, hanem az egész műveletsor címzésére is. A műveleteket kapcsos zárójelek segítségével vontuk össze egyetlen műveleti blokká, amelyre a megadott cím egyben vonatkozik. A kimenet így fest:

```
1333444 Bubó Bubó
Dr.
1777888 Ursula Nővér
3141592 Ele Fáni
1428571 Csőr Mester
1111111 Teknőc Ernő
7692307 Prof. Boa Konstrektor
```

Ez már majdnem tökéletes. Ha még azt is el tudjuk érni, hogy Bubó Doktor neve és a két betűje egy sorban szerepeljen, akkor nyert ügyünk van. Amint azt már említtettem, ez az a parancsal nem fog menni. Használhatunk azonban egy második helyettesítést, amely a " Dr. " karakterláncot a mintatér teljes tartalma után illeszti be. Prof. Boa miatt sajnos még egy ugyanilyen programblokkot kell majd írnunk. A megoldás tehát:

```
cat bubo.txt | \
sed -n '
/ Dr\. / {
s/ Dr\. / /
s/\(.*\)\1 Dr\./
}'
```

```
/ Prof\.. / {
s/ Prof\.. / /
s/\(.*\)/\1 Prof\../
}
p'
```

Figyeljük meg, hogy a „Dr.” karakterláncnak a sor utáni beszúrása során nemes egyszerűséggel a „bármiből bármennyi” (.*) szabályos kifejezést használtuk, kerek zárójelek közé téve. Maga a kifejezés nyilván a teljes sorra illeszkedett, a zárójelekre pedig azért volt szükség, hogy a behelyettesítendő kifejezés megadásánál hivatkozhassunk (\1) az illeszkedő részre. A lista immáron tökéletesen megfelel a vára-kozásnak, így nincs más hátra, mint beilleszteni a program végére a nevet és telefonszámot megcserélő parancssort, majd a teljes kimenetet a sort bemenetére irányítani. Íme a telefonkönyvet összeállító programunk teljes fényében:

4.1. lista

```
cat bubo.txt | \
sed -n ' \
/ Dr\.. / {
s/ Dr\.. / /
s/\(.*\)/\1 Dr\../
}
/ Prof\.. / {
s/ Prof\.. / /
s/\(.*\)/\1 Prof\../
}
s/([0-9]\{7\})\(([^\0-9]*\))/\2 \1/
s/^ //'
p' | sort
```

A program utolsó sorában egy újabb helyettesítés segítségével még töröltünk minden sor eleji szóközt, amelyek az előzetes feldolgozás során keletkeztek. És íme az Erdő telefonkönyve:

```
Boa Konstrektor Prof. 7692307
Bubó Bubó Dr. 1333444
Csőr Mester 1428571
Ele Fáni 3141592
Teknőc Ernő 1111111
Ursula Nővér 1777888
```

5

Az AWK használata héjprogramokban

Ebben a fejezetben a sed „nagyobbik testvéréről” az awk nevű segédprogramról lesz szó. Az awk a héjprogramokban talán leggyakrabban használt szövegszerkesztő és feldolgozó program. Számos szolgáltatása erősen hasonlít a sed képességeihez, de használata sokak számára lényegesen kényelmesebbnek tűnik majd. A legfontosabb különbség e két segédeszköz között a vezérlésükre használt nyelv utasításformája. Amint az előző fejezetben láttuk, a sed valamennyi parancsa egy-egy betű. Szerencsére nincs túl sok belőlük, de egy összetettebb sed program „visszafejtése” főleg kezdők számára meglehetősen körülményesnek tűnhet.

Az awk vezérlési nyelve ezzel szemben a C programozási nyelv szövegfeldolgozásra ki-hegyezett változata. Ennek megfelelően „értelmes” szavakkal fogalmazhatjuk meg a feldolgozó algoritmust, ami a későbbi karbantartást, illetve az olvashatóságot is könnyíti.

Az awk működésének alapelvei

A programok szerkezete és indítása

Annak ellenére, hogy az awk egyszerű, parancssorból hívható program, és mérete egyetlen rendszeren sem haladja meg a néhány száz kilobájtot, képességei gyakorlatilag azonosak egy grafikus felületen futó táblázatkezelővel.

A sed-hez hasonlóan az awk is képes fájlból venni az utasításokat, tehát ebben az értelemben tulajdonképpen nem is egyszerű segédprogram, hanem egy teljes programozási nyelv. Vannak változói, tud fájlokat kezelní, szervezhető benne ciklus, meghatározhatók benne függvények és eljárások. Összességében tehát az awk egy szövegfeldolgozásra szakosított programnyelv.

Az awk alapértelmezés szerint a szabványos bemenetről vagy a megadott fájlok ból veszi a feldolgozandó szöveget. Utasításformája ennek megfelelően – akárcsak a sed-é – kétféle lehet:

```
... | awk '{parancsok}'
```

vagy

```
awk '{parancsok}' feldolgozandó_fájlok
```

Ha a feldolgozó program külön fájlban van, annak nevét a -f kapcsoló után adhatjuk meg.

```
... | awk -f programfájl.awk
```

A program utasításait mindenek között kell megadni, parancssorban történő megadásnál pedig az egész programszöveget egyszeres idézőjelekkel kell megvédeni a héjtól. Erre azért van szükség, mert a szabályos kifejezésekhez hasonlóan az awk nyelv is tartalmaz a héj számára is értelmes utasításokat.

TIPP

Az első fejezetben esett szó a héjprogramok indításáról. Ott azt láttuk, hogy a legegyszerűbb módszer erre a

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

megadása az első sorban. Itt tulajdonképpen annak a programnak a nevét közöljük a rendszerrel, amelyik „érzi” a továbbiakban leírtakat. Ezek után talán nem meglepő, hogy a módszer az awk programokkal is működik:

```
#!/bin/awk -f  
<Az awk program sorai>
```

A -f kapcsoló megadása egyes rendszereken nem kötelező (GNU awk esetében igen), viszont semmiképpen nem hiba. Tehát jobb a békesség alapon inkább írjuk ki.

Fontos megjegyezni, hogy ebben az esetben kifejezetten hiba, ha az awk program sorait egyszeres idézőjelek közé zárjuk. Ilyenkor ugyanis a sorokat közvetlenül az awk kapja meg, vagyis a héjnak esélye sincs rá, hogy értelmezze azokat (röviden: nincs szükség védelemre). Az awk viszont megkapja az idézőjeleket is, és természetesen nem tudja mire vélni azokat.

Mezők

Az awk, miután a programot formailag (szintaktikailag) ellenőrizte, a feldolgozást a bemenetre érkező szöveg részekre bontásával kezdi. A feldolgozás soronként törtenik, a legkisebb feldolgozási egység pedig a „mező”.

Első közelítésben (ez az alapértelmezés) mező egy sornak minden olyan része, amit a környezetétől elöl és hátul egyaránt legalább egy-egy szóköz vagy tabulátor választ el. Rögtön hozzáteszem, hogy ez az alapértelmezés tetszés szerint megváltoztatható, vagyis mezőelválasztóként tetszőleges karakter felhasználható.

Miután az awk mezőkre bontotta a feldolgozandó sort, végrehajtja rajta a teljes programot. Fontos még egyszer hangsúlyozni, hogy az awk sor alapú feldolgozást végez, tehát minden program tartalmaz egy „rejtett (implicit) főciklust”, ami végigmegy a bemenet összes során. (Más nyelvekben ezt magunknak kellene megírni.)

A sor mezőire a `$i` jelöléssel hivatkozhatunk, ahol `i` a kérdéses mező sorszáma. A számozás 1-től indul, `$0` pedig a teljes sort jelenti egyben, a mezőelválasztó karakterekkel együtt.

Kírás (print)

Egyszerű példaként tegyük fel, hogy egy két oszlopot tartalmazó szövegfájlban (`lista.txt`), meg akarjuk cserélni az oszlopok tartalmát. Az egyszerűség kedvéért tegyük fel azt is, hogy az oszlopokat minden sorban egy vagy több szóköz választja el egymástól. A feladatnak megfelelő awk program a következő:

```
cat lista.txt | awk '{print $2" "$1}' > lista1.txt
```

A `print` parancs természetesen az utána következő dolgok kimenetre írását jeleneti. Ha itt egy mezőre való hivatkozás vagy változó szerepel, akkor elég a nevét leírni. Ha viszont valamilyen szöveg kiírására akarjuk rávenni, akkor azt kettős idézőjelek között kell megadni.

Példánkban a két idézőjel között egy szóköz van, tehát a `lista1.txt`-ben már mindenütt pontosan egy szóköz lesz az oszlopok között, függetlenül attól, hogy mit tartalmazott a bemenet.

Ha hosszabb listát nyomtatunk, amelynek minden elemét pontosan egy szóköznek kell elválasztania, nem kell a fenti módon megadni az elválasztójelet. Ha egyszerűen visszűvel elválasztva felsoroljuk a kiírandókat, minden elem közé pontosan egy fog bekerülni az érvényes kimeneti mezőelválasztó karakterből (alapállapotban ez is szóköz). A fenti példát tehát akár így is írhattuk volna:

```
cat lista.txt | awk '{print $2,$1}' > lista1.txt
```

TIPP

A `print` parancs paraméterek hiányában alapállapotban `$0`-t, vagyis a teljes sort küldi a kimenetre. Ha tehát a bemenet egy sorával csak annyi a célunk, hogy minden nem változtatás nélkül a kimenetre küldjük, ki sem kell írni a `$0` szimbólumot.

Buktató

A `print` parancs minden esetben kiküld egy újsor karaktert is a kiemenet után. Ezt a viselkedését megváltoztatni sem lehet, így ha újsor nélküli kiíratást akarunk megvalósítani, akkor a `printf` parancsot kell használnunk helyette (lásd később).

Egy másik érdekes eset az, amikor csak egy újsor karaktert akarunk kiíratni. Ilyenkor a helyes megoldás nem a

`print`

hanem a

`print "`

az első változat ugyanis `$0-t` ír ki (lásd fent).

Felhasználó által megadott változók

Mint minden programnyelvben, az awk-ban vannak is változók. Ezeket – a héjprogramokhoz hasonlóan – itt sem kell külön bevezetnünk. Amint leírjuk őket, azonnal létrejönnek.

A következő példában megadunk egy szöveges és egy numerikus változót, és hozzáírjuk azokat a lista minden sorához:

```
cat lista.txt | awk '{szoveg="AAA" ; szam=5 ;
                     print $2,$1,szoveg,szam}' > lista2.txt
```

Látható, hogy a változók értékére egyszerűen a nevük leírásával hivatkozhatunk, tehát a héjprogramuktól eltérően itt nincs szükség a `$` jelre. Az egymást követő utasításokat egy programblokkon belül pontosvessző vagy újsor választhatja el egymástól.

Kapcsolattartás a héjprogram és az awk program között

Amint említettem, az awk a fenti programot minden sorra végrehajtja, ami egyben azt is jelenti, hogy a két értékkedási művelet is annyiszor hajtódiék végre, ahány sor van a `lista.txt` fájlban. Ez formalag ugyan helyes, elegánsnak viszont nem mondható. Az awk két megoldást kínál erre az esetre. Egyszer a `-v` kapcsoló után értéket adhatunk bizonyos awk változóknak, még a program végrehajtása előtt. Ezzel a módszerrel a fenti példa a következő módon fest:

```
cat lista.txt | awk -v szoveg=AAA -v szam=5
                  '{print $2,$1,szoveg,szam}' > \
lista2.txt
```

Egy héjprogramban az „AAA” és az „5” karakterláncok helyett természetesen használhattunk volna két héjváltozóra való érték szerinti hivatkozást is. Ez a lehetőség azért különösen fontos, mert a héjprogramból így adhatunk át adatot az awk programnak. Figyeljük meg, hogy a -v kapcsolót annyiszor kell kiírni, ahány előzetes értékkadás történik!

A másik megoldás a BEGIN blokk használata, amiről később lesz szó.

Belső változók

Az awk-ban nemcsak a felhasználó által megadott változók léteznek, hanem maga a program is rendelkezik olyan belső változókkal, melyek gyakran használt, általános célú adatokat tárolnak, és amelyek folyamatos frissítéséről az awk gondoskodik munka közben.

Ha például a könyvtárunkban levő fájlokról vagy egy szövegfájl soraiból sorszámozott listát akarunk készíteni, nem kell a feldolgozott sorokat külön változóban számlálni. Van ugyanis az awk-nak egy NR nevű belső változója, amely minden a feldolgozandó sor sorszámát tartalmazza.

```
ls -l | awk '{print NR, " ", $9}'
```

A belső változók nevét mindenkor nagybetűkkel kell írni. Az awk legfontosabb belső változóit az 5.1. táblázat tartalmazza. Használatukról a gyakorlatok során még bőven esik majd szó.

5.1. táblázat Az AWK legfontosabb belső változói

Változó	Leírás	Alapértelmezett érték
RS	Rekordelválasztó karakter (Record Separator)	Újsor
ORS	Kimeneti rekordelválasztó karakter (Output Record Separator)	Újsor
FS	Bemeneti mezőelválasztó karakter (Field Separator)	Szóközök illetve tabulátorok
OFS	Kimeneti mezőelválasztó karakter (Output Field Separator)	Szóköz
NR	A pillanátnyi sor sorszáma (Number of Row)	Nincs
NF	A mezők száma a pillanatnyi sorban (Number of Fields)	Nincs

A mezőelválasztó karakterek kezelése

Az awk számára az alapértelmezett mezőelválasztó jel a tetszőleges hosszúságú, szóközökből és tabulátorokból álló rész. Röviden – és kissé lazán – fogalmazva tehát alaphelyzetben mezőelválasztó minden, amit nem látunk.

Az alapértelmezett viselkedést tetszés szerint módosíthatjuk, ha az FS belső változóban az általunk használni kívánt mezőelválasztó karaktert helyezzük el.

Ha pontosan egy karaktert adtunk meg, akkor egyszerűen ez lesz a mezőelválasztó. Arra azonban ügyeljünk, hogy ebben az esetben az illető karakter minden egyes előfordulása mezőelválasztó lesz még akkor is, ha több mezőelválasztó sorakozik egymás után. Ilyenkor a közbenső mezők léteznek ugyan az awk számára, de értékük nem meghatározott (üres karakterlánc).

Ha több karaktert adunk meg mezőelválasztóként, az awk az FS változó tartalmát szabályos kifejezésnek tekinti, és mezőelválasztónak a kifejezés legtágabb illeszkedését fogja venni. Ez az a pont, ahol az alapértelmezett viselkedés érhetővé válik: az alapértelmezett mezőelválasztó valójában nem szóköz, hanem a

" [\t] + "

szabályos kifejezés („\t” a tabulátor karakter jele). A „legtágabb illeszkedés” elve alapján ez szóközök és tabulátorok tetszőleges kombinációjára illeszkedik, úgy, hogy megköveteli legalább egy ilyen karakter előfordulását.

Megjegyzés

A fentiek után azt várnánk, hogy ha az awk programban az FS értékét pontosan egy szóközre állítjuk (FS= " "), akkor ezzel felülbíráljuk az alapértelmezett viselkedést, tehát minden egyes szóközzel egy újabb mező kezdődik. Ez általában így is van, a GNU awk (Linux!) esetében azonban nem. Ott egyetlen szóköz hatására visszaáll az alapértelmezett viselkedés. Egy másik, határozottan érdekes eset az, amikor az FS értékéül üres karakterláncot adunk meg (FS= " "). Ekkor az awk minden karaktert önálló mezőnek tekint, ami egyes esetekben kifejezetten hasznos lehet, hiszen a bemenet sorai ezáltal betűnként címezhetővé válnak.

Az AWK nyelvi elemei

A BEGIN és END blokkok

A feldolgozó főciklustól független műveletek végrehajtására nyújt megoldást a BEGIN és az END programblokk. A tényleges feldolgozást végző program előtt a BEGIN szó után meghatározhatunk egy olyan műveletsort, amit az awk még a sorok feldolgozása előtt, de csak egyszer fog végrehajtani. Van lehetőség arra is, hogy a sorok feldolgozása után végezzünk még valamilyen műveleteket. Ezeket az END kulcsszó után kell megadni a program végén.

Összefoglalva tehát egy awk program általános formája a következő:

```
... | awk 'BEGIN {bevezető műveletek} {főprogram} END {záró műveletek}'
```

Buktató

A BEGIN és END kulcsszavakat az awk egyes megvalósításainál kötelező nagybetűkkel írni, míg mások nem ennyire finnyásak. minden esetre a nagybetűs írásmód mindenütt működik.

Hasonlóan furcsa hibalehetőség, hogy bizonyos rendszereken a BEGIN és END utáni programblokk nyitó kapcsos zárójelének kötelező a BEGIN vagy END kulcsszóval egy sorban lennie.

Az előző szakaszban bemutatott egyszerű példa két bevezető értékkadását így is megoldhatjuk:

```
cat lista.txt | awk 'BEGIN {szoveg="AAA" ; szam=5}
                      {print $2,$1,szoveg,szam}' > lista2.txt
```

Buktató

A figyelmes szemlélő rögtön felfedezheti, hogy a szöveges változó tartalmát az értékkadásnál most idézőjelek közé zártam, az előző megoldásnál viszont nem. Ennek oka, hogy az awk változóit nem kell külön bevezetni, másrészt a nevük előtt nem használunk \$ jelet akkor sem, amikor érték szerint hivatkozunk rájuk. Ebből pedig az következik, hogy az awk egy változót és egy karakterláncot csak úgy tud megküldeniöböztetni, ha az idézőjelekkel jelezzük neki, hogy miről van szó. Röviden tehát: awk programokban a karakterláncokat kötelező kettős idézőjelek közé zárni.

Matematikai műveletek

Az awk változóival, illetve mezőivel végezhetünk matematikai műveleteket is. Ehhez semmiféle átalakításra nincs szükség. Az awk változóinak – a héjváltozókhöz hasonlóan – tulajdonképpen nincs típusuk, valamennyien egyszerű karakterlánc-ként tárolódnak. Ugyanakkor a program a kellő pillanatban számmá alakítja őket.

Az awk által ismert matematikai és egyéb függvények köre rendszerről rendszerre kissé változhat, illetve egyes függvényeknek más lehet a neve. Általánosságban azonban elmondhatjuk, hogy az awk körülbelül annyit tud a matematikából, mint egy tudományos zsebszámológép. A matematikai függvények neve minden esetben a szokásos (`sin`, `cos`, `exp` stb.), a másik nagy csoportot alkotó karakterláncokat kezelő függvényekről pedig világos leírást találhatunk az awk leírásában (`man awk`).

A következő egyszerű példában az awk matematikai képességeiből csak az összeadást használjuk. Legyen a feladat a pillanatnyi könyvtárban levő fájlok méretének összeadása. Első próbálkozásunk így fest:

```
ls -l | awk 'BEGIN {meret=0} {meret=meret+$5} END{print meret}'
```

A csak egyszer lefutó `BEGIN` blokkban a „`meret`” nevű változó értékét nullára állítjuk (és ezzel egyben meg is határozzuk magát a változót).

Megjegyzés

A fenti kezdő értékkadás az esetek túlnyomó többségében valójában fölösleges. Az awk ugyanis az első értékkadásnál magától nullázza a változókat. Gond elvileg csak akkor lehet, ha érték szerint próbálunk hozzáérni egy addig még meg nem határozott változóhoz. Mint azt korábban láttuk, héjprogramok esetében ilyenkor biztos, hogy hiha keletkezik. Ezzel szemben a legtöbb awk változat a nem meghatározott változókat is nullának (üres karakterlánc) tekinti, tehát kevésbé finnyás, mint a parancsértelmezők.

A főblokkban a „`meret`” értékéhez hozzáadjuk valamennyi sor ötödik mezőjének értékét (az `ls -l` által adott listában az ötödik elem a fájl mérete). Végül az összes sor feldolgozása után az `END` blokkban kiírjuk a végeredményt.

Megjegyzés

A főblokkban a méret nevű változó növelését „C stílusban” is megoldhattuk volna:

```
meret+=5
```

Kipróbálva művünket kiderül, hogy nem minden működik helyesen. Ha csak fájlok vannak a pillanatnyi könyvtárban, akkor minden rendben, ha viszont alkönyvtárak is nyílnak belőle, az összeg kicsit nagyobb lesz a fájlok összesített méreténél. Az ls kiemenetének tanulmányozása után nyilvánvaló a hiba: a méret mezőben alkönyvtárak esetén is van egy szám, ami egyéb utasítás hiányában szintén bekerül a végösszegbe.

A megoldás nyilván az, hogy valahogyan „kigyomláljuk” a listából az alkönyvtárak sorait. Rögtön két megoldás is kínálkozik. A már tárgyalt grep parancsot éppen ilyen műveletekre találták ki:

```
ls -l | grep '^-' | awk 'BEGIN {meret=0} {meret+=\$5} END {print meret}'
```

A grep itt eleve csak azokat a sorokat engedi az awk bemenetére, amelyek „-” karakterrel kezdődnek (d helyett), vagyis közönséges fájlokra vonatkoznak.

Efféle válogatást természetesen maga az awk is végre tud hajtani, hiszen ő is ismeri a szabályos kifejezéseket:

```
ls -l | awk \
'BEGIN {meret=0}
/^-/ {meret+=$5}
END {print meret}'
```

Ha egy kapcsos zárójelekkel elkülönített programblokk elő az itt látható módon egy szabályos kifejezést írunk, a blokk csak azokra a sorokra hajtódik végre, amelyekre az adott kifejezés illeszkedik. Az awk programokban is használhatjuk tehát azt a címzési módöt, amit a sed-nél már megismertünk. A főprogram természetezen több ilyen címzett programblokkot is tartalmazhat, akárcsak a sed programok.



Az első sor végén látható \ karakter azt jelzi a héj számára, hogy az értelmezés során ezt a sort össze kell fűznie a következővel. Az ilyen sortörést szabadon alkalmazhatjuk, ha a parancssor túl hosszú, vagy (mint itt is) olvashatóbbá akarjuk tenni a kódot. Arra azonban ügyeljünk, hogy a \ karakternek valóban utolsónak kell lennie a sorban. Ez azt jelenti, hogy még szóköz sem lehet utána!

Ez a kezdő felhasználókra leselkedő egyik legalattomosabb veszély: a szóköz nem látszik, a kapott hibaüzenetnek pedig rendszerint semmi köze a tényleges hibához.

Magán az awk programon belül nem volt szükség a sorok összefűzésére, itt ugyanis az utasításokat vagy blokkokat újsor karakter is elválasztja egymástól. Ugyanakkor nem hiba, ha itt is használjuk a \ karaktert.

Feltételes utasítás

Szabályos kifejezések illeszkedését akár az `if` parancssal is vizsgálhatjuk. Az „illeszkedik” jele a „~” karakter, míg az illeszkedés tagadását a C nyelvben szokásos módon a „! ~” kombinációval oldhatjuk meg. Ezzel a módszerrel a fenti program így fest:

```
ls -l | awk \
'BEGIN {meret=0}
{if($0!~/^d/) meret=meret+$5}
END{print meret}'
```

Mint látható, az `if` parancsban feltételként most a sor eleji „d” betű illeszkedésének tagadását adtuk meg. Az összehasonlítás alapja `$0`, vagyis a teljes sor. Az `awk` tehát minden sorra kiértékeli a feltételt, és ha igaznak találja, végrehajtja az `if` után álló (egyetlen!) utasítást. Ha nem teljesül a feltétel, akkor az `else` kulcsszó utáni parancs érvényes, de a hamis ágat nem feltétlenül kell megadni (példánkban `sincs`). Mind az `if`, mind az `else` után megadható több végrehajtandó utasítás (utasításblokk) is, kapcsos zárójelek közé zárva.

Az `if` utáni logikai kifejezés természetesen nem csak egy szabályos kifejezéssel való összehasonlítást tartalmazhat, hanem közönséges matematikai relációt is. A következő példában kiválasztjuk a pillanatnyi könyvtárból a legnagyobb méretű fájlt:

```
ls -l | awk \
'BEGIN {max=0}
{if($5>max) max=$5}
END {print "A legnagyobb fájl "max" bájt hosszúságú."}'
```

Ha a fájl nevét is ki akarjuk íratni, az `if` után már két parancsból álló blokkot kell megadnunk:

```
ls -l | awk \
'BEGIN {max=0}
{if($5>max) {max=$5 ; nev=$9}}
END {print "A legnagyobb fájl: "nev"\nMérete: "max}'
```

Az `ls` parancs kimenetében a kilencedik mező tartalmazza a nevet. A név kiírása után használt „`\n`” szimbólum egy vezérlőkarakter, nevezetesen a sorvége jel. A méret tehát már új sorban íródik ki.

Következő példaprogramunkban kiszámítjuk a pillanatnyi könyvtárban az átlagos fájlméretet. (Ennek természetesen túlságosan sok értelme nincs, de gyakorlásnak

megteszi.) Nyilván elő kell állítanunk a fájlméretek összegét és közben meg kell számolnunk, hány bejegyzés volt a könyvtárban:

```
ls -l | awk \
'BEGIN {osszeg=0; darab=0}
{if($1~/^-/) {osszeg=osszeg+$5; darab=darab+1}}
END {print "Az átlagos fájlméret: "osszeg/darab}'
```

Az előzőek ismeretében a megoldás különösebb magyarázatot nem igényel, bár azt érdemes megfigyelni, hogy az if feltételében itt a „-” karakter illeszkedését vizsgáltuk a „d” „nem-illeszkedése” helyett.

Beépített függvények

Az awk a matematikai függvényeken kívül még számos egyéb, elsősorban karakterláncok kezelésével kapcsolatos belső függvényt rendelkezik. Ezekről a program leírásában olvashatunk bővebben. Itt csak a karakterlánc hosszát visszaadó függvényt mutatom be. Legyen a feladat egy szövegfájl leghosszabb sorának kiíratása:

```
cat szoveg.txt | awk \
'BEGIN {max=0 ; sor=""}
{if(length($0)>max)
{max=length($0) ; sor=$0}
}
END {print sor}'
```

A program felépítése különösebb magyarázatot most sem igényel. A legfontosabb talán az, hogy az awk függvényeit pontosan ugyanúgy kell meghívni, mint a C nyelvben (kerek zárójelek a név után).

Ciklusok

Az awk-ban, mint minden programnyelvben, ciklust is szervezhetünk. Ennek formája teljesen azonos a C nyelvben szokásossal. Példaként írassuk ki 1-től 10-ig a számokat. Írjuk be egy tetszőleges szövegszerkesztővel a program.awk nevű fájlba a következő programot:

```
BEGIN {
for(i=1;i<11;i++) print i
}
```

A program a következő módon hajtható végre:

```
echo "" | awk -f program.awk
```

Mind a program, mind annak végrehajtása némi magyarázatot igényel. A programban csak egy BEGIN blokk van, mivel a számok kiíratását nem a sorok feldolgozássával kapcsolatban végezzük, hanem attól függetlenül. A for utáni zárójel három mezője a ciklusváltozó kezdőértékét ($i=1$), a leállási feltételt ($i<11$) és a ciklusváltozó kezelését jelenti. A „ $++$ ” az úgynevezett növelő művelet (inkrementáló operátor), amely eggyel növeli az előtte levő változó értékét. (Termézesesen létezik csökkentő művelet is, ez a „ $--$ ”.)

Bár a sorokat feldolgozó főprogram ebben az esetben nem is létezik, a program hívásakor mindenkor mindenkor adunk kell az awk-nak valamilyen bemenetet. Ennek hiányában ugyanis úgy veszi, hogy az a billentyűzetről fog érkezni. Az alapértelmezett UNIX-szerű viselkedésnek megfelelően tehát kiírja ugyan a számokat, de azután nem adja vissza a készenléti jelet, hanem türelmesen várja, hogy gépeljünk valamit (legalább egy ENTER-t). Ha ezt megtesszük, az ENTER leütése után termézesesen nem fog semmit csinálni a bemenettel, de rend a lelke mindennek...

Az echo segítségével tehát egy „semmit” (egészen pontosan egy újsor karaktert) küldünk a bemenetre, amit az awk a nem létező főprogrammal fel is dolgoz.

Az awk termézesesen ismeri a while és az until ciklusokat is. Ezek utasításformája szintén megegyezik a C nyelvvel, amint azt a következő egyszerű példa is mutatja.

```
echo | awk 'BEGIN {
    i=0
    while(i<10)
    {
        print i
        i++
    } }'
```

Fájlok kezelése

Az awk egyszerre több fájl tartalmát is képes kezelni. Megtehetjük például, hogy a szabványos bemenetről érkező adatok fogadása közben egy vagy több másik fájl tartalmát is olvassuk és kezeljük. A

```
getline < "fájlnév"
```

parancsal a megadott fájlból olvashatjuk be a következő sort. Az awk az első ilyen műveletnél megnyitja a fájlt, és nyitva is tartja. Természetesen innen tudja azt is, hogy melyik a „következő” sor.

Az így beolvasott sorhoz ugyanúgy a \$0 változón keresztül férhetünk hozzá, mint ha a szabványos bemenetről érkezett volna. Természetesen a mezőkre tördelés is ugyanúgy zajlik.

Ha a print vagy a printf parancs kimenetét akarjuk egy fájlba irányítani, azt a > és a >> műveleti jelekkel tehetjük meg, melyek jelentése ugyanaz, mint a héjprogramknál. A kimenet átirányítására is érvényes, hogy a megadott fájl a program teljes működése alatt nyitva marad. Így ha több print parancsot irányítunk ugyanabba a fájlba, valamennyi kimenet meg fog benne jelenni, nem csak az utolsó.

Ha a program egy pontján le akarunk zárni egy már megnyitott fájlt, azt a

close(fájlnév)

parancsal tehetjük meg.

Buktató

Akár a getline parancsot használjuk, akár a kimenetet irányítjuk fájlba, a fájlnévet mindenkor kettős idézőjelek között kell megadni.

Összetett gyakorlatok

Az elmondottak – bár a leggyakoribb elemeket tartalmazzák – természetesen csak töredékét jelentik az awk nyelv képességeinek. A teljes ismertetést e könyv terjedelme nem teszik lehetővé, az Olvasó azonban több könyvet is találhat ezzel a témaival kapcsolatban az Interneten, illetve a könyvesboltokban.

A továbbiakban – az előző fejezethez hasonlóan – néhány összetettebb gyakorlaton mutatom be az awk használatának lehetőségeit.

Telefonkönyv

Az előző fejezetben láttuk, hogyan lehet a sed segítségével betűrendbe szedett telefonkönyvet készíteni egy telefonszám–név párokat tartalmazó listából. Lássuk, hogy néz ki az awk segítségével megvalósított megoldás!

5.1. lista

```

1: #!/bin/sh
2: # Telefonkönyv készítése az AWK segítségével
3:
4: cat $1 | awk \
5: '{if( $2~/Dr\./ || $2 ~ /Prof\./) drprof=1; else drprof=0;
6:   for(i=2+drprof;i<=NF;i++) printf "%s ",$i
7:   if (drprof!=0) printf "%s ",$2
8:   printf "%s\n",$1
9: }' | sort

```

Az 5. sorban szereplő feltételes utasításban megvizsgáljuk, hogy az éppen feldolgozás alatt álló sor második mezőjében szerepel-e a "Dr." vagy a "Prof." rövidítés. Azt a lista szerkezetéből tudjuk, hogy ha egyáltalán szerepel, akkor csak a második mezőben lehet, az első ugyanis a telefonszám. Ha találtunk illeszkedést, a drprof változó értékét 1-re, egyébként nullára állítjuk.

A 6. sorban kezdődő ciklus végigmegy a nevet tartalmazó mezőkön. Ezek a 2. vagy 3. pozíción kezdődnek, attól függően, hogy az illetőnek van-e tudományos fokoza-ta. Éppen ennek a dilemmának az eldöntésére szolgál a drprof változó. Leállási feltételként szándékosan nem az induló értéknél kettővel nagyobb számot, hanem NF-et (a mezők száma) adtam meg, ugyanis egyáltalán nem biztos, hogy a lista minden tagjának két szóból áll a neve.

A 6. sorban nem a print, hanem a printf parancsot használtam a kiíratásra. Ennek legfőbb oka az, hogy a print magától hozzátesz a kimenethez egy újsor karaktert, amiről lebeszélni sem lehet. Ez esetünkben azzal járna, hogy a vezeték és a keresztnév külön sorba kerül. A printf viselkedése ezzel szemben a C nyelvből ismert formátumkódokkal határozható meg, az újsort pedig csak akkor küldi ki, ha a ebben szerepel a „\n” vezérlőkarakter (lásd a 8. sort).

A 7. sorban írjuk ki a név után a „Dr.” vagy „Prof.” rövidítést, de csak akkor, ha az elején is volt ilyen (drprof értéke nem nulla). Végül a kimenet a sort parancs be-menetére kerül.

Kettes számrendszer

Írunk olyan héjprogramot, amely egy parancssori paraméterként megadott tetsző-leges decimális számot kettes számrendszerbeli megfelelőjévé alakít.

5.2. lista

```

1: #!/bin/sh
2: # Decimális szám átalakítása kettes számrendszerbeli megfelelőjévé
3:
4: echo $1 | awk \
5: '{num=$0}
6: END {max=1
7: while(num/max>=1) {max*=2; k++}
8: max/=2; k--;
9: for(i=k;i>=0;i--) {digit=(num-(num % max))/max
10: printf "%i", digit
11: num-=digit*max
12: max/=2}
13: print ""
14: }'

```

A feldolgozást végző teljes „főprogramot” az 5. sor tartalmazza. Itt minden össze annyi történik, hogy az awk bemenetére küldött számot elhelyezzük a num nevű változóban. A sorok feldolgozása ezzel be is fejeződött. A matematikai lényeg az awk számára már csak utómunkálat, így az teljesen az END blokkba került.

A 7. sorban látható ciklus megkeresi kettőnek azt a hatványát, amivel a megadott szám (num) már nem osztható. A max változó tartalmazza magát a hatványt, k pedig a kitevőjét (helyiérték). A számításhoz nekünk természetesen arra a legmagasabb kitevőjű kettőhatványra van szükségünk, amellyel a megadott szám még építen osztható, így a 8. sorban „eggyel visszalépünk” az előző ciklusban.

A kettes számrendszerbeli szám jegyeit a 9. sorban számítjuk ki. Itt talán annyit érdemes megjegyezni, hogy a % a maradékos osztás (modulo) jele. A számjegyek sorra a digit nevű változóba kerülnek, amit a 10. sorban a printf parancs segítsével íratunk ki. Erre azért van szükség, mert az egyszerű print parancs újsort is küldene a számjegyek után. A következő két sorban a num változóba a pillanatnyi helyiértékkal való osztás maradékát tesszük, a helyiértéket tároló max tartalmát pedig felére csökkentjük. (Figyeljük meg a C stílusú -= és /= műveleti jeleket.) A for ciklus magja a 12. sorban záródik. A 13. sorban egy „üres” print parancs segítségével egy újsor karaktert küldünk a kiírt számjegyek után. Ne felejtse el, hogy a sima print parancs \$0-t, vagyis esetünkben a feldolgozandó számot írná ki.

Átlagolás

Tegyük fel, hogy több ember (egy osztály) több dolgozatot írt, de nem mindenki írta meg mindenkit. Az egyes dolgozatok eredményeit külön fájlokba írtuk be név, érdemjegy sorrendben. Számítsuk ki az egyes emberek átlagát és készítsünk összefoglaló listát. (Az egyszerűség kedvéért tegyük fel, hogy mindenkinél két tagból áll a neve és hogy nincs két egyforma nevű ember az osztályban.)

Ha három dolgozatot írtak, a három lista így nézhet ki:

11.txt

Kovács István	3
Nagy József	2
Stréber Elemér	5
Rosszcsont Géza	1
Rózsa Sándor	4
Kucug Balázs	2
Nemecsek Ernő	5
Virág Rózsa	5
Kala Pál	2

12.txt

Nagy József	4
Stréber Elemér	1
Rosszcsont Géza	3
Kucug Balázs	2
Nemecsek Ernő	4
Virág Rózsa	5
Kala Pál	2

13.txt

Kovács István	2
Stréber Elemér	4
Rózsa Sándor	5
Kucug Balázs	3
Nemecsek Ernő	4
Virág Rózsa	4

Az első hiba nyilván az egyes emberek „beazonosítása” a három listában. Sajnos ezzel kapcsolatban a listák semmiféle különleges tulajdonságát nem tudjuk kihasználni. Nem mindenki szerepel minden listában, és ha szerepel is, akkor sem tudjuk, hogy hányadikként.

Van azonban erre egy végtelenül egyszerű megoldás: ha egysítjük valahogyan a három listát, és a sort segítségével rendezzük, az azonos névhez tartozó bejegyzések maguktól egymás mellé kerülnek, függetlenül attól, hány volt belőlük és pontosan hol. Ráadásul azonnal betűrendes listát kapunk. Nincs más dolgunk te-hát, mint a cat segítségével egymás után a sort bemenetére küldeni a listák tartalmát. A következő programkód rögtön a teljes megoldást tartalmazza:

5.3. lista

```

1: #!/bin/sh
2: # Dolgozatok átlaga
3:
4: cat 1?.txt | sort | awk \
5: >{
6:     if(NR==1) {db=1; sum=$NF; nev=$1" "$2}
7:     if(nev==$1" "$2) {sum+=$NF; db++}

```

```

8:     else {print nev,sum/db; nev=$1" "$2; sum=$NF; db=1}
9:   }
10: END {print nev,sum/db}'

```

Mint látható, a 6. sor csak egyszer végrehajtódik majd végre: a bemenet első sorának feldolgozásakor. Itt tulajdonképpen értéket adunk annak a három változónak, amelyekre az algoritmus épül. A nev nevű változóban elhelyezzük a névsorban először szereplő nevét, a sum-ban az első dolgozatának eredményét, a megírt dolgozatokat számláló db-ben pedig egyet.

Az egészből a név megadása érdemel külön említést. A feltételekből tudjuk, hogy mindenkinél két tagból áll a neve, így nincs más dolgunk, mint egy változóban összefűzni \$1 és \$2 tartalmát. A karakterláncokat az awk-ban úgy fűzhetjük össze, hogy egyszerűen egymás mellé írjuk azokat.

A következő feltételes utasításban azt vizsgáljuk, hogy a következő sor első két mezőjében szereplő név megegyezik-e az előző sorban szereplővel. Ha igen, akkor még mindig ugyanannak az embernek az eredményéről van szó, tehát megfelelő módon növelni kell a sum és a db változók tartalmát. Ha nincs egyezés, az előző ember eredményeit ki kell írni, a most beolvasott sor tartalmával pedig újra fel kell tölteni a három változót.

Mivel az eredmények kiírása az else ágban történik, a névsorban utolsó ember adatai nem íródnak ki. Erről az END blokkban külön kell gondoskodnunk.

Megjegyzés

A karakterláncok összefűzéséről

A fenti programkód kapcsán talán az Olvasónak is eszébe jutott a kérdés: hogyan fűzzünk össze két karakterláncot, ha minden kettőt egy-egy változóban tároljuk?

Az első ötletünk nyilván az, hogy a fenti megoldáshoz hasonlóan egyszerűen egymás mellé írjuk a két változó nevét:

```

echo "egy  kettőre" | awk '{szoveg1=$1; szoveg2=$2;
szoveg3=szoveg1szoveg2;
print szoveg3}'

```

E parancs hatására azonban a kimeneten nem jelenik meg semmi, sőt még hiba sem keletkezik. Az ok egyszerű: az awk önműködően létrehoz minden ismeretlen változót az első hivatkozásnál, így amikor szoveg3-nak megpróbálunk értéket adni, a program azt hiszi, hogy a „szoveg1szoveg2” egy új változó. Így létrehozza azt, és tartalmát

(ami nincs) áttölti szöveg3-ba. A program formailag helyes, bár egyáltalán nem azt csinálja, amit akartunk. A következő trükkös megoldás már működik:

```
echo "egy kettőre" | awk '{szoveg1=$1; szoveg2=$2;
szoveg3=(szoveg1)"szoveg2";
print szoveg3}'
```

Itt tulajdonképpen azt használtuk ki, hogy idézőjel nem szerepelhet egy változó névében. Ennek megfelelően az awk pontosan tudja, hogy itt két változóról és egy üres karakterláncról van szó. Bár az előbbi is működik, a „hivatalos” megoldás azért a kerek zárójelek használata:

```
echo "egy kettőre" | awk '{szoveg1=$1; szoveg2=$2;
szoveg3=(szoveg1)(szoveg2);
print szoveg3}'
```

Névsor

Készítsünk az előbbi adatbázisból osztálynévsort! Először egy, az előbbihez hasonló megoldás juthat eszünkbe:

5.4. lista

```
1: #!/bin/sh
2: # Osztálynévsor
3:
4: cat l?.txt | sort | awk \
5: 'BEGIN {nev=""}
6: {
7:     if($1" "$2!=nev) {print $1" "$2 ; nev=$1" "$2}
8: }
```

A több példányban előforduló sorok törlésére azonban magának az operációs rendszernek is van egy segédeszköze: a uniq nevű program. Ezzel a megoldás a következőképpen fest:

```
cat l? | awk '{print $1" "$2}' | sort | uniq
```

Fontos megjegyezni, hogy a uniq csak rendezett listával boldogul, tehát a sort ebben a feldolgozási sorban sem fölösleges!

Címlista és telefonkönyv összefésülése

Tegyük fel, hogy van két listánk. Az egyikben telefonszámok és nevek, a másikban ugyanazok a nevek és címek szerepelnek. Állítsunk elő a kettőből egy olyan közös adatbázist, amely minden ember nevét, telefonszámát és címét tartalmazza. Az egyes adatok külön sorban legyenek, az egyes emberekhez tartozó blokkokat pedig egy üres sor válassza el egymástól, valahogyan így:

```
név1
telefonszám1
cím1
```

```
név2
telefonszám2
cím2
```

A két lista legyen a következő:

telefon.txt	cimek.txt
1333444 Dr. Bubó Bubó	Dr. Bubó Bubó Odvás Tölgy tér 2.
1777888 Ursula Nővér	Ursula Nővér Szavanna sétány 55/A
3141592 Ele Fáni	Ele Fáni Szavanna sétány 57.
1428571 Csőr Mester	Csőr Mester Jegenye utca 47.
1111111 Teknőc Ernő	Teknőc Ernő Mocsaras part balra
7692307 Prof. Boa Konstrektor	Prof. Boa Konstrektor Erdő körút 3.

Hogy a helyzetet bonyolítsuk, tegyük fel, hogy az első listában a telefonszámot és nevet egy szóköz, a másodikban a nevet és címét egy tabulátor választja el egymástól. Ugyanakkor az egyszerűség kedvéért feltehetjük azt is, hogy a két listában a nevek azonos sorrendben szerepelnek. (Ez azért lényegtelen most számunkra, mert ha nem így lenne, a sort segítségével akkor is könnyedén ilyen állapotba hozhatjuk őket.)

Itt egyszerre két fájlt kell kezelnünk, tehát biztosan szükségünk lesz az awk getline parancsára. Ráadásul a mezőelválasztó karakter sem egyezik a két fájlban, tehát ennek az értékét is cserélgetnünk kell majd. Mindezek ismeretében a megoldás így fest:

5.5. lista

```

1: #!/bin/sh
2: # Két fájl sorainak összefésülése
3:
4: cat telefon.txt | awk \
5: '{for(i=2;i<=NF;i++) printf "%s ", $i
6:   print ""
7:   print $1
8:   FS="\t"
9:   getline < "cimek.txt"
10:  print $2
11:  print ""
12:  FS=" "
13: }'

```

Az 5. sorban látható ciklus az első fájl egy során megy végig a második mezőtől kezdve. Tudjuk, hogy az első mező a telefonszámot tartalmazza, amit majd csak a név után kell kiíratni. A név hosszáról nem kell semmit feltételeznünk, ha a ciklusmagot NF-ig, vagyis az utolsó mezőig hajtjuk végre. Megint a printf parancsot használjuk kiíratásra, mivel ezt rá lehet venni, hogy ne küldjön sorvége jelet, ha nem akarjuk.

A 6. sorban látható „üres” print parancs írja ki a név után a sora melést, majd a 7. sorban a telefonszám következik.

Most következik majd a második fájlban tárolt adatok feldolgozása, ehhez azonban a 8. sorban előbb át kell állítanunk a mezőelválasztó karaktert pontosan egy tabulátorra. A 9. sorban beolvassuk a második fájl következő sorát. (Emlékezzünk rá, hogy ehhez nem kell külön megnyitnunk ezt a fájlt, a getline magától megteszi az első olvasási művelet előtt.) A 10. sorban \$2 már a tabulátor (és nem az első szóköz) után következő sor részleteket jelenti. Ezt egy üres sor kiíratása követi, majd a 12. sorban visszaállítjuk a mezőelválasztó karaktert az alapértelmezett értékre.

Egy sor, több sor...

Az előző részben megírt program kimenete valahogyan így néz ki:

Dr. Bubó Bubó
1333444
Odvas Tölgy tér 2.

Ursula Nővér
1777888
Szavanna sétány 55/A

Most tegyük fel, hogy csak ez a fájl áll rendelkezésünkre, de át szeretnénk alakítani úgy, hogy egy ember adatait egyetlen sor tartalmazza, név, telefonszám, cím sorrendben. Írunk erre is héjprogramot!

A feldolgozásnál nyilván megint a fájl szerkezeti sajátosságait kell kihasználnunk. Tudjuk, hogy minden negyedik sornál van „váltás” az adatbázisban, vagyis ott kezdődik egy új szövegblokk. A feladatot tehát úgy oldhatjuk meg, hogy elkezdjük „összemásolni” a sorok tartalmát, és közben figyeljük, mikor érkezünk negyedik sorra (mikor lesz NR négygyel maradék nélkül osztható). Amikor ez bekövetkezik, kiíratjuk az addig keletkezett karakterláncot, majd kezdjük előlről az egészet. Íme a kód:

5.6. lista

```
1: #!/bin/sh
2:  # Szövegblokkok kezelése I.
3:  # Bonyolult megoldás: a sorszámok figyelése
4:
5:  cat $1 | awk \
6:  '{if (NR%4==1)
7:   egysor=$0;
8:   else
9:   {
10:     if (NR%4!=0)
11:       egysor=egysor", \"$0";
12:     else
13:     {
14:       print egysor;
15:       egysor="";
16:     }
17:   }
18: }'
```

Látható, hogy a fent megfogalmazott egyetlen feltétel (NR maradék nélkül osztható – négygyel) helyett itt kettő van. Erre az egyes bejegyzéseket elválasztó vessző miatt volt szükség. Ha ugyanis csak a 10. sorban kezdődő döntési szerkezetet használjuk, akkor az első bejegyzés (a név) előre is kerül egy vessző, ami nyilván nem kívánatos. A program kimenete a következő:

```
Dr. Bubó Bubó, 1333444, Odvás Tölgy tér 2.  
Ursula Nővér, 1777888, Szavanna sétány 55/A  
Ele Fáni, 3141592, Szavanna sétány 57.  
Csőr Mester, 1428571, Jegenye utca 47.  
Teknőc Ernő, 1111111, Mocsaras part balra  
Prof. Boa Konstrektor, 7692307, Erdő körút 3.
```

Bár a fenti program sem nevezhető különösebben bonyolultnak, ezt a feladatot sokkal egyszerűbben is megoldhatjuk:

5.7. lista

```
1: #!/bin/sh  
2: # Szövegblokkok kezelése II.  
3: # Az egyszerű megoldás: az FS és RS módosítása  
4:  
5: cat $1 | awk 'BEGIN {FS="\n"; RS=""} {print $1, "$2", "$3"}'
```

A mezőelválasztó karaktert (FS) már többször használtuk, a korábban említett rekordelválasztót (RS) azonban még soha. A „rekord” az awk számára a bemeneti szövegnek az a része, amit az FS tartalma alapján mezőkre törde. „Normális” esetben RS értéke egy újsor, vagyis a program soronként dolgozza fel a bemenetet. FS és RS értéke azonban bármi lehet és ebben a megoldásban éppen ezt használjuk ki.

A sorok feldolgozása előtt FS értékét újsor karakterre, RS-t pedig üres sorra állítjuk. Vegyük észre, hogy ez pontosan igazodik az általunk feldolgozni kívánt fájl belső szerkezetéhez: a „mezők” külön sorokban szerepelnek, a különálló rekordokat pedig üres sorok választják el egymástól.

6

A héjprogramok alapvető építőelemei

100
99
98
97
96
95
94
93
92
91
90
89
88
87
86
85
84
83
82
81
80
79
78
77
76
75
74
73
72
71
70
69
68
67
66
65
64
63
62
61
60
59
58
57
56
55
54
53
52
51
50
49
48
47
46
45
44
43
42
41
40
39
38
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1

Akármilyen nyelven fejlesztünk is programot, egy idő után számos olyan építőelemet fedezhetünk fel saját munkánkban, amit újra és újra felhasználunk. Azonos módon kezeljük például a parancssori paramétereket és kapcsolókat, azonos módon hozzuk létre az átmeneti fájlokat és azonos vagy legalábbis nagyon hasonló módon készítünk önhívó algoritmusokat. Ebben a fejezetben a héjprogramok ilyen „újrahasznosítható” elemeiről lesz szó.

A parancssori paraméterek és kapcsolók kezelése

Parancssori paraméterek hiányának felismerése

Ha egy UNIX segédprogramnak nem adunk meg semmilyen parancssori kapcsolót vagy paramétert, akkor általában kiír egy rövid tájékoztató szöveget a lehetőségekről. Mivel a héjprogramok legtöbbször szintén a „segédprogram kategóriába” tartoznak, célszerű ezt a viselkedésformát ezekben is megvalósítani.

Tegyük fel, hogy olyan héjprogramot írunk, amelynek legfeljebb három kapcsolója (-a , -b és -c) lehet, és ezeken kívül egyetlen feldolgozandó fájl nevét várja parancssori paraméterként.

Azt, hogy a felhasználó hány parancssori paramétert adott meg, illetve hogy adott-e meg egyáltalán valamit, legegyszerűbben a héj \$# nevű belső változójának vizsgálatával állapíthatjuk meg:

6.1. lista

```
1: #!/bin/sh
2: # Tájékoztató szöveg kiíratása
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -eq 0 ]
7: then
8:   echo "Használat: $PROGRAMNEV [-a] [-b] [-c] <fájlnév>"
9:   exit 1
10: fi
```

Különösebb magyarázatra csak a 4. sor, illetve az ott szereplő PROGRAMNEV nevű változó szorul. A basename parancs egy program nevét adja vissza annak elérési útvonala nélkül. Ez esetünkben azért hasznos, mert a héjprogramot az elérési út

megadásával akárhonnan el lehet indítani, ebben az esetben viszont a \$0 az útvonalat is tartalmazni fogja. Ugyanakkor a tájékoztató szövegben teljesen fölösleges az útvonal megadása, ott valóban csak a program nevére vagyunk kíváncsiak. A 9. sorban az 1-es visszatérési értékkel jelezzük, hogy a program nem teljesen szabályosan állt le.

Kapcsolók felismerése és fájl létezésének vizsgálata

Következő lépésként meg kell vizsgálnunk, hogy a felhasználó milyen kapcsolókat adott meg, azok megadása helyes volt-e, illetve hogy az általa megnevezett fájl létezik-e egyáltalán.

Tegyük fel, hogy programunk egyszerre csak egy fájlt képes feldolgozni. Ez azt jelenti, hogy a parancssori paraméterek száma legfeljebb négy lehet. Ha a felhasználó ennél többet adott meg, azt nyugodtan vehetjük hibának, és ismételten kiírhatjuk a tájékoztató szöveget.

Ha a paraméterek száma megfelelő, akkor egyenként kell azokat megvizsgálnunk. Ezt a legegyszerűbben egy olyan for ciklussal tehetjük meg, amely az összes megadott paraméteren, vagyis \$*-on megy végig. Az egyes kapcsolók, illetve a fájl felismerésére egy case szerkezet szolgálhat.

6.2. lista

```

1: #!/bin/sh
2: # Tájékoztató szöveg kiíratása, kapcsolók ellenőrzése
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -eq 0 -o $# -gt 4 ]
7: then
8:   echo "Használat: $PROGRAMNEV [-abc] <fájlnév>"
9:   exit 1
10: fi
11:
12: sorszam=1
13: for i in $*
14: do
15:   case $i in
16:     "-a") echo "-a kapcsoló megadva";;
17:     "-b") echo "-b kapcsoló megadva";;
18:     "-c") echo "-c kapcsoló megadva";;
19:     *) if [ $sorszam -ne $# ]

```

```
20:         then
21:             echo "Hibás kapcsoló!";
22:             exit 2;
23:         else
24:             FAJLNEV=$i
25:         fi;;
26:     esac
27:     sorszam=`expr $sorszam + 1`
28: done
29:
30: echo "A megadott fájl neve: $FAJLNEV"
31:
32: if [ ! -f $FAJLNEV ]
33: then
34:     echo "A megadott fájl ($FAJLNEV) nem létezik!"
35:     exit 3
36: fi
37:
38: exit 0
```

A 6. sorban a paraméterek számának ellenőrzése kiegészült a fent említett felső határral. (A test parancs -o kapcsolója a logikai VAGY utasítás.)

A 13. sorban induló ciklusban vizsgáljuk meg a parancssorban megadott kapcsolókat és az utolsó pozíión szereplő fájlnevet. A sorszam nevű változóra azért van szükség, mert ezen keresztül ismerhetjük fel, hogy az utolsó parancssori paraméterhez érkeztünk. Ha ennél előbb következne olyan karakterlánc, amely nem értelmezhető érvényes kapcsolóként, a program akkor is hibaüzenettel válaszol (19-22. sorok). Végül a 32-36. sorokban megvizsgáljuk, hogy a felhasználó által megnevezett fájl létezik-e. Erre a test parancs -f kapcsolója szolgál. A logikai kifejezés elején megadott „!” a logikai tagadás (! -f = „nem létezik”).

Kapcsolók egybeírása

A UNIX segédprogramoknak több kapcsolót általában úgy is megadhatunk, hogy a „-” jelet csak egyszer írjuk ki (pl. -abc). A fenti program ezt a formát nyilván nem képes felismerni, hiszen egy ilyen karakterláncot hibás kapcsolónak tekint. Erré is van azonban megoldás:

6.3. lista

```
1: #!/bin/sh
2: # Egybeírt kapcsolókat is felismer
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -eq 0 -o $# -gt 4 ]
7: then
8:   echo "Használat: $PROGRAMNEV [-abc] <fájlnév>"
9:   exit 1
10: fi
11:
12: sorszam=1
13: for i in $*
14: do
15:   if [ $sorszam -eq $# ]
16:   then
17:     FAJLNEV=$i
18:   else
19:     case $i in
20:       -* )
21:         # Hibás kapcsoló keresése
22:         if echo $i | grep '[^-abc]' > /dev/null
23:         then
24:           echo "Hibás kapcsoló!"
25:           exit 2
26:         fi
27:
28:         # Érvényes kapcsolók keresése
29:         for kapcsolo in a b c
30:         do
31:           if [ `echo $i | grep $kapcsolo | wc -l` -eq 1 ]
32:           then
33:             echo "-$kapcsolo megadva"
34:             fi
35:           done;;
36:         *) echo "Hibás kapcsoló!"; exit 2;;
37:       esac
38:     fi
39:
40:     sorszam=`expr $sorszam + 1`
41:
42: done
43:
44: echo "A megadott fájl neve: $FAJLNEV"
45:
46: if [ ! -f $FAJLNEV ]
```

```
47: then
48: echo "A megadott fájl ($FAJLNEV) nem létezik!"
49: exit 3
50: fi
51:
52: exit 0
```

E program szerkezete és működési logikája gyakorlatilag azonos az előzőével. A lényeges eltérést a 13. sorban kezdődő ciklus magjában láthatjuk.

Az itt megfogalmazott döntési szerkezet az utolsó parancssori paramétert minden-képpen fájlnévnek tekinti (15. sor), tehát meg sem vizsgálja abból a szempontból, hogy értelmezhető-e kapcsolóként.

A 19. sorban kezdődő case szerkezet egyenként megvizsgálja az összes „-” jellel kezdődő (-*) parancssori paramétert, előbb abból a szempontból, hogy tartalmaznak-e a -, a, b és c karakterektől eltérő jelet ([^-abc]). Ha igen, akkor az nyilvánvalóan hibás kapcsoló, tehát a program leáll. Ugyanez lesz az eredmény, ha a felhasználó a „-” jelet felejtette le.

Ha az adott parancssori paraméter „túlélte” ezeket a bevezető vizsgálatokat, akkor a 29. sorban induló ciklus egyenként megvizsgálja benne az a, b és c betűk jelenlétét.

Megjegyzés

Figyeljük meg, hogy a 22. és a 31. sorban megfogalmazott döntési feltételnél két különböző eljárást használtam.

Az első esetben a logikai döntés alapja a grep parancs visszatérési értéke (vagyis a \$? tartalma). Ha volt találat, ez az érték 1 (logikai igaz), ha nem, akkor nulla (logikai hamis). A kimenetet azért kell átírányítani a /dev/null-ba, mert – az alapértelmezett viselkedésnek megfelelően – találat esetén a grep kiírná a hibás kapcsolót a képernyőre. A második esetben egy numerikus értéket állítunk elő éppen ennek a kimenetnek a felhasználásával, úgy, hogy a wc parancs segítségével megszámoltatjuk a sorait. (A -l kapcsoló hatására a wc csak a sorokat számlálja.) Jelen esetben ez az érték nyilván csak nulla (nem volt találat) vagy egy lehet. A megadott parancsot csak ez utóbbi esetben kell végrehajtani (-eq 1).

A getopt parancs

A parancssori kapcsolók kezelése annyira általános feladat, hogy maga a UNIX, illetve a héj is rendelkezik egy külön erre a célra szolgáló getopt nevű paranccsal. (A Bash esetében ez beépített művelet.) A parancs formája a következő:

```
getopt "lehetséges kapcsolók" változónév
```

A getopt tulajdonképpen pontosan ugyanazt teszi, mint előző példaprogramunk: végignézi a „–” jellel kezdődő parancssori paramétereket, és összehasonlíta azokat egy, a felhasználó által megadott mintával. Ha érvényes kapcsolót talál, elhelyezi azt a megadott változóban. Egyszeri meghívásával csak egy kapcsolót kezelhetünk, de azt megjegyzি, hogy éppen hol tartott. Visszatérési értéke igaz, ha talált még feldolgozandó paramétert és hamis, ha már nem.

Ha hibás kapcsolót talál, alapértelmezés szerint hibaüzenetet küld a kimenetre, a megadott változóban pedig egy „?” karakter helyez el. A hibaüzenetet úgy nyomhatjuk el, ha az érvényes kapcsolók felsorolását egy kettősponttal kezdjük.

Mindezek ismeretében előző programunkat a következőképpen írhatjuk át:

6.4. lista

```

1: #!/bin/sh
2: # A getopt parancs használata
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -eq 0 -o $# -gt 4 ]
7: then
8:   echo "Használat: $PROGRAMNEV [-abc] <fájlnév>"
9:   exit 1
10: fi
11:
12: while getopt ":abc" KAPCSOLO
13: do
14:   case $KAPCSOLO in
15:     "a") echo "-a kapcsoló megadva";;
16:     "b") echo "-b kapcsoló megadva";;
17:     "c") echo "-c kapcsoló megadva";;
18:     "?") echo "Hibás kapcsoló"; exit 2;;
19:   esac
20: done
21:
22: eval "FAJLNEV=$$#"

```

```

23: echo "A megadott fájl neve: $FAJLNEV"
24:
25: if [ ! -f $FAJLNEV ]
26: then
27:   echo "A megadott fájl ($FAJLNEV) nem létezik!"
28:   exit 3
29: fi
30:
31: exit 0

```

A 11. sorig a program változatlan. A 12. sorban induló while ciklus a getopt s visszatérési értékét használja, vagyis addig fut, amíg van feldolgozandó parancssori kapcsoló. A 14. sorban induló vizsgálat is a szokásos, attól eltekintve, hogy a getopt a kapcsolókból magától eltávolítja a „-” karaktert, illetve hogy a hibás kapcsolóra most a „?”-tal.

Mivel a getopt kizárólag a kapcsolókat, vagyis a „-” jellet kezdődő parancssori paramétereket kezeli, a fájlnév beolvasásáról külön kell gondoskodnunk. A 22. sorban ezt egy meglehetősen körmönfönt módszerrel valósítjuk meg. A logikus megoldás a „FAJLNEV=\$\$#” lenne, ez azonban sajnos formailag hibás. (Azért az, mert a \$\$ önmagában is értelmes, és a héjprogram folyamatazonosítóját jelenti.)

Hogy ezt a buktatót kikerüljük, előbb összeállítjuk egy karakterlánc formájában azt a végrehajtandó parancsot, amely már nem a \$# szimbólumot, hanem ennek behelyettesített értékét tartalmazza. Ezután az eval segítségével végrehajtjuk azt.

Fájl típusának vizsgálata

Az operációs rendszerek számára a fájl egyszerűen bájtok sorozata. Ez egyben azt is jelenti, hogy elméletileg a fájloknak tulajdonképpen nincs típusuk. Ugyanakkor természetesen egészen máshogyan néz ki belülről egy futtatható állomány és a telefonkönyvünk tároló szövegfájl.

Mármost ha írtunk egy héjprogramot, ami szövegfájlokon végez valamilyen műveleteket (például bekeretezi a szöveget), első közelítésben semmi nem akadályozza meg a nyájas felhasználót abban, hogy bemenetként bármit megadjon neki. Ugyanakkor egészen érdekes dolgok történnek már akkor is, ha egy ilyen állomány tartalmát mondjuk a cat parancssal megpróbáljuk „megjeleníteni” a képernyőn. Azon kívül, hogy a „szöveg” nem lesz túl olvasmányos, általában terminálunk beállításai-ban is érdekes, a normális használattal összeegyeztethetetlen változások mennek végbe, amin aztán rendszerint csak a kijelentkezés és újbóli bejelentkezés segít.

Összefoglalva tehát célszerű felvérteznünk programunkat némi értelemmel, ami lehetővé teszi számára, hogy észrevegye a felhasználó tévedését.

UNIX rendszeren a `file` nevű segédprogrammal állapíthatjuk meg egy fájl típusát. A `file` „beleolvas” a megadott fájl első szakaszába, és bizonyos jellegzetes mintákat keres benne. Ezek alapján dönti el, hogy az állomány mit tartalmaz, és erről egysoros szöveges üzenet formájában ad tájékoztatást. A héjprogramokban ezt az egy sort mi is felhasználhatjuk a típus megállapítására, úgy, hogy a `grep` segítségével bizonyos szavakat keresünk benne. Példánknál maradva, ha azt akarjuk, hogy programunk csak szövegfájlokkal működjön, akkor a „`text`” szót kell keresnünk, valahogyan így:

6.5. lista

```

1: #!/bin/sh
2: # Fájl típusának vizsgálata
3:
4: if [ `file $1 | grep 'text' | wc -l` -eq 0 ]
5: then
6:   echo "A fájl feltehetőleg nem szöveget tartalmaz!"
7: else
8:   echo "Ez egy szövegfájl."
9: fi

```

Ez a programrészlet különösebb magyarázatot a fentieken túl valószínűleg nem igényel, azt viszont érdemes megjegyezni, hogy a `file` parancs által kiírt tájékoztató szövegek nem minden rendszeren azonosak. Ha tehát hordozható programot akarunk írni, ne a teljes szöveget adjuk meg keresési feltételként, hanem – mint a fenti példában is – annak csak egy jellemző részletét.

A szabványos bemenet olvasása: szűrőként működő héjprogram

Számos olyan UNIX segédprogram létezik, amely a feldolgozandó adatokat fájlból és a szabványos bemenetről egyaránt képes fogadni. Ilyen például a már sokszor használt `grep`. Ha a keresendő minta után fájlnevet (vagy neveket) is megadunk, akkor onnan olvassa a sorokat. Megtehetjük azonban azt is, hogy a feldolgozni kívánt fájl tartalmát a `cat` parancs segítségével egy csövön át a szabványos bemenetre küldjük. Azokat a segédprogramokat, amelyek képesek erre az „átfolyó üzemmódra”, összefoglaló néven *szűrőknek* nevezzük.

Próbáljuk meg egy héjprogram segítségével „leutánozni” ezt a működést! Példaképpen írunk egy olyan programot, amely megszámolja, hogy a bemenetről érkező szövegben egy megadott betű hányszor fordul elő.

Héjprogramokban a szabványos bemenetet a `read` parancs segítségével olvashatjuk. Ennek utasításformája a következő:

 `read` változónév

A `read` soronként, vagyis minden a következő újsor karakterig olvas, és az adatokat a megadott héjváltozóban helyezi el. Ha a művelet sikeres volt, visszatérési értéke 0 (logikai IGAZ) lesz. Ha adatvége jelet érzékel, 1-es értékkel (logikai HAMIS) tér vissza. Mindez számunkra azért hasznos, mert ezt a szolgáltatást kihasználva egy egyszerű `while` ciklussal pontosan addig olvashatjuk a bemenetet, amíg van mit feldolgozni. Egy szűrőként működő héjprogram „lelke” tehát mindenkor a következőképpen néz ki:

```
 while read sor  
 do  
 .  
 .  
 done
```

Mindezzel a tudással felvértezve programunk a következőképpen fest:

6.6. lista

```
1: #!/bin/sh  
2: # Adott betű előfordulásainak megszámolása  
3: # szabványos bemenetről érkező szövegben  
4:  
5: PROGRAMNEV=`basename $0`  
6:  
7: if [ $# -ne 1 ]  
8: then  
9:   echo "Hasznalat : $PROGRAMNEV <karakter>"  
10:  exit 1  
11: fi  
12:  
13: osszesen=0  
14: egy_sorban=0  
15:  
16: while read sor  
17: do
```

```

18:     egy_sorban=`echo $sor | awk -v FS="" -v keresendo=$1 \
19:         'BEGIN {darab=0}
20:             {for(i=1;i<=NF;i++) if($i==keresendo) darab++}
21:             END {print darab}'` 
22:     osszesen=`expr $osszesen + $egy_sorban` 
23: done
24: echo $osszesen
25: exit 0

```

A 7-11. sorokban a szokásos ellenőrzést hajtjuk végre. Most parancssori paraméterként csak a keresendő betűt várjuk, így \$# értékére csak az 1-et engedhetjük meg. (Valójában azt is ellenőriznünk kellene, hogy a felhasználó pontosan egy betűt gépel-e be, de ez most nem tartozik szorosan a témahez.)

A beolvasást és feldolgozást végző ciklus a 16. sorban kezdődik. A 18-21. sorokban egy olyan awk program látható, amely betűnként halad végig a sorokon, és minden egyiknél megvizsgálja, hogy azonos-e a keresendővel. Ha igen, növeli a darab nevű számláló tartalmát, a végén pedig ezt az egyetlen számot küldi a kimenetére. Az így előállított érték csak a pillanatnyi sorban talált előfordulások száma, így a 22. sorban a részösszegeket egy újabb összeadással egy másik változóba kell gyűjtenünk.

Azt, hogy az awk minden egyes betűt külön mezőnek tekintsen, úgy lehet elérni, hogy a mezőelválasztó karakter értékeként üres karakterláncot adunk meg (-v FS= " "). Programunk egyébiránt a magyar ékezetes betűket is készségesen számlálja.

Írás a képernyőre, olvasás a billentyűzetről

Ha héjprogramunk szabványos bemenetére nem küldünk adatot, akkor ez a csatorna alapállapotban a billentyűzethez van rendelve. Ez azt jelenti, hogy az előbb tárgyalt read parancs pontosan ugyanúgy használható, de most a felhasználótól várja, hogy gépeljen valamit. Amíg ezt nem teszi meg, a program nem lép tovább.

A következő példaprogram egy egyszerű, „buta” írógép. A parancssori paraméterként megadott fájlba írja a felhasználó által gépelte sorokat, a szöveg végét pedig egyetlen sor eleji pont begépelésével jelezhetjük.

6.7. lista

```

1: #!/bin/sh
2: # Egyszerű írógép
3: # A bemenet végét magányos sor eleji pont jelzi
4:
5: OUTFILE=""
6: if [ $# -eq 0 ]
7: then
8:   while [ `echo -n $OUTFILE | wc -c` -eq 0 ]
9:   do
10:     echo -n "A kimenő fájl neve:"
11:     read OUTFILE
12:   done
13: else # A kimenő fájl neve parancssori paraméterként is megadható
14:   OUTFILE=$1
15: fi
16:
17: while read ujsor
18: do
19:   if echo $ujsor | grep ^[\.\. ] > /dev/null
20:   then
21:     echo $ujsor >> $OUTFILE
22:   else
23:     exit 0
24:   fi
25: done

```

Az 5-15. sorokban a kimeneti fájl nevét határozzuk meg. Ha van legalább egy parancssori paraméter, akkor a program az elsőt tekinti fájlnévnek, a többöt figyelmen kívül hagyja. Ha a parancssorban nem adott meg semmit a felhasználó, akkor addig kérdezgeti (while ciklus a 8. sorban), amíg be nem gépel valamit.

Ha egyszerűen leütötte az ENTER-t, vagy csupa szóközt gépelez, akkor a ciklusfeltételeiben szereplő wc -c parancs kimenetén nulla jelenik meg.

Megjegyzés

Ez a forma nem minden rendszeren fog működni. Itt ugyanis kihasználtam, hogy a Bash beépített echo parancsa a csupa szóközből álló karakterláncot egyáltalán nem jeleníti meg, hacsak az érték szerinti hivatkozást kettős idézőjelek közé nem zárjuk.

A 8. és 10. sorban az echo után látható -n kapcsoló megakadályozza, hogy az echo újsor karaktert küldjön a szöveg után. Erre a 8. sorban azért van szükség, mert a wc az újsor karaktereket is számolja, a 10. sorban pedig azért, hogy a felhasználó által begépelt fájlnév ugyanabban a sorban jelenjen meg, ahol a kérdés.

Ha megvan a fájlnév, a 17-25. sorokban látható ciklus soronként beolvassa a felhasználó által begépelt szöveget és a megadott fájl végére fűzi (">> \$OUTFILE") azokat. Közben folyamatosan vizsgálja (19. sor), hogy van-e sor eleji pont a begépelt szövegen. Ha igen, akkor kilép.

Buktató

Az echo és az újsor karakter

Avagy egy fölöttéb zavaros történet utóhatásai...

A UNIX hőskorában nagy vita volt akörül, hogy mit kell az echo parancsnak tennie, ha a „semmi” (üres karakterlánc) kiírására utasítják. Egyesek úgy gondolták, hogy ilyenkor valóban semmit nem kell kiírnia, míg mások szerint célszerűbb ilyenkor is kiküldeni a kimenetre egy újsor karaktert, ami csak „majdnem semmi”. Valahogy így született meg a -n kapcsoló, ami a kísérő újsor karakter kiírását hivatott megtiltani.

Pontosabban időnként csak lenne hivatott... Az echo ugyanis kezdetben külön program volt, akár a többi segédeszköz, később viszont beépítették egyes parancsértelmezőkbe. Az új megvalósítások némelyikeből azonban valahogy kifejtettek bizonyos képességeket, például a korábban kiválóan működő -n kapcsolót.

Szerencsére a régebbi programokkal való együttműködés biztosítása érdekében az echo hagyományos formája ezeken a rendszereken is létezik, így ha meg akarjuk kerülni a belső parancsot, csupán annyit kell tennünk, hogy kiírjuk a „valódi” echo program teljes elérési útját is (ez általában a /usr/bin/echo).

A történetnek itt akár vége is lehetne, ha – természetesen történeti okok miatt – nem lenne a UNIX-nak számos különböző, egymással időnként csak részben összeegyeztethető változata. Sajnos azonban van. Szerencsére ezek a változatok alapvetően két kategóriába sorolhatók (System V és BSD típusú rendszerek). A ma forgalomban lévő kereskedelmi UNIX-változatok gyártói általában minden rendszertípusra jellemző viselkedésformát megvalósítják a segédprogramokban, de alapállapotban – világos módon – csak az egyik szabvány szerinti viselkedés érvényesülhet. Ha mi a másikhoz ragaszkodunk, akkor ezt egy környezeti változóban jelezhetjük a segédprogramoknak. Hogy ezt a változót az adott rendszeren hogyan hívják, illetve hogy mi lehet a tartalma, a segédprogramok leírásából derül ki.

Összefoglalva tehát az Olvasó – szerencséjétől függően – időnként azt fogja tapasztalni, hogy a -n kapcsoló egyes rendszereken akkor sem működik, ha a teljes elérési út megadásával hívta meg az echo parancsot. Megoldás mindenkor van, de hogy ez pontosan mi, azt csak az echo leírásából (man echo) lehet kideríteni.

Várakozás billentyű leütésére

Számos programban előfordulnak olyan pontok, ahol a felhasználónak le kell ütnie egy tetszőleges billentyűt a folytatáshoz. A read parancs alapértelmezés szerint a következő újsor karakterig, vagyis az ENTER leütéséig várakozik a bemenetre. Ez viszont azt jelenti, hogy a „Nyomjon meg egy tetszőleges billentyűt!” utasításra a felhasználónak rögtön két (nem egészen tetszőleges) billentyűt kellene megnyomnia.

Erre a helyzetre természetesen „elődeink” is gondoltak, így a read megfelelő paraméterezésével megvalósítható a szokványos viselkedés. Íme a megoldás.

6.8. lista

```
1: #!/bin/sh
2: # Várakozás billentyű leütésére
3:
4: read -s -n1 -p "A folytatáshoz nyomjon meg egy tetszőleges billentyű!"
5: echo
6: exit 0
```

A -s kapcsolóval megtiltjuk, hogy a begépelt szöveg a képernyőn megjelenjen. A -n1 azt jelenti, hogy pontosan egy karakternyi bemenetet várunk, a -p (prompt) után pedig a kiírandó szöveget adhatjuk meg.

Átmeneti fájlok kezelése

Vannak esetek, amikor a héjprogram által előállított részeredményeket átmenetileg egy vagy több fájlban kell tárolnunk a későbbi feldolgozásig. Az „illedelmes” programok kilépésük előtt természetesen eltakarítják maguk után ezt a hulladékot.

Van azonban itt egy ennél fontosabb műszaki kérdéskör. A UNIX többfeladatos, többfelhasználós operációs rendszer, így nem zárható ki, hogy ugyanazt a programot többen is futtatják egyszerre. Ha a program egyetlen előre megadott nevet használ az általa létrehozandó átmeneti fájlok neveként, akkor az egyes példányok felülírhatják egymás adatait, ami nyilván nem kívánatos mellékhatásokkal járna.

Célszerű tehát a névben valamelyen egyedi azonosítót használni, ráadásul olyat, ami nem egyszerűen a programra, hanem annak minden példányára nézve egyedi. Ha jól belegondolunk, a UNIX által kezelt folyamatazonosító (PID) éppen ilyen tulajdonságokkal rendelkezik: soha nem lehet egy rendszeren két olyan futó program, amelyeknek megegyezik ez az azonosítója.

A folyamatazonosítóhoz a héjprogramokból a §§ belső változón keresztül férhetünk hozzá. Ha tehát az átmeneti fájl nevének például a „\$\$.tmp” karakterláncot választjuk, biztosan nem soha ütközés a programok vagy programpéldányok között.

Egy másik, talán valamivel kevésbé súlyos kérdés az átmeneti fájlok elhelyezése. Megtehetnénk természetesen, hogy minden a pillanatnyi könyvtárba vagy a programot futató felhasználó saját könyvtárába tesszük azokat, csak egyáltalán nem biztos, hogy az adott helyzetben a programnak van ezekhez írási joga. (A rendszer részét képező segédprogramoknak szinte biztosan nincs.) Kell tehát egy hely, ahol mindenkinek van írási joga, ahol minden program elhelyezheti az adatait, majd dolga végeztével törölheti azokat. A UNIX rendszereken ez a hely a /tmp könyvtár, amely gyakran külön lemezrészben (partition) található.

Összefoglalva tehát az átmeneti fájlok zökkenőmentes kezeléséhez két feltételnek kell teljesülnie:

1. Az átmeneti fájlok nevében szerepelni kell a folyamatazonosítót.
2. A programnak kilépés előtt törölnie kell az összes általa létrehozott átmeneti fájlt.

Megjegyzés

Ki mit törölhet?

A sticky bit

A UNIX fájlrendszerre érvényes jogosultsági szabályok szerint ha egy könyvtárra írási jogunk van, akkor abban bármilyen fájlt törölhetünk. A /tmp könyvtárra mindenkinek van írási joga, tehát – elvileg – bárki bármit törölhet. Ha ez valóban így lenne, az egyes felhasználók törölhetnék a mások programjai által létrehozott átmeneti fájlokat, ami azokat nyilván működésképtelenné tenné. Ennek elkerülésére szolgál

az úgynevezett „sticky bit” (kb. „ragasztó bit”), ami a /tmp könyvtárra minden rendszeren be van állítva.

A sticky bit hatására a rendszer szigorítja a törlés, illetve fájlmódosítás szabályait: hiába van valakinek írási jog a könyvtárhoz, csak azokat a fájlokat írhatja és törölheti benne, amelyeknek ő a tulajdonosa.

Nézzünk egy egyszerű példát az átmeneti fájlok kezelésére! Írunk olyan héjprogramot, amely a szabványos bemenetről érkező vagy fájlból kiolvasott szöveget bekeretezi. A keret összeállításához használjuk a |, + és - karaktereket.

A keret megrajzolásához nyilván tudnunk kell, milyen széles a szöveg, vagyis hány karakterből áll a leghosszabb sora. Ha ez megvan, akkor – célszerűen egy függvény segítségével – már csak ki kell íratnunk a sorokat úgy, hogy a megfelelő pontokra beillesztjük a keret részeit.

Ha a bemenet fájlból érkezik, nincs semmi gond. Mindössze annyit kell tennünk, hogy kétszer olvassuk végig a tartalmát. Az első menetben meghatározzuk a szélességet, a másodikban pedig kiíratjuk a megfelelően átalakított sorokat.

Ha viszont a szabványos bemenetről érkeznek a sorok, akkor nincs mód a kétszeri olvasásra. Amit egyszer beolvastunk, eltűnik a csőből. A megoldás nyilván az, hogy a feldolgozandó anyagot valahol tárolnunk kell. Bár a gond áthidalására többféle körmönfont módszert kieszelhetünk (például tárolhatnánk a teljes szöveget egy héjváltozóban), a legegyszerűbb, ha a beérkezett sorokat az első menetben átírányítjuk egy átmeneti fájlba (közben természetesen megkereshetjük a leghosszabb sort), majd a második menetben a már említett függvény segítségével ennek a fájlnak a tartalmát dolgozzuk fel.

A program egy lehetséges megvalósítása a következő:

6.9. lista

```

1:#!/bin/sh
2:# Szöveg bekeretezése
3:
4:
5:PROGRAMNEV=`basename $0`
6:
7:keret()
8:# A "bekeretezést" végző függvény
9:{
10:    also_felso="+-"
11:    i=1

```

```
12:     while [ $i -le $szelesseg ]
13:         do
14:             also_felso=$also_felso"-"
15:             i=`expr $i + 1`
16:         done
17:         also_felso=$also_felso"-+"
18:
19:     echo $also_felso
20:     while read sor
21:         do
22:             kiegészítés=`echo $sor | awk -v szelesseg=$szelesseg \
23:                         '{for(i=0;i<szelesseg-length($0);i++) k=k" "}' \
24:                         END {print k}'` \
25:             echo "| \"$sor$kiegészítés\" |"
26:         done < $1
27:     echo $also_felso
28: }
29:
30: if [ $# -ne 0 ]    # A sorok fájlkból érkeznek
31: then
32:   for NEV in $*
33:   do
34:     if [ -f $NEV ]
35:     then
36:       szelesseg=`cat $NEV | awk '{if(length($0)>hossz)
37:                                     hossz=length($0)} \
38:                                     END {print hossz}'` \
39:       keret $NEV
40:     else
41:       echo "A $NEV nevű fájl nem létezik"
42:     fi
43:   done
44: else           # A sorok a szabványos bemenetről érkeznek
45:   while read sor
46:   do
47:     echo $sor >> /tmp/$$tmp
48:   done
49:   szelesseg=`cat /tmp/$$tmp | awk '{if(length($0)>hossz)
50:                                         hossz=length($0)} \
51:                                         END {print hossz}'` \
52:   keret /tmp/$$tmp \
53:   rm /tmp/$$tmp
54: fi
```

A tényleges munkát, vagyis a bekeretezést végző függvényt a 7-28. sorok tartalmazzák. Ez a függvény a feldolgozandó fájl nevét az első paraméterében (\$1) várja. (Ügyeljünk rá, hogy ez a \$1 *nem* a program, hanem a függvény első paramétere!)

A szöveg szélességét a főprogram már meghatározta, és elraktározta a „szelesség” nevű, a függvény számára is hozzáférhető változóban. Ezen adat birtokában a 10-17. sorokban azonnal összeállíthatjuk a keret alsó és felső szegélyét. (Emlékezzünk rá, hogy karakterláncok összefűzéséhez egyszerűen csak egy más mellé kell írni a nevüket.)

A 19-26. sorokban látható ciklus küldi a kimenetre a megfelelően átalakított sorokat. A ciklus a bemenetet a \$1-ből, vagyis a függvény első paramétereként megadott fájlból veszi (26. sor). A „kiegeszites” nevű változóban annyi szóközt másol egymás után, amennyivel a pillanatnyi sor rövidebb a leghosszabbnál. Végül a 25. sorban kiírja a keret két oldalát alkotó jeleket és a megfelelően kiegészített sort.

Buktató

Egyes rendszereken (például a Linuxon) az echo a csupa szóközök-ból álló karakterláncot hordozó héjváltozó tartalmát csak akkor hajlandó megjeleníteni, ha az érték szerint hivatkozást kettős idézőjelekbe tesszük.

A főprogram 30-42. soraiban a parancssori paraméterként megadott fájlokat, a 43-52. sorokban pedig a szabványos bemenetről érkező szöveget dolgozzuk fel. A lényeg tulajdonképpen minden esetben azonos: meghározzuk a szöveg szélességét, majd a megfelelő fájlnévvel meghívjuk a keretező függvényt. Szabványos bemenet feldolgozásakor ez annak az átmeneti fájlnak a neve (/tmp/\$\$tmp), amit a leghosszabb sor keresésével párhuzamosan állítottunk elő. (A 32. sorban induló ciklus arra szolgál, hogy a program egyszerre több, névvel megadott fájl tartalmát is feldolgozhassa.)

Jelek elfogása és kezelése

Az operációs rendszerek egyik legfontosabb feladata a felhasználói programok futtatásának lehetővé tétele és a végrehajtás felügyelete. Ehhez természetesen valamilyen kapcsolat kell lennie a program és a mag (kernel) között. UNIX esetében a mag úgynevezett jelek (signals) segítségével tart kapcsolatot a futó folyamatokkal.

Minden jelnek van egy száma, egy szöveges rövidítése, és természetesen valamilyen különleges jelentése. Ha például egy héjprogram futása közben leütjük a CTRL+C billentyűket, a futó program a 2-es sorszámú jelet (SIGINT, megszakítás) fogja megkapni, és leáll. A legfontosabb jelek nevét, számát és kiváltó okát a 6.1. táblázat tartalmazza.

6.1. táblázat A héjprogramozás szempontjából legfontosabb jelek neve, száma, valamint a velük kapcsolatos rendszeresemények

Esemény	A jel neve	A jel száma	Magyarázat
exit parancs	EXIT	0	exit parancsot hajtott végre a program, vagy egyszerűen befejeződött.
Kijelentkezés	SIGHUP	1	Megszakadt a kapcsolat a terminállal.
Ctrl+C	SIGINT	2	A felhasználó leállította a programot.
Ctrl+\	SIGQUIT	3	Kilépés billentyűzetről.
kill	SIGKILL	9	Azonali, feltétel nélküli leállás.
shutdown vagy kill	SIGTERM	15	Futás befejezése (például rendszerelállásnál vagy kill parancs végrehajtásakor)
-	DEBUG	-	Nyomkövetés, minden parancs után végrehajtódik.

Látható, hogy a felsorolt jelek többsége a program leállásával kapcsolatos. Vannak azonban pillanatok, amikor nem igazán célszerű megállítani egy program futását. Ilyen helyzet lehet például, amikor a megfelelő átmeneti fájlok már létrejöttek, de a program még nem dolgozta fel, illetve nem törölte azokat.

Ha ebben a helyzetben áll le, az összes átmeneti fájl ott marad utána, ami esetleg zavarhatja a későbbi futtatást, vagy egyszerűen csak fölöslegesen foglalja valahol a helyet.

A jelek kezelésére szolgáló parancs a trap, amit többféle formában használhatunk. A

```
trap 'parancsok' jel(ek) neve vagy száma
```

formá a megadott jeleket „elfogja”, és hatásukra az egyszeres idézőjelek között megadott parancssorozatot hajtja végre. Fontos rögtön megjegyezni, hogy ez a parancssorozat csak akkor végződik a program leállásával, ha tartalmazza az exit

parancsot. Ellenkező esetben a program végrehajtása azon a ponton folytatódik, ahol a jel megérkezett. Az alábbi forma csak elfogja a jelet, de nem csinál vele semmit:

```
trap '' jel neve vagy száma
```

Ez a sor tehát tulajdonképpen az adott jel teljes figyelmen kívül hagyására utasítja a programunkat végrehajtó héjat. Végül az alábbi parancs „visszakapcsolja” a jel eredeti jelentését:

```
trap jel neve vagy száma
```

A trap parancs minden formája a program bármely pontján használható, így megfelelően kombinálva őket azt is elérhetjük, hogy programnak csak egy „érzékeny része” legyen védett a hirtelen leállítással szemben.

Különleges szerepe van az EXIT és a DEBUG jeleknek. Az EXIT jelet akkor kapja meg a program, ha szabályos módon befejezte működését, vagy valahol megszakítottuk az exit parancsal.

A DEBUG jel kezelésére megadott műveletsor valamennyi parancs után végrehajtódik, így – nevének megfelelően – nyomkövetésre használható (például kiírathatók egy változó tartalmát).

Egy program több trap parancsot is tartalmazhat, vagyis az egyes jelekhez akár külön-külön is rendelhetünk műveletsort.

A következő példaprogram „halhatatlan”, vagyis a 6.1. táblázatban említett minden jelet képes lekezelni:

6.10. lista

```

1: #!/bin/sh
2:  # Jelek kezelése
3:
4: trap 'echo "EXIT jel érkezett"' EXIT    # 0-ás jel
5: trap 'echo "HUP jel érkezett"' SIGHUP  # 1-es jel
6: trap 'echo "INT jel érkezett"' SIGINT  # 2-es jel
7: trap 'echo "QUIT jel érkezett"' SIGQUIT # 3-as jel
8: trap 'echo "TERM jel érkezett"' SIGTERM # 15-ös jel
9: trap 'echo "Sose halunk meg!"' 9   # 9-es jel (nem fogható el!)
10:
11: echo "Én egy halhatatlan program vagyok."

```

```

12: echo "A folyamatazonosítóm: $$"
13: echo "Kilépéshez üsse le az ENTER-t!"
14:
15: read s

```

A 11-13. sorokban kiíratunk némi tájékoztató szöveget, valamint a futó program folyamatazonosítóját (\$\$). A főprogram ezen kívül tulajdonképpen egyetlen read parancsból áll (15. sor), amely türelmesen várakozik arra, hogy gépeljünk neki valamit és leüssük az ENTER billentyűt.

A 4-9. sorokban minden lényeges jelhez azonos műveletet rendelünk: kiíratjuk a beérkezett jel nevét. Az EXIT jelre utaló üzenet csak a program kilépésekor fog megjelenni, az INT és QUIT jelekkel kapcsolatos szöveget pedig a megfelelő billeltyűkombinációval csalogathatjuk elő.

Ha egy másik terminálablakból kiadjuk a

```
kill <folyamatazonosító>
```

illetve a

```
kill -HUP <folyamatazonosító>
```

parancsot, a futó program a TERM, illetve a HUP jelet kapja meg. (A kill alapértelmezés szerint 15-ös, vagyis TERM jelet küld a megadott folyamatnak.)

Látható, hogy a 9. sorban megkíséreljük a lehetetlent, vagyis megpróbáljuk elfogni a feltétel nélküli leállás jelét. Ez formalag ugyan nem hibás, de a "Sose halunk meg!" szöveg akkor sem fog megjelenni, ha a

```
kill -KILL <folyamatazonosító>
```

parancsal állítjuk le a programot.

Összefoglalva tehát ha programunk valamely szakaszának „teljes védelmet” akarunk biztosítani a váratlan leállítás ellen, ezt a következő formában tehetjük meg:

```

trap 'leállító jeleket kezelő parancsok' 0 1 2 3 15
...
    a program védett része
...
trap 0 1 2 3 15

```

Zárolás

A UNIX mint többfeladatos operációs rendszer lehetővé teszi, hogy egy program egyidőben több példányban is fusson. Ugyanakkor ez a lehetőség egyes esetekben kifejezetten zavaró. Tegyük fel például, hogy egy programrendszer beállításait tartalmazó fájl szerkesztéséhez írtunk egy héjprogramot. (UNIX-on a beállításfájlok („konfigurációs” fájlok) többsége kézzel is szerkeszthető, de ha egy ilyen fájlnak szigorúan kötött vagy egyszerűen csak bonyolult a belső szerkezete, célszerűbb hozzá egy megfelelő beállítóprogramot mellékelni.)

Ha több felhasználónak is joga van módosítani a beállításokat, akkor elköpzelhető olyan helyzet, amikor egyszerre többen futtatják a beállítóprogramot. Ha ezt nem akadályozzuk meg, akkor az egyes felhasználók felülírhatják egymás munkáját, de akár tönkre is tehetik a beállításfájlt.

Ezt a legegyszerűbben úgy akadályozhatjuk meg, hogy az elsőként induló programpéldánnyal létrehozunk egy rögzített nevű és rögzített helyen lévő zárolást jelző fájlt, a program pedig valamennyi indításakor azzal kezdi működését, hogy ellenőrzi e fájl meglétét. Ha megtalálja, biztos lehet benne, hogy valaki már futtatja egy példányát, így azonnal leáll.

Kilépéskor a zároló bejegyzést természetesen törölni kell, és gondoskodnunk kell arról is, hogy e művelet elvégzése nélkül a program ne legyen leállítható. Egy lehetséges megoldást a következő példaprogram szemléltet:

6.11 lista

```

1: #!/bin/sh
2: # Példa zárolásra
3:
4: LOCKFILE="/tmp/pidlock"
5: CONFIG_FILE="proba.conf"
6:
7: trap 'rm $LOCKFILE; exit 2' 1 2 3 15
8:
9: if [ -f $LOCKFILE ]
10: then
11:   echo "Már fut egy példány a programból!"
12:   exit 1
13: else
14:   echo $$ > $LOCKFILE
15: fi
16:
17: echo -n "Új szöveg: "

```

```
18: read sor
19: echo $sor >> $CONFIG_FILE
20:
21: rm $LOCKFILE
```

A 4. és 5. sorban a zároló és a módosítandó fájl neve szerepel egy-egy változóban. A 9. sorban megvizsgáljuk, hogy a zároló fájl létezik-e. Ha igen, a program hibaüzenettel leáll. (Próbáljuk ki egy másik terminálablakból!) Ha nem, akkor létrehozzuk a zároló fájlt, és elhelyezzük benne a futó program folyamatazonosítóját.

Ez utóbbinak jelen esetben nincs különösebb jelentősége, hiszen az esetlegesen elindított többi példány csak a fájl létezését vizsgálja, a tartalmát nem. Megadhatnánk azonban olyan feltételt is, hogy az újabb példány állítsa le a régit, vagy küldjön neki egy jelet, amire annak felelnie kell. Ilyenkor kifejezetten hasznos, ha a zároló fájl a futó folyamat azonosítóját tárolja, hiszen így könnyen ki tudjuk deríteni a program azonosítóját és könnyen tudunk jeleket küldeni neki.

Kilépéskor (21. sor) töröljük a zároló fájlt, a 7. sorban megadott trap paranccsal pedig ugyanezt tesszük, ha megszakító jel érkezik. Vegyük észre, hogy a nullás, vagyis az EXIT jel most nem szerepel a trap utáni listában. Ez nem véletlen, sőt kifejezetten hiba lenne a megadása, hiszen a trap utáni műveletsor egyrészt törli a zároló fájlt, másrészt végrehajt egy exit parancsot. Utóbbi művelet viszont maga is megfelel egy EXIT jelnek, vagyis ha ez része lenne a jelek listájának, a trap újra meghívna önmagát. Természetesen a zároló fájlt már nem találná meg, így a program hibaüzenettel leállna. (Próbáljuk ki!)

Időzített végrehajtás, várakozás

Előfordulhat, hogy héjprogramunknak várakoznia kell valamilyen feltétel teljesülésére. Ez lehet egy másik folyamat befejeződése, egy fájl létrejötte, vagy egy felhasználó bejelentkezése, hogy csak néhány példát említsék.

Amennyiben a héjprogram maga indított el a háttérben egy vagy több gyermekfolyamatot, azok befejeződését a wait paranccsal várhatja meg. A paraméterek nélkül meghívott wait alapértelmezés szerint az összes háttérfolyamat befejeződéséig vár, de megadhatjuk utána egy adott gyermekfolyamat azonosítóját is.

Ha várakozás közben a feltétel teljesülését magának a programnak kell újra és újra ellenőriznie, számos esetben nem célszerű ezt az ellenőrzést a lehető legnagyobb sebességgel végezni. Ha például várakozunk egy fájl létrejöttére, tökéletesen ele-

gendő, ha egy vagy néhány másodpercenként hajtjuk végre a megfelelő vizsgálatot. Ez a felhasználó számára gyakorlatilag nem okoz érezhető lassulást, viszont nem terheljük feleslegesen a rendszert az állandó ellenőrizgetéssel.

A program lassítására szolgál a sleep parancs, amelynek paraméterként a várakozás idejét adhatjuk meg. Ez az időtartam alapértelmezés szerint másodpercen átérülő, de az m, h és d módosítókkal percek, órák sőt napok is megadhatók. A

`sleep 3 m`

parancs például 3 percig várakozik.

A következő példaprogram egy fájl létezését ellenőrzi, úgy, hogy csak három másodpercenként hajtja végre az ehhez szükséges műveletet:

6.12. lista

```

1: #!/bin/sh
2: # Lassított ciklus
3:
4: FAJLNEV="proba.txt"
5:
6: i=0
7: while [ $i -lt 10 ]
8: do
9:   echo -n $i
10:  if [ -f $FAJLNEV ]
11:  then
12:    echo "$FAJLNEV létezik"
13:  else
14:    echo "$FAJLNEV nem létezik"
15:  fi
16:  i=`expr $i + 1`
17:  sleep 3
18: done

```

A „lényeg” a 10. sorban látható, itt ellenőrizzük ugyanis a fájl létezését. A 17. sorban látható sleep parancs minden próbálkozás után 3 másodpercig várakozik, a 7. sorban induló ciklus pedig összesen tízszer próbálkozik. A programot kipróbálásához indítsuk el, egy másik terminálablakban hozzuk létre a proba.txt nevű fájlt, figyeljük meg az üzenetek sorozatát, majd töröljük a fájlt. Minderre körülbelül fél percünk van (tízszer három másodperc). A hangsúly itt a körülbelül szón van, ugyanis a sleep segítségével egészen pontos időzítés nem valósítható meg.

Megjegyzés

Létezik egy usleep parancs is, amelynek mikromásodpercben adhatjuk meg a várakozási idő hosszát, de valóban pontos időzítésre ez sem használható.

Egy hasonló fogással úgy írhatjuk át az előző szakaszban bemutatott, beállításfájl módosítására szolgáló programot, hogy ha másik futó példányt észlel, várja meg annak kilépését. Íme a megoldás:

6.13. lista

```
1: #!/bin/sh
2: # Zárolás miatti várakozás
3:
4: LOCKFILE="/tmp/pidlock"
5: CONFIG_FILE="proba.conf"
6:
7: if [ -f $LOCKFILE ]
8: then
9:   echo "Már fut egy példány a programból!"
10:  echo "Várakozunk a kilépésére..."
11:  while [ -f $LOCKFILE ]
12:  do
13:    sleep 2
14:  done
15: fi
16:
17: trap 'rm $LOCKFILE; exit 2' 1 2 3 15
18:
19: echo $$ > $LOCKFILE
20: echo -n "Új szöveg: "
21: read sor
22: echo $sor >> $CONFIG_FILE
23: rm $LOCKFILE
```

A 11-14. sorokban látható várakozó ciklus egyetlen, két másodperces várakozásból áll. Vegyük észre, hogy a trap parancs most csak az esetleges várakozás után lép életbe. Ez azért fontos, mert ha várakozás közben megszakítanánk a programot, akkor a másik futó példány által létrehozott zároló fájt törölne le.

Önhívó parancsvéghajtás teljes könyvtárszerkezetben

Munkánk során bizonyára gyakran találjuk majd szembe magunkat olyan programozási feladatokkal, amikor a feladat valamilyen ismétlődő művelet vagy műveletsor véghajtása egy könyvtárszerkezet teljes tartalmára, vagy bizonyos típusú fájlokra.

Tegyük fel például, hogy egy nagyobb programcsomagot fejlesztünk, amely esetleg több száz forrásfájlból áll, és ezek rendeltetésük szerint csoportosítva különböző alkönyvtárakban találhatók (például ./lib, ./include stb.). Pontosan tudjuk egy bizonyos változó vagy függvény nevét, de elfelejtettük, hogy meghatározása melyik fájlban szerepel. Írunk egy olyan önhívó (rekurzív) grep parancsnak megfelelő héjprogramot, amelynek parancssori paraméterként megadhatjuk a keresendő mintát, és az végignézi a könyvtárszerkezet teljes tartalmát.

A feladat megoldása nyilván a két alapműveletből fog állni:

- Elő kell állítanunk a könyvtárakban található valamennyi fájl listáját.
- Végre kell hajtanunk a lista valamennyi elemére a grep parancsot.

A kereséshez a valamennyi UNIX rendszeren megtalálható find programot használhatjuk, ha pedig – mint jelen esetben is – csupán egy műveletet kell véghajtanunk, azt egyszerűen megadhatjuk a -exec kapcsoló után. A legegyszerűbb megoldás tehát így fest:

6.14. lista

```
1: #!/bin/sh
2: # Önhívó grep parancs
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -ne 1 ]
7: then
8:     echo "Használat: $PROGRAMNEV <keresendő minta>" .
9:     exit 1
10: fi
11:
12: find . -type f -exec grep "$1" {} \;
```

A program lelke tulajdonképpen a 12. sorban látható. A „.” azt jelenti, hogy a kérést a pillanatnyi könyvtárban kezdjük, a -type f megadásával azt jeleztük, hogy csak a fájlokra vagyunk kíváncsiak (könyvtárrakra nem), a -exec után pedig a minden egyes fájlra végrehajtandó grep parancs szerepel. A „{}” szimbólumot a find a találati listával helyettesíti, a pontosvessző pedig a végrehajtandó parancssor végett jelöli. (A pontosvesszőt azért kellett levédeni, mert a héj is értelmezi.)

Példánknál maradva, ha csak a változó értékére vagy a függvény paramétere zérésére vagyunk kíváncsiak, a fenti program tökéletesen megfelel a célnak, hiszen – a grep parancshoz hasonlóan – találat esetén a megfelelő szövegsort jeleníti meg.

Előfordulhat azonban, hogy nem (vagy nem csak) a tényleges sorokra vagyunk kíváncsiak, inkább arra, hogy melyik fájlban szerepel a keresett adat. Próbáljuk meg tehát úgy átalakítani a önhívó keresőprogramot, hogy találat esetén a fájl nevét jelenítse meg.

Ezt már biztosan nem tudjuk egyetlen művelettel megoldani, hiszen megint végre kell hajtanunk a grep parancsot, majd meg kell vizsgálnunk a kimenetét. (Egész pontosan azt, hogy volt-e kimenete.) Mivel a find parancs -exec kapcsolója után csak egy művelet adható meg, most más megoldást kell keresnünk. A legegyszerűbb az lesz, ha a megfelelő fájlok nevéről egy átmeneti fájlban készítünk egy listát, majd egy ciklusba ágyazott grep segítségével ennek a sorain megyünk végig. Íme egy lehetséges megoldás:

6.15. lista

```

1: #!/bin/sh
2: # Adott szó önhívó keresése adott típusú fájlokban
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -ne 2 ]
7: then
8:   echo "A program használata: $PROGRAMNEV <kiterjesztés>
          <keresendő minta>"
9:   exit 1
10: fi
11:
12: trap 'rm /tmp/$$lista' 1 2 3 15
13: find . -name '*.'$1 > /tmp/$$lista
14:
15: while read sor
16: do
17:   if grep "$2" $sor > /dev/null

```

```
18:     then
19:         echo $sor
20:     fi
21: done < /tmp/$$lista
22:
23: rm /tmp/$$lista
```

Ennek a programnak nemcsak a fájlokban keresendő mintát, hanem azok kiterjesztését is megadhatjuk az első parancssori paraméterben. A szokásos ellenőrzés után a 13. sorban a megfelelő nevű fájlok listáját egy egyedi nevű (\$\$) átmeneti fájlban helyezzük el. A 15. sorban induló while ciklus a bemenetét ebből a fájlból kapja (21. sor). minden egyes fájlra lefuttatja a grep parancsot úgy, hogy a második parancssori paraméterben megadott mintát keresi benne. A 17. sorban látható feltétel szerkezet a grep visszatérési értéke alapján működik. Mivel a találatot okozó sorokat nem akarjuk a képernyón megjeleníteni, a grep kimenetét a /dev/nullba irányítottuk.

Az xargs parancs

Az xargs parancs segítségével úgy hívhatunk meg egy másik UNIX programot, hogy annak a parancssori paramétereit (kapcsolók, feldolgozandó fájlok nevei) az xargs szabványos bemenetére küldjük. Az xargs ebből és a saját paramétereként megadott parancs nevéből összeállít egy parancsot, és végrehajtja azt.

Ha például az ls parancsot akarjuk végrehajtani úgy, hogy az összes „txt” kiterjesztésű fájlról adjon részletes listát, a következőképpen járhatunk el:

```
echo -1 *.txt | xargs ls
```

A *.txt helyére a héj behelyettesíti az összes „txt” kiterjesztésű fájl nevét, majd az echo ezt a listát és a -1 kapcsolót átadja az xargs-nak a szabványos bemenetén keresztül. Az xargs veszi a paramétereként megadott "ls" karakterláncot, beilleszti a szabványos bemenetéről érkezett lista elé, és ami így keletkezett („ls -1 < fájlok nevei >”), azt végrehajtja, mint parancsot.

Az eredmény természetesen ugyanaz lesz, mintha kiadtuk volna az

```
ls -1 *.txt
```

parancsot, hiszen itt is majdnem pontosan ugyanaz történik, mint amit az xargs csinál: a héj összegyűjt a neveket, a listát átadja az ls programnak, az pedig „fel-dolgozza” azt.

Mármost a Kedves Olvasó ezek után nyilván úgy gondolja, hogy az xargs parancs használatának az égvilágban semmi értelme, hiszen minden, amit az xargs-szal megvalósíthatunk, egyszerűen a parancssorba is begépelhetjük. Ráadásul úgy sokkal egyszerűbben néz ki.

Ez az állítás az esetek többségében természetesen igaz is. Vannak azonban „különleges helyzetek”. Tegyük fel például, hogy egy könyvtár teljes tartalmát törölni akarjuk. Már az első lecke után tudja bármelyik kezdő felhasználó, hogy ezt az „rm *” parancssal tehetjük meg. De pontosan mi is történik itt?

A héj elolvassa a parancssort. Talál benne egy különleges karaktert („*”) amiről tudja, hogy a fájlnevek listájával kell helyettesítenie. Egy belső átmeneti tárban összegyűjt a neveket, és átadja az rm parancsnak.

Ez a módszer kiválóan működik addig, amíg a nevek listája elfér az említett tárban. Ha viszont nem, akkor az „Argument list too long” (vagy valami hasonló tartalmú) üzenetet kapjuk, és a fájlok maradnak, ahol voltak.

Pontosan az efféle patthelyzetek elkerülésére jó az xargs, ez ugyanis képes arra, hogy tetszőleges méretűre darabolja a paraméterlistát. Az alapértelmezett „darabolási hossz” eleve úgy van beállítva, hogy idomuljon a rendszer képességeihez. Ha tehát az xargs segítségével valósítunk meg egy műveletet, akkor az biztosan védve lesz a nevekkel kapcsolatos tártúlcordulások ellen.

A -n kapcsoló segítségével mi magunk is meghatározhatjuk, hogy az xargs egyszerre hány paramétert adjon át az utána következő parancsnak. Az alábbi parancs például nyolc oszlopban jeleníti meg a pillanatnyi könyvtár tartalmát:

```
ls | xargs -n 8 echo
```

A tártúlcordulás lehetőségével minden olyan esetben számolunk kell, amikor nem tudhatjuk előre, hogy héjprogramunknak milyen adatmennyiséget kell majd feldolgozna. Az előző példában bemutatott önhívó (rekurzív) grep parancs éppen ilyen: nem tudhatjuk előre, hogy a find hány megfelelő fájlt talál majd a könyvtárszerkezetben. Az xargs segítségével azonban „bombabiztosá” tehetjük a program működését:

6.16. lista

```
1: #!/bin/sh
2: # Rekurzív grep parancs: a "biztonságos" megoldás
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -ne 1 ]
7: then
8:     echo "Használat: $PROGRAMNEV <keresendő minta>"
9:     exit 1
10: fi
11:
12: find . -type f -print | xargs grep "$1"
```

7

Gyakorlatok I.

Egyszerű feladatok

Ha az Olvasó türelmesen átrágta magát az előző fejezeteken, feltehetőleg késznek érzi már magát arra, hogy önálló feladatmegoldásokon keresztül kipróbálja megszerzett tudását. Ennek megfelelően a könyv hátralevő részében feladatokat mutatnak be lehetséges megoldásaikkal együtt.

Minden szakasz három részből áll majd:

- A feladat pontos megfogalmazásából.
- A megvalósításhoz szükséges ötletek és tények felsorolásából.
- Egy lehetséges megoldásból és a hozzá tartozó magyarázatból.

Bár ez a fejezet is feldolgozható abban a stílusban, mint az előzők, én mégis azt javaslok az Olvasónak, hogy először csak a feladatot és az ötleteket nézze meg, majd ezek alapján próbálja önállóan megírni a programot.

Ha sikerült, saját megoldását feltétlenül hasonlítsa össze a könyvben szereplő kód-dal. Ha nem, nézzen bele az én megoldásomba, és – ha maradt még türelme – próbálkozzon meg újra a sajátjával.

Jó szórakozást!

Változatok egy témára: az angol ábécé betűi

Alsó tagozat...

Feladat

Írassuk ki az angol ábécé kisbetűit! (A magától értetődő megoldástól, vagyis az echo "abc . . ." parancstól most tekintsünk el.)

Első megoldás

„Ha csak kalapácsod van, minden úgy néz ki, mint egy szög.” (Murphy)

Ötletek

Az angol ábécé 26 kisbetűt tartalmaz, amelyek az ASCII kódtáblában egymás után következnek. Az „a” betű kódja a 97, a „z” betűé a 120. Egy kivételével valamennyi megoldásban ezt a tényt fogjuk kihasználni.

A UNIX-nak tulajdonképpen van is olyan eszköze, amely ASCII kód alapján képes karaktereket kiírni, ez pedig nem más, mint a jól ismert echo parancs. Egyetlen bökkenő van: az ASCII kódot oktálisan, vagyis nyolcas számrendszerben kell megadnunk. Az utasításforma a következő:

```
echo "\ooo"
```

ahol "\ooo" a kiírandó karakter ASCII kódja nyolcas számrendszerben.

Nincs más hátra, írnunk kell egy olyan függvényt, amely a decimális számokat nyolcas számrendszerbeli alakjukká váltja át. Legyen ennek a neve (a UNIX-ban szokásos rövidítések mintájára) d2o („d to o”).

Az átszámítás algoritmus a következő:

- Vesszük a decimális értéket és elosztjuk a nyolcas számrendszer azon legmagasabb helyiértékével, amely még egynél többször megvan benne.
- Az eredmény egész része lesz az adott helyiértékhez tartozó számjegy az új számrendszerben, az osztás maradéka pedig a további számolás alapja.
- Az algoritmust egyre csökkenő helyiértékekkel addig folytatjuk, amíg az egyes helyiértékhez nem érünk. Ekkor a maradék az egyesek helyén álló számjegy.

Nem elegáns ugyan, de a megoldás során kihasználhatjuk, hogy az ASCII kódok nyolcas számrendszerben legfeljebb 3 jegyűek, vagyis a legmagasabb helyiérték, amivel foglalkoznunk kell, a 64.

A megoldás

7.1. lista

```

1: #!/bin/sh
2: # Az angol ábécé betűinek kiíratása decimális
   és oktális ASCII kódjukkal együtt
3:
4: # Decimális érték oktálissá alakítása maradékos osztással
5: d2o()
6: {
7:     oct=""
8:     eredmeny=`expr $dec / 64`
9:     kivonando=`expr $eredmeny \* 64`
10:    dec=`expr $dec - $kivonando`
11:    oct=`echo -n $eredmeny | awk -v e=$oct '{e=e $0; print e}'`'
12:    eredmeny=`expr $dec / 8`
13:    kivonando=`expr $eredmeny \* 8`
14:    dec=`expr $dec - $kivonando`
15:    oct=`echo -n $eredmeny | awk -v e=$oct -v e1=$dec
          '{e=e $0 e1; print e}'`'
16: }
17:
18: # Az "a" betű ASCII kódja 97.
19: # De ha valaki nem tudja fejből, megkérdezheti a rendszertől is...
20:
21: akod=`echo -n a | od -d | awk '{if(NR==1) print $2}'`'
22:
23: i=0
24: while [ $i -le 25 ]
25: do
26:     dec=`expr $akod + $i`'
27:     olddec=$dec
28:     d2o
29:     eval "echo -e \$olddec \\\" \\\" \$oct \\\" \\\" \\\"\\\$oct\\\""
30:     i=`expr $i + 1`'
31: done

```

Az előbb vázolt maradékos osztást két expr segítségével könnyedén elvégezhetjük, ugyanis a / jel az expr esetében eleve egészosztást jelent, a % (modulo) művelet pedig az osztás maradékát adja vissza (7-15. sorok).

A dec nevű változóban tárolt decimális ASCII értéket a 24. sorban kezdődő ciklusban folyamatosan változtatjuk és átszámoltatjuk a d2o függvény segítségével. Ez az oct nevű változóban adja vissza a nyolcas számrendszerbeli megfelelőt.

A decimális és oktális értéket, valamint magát a karaktert a 29. sorban íratjuk ki.

Megjegyzés

Egyes rendszereken az echo csak akkor hajlandó elhinni, hogy tényleg oktális számot adtunk meg neki, ha az nullával kezdődik. A 29. sorban ilyenkor a literális (és ezért megkettőzött) \ után gondoskodnunk kell erről.

Vannak olyan rendszerek is (például a Linux), ahol az echo csak a -e kapcsoló hatására értelmezi az oktális értéket.

Említést érdemel még az a mód, ahogy a rendszertől „kicsaltuk” az „a” betű ASCII kódját a 21. sorban. (Ezt tulajdonképpen egy egyszerű értékadással is elintézhettük volna, de abban nincs semmi érdekes.) A megoldás kulcsa az od nevű, valamennyi UNIX rendszeren megtalálható segédprogram. Az od a bemenetére érkező adatot (többek között) képes megjeleníteni tetszőleges számrendszerben. A -d kapcsoló jelenti számára a decimális megjelenítést. Ha azonban kipróbáljuk a parancssorban az

```
echo a | od -d
```

kombinációt, két hibát is látunk. Az egyik, hogy bár egyetlen betűt küldünk ki, az od mégis két ASCII kódot adott meg. Ennek oka, hogy az „a” betűt egy újsor (\n) karakter is követte, amit az od szintén lefordított. Az echo után tehát feltétlenül meg kell adnunk a soremelést letiltó -n kapcsolót.

A másik hibát az okozza, hogy az od kimenetében az egyes bajtoknak az adatfolyamban elfoglalt pozíciója is szerepel a kimenet első oszlopában, erre pedig most nincs szükségünk. Ennek a szolgáltatásnak a letiltása természetesen lehetséges, de az egyes UNIX-okon erősen eltérő módon. Ezért egy általánosabb megoldást választottunk: az awk segítségével kiszűrjük a kimenet első sorát (NR==1) és ebből is csak a második mezőt, vagyis a kívánt ASCII kódot küldjük a kimenetre.

Második megoldás

„Ha nem megy, feszegesd!” (Murphy)

Ötletek

Az előző megoldásban kiírtattuk a decimális ASCII kódok mellett azok nyolcas számrendszerbeli megfelelőit is. Ha kicsit elmélázunk a lista felett, észrevehetjük, hogy az oktális számok formálisan ugyanúgy következnek egymás után, mint a decimálisak. Az ASCII kódok listája az „141” értékkel kezdődik (ami most nem száznegyvenegy, hanem egy-négy-egy), majd a 142-vel folytatódik, és így tovább.

Csak akkor van törés a sorban, ha a decimális logika szerint 8-ra vagy 9-re végződő szám következne. Ilyenkor a nyolcas helyiérték „kattan” eggyel feljebb.

Ha már észrevettük ezt a szabályosságot, használjuk ki!

A megoldás

7.2. lista

```
1: #!/bin/sh
2: # Az angol ábécé betűi
3: # 2. megoldás
4:
5: # Az "a" oktális ASCII kódja "141", a "z" betűé "172"
6:
7: i=141
8: while [ $i -lt 173 ]
9: do
10:   if echo $i | grep -v [89] > /dev/null
11:   then
12:     echo -e "\\\$i"
13:   fi
14:   i=`expr $i + 1`
15: done
```

A megoldás az elmondottak alapján meglehetősen kézenfekvő. „Csinálunk úgy”, mintha decimális számokra lenne szükségünk, és indítsunk egy ciklust 141-től. Az echo-nak természetesen sejtelme sem lesz róla, hogy mi nem oktálisan számolunk. Egyedül arra kell ügyelnünk, hogy kihagyjuk az összes olyan számot, amelyekben a nyolcas számrendszerben „tiltott” 8 és 9 előfordul.

Épp ezt teszi a 12. sorban a grep. A -v kapcsoló tagadja (negálja) a szabályos kifejezésben megadott egyezést, vagyis akkor ad igaz értéket, ha nincs 8 vagy 9 a beállításban. A megvalósítás a továbbiakban azonos az előzővel, eltekintve attól, hogy a ciklus nem 26-szor fut le, hanem 141-től egészen 173-ig megy el. (Vannak „hamis” értékek is a végrehajtás során, amiket kihagyunk.)

Valószínűleg az Olvasó is érzi, hogy a fenti két program valahogy nagyon körülmenyezen oldja meg a kitűzött feladatot. Elárulhatom, hogy ez a megérzés helyes. Ugyanakkor nem lehet kizártani, hogy valamilyen valós feladat megoldásához egyszer majd épp ezekből meríthetünk ötletet, pontosan a bonyolultságuk és körmönfontságuk miatt.

Harmadik megoldás

Az igazi...

Ötletek

Valljuk be őszintén, a lelkünk mélyén végig éreztük, hogy kell lennie erre a feladatra egy remek, egysoros megoldásnak. És valóban van is.

Az awk printf függvénye szintén képes karaktereket kiírni ASCII kód alapján, ráadásul megelégszik a decimális értékkal is. Ezen kívül természetesen számos egyéb műveletet is el lehet vele végezni.

Az elvégzendő műveletet úgynevezett formátumleíró karakterláncok segítségével adhatjuk meg. Aki ismeri a C nyelvet, annak ez a módszer valószínűleg ismerős lesz, hiszen maguk a vezérlőkódok is azonosak. Ezek mindenkorral a % jelrel kezdődnek, és egy behelyettesítendő érték típusára utalnak. A %c jelentése például: „írd ki azt a karaktert, amelynek ASCII kódja a formátum leírása után szerepel”. (A további vezérlőkódokról oldalakon keresztül szól az awk leírása.).

A megoldás

7.3. lista

```
1: #!/bin/sh
2: # Az angol ábécé megjelenítése
3: # A "helyes" megoldás
4:
5: awk 'BEGIN {for(i=97;i<123;i++) printf "%c\n",i}'
```

Ez tényleg egyetlen sor, és tényleg ugyanazt valósítja meg, mint az előző kettő megoldás. A legfőbb tanulság tehát az, hogy mielőtt programozni kezdünk, nem árt alaposan elgondolkodni a lehetséges eszközök körén. (Természetesen a UNIX eszközök közéletének átfogó ismerete sem nélkülözhető.)

A formátum megadásánál \n a kiírás utáni sora melésről gondoskodik. Mivel az awk-ban ciklust is szervezhetünk, arra sem volt szükség, hogy egy héjváltozóban adjuk át a kívánt ASCII kódot. Arra azonban ügyelni kell, hogy mindenzt egy BEGIN blokkba helyezzük, hiszen az awk ebben az esetben nem valamilyen bemenetet dolgoz fel. Kivételesen nem is kell semmit a bemenetére irányítanunk ahhoz, hogy működjön.

Negyedik megoldás

Töréteszt...

Ötletek

Ha veszünk egy hosszabb szöveget, abban elég nagy valószínűsséggel előfordul az angol ábécé összes betűje. Ha sikerül ezeket valahogy „egy helyre terelni” (például: az összes „a” betű következzen egymás után), az így kapott – meglehetősen unalmas – betűtengerből már könnyedén kiválogathatjuk az ábécé betűit.

A megoldás

7.4. lista

```

1: #!/bin/sh
2: # Az angol ábécé
3: # A "rendhagyó" megoldás
4:
5: man grep | awk 'BEGIN {FS=""} {for(i=1;i<=NF;i++) print $i}' | \
6: sort | uniq | tr -d "\n" | sed 's/a/Xa/ ; s/z/zX/' | tr -s X "\n" | \
7: grep "^\a"

```

Rögtön szeretném hangsúlyozni, hogy ez az egész tényleg nem komoly. Valóban csak arra szolgál, hogy gyakoroljuk rajta a UNIX eszközeinek használatát.

Ez a program is egyetlen sor, de olyan hosszúra sikerült, hogy sorvégi „\” karakterek segítségével (összefűzés) több darabra kellett törni.

Kezdjük el lépésenként visszafejteni, mi is történik itt. Először a „man grep” parancs kimenetét egy awk program bemenetére küldjük. A BEGIN blokkban a mezőelválasztó karakter értékét töröljük (`FS=" "`), ami azt eredményezi, hogy az awk minden karaktert külön mezőnek tekint majd. Ezeken egy for ciklussal megyünk végig, és egyenként kiíratjuk őket. Így a bemenetből egyetlen karakter szélességű függőleges oszlop keletkezik.

Ezt a kimenetet a sort rendezi, vagyis egymás mellé sőpri az azonos karaktereket. Elöl jönnek az „a” betűk mindahányan, aztán a teljes ábécé szép katonás rendben, végezetül pedig az írásjelek.

A uniq tovább fokozza a „rendet”: minden ismétlődő sorból csak egyet küld a kiemenetére. (Fontos megjegyezni, hogy a uniq kizárálag rendezett bemenettel tud dolgozni, tehát a feldolgozási sorból a sort semmiképpen nem hagyható ki.)

A szövegben előforduló karakterek közül immár mindegyikból csak egy halad tovább, azok is ábécésorrendben. A tr parancs a -d kapcsoló hatására törli a bemenetből a megadott karaktert, jelen esetben az összes újsort. Az egyes betűk mostantól egy sorként haladnak tovább.

A következő lépésekben a sed segítségével beillesztünk egy tetszőleges jelzést (esetünkben X-et) az „a” elő és a „z” után, majd a tr -s (substitute, helyettesít) kapcsolója segítségével lecseréljük azokat egy-egy újsor karakterre. Ezek után nincs más hátra, mint kiválasztani a keletkező három sorból azt, amelyik a kis „a” betűvel kezdődik (ez a középső sor lesz).

És íme, az angol ábécé 26 kisbetűje! Bár mindez „jó mulatság, férfimunka volt”, azért programunkat nagyobb bemenettel (például Háború és béke) ne nagyon próbáljuk ki...

A UNIX segédprogramok „magyartudásának” ellenőrzése

Beszélni magyar?!...

Feladat

Korábban volt róla szó, hogy a GNU programok közül a sed-nek egyelőre gondjai vannak a magyar ékezetes betűk kezelésével. Ugyanez más rendszereken is előfordulhat a szövegfeldolgozást végező programokkal, tehát célszerű ellenőrizni a fontosabb segédeszközök képességeit. Írunk héjprogramot, amely képes eldöntheti, hogy a grep, a sed és az awk az adott rendszeren felismeri-e az ékezetes betűket mint az [:alpha:] karakterosztály tagjait!

Ötletek

Elegendő kizárálag az ékezetes betűkkel foglalkoznunk. Ezekből (összesen 9 van) hozzunk létre egy listát, majd számoltassuk meg a három említett segédprogrammal, hogy szerintük hány megfelelő elem található benne.

A megoldás

7.5. lista

```

1: #!/bin/sh
2: # Három UNIX segédprogram magyartudásának ellenőrzése
3:
4: magyar_abc="á é í ó ö ô ú ü ű"
5:
6: [ `echo $magyar_abc | tr " " "\n" | grep "[[:alpha:]]" | wc -l` -eq 9 ] \

```

```
7:  && echo "A grep tud magyarul." || echo "A grep NEM tud magyarul."
8:
9:  [ `echo $magyar_abc | tr " " "\n" | sed "s/[[[:alpha:]]]/{X}/" | \
10: grep "X" | wc -l` -eq 9 ] \
11: && echo "A sed tud magyarul." || echo "A sed NEM tud magyarul."
12:
13: [ `echo $magyar_abc | tr " " "\n" | awk ' \
14:   [[[:alpha:]]]/{print}' | wc -l` ] \
14: && echo "Az awk tud magyarul." || echo "Az awk NEM tud magyarul."
```

A „magyar_abc” nevű változóban szóközökkel elválasztva megadjuk a magyar ékezetes magánhangzók listáját. Ezt használat előtt a `tr`-rel minden esetben úgy alakítjuk át, hogy a betűket újsor karakterek válasszák el egymástól.

Mindhárom vizsgált programnál használnunk kell valamilyen formában az `[:alpha:]` karakterosztályt. A `grep`-nél kiválogatjuk azokat a sorokat, amelyekben van ilyen karakter, a `sed`-nél lecseréljük azokat egy "X" karakterre, az `awk`-nál pedig a főprogram címzésére használjuk.

Az `if` szerkezet helyett mindhárom esetben az egyszerűsített kétirányú elágazást használtam az eredmény megjelenítésére.

Tac

Utolsó pár előre fuss...

Feladat

Írunk olyan héjprogramot, amely a `cat`-hez hasonlóan megjeleníti a szövegfájlok tartalmát, de a sorokon visszafelé haladva. (Az utolsó sort jeleníti meg először, az elsőt utoljára.) Egyéb képességeit tekintve legyen teljesen azonos a `cat` paranccsal, vagyis tudjon feldolgozni akár több fájt is, és képes legyen kezelní a szabványos bemenetről érkező szöveget. (Ez a program egyes UNIX rendszereknek `tac` néven eleve része. A Linux is ezek közé tartozik.)

Ötletek

A feladat logikai szerkezetét tekintve nagyon hasonló az előző fejezetben bemutatott szövegkeretezőhöz: azonos módon kell kezelnie a szabványos bemenetét és a parancssori paraméterként megadott fájlokat. A sorok fordított sorrendben való megjelenítéséhez először nyilván ismernünk kell a fájl hosszát, amit a wc segítségével határozhatunk meg. Ezután egy visszafelé haladó ciklusban az awk vagy a sed segítségével jeleníthetjük meg a fájl tartalmát. Ha a szöveg a szabványos bemenetről érkezik, előbb össze kell gyűjtenünk egy átmeneti fájlba.

A megoldás

7.6. lista

```
1: #!/bin/sh
2: # A tac parancs megvalósítása héjprogramként
3:
4: olvas()
5: # A $1-ben megadott fájl sorait olvassa visszafelé
6: {
7:   hossz=`cat $1 | wc -l`
8:   while [ $hossz -ge 1 ]
9:   do
10:     cat $1 | sed -n "$hossz p"
11:     hossz=`expr $hossz - 1`
12:   done
13: }
14:
15: if [ $# -ne 0 ]    # Fájlokóból érkező bemenet
16: then
17:   for nev in $*
18:   do
19:     if [ -f $nev ]
20:     then
21:       olvas $nev
22:     else
23:       echo "A $nev nevű fájl nem létezik!" 1>&2
24:     fi
25:   done
26: else                  # A szabványos bemenet olvasása
27:   touch /tmp/tactmp$$
28:   trap 'rm /tmp/tactmp$$' EXIT
29:   while read sor
30:   do
31:     echo $sor >> /tmp/tactmp$$
```

```
32:      done
33:      olvas /tmp/tactmp$$
34:      fi
35:      exit 0
```

A névvel megadott fájlok feldolgozását a 15-25. sorok, a szabványos bemenetét a 26-34. sorok végzik. Magát a fordított kiíratást egy függvény formájában valósítottam meg, tekintve, hogy ez a rész mindenkorban közös. Magát a kiírást a sed végzi (10. sor), amelynél a -n kapcsolóval le kell tiltani a sorok megjelenítését. Figyeljük meg, hogy a korábbi fejezetektől eltérően itt nem egyszeres, hanem kétszeres idézőjelek között van megadva a feldolgozási program. Erre azért van szükség, mert a kiírandó sor sorszáma most a hossz nevű változóból kerül bele, az egyszeres idézőjel pedig megakadályozná a héjat az érték szerinti behelyettesítésben.

A szabványos bemenet olvasásakor létrehozott átmeneti fájl neve természetesen tartalmazza a program folyamatazonosítóját, biztos törléséről pedig a 28. sorban megadott trap parancs gondoskodik. Mivel a törlés műveletét a kilépés (EXIT) helyez rendeltük hozzá, a 27. sorban szükség van egy touch parancsra, amely csak létrehozza az átmeneti fájlhoz tartozó bejegyzést, de nem tesz bele semmit.

Ez első látásra fölöslegesnek tűnhet, de nem az. Képzeljük el ugyanis, hogy mi történik, ha a felhasználó csak begépel a parancs nevét, majd ENTER-t üt. Se neveket nem adunk meg, sem a szabványos bemenetre nem küldünk semmit, így a program (a cat-hez teljesen hasonlóan) a billentyűzetről várja a bemenetet. A touch parancs hatására az átmeneti fájl már az első sor begépelése előtt létrejön, így ha a felhasználó gépelés helyett leüti a CTRL+C billentyűket, akkor is van mit törölnie a trap utáni rm parancsnak. A touch nélkül viszont ilyenkor hibaüzenetet kapnánk.

Érdemes még megfigyelni a 23. sorban látható kiíratást. Ez egy hibaüzenet, így nem a szabványos kimenetre, hanem a hibacsatornára kell kerülnie. Éppen erről gondoskodik az echo utáni átirányítás.

Rev

szálaB sik nepézs le jdulA

Feladat

Írunk héjprogramot, amely megfordítva írja ki a megadott szöveg (szabványos be-menet vagy fájlnév) sorait. (Ez a program rev néven szintén része számos UNIX rendszernek.)

Ötletek

A feladat rendkívül hasonló az előzőhez. Olyannyira, hogy a főprogram tulajdon-képpen változtatás nélkül felhasználható, csak a feldolgozó függvényt kell átírni. Ennél ismét kihasználhatjuk, hogy az awk minden betűt külön mezőnek tekint, ha az FS értéke üres karakterlánc.

A megoldás

7.7 lista

```

1: #!/bin/sh
2: # Szöveg sorainak megfordítása (a rev parancs megvalósítása)
3:
4: vissza()
5: # A $1-ben megadott fájl sorait visszafelé írja ki
6: {
7:   cat $1 | awk 'BEGIN {FS=""}
8:                 {for(i=NF;i>0;i--) printf "%s", $i; print ""}'
9: }
10:
11: if [ $# -ne 0 ]
12: then
13:   for nev in $*
14:   do
15:     if [ -f $nev ]
16:     then
17:       vissza $nev
18:     else
19:       echo "A $nev nevű fájl nem létezik!" 1>&2
20:     fi
21:   done
22: else

```

```

23:     touch /tmp/rev$$
24:     trap 'rm /tmp/rev$$' EXIT
25:     while read sor
26:     do
27:         echo $sor >> /tmp/rev$$
28:     done
29:     viissa /tmp/rev$$
30: fi
31: exit 0

```

A megvalósítás az előző példa után különösebb magyarázatot feltehetőleg nem igényel. Az egyetlen említésre érdemes dolog talán a 8. sorban látható `printf` parancs, amire azért van szükség, hogy a kiírt karakterek egy sorban jelenjenek meg.

TIPP

Az előző és az ebben a szakaszban megírt program kísértetiesen hasonlít egymásra. A főprogramok az átmeneti fájlok és az általuk meghívott függvény nevétől eltekintve teljesen azonosak. Ezek után joggal kérdezheti az Olvasó: tényleg muszáj ezt kétszer megírni?

Természetesen nem. Kihasználhatjuk ugyanis, hogy a UNIX-ban egy fájlnak – illetve esetünkben egy programnak – több neve is lehet, közvetlen vagy közvetett hivatkozások használatával. Mivel a `$0` minden azt a nevet tartalmazza, amellyel a programot elindítottuk, felhasználhatjuk a név szerinti döntés megfogalmazására. Az előbbi két program egyesítése tehát így nézhet ki:

7.8. lista

```

1: #!/bin/sh
2: # A rev és a tac parancs megvalósítása
   egyetlen héjprogramként
3:
4: olvas()
5: # A $1-ben megadott fájl sorait olvassa visszafelé
6: {
7:     hossz=`cat $1 | wc -l`
8:     while [ $hossz -ge 1 ]
9:     do
10:        cat $1 | sed -n "$hossz p"
11:        hossz=`expr $hossz - 1`
12:    done
13: }
14:
15:
16: viissa()

```

```
17: # A $1-ben megadott fájl sorait visszafelé írja ki
18: {
19:     cat $1 | awk 'BEGIN {FS=""}
20:                     {for(i=NF;i>0;i--) printf "%s", $i; print ""}'
21: }
22:
23:
24: NEV1="stac"          # Tac "üzemmód" neve
25: NEV2="srev"           # Rev "üzemmód" neve
26:
27: NEV=`basename $0`
28:
29: if [ $# -ne 0 ]      # Fájlokóból érkező bemenet
30: then
31:     for nev in $*
32:     do
33:         if [ -f $nev ]
34:         then
35:             if [ $NEV = $NEV1 ]  # tac üzemmód
36:             then
37:                 olvas $nev
38:             fi
39:             if [ $NEV = $NEV2 ]  # rev üzemmód
40:             then
41:                 vissza $nev
42:             fi
43:         else
44:             echo "A $nev nevű fájl nem létezik!" 1>&2
45:         fi
46:     done
47: else                  # A szabványos bemenet olvasása
48:     touch /tmp/revtactmp$$
49:     trap 'rm /tmp/revtactmp$$' EXIT
50:     while read sor
51:     do
52:         echo $sor >> /tmp/revtactmp$$
53:     done
54:     if [ $NEV = $NEV1 ]      # tac üzemmód
55:     then
56:         olvas /tmp/tactmp$$
57:     fi
58:     if [ $NEV = $NEV2 ]      # rev üzemmód
59:     then
60:         vissza /tmp/revtactmp$$
61:     fi
62: fi
63: exit 0
```

Itt a NEV1 tartalmazza az egyik, a NEV2 a másik „üzemmódhoz” tartozó nevet, a NEV változóban pedig a \$0 tartalmát helyeztük el (24-27. sorok). A basename használata itt sem felesleges, ugyanis közvetett hivatkozások használata esetén a \$0 a név előtt tartalmazni fogja a „.. /” karakterpárost is (a pillanatnyi könyvtárra való hivatkozás). A név szerinti döntési szerkezet a főprogram 35-42., illetve 54-61. soraiiban található.

Titkosítás

Enigma...

Feladat

Írunk titkosító programot, amely az ábécé „elforgatásával” teszi olvashatatlaná a megadott szöveget. Első közelítésben elegendő, ha csak a szabványos bemenetet tudja feldolgozni. Azt viszont ne felejtsük el, hogy képesnek kell lennie a visszafejtésre, vagyis az ellenkező irányú forgatásra is...

Ötletek

Nyilván rendelkeznünk kell valamilyen formában a kis- és nagybetűk listájával. Ezeket az első feladatra adott harmadik megoldással állíthatjuk elő a legegyszerűbben (eltekintve természetesen a közönséges begépeléstől). A betűk cseréjének véghajtásához a tr parancsot használhatjuk, úgy, hogy első karakterlistájában a normál ábécét, míg a másodikban az „elforgatottat” adjuk meg.

A megoldás

7.9. lista

```

1: #!/bin/sh
2: # Titkosítás az angol ábécé betűinek elforgatásával
3:
4: if [ $# -eq 0 ]
5: then
6:   echo "Hasznalat : \"$0" <csusztatas erteke>""
7:   exit 1

```

```

8:   fi
9:
10:  # a,z,A és Z ASCII kódjának meghatározása
11:  akod=`echo -n a | od -d | awk '{if(NR==1) print $2}'` 
12:  zkod=`echo -n z | od -d | awk '{if(NR==1) print $2}'` 
13:  Akod=`echo -n A | od -d | awk '{if(NR==1) print $2}'` 
14:  Zkod=`echo -n Z | od -d | awk '{if(NR==1) print $2}'` 
15:
16:  if [ $1 -lt 0 ]
17:  then
18:    csuszas=`expr $1 \* -1` 
19:  else
20:    csuszas=$1
21:  fi
22:
23:  # Vágási határ
24:  hatar=`expr $akod + $csuszas` 
25:  Hatar=`expr $Akod + $csuszas` 
26:  # Az ábécé két darabjának előállítása kis- és nagybetűkre
27:  abc1=`awk -v a=$akod -v z=$zkod -v h=$hatar \
28: 'BEGIN {for(i=a;i<=h-1;i++) printf "%c",i}'` 
29:  abc2=`awk -v a=$akod -v z=$zkod -v h=$hatar \
30: 'BEGIN {for(i=h;i<=z;i++) printf "%c",i}'` 
31:  Abc1=`awk -v a=$Akod -v z=$Zkod -v h=$Hatar \
32: 'BEGIN {for(i=a;i<=h-1;i++) printf "%c",i}'` 
33:  Abc2=`awk -v a=$Akod -v z=$Zkod -v h=$Hatar \
34: 'BEGIN {for(i=h;i<=z;i++) printf "%c",i}'` 
35:
36: while read sor
37: do
38:   if [ $1 -gt 0 ]
39:   then
40:     # Pozitív irányú csúsztatás
41:     echo $sor | tr [$abc1$abc2$Abc1$Abc2] [$abc2$abc1$Abc2$Abc1]
42:   else
43:     # Negatív irányú csúsztatás
44:     echo $sor | tr [$abc2$abc1$Abc2$Abc1] [$abc1$abc2$Abc1$Abc2]
45:   fi
46: done

```

A 11-14. sorokban az od segítségével előállítjuk a két ábécé első és utolsó betűjének kódját. A „csuszas” nevű változóba a parancssori paraméterként megadott eltolás abszolútértéke kerül, vagyis ha \$1 negatív szám, megszorozzuk -1-gyel (18. sor). A „hatar” nevű változó fogja azt az ASCII kódot tartalmazni, amelynél az ábécé vágását el kell végezni.

Mindezek birtokában a 26-34. sorokban az awk segítségével előállítjuk a megfelelő helyen elvágott betűlistákat. Figyeljük meg, hogy különbség van a kis- és nagybetűs névvel rendelkező héj változók között!

A magunk kreálta megállapodás szerint a csúsztatás pozitív értéke azt fogja jelenteni, hogy az ábécé első néhány betűje kerül a sor végére, negatív csúsztatásnál viszont az utolsó betűket másoljuk előre. \$1 még minden tartalmazza a csúsztatás értékét, így a 38. sorban felhasználhatjuk a működés vezérlésére.

Végül a tr parancs hívásakor nincs egyéb dolgunk, mint a két szótárban abc1 és abc2, illetve Abc1 és Abc2 értékét megfelelő sorrendben egymás mellé helyezni.

Ez a megoldás egyáltalán nem kezeli a magyar ékezes magánhangzókat. A program magyar változatának elkészítését az Olvasóra bízom.

Betűk megszámlálása

Költői lelkület...

Feladat

Egyes irodalomkutatók messzemenő következtetéseket szoktak abból levonni, hogy egy költő műveiben milyen gyakorisággal fordulnak elő az ábécé egyes betűi. Írunk héjprogramot, amely egy bemeneti fájl alapján elkészít egy ilyen gyakorisági táblázatot.

Ötletek

A feldolgozandó fájl sorait az awk segítségével különálló betűkre bonthatjuk, ha FS értékéül üres karakterláncot adunk meg. Ezután az ábécé betűin egy ciklussal véghaladva megszámolhatjuk és összegezhetjük az egyes karakterek előfordulásait. Az eredmény rögtön a képernyőre kerülhet, de utólagos formázás végett tárolhatjuk egy átmeneti fájlban is.

A megoldás

7.10. lista

```
1: #!/bin/sh
2: # Betűk gyakoriságának meghatározása
3:
4: PROGRAMNEV='basename $0'
5: abc="a á b c d e é f g h i í j k l m n o ó ö p q r s t u ú ü v w x y z"
6: ABC="A Á B C D E É F G H I Í J K L M N O Ó Ö P Q R S T U Ú Ü V W X Y Z"
7:
8: if [ $# -ne 1 ]
9: then
10:   echo "Használat: $PROGRAMNEV <fájlnév>"
11:   exit 1
12: fi
13:
14: if [ ! -f $1 ]
15: then
16:   echo "A megadott fájl nem létezik!"
17:   exit 2
18: fi
19:
20: lista=$abc" "$ABC
21:
22: touch /tmp/betuk$$
23: trap 'rm /tmp/betuk$$' EXIT
24:
25: for betu in $lista
26: do
27:   osszeg=0
28:   while read sor
29:     do
30:       darab=`echo $sor | awk -v FS="" '{for(i=1;i<=NF;i++) print $i}'` | \
31:         grep $betu | wc -l`
32:       osszeg=`expr $osszeg + $darab`
33:     done < $1
34:     echo $betu $osszeg >> /tmp/betuk$$
35:   done
36: tail -35 /tmp/betuk$$ | paste /tmp/betuk$$ - | head -35
37: exit 0
```

A magyar ábécé kis- és nagybetűit ebben az esetben egyszerűen megadtuk két változóban (5-6. sorok). Az alapvető ellenőrzések elvégzése után ezekből állítjuk elő azt a listát, amely a betűkön végighaladó for ciklus vezérlésére szolgál majd (20. sor). Az eredmények átmeneti tárolására létrehozunk egy átmeneti fájlt, és a trap segítségével gondoskodunk a törléséről is.

A betűkön végigszaladó ciklus a 25. sorban indul. Magjában egy másik while ciklus van, amely a bemenetét a parancssori paraméterként megadott fájlból veszi (33. sor). Ebből soronként olvassuk ki a szöveget, az awk segítségével betűkre bontjuk, a kimenetből a grep-pel kiválogatjuk azokat a sorokat, amelyek az éppen keresett betűt tartalmazzák, végül a wc-vel megszámoljuk a találatokat. A végeredmény „betű, darabszám” sorrendben az átmeneti fájlba kerül, amit a 36. sorban a könnyebb áttekinthetőség végett két oszlopba tördelünk.

Ezt a paste parancssal valósítjuk meg, amely alapértelmezés szerint két vagy több fájl nevét várja parancssori paraméterként, a tartalmukat pedig egymás mellé másolva a szabványos kimenetén jeleníti meg. Ha azonban valamelyik név helyett a „-” karaktert adjuk meg, akkor a szabványos bemenetről olvassa a sorokat.

Esetünkben a paste szabványos bemenetére az átmeneti fájl utolsó 35 sora kerül (tail -35). A „-” karakter hatására ez fog bemásolódni az első 35 sor mellé. Mivel az utolsó 35 sor ismét megjelenik a kimeneten, a head -35 parancssal csonkítjuk azt.

Ha lefuttatjuk ezt a programot, azt fogjuk tapasztalni, hogy – bár tökéletesen működik – feltűnően lassú. Ennek egy (természetesen pedagógiai okokból szándékos) programszervezési hiba az oka. Figyeljük meg, mi történik pontosan a 25-35. sorokban. A for ciklus kiválaszt egy betűt, majd a magjában levő while egyenként (!) végighalad a szöveg sorain, és mindenkorál elindítja az awk-ot. És pontosan ez a hiba.

A UNIX operációs rendszer és a hozzá tartozó segédprogramok belső logikája olyan, hogy az elindulás rendkívül „költséges” folyamat. Ha viszont már fut a program, akkor gyorsan működik. Ebből pedig az következik, hogy egy héjprogramot úgy lehet „felpörgetni”, ha a lehető legkisebbre csökkentjük a programindítások számát.

Esetünkben például a belső while ciklus tulajdonképpen fölösleges, hiszen ha a fájl tartalmát egyben küldjük az awk bemenetére, akkor is pontosan ugyanaz lesz az eredmény, viszont csak egyszer kell elindítani a programot.

Programunk vége (a 25. sortól kezdve) tehát helyesen így fest:

7.11. lista

```
25: for betu in $lista
26: do
27:   darab=`cat $1 | awk -v FS="" '{for(i=1;i<=NF;i++) print $i}'` | \
28:   grep $betu | wc -l`
29:   echo $betu $darab >> /tmp/betuk$$
30: done
31: tail -35 /tmp/betuk$$ | paste /tmp/betuk$$ - | head -35
32: exit 0
```

Ezzel a módszerrel körülbelül hússzoros gyorsulás érhető el. (A `time` parancssal ellenőrizhetjük.)

Digitális számok

A technika...

Feladat

Írunk héjprogramot, amely a zsebszámológépek kijelzőjén használt „szögletes digitális” formában jeleníti meg a parancssori paraméterként megadott számot. A forma megrajzolásához használjuk a # karaktert.

Ötletek

Az említett digitális számok alakja tulajdonképpen néhány alapelemből (jellegzetes sorokból) felépíthető. Ha ezekre kitalálunk egy kódrendszer, akkor a számjegyek soronként egy-egy számjeggyel leírhatók. A „kijelző” legyen mondjuk 9 sor magas. Ezután már csak egy olyan függvényt kell írnunk, amely a kód alapján megjeleníti a megfelelő sorokat.

A megoldás

7.12. lista

```
1: #!/bin/sh
2: # "Digitális" számok megjelenítése
3:
4: PROGRAMNEV=`basename $0`
5:
6: line0=""
7: line1=" ##### "
8: line2="#      #"
9: line3="#      "
10: line4="#      "
11:
12: forma()
13: {
14:     case $1 in
15:         1) kod="4 4 4 4 0 4 4 4 4";;
16:         2) kod="1 4 4 4 1 3 3 3 1";;
17:         3) kod="1 4 4 4 1 4 4 4 1";;
18:         4) kod="2 2 2 2 1 4 4 4 4";;
19:         5) kod="1 3 3 3 1 4 4 4 1";;
20:         6) kod="1 3 3 3 1 2 2 2 1";;
21:         7) kod="1 4 4 4 0 4 4 4 4";;
22:         8) kod="1 2 2 2 1 2 2 2 1";;
23:         9) kod="1 2 2 2 1 4 4 4 1";;
24:         0) kod="1 2 2 2 0 2 2 2 1";;
25:     esac
26: }
27:
28: rajzol()
29: {
30:     alak=""
31:     for i in $kod
32:     do
33:         eval "echo \"\$line\$i\""
34:     done
35: }
36:
37: if [ $# -ne 1 ]
38: then
39:     echo "Használat: $PROGRAMNEV <kiírandó szám>"
40:     exit 1
41: fi
42:
43: if ! expr $1 + 1 > /dev/null 2>&1
44: then
```

```

45:     echo "EZ nem szám"
46:     exit 2
47: fi
48:
49: touch /tmp/digi$$
50: trap 'rm /tmp/digi*$$' EXIT
51:
52: for szamjegy in `echo $1 | awk -v FS="" '{for(i=1;i<=NF;i++)
53:                                     print $i" ")`'
53: do
54:   forma $szamjegy
55:   rajzol | paste -d " " /tmp/digi$$ - > /tmp/digi1$$
56:   mv /tmp/digi1$$ /tmp/digi$$
57: done
58: cat /tmp/digi$$
59: exit 0

```

Az említett geometriai alapelemeket a 6-10. sorok tartalmazzák. A számjegyekből a megfelelő kódot a forma nevű függvény állítja elő. A kódrendszer lényege, hogy a kijelzőn számjegy sorait felülről lefelé olvasva felírjuk, hogy a pillanatnyi sor a 6-10. sorokban megadottak közül melyiknek felel meg.

A tényleges munkát a rajzol nevű függvény végzi, amely a kód alapján kiírja a szabványos kimenetére a megfelelő sorszámu alapelemet (line0, line0 stb.). Az eval által kiértékelt sorban látható két literális kettős idézőjel, és egy szintén literális „\$” karakter. Utóbbi természetesen a parancsbehelyettesítés jele, és azért van előtte „\", mert ezt a behelyettesítést nem a kiértékelendő karakterlánc összeálításakor, hanem az eval végrehajtásakor kell majd elvégezni. Az „i” változó tartalmát viszont azonnal be kell helyettesíteni, ott tehát nem kell a \$-t levédeni. A két levédett kettős idézőjel azért kell, mert az echo csak akkor fogja az alapelemekben szereplő szóközöket helyesen megjeleníteni, ha az érték szerinti hivatkozást idézőjelek közé zárjuk.

A 37-41. sorokban azt ellenőrizzük, hogy a felhasználó pontosan egy parancssori paramétert adott-e meg. Mivel azonban programunk csak számjegyek megjelenítésére van „kiképezve”, arról is gondoskodnunk kell, hogy ez az egyetlen paraméter biztosan szám legyen.

Erre szolgál a 43. sorban látható trükkös döntési szerkezet. Itt az expr segítségével megpróbálunk egyet hozzáadni a parancssori paraméterhez. Az if csak az expr visszatérési értékét használja a döntés során, így sem a kimenetre, sem az esetlegesen keletkező hibaüzenetre nincs szükségünk. A hibacsatorna tartalmát tehát „rákeverjük” a szabványos kimenetre (2>&1), ez utóbbit pedig a /dev/null-ba irányítjuk.

A számjegyeken az 52. sorban induló for ciklus megy végig. Az ezt vezérlő listát a szokásos awk-trükk (az FS tartalmának törlése) és közvetlen parancsbehelyettesítés segítségével állítjuk elő.

Az „elkészült” számjegyeket két átmeneti fájlban tároljuk (49. és 55. sor). Az egyik (digi1\$\$) a legfrissebb, a másik az előző állapotot tartalmazza. Erre azért van szükség, mert az 55. sorban a paste bemeneteként és kimeneteként nem adhatjuk meg ugyanazt a fájlt. (Korábban volt róla szó, hogy a héj ilyenkor előbb törli a ki-meneti fájlt, csak aztán kezdi meg a feldolgozást. Ha azonban a ki- és bemenet azonos, már nem lesz mit feldolgozni.) Végül nincs más hátra, mint a cat segítségével kiíratni az átmeneti fájl tartalmát és törölni azt (50. sor).

Szavak keresése

Keresem a szót, keresem a hangot...

Feladat

Írunk olyan héjprogramot, amely meghatározza, hogy egy adott szó vagy kifejezés hányszor szerepel egy szövegfájlban. (Figyelem! A feladat csak első olvasásra tűnik egyszerűnek!)

Ötletek

Van ugyebár a grep, ami kiírja azokat a sorokat, amelyekben szerepel a keresett karakterlánc. Ezeket megszámolhatjuk a „wc -l” segítségével, de magának a grepnek is van egy -c kapcsolója, ami ezt elvégzi. A baj csak az, hogy ez a módszer egyáltalán nem törődik azzal, ha egy sorban többször is előfordul a keresett karakterlánc. A helyes megoldás tehát az, hogy a már megtalált elemeket töröljük (például a sed segítségével „semmire cseréljük” azokat), a keresést pedig addig folytatjuk, amíg van újabb találat.

A megoldás

7.13. lista

```
1: #!/bin/sh
2: # Adott szó vagy kifejezés előfordulásainak megszámlálása
3:
4: PROGRAMNEV=`basename $0`
```

```

5:
6: if [ $# -lt 2 ]
7: then
8:   echo "Használat: $PROGRAMNEV <keresett kifejezés> <fájlok>"
9:   exit 1
10: fi
11:
12: keresendo=$1
13: shift
14:
15: trap "rm /tmp/$$tmp*" EXIT
16:
17: for nev in $*
18: do
19:   if [ -f $nev ]
20:   then
21:     darabszam=`cat $nev | grep "$keresendo" | tee /tmp/$$tmp | wc -l`
22:     uj=1
23:     while [ $uj -ne 0 ]
24:     do
25:       cat /tmp/$$tmp | sed s/"$keresendo"/> > /tmp/$$tmp1
26:       mv /tmp/$$tmp1 /tmp/$$tmp
27:       uj=`cat /tmp/$$tmp | grep -c "$keresendo"`
28:       darabszam=`expr $darabszam + $uj`
29:     done
30:   else
31:     echo "A $nev nevű fájl nem létezik!" 1>&2
32:   fi
33:   echo $nev $darabszam
34: done

```

A 12. sorban elraktározzuk az első parancssori paraméterben megadott keresendő kifejezést, majd a paramétereket a shift segítségével eggyel balra léptetjük. Erre azért van szükség, mert így csak a fájlnevek maradnak meg, amelyeken egyszerűen egy for ciklussal végighaladhatunk (17.sor).

Figyeljük meg, hogy a 15. sorban szereplő trap parancsnál a végrehajtandó műveletet most nem egyszeres, hanem kettős idézőjelek határolják. Erre azért van szükség, mert a „*” karaktert csak így tudja kifejteni a héj.

A 21. sorban először azokat a sorokat számláljuk meg, amelyekben legalább egyszer előfordul a keresett kifejezés. Mivel a többi sorral a továbbiakban biztosan nem kell foglalkoznunk, a grep kimenetét a tee parancs segítségével megkettőzzük. Az egyik ág egy átmeneti fájlba kerül, a másikon pedig a wc számlálja meg a sorokat.

A továbbiakban már csak a most keletkezett átmeneti fájl tartalmát kell feldolgozunk. Először a sed segítségével „semmire cseréljük” a keresett karakterláncnak a sorokban legelöl álló példányait (most tehát nincs g módosító a sed program végen!), majd a grep-pel újabb keresést végzünk. Most már csak a találatok számára vagyunk kíváncsiak, így a -c kapcsolóval megspórolhatjuk a wc használatát. (Ami azt illeti, ez a mondat igen érdekesen hangozhat egy laikus számára...)

Figyeljük meg, hogy a sed kimenete időlegesen egy másik átmeneti fájlba kerül, majd rögtön a következő sorban ezzel felülírjuk az eredetit. Ez első látásra pazarlásnak tűnhet, de nem az. Próbáljuk ki, mi történik, ha csak egy átmeneti fájlt használunk!



A saját farkába harapó UNIX

A sed kapcsán (4. fejezet) már volt arról szó, hogy „nem illik” egy szűrőként működő programnak ugyanazt a fájlt megadni egyszerre bemenetként és kimenetként. Ha megtesszük, akkor az eredmény egy nulla bájt hosszúságú fájl lesz. A héj ugyanis először törli annak a fájlnak a tartalmát, amelybe a kimenetet irányítottuk, és csak ezután kezdi meg a bemenet feldolgozását. Ha a kettő azonos, már nem lesz mit feldolgozni.

Fenti programunk 25. sorában azonban két program van sorba kötve (cat és sed), így a takarékkosság szent nevében megint eszünkbe juthat, hogy azonossá tegyük a ki- és bemenetet. Bár a sed kapcsán már okulhattunk volna a történtekből, most azért nem tűnik veszélyesnek a dolog, mert kiváló magyarázatunk van hozzá.

Ösztönösen úgy képzeljük ugyanis, hogy egy feldolgozási sor elemei időbeli értelemben is sorasan működnek: a sorban később következő türelmesen várakozik, amíg az előtte állók befejezik ténykedésüket, majd átveszi a *teljes* bemenetet, feldolgozza, és ha van még utána „valaki” a sorban, a kimenetet – szintén egyben – továbbadja neki.

Ha ez így lenne, programunknak tényleg működnie kellene egyetlen átmeneti fájllal is, hiszen a sed, és vele együtt a kimenet átirányítása csak második a sorban. Mire a fájl tartalma törlődik, a cat már régen kiolvasta és beküldte az ót a sed-del összekötő csőbe.

Magyarázatunk azonban sajnos téves. Egy csőhálózat tagjai valójában aszinkron módon működnek. Senki nem vár senkire, hanem mindenki a maga tempójában teszi, amit tennie kell, természetesen csak amint „bemenethez jut”. A UNIX feldolgozási sorai tehát nem modellezhetők egy vödörbrigáddal, ahol a víz végighalad ugyan a soron, de a teljes mennyiség minden pillanatban csak egy ponton tartózkodik.

Ezek tényleg csövek, amelyekben a víz a külső körülményektől független áramlik, és jószerével bármikor, bárhol lehet.

A külső körülmények közül a leglényegesebb a mag, pontosabban az ütemező. Ha ez sok időszeletet ad a sorban elől álló programnak, akkor az gyorsan ki tudja olvasni a fájl tartalmát, és mire az első adatmennyiségek megérkezik a sorozat utolsó eleméhez, a kimeneti (és egyben bemeneti) fájl már tényleg törölhető. Ha viszont az első elem – bármilyen okból kifolyólag – lassabban olvas, a törlés a feldolgozás kellős közepén történik meg.

Ez valószínűleg a legtüneményesebb hibajelenség, amit héjprogramból egyáltalán ki lehet „csiholni”. Az így elrontott programnak ugyanis vannak jó és rossz pillanatai. Ha szerencsénk van, tízszer egymás után helyesen hajtja végre a feladatot, aztán tizenegyedszerre elrontja. Teszi ezt annak ellenére, hogy minden nyiszor hajszálponosan ugyanazt csináljuk vele.

Számábrázolási pontosság

A felső határ...

Feladat

Az expr számábrázolási pontossága rendszerenként változhat. Írunk héjprogramot, amely megállapítja, hogy saját UNIX-változatunkon mi a legnagyobb, még kezelhető szám.

Ötletek

Kezdjük el az expr segítségével előállítani a kettő hatványait. Ha túlcordulás következik be, a legmagasabb helyérték utáni előjelbit egyre áll, vagyis eredményként negatív szám keletkezik. A túlcordulás ténye tehát az előző eredménnyel való összehasonlításból egyértelműen kiderül.

A megoldás

7.14. lista

```

1: #!/bin/sh
2: # Az expr számábrázolási pontosságának ellenőrzése
3:
4: kitevo=0
5: hatvany=1
6: elozen_hatvany=0
7:
8: while [ $hatvany -gt $elozen_hatvany ]
9: do
10:   elozen_hatvany=$hatvany
11:   hatvany=`expr $hatvany \* 2`
12:   kitevo=`expr $kitevo + 1`
13: done
14:
15: legnagyobb=`expr $elozen_hatvany - 1`
16: legnagyobb=`expr $legnagyobb + $elozen_hatvany`
17:
18: echo "Ezen a rendszeren az expr
           `expr $kitevo + 1` bites számábrázolást használ."
19: echo "A legnagyobb ábrázolható szám:
           2^`expr $kitevo + 1`-1 = $legnagyobb"

```

A megvalósítás a fenti leírás után különösebb magyarázatot feltehetőleg nem igényel.

Csomagoljunk

Nyilas Misi pakkot kapott...

Feladat

A UNIX „beágyazott dokumentum” szolgáltatására támaszkodva írunk olyan héjprogramot, amely a parancssori paraméterként megadott szövegfájlok ból önkibontó csomagot készít. (Ez a program shar néven része a legtöbb UNIX rendszernek.)

Ötletek

A „trükk” azon alapulhat, hogy a csomag darabjait néhány cat parancs kíséretében egyetlen fájlba írjuk, úgy, hogy azokat a beágyazott dokumentumokra jellemző hatalró jelek válasszák el egymástól. Ha ezt a szövegfájlt mint héjprogramot végrehajtjuk, a kimeneten természetesen a „becsomagolt” fájlok tartalma fog megjelenni. Ha még ezek eredeti néven való átirányításáról is gondoskodunk, a csomag futtatáskor minden részletében tökéletesen „visszaadja” az eredeti tartalmat.

A megoldás

7.15. lista

```
1: #!/bin/sh
2: # Önkibontó csomag készítése szövegfájlokból
3:
4: echo '# A kicsomagoláshoz futtassa programként ezt a fájlt!'
5: echo '# sh fájlnév <ENTER>'
6:
7: for nev in $*
8: do
9:   if [ -f $nev ]
10:  then
11:    echo "echo $nev 1>&2"
12:    echo "cat >$nev <<'==== End of $nev ==='"
13:    cat $nev
14:    echo "==== End of $nev ==="
15:  else
16:    echo "A $nev nevű fájl nem létezik!" 1>&2
17:  fi
18: done
```

A UNIX világában általában a közepes méretű programrendszerök forráskódját terjesztik ilyen csomagok formában. A felhasználó egyetlen fájlt (a csomagot) tölt le, majd azt futtatva megkapja a forráskód darabjait.

A 4-5. sorokban két sornyi tájékoztató szöveget írunk a csomag elejére. Ha a felhasználó belenéz, rögtön tudja, mit kell tennie. Figyeljük meg, hogy a sorokat # karakter előzi meg, így a keletkező héjprogramban megjegyzésnek minősülnek majd.

A program parancssori paraméterként várja a becsomagolandó fájlok neveit. A 7. sorban induló for ciklus ezeken megy végig.

A fájlok sorai valamennyi kiegészítéssel együtt a szabványos kimenetre kerülnek, így azt a felhasználónak kell majd egy fájlba irányítania. Ha ezt nem teszi, a csomag tartalma a képernyőn jelenik meg.

A 11. sorban egy echo-val kiírunk egy „echo”-t, majd kiírjuk a pillanatnyi fájl nevét, és gondoskodunk róla, hogy a „kiírt echo” kimenete a szabványos hibacsatornára kerüljön. Ennek hatására a csomag futtatásakor az éppen kibontás alatt álló fájl neve megjelenik a képernyőn.

Ezután hasonló módon kiírjuk a majdan a kiírást végző cat parancsot a << karakterrel és az '==== End of xxx ===' alakú jelet, ahol xxx a fájl neve. Ez fontos momentum, mivel ez biztosítja a határolójelek egyediségét. (Máskülönben a cat parancsok elvétenék a fájlok végét.) Végül a 13. sorban a kimenetre küldjük a fájl tartalmát, majd a határolójel másolatát is.

Ha valamelyik megadott fájl neve téves lenne, a szabványos hibacsatornán keresztül hibaüzenetet kapunk (16. sor), a feldolgozás azonban folytatódik. Az üzenet hibacsatornára való átirányításának itt kettős jelentősége van. Egyszer megakadályozzuk vele, hogy a hibaüzenet bekerüljön a csomagba, másrészt akkor is a képernyőre kerül, ha a kimenetet fájlba irányítottuk.

8

Gyakorlatok II.

Segédprogramok

Míg az előző fejezet példái meglehetősen elméletiek voltak, az itt bemutatandók gyakorlatiasabbak lesznek. Ezek a programok olyan valós feladatok megoldásai, amelyekkel bárki nap mint nap találkozhat.

Igen vagy Nem?

Ön dönt...

Feladat

Írunk függvényt, amely tetszőleges eldöntendő kérdésre igen vagy nem választ vár a felhasználótól, és addig „nem tágít”, amíg azt meg nem kapja. Tegyük lehetővé a hívó program számára, hogy engedélyezhesse, illetve tilthassa a CTRL+C-vel való kilést.

Ötletek

A feladat jelenlegi ismereteink birtokában viszonylag egyszerű, de törekedjünk az alapos megoldásra. A megválaszolandó kérdést egy végtelen ciklus segítségével tehetjük fel, amiből határozott válasz esetén a break parancsal léphetünk ki. A kiírandó szöveg legyen a függvény első, a jelek elfogását engedélyező kapcsoló pedig a második paramétere. Ha a felhasználó igennel válaszolt, a függvény visszatérési értéke a UNIX szokásainak megfelelően legyen 0 (logikai IGAZ), nemleges válasznál pedig 1 (logikai HAMIS).

A megoldás

8.1. lista

```
1: #!/bin/sh
2:
3: igen_nem()
4: {
5: # "Biztonsági" kapcsoló beállítása
6: KAPCSOLO="x"
7: if [ $# -eq 2 ]
8: then
```

```
9:         KAPCSOLO=$2
10:        fi
11: # Jelek elfogása
12: if [ $KAPCSOLO = "-s" ]
13: then
14:     trap '' 0 1 2 3 15
15:     fi
16: # Szükség esetén az alapértelmezett kérdés beállítása
17: VALASZ=""
18: if [ $# -eq 0 ]
19: then
20:     SZOVEG="Igen/Nem? "
21: else
22:     SZOVEG=$1
23:     fi
24: # Várakozás felhasználói reakcióra
25: while :
26: do
27:     echo -n $SZOVEG"
28:     read VALASZ
29:     case "$VALASZ" in
30:         [iI] | igen | IGEN | Igen )
31:             visszateresi_ertek=0
32:             break;;
33:         [nN] | nem | NEM | Nem )
34:             visszateresi_ertek=1
35:             break;;
36:         * ) echo "Válaszoljon az i vagy n billentyűvel!" ;;
37:     esac
38:     done
39: # Jelek fogadásának engedélyezése
40: if [ $KAPCSOLO="-s" ]
41: then
42:     trap 0 1 2 3 15
43:     fi
44: # Visszatérés a hívóhoz
45: return $visszateresi_ertek
46:
47: }
48:
49: if igen_nem "Kilép a programból?" -s
50: then
51:     echo "Kilépés felhasználói parancsra."
52:     exit 1
53: else
54:     echo "A program szabályosan befejeződött."
55:     exit 0
56: fi
```

A megoldás lényegét jelentő függvény a 3-47. sorokban látható. A 49-56. sorokat kitöltő „főprogram” most csak a működés bemutatására szolgál. A 6-10. sorokban egy változóban helyezzük el a felhasználó által megadott „biztonsági” kapcsoló (\$2) értékét. Ez első látásra teljesen fölöslegesnek tűnik, hiszen \$2-t közvetlenül is használhatnánk. A baj csak az, hogy ha a felhasználó nem adta meg a -s kapcsolót, akkor \$2 nem üres, hanem nem meghatározott, így a vele végzett bármilyen művelet (például a 12. sorban elvégzett összehasonlítás) hibaüzenetet eredményez. A 14. sorban a -s kapcsoló megadásától függően kapcsoljuk be a programot megszakító jelek elfogását és figyelmen kívül hagyását.

Elképzelhető, hogy a felhasználó (programozó) nem akar a függvénynek kiírandó kérdést megadni, például azért, mert az egy előző kiíratás eredményeképpen már a képernyőn van. Ilyen esetekre egy alapértelmezett alapkérdést állítunk be (18-23. sorok).

Az Ötletek szakaszban említett végtelen ciklus a 25. sorban kezdődik. (A „:” karakter az üres utasítás megfelelője, melynek visszatérési értéke minden igaz.) A ciklusból csak úgy lehet kilépni, ha a felhasználó igennel vagy nemmel válaszol (két break parancs a 32. és 35. sorban). A választ vizsgáló case szerkezetben többféle „helyes” választ is megengedtünk mind az „igen”, mind a „nem” kimondására. Ha a felhasználónak ezek ellenére sem sikerülne rátalálnia a megfelelő gombra, tájékoztató szöveget kap (36. sor), majd a ciklus egyszerűen folytatódik.

A ciklusból való kilépés után feltétlenül engedélyeznünk kell a jelek fogadását (40-43. sorok), mivel ha ezt nem tesszük meg, a főprogram is védetté válik a megszakítások ellen.

Egyszerű menürendszer

A' la carte...

Feladat

Írunk olyan függvényt, amely megjeleníti egy menü pontjait, és lehetővé teszi, hogy a felhasználó a pont sorszáma alapján válasszon közülük. A különböző „élet-helyzetekhez” igazodva tegyük lehetővé, hogy egy kapcsolóval szabályozni lehessen az előzetes képernyőtörést, illetve – önhívó használat esetére – a menüpontok kiírását. (Ha a menü már a képernyőn van, nem kell még egyszer megjeleníteni.)

Ötletek

A menüpontok szövegét célszerű parancssori paraméterként átadni a függvénynek. Mivel a pontok szövege szóközöket is tartalmazhat, a legegyszerűbb, ha ezeket az eredeti szövegben aláhúzás karakterrel jelezzük, és csak a kiíráskor változtatjuk váródi szóközzé. A választott pont sorszáma legyen a függvény visszatérési értéke.

A megoldás

8.2. lista

```
1: #!/bin/sh
2: # Egyszerű menürendszer
3:
4: menu()
5: {
6:
7: # A képernyőtörölést vezérlő kapcsoló felismerése
8:    kiiras=0
9:    if [ "$1" = "-c" ]
10:   then
11:      shift
12:      kiiras=1
13:      clear
14:    elif [ "$1" = "-p" ]
15:    then
16:      shift
17:      kiiras=1
18:    fi
19:
20: # Menüpontok kiírása
21: if [ $kiiras -eq 1 ]
22: then
23:   sorszam=1
24:   while [ $sorszam -le $# ]
25:   do
26:     pont=`eval echo $"$sorszam | sed "s/_/ /g"`
27:     echo "$sorszam. $pont"
28:     sorszam=`expr $sorszam + 1`
29:   done
30:   echo "Válasszon egy menüpontot! (1-$#)"
31: fi
32:
33: # Felhasználói válasz fogadása
34: while :
```

```

35:      do
36:          read -s -n1 valasz
37:          if [ $valasz -ge 1 -a $valasz -le $# ] 2>/dev/null
38:          then
39:              return $valasz
40:          break
41:      fi
42:      done
43:
44:  }
45:
46: # FŐPROGRAM
47:
48: MENULISTA="Első_pont Második_pont Harmadik_pont
49:                               Negyedik_pont Kilépés"
50: menu -p $MENULISTA
51: valasztott_pont=$?
52:
53: while :
54: do
55:     case $valasztott_pont in
56:         1|2|3|4) echo "Ön a(z) $valasztott_pont. pontot választotta."
57:                     menu $MENULISTA
58:                     valasztott_pont=$? ;;
59:         5) echo "Ön a kilépést választotta"
60:                     exit 0;;
61:     esac
62: done

```

A főprogram jelen esetben csak a függvény működésének bemutatására szolgál. A visszatérési érték alapján egyszerűen kiírja, hogy a felhasználó hányadik pontot választotta a menüből. A menüpontokat célszerű egy változóban (esetünkben MENULISTA) tárolni, hiszen másként a függvény minden egyes meghívásakor ki kellene azokat írni. Figyeljük meg, hogy a menüt kezelő függvény visszatérése után azonnal kiolvassuk a visszatérési értéket tartalmazó belső változó (\$?) tartalmát (51. és 58. sor). Erre azért van szükség, mert a következő parancs már módosítaná azt.

Maga a függvény az esetleges parancssori kapcsolók vizsgálatával indul. A -c hatására törli a képernyőt, a -p kapcsolóval pedig a menüpontok megjelenítésére utasíthatjuk, törlés nélkül. A kapcsolót a shift segítségével mindenkor elütetjük, hogy a továbbiakban ne zavarjon.

A 20-31. sorokban megjelenítjük a pontokat, miközben gondoskodunk a „_” karakterek szóközre való lecseréléséről is (26. sor). A 34. sorban induló végtelen ciklusban fogadjuk a felhasználói választ. Mivel a menüpontok parancssori paraméterként érkeznek, ezek száma általános (hordozható) esetben nem haladhatja meg a tízöt. Ez két dolgot jelent. Függvényünk egyrészt legfeljebb kilenc pontot képes kezelni, másrészt a felhasználói válasz biztosan egyjegyű szám, vagyis egyetlen karakter lesz. A 36. sorban ezt ki is használtuk, hiszen a -s kapcsolóval megtiltottuk a begépelt karakter megjelenítését, a -n1-gyel pedig jeleztük, hogy egyetlen karaktert várunk. A 37. sorban két numerikus összehasonlítással vizsgáljuk meg, hogy a válasz értelmes-e. Mivel a felhasználó bármit, akár betűket is begépelhet, a test parancs hibacsatornáját el kellett nyomnunk, mivel a betűk számként való összehasonlítása hibát eredményez.

Ha a válasz érvényes volt, a végtelen ciklust megszakítjuk, és a választott pont sorszámát a visszatérési értékben visszaadjuk a főprogramnak. Próbáljuk úgy is futtatni a programot, hogy az 50. sorban nem a -p, hanem a -c kapcsolóval indítjuk a függvényt!

Interaktív parancsértelmező

Hyppolit, a lakáj...

Feladat

Írunk olyan függvényt, amely egyedi célú parancsértelmezőként képes működni, vagyis bizonyos szavak begépelésére, bizonyos előre megadott műveleteket (függvényeket) tud végrehajtani. Természetesen fel kell ismernie a hibás parancsokat, képesnek kell lennie a parancsok paraméterezősére, no és rendelkeznie kell egy egyszerű beépített súgóval is. A cél most is az „újrahasznosíthatóság”, vagyis parancsértelmezőnknek a parancskészletet, a műveleteket és a súgószövegeket paraméterként kell kezelnie (és nem „bedrótozva” tartalmaznia).

Ötletek

Bár a megvalósítandó képességek listája első látásra talán „elborzasztó”, a program nem is lesz olyan hosszú. A parancsok legyenek egyszavasak, a súgószövegek pedig legfeljebb egysorosak. (Aki ennél többre vágyik, az úgysem héjprogramot fog írni.) minden parancshoz tartozzon egy függvény, amit a parancsértelmező a kulcsszót felismerve meghívhat. (Maga a függvény természetesen elindíthat más függvé-

nyeket, sőt héjprogramokat is.) A parancsok és a meghívandó függvények listáját egy-egy szóközökkel tagolt karakterláncként adhatjuk át. A súgószövegek feltétlenül maguk is tartalmaznak majd szóközöket, így ott a szomszédos elemek elválasztására valamilyen más, ritkán használt karaktert kell alkalmaznunk. Legyen ez például a pontosvessző. A felhasználói parancsokat egy while ciklus és egy read parancs kombinációjával fogadhatjuk, a súgóhoz pedig tartozzon egy „bedrótozott” parancs (például a „help”).

A megoldás

8.3. lista

```
1: #!/bin/sh
2: # Interaktív parancsértelmező
3:
4: # A függvények meghatározásai
5:
6: fp1()
7: {
8:     echo "Első függvény"
9: }
10:
11: fp2()
12: {
13:     echo "Második függvény"
14:     i=1
15:     for p in $*
16:     do
17:         echo ${!i}. paraméter: ${p}
18:         i=`expr $i + 1`
19:     done
20: }
21:
22: fp3()
23: {
24:     echo "Harmadik függvény"
25: }
26:
27: kilepes()
28: {
29:     echo "Kilépünk a programból"
30:     exit 0
31: }
32:
33: ertelmezo()
34: {
```

```
35:
36:     # Induló készenléti jel kiírása
37:
38:     echo -n "PROMPT> "
39:
40:     # Parancsok fogadása
41:
42:     while read parancssor
43:     do
44:
45:         # A parancs és a paraméterek különválasztása
46:
47:         parancs=`echo "$parancssor" | awk '{print $1}'` 
48:         parameterek=`echo "$parancssor" | \
49:                         awk '{for(i=2;i<NF;i++) printf "%s ", $i}'` 
50:
51:         # Súgószöveg megjelenítése
52:
53:         if [ $parancs = "help" ]
54:         then
55:             echo "Érvényes parancsok:"
56:             i=1
57:             for p in $1
58:             do
59:                 echo -n ${p} : "
60:                 echo "$3" | awk -v s=$i -v FS=\; '{print ""$s""}' 
61:                 i=`expr $i + 1` 
62:             done
63:
64:         else
65:
66:             # A parancs sorszáma
67:
68:             sorszam=`echo "$1" | awk -v p=$parancs \
69:                         'BEGIN {s=0}
70:                           {for(i=1;i<NF;i++) if(p==$i) s=i}
71:                           END {print s}'` 
72:
73:             # A parancs végrehajtása
74:
75:             if [ $sorszam -eq 0 ]
76:             then
77:                 echo "Értelmetlen parancs!"
78:             else
79:                 fuggveny=`echo "$2" | awk -v s=$sorszam \
80:                               '{print $s}'` 
81:                 fuggveny=$fuggveny" "$parameterek"
82:                 eval "$fuggveny"
```

```

83:         fi
84:         fi
85:
86:         # Új készenléti jel kiírása
87:         echo -n "PROMPT> "
88:
89:     done
90: }
91:
92: PARANCSOK="p1 p2 p3 quit"
93: FUGGVENYEK="fp1 fp2 fp3 kilepes"
94: SUGOSZOVEGEK=" Első parancs; Második parancs; \
95:   Harmadik parancs; Kilépés a programból"
96:
97: echo "Ez egy parancsértelmező héjprogram"
98: echo "Segítség kérése: help <ENTER>"
99:
100: ertelmezo "$PARANCSOK" "$FUGGVENYEK" "$SUGOSZOVEGEK"

```

A 91. sorban kezdődő főprogram ismét csak a használatot kívánja bemutatni, a „lényeget” az „ertelmezo” nevű függvény tartalmazza. Ez a 100. sorban három héjváltozóban kapja meg az értelmezhető parancsok, a hozzájuk tartozó függvények (meghatározásaiak a 4-32. sorokban), és a súgószövegek listáját.

A parancsértelmező függvény először kiír egy készenléti jelet, majd elindul a parancsokat fogadó ciklus. A parancsok után tetszőleges számú paraméter is megadható, maga a parancs minden a begépelt sor első szava. A ciklusmagban tehát először különválasztjuk az első szót, és a paramétereket tartalmazó többet (47-49. sorok).

Ha a begépelt parancs a „help” szó volt, megjelenítjük a súgószövegeket. Egy for ciklussal végigmegyünk az első paraméterként megadott parancslistán, majd a harmadik paraméterben szereplő súgószövegek közül kiválasztjuk a megfelelő sor-számút. E két adatot egymás mellett, kettősponttal elválasztva jelenítjük meg a képernyőn (59-60. sor). A ciklusnak természetesen minden lépésben növelnie kell azt a sorszámot is, amely alapján a 60. sor awk programja „tájékozódik”.

Ha a felhasználó nem segítséget kért, akkor a 68-71. sorokban megállapítjuk, hogy a lista hányadik parancsát gépelte be. Ha az eredmény nulla, a begépelt szöveg egyik listaelemmel sem volt azonos, tehát értelmetlen parancs. Ezt a tényt a 77. sorban közöljük a felhasználóval, majd kiírjuk a következő készenléti jelet.

Ha a parancs érhetőnek bizonyul, a függvények listájából (második parancssori paraméter) kiemeljük a megfelelő sorszámút (79. sor), utána másoljuk az esetlegesen megadott paramétereket (81. sor), majd egy eval segítségével elindítjuk. A többi már a függvény dolga.

Figyeljük meg, hogy ebben a programban az érték szerinti hivatkozások nagyon sok helyen kettős idézőjelben szerepelnek. Ez most létfontosságú, mivel a szóközt is tartalmazó karakterláncokat a legtöbb helyen csak így lehet egyben tartani.

Számból szöveg

Számlaadási kötelezettség...

Feladat

Cégünk kereskedelemmel foglalkozik, és a forgalmazott áruk árát egy központi adatbázisban tárolja. A számlákon az összegnek számmal és betűvel kiírva is szerepelnie kell, az adatbázis azonban természetesen csak a számokat tartalmazza. Írunk olyan héjprogramot, amely a szám alapján előállítja annak szöveges formáját (például a 218-ból a „kétszáztizenyolc” karakterláncot). A „biztonság kedvéért” a program legyen képes milliárdos nagyságrendű számokat is kezelní.

Ötletek

Mindenekelőtt egy kis nyelvtan. A magyar nyelvben a számok szöveggel való leírásakor a jegyeket gondolatban hármas csoportokba rendezzük, ezeket különálló számokként megnevezzük, majd a kapott számnevek után a „milliárd”, „millió” és „ezer” szavakat illesztjük. Ha a szám nagyobb, mint 2000, az egyes részeket kötőjellel elválasztva írjuk le. A kerek ezreseket és milliókat egybeírjuk.

Tulajdonképpen nincs más dolgunk, mint ezt az algoritmust héjprogram formájában megvalósítani. Első lépésként a parancssori paraméterként kapott számba a megfelelő helyekre szóközöket illesztünk (hármas tördelés). A feldolgozás egyes lépéseire célszerű lesz külön függvényeket írni. Szükségünk lesz tehát egy olyan függvényre, amely az egyes és százas, és egy másikra, amely a tízes helyiértéken levő számjegyeket képes szöveggé lefordítani. Ügyeljünk rá, hogy a tízesek megnevezésekkel néha az is számít, hogy az egyes helyiértéken mi van („tíz”/„tizen”, illetve „húsz”/„huszon”). Célszerű egy olyan függvényt is írni, amely egy teljes szám-hármas feldolgozását elvégzi.

Ha a hármas mezők elkészültek szöveges formában, nincs más hátra, mint helyzetük alapján hozzájuk illeszteni a megfelelő utótagot (ezer, millió, milliárd) és a szükséges számú kötőjelet.

A megoldás

8.4. lista

```
1: #!/bin/sh
2: # Számok szöveges formáját előállító program
3:
4: PROGRAMNEV=`basename $0`
5: UTOTAGOK="milliárd millió ezer"
6:
7: # Az egyes és százas helyiértékeket feldolgozó függvény
8:
9: szamjegy()
10: {
11:     case $1 in
12:         1) echo "egy";;
13:         2) echo "kettő";;
14:         3) echo "három";;
15:         4) echo "négy";;
16:         5) echo "öt";;;
17:         6) echo "hat";;;
18:         7) echo "hét";;;
19:         8) echo "nyolc";;;
20:         9) echo "kilenc";;
21:     esac
22: }
23:
24: # A tízes helyiértéket feldolgozó függvény
25:
26: tizes()
27: {
28: # A tíz és húsz szavaknál számít az utolsó jegy is!
29:     case $1 in
30:         0) echo "";;
31:         1) if [ $2 -eq 0 ] ; then echo "tíz"; else echo "tizen"; fi;;
32:         2) if [ $2 -eq 0 ] ; then echo "húsz"; else echo "huszon"; fi;;
33:         3) echo "harminc";;
34:         4) echo "negyven";;
35:         5) echo "ötven";;
36:         6) echo "hatvan";;
37:         7) echo "hetven";;
38:         8) echo "nyolcvan";;
```

```
39:      9) echo "kilencven";;
40:      esac
41: }
42:
43: # Egy teljes hármas számmező feldolgozása
44:
45: mezo()
46: {
47:     MEZO=""
48:     jegy1=`echo $1 | awk -v FS="" '{print $1}'`"
49:     jegy2=`echo $1 | awk -v FS="" '{print $2}'`"
50:     jegy3=`echo $1 | awk -v FS="" '{print $3}'`"
51:
52:     if [ $jegy1 -ne 0 ]
53:     then
54:         MEZO=`szamjegy $jegy1`"száz"
55:     fi
56:
57:     if [ $jegy2 -ne 0 ]
58:     then
59:         MEZO=$MEZO`tizes $jegy2 $jegy3`"
60:     fi
61:
62:     if [ $jegy3 -ne 0 ]
63:     then
64:         MEZO=$MEZO`szamjegy $jegy3`"
65:     fi
66:     echo $MEZO
67: }
68:
69: # FŐPROGRAM KEZDETE
70:
71: # Van-e parancssori paraméter?
72:
73: if [ $# -ne 1 ]
74: then
75:     echo "Használat : $PROGRAMNEV szám" 1>&2
76:     exit 1
77: fi
78:
79: # A megadott karakterlánc értelmezhető-e számként?
80:
81: if ! expr $1 + 1 > /dev/null 2>&1
82: then
83:     echo "Ez nem szám!" 1>&2
84:     exit 2
85: fi
86:
```

```
87: # A számjegyek számának meghatározása
88:
89: jegyek_szama=`echo $1 | awk '{print length($0)}'`
90:
91: if [ $jegyek_szama -gt 12 ]
92: then
93:     echo "Túl nagy szám!" 1>&2
94:     exit 2
95: fi
96:
97: # A kiegészítő nullák számának meghatározása
98:
99: nullak_szama=`expr $jegyek_szama % 3`
100:
101: if [ $nullak_szama -ne 0 ]
102: then
103:     nullak_szama=`expr 3 - $nullak_szama`
104: fi
105:
106: # A szám kiegészítése az elején a megfelelő számú nullával
107:
108: harmas_lista=`echo $1 | awk -v n=$nullak_szama \
109: '{for(i=1;i<=n;i++) printf "%s", "0" ; print $0 }'`
110:
111: # A nullákkal kiegészített szám hármas csoportokra tördelése
112:
113: harmas_lista=`echo $harmas_lista | awk \
114: 'BEGIN {FS=""}
115: {for(mezo=1;mezo<=NF/3;mezo++)
116: {
117:     for(jegy=1;jegy<=3;jegy++)
118:     {
119:         printf "%s", $( (mezo-1)*3+jegy)
120:     }
121:     printf " "
122: }
123: }'`
```

124:

```
125: # Az "utótagok" listájának előállítása
126:
127: mezok_szama=`echo $harmas_lista | wc -w`
128: utotagok=`echo $UTOTAGOK | awk -v m="$mezok_szama" \
129: '{for(i=1;i<=NF;i++){if(i>4-m) printf "%s ",$i}}'`
```

130:

```
131: # A számhármasok szöveggé alakítása
132:
133: eredmeny=""
134: for szamharmas in $harmas_lista
```

```

135: do
136:   if [ $szamharmas -ne 0 ]
137:     then
138:       eredmeny=$eredmeny`mezo $szamharmas` "
139:     else
140:       eredmeny=$eredmeny$"X "
141:     fi
142:   done
143:
144: # A mezőkre jellemző utótagok beillesztése
145:
146: for utotag in $utotagok
147: do
148:   eredmeny=`echo $eredmeny | sed "s/ /$utotag-/"`
149: done
150:
151: # A csupa nullát tartalmazó mezők kezelése
152:
153: eredmeny=`echo $eredmeny | sed "s/X.*--//g"`
154: eredmeny=`echo $eredmeny | sed "s/-X$//"`
155:
156: # A kötőjel használatának kikapcsolása 2000 alatt
157:
158: if [ $1 -le 2000 ]
159: then
160:   eredmeny=`echo $eredmeny | sed "s/-//"`
161: fi
162:
163: # Az eredmény visszaadása a szabványos kimeneten
164:
165: echo $eredmeny
166: exit 0

```

A 9-22. sorokban látható függénnyel egy korábbi fejezetben már találkoztunk, így ebben most nincs semmi említésre méltó. A tízes helyértéket kezelő függvény (26-41. sorok) már érdekesebb. Tartalmaz egy ugyanolyan case szerkezetet, mint az előző, de a bevezetőben említett „elnevezési anomália” miatt feltétlenül két parancssori paramétert vár. Ha a második (az egyes helyérték) tartalma nulla, a „tízen” helyett „íz”, „huszon” helyett pedig „húsz” lesz a visszatérési érték. Ha a tízesek helyén áll nulla, akkor üres karakterláncot ad vissza.

A harmadik, egy teljes hármas számmezőt feldolgozó függvény működésének megértéséhez először a főprogramot kell szemügyre vennünk.

A 71-77. sorokban ellenőrizzük, hogy a program egyáltalán kapott-e a parancssorban paramétert, a 81-85. sorokban pedig arról győződünk meg, hogy amit a felhasználó begépelet, értelmezhető-e számként. Ehhez az expr-nek azt a – korábban már alkalmazott – tulajdonságát használjuk ki, hogy nem szám paraméterek esetén visszatérési értéke 1, így felhasználható egy döntési szerkezet feltételeként. (A kimenetét a hibacsatornával együtt természetesen a /dev/null-ba kell irányítanunk, különben összerondítja a képernyőt.)

Mivel programunk „csak” milliárdos összegeket képes kezelni, a 89-95. sorokban kizártuk a 12 jegynél hosszabb számokat.

A bevezető ötletek felsorolásakor említettem, hogy a feldolgozás egyszerűsítése végett célszerű a beérkezett számot hármas csoportokra tördelni. Mivel a számot szöveggé alakító függvények úgyis üres karakterláncot küldenek vissza, ha bármely pozíciót nullát találnak, a hármas tördelést megkönnítendő megtehetjük, hogy ha a jegyek száma nem osztható hárommal, megfelelő számú nullával egészítjük ki az elején a parancssori paramétert. A 97-109. sorokban éppen ez történik. Előbb egy maradékos osztással meghatározzuk a szükséges nullák számát (99-104. sorok), majd egy awk program segítségével be is illesztjük azokat a megadott szám elő (108-109. sorok).

A hármas tördelést a 113-123. sorokban ismét egyetlen awk program segítségével valósítjuk meg. Hogy az utótagok („ezer”, „millió”, „milliárd”) közül hánnyra lesz szükség, azt nyilván a keletkezett hármas mezők száma szabja meg. A megfelelően csonkított listát a 127-129. sorokban állítjuk elő.

Ha mindezzel elkészültünk, jöhet a lényeg. A keletkezett számhármasokat a 134-142. sorokban látható ciklus alakítja szöveggé a harmadik – még nem tárgyalt – függvény segítségével. Az eredmény csökkenő helyiértékek szerint az „eredmeny” nevű változóba kerül, úgy, hogy a csupa nullából álló számhármasok helyét egy „X” jelzi, a többieket pedig egy-egy szóköz (138. sor vége!) választja el egymástól.

A szöveggé alakítást végző mezo nevű függvény egy hármas mezőt vár paraméterként. Ezt először különálló jegyekre tördeli (48-50. sorok), majd meghívja az egyes helyiértékeket kezelő két függvényt. Ezek kimeneteit a MEZO nevű változóban egymás után másolja, majd a teljes tartalmat visszaadja a főprogramnak.

Ez a szöveggé alakítást elvégző ciklus befejeződése után egy másik ciklussal beilleszti a szóközök helyére a megfelelő utótagokat (ezek listáját már korábban elkezítettük), valamint egy-egy kötőjelet (146-149. sorok). Ilyenkor természetesen a csupa nullából álló blokkot jelző „X” után is bekerül a pozíciójának megfelelő

utótag, amit törölünk kell. Ráadásul ebből a helyzetből is kétféle képzelhető el (sorközi vagy sorvégi „x”), így rögtön két sed programra van szükség, amelyek törlik (semmire cserélik) a megfelelő részeket.

Végezetül a kötőjeleket el kell távolítanunk a karakterláncból, ha a megadott szám kisebb, mint 2000 (158-161. sorok).

Telefonköltség kiszámítása

Közös író...

Feladat

Munkahelyünkön az egyik telefont (melléket) közösen használjuk egy kollégánkkal, de a költségeket külön álljuk. A céget saját telefonközpontja egy jelszóval elérhető weblapon havi bontásban rendelkezésünkre bocsátja a híváslistát. Írunk héjprogramot, amely a letöltött weblapok és az általunk hívott telefonszámok listája alapján kiszámítja a ránk eső költséget. Tételezzük fel, hogy nincsenek olyan számok, amelyeket mindenben szoktunk hívni. A program legyen képes éves és havi bontásban is megjeleníteni az adatokat, illetve összesíteni az egyes telefonszámokkal kapcsolatos hívási díjakat is.

A weblapok a kérdéses adatokat egy-egy táblázat formájában tartalmazzák. A táblázat oszlopai a következők: hívás kezdete (nap, óra, perc), időtartama, a hívott szám, körzet (helyi, mobil, vidék, külföld), fizetendő összeg. A lapok külön fájlokban találhatók, amelyek neve a következő alakú: év-hónap.htm, ahol a hónap neve szöveggel szerepel.

Ötletek

A HTML fájlokat nyilván át kell alakítanunk egyszerű szöveggé, hiszen a formázással kapcsolatos kódok csak zavarának a feldolgozást. Ehhez használhatjuk a html2text nevű ingyenes segédprogramot. Az említett táblázatból számunkra csak azok a sorok fontosak, amelyek általunk hívott számot tartalmaznak. Így a feldolgozás következő lépése az ezeket a sorokat tartalmazó átmeneti fájl előállítása lehet.

A különböző szempontok szerinti összegzéseket célszerű egy-egy függvény formájában megvalósítani. A legegyszerűbb a teljes költség kiszámítása, hiszen itt csupán össze kell adnunk a már elkészült átmeneti fájl utolsó mezőjében található értékeket.

Az évet és a hónapot a táblázat nem, csak a fájlok neve tartalmazza, így az éves és havi bontás előállításához ezt az adatot hozzá kell másolnunk minden sorhoz. Cél-szerű az évet minden sor elejére beilleszteni, mivel ha ez megvan, a teljes adattömeget egyetlen sort segítségével év szerint rendezhetjük.

A legnagyobb kihívást a hónapok szerinti rendezés fogja jelenteni, hiszen a hónapok szöveggel és nem számmal szerepelnek. Ezt úgy hidalhatjuk át, hogy az év mellett a hónapot is bemásoljuk a táblázatok minden sora elé, majd egy megfelelően összeállított sed program segítségével minden hónap neve elő beillesztjük a sorszámát (január elő 01-et, február elő 02-t stb.). Így ismét olyan adatbázishoz jutunk, amit a sort segítségével rendezhetünk.

Végezetül egy awk program segítségével megint össze kell adnunk a táblázatok utolsó oszlopában szereplő számokat, de most folyamatosan figyelni kell az év, illetve a hónap mező változását is. Váltásnál ki kell íratni a részeredményeket.

A megoldás

8.5. lista

```
1: #!/bin/sh
2: # Telefonköltség megállapítása a
   híváslistát tartalmazó HTML fájlok alapján
3:
4: trap "rm /tmp/$$htm2txt*.tmp /tmp/$$koltseg*.tmp
   2>/dev/null" EXIT
5:
6: HIVOTT_SZAMOK="szamok.dat"
7: HONAPOK="jan feb marc apr maj jun jul aug szep okt nov dec"
8:
9: # FÜGGVÉNYEK MEGHATÁROZÁSA
10:
11: # A teljes költség kiszámítása
12:
13: teljes_koltseg()
14: {
15:     cat $1 | awk '{sum+=$NF} END {print sum, "Ft"}'
16: }
17:
18: # Éves bontás
19:
20: eves_osszkoltseg()
21: {
22:     cat /tmp/$$koltseg.tmp | sort | \
```

```
23:      awk 'BEGIN {elozo_ev=""}
24:      {
25:          if($1==elozo_ev)
26:              {sum+=$NF}
27:          else
28:              {if(NR!=1) print elozo_ev".",sum" Ft" ; elozo_ev=$1; sum=$NF}
29:      }
30:      END {print elozo_ev".",sum" Ft" }'
31:  }
32:
33: # Havi bontás
34:
35: havi_osszkoltseg()
36: {
37:     sed_program1=""
38:     sed_program2=""
39:     hanyadik_honap=1
40:     for honap in $HONAPOK
41:         do
42:
43:         # Nulla beillesztése a hónap sorszáma elő
44:         if [ `echo -n $hanyadik_honap | wc -c` -eq 1 ]
45:             then
46:                 hanyadik_honap="0"$hanyadik_honap
47:             fi
48:
49:         # A számcsereit végző sed programok összeállítása
50:         sed_program1="$sed_program1"
51:                         s/$honap/$hanyadik_honap$honap/ ;
52:         sed_program2="$sed_program2"
53:                         s/$hanyadik_honap$honap/$honap/ ;
54:         hanyadik_honap=`expr $hanyadik_honap + 1`
55:
56:         done
57:
58:         cat /tmp/$$koltseg.tmp | sort | sed "$sed_program1" | sort | \
59:         sed "$sed_program2" | awk \
60:         'BEGIN {elozo_ev="" ; elozo_honap=""}
61:         {
62:             if($2==elozo_honap)
63:                 {sum+=$NF}
64:             else
65:                 {
66:                     if(NR!=1) print elozo_ev".",elozo_honap".",sum" Ft"
67:                     elozo_ev=$1; elozo_honap=$2; sum=$NF
68:                 }
69:         }
70:         END {print elozo_ev".",elozo_honap".",sum" Ft" }'
```

```
69: }
70:
71: # Hívott számok szerinti bontás
72:
73: szamok_osszkoltsége()
74: {
75:     for szam in `cat $HIVOTT_SZAMOK`
76:         do
77:             cat /tmp/$$koltseg.tmp | grep $szam >>
78:                             /tmp/$$koltseg1.tmp
79:             echo $szam" "`teljes_koltseg /tmp/$$koltseg1.tmp`"
80:             rm /tmp/$$koltseg1.tmp
81:         done
82:     }
83: # FŐPROGRAM
84:
85: # A megadott kapcsoló előzetes ellenőrzése
86:
87: if [ $# -ne 0 ]
88: then
89:     if [ `echo " \"$1\" | grep "\-\(h\|e\|s\)" | wc -l` -ne 1 ]
90:         then
91:             echo "Ismeretlen kapcsoló"
92:             exit 1
93:         fi
94:     fi
95:
96: # Az egyes hónapok adatait tartalmazó HTML fájlok kiválogatása
97:
98: grephonap=`echo "\($HONAPOK\)" | sed 's/ /\\\|/g'`"
99: lista=`ls | grep $grephonap`"
100:
101: # A HTML fájlok szöveggé alakítása
102:
103: for fajlnev in $lista
104: do
105:     ev_honap=`echo $fajlnev | sed "s/\./-/"`
106:             awk 'BEGIN {FS="-"}{print $1,$2}'`"
107:     html2text -nobs $fajlnev | tr "[|_]" [" "]] | \
108:     grep "[0-9]\:" >> /tmp/$$htm2txt1.tmp
109:     cat /tmp/$$htm2txt1.tmp | \
110:     awk -v evho="$ev_honap" '{print evho,$0}' >>
111:                                     /tmp/$$htm2txt.tmp
112:     rm /tmp/$$htm2txt1.tmp
113: done
114: # A listában szereplő számok kikeresése
115:
```

```

116:   for szam in `cat $HIVOTT_SZAMOK`
117:   do
118:     cat /tmp/$$htm2txt.tmp | grep $szam >> /tmp/$$koltseg.tmp
119:   done
120:
121:   # A kapcsolókkal vezérelt műveletek indítása
122:
123:   if [ $# -eq 0 ] # Az összköltség kiírása (alapértelmezett
124:       viselkedés)
125:   then
126:     teljes_koltseg /tmp/$$koltseg.tmp
127:   else                      # Az egyes kapcsolók kezelése
128:     case $1 in
129:       "-h") havi_osszkoltseg;;
130:       "-e") eves_osszkoltseg;;
131:       "-s") szamok_osszkoltsege;;
132:     esac
133:   fi
134:   exit 0

```

- ▶ A feldolgozás során több átmeneti fájlra lesz szükségünk, amelyek törléséről a 4. sorban megadott trap parancsal gondoskodunk. Ezt követi a hívott számokat tartalmazó fájl nevének megadása, valamint a hónapok rövidítése (ezek szerepelnek a fájlnevekben). A 9-82. sorok a különböző szerepeket betöltő függvények meghatározását tartalmazzák. Ezekre később még visszatérünk.

A 87-94. sorokban csupán azt ellenőrizzük, hogy a felhasználó nem adott-e meg hibás kapcsolót. Figyeljük meg, hogy a grep bemenetére nem egyszerűen \$1-et küldjük az echo-val, hanem megelőzi azt egy szóköz is. Erre a látszólag értelmetlen kiegészítésre azért van szükség, mert a -e kapcsolót az echo is „megérte”, és valóban kapcsolóként akarja kezelní. Erről csak úgy lehet lebeszélni, ha a „-e” karakterláncot megelőzi valami. (Második pozíción álló dolog már biztosan nem lehet kapcsoló.)

A program jelenlegi formájában a pillanatnyi könyvtárban keresi a feldolgozandó adatokat tartalmazó HTML fájlokat. Mivel nem zárható ki, hogy ebben a könyvtárban más weblapokat is tárolunk, össze kell állítanunk egy listát a megfelelő fájlok neveiből. (Nem elég a *.htm, sőt mivel a hónapok rövidítései nem azonos hosszúságúak, megfelelő számú kérdőjelet sem használhatunk.)

Ehhez a 98. sorban először összeállítunk a hónapok neveiből egy szabályos kifejezést, amely logikai VAGY jellet elválasztva tartalmazza azokat ("(jan|feb|marc...)"). Mivel ezt a következő sorban egy grep parancs fogja

használni, ügyelnünk kell a különleges karakterek különleges jelentésének bekapcsolására is (" \ (jan\ |feb\ |marc... \) "). Ezután a 99. sorban az ls kimenetéből kiválogatjuk a megfelelő fájlneveket, és a teljes kimenetet a „lista” nevű változóban helyezzük el.

A 103. sorban induló for ciklus ezen a listán fog végigmenni, és a html2text programmal az összes weblapot egyszerű szöveggé alakítja. Az egész kimenet egyetlen átmeneti fájlba kerül, de mivel bizonyos feldolgozási műveletek során tudnunk kell majd, hogy melyik sor, melyik évben és melyik hónapban keletkezett, ezt az adatot be kell illesztenünk minden sor elé. Ehhez mindenekelőtt a 105-106. sorokban kiszűrjük az éppen feldolgozás alatt álló fájl nevétől az évszámot és a hónapot, és eltesszük egy változóba.

A HTML fájlok átalakítását a 107-108., a kiegészítő adat beszúrását pedig a 109-110. sorokban láthatjuk. A html2text parancs -nobs kapcsolója azt akadályozza meg, hogy a szöveg formázásához szükséges vezérlőkarakterek a kimenetre kerüljenek, a sorban utána következő tr segítségével pedig a táblázat függőleges és vízszintes elválasztó vonalait jelölő karaktereket szűrjük ki. Az utolsóként álló grep parancs csak azokat a sorokat engedi át a szövegből, amelyek számunkra érdekes bejegyzést tartalmaznak (a weblapon egyéb szöveg is lehet). Mivel a feladat leírása szerint a táblázat első oszlopa a hívás kezdetét tartalmazza, olyan sorokat keresünk, amelyekben van legalább egy számjegyet követő kettőspont (az időbényegen az órát, percet és másodpercet tartalmazó mezőket kettőspont választja el egymástól). Ez egy viszonylag lazán megfogalmazott szűrési feltétel, de esetünkben elegendő.

A 110. sorban induló awk program az évet és hónapot az előbb előállított héjváltozóból kapja meg. Az érték szerinti hivatkozást azért kellett kettős idézőjelek közé tenni, mert a változó szóközt is tartalmaz.

A 116-119. sorokban látható for ciklus az általunk hívott számokat válogatja ki az előbb elkészült – immár tisztán szöveges – adatbázisból, és a kimenetet egy másik átmeneti fájlban helyezi el. Ezt fogják felhasználni a költséget számító függvények, amelyeket a 123-132. sorokban indítunk a kapcsoló vizsgálata alapján.

A teljes költség kiszámítása a legegyszerűbb, hiszen nincs más dolgunk, mint egy awk program segítségével összeadni a táblázat utolsó oszlopában található (\$NF) számokat (13-16. sorok). Mivel ezt a függvényt „másra” is lehet majd használni, úgy írtam meg, hogy a feldolgozandó fájl nevét parancssori paraméterként várja.

Az éves összköltség (20-31. sorok) kiszámításánál először arról kell gondoskod-

nunk, hogy az azonos évszámhoz tartozó bejegyzések egymás után következzenek. Mivel az évszámot az első mező tartalmazza, ehhez elegendő egy sort beiktatása a feldolgozási sorba. Az utána következő awk program figyeli az első mező tartalmát, és ha változást észlel, kiírja az addig begyűjtött összköltséget. Mivel az utolsó évszámnál nem lesz ilyen váltás, szükségünk van egy ugyanilyen kiírást végző END blokkra is.

Az egyes számokkal kapcsolatos összköltség kiszámítása (73-81. sorok) az a pont, ahol „újrahasznosíthatjuk” az elsőként tárgyalt, összköltséget kiszámító függvényt. Itt ugyanis egy ciklussal végigmehetünk az egyes számokon, egy átmeneti fájlba ki-válogathatjuk az azokat tartalmazó sorokat, majd ezzel a függénnyel elvégezhetjük az összegzést. Pontosan ez történik a 77-79. sorokban.

Amint azt a bevezetőben már előrebocsátottam, a legnehezebb feladat a havi összesítések előállítása. Első célunk, hogy minden hónap neve elé beillesszük a sorszámát, s így a sort segítségével rendezhető listához jussunk. Mivel kétjegyű sorszámok is vannak, az egyjegyűek előre be kell illesztenünk egy nullát is (46. sor). A cserék elvégzésére a 40-54. sorokban látható, a hónapok nevein végighaladó ciklussal először előállítunk két sed programot. Azért kettőt, mert a sorszámok beszúrására csak a rendezéshez van szükségünk, a kimenetben viszont ezeket már nem akarjuk viszontlátni. Így az első program (`sed_program1`) a sorszámok beszúrá-sát, a második (`sed_program2`) pedig a törlését végzi. (Ehhez nyilván csak a csere sorrendjét kell megváltoztatni.)

Az 56-68. sorok egyetlen „gigantikus” feldolgozási sort tartalmaznak. Az általunk hívott számok adatait tartalmazó átmeneti fájlt először évek szerint rendezzük. Ebben a listában a hónapok még betűrendben szerepelnek, így következő lépésként az előbb előállított első sed program segítségével beszúrjuk a hónapok neve előre a sorszámukat. A kimenetet egy újabb sort-tal immár hónapok szerint is rendezet-té lehetjük, majd – mivel a továbbiakban már semmi szerepük nem lesz – a sorszámokat a második sed programmal töröljük. A sort záró awk program ugyanúgy működik, mint az éves bontás előállításánál: figyeli, mikor változik a második oszlop (hónap) tartalma, és kiírja az addig begyűjtött összeget. (Mivel a lista évek szerint is rendezett, elegendő a hónap változásának figyelése.)

Felhasználók bejelentkezésének figyelése

A Nagy Testvér figyel téged!

Feladat

Írunk héjprogramot, amely a háttérben folyamatosan fut, és üzenetet küld a képernyőre, ha egy újabb felhasználó jelentkezett be vagy ki. Oldjuk meg, hogy programkat csak egy példányban lehessen elindítani, illetve hogy a -stop kapcsoló hatására leállítsa önmagát. A program írja ki a be- vagy kijelentkezett felhasználó valódi nevét is.

Ötletek

A pillanatnyilag bejelentkezett felhasználókról a who parancs segítségével kérhetünk listát, a valódi neveket pedig a /etc/passwd fájlban találjuk meg. Ha a who kimenetét egy átmeneti fájlban tároljuk, amelynek tartalmát rendszeresen frissítjük, a régi és új lista összehasonlításával megállapíthatjuk, milyen változások történtek a legutolsó ellenőrzés óta. (Aki a régi listában szerepel, de az újban nem, az nyilván kijelentkezett, aki viszont csak az újban van meg, az belépett.)

A megoldás

8.6. lista

```

1: #!/bin/sh
2: # Be/Kijelentkezések figyelése
3:
4: PROGRAMNEV='basename $0'
5: NAME1="/tmp/$$.wholist"
6: NAME2="/tmp/$$.wholist_elozo"
7:
8: # Kilencestől eltérő jellel való leállítás kezelése
9:
10: trap "rm ./login.lock $NAME1 $NAME2 ; exit 1" 1 2 3 15
11:
12: # Zároló fájl vizsgálata és kezelése
13:
14: if [ -f ./login.lock ]
15: then
16:   if [ `echo $1 | grep "stop" | wc -l` -eq 1 ]
17:   then

```

```
18:         killpid=`cat ./login.lock`
19:         kill -9 $killpid
20:         rm /tmp/$killpid.wholist 2>/dev/null
21:         rm /tmp/$killpid.wholist_elozo 2>/dev/null
22:         rm ./login.lock
23:         exit 2
24:     fi
25:     echo "Már fut egy példány a programból!"
26:     echo "Leállítás: $PROGRAMNEV -stop"
27:     exit 1
28: fi
29:
30: # Saját PID-jét a zároló fájlba helyezi
31:
32: echo $$ > ./login.lock
33:
34: # Ha nem tudott volna takarítani az előző leállítás során
35:
36: rm /tmp/*.wholist 2>/dev/null
37: rm /tmp/*.wholist_elozo 2>/dev/null
38:
39: # A kezdeti lista előállítása
40:
41: who > $NAME1
42:
43: # Ciklikus ellenőrzés
44:
45: while :
46: do
47:     mv $NAME1 $NAME2
48:     who > $NAME1
49:
50:     # Bejelentkezők
51:     while read sor
52:     do
53:         if ! cat $NAME2 | grep "$sor" > /dev/null
54:         then
55:             felhasznalo=`echo $sor | awk '{print $1}'`
56:             valodi_nev=`cat /etc/passwd | grep $felhasznalo | \
57:                         awk -v FS=: '{print $5}'`
58:             echo $felhasznalo \($valodi_nev\) bejelentkezett
59:         fi
60:     done < $NAME1
61:
62:     # Kijelentkezők
63:     while read sor
64:     do
65:         if ! cat $NAME1 | grep "$sor" > /dev/null
```

```

66:      then
67:          felhasznalo=`echo $sor | awk '{print $1}'`
68:          valodi_nev=`cat /etc/passwd | grep $felhasznalo | \
69:                      awk -v FS=: '{print $5}'` 
70:          echo $felhasznalo \($valodi_nev\) kijelentkezett
71:      fi
72:      done < $NAME2
73:
74:      # Várakozás a következő ciklusra
75:      sleep 5
76:
77:  done

```

A több példányban való futtatás megakadályozására nyilván egy zároló fájlra lesz szükségünk. Ez esetünkben egy, a pillanatnyi könyvtárban elhelyezett .login.lock nevű rejttet fájl lesz, ami a futó program folyamatazonosítóját (PID-jét) tartalmazza.

A program e fájl létezésének vizsgálatával indul. Ha van ilyen nevű bejegyzés a pillanatnyi könyvtárban, akkor már biztosan fut egy példány a programból, így a most indított csak megjelenít egy tájékoztató szöveget és leáll.

Ha a -stop kapcsolóval indítjuk a második példányt, akkor az kiolvassa a zároló fájlból az előzőleg indított program PID-jét, SIGKILL jellel leállítja azt, és eltakarítja utána mind a zároló, mind az átmeneti fájlokat. Azért célszerű ezt a durvának tűnő megoldást alkalmazni, mert ha a megállító jel a 75. sorban látható sleep végrehajtása közben érkezik (márpédig erre elég nagy az esély), akkor a program komótosan megvárja, amíg letelik az alvási időszak, csak aztán hajlandó megállni.

A gyorsaságért cserébe viszont a leállító programnak kell a takarítást elvégeznie, hiszen a SIGKILL jel nem fogható el, így a futó példánynak esélye sincs rá, hogy „tisztán távozzon”.

Vegyük észre, hogy a törlendő átmeneti fájlok nevében most nem a jelenlegi (\$\$), hanem az előző program folyamatazonosítója (\$killpid) szerepel, hiszen az hozta létre őket!

Előfordulhat természetesen, hogy a programot nem önmagával állítjuk le, hanem egy parancssori kill paranccsal, vagy egyszerűen egy rendszerleállás miatt szakad meg a futása. Az ilyen esetek ellen véd a 10. sorban megadott trap parancs, amely elfogja a leállító jeleket, törli a zároló és átmeneti fájlokat, és leállítja a programot.

Végezetül az is megeshet, hogy programunk nem „önmagától”, hanem a parancssorból kap egy SIGKILL jelet. Mivel ezt nem tudja kezelni, az átmeneti és a zároló fájlok is maradnak, ahol vannak. (Ahogy mondani szokás, balta ellen nincs orvosság.) A zároló fájlt ilyenkor is el lehet távolítani a -stop kapcsolóval (bár ilyenkor nincs mit megállítani), a szanaszét hagyott átmeneti fájlok csendes törléséről pedig a 36-37. sorok gondoskodnak.

Ha programunk nem talál magából másik példányt, a 32. sorban létrehozza a zároló fájlt, és beleírja a folyamatazonosítóját. A 41. sorban létrejön az első lista a bejelentkezett felhasználókról, majd az időszakos ellenőrzéseket a 45. sorban induló végtelen ciklus végzi. Ez először létrehoz egy újabb listát (48. sor), majd ezt két szempontból összeveti a régivel. Itt két, szerkezetileg teljesen azonos while ciklus következik, amelyek bemenetüket az egyik, illetve másik átmeneti fájlból veszik. Először az új listában keresünk olyan bejegyzéseket, amelyek a régiben még nem szerepeltek (bejelentkezők), majd a két lista szerepet cserél. Találat esetén a valódi nevet a /etc/passwd megfelelő sorának ötödik mezőjéből olvassuk ki, úgy, hogy az awk mezőelválasztó karaktereként a kettőspontot adjuk meg.

Mivel ezt a műveletsort nem volna célszerű túl gyakran ismételgetni, a 75. sorban öt másodperc szünetet iktatunk be.

Felhasználók lemezfoglalásának figyelése

Adj helyet magad mellett...

Feladat

Írunk héjprogramot, amely az összes befűzött fájlrendszeret megvizsgálva megállapítja, hogy az egyes felhasználók, illetve felhasználói csoportok összesen mekkora lemezterületet foglalnak el. A program jelenítse meg a felhasználók valódi neveit is, illetve legyen képes áttekintést adni arról, hogy ki melyik fájlrendszeren, mekkora helyet foglal el, és mely könyvtárszerkezetek tartoznak hozzá. Tételezzük fel, hogy a különböző felhasználókhöz tartozó könyvtárszerkezetek valamennyi fájlrendszeren közvetlenül a gyökérből nyílnak.


```
31:                     printf "%s\t%s\n", elozo,oszeg
32:                     elozo=$1 ; oszeg=$3
33:                 }
34:             }
35:         }
36:     END { printf "%s\t%s\n", elozo,oszeg }'
37:             | grep -v "root" > $TMP5
38: # -d kapcsoló esetén összesített és részletes lista
39:
40: if [ $D -eq 0 ]
41: then
42:     paste $TMP5 $TMP1
43: else
44:     paste $TMP5 $TMP1 > $TMP6
45:     while read sor
46:     do
47:         nev=`echo $sor | awk '{print $1}'`"
48:         echo $sor
49:         paste $TMP4 $TMP3 | grep $nev | awk
50:             '{printf "\t%s\t%s\n", $4,$3}'"
51:         done < $TMP6
52:     fi
53: }
54: # Eredmények megjelenítése csoportok szerinti bontásban
55:
56: csoportok()
57: {
58:     paste $TMP4 $TMP3 | awk '{print $2,$1,$3,$4}' | \
59:         grep -v "root" | sort | \
60:             awk '{ if(NR==1)
61:                     { elozo=$1; oszeg=$3 }
62:                     else
63:                     {
64:                         if($1==elozo)
65:                             { oszeg+=$3 }
66:                         else
67:                             {
68:                                 printf "%s\t%s\n", elozo,oszeg
69:                                 elozo=$1 ; oszeg=$3
70:                             }
71:                     }
72:             }
73:     END { printf "%s\t%s\n", elozo,oszeg }'
74:             | grep -v "root" > $TMP5
75: if [ $D -eq 0 ]    # -d kapcsoló nélkül
76: then
```

```
77:         cat $TMP5
78:     else                      # -d kapcsolóval
79:         paste $TMP4 $TMP3 | grep -v "root" | sort > $TMP6
80:         while read sor
81:             do
82:                 csoporthev=`echo $sor | awk '{print $1}'` 
83:                 echo $sor
84:                 cat $TMP6 | grep $csoporthev | \
85:                     awk '{printf "\t%s\t%s\t%s\n", $1,$3,$4}' 
86:             done < $TMP5
87:         fi
88:     }
89:
90: # FÓPROGRAM
91:
92: # Kapcsolók hiányában tájékoztató üzenet küldése
93:
94: if [ $# -eq 0 ]
95: then
96:     echo "Felhasználók lemezfoglalásának összesítése"
97:     echo "Használat: $PROGRAMNEV [-du]"
98:     echo "-u : felhasználó szerinti összesítés"
99:     echo "-g : csoport szerinti összesítés"
100:    echo "-d : a könyvtárak neve is megjelenik"
101:    echo "A -u és -g kapcsoló nem használható együtt!"
102:    exit 1
103: fi
104:
105: # Kapcsolók vizsgálata
106:
107: U=0 ; D=0 ; G=0
108: while getopts ":udg" KAPCSOLO
109: do
110:     case $KAPCSOLO in
111:         u) U=1;;
112:         d) D=1;;
113:         g) G=1;;
114:         *) echo "Ismeretlen kapcsoló"
115:             exit 2;;
116:     esac
117: done
118:
119: # -u és -g közül egyet használni kell
120:
121: if [ `expr $U + $G` -eq 0 ]
122: then
123:     echo "A -u és -g kapcsolók közül egyet kötelező megadni!"
124:     exit 3
```

```
125: fi
126:
127: # -u és -g egyidejűleg nem használható
128:
129: if [ `expr $U + $G` -eq 2 ]
130: then
131:   echo "A -u és -g kapcsoló nem használható együtt!"
132:   exit 4
133: fi
134:
135: # Felhasználók valódi neveinek kigyűjtése
136:
137: cat /etc/passwd | awk -v FS=":" \
138: '{if(($3>=500) && ($3<1000)) print $0}' | sort | \
139: awk -v FS=":" '{print $5}' > $TMP1
140:
141: # Befűzött fájlrendszerek kigyűjtése
142:
143: mount | awk '/\dev\hd/ {print $3}' | \
144: grep -v "^\/$" > $TMP2
145:
146: # Helyfoglalások felmérése
147:
148: for fajlrendszer in `cat $TMP2`
149: do
150:   eval "ls -ld $fajlrendszer/*" | grep ^d | awk
151:     '{print $3,$4}' >> $TMP4
152:   du -ks `ls -ld $fajlrendszer | \
153:             grep ^d | awk -v f=$fajlrendszer \
154:               '{print f"/"$9}'` >> $TMP3
155: done
156: # Kimenet előállítása a megfelelő függvényekkel
157:
158: # Felhasználók szerinti bontás
159:
160: if [ $U -eq 1 ]
161: then
162:   felhasznalok
163: fi
164:
165: # Csoportok szerinti bontás
166:
167: if [ $G -eq 1 ]
168: then
169:   csoportok
170: fi
171:
172: exit 0
```

A program csak valamilyen kapcsoló megadásával indítható. Ha a felhasználó parancssori paraméterek nélkül próbálkozik, tájékoztató üzenetet kap a lehetőségekről (94-103. sorok). Összesen négy lehetséges kapcsolókombináció van, mivel a -u és -g (felhasználók és csoportok szerinti bontás) egyidejű használata értelmetlen. A megadott kapcsolókat a 107-117. sorokban gyűjtjük be a getopt parancs segítségével. minden beállított kapcsolónál a neki megfelelő nagybetűs héjváltozó értéke 1-re áll.

Az értelmetlen kombinációkat ezek összeadásával ellenőrizzük (121-133. sorok). Valójában akkor sem történne baj, ha csak a -d kapcsoló lenne megadva, hiszen a megfelelő feldolgozó függvények ekkor el sem indulnak (160-170. sorok). Ugyanakkor ez a hiba teljesen fölöslegessé teszi a főprogramban előállított „nyers” listák létrehozását is. Mivel pedig a fájlrendszer végigpásztázása nagyobb merevlemezek illetve sok felhasználó esetén igen időigényes lehet, egyszerűbb az említett tévedést már a program elején kiszűrni.

A 137-139. sorokban a /etc/passwd fájlból egy átmeneti fájlba gyűjtjük a rendszeren bejegyzett valódi felhasználók bejelentkezési és valódi nevét. A UNIX rendszereken általában számos „virtuális” felhasználó is be van jegyezve. A nagyobb programrendszer általában ilyen nem hús-vér felhasználók nevében futnak.

A virtuális felhasználók számosnájához (UID) egy – rendszerenként változó – folytonos tartományba esnek, amit a keresés során ki kell hagynunk. Linux operációs rendszert feltételezve a kérdéses tartomány a 0-500-as azonosítókat foglalja magában. Vannak egészen magas UID-del rendelkező virtuális felhasználók is, így egy felső korláttal (esetünkben 1000) ezeket is ki kell zárni. Vegyük észre, hogy az átmeneti fájlba a felhasználók betűrendes listája kerül! Ennek a későbbiekben még jelentősége lesz.

A 143-144. sorokban a befűzött fájlrendserek listáját állítjuk elő egy másik átmeneti fájlban. A főkönyvtárat („/”) kihagyjuk (144. sor), hiszen azon felhasználói adatoknak normális esetben nincs helye. (A példában feltételeztem, hogy Linux operációs rendszert és kizárolag IDE merevlemezeket használunk. Ez tükröződik a mount kimenetét feldolgozó awk program címzésében.)

A 148-154. sorokban következik a munka dandárja, vagyis a fájlrendserek végigpásztázása. Ehhez egy for ciklussal haladunk végig a TMP2 átmeneti fájl tartalmán, vagyis a számunkra lényeges fájlrendserek listáján. Valamennyi fájlrendszer gyökerében végrehajtunk egy ls -1d parancsot, amelynek kimenetéből rögtön kiszűrjük az esetlegesen jelenlevő közönséges fájlokat (grep ^d). Az immár csak könyvtárakat tartalmazó listából a tulajdonost és a tulajdonoscsoportot a negyedik átmeneti fájlban helyezzük el. (Később ebből fogjuk majd tudni, melyik könyvtár kinek a tulajdona.)

A 151-153. sorokban egy ugyanilyen listából csak a könyvtárak neveit gyűjtjük ki, majd ezt a listát parancssori paraméterként a `du -ks` parancsnak adjuk át. A `-s` kapcsoló hatására a `du` minden könyvtárról csak összesítést jelenít meg, a `-k` hatására pedig a helyfoglalás mértékegysége a kilobájt lesz. A kapott helyfoglalási lista a harmadik átmeneti fájlba kerül.

Ezzel készen is állunk az összesítés elvégzésére. Ha a felhasználó a `-u` kapcsolót használta, akkor a „felhasznalok”, ha a `-g-t`, akkor a „csoportok” nevű függvény fog lefutni.

Ha felhasználók szerint összesítünk, akkor először a `paste` segítségével egymás mellé másoljuk a negyedik és a harmadik átmeneti fájl oszlopait. Így egy olyan lista keletkezik, amely valamennyi fájlrendszer valamennyi, a gyökérből nyíló könyvtárról tartalmazza a tulajdonost, a tulajdonoscsoportot, a kilobájtból kifejezett méretet és a teljes elérési úttal kiegészített nevet.

A `paste` kimenetét rögtön egy sort kapja meg, amely – mivel minden sorban a felhasználó bejelentkezési neve van elő – név szerint rendezi azt. Ezután egy `awk` program összegzi az azonos névhez tartozó bejegyzéseket, és az eredményeket az ötödik átmeneti fájlba írja. (A két oszlop elválasztására tabulátort (`\t`) használunk.) A rendszergazda esetleges helyfoglalására (például `lost+found` könyvtár) nem vagyunk kíváncsiak, így ezt egy `grep`-pel kiszűrjük a kimenetből. (A `-v` kapcsoló a megadott keresési feltétel tagadása.)

Ha a kapcsolók között nem szerepel a `-d`, akkor nincs más hátra, mint egymás mellé másolni a felhasználói neveket tartalmazó első, és az összesített helyfoglalásokat tároló ötödik átmeneti fájl tartalmát. Mivel minden a felhasználók bejelentkezési neve szerint rendezett, éppen a megfelelő adatok kerülnek egymás mellé! (A `paste` alapértelmezés szerint tabulátorral választja el az összemásolt adatokat, így a lista továbbra is szépen lesz formázva.)

Ha a `-d` kapcsoló is szerepelt, a helyzet kissé bonyolultabb. Ekkor is célszerű megjeleníteni a felhasználónkénti összesített eredményt, de most ki kell íratnunk minden név után a megfelelő könyvtárak neveit is. Ehhez nyilván egy, a neveken végighaladó ciklusra lesz szükség. Az előbbihez teljesen hasonló módon tehát a `paste` segítségével összemásoljuk az első és ötödik átmeneti fájl tartalmát, de a kimenet most nem a képernyőre, hanem egy újabb átmeneti fájlba kerül.

Ennek a sorain megyünk végig egy `while` ciklussal (45-50. sorok), úgy, hogy a ciklusmag szabványos bemenetét az előbb előállított hatodik átmeneti fájlból vesszük. Kiírjuk minden egyes sor tartalmát, kiemeljük belőle a bejelentkezési nevet (47.

sor), majd a negyedik és harmadik átmeneti fájl összemásolásával keletkező listából kikeressük azokat a sorokat, amelyek ezt a nevet tartalmazzák. A kimenetből csak a könyvtárak nevét és méretét (harmadik és negyedik mező) jelenítjük meg egy awk program segítségével.

A felhasználói csoportok szerinti összesítés elkészítése sokban hasonlít az előző-höz. Itt is a harmadik és negyedik átmeneti fájl összeillesztésével kezdődik a munka, de az első mező most a csoport, és nem a tulajdonos nevét fogja tartalmazni (58. sor). Az 59. sorban meghívott sort így most csoportok szerint rendezett listát fog előállítani, amit azután egy, az előző függvényhez teljesen hasonló awk programmal összesítünk.

A kimenet mégint az ötödik átmeneti fájlba kerül, de ha nem volt megadva -d kapcsoló, nincs is más dolgunk, mint kiíratni ennek tartalmát. Ha volt, akkor egy, az előzőhöz hasonló while ciklussal végigmegyünk a csoportok szerint összesített listán, kiírjuk a sorait, kiemeljük belőle a csoport nevét (82. sor), majd kikeressük az ezt tartalmazó bejegyzéseket a harmadik és negyedik átmeneti fájl oszlopait egymás mellett tartalmazó átmeneti fájlból. A kimeneten ismét csak a számunkra lényeges mezőket jelenítjük meg egy awk program segítségével.

A program futása bármikor megszakítható. Ilyenkor a 13. sorban megadott trap parancs gondoskodik az átmeneti fájlok törléséről. Az rm parancs hibacsatornáját azért kell elnyomni, mert egyes átmeneti fájlok a program bizonyos pontjain még nem léteznek, így ilyenkor hibaüzenet keletkezne.

9

Tippek, trükkök

Kiegészítés Bash-felhasználóknak

Bármilyen nyelven fejlesztünk is programot, az igazán jó darabok nem trükkökre, hanem a rendszer szabványos szolgáltatásaira épülnek. A jó program ugyanis nem csak egy rendszeren működik, hanem hordozható. Ugyanakkor nem szeretnék senkit lebeszélni a rendhagyó, illetve nem hordozható megoldások használatáról, hiszen ezek néha sokkal áttekinthetőbb vagy hatékonyabb kódot eredményeznek az adott rendszeren.

A legjobb megoldás valószínűleg az, ha munkánk során mi magunk találjuk meg az arany középutat az „elvetemülten eredeti” és a „kockafejűen szabványos” között. Mindenesetre ebben a fejezet a nem egészen „szögletes” dolgokról esik szó: a Bash egyedi, más parancsértelmezőkben nem feltétlenül használható szolgáltatásait fogjuk néhány egyszerű példán keresztül áttekinteni.

Változó „nem meghatározott” tétele

Nem kell, hogy átírd a múltat...

Többször esett már szó arról, hogy az üres és a nem meghatározott (definiálatlan) változó fogalmilag, sőt néha gyakorlati szempontból sem azonos. Egy változó üres, ha neve szerepel a héj megfelelő belső táblázatában, de értéke üres karakterlánc, viszont nem meghatározott, ha a héjnak a „neve sem ismerős”.

Az üres változóra bármikor hivatkozhatunk érték szerint, a nem meghatározott esetében azonban ez általában formai (szintaktikai) hibának minősül. Amint arról az első fejezetben már volt szó, egy változó akkor válik meghatározottá, amikor először értéket adunk neki. Neve a továbbiakban bent marad az említett belső táblázatban, rendszerint akkor is, ha a tartalmát töröljük.

A Bash parancsértelmező esetében az üres és a nem meghatározott változó fogalmilag kissé (de nem teljesen!) összemosódik. Akkor sem keletkezik formai hiba, ha nem meghatározott változóra hivatkozunk érték szerint, viszont ha töröljük egy változó tartalmát, akkor az nem üressé, hanem nem meghatározottá válik. A „nem meghatározottság” az unset parancsal is előidézhető.

A nem meghatározottság kezelésére szolgáló – az első fejezetben említett – módszerek természetesen továbbra is működnek, amint azt a következő példaprogram is szemlélteti.

9.1. lista

```
1: #!/bin/sh
2: # Változó törlése (nem meghatározottá tétele)
3:
4: PROGRAMNEV=`basename $0`
5:
6: if [ $# -eq 0 ]
7: then
8:   echo "Használat: $PROGRAMNEV [u|d]"
9:   exit 1
10: fi
11:
12: # Kezdeti érték
13:
14: valtozo=3
15: echo "A változó eredeti értéke: $valtozo"
16:
17: # Kétféle törlés
18:
19: case $1 in
20: "u") unset valtozo;; # Ugyanaz, mint valtozo= vagy valtozo=""
21: "d") valtozo=" ";;
22: *) echo "Ismeretlen kapcsoló!"
23:   exit 2;;
24: esac
25:
26: ertek=${valtozo:-Nem meghatározott}
27:
28: echo "A változó törlés utáni értéke: $ertek"
29: exit 0
```

Ha az u kapcsolót használjuk, a 20. sorban a változó nem meghatározottá válik, így a 26. sorban az „ertek” nevű változóba a „Nem meghatározott” szöveg kerül. A d kapcsoló hatására ellenben egy szóköz kerül a változóba, így az továbbra is meghatározott marad.

Parancsbehelyettesítések egymásba ágyazása

A UNIX harmadik bugyra...

A könyvben eddig bemutatott programokban számtalan példát láthattunk a parancsbehelyettesítés használatára. Olyan helyzet is sokszor merült fel, amikor az egyik parancsbehelyettesítés eredményét közvetlenül utána egy másik használta fel. Ilyenkor a gyakorlottabb (bátrabb) felhasználó rögtön arra gondol, vajon nem lehetne-e ezeket egymásba ágyazni, s így megspórolni azt a változót, amelyben a részeredményt tároltuk.

Természetesen lehet, DE... Nézzünk talán egy kifejezetten egyszerű példát. Négy számon akarunk az expr segítségével elvégezni egy „összetett” (értsd: zárójelek is vannak benne) matematikai műveletet. Például kettő összegét megszorozzuk a harmadikkal és az eredményt elosztjuk a negyedikkel. Az expr egyszerre csak két számot hajlandó kezelni, tehát ehhez legalább háromszor kell meghívni, ha csak nem döntünk a parancsbehelyettesítések egymásba ágyazása mellett. A következő példaprogramban szintenként mutatom be a „beágyazás” menetét.

9.2. lista

```

1: #!/bin/sh
2:  # Parancsbehelyettesítések egymásba ágyazása
3:  # Egy bonyolult számítás: ((2+3)*6)/10
4:
5:  szam1=2 ; szam2=3 ; szam3=6 ; szam4=10
6:
7:  eredmeny=`expr $szam1 + $szam2` 
8:  eredmeny=`expr `expr $szam1 + $szam2` \* $szam3` 
9:  eredmeny=`expr `expr `expr $szam1 + $szam2` \` \* $szam3` / $szam4` 
10:
11: eredmeny=$(expr $szam1 + $szam2)
12: eredmeny=$(expr $(expr $szam1 + $szam2) \* $szam3)
13: eredmeny=$(expr $(expr $(expr $szam1 + $szam2)
14:                                \* $szam3) / $szam4)
15: echo $eredmeny

```

A program ugyan működik, de ami a 9. sor kinézetét illeti, arra a legvisszafogottabb kifejezés talán az lehet, hogy „érdekes”. A gondot nyilvánvalóan az okozza, hogy a parancsbehelyettesítés elejét és végét ugyanaz a karakter jelzi. És bár

a szimmetria szép doleg, ebben a formában sajnos az áttekinthetőség, illetve kezelhetőség rovására megy, hiszen a parancsbehelyettesítésen belül megjelenő egyszeres idézőjeleket megfelelő mértékű védelemmel kell ellátnunk. Ellenkező esetben ugyanis a héj azt fogja hinni, hogy vége van a behelyettesítendő műveletsornak.

Jó közelítéssel azt mondhatjuk, hogy minden egyes beágyazási szint egy-egy újabb „garnitúra” \ karakter megjelenését vonja maga után. Ez pedig azt jelenti, hogy a harmadik szinten már a védelem védelmét kell védenünk a héj értelmező mechanizmusától. Az eredmény fent látható.

Természetesen minden gondunk egy csapásra megoldódna, ha megszűnne a jelölés szimmetriája. Valószínűleg erre az egyszerű dologra jöttek rá a Bash parancsér telmező fejlesztői, amikor a parancsbehelyettesítés másik jelölésére vezették a

`$ (parancssor)`

szerkezetet. A példaprogram 11-13. soraiban ezzel a jelöléssel is ugyanazt valósítottam meg, mint az előző három sorban. Az eredmény valószínűleg önmagáért beszél. Csak a szorzójelet kellett levédenünk, de azt amúgy is kellett volna. Az új megoldás tehát egyszerű, áttekinthető, de ha rendszerünkön nincs Bash, akkor nem működik.

A let parancs

Felsőbb matematika...

Az előző részben az expr hiányos képességei tulajdonképpen csupán jó ürügyet adtak a parancsbehelyettesítések egymásba ágyazásának bemutatására. Viszont már a matematikánál tartunk, érdemes megemlíteni a let parancsot.

Ez tisztán matematikai szempontból ugyanazokkal a képességekkel bír, mint az expr, de kettőnél több tényezőt is kezel, sőt ismeri a zárójeleket is. Sajnos semmi sem lehet tökéletes, mint az a következő példából kiderül.

9.3. lista

```

1: #!/bin/sh
2: # A let parancs használata
3:
4: szam1=2 ; szam2=3 ; szam3=6 ; szam4=30 ; szam5=10
5:
6: # Egyszeres mélységű zárójelezés és let parancs
7:

```

```
8: let eredmeny=($szam1+$szam2)*$szam3/$szam5
9: echo $eredmeny
10:
11: # Kétszeres mélységű zárójelezés és let parancs
12:
13: let eredmeny=(\($szam1+$szam2\)*$szam3+$szam4)/$szam5
14: echo $eredmeny
15:
16: # Kétszeres mélységű zárójelezés és a $(...) szerkezet
17:
18: eredmeny=$(( ($szam1+$szam2)*$szam3+$szam4)/$szam5 ))
19: echo $eredmeny
```

Az egyszeres mélységű zárójelezés akadályát a let még simán veszi, sőt kivételese a szorzójelet sem kellett levédenünk. (További előny, hogy az értékadáshoz nem kell parancsbehelyettesítést használnunk.) Kétszeres zárójelezésnél viszont már gondok vannak, mivel a kettős kerek zárójel a Bash számára értelmes, csak éppen egészen más jelent, mit amire használni akarjuk (lásd később).

Mulatságos módon azonban itt is érvényesül a „kutyaharapást szőrével” elv: a kétszeres (és többszörös) mélységű zárójelezés nem mással oldható meg, mint magával a dupla zárójelezéssel. A Bash ugyanis a

```
$(( aritmetikai kifejezés ))
```

szerkezetet használja az aritmetikai kifejezések jelzésére és kiértékelésére. Amint az a 18. sorban látható, működése gyakorlatilag teljesen azonos a let parancséval, viszont az áttekinthetőség szempontjából ez tűnik a tökéletes megoldásnak.

Közvetett változóhasználat

Tudom, hogy tudod, hogy tudom...

Ha van egy héjváltozónk, abban tetszőleges karakterláncot elhelyezhetünk. Ha hozzá akarunk férfi ehhez a tartalomhoz, a változó neve elő csupán egy \$ jelet kell írni. De mi van akkor, ha van egy olyan változónk, amely egy másik változó nevét tartalmazza, és mi az utóbbi tartalmához akarunk hozzáérni. A lehetőségeket a következő példaprogram mutatja be.

9.4. lista

```

1: #!/bin/sh
2: # Közvetett változóhasználat
3:
4: valtozo=3
5: valtozo_neve="valtozo"
6:
7: # Ez a cél...
8: echo $valtozo
9:
10: # 1. Megoldás: parancssor összeállítása és végrehajtása
11: parancs="echo $"$valtozo_neve"
12: eval "$parancs"
13:
14: # 2. Megoldás: Ugyanez parancsbehelyettesítéssel
15: eval `echo "echo $"$valtozo_neve`'
16:
17: # 3. Megoldás: Egyetlen literális $ jellel
18: eval echo \$\$valtozo_neve
19:
20: # 4. Megoldás: Csak Bash-ben
21: echo ${!valtozo_neve}

```

Az első három megoldás bármely rendszeren működik, a negyediket azonban ismét csak a Bash ismeri.

Az üres parancs

Édes semmittevés...

A „:” (SEMMI, NOP) parancs a true hasonszavaként használható. Amint a második fejezetben láttuk, ennek leginkább a végtelen ciklusok szervezésénél vehetjük hasznát, mivel visszatérési értéke mindig 0, vagyis logikai IGAZ.

```

while :
do
... parancsok ...
done

```

A minden igaz visszatérési értéknek azonban vannak egyéb – mondhatni szédítő – távlatokat nyitó – felhasználási lehetőségei is. Azt például, hogy egy if-es szerkezetnek kizárálag hamis ága legyen, így oldhatjuk meg:

```
if feltétel  
then :  
else  
    parancsok  
fi
```

Ennek így nincs túl sok értelme, hiszen a feltétel tagadásával körülbelül ugyanitt tartanánk, de a fenti megoldás formailag helyes és működőképes.

A lehetséges felhasználások körét bővítendő lássuk mit tehetünk, ha egy fájlnak csak a tartalmát akarjuk törölni, magát a bejegyzést azonban nem. Íme a megoldás:

```
: > fájlnév
```

Ez valójában megint ugyanaz, mint a

```
cat /dev/null > fájlnév
```

de tény, hogy rövidebb.

Végezetül a „:” parancs segítségével azt is elérhetjük, hogy egy fájl utolsó módosításának dátuma a jelenlegi idő legyen, anélkül, hogy ténylegesen módosítanánk benne valamit:

```
: >> fájlnév
```

Természetesen erre is létezik másik UNIX parancs, olyannyira, hogy több példaprogramban már használtuk is:

```
touch fájlnév
```

C stílusú megoldások

Aki Á-t mond...

A héjprogramok két legfőbb furcsasága a ciklusok szervezésének, illetve a döntési szerkezetek megadásának módja volt. Bár, mint láthattuk, a test parancsban nincs semmi különleges, kissé furcsán érezzük magunkat, amikor mondjuk két szám összehasonlításánál egyenlőségiel helyett „-eq”-t kell írnunk. Hasonlóan fura a for ciklus működése, bár ez a forma bizonyos helyzetekben (például parancssori kapcsolók vagy paraméterek kezelése) kifejezetten hasznos lehet.

Aki Bash parancsértelmezőt használ, eltérhet a UNIX hagyományaitól: megadhat a C nyelvből ismert „numerikus üzemmódban működő” for ciklust, és a logikai összehasonlításoknál is használhatja a megszokott relációs jeleket. Mindezt viszont csak a program hordozhatóságának rovására teheti.

A C utasításforma jelzésére ciklusszervezésnél a kettős kerek zárójeleket, logikai műveleteknél pedig a kettő szögletes zárójeleket kell használnunk. A következő két példaprogram feltehetőleg önmagáért beszél.

9.5. lista

```

1: #!/bin/sh
2: # C stílusú ciklusok a Bash-ben
3:
4: # "Numerikus" for ciklus
5:
6: for((i=1;i<=10;i++))
7: do
8:   echo -n $i" "
9: done
10: echo
11:
12: # C nyelvre emlékeztető while ciklus
13:
14: i=1
15: while ((i<=10))
16: do
17:   echo -n $i" "
18:   let i=$i+1
19: done
20: echo

```

9.6. lista

```

1:#!/bin/sh
2:# Matematikai feltételek megfogalmazása szabályos relációs jelekkel
3:
4:PROGRAMNEV=`basename $0`
5:
6:if [[ $#<2 ]]
7:then
8:  echo "Használat: $PROGRAMNEV <szám1> <szám2>"
9:  exit 1
10:fi
11:

```

```
12: if [[ $1>$2 ]]
13: then
14:   echo $1\>$2
15: elif [[ $1<$2 ]]
16: then
17:   echo $1\<$2
18: else
19:   echo $1\=$2
20: fi
21:
22: exit 0
```

Az első program egyszerűen kiírja a számokat egytől tízig, de ehhez két C stílusú ciklust használ. A második összehasonlítja a parancssori paraméterként megadott két számot, de a döntési szerkezetben szabályos relációs jeleket alkalmaz.

Gyermekhéj és névtelen függvény használata

Egysége tömörülve...

Több példaprogramunkban is előfordult, hogy egy teljes ciklusmag bemenetét vagy kimenetét irányítottuk át fájlba egyetlen parancssal. Ugyanakkor olyan helyzet is akadt, amikor például több – esetleg egymás után következő – parancs kimenetét kellett volna hasonlóan kezelni, de ezt csak úgy tudtuk megoldani, hogy egyenként megadtuk a megfelelő átirányításokat. Az ilyen esetek kezelésére vezették be a Bash-ben a gyermekhéj és a névtelen függvény fogalmát.

Alapvetően mindenkorra való, hogy programunk összetartozó részeit egyetlen egységgé fogjuk össze velük. Az ilyen programblokknak önálló be- és kimenete lehet, amit a ciklusokhoz hasonlóan egyetlen parancssal irányíthatunk át. Gyermekhéj használatakor a kerek, névtelen függvény meghatározásakor pedig a kapcsos zárójelekkel kell jelezni a blokk kezdetét és végét.

Kerek zárójelezésnél – az elnevezésnek megfelelően – egy új parancsértelmező indul el, és a megadott programblokkot ez hajtja végre. Kapcsos zárójelekkel való csoporthozzásnál ugyanaz a héj végzi a teljes munkát.

A két megoldás közti leglényegesebb különbség ennek megfelelően a változók használatában van. A gyermekhéj – mint minden héj – önálló változókészettel rendelkezik. Öröklí ugyan az őt indító szülőprogram változóit, de azoknak csak a má-

solatával rendelkezik. Így tartalmukat módosítani is csak helyileg tudja: a módosítás a szülőprogram változóinak tartalmát nem érinti, de a gyermekhét a saját másolatával természetesen azt kezd, amit akar.

A névtelen függvény ugyanakkor ugyanazokat a változókat „látja”, mint a főprogram, és az általa végrehajtott módosítások is kihatnak a főprogramra, amint az a következő példaprogramból látható.

9.7. lista

```

1: #!/bin/sh
2: # Gyermekhét és névtelen függvény
3:
4: szam=3
5: szoveg="Hello világ!"
6:
7: # Gyermekhét indítása (saját változókészettel)
8:
9: (
10:    szam=5
11:    szoveg="Hello world!"
12:    belso_valtozo="Belső"
13:    echo $szam
14:    echo $szoveg
15: )
16: echo $szam
17: echo $szoveg
18: echo "A belső változó tartalma: ${belso_valtozo:-Nem meghatározott}"
19:
20: # Névtelen függvény indítása
21:
22: {
23:    szam=5
24:    szoveg="Hello world!"
25:    echo $szam
26:    echo $szoveg
27:    belso_valtozo="Belső"
28: }
29: echo $szam
30: echo $szoveg
31: echo "A belső változó tartalma: ${belso_valtozo}

```

A gyermekhét a 10-11. sorokban átírja a főprogram változóinak másolatát, de a 16-17. sorokban kiadott echo parancsok még mindig a régi értéket jelenítik meg.

A gyermekhéjban meghatározott belso_valtozo tartalmáról a főprogram nem tud, az számára nem meghatározott. A névtelen függvénynél ugyanakkor az „áthalás” oda-vissza tökéletes.

Tömbök használata

Rejtett dimenziók...

Az Olvasó fejében valószínűleg több példa kapcsán is felmerült már, milyen hasznos lenne, ha a héj támogatná valamilyen módon a tömbök használatát. Ez szerencsére a Bash fejlesztőinek is eszébe jutott, így ez a héj ismeri a tömb fogalmát, bár csak az egydimenziósét.

Ahogy a változókat, a tömböket sem kell bevezetnünk. Létrejönnek, amint az első elemüknek értéket adunk. A tárfoglalásra ténylegesen csak ilyenkor kerül sor, így a Bash tömbjeinek mérete elvben korlátlan. Az elemek indexelésére – akárcsak más nyelvekben – itt is egész számokat használhatunk, amelyeket a tömb neve után szögletes zárójelek között kell megadnunk:

```
tömb_neve[index értéke]
```

Az indexelés nullától indul, és indexként nemcsak számok, hanem olyan aritmetikai kifejezések is megadhatók, amelyek értéke egész. A tömbelemekre érték szerint a következőképpen hivatkozhatunk:

```
 ${tömb_neve[index értéke]}
```

Akárcsak a parancssori paraméterekre, a tömb elemire is hivatkozhatunk egyben, ha indexszám helyett a * vagy a @ karaktert adjuk meg. (Ezek jelentése is azonos a parancssori paraméterknél megszokottal.)

A következő egyszerű program a tömbök kezelésének fortélyait mutatja be.

9.8. lista

```
1: #!/bin/sh
2: # Tömbök használata a Bash héjban
3:
4: # Tömbelemek feltöltése
5:
6: for i in `seq 2 2 20`
7: do
```

```
8:      tomb[$i/2]=$i
9:      done
10:
11:     # Kiírás egyenként
12:
13:     for((k=1;k<=10;k++))
14:     {
15:         echo ${tomb[$k]}
16:     }
17:
18:     # Kiírás egyben
19:
20:     echo ${tomb[*]}
```

Függelékek

I. függelék

UNIX összefoglaló

A UNIX használata során a felhasználók egyik legnagyobb gondja az, hogy nem tudják, milyen feladatra léteznek segédprogramok, és mit kell maguknak elkészíteniük. Ezért ebben a függelékben a legfontosabb UNIX parancsokat tekintem át, természetesen a teljesség igénye nélkül. Ez a lista valóban csak arra jó, hogy megtudjuk, mit nem tudunk. A részletes leírásokért olvassuk el a megfelelő parancs leírását.

at

Időzített parancsvégrehajtás.

awk

Teljes szövegfeldolgozó nyelv, melynek aritmetikai képességei sem elhanyagolhatók.

basename

Egy fájl nevét adja vissza az elérési út nélkül.

bc

Tetszőleges pontosságú számológép.

cat

Szövegfájlok tartalmát jeleníti meg.

cd

Munkakönyvtár váltása. A paraméterek nélküli cd parancs visszavisz a saját könyvtárunkba. A héjprogramban kiadott könyvtárváltások csak a programra érvényesek, arra a héjra, ahonnan a programot futtatjuk, már nem.

clear

Törli a képernyőt.

date

A dátumot adja vissza vagy állítja be. A kiírás formátuma kapcsolókkal szabályozható.

df

Az egyes fájlrendszer foglaltságáról ad összefoglaló táblázatot.

du

A lemezen lefoglalt területről ad felvilágosítást.

echo

A megadott szöveget a szabványos kimenetre (alapértelmezés szerint a képernyőre) küldi. A szöveget újsor karakter követi, hacsak meg nem adtuk a -n kapcsolót.

egrep

Működése azonos a grep parancséval, de többféle, a szabályos kifejezésekben használható elemet ismer.

exit

Megszakítja a program futását. Megadhatunk utána egy 0-255 tartományba eső egész számot is, ami a program visszatérési értéke lesz. Ha nem adunk meg semmit, a program az utolsó parancs visszatérési értékével (\$?) tér vissza.

fgrep

Literális karakterlánc keresésére szolgáló parancs. Nagyobb testvérével, a grep-pel ellentétben nem ismeri a szabályos kifejezéseket, viszont cserébe gyorsabban működik.

file

Fájl típusát lehet vele meghatározni.

find

Általános fájlkereső. Képes név, típus és számos egyéb jellemző alapján keresni. A megtalált fájlokon azonnal végre is hajthat valamelyen műveletet.

grep

Szabályos kifejezésekkel vezérelhető általános keresőprogram (lásd még egrep és fgrep).

head

Egy szövegfájl első sorait megjeleníti meg. Alapértelmezés szerint az első tíz sort írja ki, de más érték is megadható.

kill

Jelet küldhetünk vele egy általunk indított folyamatnak. A kiküldeni kívánt jel első paraméterként névvel és számmal egyaránt megadható. A második paraméter

a „megcélzott” program folyamatazonosítója. Ha csak folyamatazonosítót adunk meg, az alapértelmezett 15-ös jelet (SIGTERM) küldi el a program. Ha egy futó folyamatot azonnal, feltétel nélkül meg akarunk állítani, a 9-es (SIGKILL) jelet használjuk.

mkdir

Könyvtár létrehozása

mount

Segítségével új fájlrendszert fűzhetünk be, vagy leválaszthatunk egy befűzöttet. Erré általában csak a rendszergazda (root) feljogosult, viszont paraméterek nélkül bármely felhasználó kiadhatja ezt a parancsot. Ilyenkor csak összesítést ad a befűzött fájlrendszerekről.

nl

Beszámolza egy fájl sorait, úgy, hogy az üres sorokat nem számolja.

od

Tetszőleges bemeneti adatokat jelenít meg különböző formátumokban (például nyolcas, tízes, tizenhatos számrendszerben). Képes a nem nyomtatható karakterek megjelenítésére is.

paste

Függőlegesen egymás mellé másolja a megadott fájlok tartalmát.

pwd

A pillanatnyi könyvtár nevét és teljes elérési útját megjeleníti meg.

ps

A futó folyamatokról ad felvilágosítást.

read

A szabványos bemenetről egy sort olvas be a megadott változóba.

rev

Szöveg sorait írja ki visszafelé. (A jobb szélső karakterrel kezdődnek a kimenet sorai.)

rm

Alapvető feladata fájlok törlése, de a -r kapcsoló hatására az alkönyvtárakat is törli.

rmdir

Könyvtár törlésére szolgál.

seq

Számsorozatot állít elő adott lépésközzel. (Általában a for ciklusok listájának előállítására használatos.)

shift

A héjprogram parancssori paramétereit egy pozícióval balra mozgatja. Az első elem ilyenkor elvész. Megadható utána egy egész szám is. Ilyenkor ennyivel mozdítja balra a paramétereket.

sleep

Adott ideig várakozik. A várakozás hosszát alapállapotban másodpercben várja, de ez megadható neki percben, órában, sőt napban is.

sort

Alapértelmezés szerint ábécésorrendbe rendezi a bemenetét, szükség esetén azonban képes szám szerinti (numerikus) rendezésre is.

tac

Szöveg sorait írja ki fordított sorrendben. (A legutolsó sor lesz a legelső.)

tail

Egy szövegfájl utolsó sorait megjeleníti meg. Alapértelmezés szerint az utolsó tíz sort írja ki, de más érték is megadható.

tee

„Elágazást” hozhatunk vele létre egy feldolgozási sorban. Bemenetét elismétli a szabványos kimenetén, miközben a megadott nevű fájlba is kiírja.

time

Egy parancs futásidejét mérhetjük le vele.

touch

A megadott fájl utolsó módosítási idejét a pillanatnyi időre állítja, anélkül, hogy ténylegesen végezne rajta valamilyen műveletet. Ha a fájl nem létezik, létrehozza, de üresen hagyja.

tr

Karakterhelyettesítő program. Két, szigorúan azonos hosszúságú karakterláncot kell megadnunk neki. A bemenetére érkező szövegben minden, az első karakterlánc n-edik helyén álló betűt a második lista n-edik helyén álló betűre cserél. A nem nyomtatható karaktereket is kezeli.

trap

Jelek elfogására és kezelésére használható.

uname

Az operációs rendszer típusát (Linux, AIX stb.) lehet vele lekérdezni.

uniq

Törli a bemenetből az ismétlődő sorokat. Csak rendezett bemenetet képes feldolgozni, ezért általában a sort parancsal együtt használjuk.

wait

Megvárja, amíg lefutnak a háttérben indított gyermekfolyamatok. Ha folyamatazonosítót is megadunk neki, csak a megfelelő folyamat befejeződését várja meg.

wc

Sorokat, szavakat és karaktereket számlál.

who

A bejelentkezett felhasználók listáját jeleníti meg.

II. Függelék

Hol a hiba?

A műszerek, számítástechnikai eszközök használati utasításának végén általában van egy olyan rész, ahol hibajelenségeket és azok legvalószínűbb okait sorolják fel. (Angolul ezt a részt hívják „Troubleshooting”-nak.) Ez a függelék ugyanezt a szerepet kívánja betölteni a héjprogramokkal, illetve a héjprogramozással kapcsolatban.

Alapvető hibák

Az általunk írt program egyáltalán nem indul el. Az Enter leütése után a "Command not found" üzenetet kapjuk a rendszertől.

1. A pillanatnyi könyvtár (.) nem szerepel a PATH környezeti változó tartalmában. Próbáljuk ki a ./programnév <ENTER> parancsot.
2. A héjprogram első sorában a „#!” után elírtuk a parancsértelmező nevét vagy elérési útját.

Értékkadásnál a "Command not found" üzenetet kapjuk.

Az egyenlőségjel valamelyik oldalán szóköz van.

A program futtatásakor az "Unexpected end of file" üzenetet kapjuk.

Valahol lemaradt egy záró idézőjel. Hogy hol, illetve hogy egyszeres vagy kétszeres, azt ebből sajnos nem lehet tudni. Néha meglehetősen nehéz az ilyen hiba felderítése.

A parancsértelmező olyan helyen jelez hibát, ahol formalag látszólag minden rendben van.

1. Nem meghatározott változóra hivatkoztunk érték szerint.
2. A program egy korábbi pontján sortörést (sorvégi "\" karakter) használtunk, de maradt utána egy szóköz.
3. Lemaradt egy korábbi parancsbehelyettesítés záró idézőjele.

A test és az if hibái

Egy test parancs érthetetlen üzenettel válaszol (például "unary operator expected").

1. Ha a test kiírása helyett szögletes zárójeleket használtunk, akkor lemaradt a nyitó vagy csukó zárójel mellől a kötelező szóköz.
2. A logikai kifejezésben egy változó tartalmát használtuk, de az érték szerinti hivatkozásból kifejtettük a \$ jelet.
3. Programszervezési hiba miatt nem meghatározott változával akartunk valamilyen logikai műveletet végezni (pl. összehasonlítottuk egy másik változával).

Egy logikai elágazásnál (if-es szerkezet) a parancsértelemező formai hibát jelez, de a fentiek alapján biztosak vagyunk benne, nem a test parancsal van gond.

1. Lefelejtettük a „then” kulcsszót az igaz ág elől.
2. A „then” kulcsszót pontosvesszőnek vagy újsor karakternek kell elválasztania a logikai feltételtől. (Legjobb, ha minden új sorba írjuk.)
3. Lemaradt a szerkezetet záró „fi” kulcsszó.
4. Lemaradt az „elif” után is kötelező „then” kulcsszó.

Matematikai hibák

Az expr nem hajlandó elvégezni egy matematikai műveletet, hibaüzenettel válaszol.

1. Az elvégzendő művelet valamelyik tagja nem egész szám. Az expr a tizedespontot is tartalmazó számokat szövegnek tekinti, és akként is kezeli.
2. Egybeírtuk a műveleti jelet és az egyik tényezőt. A műveleti jelet mindenkorral szóköznek kell határolnia.
3. A művelet egyik vagy mindenkorral tényezője érték szerinti hivatkozás, de az egyik változó üres.
4. Lemaradt az érték szerinti hivatkozásból a \$ jel.
5. Olyan műveleti jelet használtunk, amely a héj számára különleges karakter (például „*”). Az adott karaktert literálissá kell tenni.

A let parancs megmakacsolja magát, és formai hibára panaszkodik.

Ha egyszeresnél nagyobb mélységű zárójelezést használunk, a belső zárójeleket már védeni kell az értelmezés elől, mivel a kettős kerek zárójel egyes helyeken (például Bash) önálló jelentéssel bír.

Kíratással kapcsolatos hibák

Az echo furcsán viselkedik, semmit nem ír ki, hibaüzenetet küld, vagy nem egészben az a szöveg jelenik meg a képernyőn, amit megadtunk.

1. A kiírandó szöveg olyan betűkombinációt tartalmaz, amit az echo kapcsoló-ként értelmez (például „-e”). Ilyenkor a helyzettől függően a kérdéses karaktert le kell védeni, több darabra kell töri a kiírandó szöveget, vagy meg kell adni egy üres karakterláncot (" ") a kiírandó szöveg előtt (echo " \" -e").
2. Nem meghatározott változó tartalmát akarjuk kiíratni.
3. Olyan szöveget (változó tartalmát) akarunk kiíratni, amely a héj számára értelmes karaktert tartalmaz (például „*”). A karaktert literálissá kell tenni („*”), illetve a változóra való érték szerinti hivatkozást kettős idézőjelek közé kell zárni (echo "\$változó").
4. A „gyári” echo helyett a héj saját belső echo parancsa lép működésbe. Ki kell írni a külső echo program teljes elérési útját.
5. Olyan kapcsolót használunk, amit az echo a jelenlegi beállításokkal nem kezel. Ilyenkor a megoldás általában egy környezeti változó értékének megfelelő beállítása. Az viszont, hogy melyik ez a változó és mit kell tartalmaznia, csak az echo leírásából derül ki.

A hibacsatornát a kimenettel együtt a /dev/null-ba irányítottuk, a hibaüzenetek mégis megjelennek a képernyőn. A kimenet ugyanakkor valóban eltűnik.

A „2>&1” átirányításnak utolsóként kell szerepelnie a parancssorban.

Beágyazott dokumentum kiíratásakor nem pontosan a megadott szöveg jelenik meg a képernyőn.

A szöveg a héj számára értelmezhető karaktereket tartalmaz. Ezek értelmezését úgy lehet megtiltani, hogy a beágyazott dokumentum végét jelző szöveget kettős idézőjelek közé zárjuk.

Nyűgös parancsértelmezők

Egy névvel rendelkező vagy névtelen függvény meghatározásában formai hiba van.

Egyes parancsértelmezők nem szeretik, ha a kapcsos zárójellel azonos sorban kezdődik az első parancs. Néha a blokk végét jelző zárójelre is hasonló megkötés érvényes.

Beágyazott dokumentum kiíratásakor formai hibát kapunk, vagy eltűnnek a sor eleji szóközök.

Egyes parancsértelmezők nem szeretik, ha a beágyazott dokumentum sorai nem a legelső pozíión kezdődnek. Ilyenkor általában segít, ha a sort egy „\” karakterrel kezdjük, amely a kiíratáskor természetesen nem fog megjelenni. Ha mégsem, akkor nincs megoldás.

X-akták

Egy változó KEZDETI értékét parancsbehelyettesítéssel határozzuk meg. A program formailag teljesen helyes, de amikor ki akarjuk íratni a változó tartalmát, mégis hibaüzenetet kapunk.

A változó ilyen esetben csak akkor jön létre, ha a parancsbehelyettesítésben szereplő műveletsornak VAN kimenete. Ha nincs, a program formailag helyes ugyan, az adott változó azonban nem meghatározott marad. Mivel ez csak az első érték szerinti hivatkozásnál fog kiderülni, általában teljesen máshol kapjuk a hibaüzenetet, mint ahol a tényleges hiba van. Ilyenkor a kezdeti értékadást máshogyan kell megoldani.

Az awk „elhatározta”, hogy a számára parancssorban megadott programot megnyitandó fájlként fogja értelmezni. Mivel pedig ilyen nevű fájlt nem talál (miért is találna?), "File not found" üzenettel leáll. Akár hogy vizsgáljuk, a kérdéses sor formailag helyes, sőt megeshet, hogy máshol, tökéletesen működik.

A legvalószínűbb ok, hogy a -v kapcsoló után érték szerint hivatkozunk egy olyan héjváltozóra, amely szóközt is tartalmaz. Ilyenkor az awk az első szóköz előtti részt tekinti a változó értékének, a többöt programnak, a valódi programot pedig megnyitandó fájlnak. Innen az üzenet. A megoldás csupán annyi, hogy az érték szerinti hivatkozást kettős idézőjelek közé tessük, ez ugyanis egyben tartja a változó tartalmát, bármi legyen is benne.

Programunk időnként jól, néha viszont rosszul működik, annak ellenére, hogy józan emberi számítás szerint mindenkor ugyanazt az eredményt kellene kapnunk.

Minden valószínűség szerint sikerült valahol egy hosszabb feldolgozási láncnak ugyanazt a fájlt megadni bemenetként és kimenetként.

Programunkat megszálta a Gonosz. Közelebbről meg nem határozható, paranormális jelenséget produkál. Bár formailag tökéletesnek tűnik, mindig pontosan ugyanott, pontosan ugyanazt a tökéletesen értelmetlen dolgot műveli. Bevetettünk már mindenféle nyomkövetési trükköt, eljártuk az esőtáncot, kikértük az asztrológus szakorvos véleményét, a mellékhatalások tekintetében elolvastuk gyógyszerészünket, teliholdnál éjjelkor feketé kakas vérét vettük. Scully ügynök szülési szabadságon, Muldernek csak az üzenetrögzítője válaszol. Matt...

Bár az igazság alapvetően odaát van, megeshet, hogy a szövegszerkesztés hevében valamilyen nem nyomtatható karakter keveredett a kódszövegbe. Erre pedig egyes parancsértelezők egészen meglepő lépésekre szánják el magukat. A hiba oka és helye – természetéből adódóan – nem látható, így meglehetősen nehéz orvosolni. Mindazonáltal komoly segítség lehet, hogy az od parancs a nem nyomtatható karaktereket is képes megjeleníteni „látható formában”.

Tárgymutató

Tárgymutató

- | | | | |
|-------------------|---------------|---------------------------|---|
| # karakter | 4 | „~” karakter | 86 |
| \$ jel | 5, 51, 80, 83 | „--” | 88 |
| \$# belső változó | 27, 101 | „-” karakter | 7, 87 |
| \$# szimbólum | 107 | „++” | 88 |
| \$i jelölés | 79 | „=” karakter | 7 |
| % jel | 91 | „man grep” parancs | 140 |
| & műveleti jel | 14 | „regexp” | 39 |
| && parancs | 23 | + jel | 68 |
| * karakter | 50, 51 | < műveleti jel | 10 |
| * operátor | 60 | > műveleti jel | 10, 89 |
| / karakter | 56 | >> műveleti jel | 10, 89 |
| \ karakter | 16, 71, 85 | 0 paraméter | 8 |
| ^ jel | 52 | A | |
| műveleti jel | 11 | -a kapcsoló | 12 |
| parancs | 23 | awk | 41, 70, 77, 145, 150,
152, 173, 179, 181 |
| „!~” kombináció | 86 | belőváltozó | 81 |
| „#!” | 3 | kiíratás (<i>print</i>) | 79 |
| „&” műveleti jel | 61 | mező | 78 |
| „:” parancs | 206 | | |
| „?” karakter | 7, 106 | | |
| „\n” szimbólum | 86 | | |

*mezőelválasztó
karakter* 82

változó 80

awk printf függvény 138
awk program 110, 140, 142

D

DEBUG jel 119
digit változó 91
du 191

B

basename 101
Bash 201
Bash parancsértelmező 7, 20
beágyazott dokumentum 18
BEGIN blokk 81, 83, 88
Bourne héj 3
break 32, 165

C, Cs

case szerkezet 26, 102
cat 9, 13, 161
cd 26
chmod 3
CTRL+C billentyűk 118
CTRL+D billentyűk 10
csővezeték (pipe) 11

E

echo 112, 162, 210
egyszeres idézőjel 48, 50
elif kulcsszó 23
END blokk 83, 91, 93
eval 18, 174
EXIT 144
EXIT jel 119, 122
exit 5, 32, 35, 118,
122
expr 15, 31, 155, 159, 179,
204

F

-f kapcsoló 56, 103
file segédprogram 108
find program 125
folyamataazonosító (PID) 114
for 28, 30, 31, 152,
207

FS belső változó 82
függvény 33

if 23, 86
INT jel 120

G

g kapcsoló 60, 61
getline 95
getopts 106
GNU awk 82
GNU sed 64, 70
grep 41, 56, 70, 85, 105, 142,
156, 196

J

jelentésmódosító jel 44
(...) 45
* 44
 $\{x,\}$ 45
 $\{x,y\}$ 45
 $\{x\}$ 45
| 46
+ 44

H

héj 47
help 171
html2text segédprogram
180, 185
HUP jel 120

K

kétszeres idézőjel 48
kill 120, 189

L

let 204
ls 3, 12, 127, 191

idézőjel 8
„visszafelé hajló”
egyszeres idézőjel 8, 48
egyszeres idézőjel 8
kettős idézőjel 8

M

matematikai műveletek 84
max változó 91

mezőelválasztó karakter (FS)

98

R

N

-n kapcsoló 64, 112

NR belső változó 81

num változó 91

rajzol függvény 155

read 109, 171

rekordelválasztó (RS) 98

return 35

rev program 145

rm 144

O

S, Sz

-o kapcsoló 103

-s kapcsoló 113

sed program 41, 55, 77, 141, 142, 158, 186

a (Append) 62

c (Change) 62

d (Delete) 58

i (Insert) 62

n (Next) 65

p (Print) 57

q (Quit) 65

r (Read) 64

s (Substitute) 58, 69

w (Write) 64

y (Transform) 63

seq segédprogram 30

sh parancsértelmező 4

Q

QUIT jel 120

sha-bang jelölés 3
 shift 9, 157, 169
 SIGKILL jel 189
 sleep 123
 sort 68, 90, 92, 140, 196
 stderr 13
 stdin 13
 sticky bit 114
 -stop kapcsoló 187
 stream editor 41
 szabályos kifejezés 39, 47, 49
 \$ 43
 . 43
 [...] 44
 [c1-c2] 44
 [karakterek] 43
 \c 42
 ^ 43
 „c” 42
 szűrők 108

T

tac 142
 tee 12, 31
 TERM jel 120
 test 23, 26, 103, 170,
 207
 tizedespont 65
 tizedesvessző 65

touch 144
 tr 64, 140, 148
 trap 118, 122, 124, 144, 152,
 157, 189

U

uniq 94, 140
 until 28, 88
 usleep 124

V

-v kapcsoló 80
 változók 5
 Végtelen ciklus 32

W

wait 122
 wc 105
 while ciklus 28, 31, 88, 107,
 109, 152, 171, 190, 196
 who 187

X

xargs 127



Ezt a könyvet elsősorban azok a UNIX operációs rendszerrel még csak ismerkedő számítógép-használók forgathatják haszonnal, akik a UNIX okozta „első traumán” már túlvannak. Értem ez alatt a néha kissé fura írásmódot, a más operációs rendszerektől eltérő gondolkodásmódot, no meg a sokat emlegetett szűkszavúságot.

Ez a könyv tehát bevezető jellegű, de nem azokhoz szól, akik még semmit nem tudnak a UNIX operációs rendszerről. Ennek megfelelően nem, vagy csak érintőlegesen tárgyalja például a UNIX alapfogalmait. Kivételt képeznek azok a téma körök, amelyek valamilyen módon szervesen kapcsolódnak a héjprogramozás elméletéhez vagy gyakorlatához.

Ahhoz tehát, hogy az itt leírtakat megértse, az Olvasónak már eleve rendelkeznie kell számos alapismerettel. Legalább érintőlegesen ismernie kell például a jogosultságok rendszerét, bizonyos alapvető UNIX parancsokat, alapszinten értenie kell, hogy mi az a szabványos be- és kimenet, és természetesen hatékonyan kell tudnia használni legalább egy szövegszerkesztőt. Összefoglalva tehát ez a könyv azoknak a középhaladóknak vagy haladóknak íródott, akik a tanulás folyamatában eljutottak arra a pontra, ahonnan a UNIX rendszert már szeretnék saját munkájukkal kapcsolatos tényleges feladatok megoldására is használni.

A főbb téma körök:

- Alapelemek
- Parancssori paraméterek
- Csövek
- Beágyazott dokumentumok
- Programvezérlési szerkezetek
- Függvények
- Keresés, szűrés, szövegfeldolgozás,
- Szabályos kifejezések
- A sed használata héjprogramokban
- Az AWK használata héjprogramokban
- Ciklusok
- A héjprogramok alapvető építőelemei
- Segédprogramok
- Tippek, trükkök

ISBN: 963 9301 10 8



ISBN szám: 963 9301 10 8

Ára: 2660 Ft

9 789639 301108