Hatékony AWK programozás

A GNU Awk felhasználói kézikönyve
1.0.4 kiadás
1999 április

Arnold D. Robbins
fordította Iványi Péter

"To boldly go where no man has gone before" a Paramount Pictures Corporation regisztrált védjegye.

A *Hatékony AWK programozás* 1.0.4 kiadása
a 3.0.4 (vagy későbbi) GNU AWK implementációt mutatja be.

Kiadta:

Specialized Systems Consultants, Inc. (SSC)
PO Box 55549
Seattle, WA 98155 USA
Phone: +1-206-782-7733
Fax: +1-206-782-7191
E-mail: `sales@ssc.com`
URL: `http://www.ssc.com/`

Free Software Foundation
59 Temple Place — Suite 330
Boston, MA 02111-1307 USA
Phone: +1-617-542-5942
Fax: +1-617-542-2652
E-mail: `gnu@gnu.org`
URL: `http://www.fsf.org/`

ISBN 1-882114-26-4

Borítóterv: Amy Wells Wood.

To Miriam, for making me complete.

To Chana, for the joy you bring us.

To Rivka, for the exponential increase.

To Nachum, for the added dimension.

Előszó

Ez a könyv az `awk` nyelvet mutatja be, és hogy hogyan lehet azt hatékonyan használni. Már ismerned kell néhány alapvető rendszerparancsot, úgy mint a `cat`, az `ls`[1], illetve tisztában kell legyél a shell lehetőségeivel, például bemenet/kimenet (I/O) átirányítás fogalmával és a csövek (pipe) használatával.

Az `awk` nyelv különböző implementációja sokféle számítógépen elérhető. Ez a könyv, bár az `awk` nyelv általános tulajdonságait is elmagyarázza, az `awk` egy speciális implementációját mutatja be, a `gawk`-ot ("GNU awk"). A `gawk` sokféle Unix rendszeren fut, kezdve a PC-ktől a Cray számítógépekig. A `gawk` futtatható MS-DOS, OS/2 PC, Atari, Amiga és VMS operációs rendszereken is.

Az `awk` és a `gawk` története

Az `awk` név az alkotók kezdőbetűiből áll össze: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. Az eredeti `awk` 1977-ben az AT&T Bell laboratóriumában készült. 1985-ben egy új verzió továbbfejlesztette a nyelvet, bevezette a felhasználó által definiálható függvényeket, több bemeneti folyamot (stream) és a dinamikus reguláris kifejezéseket. Ezt a verziót használta a Unix System V Release 3.1. A System V Release 4 -el érkező újabb verzió további lehetőségeket adott a nyelvhez, és le is tisztította a nyelv "sötét sarkait". A "POSIX Command Language and Utilities" (POSIX parancsnyelv és segédprogram) szabvány letisztázta a nyelv alapjait az eredeti (Bell laboratórium) `awk` és a `gawk` tervezőinek segítségével. A GNU implementációt, a `gawk`-ot, 1986-ban Paul Rubin és Jay Fenlason írta Richard Stallman tanácsai alapján. John Woods írta a kód egy részét. 1988-ban és 1989-ben David Trueman, Arnold Robbins segítségével, alaposan átdolgozta a `gawk`-ot, hogy kompatíbilis legyen az újabb `awk` verziókkal. A jelenlegi fejlesztés a hibák kijavítására, a teljesítmény javítására, a szabvánnyal való kompatíbilitás biztosítására és néha új lehetőségek kipróbálására koncentrál.

A GNU Project és ez a könyv

A Szabad Szoftver Alapítvány (Free Software Foundation - FSF) egy nonprofit szervezet, amely a szabadon terjeszthető szoftverek készítésére és terjesztésére jött létre. Richard M. Stallman, az eredeti Emacs editor írója, alapította. A GNU Emacs a legelterjedtebb Emacs verzió. A GNU project a Szabad Szoftver Alapítvány folyamatos erőfeszítése, hogy teljes, szabadon terjeszthető és POSIX kompatíbilis környezetet hozzon létre. (A

[1] Ezek a parancsok elérhetők POSIX kompatíbilis rendszereken, illetve hagyományos Unix rendszereken. Ha valamilyen más operációs rendszert használsz, akkor is tisztában kell lenned az I/O átirányítás fogalmával és csövek (pipe) használatával.

GNU jelentése: "GNU's not Unix".) Az FSF a "GNU General Public License" (vagy GPL) licencet használja annak biztosítására, hogy a szoftverei forráskódja mindig hozzáférhető legyen a felhasználók számára. A GPL egy példányát megtalálhatod a könyv végén (lásd [GNU GENERAL PUBLIC LICENSE], 307. oldal). A GPL a `gawk` C forráskódjára vonatkozik. Egy parancsértelmező (shell), egy editor (Emacs), hordozható, optimalizáló C, C++ és Objective-C fordítók (compiler), egy debugger és több tucat nagyobb és kisebb segédprogram (mint a `gawk`) a gyümölcsei a project-nek, és mind szabadon hozzáférhető. E dokumentum irásának pillanatában (1997 eleje) a GNU operációs rendszer kernele (a HURD), bár hozzáférhető, de a fejlesztés egy korai stádiumában van.

Amíg a GNU operációs rendszer el nem készül, a Linux rendszert érdemes használni, ami egy szabadon hozzáférhető, Unix jellegű operációs rendszer 80386, DEC Alpha, Sun SPARC és más rendszerekre. Sok könyvet írtak a Linuxról, az egyik szabadon elérhető a *Linux Installation and Getting Started* Matt Welsh által. Többféle Linux disztribúció elérhető, gyakran számítógépes boltokban vagy egy könyv CD mellékleteként. (További három, szabadon terjeszthető Unix jellegű operációs rendszer létezik, a NetBSD, a FreeBSD és az OpenBSD. Ezek a 4.4-Lite Berkeley Software Distribution -on alapulnak és a `gawk`-ot használják mint `awk`.)

Ez a könyv, amit éppen olvasol, szintén ingyenes. A könyvbeli információ szabadon hozzáférhető bárki számára, a könyv forrása a `gawk`-al egy közös csomagban található. Bárki lemásolhatja ezt a könyvet, annyiszor ahányszor csak akarja. (Szentelj egy percet a másolási engedély elolvasásának a "Copyright" oldalon.)

Ha pénzt fizettél ezért a könyvért, akkor amiért igazából fizettél az a könyv nyomtatása és kötése és a kiadó esetleges extra költségei. Mindent megtettünk, hogy ez a költség lehetőleg alacsony legyen; az emberek egy jelentős része a kötött könyvet részesíti előnyben, mint egy 330 oldalból álló fénymásolt, valamilyen módon összekötött anyagot (nem is beszélve a másolási időről és munkáról). Ugyanez igaz amikor a könyvet a forrásból gyártod le; a kötött könyv ára csak egy kicsit több mint ha magad nyomtatnád ki a teljes könyvet. Maga a könyv több előzetes kiadást élt meg. *A GAWK kézikönyv* egy piszkozatán kezdtem el dolgozni másokkal együtt, Diane Close, Paul Rubin és Richard Stallman, 1988 végén. Kb. 90 oldal volt és alig mutatta be az eredeti, "régi" `awk` verziót. Többszöri átdolgozás után *A GAWK kézikönyv* első változata 1989 oktoberében jelent meg, mint 0.11 Beta kiadás. További jelentős változtatások után a 0.13-as kiadás 1991 decemberében jelent meg. David Trueman, Pat Rankin és Michal Jaegermann írta a 0.13 -as kiadás egyes fejezeteit. Ezt az anyagot adta ki az FSF, mint kötött könyvet 1992-ben. Ezután több kisebb átdolgozás következett, úgy mint a 0.14-es kiadás 1992 novemberében, amit az FSF 1993 januárban adott ki, és a 0.16 -os kiadás 1993 augusztusában.

A *Hatékony AWK programozás* 1.0-ás kiadása *A GAWK kézikönyv* egy jelentős átdolgozása. Az FSF hozzájárult ahhoz, hogy most már én vagyok az elsődleges szerzője a könyvnek. Úgy éreztem, hogy találóbb cím is kellene ennek az új könyvnek.

A *Hatékony AWK programozás* feltétlenül fog még fejlődni. Egy elektronikus formája a `gawk` disztribúcióban is megtalálható. Ha hibát találsz a könyvben, kérlek jelentsd azt! Lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal, hogy hogyan lehet a problémákat jelenteni elektronikus formában, vagy írj nekem az FSF címére.

Köszönetnyilvánítás

Elismerésemet szeretném kifejezni Richard M. Stallman-nak, egy jobb világ víziójáért és bátorságáért, hogy megalapította az FSF-t és elindította a GNU project-et.

A GAWK kézikönyv eredeti tervezete az alábbi köszönetnyilvánítást tartalmazta:

> Sok embernek jár köszönet azért, hogy ennek a kézikönyvnek az elkészítésében közreműködött. Jay Fenlason sok ötletet adott és mintaprogramokat írt. Richard Mlynarik és Robert Chassell megjegyzésekkel segítette a kézikönyv létrejöttét. John W.Pierce (UC San Diego, Kémia tanszék) *A Supplemental Document for* `awk` című cikke pontosan bemutatta ennek a dokumentumnak és az `awk` nyelvnek azon hiányosságait, ami egyébként elkerülte volna a figyelmünket.

Az alábbi emberek hasznos tanácsokat adtak a *A GAWK kézikönyv* 0.13-as kiadásához: Rick Adams, Michael Brennan, Rich Burridge, Diane Close, Christopher ("Topher") Eliot, Michael Lijewski, Pat Rankin, Miriam Robbins és Michal Jaegermann.

Az alábbi emberek hasznos tanácsokat adtak a *Hatékony AWK programozás* 1.0-ás kiadásához: Karl Berry, Michael Brennan, Darrel Hankerson, Michal Jaegermann, Michael Lijewski és Miriam Robbins. Pat Rankin, Michal Jaegermann, Darrel Hankerson és Scott Deifik az 1.0-ás kiadás bekezdéseit frissítették.

Robert J. Chassell a Texinfo használatában nyújtott segítséget. Azért külön köszönet illeti, hogy *nem* engedte, hogy a könyv címe a *Hogyan Gawkoljunk illedelmesen* legyen. Karl Berry a Texinfo TeX részében segített sokat.

David Trueman-nak külön köszönet jár, mivel értékes segítséget nyújtott a `gawk` fejlesztésében, hogy az mindig kellően gyors legyen, és minél kevesebb hiba legyen a kódban. Bár ma már nem vesz részt a `gawk` fejlesztésében, nagy öröm volt vele dolgozni ezen a project-en.

Scott Deifik, Darrel Hankerson, Kai Uwe Rommel, Pat Rankin és Michal Jaegermann (nem fontossági sorrendben) a `gawk` hordozhatóságáért felelős csoport tagjai már hosszú ideje. Munkájuk és segítségük nélkül a `gawk` közel sem lenne olyan jó program, mint amilyen ma. Mindig is öröm volt és ma is az, hogy velük dolgozhatok.

Jeffrey Friedl felbecsülhetetlenül értékes segítséget nyújtott néhány, a `gawk` 3.0-ás verziójának kibocsátása előtti, a reguláris kifejezésekkel kapcsolatos hiba megtalálásában.

David és én meg szeretnénk köszönni Brian Kernighan (Bell laboratórium) felbecsülhetetlenül értékes segítségét a `gawk` tesztelése során, ill. hogy segített megérteni több problémás kérdést a nyelvvel kapcsolatban. Sem a `gawk`, sem a dokumentációja nem lenne olyan jó mint most a segítsége nélkül.

Meg szeretném köszönni Marshall és Elaine Hartholz-nak (Seattle) és Dr. Bert és Rita Schreiber-nek (Detroit) a hosszú és nyugodt vakációkat, amit otthonukban töltöttem, így jelentősen előre tudtam haladni a `gawk` és e könyv elkészítésében. Phil Hughes (SSC) szintén jelentősen hozzásegített a cél eléréséhez, mivel kölcsönadta a Linux laptop-ját, nem is egyszer, de kétszer, így lehetővé tette számomra, hogy az otthonomtól távol is dolgozhassam.

Legvégül, megköszönöm feleségemnek, Miriam, a türelmét e project többszöri átdolgozása során, azt hogy elsőként átolvasta és javította hibákat, illetve hogy megosztott velem egy számítógépet. Meg szeretném köszönni szüleimnek a szeretetüket és a jóindulatukat amivel felneveltek. Ezenkívül hálával gondolok Istenre a sok lehetőségért amit számomra nyújtott, és azokért a nekem adott “ajándékokért” amivel e lehetőségeket fel tudtam használni.

Arnold Robbins
Atlanta, Georgia
February, 1997

1. Bevezetés

Ha Te is olyan felhasználó vagy, mint megannyi más computer felhasználó, akkor gyakran szeretnél módosítani szöveg file-okat, amikben bizonyos minták sűrűn fordulnak elő, vagy bizonyos sorokból adatot szeretnél kinyerni, és minden mást eldobni. Egy C vagy Pascal proramot írni ilyen feladatra időt rabló és kényelmetlen, ill. több sornyi kódot igényel. Valószínűleg könnyebben megoldható a feladat `awk`-al.

Az `awk` program egy olyan speciális célú programnyelvet képes értelmezni, (mint egy interpreter - a fordító), amely lehetővé tesz egyszerű adatformázási munkákat néhány sor kóddal.

Az `awk` GNU megvalósítását `gawk`-nak hívják; teljesen kompatíbilis az `awk` System V Release 4 verziójával. Ezenkívül a `gawk` teljesen kompatíbilis az `awk` programozási nyelv POSIX szabványban rögzített definíciójával. Ez azt jelenti, hogy minden jól megírt `awk` programnak működnie kell a `gawk`-al is. Így nincs értelme különbséget tenni a `gawk` és az `awk` implementációja között.

Az `awk` használatával:

- kis, személyes adatbázist lehet üzemeltetni,
- összefoglalókat, jelentéseket lehet generálni,
- adatokat lehet ellenőrizni,
- tárgymutatót lehet generálni, és egyéb dokumentum feldolgozási, előkészítési feladatokat lehet elvégezni,
- ráadásul algoritmusokat lehet kipróbálni, amiket később át lehet ültetni más programozási nyelvekre.

1.1. A könyv használata

Az `awk` kifejezés vonatkozik magára a programra és a programozási nyelvre is. A könyv során ha szükséges a megkülönböztetés, akkor a programot "az `awk` segédprogram" és a nyelvet "az `awk` programozási nyelv" kifejezéssel jelöljük. A `gawk` kifejezés a GNU "project" részeként kifejlesztett `awk` verziót jelöli. A könyv célja, hogy bemutassa az `awk` programozási nyelvet és az `awk` program futtatási lehetőségeit.

A könyv fő célja bemutatni az `awk` programozási nyelv POSIX szabványban definiált tulajdonságait egy konkrét implementáción, a `gawk`-on, keresztül. Ugyanakkor megpróbáljuk felhívni a figyelmet a `gawk` és más `awk` megvalósítások közötti fontos különbségekre, illetve minden a `gawk` által implementált, de a POSIX szabványban nem rögzített lehetőségre is.

A könyv egyszerre próbál felhasználói és referencia kézikönyvként szolgálni, ezért ha kezdő felhasználó vagy, nyugodtan ugord át a bonyolultabb részeket. A sok kereszthivatkozás sem fontos most számodra, mivel ezek a profi felhasználók számára hasznosak, és a könyv "on-line" verziójához szükségesek.

Az `awk` *program* kifejezés az általad írt programra vonatkozik, amit az `awk` programozási nyelven írtál.

Lásd 2. Fejezet [Kezdő lépések `awk`-ban], 9. oldal, ami elmagyarázza az abszolút minimum és szükséges tudást az `awk` használatához.

Néhány hasznos "egysoros" program található a könyv következő fejezetében, (lásd 3. Fejezet [Hasznos egysoros programok], 21. oldal), hogy ráérezz az `awk` nyelvre.

További `awk` példaprogramokat találsz az alábbi fejezetekben. (Lásd 15. Fejezet [A Library of `awk` Functions], 169. oldal, illetve lásd 16. Fejezet [Practical `awk` Programs], 205. oldal).

A `awk` programozási nyelv összegzését a gyors keresés érdekében megtalálhatod a függelékben, A. Függelék [`gawk` Summary], 257. oldal. Ha csak a memoriádat akarod felfrissíteni az `awk` nyelv egy adott részleteről, akkor ezt a fejezetet neked találták ki.

Ha ismeretlen kifejezéssel találkozol, üsd fel a (lásd D. Függelék [Glossary], 299. oldal) fejezetet.

A legtöbb esetben teljes `awk` programot adunk meg mint példát, de a haladó fejezetekben előfordul, hogy csak egy programrészletet használunk a koncepció bemutatására.

A könyv főleg olyan felhasználókat céloz meg, akik még nem ismerik az `awk` programozási nyelvet, ugyanakkor sok információt tartogat a profi felhasználók számára is. Ilyen például az `awk` POSIX definíciója, és a példaprogramok jelentős része a 15. Fejezet [A Library of `awk` Functions], 169. oldal, és 16. Fejezet [Practical `awk` Programs], 205. oldalán.

Sötét sarkok

Ki nyitotta ki azt az elsötétített ablakot?!?
Gróf Dracula

A POSIX szabvány (és a *The Gawk Manual*) megjelenéséig az `awk` sok tulajdonsága gyengén vagy egyáltalán nem volt dokumentálva. Ezeket hívják "sötét sarkoknak". A könyvben az "(s.s.)" jelzéssel hívjuk fel rájuk a figyelmet, és a tárgymutatóban is megtalálhatóak a "sötet sarkok" címszó alatt.

1.2. Tipográfiai jelölések

Ez a könyv GNU Texinfo formátumot használja a szöveg szedésére. A Texinfo file-ból el lehet készíteni a dokumentum nyomtatott vagy "online" verzióját. Ebből kifolyólag a szedési minta eltér más, általad olvasott könyvekétől.

Az általad begépelhető parancssorokat megelőzi a "shell" elsődleges és másodlagos "prompt"-ja, '$' és '>'. A parancs kimenetét, eredményét a "⊣" jel előzi meg. Ez általában a parancs szabványos kimenetét jelöli. A hibaüzenetek és egyéb üzenetek előtt az "error" jel lesz feltüntetve. Például:

```
$ echo hi on stdout
⊣ hi on stdout
$ echo hello on stderr 1>&2
error hello on stderr
```

A szövegben a parancsnevek `ezzel a karakterrel` jelennek meg. A kódrészleteket ugyanezzel a betűvel írjuk, de idézőjelek között, '`például így`'. Néhány dolgot *hangsúlyozni* szeretnénk az előbbi módon, a különösen fontosakat pedig **így fogjuk írni**. Egy új kifejezés első előfordulásánál általában a *definícióját* is megadjuk, és ez olyan betűtípussal fog megjelenni, mint amit a "definíció" szónal használtunk ebben a mondatban. A file-nevek pedig így fognak kinézni: '`/path/to/ourfile`'.

Az általad begépelt betűk szedése *`ilyen lesz`*. Van néhány speciális karakter, a "vezérlő karakterek", amelyeket úgy tudsz begépelni, hogy lenyomva tartod a *`CONTROL`* billentyűt, amíg megnyomod a kívánt, másik billentyűt. Például a *`Control-d`* esetén először megnyomod és lenyomva tartod a *`CONTROL`* billentyűt, majd megnyomod a *`d`* billentyűt. Végül mindkét billentyűt felengeded.

1.3. Adat file-ok a példákhoz

A példák jelentős része két minta adat file-t használ. Az első file, '`BBS-list`', néhány elektronikus faliújságként szolgáló rendszer listája. A második file, '`inventory-shipped`', néhány szállítmány listáját tartalmazza havi bontásban. Mindkét file-ban, minden egyes sort egy *rekordnak* tekintünk.

A '`BBS-list`' file-ban, minden rekord tartalmazza a számítógép nevét, telefonszámát, az átviteli sebességét (baud) és egy kódot ami megmondja, hogy a gép hány órát van üzemben. Az '`A`' kód az utolsó oszlopban azt jelenti, hogy a faliújság napi 24 órát van üzemben. A '`B`' kódú számítógépek csak este és a hétvégi órákban üzemelnek. A '`C`' kód jelentése pedig, hogy a faliújság csak a hétvégén elérhető.

```
aardvark     555-5553     1200/300          B
alpo-net     555-3412     2400/1200/300     A
barfly       555-7685     1200/300          A
bites        555-1675     2400/1200/300     A
camelot      555-0542     300               C
core         555-2912     1200/300          C
fooey        555-1234     2400/1200/300     B
foot         555-6699     1200/300          B
```

```
macfoo       555-6480     1200/300          A
sdace        555-3430     2400/1200/300     A
sabafoo      555-2127     1200/300          C
```

A második file, '`inventory-shipped`', az egy év során teljesített szállítmányok listáját tartalmazza. Minden rekord tartalmazza a hónapot, a zöld rekeszek, a piros dobozok, a narancsos zsákok és a kék csomagok számát. A file 16 bejegyzése magába foglalja egy év 12 hónapját és a következő év első négy hónapját.

```
Jan  13  25  15 115
Feb  15  32  24 226
Mar  15  24  34 228
Apr  31  52  63 420
May  16  34  29 208
Jun  31  42  75 492
Jul  24  34  67 436
Aug  15  34  47 316
Sep  13  55  37 277
Oct  29  54  68 525
Nov  20  87  82 577
Dec  17  35  61 401

Jan  21  36  64 620
Feb  26  58  80 652
Mar  24  75  70 495
Apr  21  70  74 514
```

2. Kezdő lépések awk-ban

Az `awk` alapvető feladata, hogy olyan sorokat (vagy más szövegegységeket) keressen file-okban, amelyek tartalmaznak egy bizonyos mintát. Amikor egy sor illeszkedik egy mintára, az `awk` segédprogram egy megadott feladatot vagy tevékenységet végez el a soron. Ezt addig ismétli minden soron, amíg a file-ok végére nem ér.

Az `awk`-ban írt programok különböznek a más programozási nyelven írt programoktól, mert az `awk` programok *adatvezéreltek*; ez azt jelenti, hogy először azt az adatot adod meg amivel dolgozni szeretnél, és csak utána adod meg, hogy mit szeretnél az adattal csinálni, ha megtaláltad. A legtöbb programozási nyelv *procedurális*; apró részletekre lebontva meg kell adnod a programnak, hogy mit csináljon. Amikor procedurális programozási nyelvvel dolgozol, általában nehéz pontosan megadni, hogy a programod milyen adatot fog feldolgozni. Mivel ez az `awk`-ban nem így van, ezért fogod könnyűnek találni az `awk` programok írását és olvasását.

Az `awk` segédprogram futásához meg kell adnod egy `awk` *programot*, ami definiálja az `awk` számára hogy mit kell csinálnia. A program szabályokat tartalmaz. (Tartalmazhat függvény definíciót is, ami már bonyolultabb, és így most nem mélyedünk el benne, lásd 13. Fejezet [User-defined Functions], 153. oldal.) Minden szabály tartalmaz egy mintát, amit keresünk, és egy tevékenységet amit végre kell hajtani, ha a mintát megtalálta a program.

Szintaktikailag egy szabály egy mintából, majd egy azt követő tevékenységből áll. A tevékenységet kapcsos zárójelek közé kell tenni, hogy elválasszuk a mintától. A szabályok általában egy új sorral vannak elválasztva. Egy `awk` program így néz ki:

```
minta { tevékenység }
minta { tevékenység }
...
```

2.1. Egy rózsa tövisekkel

Az `awk` programozási nyelv hosszú évek során fejlődött. Erről további részletek találhatók egy későbbi fejezetben, 17. Fejezet [The Evolution of the `awk` Language], 251. oldal. Az ebben a könyvben bemutatott nyelvet gyakran hívják az "új `awk`"-nak.

Ennek következtében sok rendszer többféle `awk` verziót tartalmaz. Néhány rendszer rendelkezik egy `awk` segédprogrammal, ami az eredeti `awk` implementációja, és egy `nawk` programmal, ami az új verzió. Más rendszereken `oawk`-nak hívják a "régebbi `awk`" implementációt, és az `awk` nevet használják az új verzióra. Előfordul, hogy csak egy verzió található meg a rendszeren, általában az újabb.[1]

[1] Ezek a rendszerek a `gawk` implementációt használják leggyakrabban!

Ebben a helyzetben bonyolult lehet azt eldönteni, hogy melyik `awk` verziót használd amikor a programot írod. A legjobb tanács, amit adhatunk, hogy olvasd el a rendszereden levő dokumentációt. Ellenőrizd, hogy `awk`, `oawk`, `nawk` vagy `gawk` program létezik-e a rendszereden. Jó esélyed van arra, hogy az új verzió valamilyen implementációját megtalálod a gépeden, és azt tanácsoljuk, hogy ezt használd. (Természetesen, ha ezt a könyvet olvasod, nagyon valószínű, hogy a `gawk` implementációt megtalálod a rendszereden!)

Amikor olyan részletre hivatkozunk aminek bármely POSIX kompatíbilis implementációban szerepelnie kell, az `awk` nevet fogjuk használni, csak a GNU implementációra jellemző részeknél használjuk a `gawk` nevet.

2.2. Hogyan futtassunk `awk` programokat

Sokféle lehetőség van egy `awk` program futtatására. Ha a program rövid, a legegyszerűbb a parancssorban elhelyezni, például:

`awk '`*program*`'` *input-file1 input-file2* ...

ahol a *program* mintákat és tevékenységeket tartalmaz, ahogy korábban elmagyaráztuk. (Miért használunk idézőjeleket? A magyarázatot megtalálod alább, 2.2.1. Bekezdés [Egyszer használatos eldobható `awk` programok], 10. oldal.)

Amikor egy program hosszabb, kényelmesebb egy file-ban elhelyezni és így futtatni:

`awk -f` *program-file input-file1 input-file2* ...

2.2.1. Egyszer használatos, eldobható `awk` programok

Ha már jobban megismerted az `awk` programozási nyelvet, gyakran fog előfordulni, hogy az egyszerűbb programokat azonnal megírod a terminál előtt ülve, amint használni szeretnéd. Ebben az esetben az `awk` parancs a:

`awk '`*program*`'` *input-file1 input-file2* ...

ahol a *program* mintákat és tevékenységeket tartalmaz.

Ez a parancsformátum arra utasítja a *shell*-t, vagy a parancs értelmezőt, hogy indítsa el az `awk` segédprogramot, ami majd használni fogja a *programot* a bemeneti file-ok tartalmának feldolgozására. Az idézőjelek azért vannak a *program* körül, hogy a shell ne értelmezze ezt a részt. Ráadásul, az idézőjelek arra is utasítják a shell-t hogy az egészet egyetlen argumentumként adja át az `awk`-nak, így lehetővé téve több sorból álló *program* megadását is.

Ez a forma lényegében megfelel közepes méretű `awk` programok shell script-ből való futtatására is, mivel így elkerülhető a külön file használata az `awk` programhoz. Az egy file-ból álló programok megbízhatóbbak, mivel nincs másik file, amire szükség volna, és esetleg rossz helyen lenne.

A 3. Fejezet [Hasznos egysoros programok], 21. oldal, bemutat néhány rövid, egyszerű programot.

Mint érdekesség, megjegyezzük, hogy az

```
awk '/foo/' files ...
```

parancs lényegében ugyanaz mint a

```
egrep foo files ...
```

2.2.2. Az awk futtatása bemeneti file-ok nélkül

Az `awk` segédprogram bemeneti file-ok nélkül is futtatható. Ha begépeled:

```
awk 'program'
```

akkor az `awk` *programot* a *szabványos bemenetre* fogja alkalmazni, ami általában az a szöveg, amit a terminálon begépelsz. Ez egészen addig tart amíg be nem gépeled hogy *Control-d*, vagyis jelzed hogy a "file"-nak vége. (Más operációs rendszereken a file vége jel más lehet, pl. OS/2 és MS-DOS alatt a *Control-z*.)

Például az alábbi program kinyomtat egy barátságos tanácsot, (ollózva Douglas Adams művéből, *Galaxis útikönyv stopposoknak*), aminek segítségével elkerülheted a számítógép által okozott fejfájást. ('BEGIN'-t még eddig nem magyaráztuk el)

```
$ awk "BEGIN { print \"Don't Panic!\" }"
⊣ Don't Panic!
```

Ez a program nem olvas be semmit. A '\' jel minden belső macskaköröm előtt szükséges a shell miatt, különösen mivel idézőjelet és macskakörmöt kevertünk a szövegben.

A következő egyszerű `awk` program a `cat` segédprogramot emulálja; amit begépelsz, kinyomtatja. (Hogy ez miért működik, azt hamarosan eláruljuk.)

```
$ awk '{ print }'
Now is the time for all good men
⊣ Now is the time for all good men
to come to the aid of their country.
⊣ to come to the aid of their country.
Four score and seven years ago, ...
⊣ Four score and seven years ago, ...
What, me worry?
⊣ What, me worry?
Control-d
```

2.2.3. Hosszú programok futtatása

Előfordul, hogy az `awk` programod hosszú lesz. Ebben az esetben kényelmesebb lehet a programot egy külön file-ba tenni. Az `awk` parancs ennek a file-nek a futtatására:

```
awk -f source-file input-file1 input-file2 ...
```

A '-f' argumentum adja meg az awk-nak, hogy a végrehajtandó program a *source-file*-ban van. A *source-file*-nak bármilyen neve lehet. Például az alábbi programot

```
BEGIN { print "Don't Panic!" }
```

beteheted az 'advice' file-ba. A parancs:

```
awk -f advice
```

ugyanazt fogja csinálni, mint a :

```
awk "BEGIN { print \"Don't Panic!\" }"
```

amit már korábban elmagyaráztunk. (Lásd 2.2.2. Bekezdés [Az awk futtatása bemeneti file-ok nélkül], 11. oldal.) Érdemes megjegyezni, hogy a file-név köré, amit a '-f' argumentummal definiálsz, nem kellenek idézőjelek, mivel a file-nevek általában nem tartalmaznak speciális shell karaktereket. Természetesen az idézőjelek az 'advice' file-on belül sem kellenek az awk program köré. Az idézőjelek csak a parancssorban megadott awk programok esetén kellenek.

Ha azonosítani szeretnéd awk programjaidat, akkor használhatod a '.awk' kiterjesztést a file név után. Ez nem befolyásolja az awk program működését, de egyszerűbbé teheti file-jaid nyilvántartását.

2.2.4. Futtatható awk programok

Miután megtanultad az awk nyelvet, önmagában futtatható programokat is írhatsz a '#!' mechanizmus segítségével. A legtöbb Unix rendszeren[2] ez működni fog (és valamikor a GNU rendszeren is.)

Például kiegészítheted az 'advice' file-t az alábbiak szerint:

```
#! /bin/awk -f

BEGIN { print "Don't Panic!" }
```

Miután a file-t futtathatóvá tetted (a chmod segédprogrammal), ha egyszerűen begépeled a shell-nek, hogy 'advice', a rendszer úgy állít be mindent, mintha az awk[3] segédprogramot a 'awk -f advice' paranccsal futtatnád.

```
$ advice
⊣ Don't Panic!
```

[2] Az '#!' mechanizmus működik Linux rendszereken, azokon a Unix rendszereken amelyek a Berkeley Unix, System V Release 4 családba tartoznak és néhány System V Release 3 rendszeren is.

[3] A '#!' jellel kezdődő sor tartalmazza az interpreter teljes nevét és egy opcionális argumentumot az interpreter számára. Az operációs rendszer futtatja az interpretert a file-ban megadott argumentummal és az elindított programnak megadott teljes argumentum listával. Az első argumentum a listában az awk program neve. A további argumentumok vagy awk opciók vagy adat file-ok vagy mindkettő.

Az önmagában futtatható `awk` programok akkor is jó szolgálatot tehetnek, ha olyan programot szeretnél írni, amit más felhasználók is futtathatnak anélkül, hogy ismernék az `awk`-ot.

Figyelem: Ne tegyél egynél több argumentumot a '`#!`' jellel kezdődő sorba az `awk` után. Ez nem működik. Az operációs rendszer az interpreter utáni részt úgy kezeli, mint egyetlen argumentum, és így adja át az `awk`-nak. Következésképpen furcsa dolgok történhetnek: a legvalószínűbb, hogy a program használatáról kapsz valamilyen ismertetést.

Néhány régebbi rendszer nem támogatja a '`#!`' mechanizmust. Ugyanazt a hatást érheted el egy shell script-el, aminek valahogy így kell kinéznie:

```
: Ezt biztosítja, hogy az alap shell fogja végrehajtani.
awk 'program' "$@"
```

Ebben az esetben *alapvető* fontosságú, hogy a *programot* idézőjelek közé tedd, így a shell nem fogja értelmezni. Ha mégis lehagynád, csak egy shell varazsló a megmondhatója, hogy mi fog történni.

A `"$@"` jel hatására a shell minden, a parancssorban megadott argumentumot átad az `awk` segédprogramnak, anélkül hogy azokat értelmezné. Az első sorban elhelyezett kettőspont segítségével a program még C shell-t használó embereknek is működni fog. (Nem minden régebbi rendszer ismeri ezt a módszert, de a legtöbb igen.)

2.2.5. Megjegyzések az awk programokban

Egy *megjegyzés* lényegében egy szöveg, ami az emberek számára került be a programba, és nem igazán része a programnak. A megjegyzések például elmagyarázhatják, hogy a program mit csinál, és hogyan működik. Szinte minden programozási nyelvben lehet megjegyzéseket elhelyezni, mivel a programokat általában nehezebb megérteni a megjegyzések által adott extra segítség nélkül.

Az `awk` programozási nyelvben a megjegyzések egy '`#`' karakterrel kezdődnek és a sor végéig tartanak. A '`#`' karakternek nem kell a sor első karakterének lennie. Az `awk` nyelv a karakter utáni teljes sort eldobja. Például az alábbiakkal egészíthetjük ki az '`advice`' programot:

```
# Ez a program egy barátságos üzenetet nyomtat ki. Így
# segít abban, hogy ne félj az új dolgoktól.
BEGIN { print "Don't Panic!" }
```

Ha akarsz, tehetsz megjegyzéseket az egyszer használatos programokba is, de ez nem túl hasznos; a megjegyzések fő célja, hogy segítsen téged vagy más felhasználókat a program megértésében egy későbbi időpontban.

Figyelem: Ahogy azt már korábban megemlítettük, 2.2.1. Bekezdés [Egyszer használatos eldobható `awk` programok], 10. oldal, a kicsi és közepes programokat idézőjelek közé teheted, hogy önmagukban is futtatható shell script-ek legyenek. Ha ezt a módszert választod *ne* használj idézőjelet a megjegyzésekben (sehol a programodban). A shell ezt úgy értelmezné mint a lezárását a kezdő idézőjelnek. Az eredmény? A shell jelezni fogja hogy nincs azonos számú kezdő és záró idézőjel, és ha ezután az `awk` mégis elindulna, akkor valószínűleg hibaüzeneteket fog kiírni. Például:

```
awk 'BEGIN { print "hello" } # let's be cute'
```

2.3. Egy nagyon egyszerű példa

Az alábbi parancs egy egyszerű `awk` programot futtat le, ami a '`foo`' szó minden előfordulását megkeresi a '`BBS-list`' file-ban.

```
awk '/foo/ { print $0 }' BBS-list
```

Amikor az `awk` olyan sort talál, ami tartalmazza a '`foo`' szót, kinyomtatja az adott sort, mivel a '`print $0`' ezt jelenti: nyomtasd ki az aktuális sort. (Ugyanez érhető el az egyszerű '`print`' paranccsal, így az is használható.)

A példában '`/`' jelek veszik körül a '`foo`' szót az `awk` programban. Ezek a jelek határozzák meg, hogy a '`foo`' mintát szeretnénk megtalálni. Az ilyen mintákat *reguláris kifejezésnek* nevezzük és később tárgyaljuk részletesen (lásd 4. Fejezet [Reguláris kifejezések], 23. oldal). Lehet, hogy a minta a szónak csak egy részére illeszkedik. Az `awk` program körüli idézőjelek hatására a shell semelyik speciális shell karaktert (ami esetleg előfordulhat a mintában) nem fogja értelmezni.

A program eredménye:

```
$ awk '/foo/ { print $0 }' BBS-list
⊣ fooey        555-1234     2400/1200/300     B
⊣ foot         555-6699     1200/300          B
⊣ macfoo       555-6480     1200/300          A
⊣ sabafoo      555-2127     1200/300          C
```

Egy `awk` szabályban vagy a minta vagy a tevékenység elhagyható, de egyszerre mindkettő nem. Ha a minta nincs megadva, akkor az adott tevékenység minden sorra végrehajtódik. Ha a tevékenység lett elhagyva, akkor az alaptevékenység hajtódik végre, kinyomtat minden olyan sort, amire a minta illeszkedik.

Ezért van az, hogy a fenti példában elhagyhatjuk a tevékenységet (a `print` és a kapcsos zárójeleket) és az eredmény ugyanaz lesz: minden olyan sort kinyomtat, ami illeszkedik a '`foo`' mintára. Összehasonlításképpen, ha a `print` tevékenységet elhagyjuk, de a kapcsos zárójeleket megtartjuk, ami egy üres tevékenységet jelent, akkor egyetlen sort sem fog a program kinyomtatni.

2.4. Egy példa két szabállyal

Az `awk` segédprogram a bemeneti file-okat olvassa soronként és megpróbálja illeszteni a mintákat minden egyes sorra. Ha több minta is illeszkedik, akkor több tevékenység hajtodik végre, abban a sorrendben, ahogy az `awk` programban definiálva lettek. Ha egyetlen minta sem illeszkedik, akkor nem történik semmi.

Miután az aktuális sorra illeszkedő szabályokat feldolgozta (lehet hogy egyet sem), az `awk` beolvassa a következő sort (de érdemes figyelembe venni a 9.7. Bekezdés [A `next` kifejezés], 111. oldal, és a 9.8. Bekezdés [A `nextfile` kifejezés], 112. oldal által leírtakat). Ezt a folyamatot a file végéig ismétli.

Például az alábbi `awk` program:

```
/12/  { print $0 }
/21/  { print $0 }
```

két szabályt tartalmaz. Az első szabályban a minta az '`12`' szó, és a '`print $0`' a tevékenység. A második szabály mintája a '`21`' szó, és a tevékenység szintén a '`print $0`'. Mindegyik szabály körül kapcsos zárójelek vannak, külön-külön.

Ez az `awk` program kinyomtat minden olyan sort, amelyik *vagy* az '`12`' *vagy* a '`21`' szót tartalmazza. Ha egy sor mindkét szót tartalmazza, akkor kétszer lesz kinyomtatva, mindegyik szabály által egyszer.

Ha a fenti programot a két mintafile-lal ('`BBS-list`' és '`inventory-shipped`') futtatjuk, akkor az alábbi eredményt kapjuk:

```
$ awk '/12/ { print $0 }
>      /21/ { print $0 }' BBS-list inventory-shipped
⊣ aardvark     555-5553     1200/300          B
⊣ alpo-net     555-3412     2400/1200/300     A
⊣ barfly       555-7685     1200/300          A
⊣ bites        555-1675     2400/1200/300     A
⊣ core         555-2912     1200/300          C
⊣ fooey        555-1234     2400/1200/300     B
⊣ foot         555-6699     1200/300          B
⊣ macfoo       555-6480     1200/300          A
⊣ sdace        555-3430     2400/1200/300     A
⊣ sabafoo      555-2127     1200/300          C
⊣ sabafoo      555-2127     1200/300          C
⊣ Jan  21  36  64 620
⊣ Apr  21  70  74 514
```

Érdemes megfigyelni, hogy a '`sabafoo`' kezdetű sor a '`BBS-list`' file-ból kétszer lett kinyomtatva, mindegyik szabály kinyomtatta egyszer.

2.5. Egy bonyolultabb példa

A következő példa talán jobban bemutatja, hogy az `awk` mire is képes, úgy mint összegzésre, kiválasztásra és egy másik segédprogram kimenetének újrarendezésére. A példa tartalmaz olyan programozási megoldásokat, amiket eddig nem tárgyaltunk, így ne aggódj, ha nem érted minden részletét.

```
ls -lg | awk '$6 == "Nov" { sum += $5 }
```

```
           END { print sum }'
```

A parancs kinyomtatja az adott könyvtárban levő és legutoljára novemberben (bármely év) módosított file-ok összegzett méretét byte-ban. (Ha C shell-t használsz, akkor az első sor végén '`;\`' -t kell begépelni; a POSIX szabvánnyal kompatíbilis shell-ek esetén, mint a Bourne shell vagy a GNU Bourne-Again shell, a példát begépelheted, ahogy fent látod.)

Az '`ls -lg`' része a példának egy rendszerparancs, ami kilistázza a file-okat a könyvtárban, a file méretét és az utolsó módosítás dátumát. Az eredmény így néz ki:

```
-rw-r--r--  1 arnold   user   1933 Nov  7 13:05 Makefile
-rw-r--r--  1 arnold   user  10809 Nov  7 13:03 gawk.h
-rw-r--r--  1 arnold   user    983 Apr 13 12:14 gawk.tab.h
-rw-r--r--  1 arnold   user  31869 Jun 15 12:20 gawk.y
-rw-r--r--  1 arnold   user  22414 Nov  7 13:03 gawk1.c
-rw-r--r--  1 arnold   user  37455 Nov  7 13:03 gawk2.c
-rw-r--r--  1 arnold   user  27511 Dec  9 13:07 gawk3.c
-rw-r--r--  1 arnold   user   7989 Nov  7 13:03 gawk4.c
```

Az első mező tartalmazza az olvasási-írási jogokat, a második a link-ek számát a file-ra. A harmadik mező adja meg a file tulajdonosának azonosítóját. A negyedik mező azonosítja a csoportot, míg az ötödik mező tartalmazza a file méretét byte-okban. A hatodik, hetedik és nyolcadik mező az utolsó módosítás hónapját, napját és óráját adja meg. Végül a kilencedik mező a file neve.

A '`$6 == "Nov"`' kifejezés az `awk` programban egy ellenőrzés, hogy az '`ls -lg`' kimenetében a hatodik mező megegyezik-e a '`Nov`' szóval. Minden olyan sorra, aminél a hatodik mező megegyezik a '`Nov`' szóval, a '`sum += $5`' tevékenység hajtódik végre. Ez hozzáadja az ötödik mező tartalmát a `sum` változóhoz. Miután az `awk` befejezte a bemenet olvasását, a `sum` változó fogja tartalmazni azon file-ok méretének összegét, amelyeknél illeszkedést talált. (Ez azért fog működni, mert minden `awk` változó automatikusan zérus értéket kap kezdetben.)

Az `ls`-ből származó utolsó sor beolvasása után az `END` szabály hajtódik végre és az `awk` kinyomtatja a `sum` változó értékét. Ebben a példában a `sum` változó értéke 80600 lesz.

További bonyolult `awk` programozási technikákat mutatunk be a későbbi fejezetekben (lásd 8.2. Bekezdés [Tevékenységek áttekintése], 102. oldal), de mielőtt továbblépnél, fontos megérteni, hogy az `awk` hogyan dolgozza fel a bemenetet, és jeleníti meg az eredményt. A mezők manipulálásával és a `print` paranccsal több hasznos és meggyőző jelentést, összefoglalót készíthetsz.

2.6. `awk` kifejezések és sorok

Általában egy sor egy kifejezést vagy szabályt tartalmaz az `awk` programokban, például:

```
awk '/12/  { print $0 }
     /21/  { print $0 }' BBS-list inventory-shipped
```

Ugyanakkor a `gawk` nem veszi figyelembe az új sort az alábbi jelek után:

```
,    {    ?    :    ||    &&    do    else
```

Az új sor bármely más esetben egy kifejezés végét jelenti. (Egy új sor kezdésének lehetősége a kérdőjel és a kettőspont után anélkül, hogy új kifejezés kezdődne, egy `gawk` kiegeszítés. A kérdőjel és a kettőspont, mint egy feltételes kifejezés jelennek itt meg, amit a 7.12. Bekezdés [Feltételes kifejezések], 92. oldal fejezetben magyarázunk el.)

Ha egy kifejezést két sorban szeretnél leírni, és egy olyan pontnál új sort kezdeni, ahol alapesetben a kifejezés végét jelentené az új sor, akkor az első sor végére egy '\' jelet kell írni. A '\' jel legyen az utolsó a sorban. Ez a módszer bárhol megengedett, akár egy kifejezésben, egy szöveg közepén vagy egy reguláris kifejezésben. Például:

```
awk '/Ez a reguláris kifejezés túl hosszú, ezért\
 ebben sorban folytatjuk./ { print $1 }'
```

A könyvben bemutatott példákban általában nem használjuk ezt a lehetőséget, mivel a `gawk`-ban a sor hossza nem limitált; egyszerűen a programokat teheti olvashatóbbá. Ugyanezen oknál fogva, és az érthetőség kedvéért a példaprogramokban csak rövid kifejezéseket használtunk. A '\' jel használata talán a leghasznosabb, ha az `awk` programodat egy külön file-ban tárolod, és nem parancsként gépeled be. Fontos megjegyezni, hogy sok `awk` implementáció érzékeny arra, hogy hol használjuk a '\' jelet. Például nem engedélyezett a szövegek két sorba vágása a '\' jellel bizonyos `awk` implementációkban. Ha különböző `awk` implementációkkal is szeretnéd a programodat futtatni, a legegyszerűbb ha a szövegeket és reguláris kifejezéseket nem vágod szét a '\' jellel.

Figyelem: a '\' jel fent leírt használata nem fog működni C shell-el. A módszer csak külön file-be írt programok és a POSIX szabvánnyal kompatíbilis shell-ek, a Bourne shell vagy a Bash, esetén működik. De a C shell (`csh`) másképp viselkedik. A C shell esetén két '\' jelet kell használni a sor szétvágásárá, ráadásul a C shell-ben minden sort egy '\' jellel kell lezárni, hogy a következő sorban folytatható legyen. Tehát:

```
% awk 'BEGIN { \
?   print \\
?       "hello, world" \
? }'
 ⊣ hello, world
```

Itt a '`%`' és a '`?`' jelek a C shell elsődleges és másodlagos "prompt"-jai, mint a '`$`' és a '`>`' jelek a szabványos shell-ek esetén.

Az `awk` egy sor orientált programozási nyelv. Minden szabály tevékenység részének ugyanabban a sorban kell kezdődnie, mint a keresési minta. Ahhoz, hogy a tevékenység és a minta külön sorba kerülhessen a '\' jelet kell használni a fent leírt módon – ez az egyetlen megoldási lehetőség erre.

Fontos megjegyezni, hogy a '\' jel használata és a megjegyzések nem keverhetők. Amint egy '#' jelet beolvas az awk *minden* további karaktert eldob a sorban. Például:

```
$ gawk 'BEGIN { print "dont panic" # a friendly \
>                                    BEGIN rule
> }'
error  gawk: cmd. line:2:                BEGIN rule
error  gawk: cmd. line:2:                ^ parse error
```

Itt úgy tűnik, mintha a '\' jel megengedné a megjegyzés folytatását a következő sorban, de a '\' jelet az awk soha nem veszi észre, mert az a megjegyzésben "el van rejtve". Így a 'BEGIN' kifejezés egy hibát generál.

Ha egy kifejezés egy szabályon belül nagyon rövid, lehetőség van további kifejezéseknek ugyanabba a sorba helyezésére. Ezeket a kifejezéseket a ';' jellel kell elválasztani egymástól.

Ez természetesen igaz a szabályokra is, így a fenti program az alábbi formában is helyes:

```
/12/ { print $0 } ; /21/ { print $0 }
```

Megjegyzés: az egy sorba kerülő szabályok ';' jellel való elválasztása nem része az eredeti awk nyelvnek; a kifejezések egységes kezelése érdekében került bevezetésre.

2.7. Egyéb awk sajátosságok

Az awk programozási nyelv számos olyan előre definiált vagy beépített változó elérését biztosítja, amelyek valamilyen extra információt nyújtanak a programnak. Más változók segítségével a program befolyásolhatja az awk működését.

Ráadásul, az egyszerű számítások és szöveges műveletek elvégzésére számos beépített függvény áll rendelkezésre az awk-ban.

A könyv során fokozatosan mutatjuk be a beépített változókat és függvényeket. A szisztematikus definíciójukat két fejezetben találhatod meg: 10. Fejezet [Beépített változók], 115. oldal, 12. Fejezet [Beépített függvények], 133. oldal.

2.8. Mikor használjunk awk-ot

Talán elgondolgoztál már azon, hogy hogyan is lehet az awk hasznos számodra. Az egyik válasz, hogy okosan megtervezett mintákkal, mezőelválasztó jelekkel, matematikai kifejezésekkel és más kiválasztási kritériumokkal bonyolult összefoglalásokat, jelentéseket tudsz készíteni. Az awk nyelv nagyon hasznos lehet nagy mennyiségű nyers adat feldolgozására, például más segédprogramok kimenetének összegzésére (lásd 2.5. Bekezdés [Egy bonyolultabb példa], 15. oldal).

Az `awk`-ban írt programok általában kisebbek, mint más programozási nyelven írt programok. Ezért egyszerű az `awk` programok írása és használata. Gyakran a leggyorsabb módszer, egyszerűen csak leülni a terminálhoz és megírni az `awk` programot, majd a használat után egyszerűen eldobni. Mivel az `awk` programok interpretáltak (egy értelmező dolgozza fel és hajtja végre), elkerülhető a fordítási rész a tipikus szerkesztés-fordítás-tesztelés-hibakeresés software fejlesztési ciklusból.

Több bonyolult programot írtak már `awk`-ban, mint például egy teljes assemblert 8 bites mikroprocesszorokra (további részletek a lásd D. Függelék [Glossary], 299. oldal) és egy microcode assemblert speciális Prolog számítógépek számára. Ugyanakkor ezek az alkalmazások már megmutatják az `awk` korlátait is.

Ha, mondjuk, néhány száz sorból álló programokat írsz `awk`-ban, akkor érdemes megfontolni egy másik programozási nyelv használatát. Az Emacs Lisp egy jó választás ha bonyolult szöveg és minta illesztéseket kell csinálni. A shell is alkalmas erre, ráadásul más rendszer segédprogramok is elérhetők. A C, C++ és Lisp programozási nyelvek szintén jó környezetet biztosítanak rendszer programozáshoz és nagyobb munkák kivitelezéséhez. Az ezen a nyelveken írt programok általában hosszabbak, mint az `awk` változataik, de egyszerűbb őket karbantartani, és általában gyorsabbak.

3. Hasznos egysoros programok

Sok hasznos `awk` program rövid, csak egy vagy két sorból áll. Az alábbiakban bemutatunk néhányat. Néhány program olyan programozási megoldásokat is tartalmaz, amelyeket eddig nem tárgyaltunk, de a programhoz fűzött magyarázatokkal talán érthető lesz, hogy mi is történik.

A legtöbb példa egy '`data`' nevű adat file-t használ, ami csak a file helyét jelöli; ha használni akarod a programokat, akkor a saját file-od nevét kell a '`data`' helyére írni.

```
awk '{ if (length($0) > max) max = length($0) }
     END { print max }' data
```

A program kinyomtatja a leghosszabb sor hosszát a file-ban.

```
awk 'length($0) > 80' data
```

A program kinyomtat minden olyan sort, ami hosszabb mint 80 karakter. Az egyetlen szabályban egy feltétel szolgál mintaként, és nincs tevékenység megadva (az alaptevékenységet (nyomtasd ki a sort) használjuk).

```
expand data | awk '{ if (x < length()) x = length() }
                   END { print "maximum line length is " x }'
```

A program kinyomtatja a '`data`' file-ban szereplő leghosszabb sor hosszát. A bemenetet az `expand` parancs dolgozza fel először; lecseréli a tab karaktereket azonos számú szóközre, így amikor a sor hosszát vizsgálja, lényegében a jobb oldali margó oszlopszámát veszi figyelembe.

```
awk 'NF > 0' data
```

A program kinyomtat minden olyan sort, amiben legalább egy mező szerepel. Ez egy egyszerű megoldás az üres sorok eltávolítására (lényegében egy olyan, az eredetihez hasonló file-t hozunk létre, amiből az üres sorok hiányoznak).

```
awk 'BEGIN { for (i = 1; i <= 7; i++)
               print int(101 * rand()) }'
```

A program hét darab zérus és 100 közé eső véletlen számot generál.

```
ls -lg files | awk '{ x += $5 } ; END { print "total bytes: " x }'
```

A program a *files* összesített méretét adja meg byte-ban.

```
ls -lg files | awk '{ x += $5 }
                     END { print "total K-bytes: " (x + 1023)/1024 }'
```

A program a *files* összesített méretét adja meg kilobyte-ban.

```
awk -F: '{ print $1 }' /etc/passwd | sort
```

A program az összes felhasználó belépési nevének rendezett listáját adja vissza.

`awk 'END { print NR }' data`
: A program a sorok számát adja meg a file-ban.

`awk 'NR % 2 == 0' data`
: A program a '`data`' file páros számú sorait írja ki. Ha az '`NR % 2 == 1`' kifejezést használod, akkor a páratlan számú sorokat kapod meg.

4. Reguláris kifejezések

A *reguláris kifejezések*, vagy *regexp*, szavak (betűk bizonyos halmazának) leírására alkalmasak. Mivel a reguláris kifejezések alapvető részei az `awk` programozási nyelvnek, ezért egy önálló fejezetben tárgyaljuk őket.

A '`/`' karakterek közé tett reguláris kifejezések szolgálnak `awk` mintaként és illeszkednek minden olyan sorra, amelyik az adott halmazhoz tartozik.

A legegyszerűbb reguláris kifejezés betűk és/vagy számok sorozata. Ezek a reguláris kifejezések minden olyan szóra illeszkednek, amelyek tartalmazzák az adott sorozatot. Így a '`foo`' reguláris kifejezés illeszkedik minden olyan szóra ami tartalmazza '`foo`'-t. Továbbá a `/foo/` minta illeszkedik minden olyan bemeneti sorra amelyben a '`foo`' karaktersorozat, *bárhol* a sorban, előfordul. Más reguláris kifejezések lehetővé teszik bonyolultabb betűhalmazok definiálását.

Kezdetben a példák egyszerűek lesznek, de ahogy többet mondunk el arról, hogyan működnek a reguláris kifejezések, úgy egyre bonyolultabb példákat mutatunk.

4.1. Hogyan használjuk a reguláris kifejezéseket

Egy reguláris kifejezés mintaként használható ha '`/`' karakterek közé tesszük. Ebben az esetben egy reguláris kifejezés minden sor teljes szövegével összehasonlításra kerül. (Általában a szövegnek csak egy részéhez kell illeszkednie, hogy az összehasonlítás sikeres legyen.) Például ez a program kinyomtatja minden olyan sor második mezőjét, amely bárhol tartalmazza a '`foo`' karaktersorozatot:

```
$ awk '/foo/ { print $2 }' BBS-list
 ⊣ 555-1234
 ⊣ 555-6699
 ⊣ 555-6480
 ⊣ 555-2127
```

A reguláris kifejezések alkalmasak kifejezések keresésére is. Ebben az esetben a keresett kifejezést adja meg a reguláris kifejezés; ugyanakkor amiben keresünk nem feltétlenül a teljes sor. Két operátor, a '`~`' és a '`!~`', használható a keresésnél. Ezekkel az operátorokkal leírt kifejezések használhatók mint minták az `if`, `while`, `for` és a `do` kifejezésekben.

exp ~ /*regexp*/

A kifejezés igaz, ha a *regexp* kifejezés illeszkedik az *exp* kifejezésre (mint karaktersorozatra). Az alábbi példa kikeres minden olyan sort, amelyben az első mező tartalmazza a nagy '`J`' betűt:

```
$ awk '$1 ~ /J/' inventory-shipped
⊣ Jan  13  25  15 115
⊣ Jun  31  42  75 492
⊣ Jul  24  34  67 436
⊣ Jan  21  36  64 620
```

Ez is ugyanazt csinálja:

```
awk '{ if ($1 ~ /J/) print }' inventory-shipped
```

exp `!~ /`*regexp*`/`
: A kifejezés igaz, ha a *regexp* kifejezés *nem* illeszkedik az *exp* kifejezésre (mint karaktersorozatra). Az alábbi példa kiválaszt minden olyan sort, amelyben az első mező *nem* tartalmazza a nagy 'J' betűt:

```
$ awk '$1 !~ /J/' inventory-shipped
⊣ Feb  15  32  24 226
⊣ Mar  15  24  34 228
⊣ Apr  31  52  63 420
⊣ May  16  34  29 208
...
```

Amikor egy reguláris kifejezést '/' karakterek között írunk le, általában *konstans reguláris kifejezésnek* hívjuk, mint például az `5.27` egy szám konstans, és a `"foo"` egy szöveg konstans.

4.2. Escape szekvenciák

Néhány karaktert nem lehet közvetlenül használni egy szöveg konstansban (`"foo"`) vagy konstans reguláris kifejezésben (`/foo/`), mivel speciális jelentésük van. Ezeket a karaktereket úgynevezett *escape szekvenciákkal* írhatjuk le; a nevüket onnan kapták, hogy egy '\' (escape) karakterrel kezdődnek.

Az escape szekvenciák egyik használati lehetősége, hogy a macskaköröm karaktert egy szövegben elhelyezzük. Egy önmagában álló macskaköröm karakter a szöveg lezárását jelentené, ezért kell a '\"' karaktersorozatot használni a szövegen belül. Például:

```
$ awk 'BEGIN { print "He said \"hi!\" to her." }'
⊣ He said "hi!" to her.
```

A '\' karakter maga is egy speciális karakter és nem szerepelhet önállóan a szövegben; a '\\' karaktersorozatot kell használni egy szövegben vagy egy reguláris kifejezésben. Például azt a szöveget, amely egy macskakörmöt és egy '\' karaktert tartalmaz, így kell leírni: `"\"\\"`.

A '\' karakter másik alkalmazási köre a kinyomtathatatlan karakterek reprezentációja, mint például a tab és az új sor karakter. Bár semmi nincs ami megakadályozhatna, hogy ezeket a karaktereket közvetlenül használd egy szövegben vagy egy reguláris kifejezésben, de lehet hogy az összhatás elég csúnya lenne a használatukkal.

Az alábbiakban az `awk`-ban használható escape szekvenciák egy táblázatát adjuk meg. Hacsak nem írjuk másképp, ezek az escape szekvenciák mind a szövegekben, mind a reguláris kifejezésekben érvényesek.

`\\` A '`\`' karakter maga.

`\a` A figyelmeztetés (alert) karakter, *Control-g*, ASCII code 7 (BEL).

`\b` Visszatörlés (backspace), *Control-h*, ASCII code 8 (BS).

`\f` Lapdobás (formfeed) karakter, *Control-l*, ASCII code 12 (FF).

`\n` Új sor karakter, *Control-j*, ASCII code 10 (LF).

`\r` Kocsi vissza (carriage return) karakter, *Control-m*, ASCII code 13 (CR).

`\t` Horizontális tab karakter, *Control-i*, ASCII code 9 (HT).

`\v` Vertikális tab karakter, *Control-k*, ASCII code 11 (VT).

`\`*nnn* Az oktális *nnn* szám egy, kettő vagy három jegyű. A számjegyek '`0`' és '`7`' közöttiek lehetnek. Például az ASCII ESC (escape) karakter kódja a '`\033`'.

`\x`*hh*... A hexadecimális *hh* szám számjegyei a '`0`'-tól '`9`'-ig és vagy az '`A`'-tól '`F`'-ig vagy az '`a`'-tól '`f`'-ig terjedő karakterek lehetnek. Mint az ANSI C-ben, az escape szekvencia addig tart, ameddig érvényes hexadecimális számjegyek szerepelnek egymás után és az első érvénytelen karakternél véget ér. Ugyanakkor több mint két számjegy használata nem definiált viselkedést vált ki. (A '`\x`' escape szekvencia POSIX `awk`-ban nem megengedett.)

`\/` Erre az escape szekvenciára csak reguláris kifejezéseknél lehet szükség. Olyan reguláris kifejezésnél kell használni, amikor maga a reguláris kifejezés is tartalmaz '`/`' karaktert. Mivel az önmagában álló '`/`' karakter a reguláris kifejezés végét jelölné, ezért kell az escape szekvenciát használni, hogy a reguláris kifejezés további részeit is figyelembe vegye az `awk`.

`\"` Erre az escape szekvenciára csak szövegeknél lehet szükség. Ezt akkor kell használni, ha egy szövegben a macskaköröm karaktert is szeretnénk leírni. Mivel a szövegek macskakörmökkel vannak határolva, az escape szekvencát kell használni ahhoz, hogy az `awk` a szöveg további részét is figyelembe vegye.

A `gawk`-ban van még további két escape szekvencia, aminek speciális jelentése van a reguláris kifejezésekben, Lásd 4.4. Bekezdés [Csak a `gawk` által értelmezett reguláris kifejezés operátorok], 32. oldal.

Mi történik ha egy szövegben olyan karakter elé teszel '\' karaktert amelyik nem szerepel a fenti táblázatban? A POSIX szabvány ezt az esetet tudatosan nem definiálja. Két lehetőség van:

- Eltávolítani a '\' karaktert. Így viselkedik a UNIX `awk` és a `gawk` is. Például, `"a\qc"` ugyanaz mint az `"aqc"`.
- Érintetlenül hagyni a '\' karaktert. Néhány `awk` implementáció ezt a módszert használja. Ebben az esetben a `"a\qc"` megfelel a `"a\\qc"` szövegnek.

Egy reguláris kifejezésben egy '\' karakter minden olyan karakter előtt, amely nem szerepel a fenti táblázatban vagy 4.4. Bekezdés [Csak a `gawk` által értelmezett reguláris kifejezés operátorok], 32. oldal alatt, azt jelenti, hogy a karaktert nem szabad értelmezni mint reguláris kifejezés operátort. Például a `/a\+b/` a '`a+b`' három karakterre illeszkedik.

A teljes hordozhatóság érdekében a '\' karaktert csak akkor használd más karakterek előtt ha szükséges.

Érdekes kérdés lehet annak eldöntése is, hogy ha egy reguláris kifejezésekben használt operátort (metakaraktert) (lásd 4.3. Bekezdés [Reguláris kifejezés operátorok], 26. oldal) oktális vagy hexadecimális escape szekvenciával írunk le, akkor az `awk` mint egyszerű karakter vagy mint reguláris kifejezés operátor fogja azt kezelni.

Történelmileg, az ilyen karakterek mint egyszerű karakterek lesznek kezelve (s.s.). Ugyanakkor a POSIX szabvány azt írja elő, hogy mint reguláris kifejezés metakarakter kell őket kezelni; a `gawk` pontosan így tesz. A `gawk` "compatibility" módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) viszont mint egyszerű karakter kezeli az így leírt escape szekvenciákat, például `/a\52b/` megegyezik a `/a\*b/` mintával.

Összefoglalva:

1. A fenti táblázatban szereplő összes escape szekvencia kerül először feldolgozásra a szövegekben és reguláris kifejezésekben, amint az `awk` beolvasta.
2. A `gawk` feldolgozza a konstans és dinamikus reguláris kifejezéseket (lásd 4.7. Bekezdés [Dinamikus reguláris kifejezések használata], 35. oldal) a speciális operátorok számára (4.4. Bekezdés [Csak a `gawk` által értelmezett reguláris kifejezés operátorok], 32. oldal).
3. Bármely karakter előtt előforduló '\' karakter esetén a karaktert nem kezeli speciálisan, hanem magát a karaktert használja.

4.3. Reguláris kifejezés operátorok

Az egyszerű reguláris kifejezéseket az úgynevezett *reguláris kifejezés operátorokkal* vagy *metakarakterekkel* kombinálva összetettebb és sokoldalúbb mintákat lehet készíteni.

A fent bemutatott escape szekvenciák, 4.2. Bekezdés [Escape szekvenciák], 24. oldal, érvényesek a reguláris kifejezésekben is. A reguláris kifejezések feldolgozása során az `awk` először ezeket a karakter sorozatokat konvertálja az aktuális karakterré.

Itt közöljük a metakarakterek táblázatát. Minden olyan karakter, amely nem escape szekvencia és nem szerepel az alábbi táblázatban egyszerűen önmagát jelenti.

`\` Ezzel a karakterrel elnyomható a követő karakter speciális jelentése a mintában. Például a

```
\$
```

a '`$`' karakterre illeszkedik.

`^` Ez a karakter a sor elejére illeszkedik. Például:

```
^@chapter
```

illeszkedik a '`@chapter`'-re egy sor elején, ami alkalmas a fejezetek kezdetének megtalálására egy Texinfo file-ban. A '`^`' karaktert "*horgonynak*" is szokták nevezni, mivel a használatával a minta csak a sor elejére illeszkedik, oda horgonyozza a mintát.

Fontos tudni, hogy a '`^`' karakter nem illeszkedik egy szövegben elrejtett új sor kezdetére, így az alábbi feltétel hamis:

```
if ("line1\nLINE 2" ~ /^L/) ...
```

`$` Hasonló a '`^`' karakterhez, de a sor végéhez illeszkedik, például:

```
p$
```

illeszkedik azokra a sorokra, amelyek '`p`'-vel végződnek. A '`$`' karakter szintén egy "horgony", és ahogy a '`^`' karakter úgy a '`$`' karakter sem illeszkedik az elrejtett új sor végére, tehát az alábbi feltétel is hamis:

```
if ("line1\nLINE 2" ~ /1$/) ...
```

`.` A pont karakter egyetlen (bármely) karakterre illeszkedik, még az új sor karakterre *is*. Például:

```
.P
```

illeszkedik egy olyan karakterre, amit egy '`P`' követ a szövegben. Összeállíthatunk olyan reguláris kifejezést is, mint '`U.A`', amely illeszkedik bármely három karakterre, ami '`U`' karakterrel kezdődik és az '`A`' karakterrel végződik.

Szigorú POSIX módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), a '`.`' karakter nem illeszkedik a NUL karakterre. Ennek a karakternek minden bitje zérus, egyébként a NUL karakter is olyan, mint bármely más karakter. Léteznek olyan `awk` verziók is, amelyek nem képesek a NUL karaktert keresni.

`[...]` A *karakter lista* illeszkedik bármelyik karakterre, de csak egyre, ami szerepel a szögletes zárójelek között. Például az

```
[MVX]
```

illeszkedik az '`M`', '`V`' vagy '`X`' karakterek bármelyikére a szövegben.

Karakter tartományok esetén a mínusz karaktert kell használni a kezdő- és a végkarakter között, például a

```
[0-9]
```

illeszkedik bármelyik számjegyre. Több tartomány is megadható. Pl. a `[A-Za-z0-9]` listával írhatjuk le az összes "alfanumerikus" karaktert.

Ahhoz, hogy a '`\`', '`]`', '`-`' vagy '`^`' karakterek valamelyikét figyelembe vehessük a szögletes zárójelek között a '`\`' karaktert kell eléjük tenni, például:

```
[d\]]
```

illeszkedik vagy a '`d`' vagy a '`]`' karakterre.

A '`\`' ilyetén használata a karakter listákban kompatíbilis más `awk` implementációkkal, és a POSIX szabvány is a fenti megoldást írja elő. A POSIX által definiált "Kiterjesztett Reguláris Kifejezések" (KRK) csak egy részhalmaza az `awk` reguláris kifejezéseinek. A POSIX "Kiterjesztett Reguláris Kifejezések" a hagyományos `egrep` segédprogram által elfogadott reguláris kifejezéseken alapulnak.

A *karakterosztályok* nem olyan régen lettek bevezetve a POSIX szabványba. A karakterosztály azon karakterlisták speciális leírása, amelyek valamilyen értelemben egyedi tulajdonsággal rendelkeznek, de az aktuális karakterek országonként és/vagy ábécénként eltérhetnek. Például az ábécé betűi mások az USA-ban, mint Franciaországban.

A karakter osztályok csak a karakter listák szögletes zárójelein belül érvényesek. A karakter osztály a '`[:`' karakterekkel kedődik, amit az osztályt leíró kulcsszó és a záró '`:]`' karakterek követnek. A POSIX szabványban definiált karakter osztályok az alábbiak.

`[:alnum:]`
: Alfanumerikus karakterek.

`[:alpha:]`
: Az ábécé betűi.

`[:blank:]`
: Szóköz és tabulátor karakter.

`[:cntrl:]`
: Vezérlőkarakterek.

`[:digit:]`
: Számjegyek.

`[:graph:]`
: Azon karakterek, amelyek nyomtathatóak és láthatóak. (A szóköz karakter nyomtatható, de nem látható, míg az '`a`' karakter mindkettő.)

`[:lower:]`
: Az ábécé kisbetűi.

`[:print:]`
: Nyomtatható karakterek (a nem vezérlőkarakterek).

`[:punct:]`
: Írásjelek (nem az ábécé betűi, nem számjegyek, nem vezérlőkarakterek és nem szóköz, tabulátor).

`[:space:]`
: Szóköz, tabulátor, lapdobás karakterek, hogy csak néhányat említsünk.

`[:upper:]`
: Az ábécé nagybetűi.

`[:xdigit:]`
: Hexadecimális számjegyek.

A POSIX szabvány kiadása előtt az alfanumerikus karakterek kereséséhez az `/[A-Za-z0-9]/` mintát kellett használni. Ha az ábécében más betű is szerepel (pl. ékezetes karakterek) azokat a minta nem veszi észre. A POSIX karakter osztályok használatával csak a `/[[:alnum:]]/` mintát kell leírni, hogy az ábécé összes betűjét felismerje a keresésnél.

Van két további speciális karaktersorozat, ami a karakterlistákon belül előfordulhat. Az egyik azokat a nem ASCII karaktereket reprezentálja, amelyeket csak több mint egy karakterrel írhatunk le (*több karakterrel leírható szimbólumok*), a másik azokat a karaktereket írja le, amelyeknek nem csak egy megjelenési formája van, de pl. egy rendezés során azonosnak kell őket tekinteni. (Például a franciában az egyszerű "e" és az "è" karakterek megegyeznek egy rendezés során.)

Több karakterrel leírható szimbólumok
: Ezeket az elemeket a '`[.`' és a '`.]`' karakterek között kell leírni. Például, ha '`ch`' egy ilyen elem, akkor a `[[.ch.]]` reguláris kifejezés illeszkedni fog az adott szimbólumra, míg a `[ch]` vagy a '`c`' vagy a '`h`' karakterre.

Azonossági osztályok

Az *azonossági osztály* egy speciális név azokra a karakterekre amelyek a helyi ábécé szerint azonosak, de a megjelenési formájuk különböző. A nevet a '`[=`' és a '`=]`' jelek közé kell tenni. Például az '`e`' név jelentheti a "e," "è" és "é." karaktereket. Ebben az esetben a `[[=e=]]` illeszkedhet a '`e`', '`é`' vagy '`è`' karakterek bármelyikére.

Ezek az osztályok a nem angol nyelvterületeken lehetnek nagyon hasznosak.

Figyelem: A `gawk` jelenleg a reguláris kifejezések olyan implementációját használja, amelyik csak a POSIX szabványban rögzített karakter osztályokat ismeri fel, és az utóbbi két osztályt nem.

`[^ ...]` A *komplemens karakter listában* az első karakter a '`[`' után a '`^`' karakter *kell legyen*. A lista illeszkedik minden olyan karakterre, amelyik nem szerepel a szögletes zárójelek között. Például:

```
[^0-9]
```

illeszkedik minden olyan karakterre, amelyik nem számjegy.

`|` A *alternatív operátor* választási lehetőséget ad, például:

```
^P|[0-9]
```

illeszkedik vagy a '`^P`'-re vagy a '`[0-9]`', ami azt jelenti, hogy a minta illeszkedik bármely sorra, ami '`P`' karakterrel kezdődik, vagy tartalmaz egy számjegyet.

Az operátor a lehető legnagyobb reguláris kifejezést használja a karakter két oldaláról, vagy más szavakkal a '`|`' operátornak van a legalacsonyabb precedenciája a reguláris kifejezések között.

`(...)` A zárójelek a reguláris kifejezések csoportosítására alkalmasak, ugyanúgy mint a matematikában. Az alternatív operátor körüli reguláris kifejezés meghatározására is alkalmas, például a '`@(samp|code)\{[^}]+\}`' kifejezés illeszkedik a '`@code{foo}`' és a '`@samp{bar}`' szövegre is.

`*` Ez a szimbólum azt jelenti, hogy a megelőző reguláris kifejezést annyiszor kell ismételni, ha szükséges, amíg egyezést nem talál. Például:

```
ph*
```

esetén a '`*`' szimbólumot a megelőző '`h`' karakterre (mint reguláris kifejezésre) alkalmazva egy olyan mintát keres, amiben szerepel egy '`p`' majd akárhány '`h`' karakter. Ez a minta az önmagában álló '`p`' karakterre is illeszkedik.

A '`*`' szimbólum a lehető *legrövidebb*, megelőző reguláris kifejezésre lesz alkalmazva (hosszabb reguláris kifejezés esetén zárójelet kell használni), és annyi ismétlést próbál találni, amennyi lehetséges. Például:

```
awk '/\(c[ad][ad]*r x\)/ { print }' sample
```

kinyomtat a '`sample`' file-ból minden olyan sort, amely tartalmazza a '`(car x)`', '`(cdr x)`', '`(cadr x)`', stb. kifejezések valamelyikét. Fontos a '`\`' karakter használata a zárójelek előtt.

`+` A '`+`' szimbólum hasonló a '`*`' szimbólumhoz, de a megelőző reguláris kifejezést legalább egyszer meg kell találnia szövegben. Így a

```
wh+y
```

illeszkedik a '`why`' és a '`whhy`' szavakra, de a '`wy`' szóra nem. Ugyanakkor a '`wh*y`' minta illeszkedik mind a három szóra. A fenti példa tehát így írható le egyszerűbben:

```
awk '/\(c[ad]+r x\)/ { print }' sample
```

`?` A szimbólum hasonlít a '`*`' szimbólumra, de a megelőző kifejezés csak egyszer vagy egyszer sem fordulhat elő a szövegben, például:

```
fe?d
```

illeszkedik a '`fed`' és '`fd`' szavakra, de semmi másra nem.

`{`*n*`}`
`{`*n*`,}`
`{`*n*`,`*m*`}` Egy vagy két szám kapcsos zárójelek között írja le az *intervallum kifejezéseket*. Ha csak egy szám szerepel a kapcsos zárójelek között, akkor a megelőző reguláris kifejezést pontosan *n*-szer kell ismételni az illeszkedéshez. Ha két szám szerepel kapcsos zárójelek között, vesszővel elválasztva, akkor a megelőző reguláris kifejezést *n* és *m* közötti alkalommal kell megismételni az illeszkedéshez. Ha csak egy szám és utána egy vessző van kapcsos zárójelek között, akkor legalább *n*-szer kell megismételni a megelőző reguláris kifejezést az illeszkedéshez.

`wh{3}y` illeszkedik a '`whhhy`' szóra, de sem a '`why`' sem a '`whhhhy`' szavakra nem.

`wh{3,5}y` illeszkedik a '`whhhy`', '`whhhhy`' és a '`whhhhhy`' szavak valamelyikére, de másra nem.

`wh{2,}y` illeszkedik a '`whhy`', '`whhhy`', stb. szavakra.

Nem voltak intervallum kifejezések az eredeti `awk`-ban, de a POSIX szabvány azért vezette be őket, hogy az `awk` és az `egrep` következetesen legyenek definiálva.

Mindemellett régi programok használhatják a '`{`' és a '`}`' karaktereket reguláris kifejezésekben, ezért a `gawk` alapesetben *nem* engedi meg intervallum kifejezések használatát a reguláris kifejezésekben. Ha '`--posix`' vagy a '`--re-interval`' parancssori opció (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) definiálva van, intervallum kifejezések megengedettek a reguláris kifejezésekben.

A reguláris kifejezésekben a '`*`', '`+`', '`?`' operátoroknak és a kapcsos zárójeleknek van a legmagasabb precedenciájuk, majd ezt követi az összefűzés operátor. A legalacsonyabb precedenciája a '`|`' operátornak van. Mint a matematikában, a zárójelekkel megváltoztatható az operátorok kiértékelési sorrendje.

A karakterosztályok és intervallum operátorok használata nem engedélyezett ha a `gawk` "compatibility" módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) fut.

A következő bekezdés tárgyalja a GNU specifikus reguláris kifejezés operátorokat, ill. további részleteket ad arról, hogy különböző parancssori opciók esetén a `gawk` hogyan értelmezi a karaktereket a reguláris kifejezésekben.

4.4. Csak a `gawk` által értelmezett reguláris kifejezés operátorok

A reguláris kifejezéseket használó GNU software-ek néhány extra operátort is ismernek, amelyeket ebben a bekezdésben tárgyalunk. Ezek az operátorok jelen esetben csak a `gawk`-ban találhatók meg, és más `awk` implementációkkal nem használhatók.

Ezen operátorok legtöbbje szavak illesztésére alkalmas. Jelen esetben egy *szót* betűk, számjegyek és a '`_`' karakter sorozataként definiálhatunk.

`\w` Ez az operátor illeszkedik a szavakat felépítő bármely karakterre, például betűk, számjegyek és az aláhúzás karakterre. Lényegében az alábbi kifejezés rövidítése `[[:alnum:]_]`.

`\W` Ez az operátor illeszkedik a szavakat fel nem építő karakterekre. Röviden `[^[:alnum:]_]`.

`\<` Ez az operátor illeszkedik egy szó elején előforduló üres karaktersorozatra. Például, az `/\<away/` minta illeszkedik a '`away`' szóra, de nem a '`stowaway`' szóra.

`\>` Ez az operátor illeszkedik egy szó végén előforduló üres karaktersorozatra. Például a `/stow\>/` minta illeszkedik a '`stow`' szóra, de nem a '`stowaway`' szóra.

`\y` Ez az operátor illeszkedik vagy a szó elején vagy a végén elhelyezkedő üres karaktersorozatra. Például a '`\yballs?\y`' illeszkedik vagy a '`ball`' vagy a '`balls`' különálló szóra.

`\B` Ez az operátor illeszkedik a szó közben előforduló üres karakter sorozatra. Más szavakkal, a '`\B`' operátor illeszkedik a szavakat alkotó karakterek közti üres karaktersorozatra. Például, a

/\Brat\B/ minta illeszkedik a '`crate`' szóra, de nem illeszkedik a '`dirty rat`' kifejezésre. A '`\B`' operátor lényegében a '`\y`' operátor ellentéte.

Más GNU software-ekben a szó határoló operátor a '`\b`', de ez összeütközésben van az `awk` nyelv definíciójával, ahol a '`\b`' a törlés (backspace) karakter, így a `gawk` egy másik operátort használ.

Megoldás lenne ha a GNU operátoroknál két '`\`' jelet kellene használni, de ez valószínűleg túl zavaró lenne, ezért használja a `gawk` a '`\y`' operátort, ami talán két rossz megoldás közül a kevésbé rossz.

A különböző parancssori opciók (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) befolyásolják a `gawk` interpretert, hogy hogyan értelmezze a reguláris kifejezésekben előforduló karaktereket:

nincs opció megadva
: Alapesetben a `gawk` biztosítja az POSIX reguláris kifejezés operátorok és a GNU operátorok, 4.3. Bekezdés [Reguláris kifejezés operátorok], 26. oldal, használatát, de intervallum kifejezéseket nem lehet használni.

`--posix`
: Csak a POSIX operátorok engedélyezettek, a GNU operátorokat nem ismeri fel (így a '`\w`' megfelel a '`w`' betűnek). Intervallum operátorokat lehet használni.

`--traditional`
: Hagyományos Unix `awk` reguláris kifejezéseket lehet csak használni. A GNU operátorok, intervallum kifejezések és karakterosztályok (`[[:alnum:]]` és így tovább) nem állnak rendelkezésre. Oktálisan vagy hexadecimálisan leírt karakterek önmagukat jelentik, még akkor is ha reguláris kifejezés metakarakterek lennének normális esetben.

`--re-interval`
: Intervallum kifejezések használatát engedélyezi, még akkor is ha a '`--traditional`' opció is meg lett adva.

4.5. Kis- és nagybetűk az illesztésekben

Lényeges, hogy a reguláris kifejezésekben kisbetűt vagy nagybetűt használunk a normál (nem meta-) karakterek illesztésénél. Így a '`w`' karakter egy reguláris kifejezésben csak a '`w`' kisbetűre fog illeszkedni a '`W`'-re nem.

A legegyszerűbb módja a kisbetűs, nagybetűs írásmódtól független illesztésnek a karakterlisták használata: '`[Ww]`'. Ugyanakkor ez elég kényelmetlen lehet ha sokat kell használni, ráadásul a reguláris kifejezéseket is elbonyolítja. Két lehetőség van a probléma megoldására.

Az első esetben, a program egy adott pontján az adatot átalakítjuk csak kisbetűs vagy csak nagybetűs írásmódba a `tolower` vagy a `toupper` beépített függvényekkel (amiket még eddig nem tárgyaltunk, lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Például:

```
tolower($1) ~ /foo/  { ... }
```

az illesztés előtt az első mező minden karakterét átalakítja kisbetűvé. Ez bármely POSIX kompatíbilis `awk` implementációban működni fog.

A másik módszer csak `gawk`-ban használható, amikor is az `IGNORECASE` beépített változót át kell állítani egy nem zérus értékre. Amikor az `IGNORECASE` változó nem zérus, *minden* reguláris kifejezés és szöveg művelet esetén figyelmen kívül hagyja a kisbetű-nagybetű különbséget. A változó állítása a programon belül befolyásolja, hogy mely részeknél tegyen különbséget a kis- és nagybetűk között. Mivel kezdetben minden változó zérus értéket kap, így az `IGNORECASE` változó is, ezért tesz különbséget a kis- és nagybetűk között alapesetben.

```
x = "aB"
if (x ~ /ab/) ...   # ez feltétel hamis

IGNORECASE = 1
if (x ~ /ab/) ...   # most már sikeres lesz
```

Általában az `IGNORECASE` változó nem használható arra, hogy csak egyes szabályok esetén tegyen különbséget kis- és nagybetűk között és másoknál pedig nem, mivel nincs arra lehetőség hogy csak egy szabályra vagy mintára állítsuk be.[1] A megoldást vagy a karakterlisták vagy a `tolower` beépített függvény adja. Az `IGNORECASE` változót lényegében az összes szabályra érdemes be- vagy kikapcsolni.

Az `IGNORECASE` változó beállítható a parancssorból vagy a `BEGIN` szabályban (lásd 14.2. Bekezdés [Other Command Line Arguments], 165. oldal; és lásd 8.1.5.1. Bekezdés [Kezdő és lezáró tevékenységek], 100. oldal) Az `IGNORECASE` változó parancssorból történő beállítása esetén a program szerkesztése nélkül biztosítható, hogy a program nem fog különbséget tenni a kis- és nagybetűk között.

A `gawk` 3.0-ás verziója előtt az `IGNORECASE` változó értéke csak a reguláris kifejezésekkel végzett műveleteket érintette, és nem volt hatással a '`==`', '`!=`' és hozzájuk hasonló szövegösszehasonlító operátorokra. A 3.0-ás verziótól kezdve mind a reguláris kifejezésekkel, mind a szövegekkel végzett összehasonlítás esetén az `IGNORECASE` változó értéke hatással van a végeredményre.

A `gawk` 3.0-ás verziójától kezdve a kis- és nagybetűs karakterek azonossága az ISO-8859-1 (ISO Latin-1) karakterkészlet alapján dől el. Az eredeti 128 ASCII karakterkészlet csak egy részhalmaza ennek a karak-

[1] Azért erre is van lehetőség:

```
IGNORECASE=1 && /foObAr/ { .... }
IGNORECASE=0 || /foobar/ { .... }
```

de ez nagyon csúnya programozási megoldás, ezért kerülni kell.

terkészletnek, és az ISO-8859-1 segítségével több európai nyelv karakterei is használhatók.

Az IGNORECASE változó értéke nem számít ha a gawk "compatibility" módban fut (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). Ebben az esetben a kis- és nagybetűk közötti különbség mindig fontos.

4.6. Mennyi szöveg illeszkedik?

Vegyük a következő példát:

```
echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
```

Ez a példa a sub függvényt használja, (amit eddig még nem tárgyaltunk, lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal) hogy a bemeneti soron módosítást végezzen. A /a+/ reguláris kifejezés "egy vagy több 'a' karakterre illeszkedik" és amire lecserélnénk az az '<A>' szöveg.

A bemenet négy 'a' karaktert tartalmaz. Mi lesz az eredmény? Más szavakkal, mennyi az "egy vagy több" – az awk kettő, három vagy négy 'a' karakterre fog illeszteni?

A válasz az, hogy az awk (és a POSIX is) mindig a lehető *leghosszabb* bemeneti karaktersorozatot fogja illeszteni. Így, ebben a példában, elsőre mind a négy karakterre illeszkedik a minta és egyszerre cseréli le a '<A>' kifejezésre.

```
$ echo aaaabcd | awk '{ sub(/a+/, "<A>"); print }'
⊣ <A>bcd
```

Egyszerű illeszkedik/nem illeszkedik teszteknél ez nem olyan fontos, de amikor regexp alapú mező és rekord feldarabolást (lásd 5.1. Bekezdés [Hogyan történik a feldarabolás rekordokra], 37. oldal és lásd 5.5. Bekezdés [Hogyan történik a mezőelválasztás], 44. oldal) kell csinálni és a match, sub, gsub és a gensub függvényeket kell használni, akkor ez nagyon fontossá válhat.

4.7. Dinamikus reguláris kifejezések használata

A '~' vagy '!~' operátorok esetén nem kötelező hogy a jobb oldalon egy reguláris kifejezés konstans álljon (valamilyen szöveg '/' jelek között), hanem bármilyen kifejezés állhat ott. Az awk a kifejezést kiértékeli és ha szükséges átalakítja szöveggé; a szöveg tartalmát használja, mint reguláris kifejezés. Az ilyen reguláris kifejezéseket *dinamikus reguláris kifejezésnek* hívják. Például:

```
BEGIN { identifier_regexp = "[A-Za-z_][A-Za-z_0-9]+" }
$0 ~ identifier_regexp    { print }
```

beállítja a identifier_regexp változót egy olyan reguláris kifejezésre, ami az awk változók neveire illeszkedik, majd megpróbálja illeszteni a bemeneti adatokra.

Figyelem: Amikor a '~' és a '!~' operátorokat használod jelentős különbség van a '/' jelek között megadott regexp konstansok és a macskakörmök között megadott szöveg konstansok között. Amikor a szöveg konstanst használod, tudnod kell, hogy a szöveg *kétszer* lesz feldolgozva; egyszer amikor az `awk` beolvassa a programodat, másodszor amikor a bal oldalon levő szöveget megpróbálja illeszteni a jobb oldali mintához. Ez igaz minden szöveg tartalmú kifejezésre (így a fenti `identifier_regexp` változóra is) és nem csak a szöveg konstansokra.

Miért fontos, hogy a szöveg kétszer lesz feldolgozva ? A válasz az escape szekvenciákat érinti és különösen a '\' karaktert. Ahhoz, hogy a '\' karaktert egy reguláris kifejezésben használhassuk mint egyszerű karaktert, a '\' karaktert kétszer kell leírni.

Például a `/\*/` kifejezés egy olyan regexp konstans ami megfelel az egyszerű '*' karakternek. Csak egy '\' karakter kell! Ugyanezt megcsinálni egy szöveg konstans esetén így lehet, `"\\*"`. Az első '\' védi a másodikat és valójában csak két karakter van a szövegben a '\' és a '*' karakterek.

Tehát ha regexp és szöveg konstansok is használhatók reguláris kifejezések leírására, melyiket használjuk? A "regexp konstansokat", több okból is.

1. A szöveg konstansokat bonyolultabb írni és nehezebb olvasni is. Ha regexp konstansokat használsz kevesebbet hibázhatsz. Gyakori hiba, hogy az emberek nem értik a két konstans közötti különbséget.
2. A regexp konstansok használata hatékonyabb is: az `awk` a reguláris kifejezéseket egy belső struktúrában tárolja, így a mintaillesztés gyorsabb. Amikor szöveg konstanst használsz, az `awk`-nak először át kell alakítania a kifejezést ebbe a belső struktúrába, és csak utána tud mintaillesztést csinálni.
3. A regexp-ek használata egy szebb programozási stílus; világosan mutatja, hogy egy reguláris kifejezés illesztést akartál csinálni.

5. Bemeneti file-ok olvasása

A tipikus `awk` programokban a bemenet vagy a standard bemenet (általában a billentyűzet, de gyakran egy cső (pipe) valamilyen másik parancsból), vagy file-ok, amelyek nevét az `awk` parancssorában kell megadni. Az `awk` a bemeneti file-okat sorrendben olvassa be, és előbb befejezi az egyiket, mielőtt elkezdené a következőt. Az éppen aktuális file nevét a `FILENAME` (lásd 10. Fejezet [Beépített változók], 115. oldal) beépített változóból lehet kiolvasni.

Az `awk` a bemenetet *rekordokban* olvassa be és dolgozza fel a programodban megadott szabályok szerint, egyszerre csak egyet. Alapesetben egy rekord egy sornak felel meg. Ezenkívül automatikusan minden rekordot feldarabol úgynevezett *mezőkre*, ezzel kényelmesebbé téve a rekord részeinek elérését a programod számára.

Ritkán, de előfordulhat, hogy a `getline` parancsot kell használnod. A `getline` parancs nagyon fontos része a nyelvnek, mivel adatot tud beolvasni bármilyen és bármennyi file-ból, ráadásul ezeket a file-okat nem kötelező megadni az `awk` parancssorában (lásd 5.8. Bekezdés [Explicit beolvasás `getline`-al], 53. oldal).

5.1. Hogyan történik a feldarabolás rekordokra

Az `awk` segédprogram a bemenetet rekordokra és mezőkre darabolja fel a programod számára. A rekordok egy úgynevezett *rekordelválasztóval* vannak elválasztva egymástól. Alapesetben a rekordelválasztó az új sor karakter. Ezért van az, hogy alapesetben egy sor megfelel egy rekordnak. Más karakter is megadható mint rekordelválasztó, ha az `RS` beépített változót beállítjuk a kívánt karakterre.

Az `RS` értékének megváltoztatása ugyanúgy történik mint bármilyen más változóé, az értékadó operátorral, '`=`' (lásd 7.7. Bekezdés [Értékadó kifejezések], 84. oldal). Az új rekordelválasztót idézőjelek közé kell tenni, mint egy szöveg konstanst. Általában az értékadást a legjobb azelőtt végrehajtani mielőtt bármilyen adatot a program feldolgozna, így minden adat a megfelelő elválasztó karakterrel lesz kezelve. Ehhez a `BEGIN` (lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal) szabályt kell használni. Például:

```
awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
```

megváltoztatja az `RS` értékét a `"/"` karakterre mielőtt bármilyen adatot beolvasna, így a rekordelválasztó karakter a `"/"` lesz. Ezután elkezdi olvasni a file tartalmát és az `awk` program második szabályát alkalmazza (tevékenységet minta nélkül), ami kinyomtat minden rekordot. Mivel a `print` kifejezés minden kinyomtatott rekordhoz egy új sort ad, a program lényegében a bemenetet a kimenetre másolja át úgy, mintha minden `"/"` karaktert lecserélnénk egy új sor karakterre. Az eredmény a '`BBS-list`' file-al:

```
$ awk 'BEGIN { RS = "/" } ; { print $0 }' BBS-list
⊣ aardvark     555-5553     1200
⊣ 300          B
⊣ alpo-net     555-3412     2400
⊣ 1200
⊣ 300     A
⊣ barfly       555-7685     1200
⊣ 300          A
⊣ bites        555-1675     2400
⊣ 1200
⊣ 300     A
⊣ camelot      555-0542     300               C
⊣ core         555-2912     1200
⊣ 300          C
⊣ fooey        555-1234     2400
⊣ 1200
⊣ 300     B
⊣ foot         555-6699     1200
⊣ 300          B
⊣ macfoo       555-6480     1200
⊣ 300          A
⊣ sdace        555-3430     2400
⊣ 1200
⊣ 300     A
⊣ sabafoo      555-2127     1200
⊣ 300          C
⊣
```

Érdemes megfigyelni, hogy a '`camelot`' kezdetű sor nem lett feldarabolva. Az eredeti file-ban (lásd 1.3. Bekezdés [Adat file-ok a példákhoz], 7. oldal) a sor így néz ki:

```
camelot      555-0542     300               C
```

Csak egyetlen baud érték van megadva és nincs `"/"` karakter a sorban.

Egy másik lehetőség a rekordelválasztó megváltoztatására a parancssor használata (lásd 14.2. Bekezdés [Other Command Line Arguments], 165. oldal).

```
awk '{ print $0 }' RS="/" BBS-list
```

Ez a parancssor beállítja az `RS` változót a '`/`' karakterre, mielőtt elkezdené a '`BBS-list`' file feldolgozását.

A speciális karakter, mint a '`/`' karakter használata az esetek legnagyobb részében nem okoz problémát, de a következő speciális parancssor csak egy meglepő '`1`'-est nyomtat ki. Az `NF` beépített változó értéke az aktuális rekordon belüli mezők számát adja meg. Jelen esetben egyetlen mező van, ami az új sor karaktert tartalmazza.

```
$ echo | awk 'BEGIN { RS = "a" } ; { print NF }'
⊣ 1
```

Amint eléri a bemenet végét a jelenlegi rekordot lezárja, még akkor is ha az utolsó karakter a file-ban nem a rekordelválasztó volt (s.s.).

Az üres szövegnek, "" (szöveg, amiben nincs karakter), mint az RS értéke, speciális jelentése van: azt jelenti, hogy a rekordok egy vagy több üres sorral vannak elválasztva és semmi mással. Lásd 5.7. Bekezdés [Több sorból álló rekordok], 50. oldal, további részletekért.

Ha az RS értékét az awk futása közben változtatod meg, akkor az új értéket csak az új rekord beolvasásától kezdve veszi figyelembe, az éppen aktuális (és az előzőleg feldolgozott) rekordokat nem befolyásolja.

Miután megtalálta a rekord végét, a gawk az RT változót beállítja arra a karakterre/szövegre, ami illeszkedett az RS rekordelválasztóra.

Valójában az RS értéke nem csak egy karakter lehet, hanem bármilyen reguláris kifejezés (lásd 4. Fejezet [Reguláris kifejezések], 23. oldal). Általában egy rekord a megadott reguláris kifejezés kezdeténél végződik; a következő rekord az illeszkedő reguláris kifejezés végénél kezdődik. Ez a szabály érvényesül alapesetben is, amikor az RS az új sor karakter: a rekord a következő illeszkedő reguláris kifejezésnél ér véget (az új sor karakternél), a következő rekord pedig a reguláris kifejezés végénél kezdődik (vagyis a következő sor első karakterénél). Mivel az új sor karakter illeszkedik az RS-re ezért egyik rekordnak sem része.

Amikor az RS értéke csak egy karakter, akkor az RT is ugyanazt a karaktert fogja tartalmazni. Ugyanakkor, ha az RS értéke egy reguláris kifejezés az RT sokkal hasznosabb lehet; azt az aktuális szövegrészletet tartalmazza, ami illeszkedett a reguláris kifejezésre.

A fentieket az alábbi példa illusztrálja, ahol az RS egy olyan reguláris kifejezés amelyik vagy az új sor karakterre vagy egy olyan szövegre illeszkedik, amely egy vagy több nagybetűt tartalmaz, ill. előtte és/vagy mögötte egy szóköz karakter lehet (lásd 4. Fejezet [Reguláris kifejezések], 23. oldal).

```
$ echo record 1 AAAA record 2 BBBB record 3 |
> gawk 'BEGIN { RS = "\n|( *[[:upper:]]+ *)" }
>             { print "Record =", $0, "and RT =", RT }'
⊣ Record = record 1 and RT =  AAAA
⊣ Record = record 2 and RT =  BBBB
⊣ Record = record 3 and RT =
⊣
```

Az utolsó sor egy üres sor. Ez azért van, mert az utolsó RT értéke egy új sor és a print kifejezés mindig hozzáad egy lezáró új sor karaktert a kimenethez.

Lásd 16.2.8. Bekezdés [A Simple Stream Editor], 241. oldal, ahol további példák találhatók az RS és RT használatára.

Az RS használata mint reguláris kifejezés és az RT változó az awk nyelv gawk kiegészítései; "compatibility" módban nem használhatók (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). "Compatibility" módban az `RS`-nek csak az első karakterét használja a rekord végének megállapítására.

Az `awk` segédprogram azt is számontartja, hogy eddig hány darab rekordot olvasott be az aktuális bemeneti file-ból. Ezt az értéket az `FNR` beépített változóban lehet megtalálni. Amikor egy új file-t kezd el olvasni a változó értéke mindig lenullázódik. Egy másik beépített változó, az `NR`, az összes file-ból az összes eddig beolvasott rekordok számát tárolja. Kezdeti értéke zérus és soha nem nullázódik le automatikusan.

5.2. Mezők elérése

A beolvasott rekordot az `awk` automatikusan feldarabolja *mezőkre*. Alapesetben a mezőket szóközök vagy tab vagy új sor karakterek[1] választják el, mint a szavakat egy mondatban; hasonló karakterek, mint lapdobás (formfeed) *nem* szolgálnak elválasztóként az `awk`-ban.

A mezők célja, hogy kényelmesebbé tegyék számodra a rekordok feldolgozását. Nem kötelező őket használni – dolgozhatsz csak a rekorddal – de a mezők teszik az `awk` programokat igazán hasznossá.

Az `awk` programban egy mezőre egy dollár jellel, '`$`', és az utána következő számmal lehet hivatkozni. Így, a `$1` az első, a `$2` a második mezőre utal. Vegyük a következő sort:

```
This seems like a pretty nice example.
```

Itt az első mező vagy `$1` a '`This`'; a második mező vagy `$2` a '`seems`' és így tovább. Az utolsó mező vagy `$7` az '`example.`'. Mivel nincs szóköz a utolsó '`e`' betű és a lezáró '`.`' pont között ezért a pont a mező része lesz.

Az `NF` beépített változó adja meg, hogy az aktuális rekordban hány mező van. Az `awk` automatikusan frissíti az `NF` értékét minden új rekord beolvasásánál.

Akárhány mező van a rekordban, az utolsó rekordra a `$NF`-el is lehet hivatkozni. Így a fenti példában az `$NF` ugyanaz lenne mint a `$7`, ami az '`example.`'. Hogy ez miért működik, azt egy kicsit később magyarázzuk el. Ha az utolsó utáni mezőre hivatkozol, például a `$8`-ra amikor csak hét mező van a rekordban, egy üres szöveget kapsz eredményül.

A `$0` egy speciális eset, a teljes rekordot reprezentálja. `$0`-t használhatod ha a mezőkkel nem akarsz foglalkozni.

Még egy példa:

```
$ awk '$1 ~ /foo/ { print $0 }' BBS-list
⊣ fooey        555-1234     2400/1200/300     B
⊣ foot         555-6699     1200/300          B
⊣ macfoo       555-6480     1200/300          A
⊣ sabafoo      555-2127     1200/300          C
```

[1] POSIX `awk`-ban az új sor karakter nem mezőelválasztó

Ez a példa minden olyan rekordot kinyomtat a '`BBS-list`' file-ból, amelynek az első mezőjében előfordul a '`foo`' szó. A '`~`' operátor az *illesztő operátor* (lásd 4.1. Bekezdés [Hogyan használjuk a reguláris kifejezéseket], 23. oldal); azt ellenőrzi, hogy a megadott kifejezés (itt a `$1` mező) illeszkedik-e a reguláris kifejezésre.

Ezzel ellentétben, a következő példa a '`foo`' szót keresi *a teljes rekordban* és csak az első és az utolsó mezőt nyomtatja ki az illeszkedő rekordoknál.

```
$ awk '/foo/ { print $1, $NF }' BBS-list
⊣ fooey B
⊣ foot B
⊣ macfoo A
⊣ sabafoo C
```

5.3. Nem konstans mezőazonosító számok

A mezőazonosító szám nem csak konstans lehet. Az `awk` nyelvben a '`$`' karakter után bármilyen kifejezés állhat. A kifejezés értéke adja meg a mező számát. Ha kifejezés értéke szöveg, akkor azt átalakítja számmá. Például:

```
awk '{ print $NR }'
```

Ha emlékszem, akkor az `NR` az eddig beolvasott rekordok számát tartalmazza; az első rekordnál egy, a másodiknál kettő, és így tovább, az értéke. Így ez a példa kinyomtatja az első mezőt az első rekordnál, a második mezőt a második rekordnál, és így tovább. A huszadik rekordnál a huszadik mezőt nyomtatja ki; de mivel valószínűleg a rekordban nincs 20 mező ezért csak egy üres sort fog kinyomtatni.

Itt egy másik példa, ahol egy kifejezést használunk a mezőazonosító számnak:

```
awk '{ print $(2*2) }' BBS-list
```

Az `awk` először kiértékeli a '`(2*2)`' kifejezést, majd az eredményül kapott számot használja a mező azonosítására. A '`*`' jel szorzást jelent, így a '`2*2`' kifejezés értéke négy lesz. A zárójelek azért kellenek, hogy előbb a szorzás hajtódjon végre és csak utána a mező azonosítás; zárójelek mindig kellenek, ha matematikai műveletet használunk a mezőazonosító szám előállítására. Végül is ez a példa a '`BBS-list`' file minden sorából a negyedik mezőt fogja kinyomtatni. (Az `awk` nyelv operátorainak a precedencia listája, csökkenő sorrendben a 7.14. Bekezdés [Operátorok precedenciája (Hogyan ágyazhatók operátorok egymásba)], 94. oldal alatt található meg.)

Ha a mezőazonosító szám a kiértékelés után zérus lesz, akkor a teljes rekordot kapod eredményül. Így a `$(2-2)` kifejezés ugyanaz mint a `$0`. Negatív számok nem megengedettek mezőazonosítóként; ha mégis előfordulna, akkor valószínűleg az `awk` program leáll. (A POSIX szabvány nem definiálja a viselkedést negatív szám esetére. A `gawk` leállítja a programot negatív szám esetén, más `awk` implementációk másképp viselkedhetnek.)

Ahogy azt korábban elmondtuk, 5.2. Bekezdés [Mezők elérése], 40. oldal, az `NF` beépített változó (lásd 10. Fejezet [Beépített változók], 115. oldal) a mezők számát adja meg az aktuális rekordban. Így a `$NF` kifejezés nem egy speciális kifejezés, csak a közvetlen következménye az `NF` használatának, mint mezőazonosító szám.

5.4. Mező tartalmának megváltoztatása

Egy mező tartalmát meg is változtathatod a programon belül. (Bár ez a bemenetet megváltoztatja az `awk` számára, valójában a tényleges bemenet változatlan; az `awk` *soha* nem módosítja a bemeneti file-t.)

Vegyük a következő példát és a kimenetét:

```
$ awk '{ $3 = $2 - 10; print $2, $3 }' inventory-shipped
⊣ 13 3
⊣ 15 5
⊣ 15 5
...
```

A '`-`' jel a kivonás, így ez a program a harmadik mezőnek, `$3`, új értéket ad, a második mezőből tizet vonva ki, '`$2 - 10`'. (Lásd 7.5. Bekezdés [Matematikai operátorok], 82. oldal.) Ezután a második és a harmadik mezőt kinyomtatja az új értékkel.

Ahhoz, hogy ez működjön, a második mezőnek, `$2`, olyan szöveget kell tartalmaznia, ami egy értelmes szám; a szöveget átalakítja számmá, hogy a matematikai műveletet elvégezhesse. A kivonás eredményét átalakítja szöveggé, amit végül hozzárendel a harmadik mezőhöz. Lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal.

Amikor egy mező tartalmát megváltoztatod, az `awk` a rekord szövegét újra kiértékeli, hogy a rekord tükrözze a változást. Így a `$0` a megváltozott mezőt fogja tartalmazni. A következő program a bemeneti file-t nyomtatja ki úgy, hogy minden sorban a második mező értékéből tizet kivon.

```
$ awk '{ $2 = $2 - 10; print $0 }' inventory-shipped
⊣ Jan 3 25 15 115
⊣ Feb 5 32 24 226
⊣ Mar 5 24 34 228
...
```

Olyan mezőkhöz is rendelhető tartalom, amelyek nem részei a rekordnak. Például:

```
$ awk '{ $6 = ($5 + $4 + $3 + $2)
>        print $6 }' inventory-shipped
⊣ 168
⊣ 297
⊣ 301
...
```

A `$6` nem létezik a rekordban, mi hoztuk létre és a `$2`, `$3`, `$4` és a `$5` mezők tartalmának összegével töltöttük fel. A '`+`' jel összeadást jelent. Az '`inventory-shipped`' file esetén a `$6` mező az egy hónapban elküldött összes csomag számát jelenti.

Egy új mező létrehozása esetén, a bemeneti rekord belső reprezentációja is megváltozik – a `$0` értéke. Így a mező létrehozása után egy '`print $0`' kifejezés kinyomtatja a rekordot az új mezővel együtt, a megfelelő mezőelválasztót használva.

A rekord új kiértékelése befolyásolni fogja az `NF` értékét (a mezők száma; lásd 5.2. Bekezdés [Mezők elérése], 40. oldal), ugyanakkor az `NF` és az eddig nem tárgyalt *kimeneti mezőelválasztó*, `OFS` (lásd 6.3. Bekezdés [Kimeneti elválasztó], 63. oldal) is befolyásolva lesz a kiértékelés által. Például az általad létrehozott legnagyobb mezőazonosító számot fogja az `NF` tartalmazni.

Ugyanakkor csak egy egyszerű hivatkozás egy olyan mezőre, ami nem szerepel a rekordban, nem fogja megváltoztatni sem a `$0` sem az `NF` értékét. A hivatkozás egy üres szöveget fog generálni, például:

```
if ($(NF+1) != "")
    print "can't happen"
else
    print "everything is normal"
```

program részlet az '`everything is normal`' szöveget fogja kinyomtatni, mivel a `NF+1`-ik mező természetesen nem szerepel a rekordban. (Az `awk if-else` kifejezésről további információ a lásd 9.1. Bekezdés [Az `if-else` kifejezés], 105. oldal alatt található. A '`!=`' operátorról pedig a 7.10. Bekezdés [Variable Typing and Comparison Expressions], 88. oldal ad további információt.)

Fontos megjegyezni, hogy egy létező mező tartalmának megváltoztatása befolyásolni fogja a `$0` értékét, de nem fogja megváltoztatni `NF` értékét, még akkor sem ha a mező új értéké az üres szöveg. Például:

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""
>                         print $0; print NF }'
⊣ a::c:d
⊣ 4
```

A mező még mindig ott van; csak éppen üres, amit a két egymást követő kettőspont is jelez.

Ez a példa bemutatja, hogy mi történik, ha egy új mezőt hozunk létre.

```
$ echo a b c d | awk '{ OFS = ":"; $2 = ""; $6 = "new"
>                         print $0; print NF }'
⊣ a::c:d::new
⊣ 6
```

A közbenső, `$5` mező is létrejön egy üres szöveggel (a második dupla kettőspont mutatja) és az `NF` értékét hatra állítja.

Végül, ha az `NF` értékét csökkentjük, akkor mező(ke)t dobunk el és a `$0` új értéket kap a kiértékelés után. Például:

```
$ echo a b c d e f | ../gawk '{ print "NF =", NF;
>                              NF = 3; print $0 }'
⊣ NF = 6
⊣ a b c
```

5.5. Hogyan történik a mezőelválasztás

Ez a fejezet egy kicsit hosszabb lesz, mivel az `awk` egyik alapvető működési elvét magyarázza el.

5.5.1. A mezőelválasztás alapjai

A *mezőelválasztó*, vagy egy karakter vagy egy reguláris kifejezés, adja meg, hogy az `awk` hogyan darabolja fel a bemeneti rekordot mezőkre. Az `awk` a mezőelválasztóra illeszkedő karaktersorozatokat keres a bemeneti rekordban és a mezők azok a szövegek lesznek amelyek az illeszkedő részek között helyezkednek el.

Az alábbi példákban a szóköz helyett a "•" jelet fogjuk használni a kimenetben.

Ha a mezőelválasztó az '`oo`', akkor az alábbi sor:

```
moo goo gai pan
```

az '`m`', '`•g`' és a '`•gai•pan`' mezőkre lesz feldarabolva. A második és a harmadik mező előtti szóköz is a mező része lesz.

A mezőelválasztót az `FS` beépített változó tartalmazza. Shell programozók figyelem! Az `awk` *nem* használja az `IFS` nevet, amit a POSIX szabványos shell-ek használnak (Bourne shell, `sh`, vagy a GNU Bourne-Again Shell, Bash).

Egy `awk` programon belül az `FS` értékét az '`=`' értékadó operátorral változtathatod meg (lásd 7.7. Bekezdés [Értékadó kifejezések], 84. oldal). A legjobb alkalom erre a program eleje, mielőtt bármilyen adatot a program beolvasna, így a legelső rekord is a megfelelő mezőelválasztóval lesz feldolgozva. Például a `BEGIN` minta (lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal) használatával teheted ezt meg. Az alábbi példában az `FS` értéke a `","` lesz:

```
awk 'BEGIN { FS = "," } ; { print $2 }'
```

és ha a bemeneti sor:

```
John Q. Smith, 29 Oak St., Walamazoo, MI 42139
```

akkor az `awk` program a '`•29•Oak•St.`' szöveget fogja kinyomtatni.

Előfordul, hogy az adatod olyan helyen is tartalmaz elválasztó karaktert, ahol azt nem várnád. Például személy nevek esetén a fenti példa sorban tudományos cím vagy egyéb adat is megadható, mint '`John Q. Smith, LXIX`'. Tehát:

```
John Q. Smith, LXIX, 29 Oak St., Walamazoo, MI 42139
```

sor esetén a program a '`•LXIX`'-et fogja kinyomtatni és nem a '`•29•Oak•St.`'. Ha a lakcímeket szeretted volna kigyűjteni, természetesen meglepődnél. A tanulság az, hogy az adatstruktúrát és az elválasztó karaktereket gondosan kell megválasztani, hogy az ilyen problémák elkerülhetők legyenek.

Amint azt tudod, alapesetben a mezőket szóközök, tab és új sor karakterek választják el egymástól; nem csak egy szóköz: két szóköz egymás után nem generál egy üres mezőt. Az `FS` alapértéke a `" "`, egy szóközt tartalmazó szöveg. Ha ezt a normális módon értelmeznénk, minden szóköz egy mezőt választana el, így két szóköz egymás után egy üres mezőt generálna. Az ok amiért nem ez történik az, hogy egyetlen szóköz mint az `FS` értéke egy speciális eset.

Ha az `FS` bármilyen más karaktert tartalmaz, pl. a `","`, akkor a karakter minden előfordulása két mezőt választ el egymástól. Két egymás utáni megjelenése egy üres mezőt határol. Ha egy rekord elején vagy végén fordul elő, az is üres mezőt jelent. A szóköz karakter az egyetlen kivétel, ami nem követi ezt a szabályt.

5.5.2. Reguláris kifejezések mint mezőelválasztók

Az előző alfejezet bemutatta egy karakter használatát mint mezőelválasztó. Általánosítva, az `FS` értéke bármilyen reguláris kifejezés lehet. Ebben az esetben, minden illeszkedés a rekordon belül két mezőt választ el egymástól. Például:

```
FS = ", \t"
```

esetén minden olyan szöveg, ami egy vesszőből, egy szóköz és egy tab karakterből áll, elválaszt két mezőt. ('`\t`' egy escape szekvencia (lásd 4.2. Bekezdés [Escape szekvenciák], 24. oldal), ami a tab karaktert helyettesíti.)

Egy kicsit bonyolultabb példa: tegyük fel, hogy a szóköz karaktert ugyanúgy szeretnéd használni, mint más karaktereket a mezők elválasztására. Ebben az esetben az `FS`-t a `"[ ]"` reguláris kifejezésre kell beállítani (nyitó szögletes zárójel, szóköz, záró szögletes zárójel). Ez a reguláris kifejezés csak egy szóközre illeszkedik és semmi másra (lásd 4. Fejezet [Reguláris kifejezések], 23. oldal).

Van egy fontos különbség a '`FS = " "`' és a '`FS = "[ \t\n]+"`' kifejezések között. Mindkét esetben a mezőket egy vagy több szóköz, tab és/vagy új sor karakter választja el, de amikor az `FS` értéke a `" "`, az `awk` először levágja a kezdő és záró szóközöket, tab és új sor karaktereket és csak utána kezdi el feldolgozni a rekordot.

Az alábbi példa csak egy '`b`'-t fog kinyomtatni:

```
$ echo ' a b c d ' | awk '{ print $2 }'
⊣ b
```

de ez egy '`a`'-t nyomtat ki (extra szóközök vannak a betűk körül):

```
$ echo ' a  b  c  d ' | awk 'BEGIN { FS = "[ \t]+" }
>                                 { print $2 }'
⊣ a
```

Ebben az esetben az első mező *üres*.

A kezdő és záró szóköz, tab és új sor karakterek levágása a `$0` új kiértékelésénél is fontos szerepet játszik, így:

```
$ echo '   a b c d' | awk '{ print; $2 = $2; print }'
⊣    a b c d
⊣ a b c d
```

Az első `print` kifejezés kinyomtatja az eredeti rekordot. A `$2` mező értékadása után újraértékeli a `$0` tartalmát; a `$1`-tól az `$NF`-ig a mezőket összefűzi az `OFS` tartalmát használva elválasztásra. Mivel a kezdő és záró szóköz karaktereket nem vette figyelembe a `$1` megállapításánál, így azok most sem részei az új `$0`-nak. Végül az utolsó `print` kifejezés kiírja az `$0` új tartalmát.

5.5.3. Minden karakter egy mező

Előfordulhat, hogy egy rekord minden karakterét meg szeretnéd vizsgálni külön-külön. `gawk`-ban ez egyszerű, egy üres szöveget (`""`) kell az `FS`-hez rendelni. Ebben az esetben minden karakter egy önálló mező lesz:

```
$ echo a b | gawk 'BEGIN { FS = "" }
>                 {
>                     for (i = 1; i <= NF; i = i + 1)
>                         print "Field", i, "is", $i
>                 }'
⊣ Field 1 is a
⊣ Field 2 is
⊣ Field 3 is b
```

Hagyományosan, ha az `FS` értéke `""`, akkor az `awk` viselkedése nem definiált. Ebben az esetben a Unix `awk` az egész rekordot egy mezőnek tekinti (s.s.). "Compatibility" módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), ha az `FS` értéke `""` a `gawk` is így viselkedik.

5.5.4. Az `FS` beállítása parancssorból

Az `FS` értéke a parancssorból is beállítható, a '`-F`' opció használatával:

`awk -F, '`*program*`'` *input-files*

hatására az `FS` értéke a '`,`' karakter lesz. Fontos észrevenni, hogy az opciót a nagy '`F`' betűvel lehet megadni, míg a kis '`f`' betű az `awk` programot tartalmazó file-t adja meg. A '`-F`' és '`-f`' -nek semmi köze egymáshoz, a kis- és nagybetű megkülönböztetés fontos. Természetesen a két opciót használhatod egyszerre.

A '-F' utáni argumentum feldolgozása ugyanúgy történik mintha az FS-t a programban állítanánk be. Ez azt jelenti, hogy ha a mezőelválasztó egy speciális karaktert tartalmaz, akkor azt megfelelően védeni kell a '\' karakterrel. Például ha a mezőelválasztó egy '\' karakter, akkor ezt kell begépelni:

```
# ugyanaz mint FS = "\\"
awk -F\\\\ '...' files ...
```

Mivel a '\' karakter a shell számára is speciális karakter, ezért az awk csak a '-F\\' kifejezést fogja megkapni. Ezután az awk feldolgozza a '\\' escape szekvenciát (lásd 4.2. Bekezdés [Escape szekvenciák], 24. oldal), ami végül a '\' jelet adja meg mint mezőelválasztó.

Egy speciális eset, ha "compatibility" módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) az '-F' után csak egy 't' betű áll. Ekkor az FS valójában a tab karaktert kapja értékül. Ez azért van, mert ha a '-F\t' gépeled be idézőjelek nélkül, akkor a '\' jelet a shell eldobja és az awk azt gondolja, hogy a tab karaktert akartad megadni és nem a 't' betűt, mint mezőelválasztó. Ha tényleg csak a 't' betűvel akarod elválasztani a mezőket, akkor a '-v FS="t"' kifejezést kell használni (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

Például készítsünk egy 'baud.awk' nevű file-t, ami a /300/ mintát és a 'print $1' tevékenységet tartalmazza:

```
/300/   { print $1 }
```

Állítsuk be az FS-t a '-' karakterre, majd futtassuk a programot a 'BBS-list' file-al. Az alábbi parancs kilistázza azoknak a gépeknek a nevét és a telefonszámuk első három jegyét, amelyek 300 baud-al működnek:

```
$ awk -F- -f baud.awk BBS-list
⊣ aardvark     555
⊣ alpo
⊣ barfly       555
...
```

A második sor nem egészen tökéletes. Az eredeti file-ban (lásd 1.3. Bekezdés [Adat file-ok a példákhoz], 7. oldal) így nézett ki:

```
alpo-net     555-3412     2400/1200/300     A
```

A '-' karakter szerepel a rendszer nevében, így nem a telefonszámot írja ki, ahogy azt szeretnénk. Ez is mutatja mennyire fontos, hogy gondosan válasszuk meg a mezőket és a mezőelválasztókat.

A Unix rendszereken a jelszó (passwd) file-ban minden felhasználóhoz tartozik egy bejegyzés (egy sor). A mezők kettősponttal vannak elválasztva. Az első mező a bejelentkezési név, a második a felhasználó jelszava. (A legtöbb rendszeren ma már nem elérhető a jelszó a felhasználók számára.) Egy bejegyzés így nézhet ki:

```
arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
```

Az alábbi program végignézi a jelszó file-t és kilistázza azokat a felhasználókat akiknél nincs jelszó megadva:

```
awk -F: '$2 == ""' /etc/passwd
```

5.5.5. Összefoglalás a rekordok mezőkre darabolásáról

A POSIX szabvány szerint az `awk`-nak úgy kell viselkednie, mintha minden rekordot a beolvasás során darabolna fel mezőkre. Ez azt jelenti, hogy ha megváltoztatod az `FS` értékét miután a rekordot beolvasta, akkor a mezők feldarabolása azt az állapotot kell tükrözze, ami a régi `FS` használatával érvényes.

Ugyanakkor sok `awk` implementáció nem így működik. Ezek a programok csak akkor darabolják fel a a rekordot mezőkre, amikor hivatkoznak egy mezőre, így a mezők az éppen aktuális `FS` mezőelválasztó szerint lesznek megállapítva. (s.s.) Ezt a viselkedést nehéz felfedezni. Az alábbi program illusztrálja a különbséget a két megoldás között. (A `sed`[2] parancs a '`/etc/passwd`' file-nak csak az első sorát nyomtatja ki.)

```
sed 1q /etc/passwd | awk '{ FS = ":" ; print $1 }'
```

Egy rossz `awk` implementáció esetén a program a

```
root
```

sort nyomtatja ki, míg a `gawk` valami ehhez hasonlót fog kiírni:

```
root:nSijPlPhZZwgE:0:0:Root:/:
```

Az alábbi táblázat összefoglalja, hogy az `FS` értékétől függően a mezők hogyan lesznek elválasztva . (A '`==`' jelentése egyenlő.)

`FS == " "` A mezőket szóközök, tab és új sor karakterek határolják. Ha a rekord elején vagy végén szerepelnek, akkor nem lesznek figyelembe véve. Ez az alapbeállítás.

`FS ==` *egy bármilyen karakter*
: A mezőket az adott karakter választja el. Ha egymás után fordulnak elő, akkor egy üres mezőt határolnak. Ha a rekord legelején vagy legvégén egy ilyen karakter előfordul akkor az első vagy az utolsó mező egy üres mező lesz. A karakter akár speciális reguláris kifejezés operátor karakter is lehet; nem kell '\' jel elé.

`FS ==` *regexp*
: A mezőket olyan karaktersorozatok választják el, amelyek illeszkednek a *regexp* reguláris kifejezésre. A rekord elején vagy végén előforduló illeszkedő kifejezés hatására az első vagy az utolsó mező egy üres mező lesz.

`FS == ""` A rekord minden karaktere egy önálló mező.

[2] A `sed` segédprogram egy "karakter folyam (stream) szerkesztő" program. Viselkedése szintén POSIX szabványban definiált.

5.6. Meghatározott szélességű adatok beolvasása

(Ezt a fejezetet kezdő felhasználók nyugodtan átugorhatják első olvasásnál, mivel az itt leírt programozási lehetőség csak kísérleti, és bonyolult lehet megérteni.)

A `gawk` 2.13-as verziója vezette be azt a megoldást, amivel adott szélességű mezőket lehet kezelni, és nincs mezőelválasztó. Régi FORTRAN programok bemeneteként fordulhat elő ilyen adat, ahol a számok között nincs elválasztás.

Lényegében egy olyan táblázatról beszélünk, ahol az oszlopok szóközökkel vannak igazítva és *az üres mező csak egy szóköz*. Ilyen környezetben az `awk` mezőelválasztó stratégiája nem igazán tökéletes. Habár egy hordozható program a `substr` függvény használatával meg tudja oldani a problémát (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal), a megoldás nem túl szép és különösen nem lenne hatékony sok rekord esetén.

A rekord adott szélességű mezőkre darabolásához a `FIELDWIDTHS` beépített változónak kell egy olyan szöveget megadni, ahol a szövegben a mezők szélességét jelentő számokat szóközök választanak el. Minden szám egy mező szélességét adja meg, a mezők közti szóközöket is beleszámolva. Ha bizonyos oszlopokkal nem akarsz foglalkozni, akkor definiáld egy külön mezőbe a szélesség megadásával, majd ne vedd figyelembe a keletkezett mezőt.

Az alábbi adatsor a `w` Unix segédprogram kimenete és alkalmas a `FIELDWIDTHS` használatának bemutatására.

```
 10:06pm  up 21 days, 14:04,  23 users
User     tty       login  idle   JCPU   PCPU  what
hzuo     ttyV0     8:58pm            9      5  vi p24.tex
hzang    ttyV3     6:37pm    50                -csh
eklye    ttyV5     9:53pm            7      1  em thes.tex
dportein ttyV6     8:17pm  1:47                -csh
gierd    ttyD3    10:00pm     1                elm
dave     ttyD4     9:47pm            4      4  w
brent    ttyp0    26Jun91  4:46  26:46   4:41  bash
dave     ttyq4    26Jun9115days     46     46  wnewmail
```

Az alábbi program a fenti adatból kiválogatja az üresjárati (idle) időtartam hosszát, átkonvertálja másodpercbe, majd kiírja az első két mezőt és az üresjárati időt másodpercben. (A program olyan megoldásokat is tartalmaz, amiket eddig nem tárgyaltunk.)

```
BEGIN  { FIELDWIDTHS = "9 6 10 6 7 7 35" }
NR > 2 {
    idle = $4
    sub(/^  */, "", idle)   # strip leading spaces
    if (idle == "")
        idle = 0
    if (idle ~ /:/) {
        split(idle, t, ":")
        idle = t[1] * 60 + t[2]
    }
    if (idle ~ /days/)
        idle *= 24 * 60 * 60

    print $1, $2, idle
}
```

Az eredmény:

```
hzuo      ttyV0  0
hzang     ttyV3  50
eklye     ttyV5  0
dportein  ttyV6  107
gierd     ttyD3  1
dave      ttyD4  0
brent     ttyp0  286
dave      ttyq4  1296000
```

Egy másik (talán praktikusabb) példa a szavazókártyák feldolgozása. Az USA egyes területein úgy kell szavazni, hogy lyukat kell ütni egy kártyába. Ezeket a kártyákat használjak a szavazatszámlálás során. Mivel az is előfordulhat, hogy valaki az adott kérdésben nem akar szavazni egyes oszlopok üresek lehetnek. Az ilyen adatok feldolgozására az `gawk` jól használhatná a `FIELDWIDTHS` megoldást. (Persze az egy másik kérdés, hogy a `gawk` hogyan kerülne a kártyaolvasó gépbe!)

Ha értéket adunk az `FS` változónak a `gawk` visszatér az eredeti mező darabolási metódushoz. Mivel valószínűleg nem akarod tudni az `FS` értékét, csak visszakapcsolni normál módba, használhatod a '`FS = FS`' kifejezést is.

Ez a lehetőség még csak kísérleti, idővel változhat. Figyelj oda, mert a `gawk` egyáltalán nem ellenőrzi a `FIELDWIDTHS`-nek megadott értékeket.

5.7. Több sorból álló rekordok

Ha egy adatbázisban egy sor nem tudja kényelmesen tárolni az összes információt, akkor több soros rekordot érdemes használni.

Az első lépés a megfelelő adatformátum kiválasztása: hogyan definiálsz egy rekordot? Mi választja el a rekordokat?

Az egyik megoldás valamilyen szokatlan karakter vagy szöveg használata rekordelválasztóként. Például használhatod a lapdobás (formfeed) karaktert ('`\f`' az `awk`-ban mint a C programozási nyelvben is), így minden rekord egy oldal a file-ban. Ehhez csak az `RS` változót kell az `"\f"`-re beállítani (egy olyan szövegre, ami csak a a lapdobás karaktert tartalmazza). Bármilyen más karakter is megfelelő, ha biztos vagy benne, hogy nem fog a rekordon belül előfordulni.

A másik megoldás, hogy üres sorok választják el a rekordokat. Ha az `RS` értéke egy üres szöveg, akkor a rekordokat egy vagy több üres sor választhatja el. Ebben az esetben a rekord mindig az első üres sornál ér véget, és a következő rekord az első nem üres sornál kezdődik - nem számít, hogy hány üres sor van a két rekord között, mindig egy elválasztóként lesznek kezelve.

Ugyanezt a hatást érheted el az `"\n\n+"` kifejezés használatával. Ez a reguláris kifejezés illeszkedik a rekord utáni új sorra és a követő egy vagy több üres sorra. Ráadásul a reguláris kifejezések a lehető leghosszabb mintára illeszkednek (lásd 4.6. Bekezdés [Mennyi szöveg illeszkedik?], 35. oldal), így a következő rekord csak az üres sorok után kezdődik - nem számít, hogy hány üres sor van a két rekord között, mindig egy elválasztóként lesznek kezelve.

Van egy fontos különbség a '`RS = ""`' és a '`RS = "\n\n+"`' kifejezések között. Az első esetben az adat file-ban előforduló kezdő és záró üres sorokat nem veszi figyelembe, egyszerűen eldobja azokat. A második esetben ez nem történik meg. (s.s.)

Most, hogy a bemenetet feldaraboltuk több sorból álló rekordokra, a második lépés a rekordokon belüli mezők megállapítása. Az egyik megoldás, hogy a sorokat hagyományos módon feldaraboljuk mezőkre, de amikor az `RS` értéke egy üres szöveg az új sor karakter mindig mezőelválasztóként viselkedik. Ez csak egy ráadás az `FS`-ben megadott elválasztóhoz.

Ugyanakkor ez probléma is lehet, ha az új sor karaktert nem akarod mezőelválasztóként használni. Mivel ezt a speciális beállítást kikapcsolni nem lehet, csak a `split` függvény használatával tudod megfelelően feldarabolni a rekordot (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal).

Egy másik megoldás a rekordok feldarabolására, hogy minden mezőt külön sorba teszünk: ekkor az `FS`-t a `"\n"` szövegre kell beállítani. (Ez az egyszerű reguláris kifejezés csak egy új sor karakterre illeszkedik.)

Egy praktikus példa az így elrendezett adatokra egy levelezési címeket tartalmazó lista, ahol minden bejegyzést egy üres sor választ el. Egy ilyen file, '`addresses`', így nézhet ki:

```
Jane Doe
123 Main Street
Anywhere, SE 12345-6789

John Smith
```

```
456 Tree-lined Avenue
Smallville, MW 98765-4321

...
```

Egy egyszerű program a file feldolgozására:

```
# addrs.awk --- simple mailing list program

# Records are separated by blank lines.
# Each line is one field.
BEGIN { RS = "" ; FS = "\n" }

{
      print "Name is:", $1
      print "Address is:", $2
      print "City and State are:", $3
      print ""
}
```

A programot futtatva ezt az eredményt kapjuk:

```
$ awk -f addrs.awk addresses
 ⊣ Name is: Jane Doe
 ⊣ Address is: 123 Main Street
 ⊣ City and State are: Anywhere, SE 12345-6789
 ⊣
 ⊣ Name is: John Smith
 ⊣ Address is: 456 Tree-lined Avenue
 ⊣ City and State are: Smallville, MW 98765-4321
 ⊣
...
```

Egy másik programot is bemutatunk a címlisták feldolgozására egy későbbi fejezetben, lásd 16.2.4. Bekezdés [Printing Mailing Labels], 233. oldal.

Az alábbi táblázat összefoglalja, hogy a rekordok hogyan lesznek feldarabolva az `RS` értékétől függően (a '`==`' egyenlőséget jelent):

`RS == "\n"`
: A rekordokat az új sor karakter választja el. Tehát minden sor egy rekord, az üres sorok is. Ez az alapbeállítás.

`RS ==` *egy bármilyen karakter*
: A rekordokat a megadott karakter választja el. Egymás után előforduló karakterek egy üres rekordot jelölnek.

`RS == ""`
: A rekordokat egy vagy több üres sor választja el. Az új sor karakter mindig mezőelválasztóként viselkedik, az `FS`-től függetlenül. Kezdő és záró üres sorokat a file-ban nem veszi figyelembe.

`RS == ` *regexp*
: A rekordokat a reguláris kifejezésre illeszkedő szövegek választják el. A bemenet elején vagy végén illeszkedés a reguláris kifejezésre egy üres rekordot generál.

A `gawk` mindig beállítja az `RT` változót az `RS`-re illeszkedő szövegre.

5.8. Explicit beolvasás `getline`-al

Eddig a bemenetet az `awk` program számára vagy a szabványos bemenetről (általában a terminálról, néha egy másik program kimenetéből) vagy a parancssorban megadott file-okból olvastuk be. Az `awk` nyelvben a `getline` beépített függvénnyel lehet explicit módon a bemenet olvasását irányítani.

5.8.1. A `getline` bemutatása

A `getline` függvény hasznos lehet sokféleképpen, de kezdő felhasználóknak nem ajánlott. Azért itt tárgyaljuk, mivel minden a bemenettel kapcsolatos dolgot ebben a fejezetben tárgyalunk. A `getline` függvény bemutatására használt példákban előfordul olyan programozási megoldás, amit eddig még nem tárgyaltunk, így ezt a részt ajánlott *újra* elolvasni miután végigolvastad, és már jól ismered a könyvet.

A `getline` visszatérési értéke egy, ha sikeresen beolvasott egy rekordot és zérus ha elérte a bemenet végét. Ha valami hiba volt az olvasás során, például a file nem érhető el, akkor a visszatérési értéke -1. Ebben az esetben a `gawk` az `ERRNO` változót beállítja egy, a hibát leíró szövegre.

Az alábbi példákban a *command* egy shell parancsot helyettesít.

5.8.2. A `getline` használata argumentum nélkül

Az argumentum nélkül meghívott `getline` függvény az aktuális file-ból olvas be rekordot. Csak annyit csinál, hogy beolvassa a következő rekordot és feldarabolja mezőkre. Ez a viselkedés akkor lehet hasznos, ha befejezted az aktuális rekord feldolgozását és *közvetlenül utána* valamilyen speciális műveletet akarsz a következő rekordon elvégezni. Itt egy példa:

```
awk '{
     if ((t = index($0, "/*")) != 0) {
          # value will be "" if t is 1
          tmp = substr($0, 1, t - 1)
          u = index(substr($0, t + 2), "*/")
          while (u == 0) {
               if (getline <= 0) {
                    m = "unexpected EOF or error"
                    m = (m ": " ERRNO)
                    print m > "/dev/stderr"
                    exit
               }
               t = -1
               u = index($0, "*/")
          }
          # substr expression will be "" if */
          # occurred at end of line
          $0 = tmp substr($0, t + u + 3)
     }
     print $0
}'
```

Ez az program kitöröl minden a C programozási nyelvben szokásos megjegyzést, '`/* ... */`', a bemenetből. Ha a '`print $0`' kifejezést lecseréled valamilyen másik kifejezésre, akkor bonyolultabb műveletet is végezhetsz a megjegyzésektől mentes bemeneten, pl. reguláris kifejezésekkel változókat, stb. kereshetsz. A programnak van egy apró hibája – ugyanis nem működik, ha egy megjegyzés az adott sorban végződik és egy másik megjegyzés ugyanabban a sorban kezdődik.

Ha a `getline` függvényt így használod, akkor frissíti az `NF` (mezők száma; lásd 5.2. Bekezdés [Mezők elérése], 40. oldal), az `NR` (az eddig beolvasott rekordok száma; lásd 5.1. Bekezdés [Hogyan történik a feldarabolás rekordokra], 37. oldal), az `FNR` (az ebből a file-ból beolvasott rekordok száma) és a `$0` változó értékét.

Megjegyzés: A `getline` az új `$0` értéket használja bármilyen további szabályban megadott mintaillesztésre. A `$0` azon értéke, ami az aktuális szabályt aktiválta elveszett (s.s.). Ezzel ellentétben a `next` kifejezés beolvassa a következő rekordot és a feldolgozást az első szabálytól kezdi. Lásd 9.7. Bekezdés [A `next` kifejezés], 111. oldal.

5.8.3. Beolvasás egy változóba `getline`-al

A '`getline` *var*' kifejezés beolvassa a következő rekordot és a *var* változóban tárolja. Semmilyen más feldolgozás nem történik.

Például tegyük fel, hogy a következő sor a bemeneten egy megjegyzés vagy egy speciális szöveg és be akarod olvasni és tárolni, de egy szabályt sem akarsz aktiválni. A `getline` ezen formája ezt teszi lehetővé, ráadásul az `awk` fő rekord-beolvasás-és-minden-szabály-ellenőrzése ciklus az így beolvasott rekordot soha nem látja.

Az alábbi példa a bemenet minden második sorát felcseréli az előzővel, tehát ha a bemenet:

```
wan
tew
free
phore
```

akkor az eredmény:

```
tew
wan
phore
free
```

A program:

```
awk '{
     if ((getline tmp) > 0) {
          print tmp
          print $0
     } else
          print $0
}'
```

Ha a `getline` függvényt így használod, akkor csak az `NR` és az `FNR` (és természetesen a *var*) változók értéke változik meg. A beolvasott rekordot nem darabolja fel mezőkre, így sem a a mezők sem a `$0` és az `NF` értéke nem változik meg.

5.8.4. Beolvasás file-ból `getline`-al

A '`getline <` *file*' kifejezés beolvassa a következő rekordot a *file*-ból. Itt a *file* egy szöveg értékű kifejezés kell legyen, ami a file nevét adja meg. A '`<` *file*' kifejezést *átirányításnak* is szokták nevezni, mivel azt adja meg, hogy a bemenet valahonnan máshonnan (más irányból) jöjjön.

Például az alábbi program egy új rekordot olvas be a '`secondary.input`' file-ból, ha az első mező értéke tíz az aktuális rekordban.

```
awk '{
    if ($1 == 10) {
         getline < "secondary.input"
         print
    } else
         print
}'
```

Mivel nem a fő bemenetet használja, az `NR` és az `FNR` változók értéke nem változik meg, de az újonnan beolvasott rekordot feldarabolja mezőkre a normális módon, így a `$0` és az `NF` is megváltozik.

A POSIX szabvány szerint a '`getline <` *expression*' nem egyértelmű, ha a kifejezés tartalmaz a '`$`'-on kívül egyéb operátort; például a '`getline < dir "/" file`' nem egyértelmű, mert az összefűzés operátor nincs zárójelek között. Ha több `awk` implementációval is használni szeretnéd a programodat, akkor a '`getline < (dir "/" file)`' kifejezést érdemes használni.

5.8.5. Beolvasás file-ból egy változóba a `getline`-al

A '`getline` *var* `<` *file*' kifejezés beolvassa a következő rekordot a *file*-ból és a *var* változóban tárolja. Mint előbb, a *file* itt is egy szöveg értékű kifejezés kell legyen, ami a file nevét adja meg.

Ebben az esetben egyetlen beépített változó tartalma sem változik meg és a rekord nem lesz feldarabolva mezőkre. Egyedül a *var* változó kap új értéket.

Például az alábbi program a bemeneti file minden sorát kinyomtatja, kivéve azokat a rekordokat amelyek a '`@include` *filename*' kifejezést tartalmazzák. Ezeket a rekordokat a *filename* file tartalmával helyettesíti.

```
awk '{
     if (NF == 2 && $1 == "@include") {
          while ((getline line < $2) > 0)
               print line
          close($2)
     } else
          print
}'
```

Érdemes megfigyelni, hogy az extra file neve nincs beépítve a programba, hanem a második mezőből olvassuk ki.

A `close` függvény biztosítja, hogy ha két azonos '`@include`' sor van a fileban, akkor a megadott file tartalma kétszer lesz kinyomtatva. Lásd 6.8. Bekezdés [Bemeneti és kimeneti file-ok és csövek lezárása], 74. oldal.

A program egyik hibája, hogy beágyazott '`@include`' kifejezéseket nem tud feldolgozni (egy '`@include`' egy másik '`@include`' kifejezéssel megadott file-ban), mint ahogy egy igazi macro feldolgozó programnak kellene. Lásd

16.2.9. Bekezdés [An Easy Way to Use Library Functions], 243. oldal, amely tartalmaz egy megoldást beágyazott '`@include`' kifejezésekre.

5.8.6. Beolvasás csőből (pipe) `getline`-al

Egy másik program kimenetét átirányíthatod a `getline`-ba, a '*command* `| getline`' kifejezéssel. Ebben az esetben a *command* mint egy shell parancs fog lefutni, és a kimenetét az `awk` fogja használni, mint bemenetet. A `getline` egyszerre csak egy rekordot olvas be a csőből.

Például az alábbi program a bemenetét a kimenetre másolja, kivéve azokat a sorokat amelyek egy '`@execute`' szöveget tartalmaznak, amikor is a rekord többi részét mint shell parancs hajtja végre és az eredményt nyomtatja ki.

```
awk '{
     if ($1 == "@execute") {
          tmp = substr($0, 10)
          while ((tmp | getline) > 0)
               print
          close(tmp)
     } else
          print
}'
```

A `close` függvény biztosítja, hogy ha két azonos '`@include`' sor van a fileban, akkor a megadott file tartalma kétszer lesz kinyomtatva. Lásd 6.8. Bekezdés [Bemeneti és kimeneti file-ok és csövek lezárása], 74. oldal.

Tehát, ha ez a bemenet:

```
foo
bar
baz
@execute who
bletch
```

a program ezt az eredményt produkálja:

```
foo
bar
baz
arnold     ttyv0   Jul 13 14:22
miriam     ttyp0   Jul 13 14:23     (murphy:0)
bill       ttyp1   Jul 13 14:23     (murphy:0)
bletch
```

A program végrehajtotta a `who` parancsot és annak eredményét nyomtatja ki. (Ha kipróbálod a programot, valószínű, hogy más eredményt fogsz kapni, mivel azokat a felhasználókat fogja kinyomtatni akik be vannak jelentkezve a rendszeren.)

A `getline` ilyen használata esetén a bemenetet feldarabolja, beállítja az `NF` értékét és a `$0`-t újraértékeli. Az `NR` és az `FNR` értéke nem változik meg.

A POSIX szabvány szerint a '*expression* `| getline`' nem egyértelmű, ha a kifejezés tartalmaz a '`$`'-on kívül egyéb operátort; például a '`"echo " "date" | getline`' nem egyértelmű, mert az összefűzés operátor nincs zárójelek között. Ha több `awk` implementációval is használni szeretnéd a programodat, akkor a '`("echo " "date") | getline`' kifejezést érdemes használni. (A `gawk` helyesen kezeli ezt az esetet, de nem érdemes ebben bízni. A zárójelekkel egyébként is jobban olvasható, hogy mi is történik.)

5.8.7. Beolvasás csőből (pipe) egy változóba `getline`-al

Ha a '*command* `| getline` *var*' kifejezést használod, akkor a *command* kimenete a `getline`-ba lesz átirányítva majd a *var* változót is beállítja. Például az alábbi program beolvassa a mai dátumot és a pontos időt a `current_time` változóba a `date` segédprogram segítségével, majd kinyomtatja:

```
awk 'BEGIN {
      "date" | getline current_time
      close("date")
      print "Report printed on " current_time
}'
```

Ebben az esetben egyetlen beépített változó sem változik meg és a rekordot sem darabolja fel mezőkre.

5.8.8. Összefoglaló a `getline` változatairól

A `getline` használata esetén bár a `$0` és az `NF` értéke lehet hogy megváltozik, de a beolvasott rekordot nem teszteli minden mintával az `awk` programban, ami normális esetben történne. Ugyanakkor a beolvasást követő szabályokban az új rekordot használja.

Sok `awk` implementáció egyre korlátozza az egyszerre megnyitható csövek számát. A `gawk`-ban nincs ilyen korlátozás, annyi csövet nyithatsz meg, amennyit az operációs rendszer megenged.

A `getline`-nak egy érdekes mellékhatása van, ha a `BEGIN` szabályon belül használjuk. Mivel az átirányítás nélküli `getline` a parancssorban megadott file-okból olvas, az első `getline` kifejezés beállítja a `FILENAME` változót is. Általában a `FILENAME`-nek nincs értéke a `BEGIN` szabályon belül, mivel a file-ok feldolgozása még nem kezdődött meg (s.s.). (Lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal, és lásd 10.2. Bekezdés [Információt hordozó beépített változók], 117. oldal.)

Az alábbi táblázat összefoglalja a `getline` hat lehetséges használati módját, illetve megadja, hogy mely változók értéke változik meg.

`getline` beállítja a `$0`, `NF`, `FNR` és az `NR` változókat.

`getline` *var*
: beállítja a *var*, `FNR` és az `NR` változókat.

`getline <` *file*
: beállítja a `$0` és az `NF` változókat.

`getline` *var* `<` *file*
: beállítja a *var* változót.

command `| getline`
: beállítja a `$0` és az `NF` változókat.

command `| getline` *var*
: beállítja a *var* változót.

6. Kimenet megjelenítése

Az egyik leggyakoribb tevékenység a *nyomtatás*, vagyis a teljes bemenet vagy csak a bemenet egy részének a kinyomtatása. Egyszerű nyomtatás esetén a `print`-et érdemes használni. Bonyolultabb formázásra és nyomtatásra a `printf` alkalmas. Mindkét lehetőséget ebben a fejezetben tárgyaljuk.

6.1. A `print` kifejezés

A `print` kifejezés egyszerű, szabványos formázással nyomtatja ki az eredményt, csak meg kell adni a szövegeket, számokat egy vesszővel elválasztott listában. A nyomtatás során szóközök választják el a megadott lista elemeit, majd az egészet egy új sor karakter zárja. Egy kifejezés valahogy így néz ki:

`print` *elem1*`,` *elem2*`, ...`

Ha akarod, akkor a teljes listát zárójelek közé teheted. A zárójel kötelező, ha a lista bármely eleme használja a '`>`' karaktert (összehasonlító operátor), mivel összekeverhető az átirányítás operátorral (lásd 6.6. Bekezdés [A `print` és a `printf` kimenetének átirányítása], 70. oldal).

A kinyomtatandó elemek lehetnek szöveg vagy szám konstansok, az aktuális rekord mezői (mint a `$1`), változók vagy bármilyen `awk` kifejezés. A numerikus kifejezéseket előbb szöveggé konvertálja, és csak utána nyomtatja ki.

A `print` kifejezésnek teljesen szabadon megadhatod, hogy *mit* nyomtasson ki, de két kivételtől eltekintve nem tudod befolyásolni, hogy *hogyan* jelenítse meg az eredményt – hány számjegyet, exponenciális vagy lebegőpontos formát használjon és így tovább. (A kivételeket két bekezdésben mutatjuk be, lásd 6.3. Bekezdés [Kimeneti elválasztó], 63. oldal és a 6.4. Bekezdés [A numerikus adatok formátumának megadása a `print` esetén], 64. oldal.) A megjelenítési forma megadásához a `printf` kifejezést kell használni (lásd 6.5. Bekezdés [Nyomtatás a `printf` kifejezéssel], 64. oldal).

Az önmagában álló '`print`' kifejezés megegyezik a '`print $0`' kifejezéssel: kinyomtatja a teljes rekordot. Egy üres sor nyomtatásához a '`print ""`' kifejezést kell használni, ahol a `""` egy üres szöveg.

Egy adott szöveget szöveg konstansként érdemes kinyomtatni. Ha elfelejted megadni a macskakörmöket, akkor a szavak mint `awk` kifejezések lesznek kezelve, és a program valószínűleg hibát fog jelezni. Emlékezz arra is, hogy minden elem közé egy extra szóközt is nyomtat.

Minden `print` kifejezés legalább egy sort nyomtat, persze lehet hogy többet is. Ha egy szöveg tartalmazza az új sor karaktert, akkor azt is kinyomtatja, vagyis a szöveg többi részét egy új sorba nyomtatja ki. Egy `print` kifejezés tetszőleges számú sort nyomtathat ki.

6.2. Példák a print kifejezéssel

Az első példa bemutatja a beágyazott új sor karakterek nyomtatását (a '\n' escape szekvencia az új sor karaktert reprezentálja; lásd 4.2. Bekezdés [Escape szekvenciák], 24. oldal):

```
$ awk 'BEGIN { print "line one\nline two\nline three" }'
⊣ line one
⊣ line two
⊣ line three
```

A második példa minden bemeneti rekordból kinyomtatja az első két mezőt egy szóközzel elválasztva:

```
$ awk '{ print $1, $2 }' inventory-shipped
⊣ Jan 13
⊣ Feb 15
⊣ Mar 15
...
```

Gyakori hiba a `print` használata során, hogy nem teszünk vesszőt az elemek közé. Ennek az a hatása, hogy közvetlenül egymás után írja ki az elemeket, mivel ez az írásmód az `awk`-ban az összefűzést jelenti. Tehát ugyanaz a példa vesszők nélkül:

```
$ awk '{ print $1 $2 }' inventory-shipped
⊣ Jan13
⊣ Feb15
⊣ Mar15
...
```

Ha valaki nem ismeri az '`inventory-shipped`' file tartalmát, akkor a fenti példák eredményei nem mondanak túl sokat. Ilyenkor jól jöhet egy fejléc, jelezve, hogy az első mező a hónap (`$1`) a második mező (`$2`) pedig a zöld rekeszek száma. A fejlécet a `BEGIN` mintán belül nyomtatjuk ki, hogy csak egyszer jelenjen meg (lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal):

```
awk 'BEGIN {  print "Hónap Rekesz"
              print "----- ------" }
           {  print $1, $2 }' inventory-shipped
```

Kitalálod, hogy mi fog történni? A program futtatása után ezt kapjuk:

```
Month Crates
----- ------
Jan 13
Feb 15
Mar 15
...
```

Az adatok és a fejléc nem kerül egy oszlopba! A probléma persze megoldható ha megfelelő számú szóközt teszünk a mezők közé:

```
awk 'BEGIN { print "Hónap Rekesz"
             print "----- ------" }
           { print $1, "     ", $2 }' inventory-shipped
```

Könnyű elképzelni, hogy több oszlop igazítása ezzel a módszerrel elég bonyolult lehet. Kettő vagy három oszlop esetén a szóközök még könnyen kiszámolhatók, de több oszlop esetén könnyű eltéveszteni. Ezért találták ki a `printf` kifejezést (lásd 6.5. Bekezdés [Nyomtatás a `printf` kifejezéssel], 64. oldal), aminek az egyik specialitása az oszlopok igazítása.

Mellékesen, egy `print` vagy `printf` kifejezés folytatható a következő sorban, ha a vessző után egy új sor karakter áll (lásd 2.6. Bekezdés [Statements/Lines], 16. oldal).

6.3. Kimeneti elválasztó

Ahogy azt korábban leírtuk, a `print` kifejezés argumentumait vessző választja el. Ennek hatására a kinyomtatás során az elemek között egy szóközt nyomtat. Ennek persze nem kell így lennie; a szóköz csak az alapbeállítás. Bármilyen karaktersorozat megadható, mint *kimeneti mezőelválasztó*, az `OFS` beépített változónak. A változó kezdeti értéke a `" "` szöveg, egy szóköz.

Egy teljes `print` kifejezés kimenete egy *kimeneti rekord*. Minden `print` kifejezés kinyomtat egy kimeneti rekordot, majd egy szöveget amit úgy hívnak, hogy *kimeneti rekordelválasztó*, az `ORS` beépített változóban megadott értéket. Az `ORS` kezdeti értéke a `"\n"`, az új sor karakter; így alapesetben minden `print` kifejezés egy új sorral zárja a rekordot.

Természetesen megváltoztathatod, hogy a kimeneti rekordok és mezők hogyan legyenek elválasztva, ha új értéket adsz az `OFS` és/vagy `ORS` változóknak. Ezt általában a `BEGIN` szabályban érdemes megtenni, (lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal), a bemenet feldolgozásának megkezdése előtt. A változókat beállíthatod a parancssorból is a bemeneti file-ok előtt vagy a '`-v`' parancssori opció használatával (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

Az alábbi példa kinyomtatja az első és a második mezőt egy pontos vesszővel elválasztva és a rekordok közé egy új, üres sort illesztve:

```
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
>            { print $1, $2 }' BBS-list
⊣ aardvark;555-5553
⊣
⊣ alpo-net;555-3412
⊣
⊣ barfly;555-7685
...
```

Ha az `ORS` nem tartalmaz egy új sor karaktert, akkor minden kimenet egy sorba lesz kinyomtatva, kivéve persze ha valami más explicit módon nincs megadva az új sor karakter nyomtatása.

6.4. A numerikus adatok formátumának megadása a `print` esetén

Amikor a `print` kifejezéssel egy számot nyomtatunk ki, az `awk` átalakítja a számot szöveggé, majd ezt a szöveget nyomtatja ki. Az `awk` az `sprintf` függvényt használja a konvertáláshoz (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Egyelőre elég annyit elmondani, hogy az `sprintf` úgynevezett *formátumleírót* használ, ami megadja, hogy a szám (vagy szöveg) hogyan fog megjelenni. A különböző formátumokat részletesen tárgyaljuk egy későbbi alfejezetben, 6.5.2. Bekezdés [Formátumleíró betűk], 65. oldal.

Az `OFMT` beépített változó tartalmazza az alap formátumleírót, amit a `print` használ a számok szöveggé konvertálása során. Az `OFMT` alapértéke `"%.6g"`. Ha más értéket adunk meg az `OFMT`-nek, akkor szabályozható, hogy a számokat hogyan nyomtassa ki a `print`. Egy rövid példa:

```
$ awk 'BEGIN {
>   OFMT = "%.0f"  # minden számot integerként nyomtat ki
>   print 17.23 }'
 ⊣ 17
```

A POSIX szabvány szerint az `awk` viselkedése nem definiált, ha az `OFMT` nem lebegőpontos formátumleírót tartalmaz (s.s.).

6.5. Nyomtatás a `printf` kifejezéssel

Ha a `print` lehetőségeinél jobban akarod kontrollálni a nyomtatási formátumot, akkor a `printf` kifejezést használd. A `printf` -el megadhatod a nyomtatási szélességet minden elemre és a számokat különböző formátumban (különöző számrendszerben, exponenciális vagy lebegőpontos alakban, megadott számjegy pontossággal) nyomtathatod ki. Mindezt egy *formátumszöveg* megadásával érheted el.

6.5.1. A `printf` bemutatása

A `printf` kifejezést így kell használni:

`printf` *formátum*, *elem1*, *elem2*, ...

Az argumentumokat zárójelek közé is teheted. A zárójel kötelező, ha bármelyik elem a '`>`' összehasonlító operátort tartalmazza, mert az `awk` összekeverheti az átirányítás operátorral (lásd 6.6. Bekezdés [A `print` és a `printf` kimenetének átirányítása], 70. oldal).

A `printf` és a `print` közötti különbség a *format* argumentum. Ez egy szöveg értékű kifejezés; megadja, hogy a többi argumentumot hogyan nyomtassa ki. A kifejezést *formátumszövegnek* hívják.

A formátumszöveg nagyon hasonló az ANSI C-ben használt `printf` függvény argumentumához. A *format* kifejezés jelentős része egyszerű szöveg, amit csak ki kell nyomtatni. A szöveg között vannak elszórva a formátumleírók. Minden elemhez tartozik egy. A formátum szöveg veszi a következő argumentumot, és az adott helyen kinyomtatja.

A `printf` kifejezés nem kezd új sort a nyomtatás végén automatikusan. Csak annyit és azt nyomtat ki, amit a formátumszöveg megad, így ha egy új sor karakter is kell a szöveg végére, akkor azt explicit módon kell megadni. Az `OFS` és az `ORS` változóknak nincs hatása a `printf` kifejezésre, így:

```
BEGIN {
   ORS = "\nOUCH!\n"; OFS = "!"
   msg = "Don't Panic!"; printf "%s\n", msg
}
```

A program még mindig csak annyit nyomtat ki, hogy '`Don't Panic!`'.

6.5.2. Formátumleíró betűk

Egy formátumleíró a '`%`' karakterrel kezdődik és egy betűvel végződik. (Ha csak a '`%`' karaktert akarod kinyomtatni, akkor a '`%%`' karaktersorozatot kell használni.) A formátumleíró betű határozza meg a kinyomtatandó adat típusát. A leíró többi része csak *módosítja* a megjelenést, mint például a nyomtatási szélességet.

Itt egy lista a formátumleíró betűkről:

`c` Ez egy ASCII karaktert nyomtat ki. A számokat is átkonvertálja ASCII karakterré, így a '`printf "%c", 65`' kifejezés a nagy '`A`' betűt nyomtatja ki. Egy szöveg esetén az első karaktert nyomtatja ki.

`d`
`i` A két betű megegyezik, mindkettő egy decimális egészt (integer) nyomtat ki. A '`%i`' az ANSI C-vel kompatíbilis formátumleíró betű.

`e`
`E` Ez egy exponenciális alakú számot nyomtat ki. Például,

```
printf "%4.3e\n", 1950
```

'`1.950e+03`' -t fog kinyomtatni, négy értékes jegyre, amiből három számjegy a tizedes pont után áll. A '`4.3`' egy leíró módosító és később tárgyaljuk. A '`%E`' egy nagy '`E`' betűt használ a számban a kis '`e`' betű helyett.

`f` Ez egy lebegőpontos számot nyomtat ki, például:

```
printf "%4.3f", 1950
```

'1950.000' -t jeleníti meg négy értékes jegyre, amiből három számjegy a tizedes pont után áll. A '4.3' egy leíró módosító és később tárgyaljuk.

`g`
`G` Ez vagy exponenciális vagy lebegőpontos alakban nyomtatja ki a számot, amelyik a rövidebb azt használja. Ha exponenciális alakot használ, akkor a '%G' egy nagy 'E' betűt nyomtat a számban a kis 'e' betű helyett.

`o` Ez egy előjel nélküli, oktális, egész számot (integer) nyomtat ki. (Oktális módban, vagy nyolcas alapú számrendszerben, a számjegyek '0' és '7' közöttiek lehetnek; a decimális nyolc oktális alakban '10' lesz.)

`s` Kinyomtatja az adott szöveget.

`x`
`X` Ez egy előjel nélküli, hexadecimális, egész számot (integer) nyomtat ki. (Hexadecimális módban, vagy 16-os alapú számrendszerben a számjegyek '0-9' és 'a-f' közöttiek lehetnek. A hexadecimális 'f' a decimális 15-nek felel meg.) A '%X' használata esetén a nagy 'A' és 'F' közötti betűket fogja használni.

`%` Ez nem igazán formátumleíró betű, de ha a '%' karakter után áll, akkor a '%%' karaktersorozat hatására egy '%' karaktert fog megjeleníteni. Nem használja a `printf` argumentumait, és minden leíró módosítót figyelmen kívül hagy.

Amikor az egész (integer) formátumleíró betűt használjuk egy olyan értéknél, ami nem reprezentálható a C `long` típussal, akkor a `gawk` automatikusan a '%g' formázási módba kapcsol. Más `awk` verziók valótlan értéket nyomtathatnak, vagy valami furcsa dolgot csinálhatnak ebben az esetben (s.s.).

6.5.3. A `printf` formátum módosítói

A formátum leírók módosító komponenseket is tartalmazhatnak, amelyek meghatározzák, hogy az elemből mennyi jelenik meg. A módosító komponenseket a '%' karakter és a formátumleíró betű között kell elhelyezni. Az alábbi példákban a "•" jelet használjuk a szóközök helyén a kimenetben. A módosítók az alábbi felsorolás sorrendjében jelenhetnek meg:

`-` A mínusz jel balra igazítja az argumentumot a megadott szélességen belül (lásd lent). Általában az argumentumok jobbra igazítva jelennek meg. Így,

```
printf "%-4s", "foo"
```

ezt adja eredményül: '`foo•`'.

szóköz
: Numerikus konverzió esetén, ha a szám pozitív egy szóköz, ha negatív akkor egy mínusz karaktert nyomtat ki.

`+`
: A plusz jel hatására a számok előtt mindig fog előjelet nyomtatni még akkor is ha a szám pozitív. A szélesség megadásával együtt lehet használni és erősebb mint a szóköz módosító (lásd fent).

`#`
: Bizonyos formátumok esetén egy "alternatív formát" használ. A '`%o`' esetén a szám előtt egy zérus számjegy karaktert nyomtat. Ha az '`%x`' vagy '`%X`' formátumot használjuk, akkor a nem zérus számok előtt egy '`0x`' vagy egy '`0X`' szöveget nyomtat. A '`%e`', '`%E`' és '`%f`' esetén az eredmény mindig tartalmaz egy tizedes pontot, míg a '`%g`' és '`%G`' esetén a szám végén álló zérus karaktereket is mindig kinyomtatja.

`0`
: Egy kezdő zérus karakter azt adja meg, hogy a hely kitöltéséhez zérus számjegy karaktereket használjon a szóközök helyett. Ez a nem numerikus értékekre is igaz (s.s.). Ennek csak akkor van hatása, ha a nyomtatási szélesség nagyobb mint a kinyomtatandó adat szélessége.

szélesség
: Ez a szám megadja a kívánt minimális nyomtatási szélességet. Egy szám a '`%`' karakter és a formátumleíró betű között megnöveli a minimális nyomtatási szélességet. Alapesetben a szóközöket a bal oldalon helyezi el. Például,

```
printf "%4s", "foo"
```

eredménye: '`•foo`'.

A *szélesség* a minimumot adja meg és nem a maximumot. Így ha a kinyomtatandó elem szélesebb mint a megadott érték ettől függetlenül az egész elemet kinyomtatja, például:

```
printf "%4s", "foobar"
```

eredménye: '`foobar`'.

A *szélesség* előtti mínusz jel hatására az extra szóközöket a jobb oldalra rakja.

.pontosság
: Ez a szám a pontosságot adja meg. Az '`e`', '`E`' és '`f`' formátum esetén a tizedes pont után álló számjegyek számát adja meg. A '`g`' és a '`G`' formátumnál az értékes jegyek számát határozza meg. A '`d`', '`o`', '`i`', '`u`', '`x`' és '`X`' formátumoknál a minimálisan kinyomtatandó számjegyek számát, míg egy szövegnél a maximálisan kinyomtatandó karakterek számát írja elő. Így,

```
printf "%.4s", "foobar"
```

eredménye: '`foob`'.

A C programozási nyelv `printf` kifejezésében használható dinamikus *szélesség* és *pontosság* meghatározás (`"%*.*s"`) az `awk`-ban is engedélyezett. Ebben az esetben nem a formátum szövegben kell megadni a *szélesség* és/vagy *pontosság* értékét, hanem mint argumentum kell átadni a kifejezésnek, például:

```
w = 5
p = 3
s = "abcdefg"
printf "%*.*s\n", w, p, s
```

pontosan ugyanaz, mint

```
s = "abcdefg"
printf "%5.3s\n", s
```

Mindkét program a '••abc' kifejezést nyomtatja ki.

Az `awk` korábbi verziói nem támogatták ezt a lehetőséget. Ha ilyen `awk`-ot kell használnod, akkor az összefűzés operátorral szimulálhatod a dinamikus módot:

```
w = 5
p = 3
s = "abcdefg"
printf "%" w "." p "s\n", s
```

Bár nem egyszerű olvasni ezt a formát, de működik.

A C programozók már hozzászokhattak a '`l`' és '`h`' megadásához a `printf` formátumszövegében, de az `awk`-ben ezek nem használhatók. A legtöbb `awk` implementáció csendben figyelmen kívül hagyja ezeket a módosítókat. Ha a '`--lint`' opció meg van adva a parancssorban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor a `gawk` figyelmeztet ezen módosítók használata esetén. Ha a '`--posix`' opció van megadva, akkor használatuk végzetes hiba, a program azonnal leáll.

6.5.4. `printf` példák

Ez a példa egy igazított táblázatot nyomtat ki a `printf` segítségével:

```
awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
```

kinyomtatja a '`BBS-list`' file-ban található rendszerek nevét (`$1`) egy tíz karakter szélességű mezőbe, balra igazítva. Ezenkívül kinyomtatja a telefonszámot (`$2`) is. Az alábbi két oszlopból álló táblázat az eredmény:

```
$ awk '{ printf "%-10s %s\n", $1, $2 }' BBS-list
⊣ aardvark   555-5553
⊣ alpo-net   555-3412
⊣ barfly     555-7685
⊣ bites      555-1675
⊣ camelot    555-0542
⊣ core       555-2912
⊣ fooey      555-1234
⊣ foot       555-6699
⊣ macfoo     555-6480
⊣ sdace      555-3430
⊣ sabafoo    555-2127
```

Észrevetted, hogy a telefonszámot nem mint számot nyomtattuk ki? A telefonszámokat mint szöveg kellett kinyomtatni, mivel szerepel egy mínusz jel a számban. Ha mint számot nyomtatjuk ki, akkor csak az első három számjegyet kapnánk meg eredményül, '555'.

A telefonszám formátumánál nem adtunk meg szélességet, mivel az utolsó elem a sorban, így nem akarunk szóközöket utána.

A táblázatot szebbé tehetjük, ha egy fejlécet adunk az oszlopok tetejéhez. Ehhez a `BEGIN` mintát kell alkalmazni (lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal), mivel így csak egyszer, az `awk` program legelején nyomtatja ki a fejlécet:

```
awk 'BEGIN { print "Név       Szám"
             print "---       ----" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

Észrevetted, hogy a `print` és `printf` kifejezéseket is használtunk a fenti példában? Használhatunk csak `printf` kifejezést is:

```
awk 'BEGIN { printf "%-10s %s\n", "Név", "Szám"
             printf "%-10s %s\n", "---", "----" }
     { printf "%-10s %s\n", $1, $2 }' BBS-list
```

Mivel ugyanazt a formátumot használtuk a fejléc és az adatok nyomtatására, így biztosak lehetünk benne, hogy megfelelően lesznek igazítva.

Ha hangsúlyozni akarjuk, hogy ugyanazt a formátumot kell használni mindegyik esetben, akkor a formátumot egy változóban is tárolhatjuk:

```
awk 'BEGIN { format = "%-10s %s\n"
             printf format, "Név", "Szám"
             printf format, "---", "----" }
     { printf format, $1, $2 }' BBS-list
```

Most már tudod, hogyan kellett volna használni a `printf` kifejezést az '`inventory-shipped`' file-ból nyert táblázat nyomtatásánál (lásd 6.1. Bekezdés [A `print` kifejezés], 61. oldal). Meg tudod csinálni?

6.6. A `print` és a `printf` kimenetének átirányítása

Eddig csak azokkal az esetekkel foglalkoztunk, amikor a nyomtatás a szabványos kimeneten, általában a terminálon jelent meg. A `print` és a `printf` is képes a kimenetet *átirányítani*.

Az átirányítást a `print` vagy a `printf` kifejezés után kell írni és a formája ugyanolyan, mint a shell-ben használt átirányítás.

Az átirányításnak három formája van: kimenet egy file-ba, kimenet hozzáfűzése egy file-hoz és kimenet átirányítása egy csővön (pipe-on) keresztül egy másik parancsba. A `print` kifejezéssel mutatjuk be az átirányítás használatát, de ugyanígy minden érvényes a `printf` kifejezésre is.

`print` *elemek* `>` *output-file*

Az átirányítás ezen formája az *output-file* file-ba nyomtatja az eredményt. Az *output-file* név bármilyen kifejezés lehet, az értékét átalakítja szöveggé, és ezt használja mint a file neve (lásd 7. Fejezet [Kifejezések], 77. oldal).

Ebben az esetben az *output-file* file-t először letörli és csak utána kezd el az új, üres file-ba nyomtatni. További átirányítások a *output-file* file-ba már nem törlik a file-t, hanem hozzáfűzik a kimenetet. Ha az *output-file* file nem létezik, akkor létrehozza.

Például az alábbi `awk` program a BBS neveket a '`name-list`' file-ba írja, míg a telefonszámokat a '`phone-list`' file-ba. Minden név vagy szám egy külön sorba kerül.

```
$ awk '{ print $2 > "phone-list"
>        print $1 > "name-list" }' BBS-list
$ cat phone-list
 ⊣ 555-5553
 ⊣ 555-3412
...
$ cat name-list
 ⊣ aardvark
 ⊣ alpo-net
...
```

`print` *elemek* `>>` *output-file*

Ebben az esetben a kimenetet egy már létező *output-file* file tartalmához fűzi hozzá. A különbség az egyszerű átirányítás (lásd fent) és e forma között az, hogy a file tartalma (ha van) az utóbbi esetben nem törlődik. Ha az *output-file* file nem létezik, akkor az `awk` létrehozza.

`print` *elemek* `|` *parancs*

Lehetőség van arra is, hogy a kimenetet egy másik programnak adjuk át egy csövön (pipe) keresztül. Ebben az esetben az `awk` megnyitja a csövet a *parancs* felé, majd az *elemek*et kiírja a csőbe.

A *parancs* átirányítási argumentum egy `awk` kifejezés. Az értékét egy szöveggé alakítja, ami megadja a végrehajtandó shell parancsot.

Az alábbi példa két file-t hoz létre, a BBS nevek egy rendezetlen és egy ábécé sorrendben visszafelé rendezett listáját:

```
awk '{ print $1 > "names.unsorted"
       command = "sort -r > names.sorted"
       print $1 | command }' BBS-list
```

A rendezetlen listát egy normál átirányítással írjuk ki, míg a rendezést a `sort` segédprogram végzi el, ami egy csövön keresztül kapja az adatokat.

A következő példa átirányítással küld egy üzenet a '`bug-system`' levelezési listára. Ilyen program hasznos lehet ha a rendszer adminisztrációra használt `awk` program hibát talál valahol.

```
report = "mail bug-system"
print "Awk script failed:", $0 | report
m = ("at record number " FNR " of " FILENAME)
print m | report
close(report)
```

Az üzenetet összefűzéssel készíti el, és az `m` változóban tárolja, majd egy csövön keresztül a `mail` programnak küldi tovább.

A `close` függvény alkalmazása fontos, mivel ez lezárja a kimeneti csövet. Lásd 6.8. Bekezdés [Bemeneti és kimeneti file-ok és csövek lezárása], 74. oldal. A példa azt is bemutatja, hogy hogyan használhatunk egy változót a *file* vagy a *parancs* reprezentálására. A változók azért is hasznosak, mert egyébként minden alkalommal pontosan ugyanúgy kellene leírni a hosszú szövegeket.

A '`>`', '`>>`' vagy '`|`' átirányítás arra kéri a rendszert, hogy nyisson meg egy file-t vagy csövet, de csak akkor, ha az adott *file*-t vagy *parancs*ot még nem használta a program vagy ha az utolsó használat végén le lett zárva.

Mint ahogy azt már korábban írtuk (lásd 5.8.8. Bekezdés [Összefoglaló a `getline` változatairól], 58. oldal), néhány `awk` implementáció csak egy cső (pipe) megnyitását engedélyezi. A `gawk`-ban nincs ilyen korlátozás, annyi csövet lehet megnyitni, amennyit az operációs rendszer engedélyez.

6.7. Speciális file nevek `gawk`-ban

Hagyományosan egy program összesen három karakterfolyamot (streamet) használ beolvasásra és kiírásra. Ezek a *szabványos bemenet*, *szabványos kimenet* és a *szabványos hibakimenet*, amelyek általában a terminálodhoz kapcsolódnak, de gyakran a shell átirányítja őket a '<', '<<', '>', '>>', '>&' és a '|' operátorok valamelyikével. A szabványos hibakimenet a hibaüzeneteket jeleníti meg; azért van két különböző karakterfolyam, a szabványos kimenet és a szabványos hibakimenet, mert így külön-külön lehet őket átirányítani.

Néhány `awk` implementáció esetén az egyetlen lehetőség egy `awk` programban a hibaüzenetek megjelenítésére a szabványos hibakimeneten keresztül az alábbi módszer:

```
print "Serious error detected!" | "cat 1>&2"
```

Ez úgy működik, hogy egy csövet nyit meg egy olyan shell parancs felé, amelyik el tudja érni a szabványos hibakimenetet. A szabványos hibakimenet értékét az `awk`-tól veszi át. Ez a megoldás nem elegáns és nem is hatékony, mivel egy másik programot kell indítani. Így az emberek jelentős része nem ezt a módszert használja, helyette a hibaüzeneteket a terminálra küldi, például így:

```
print "Serious error detected!" > "/dev/tty"
```

Általában ennek ugyanaz a hatása, de nem mindig: habár a terminál a szabványos hibakimenet az esetek nagy részében, de át lehet irányítani, és ebben az esetben a terminálra írás nem jó megoldás. Ráadásul ha az `awk` a háttérben fut, akkor nincs is terminál hozzárendelve, és a '`/dev/tty`' megnyitása nem lesz sikeres.

A `gawk` speciális file neveket biztosít a három alap karakterfolyam eléréséhez. Ha a kimenetet vagy a bemenetet átirányítjuk, és a file neve megegyezik valamelyik speciális névvel, akkor a `gawk` az adott karakterfolyamot fogja használni.

'`/dev/stdin`'
: A szabványos bemenet (0 file leíró).

'`/dev/stdout`'
: A szabványos kimenet (1 file leíró).

'`/dev/stderr`'
: A szabványos hibakimenet (2 file leíró).

'`/dev/fd/`*N*'
: Az *N* file leírónak megfelelő file. Ezeket a file-okat az `awk`-ot elindító programnak kell megnyitnia (általában a shell). Ha csak nem bravúroskodsz, akkor csak a 0, 1 és a 2 file leírók állnak rendelkezésre.

A '`/dev/stdin`', '`/dev/stdout`' és '`/dev/stderr`' file-ok megegyeznek a '`/dev/fd/0`', '`/dev/fd/1`' és '`/dev/fd/2`' file-okkal sorrendben, de a nevük kifejezőbb.

A helyes módszer egy hiba megjelenítésére egy `gawk` programból a '`/dev/stderr`' használatával:

```
print "Serious error detected!" > "/dev/stderr"
```

A `gawk` olyan speciális file-ok elérését is biztosítja, amelyek az éppen futó `gawk`-ról adnak információt. Minden ilyen “file” csak egy rekordnyi információt tartalmaz. Ha többször akarod kiolvasni az értékét, akkor előbb le kell zárni a `close` függvénnyel (lásd 6.8. Bekezdés [Bemeneti és kimeneti file-ok és csövek lezárása], 74. oldal). A file-ok:

‘`/dev/pid`’
: A file olvasása esetén visszaadja a futó program azonosítóját (ID) decimális számként egy új sor karakterrel lezárva.

‘`/dev/ppid`’
: A file olvasása esetén visszaadja a program szülőjének azonosítóját (ID) decimális számként egy új sor karakterrel lezárva.

‘`/dev/pgrpid`’
: A file olvasása esetén visszaadja a futó program csoport azonosítóját decimális számként egy új sor karakterrel lezárva.

‘`/dev/user`’
: A file olvasása egyetlen rekordot ad vissza egy új sor karakterrel lezárva. A mezőket szóközök választják el. A mezők jelentése:

  `$1` A `getuid` rendszer függvény visszatérési értéke (a valódi felhasználói azonosító, ID).

  `$2` A `geteuid` rendszer függvény visszatérési értéke (az effektív felhasználói azonosító).

  `$3` A `getgid` rendszer függvény visszatérési értéke (a felhasználó valódi csoport azonosítója).

  `$4` A `getegid` rendszer függvény visszatérési értéke (a felhasználó effektív csoport azonosítója).

  Ha van, a többi mező a `getgroups` rendszer függvény által visszaadott csoport azonosítókat tartalmazza. (Nem minden rendszer támogatja, hogy egy felhasználó több csoportba tartozhat.)

Ezeket a speciális file-okat a parancssorban is lehet használni mint adat file-ok vagy az `awk` programon belül átirányítással, de nem használhatók mint program file a ‘`-f`’ opcióval.

“Compatibility” módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) a `gawk` nem ismeri fel ezeket a speciális file-okat.

Figyelem: Ha a rendszereden nincs ‘`/dev/fd`’ könyvtár (vagy bármely más fent megadott speciális file), akkor a `gawk` maga fogja értelmezni a megadott file nevet. Például a ‘`/dev/fd/4`’ mint kimenet használata esetén a program a 4-es file leíró által megadott file-ba fog írni és nem egy olyan file leíróba ami a 4-es file leíró másolata. Általában ez nem annyira érdekes; de fontos, hogy *ne* zárjuk le a 0, 1 és 2 -es file leírókat, ugyanis ebben az esetben az `awk` viselkedése megjósolhatatlan.

A futó `awk` programról információt adó file-ok lehet, hogy nem lesznek benne a `gawk` későbbi verzióiban, lásd C.3. Bekezdés [Probable Future Extensions], 297. oldal.

6.8. Bemeneti és kimeneti file-ok és csövek lezárása.

Ha többször ugyanazt a file nevet vagy shell parancsot használjuk a `getline`-al (lásd 5.8. Bekezdés [Explicit beolvasás `getline`-al], 53. oldal) egy `awk` programon belül, a file-t csak az első alkalommal nyitja meg (a parancsot csak az első alkalommal hajtja végre). Ezzel egy időben beolvassa az első rekordot. A következő alkalommal, amikor ugyanazt a file-t vagy shell parancsot hasnáljuk a `getline`-al, a következő rekordot olvassa be.

Ugyanez érvényes a csövekre is ha írunk bele; az adott file-ra vagy parancsra emlékszik az `awk` és az első alkalom után mindig ugyanabba a file-ba írja vagy ugyanannak a parancsnak küldi a kimenetet. A file vagy cső addig marad nyitva, amíg az `awk` ki nem lép.

Ez persze azt jelenti, hogy ha ugyanazt a file-t többször szeretnéd beolvasni az elejétől vagy ugyanazt a shell parancsot szeretnéd többször lefuttatni, akkor extra lépéseket kell tenned. A `close` függvényt kell használni:

`close(`*filenév*`)`

vagy

`close(`*parancs*`)`

A *filenév* vagy *parancs* argumentum bármilyen kifejezés lehet, de az értéke *pontosan* ugyanaz kell legyen mint amit a megnyitásnál használtunk (a szóközök és egyéb "extra" karakterek is fontosak). Például ha egy csövet így nyitsz meg:

```
"sort -r names" | getline foo
```

akkor bezárni így kell:

```
close("sort -r names")
```

Miután ez a függvény lefutott a következő `getline`, `print` vagy `printf` kifejezésnél ugyanazt a file-t újra megnyitja vagy ugyanazt a parancsot újra lefuttatja.

Mivel a lezárásnál ugyanolyan értékű kifejezést kell használni mint a megnyitásnál ezért érdemes a file nevét vagy a parancsot egy változóban tárolni. Az előbbi példa tehát így fog kinézni:

```
sortcom = "sort -r names"
sortcom | getline foo
...
close(sortcom)
```

Ez segít elkerülni a nehezen megtalálható "elgépelési" hibákat az `awk` programodban.

Az alábbiakban leírunk néhány indokot, hogy a kimenetet miért érdemes/kell lezárni:

- Ugyanabból az `awk` programból hozzuk létre a file-t, mint amiből később visszaolvassuk. A létrehozás, írás után le kell zárni a file-t, így a `getline` függvénnyel elölről olvasható.
- Ugyanabból az `awk` programból több file-ba írunk. Ha nem zárjuk le a file-okat, lehet, hogy egy idő után több file-t szeretnénk nyitva tartani egyszerre, mint amennyit a rendszer engedélyez. Így ha egy file-al végeztünk, zárjuk le.
- Egy parancsot szeretnénk befejezni. Ha a kimenetet egy csőbe irányítjuk, akkor a cső addig próbál valamit beolvasni ameddig nyitva van. Ez gyakran azt jelenti, hogy a parancs nem igazán tud továbblépni, amíg a cső nyitva van. Például ha a kimenetet a `mail` programba irányítottuk, akkor az aktuális üzenetet addig nem küldi el, amíg a csövet le nem zárjuk.
- Ugyanazt a programot szeretnénk lefuttatni kétszer. Ez nem ugyanaz, mint amikor további adatot adunk át az elsőre elindított parancsnak!

  Például a kimenetet átirányítjuk a `mail` programba. Ha több sort is kiírunk, akkor azok ugyanabba az üzenetbe kerülnek. Ezzel ellentétben ha a csövet lezárjuk minden sor után, akkor minden sor egy különálló üzenet lesz.

A `close` függvény zérust ad vissza, ha a lezárás sikeres volt, egyébként valamilyen zérustól különböző értéket. Ebben az esetben a `gawk` beállítja az `ERRNO` változót egy, a hibát leíró üzenetre.

Ha több file-t próbálsz megnyitni, mint amit a rendszer engedélyez, akkor a `gawk` megpróbálja a rendelkezésre álló nyitott file-okat megosztani (multiplex), és az új file-t is megnyitni. Ez a lehetőség az operációs rendszertől is függ; nem mindig működik, ezért jó programozási szokás és könnyíti a hordozhatóságot ha a file-t mindig lezárod, miután már nem használod.

7. Kifejezések

A kifejezések az `awk` minták és tevékenységek alap építőkövei. Egy kifejezés kiértékelése egy értéket ad, amit kinyomtathatsz, tesztelhetsz, egy változóban eltárolhatsz vagy egy függvénynek átadhatsz mint argumentumot. Továbbá egy kifejezéssel új értéket rendelhetsz egy változóhoz vagy mezőhöz az értékadó operátorral.

Egy kifejezés önmagában szolgálhat mint egy minta vagy egy tevékenység. A legtöbb kifejezés olyan más kifejezéseket tartalmaz, amelyek adatokon dolgoznak. Mint más nyelvekben, az `awk`-ban is egy kifejezés tartalmazhat változót, tömb elemre hivatkozást, konstans elemet, függvényhívást és ezek bármilyen kombinációját különböző operátorral.

7.1. Konstans kifejezések

A legegyszerűbb kifejezés egy konstans, aminek mindig ugyanaz az értéke. Háromféle konstans van: számkonstans, szövegkonstans és reguláris kifejezés konstans.

7.1.1. Szám- és szövegkonstansok

Egy *számkonstans* értéke maga a szám. A szám lehet egész, lebegőpontos vagy exponenciális alakú valós szám.[1] Alább bemutatunk néhány számkonstanst; mindegyiknek ugyanaz az értéke:

```
105
1.05e+2
1050e-1
```

A szövegkonstans karakterek sorozatából áll és macskakörmök veszik körül, például:

```
"parrot"
```

Ez egy olyan szöveget reprezentál, aminek a tartalma: '`parrot`'. A `gawk`-ban a szövegek bármilyen hosszúak lehetnek, és bármely 8 bittel leírható ASCII karaktert tartalmazhatják, az ASCII NUL-t is. Más `awk` implementációknak néhány speciális karakter problémát okozhat.

7.1.2. Reguláris kifejezés konstansok

Egy reguláris kifejezés konstans egyszerűen a '`/`' karakterek között leírt reguláris kifejezés, mint a `/^beginning and end$/`. Leggyakrabban reguláris kifejezés konstansokat használunk, de a '`~`' és a '`!~`' operátorokkal "dinamikus" reguláris kifejezéseket is lehet használni (amik egy reguláris kifejezést tartalmazó egyszerű szövegek vagy változók).

---

[1] A számok belső reprezentációja double. Ha nem tudod, hogy ez mit jelent ne aggódj; ez nem olyan fontos.

7.2. Reguláris kifejezés konstansok használata

Ha a reguláris kifejezés konstans a '`~`' vagy a '`!~`' operátor jobb oldalán áll, akkor magát a reguláris kifejezést jelenti, amit illeszteni szeretnénk.

A reguláris kifejezés konstansok (mint a `/foo/`) használhatók mint egyszerű kifejezések is. Ha a reguláris kifejezés konstans önmagában áll, az megegyezik azzal az esettel, mintha a mintában lett volna megadva, például: '`($0 ~ /foo/)`' (s.s.) (lásd 8.1.3. Bekezdés [Kifejezések mint minták], 98. oldal). Ez azt jelenti, hogy az alábbi két programrészlet

```
if ($0 ~ /barfly/ || $0 ~ /camelot/)
    print "found"
```

és

```
if (/barfly/ || /camelot/)
    print "found"
```

teljesen megegyezik.

Ennek a szabálynak az a furcsa következménye, hogy bár az alábbi kifejezés nem hibás, de nem azt csinálja, mint amit valószínűleg elvárnánk:

```
# figyelem: a /foo/ nem a ~ operátor bal oldalán van
if (/foo/ ~ $1) print "found foo"
```

Elméletileg a `$1` mezőre a `/foo/` reguláris kifejezést próbálja illeszteni. Valójában a '`/foo/ ~ $1`' kifejezés ezzel egyezik meg: '`($0 ~ /foo/) ~ $1`'. Más szavakkal, először a `/foo/` reguláris kifejezést illeszti a teljes rekordra, aminek az eredménye egy vagy zérus attól függően, hogy az illesztés sikerül-e vagy sem. Azután ezt az eredményt próbálja meg illeszteni az első mezőre.

Mivel valószínű, hogy ilyen tesztet soha nem akarsz elvégezni, ezért a `gawk` figyelmeztet ha ilyen szerkezetet talál a programban.

Egy másik következménye a fenti szabálynak, hogy az alábbi értékadás

```
matches = /foo/
```

vagy zérust vagy egyet tárol a `matches` változóban, attól függően, hogy mi az aktuális bemeneti rekord értéke.

Ez az `awk` tulajdonság soha nem volt megfelelően dokumentálva a POSIX szabvány előtt.

Reguláris kifejezés konstansok használhatók a `gensub`, `sub` és `gsub` függvények első argumentumaként és a `match` függvény második argumentumaként (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Az `awk` modern implementációi és a `gawk` megengedi, hogy a `split` függvény harmadik argumentuma reguláris kifejezés konstans legyen. Régebbi `awk` implementációkban ez nem megengedett (s.s.).

Sajnos ez kavarodást okozhat a felhasználó által definiált függvények (lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal) esetén, ha egy reguláris kifejezés konstanst adunk meg mint argumentum, például:

```
function mysub(pat, repl, str, global)
{
    if (global)
        gsub(pat, repl, str)
    else
        sub(pat, repl, str)
    return str
}
{
    ...
    text = "hi! hi yourself!"
    mysub(/hi/, "howdy", text, 1)
    ...
}
```

A példában egy reguláris kifejezés konstanst szeretnénk átadni a `mysub` függvénynek, ami továbbadja azt vagy a `sub` vagy a `gsub` függvénynek. Valójában a `pat` paraméter zérus vagy egy attól függően, hogy a rekord (`$0`) tartalmazza-e a `/hi/` szöveget.

Mivel nem valószínű, hogy az illesztés eredményét szeretnéd átadni mint argumentum, ezért a `gawk` figyelmeztet ha egy reguláris kifejezés konstanst talál egy a felhasználó által definiált függvény argumentum listájában.

7.3. Változók

A változókban olyan értéket tárolhatunk, amelyet a programban később szeretnénk felhasználni. A változókat teljesen szabadon lehet a programon belül manipulálni. Az `awk` parancssorában kezdőértéket adhatunk meg a változóknak.

7.3.1. Változók használata egy programban

A változókkal nevet adhatunk egy értéknek, amire később a név segítségével hivatkozhatunk. Már több példában használtunk változókat. A változó neve betűket, számokat és aláhúzás karaktert tartalmazhat, de nem kezdődhet számjeggyel. A kis- és nagybetűs írásmód fontos, mivel az `a` és az `A` két, független változót jelöl.

Önmagában egy változó neve egy érvényes kifejezés; a változó jelenlegi értékét reprezentálja. A változóknak új értéket adhatunk az *értékadó operátorral* vagy megváltoztathatjuk a *növelő* vagy *csökkentő operátorral*. Lásd 7.7. Bekezdés [Értékadó kifejezések], 84. oldal.

Néhány változónak speciális, beépített jelentése van; például `FS` a mezőelválasztót és az `NF` a mezők számát adja meg. A 10. Fejezet [Beépített változók], 115. oldal, tartalmazza a beépített változók listáját. Ezeket a beépített változókat ugyanúgy használhatjuk mint más változókat, de az `awk` is megváltoztathatja az értéküket. Minden beépített változó neve csupa nagybetűből áll.

Az `awk` változók értéke szám vagy szöveg lehet. Alapesetben minden változó kezdőértéke az üres szöveg, ami zérusnak felel meg ha számmá konvertáljuk. Ezért nincs szükség a változók "inicializálására" az `awk`-ban, mint például a C programozási nyelvben.

7.3.2. Értékadás változóknak a parancssorban

Bármelyik `awk` változónak kezdő érték adható a parancssorban az `awk` parancssori argumentumai között. (lásd 14.2. Bekezdés [Other Command Line Arguments], 165. oldal). Az értékadás formája:

változó`=`*text*

Ilyen formában beállítható egy változó értéke az `awk` futtatása kezdetén. Az értékadás a bemeneti file-ok között is elhelyezhető.

Ha az értékadás előtt a '`-v`' opciót használjuk, például így:

`-v` *változó*`=`*text*

akkor a változót állítja be legelőször, még a `BEGIN` szabály lefutása előtt. A '`-v`' opciónak és az értékadásnak meg kell előznie az összes bemeneti file-t és a program szövegét is. (Lásd 14.1. Bekezdés [Command Line Options], 161. oldal, további információk a '`-v`' opcióról.)

Ellenkező esetben az értékadás csak akkor történik meg, amikor az `awk` a feldolgozásban odaér, vagyis miután feldolgozta a megelőző bemeneti file-t. Például:

```
awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
```

kinyomtatja az `n`-edik mezőt mindegyik bemeneti rekordból. Mielőtt az első file-t elkezdené olvasni beállítja az `n` változó értékét négyre. Ennek hatására az '`inventory-shipped`' file-ból a negyedik mezőt fogja kinyomtatni. Miután befejezte az első file feldolgozását és mielőtt elkezdené feldolgozni a másodikat az `n` változót kettőre állítja, így a '`BBS-list`' file-ból a második mezőt nyomtatja ki.

```
$ awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
⊣ 15
⊣ 24
...
⊣ 555-5553
⊣ 555-3412
...
```

A parancssori argumentumokat explicit módon is meg lehet vizsgálni egy `awk` programban, mivel az `ARGV` tömbben rendelkezésre állnak (lásd 10.3. Bekezdés [Az `ARGC` és az `ARGV` változók használata], 120. oldal).

Az `awk` a parancssori értékadásnál is figyelembe veszi az escape szekvenciákat (s.s.) (lásd 4.2. Bekezdés [Escape szekvenciák], 24. oldal).

7.4. Szövegek és számok konverziója

Szövegek számmá és számok szöveggé konvertálhatók ha az `awk` program úgy kívánja. Például ha vagy a `foo` vagy a `bar` értéke a '`foo + bar`' kifejezésben szöveg értékű, akkor az összeadás előtt először a változó értéke átkonvertálódik számmá. Ha egy szám jelenik meg szöveg összefűzésnél, akkor a számot átkonvertálja szöveggé, így:

```
two = 2; three = 3
print (two three) + 4
```

a program a (numerikus) 27-et fogja kinyomtatni. Először a `two` és `three` változók numerikus értékeit átkonvertálja szöveggé és összefűzi őket, majd az így kapott szöveget visszaalakítja számmá (23) amihez négyet ad.

Ha valamiért egy számot mindenáron szeretnél szöveggé alakítani, akkor hozzá kell fűzni egy üres szöveget, `""`. Ha egy szöveget kell átalakítani számmá, akkor hozzá kell adni zérust.

A szöveg számmá konvertálása úgy történik, hogy a szöveg elején elhelyezkedő értelmes numerikus kifejezést alakítja számmá: `"2.5"` konvertálás után 2.5, `"1e3"` értéke 1000 és `"25fix"` numerikus értéke 25. Olyan szöveg, ami nem értelmezhető számként, a konvertálás után zérus értékű lesz.

A számok szöveggé konvertálását a `CONVFMT` beépített `awk` változó kontrollálja (lásd 10. Fejezet [Beépített változók], 115. oldal). A számokat a `sprintf` függvénnyel (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal) alakítja át, ahol a formátum leírót a `CONVFMT` változó adja meg.

A `CONVFMT` alapértéke a `"%.6g"`, ami legalább hat értékes jegyre nyomtatja ki a számot. Előfordulhat, hogy néhány alkalmazás esetén nagyobb pontossággal szeretnél konvertálni, de vedd figyelembe, hogy a dupla (double) pontosság általában csak 16 vagy 17 értékes jegyet képes tárolni.

Furcsa eredményt kaphatsz, ha a `CONVFMT`-ben nem adod meg a `sprintf` függvénynek, hogy hogyan nyomtasson lebegő pontos számot. Például ha elfelejted megadni a '`%`' karaktert a formátumban, akkor minden szám ugyanarra a konstans szövegre lesz konvertálva.

Egy speciális eset, ha a szám egész, akkor a konvertálás eredményeként kapott szöveg *mindig* egy egész számot fog tartalmazni, attól függetlenül, hogy mi a `CONVFMT` értéke. Például:

```
CONVFMT = "%2.2f"
a = 12
b = a ""
```

`b` értéke `"12"` és nem `"12.00"` (s.s.).

A POSIX szabvány előtt az `awk` az `OFMT` változót használta a számok szöveggé konvertálásánál. Az `OFMT` azt adja meg, hogy a `print` milyen formában nyomtasson ki egy számot. A `CONVFMT`-t pont azért vezették be, hogy elkülönítsék a nyomtatás és a konvertálás formáját. Mindkét változónak (a `CONVFMT` és a `OFMT`) ugyanaz az alapértéke: `"%.6g"`. A legtöbb esetben az öreg `awk` programok viselkedése nem fog megváltozni, de érdemes fejben tartani az `OFMT` ezen specialitását, ha a programodat más `awk` implementációkhoz akarod igazítani. Egyébként ebben az esetben a programod módosítása helyett azt tanácsoljuk, hogy magát a `gawk`-ot fordítsd le és használd. Lásd 6.1. Bekezdés [A `print` kifejezés], 61. oldal, alatt további információ található a `print` kifejezésről.

7.5. Matematikai operátorok

Az `awk` nyelvben a megszokott matematikai operátorokat lehet használni, a precedenciaszabályok sem különbözőek, és pontosan úgy működnek, ahogy az elvárható.

Az alábbi '`grades`' file egy osztályba járó tanulók nevét és három teszt eredményét tartalmazza (ez egy kis osztály):

```
Pat   100 97 58
Sandy  84 72 93
Chris  72 92 89
```

A program kinyomtatja a tanulók átlagát:

```
$ awk '{ sum = $2 + $3 + $4 ; avg = sum / 3
>        print $1, avg }' grades
 ⊣ Pat 85
 ⊣ Sandy 83
 ⊣ Chris 84.3333
```

Az alábbi táblázat felsorolja az `awk`-ban használható matematikai operátorokat:

`-` *x*
: Negálás.

`+` *x*
: Unáris plusz. A kifejezést számmá konvertálja.

x `^` *y*
x `**` *y*
: Hatványozás: *x*-et az *y*-adik hatványra emeli. '`2 ^ 3`' értéke nyolc. A '`**`' karakter azonos a '`^`' karakterrel. (A POSIX szabvány csak a '`^`' karaktert definiálja hatványozó operátorként.)

x `*` *y*
: Szorzás.

x `/` *y*
: Osztás. Mivel minden szám valós az `awk`-ban, ezért az eredmény nem lesz egészre kerekítve: '`3 / 4`' eredménye 0.75.

x `%` *y*
: Maradék számítás. Az osztás eredményét lefelé kerekítve, beszorozva *y*-al és kivonva *x*-ből lesz a végső eredmény. Ez az operátor úgy is ismert mint "trunc-mod" operátor. Az alábbi reláció mindig igaz:

```
b * int(a / b) + (a % b) == a
```

Egy valószínűleg nem kívánatos mellékterméke a fenti definíciónak, hogy ha *x* negatív, akkor *x* % *y* eredménye is negatív lesz, így

```
-17 % 8 = -1
```

Az eredmény előjele más `awk` implementációkban eltérő lehet.

x + *y* Összeadás.

x - *y* Kivonás.

A maximális hordozhatóság érdekében ne használd a '`**`' operátort.

7.6. Szövegösszefűzés

Akkor jó ötletnek tűnt.
Brian Kernighan

Csak egy szöveg operátor van: összefűzés; viszont nincs karakter ami jelölné. Az összefűzéshez egyszerűen egymás mellé kell írni a kifejezéseket, például:

```
$ awk '{ print "Field number one: " $1 }' BBS-list
⊣ Field number one: aardvark
⊣ Field number one: alpo-net
...
```

Ha nem lenne szóköz a kettőspont után, akkor az eredmény így nézne ki:

```
$ awk '{ print "Field number one:" $1 }' BBS-list
⊣ Field number one:aardvark
⊣ Field number one:alpo-net
...
```

Mivel az összefűzésnek nincs explicit operátora, gyakran zárójelek közé kell tenni a kifejezéseket ahhoz, hogy az összefűzés valóban megtörténjen. Például az alábbi példában nem kapcsolja össze a `file` és a `name` tartalmát, mint ahogy azt elvárnánk:

```
file = "file"
name = "name"
print "valami" > file name
```

Így kell leírni helyesen:

```
print "valami" > (file name)
```

Az tanácsoljuk, hogy kivéve a legegyértelműbb helyzeteket, érdemes zárójelek közé tenni az összefűzendő kifejezéseket.

7.7. Értékadó kifejezések

Az *értékadás* egy olyan kifejezés ami új értéket rendel egy változóhoz. Például a `z` változónak így adjuk meg a numerikus egy értéket:

```
z = 1
```

A kifejezés kiértékelése után a `z` változó értéke egy lesz. A `z` változó korábbi értéke elveszik, akármi is volt az.

Az értékadásnál megadhatunk szöveg értéket is. Például az alábbi kifejezés a `"this food is good"` értéket tárolja el a `message` változóban:

```
thing = "food"
predicate = "good"
message = "this " thing " is " predicate
```

(A példa mutatja a szöveg összefűzést is.)

Az '`=`' (egyenlőség) jel az *értékadó operátor*. Ez a legegyszerűbb értékadó operátor, mivel a jel jobb oldalán álló értéket változtatás nélkül tárolja a változóban.

A legtöbb operátornak (mint összeadás, összefűzés, stb) nincs más hatása csak az, hogy az adott értéket kiszámolja. Ha nincs szükséged az értékre akkor akár ne is használd az adott operátort. Az értékadó operátor ettől különböző; bár a jobb oldal kiértékelésével megkapott értékre elméletileg mondhatod, hogy nincs szükséged, de a változóban mindenképpen el fogja tárolni. Ezt *mellékhatásnak* hívják.

Az értékadás bal oldalán nem kötelező egy változónak állnia (lásd 7.3. Bekezdés [Változók], 79. oldal); ez éppen lehet egy mező (lásd 5.4. Bekezdés [Mező tartalmának megváltoztatása], 42. oldal) vagy egy tömb eleme (lásd 11. Fejezet [Tömbök az `awk`-ban], 123. oldal). Ezeket *lvalue*-nak hívják mivel az értékadó operátor bal oldalán állhatnak. A jobb oldali kifejezés bármilyen kifejezés lehet, ennek az értékét tárolja egy változóban, mezőben vagy egy tömb elemben. (Az ilyen értéket *rvalue*-nak hívják.)

Fontos megjegyezni, hogy a változóknak *nincs* állandó típusa. A változó típusát az adja meg, hogy éppen milyen értéket tárol. A következő program részletben a `foo` változónak először szám értéke van majd az értéke szöveg lesz:

```
foo = 1
print foo
foo = "bar"
print foo
```

Amikor a második alkalommal a `foo` egy szöveg értéket kap, akkor az előző szám értékét teljesen elfelejti.

Ha egy szöveg nem számmal kezdődik, akkor a numerikus értéke zérus. Így az alábbi kód végrehajtása után a `foo` értéke öt:

```
foo = "a string"
foo = foo + 5
```

(Figyelem, ha egy változónak néha szám és néha szöveg értéke van, az zavaró lehet és rossz programozási stílus. A fenti példa azt mutatja be, hogy az `awk` hogyan működik és *nem* azt, hogy hogyan kell programot írni!)

Az értékadás is egy kifejezés és az értéke megegyezik a jobb oldal kiértékelés utáni értékével. Így a '`z = 1`' mint kifejezés értéke egy. Ennek egyik következménye, hogy többszörös értékadást is lehet egymás után írni:

```
x = y = z = 0
```

eredménye, hogy mind a három változó értéke zérus lesz, mivel a '`z = 0`' értéke zérus, amit az '`y`' változóban tárol, majd a '`y = z = 0`' zérus értékét tárolja el az `x` változóban.

Értékadás minden olyan helyen használható, ahol kifejezés szerepelhet. Például ez is érvényes '`x != (y = 1)`', ami először egyet rendel az `y`-hoz, majd ellenőrzi, hogy az `x` értéke egy-e. Ugyanakkor ezt a programozási stílust nehéz olvasni; az egyszer használatos programok kivételével beágyazott értékadást nem érdemes használni.

Bár az '`=`' operátor nem, de más operátorok felhasználják a változó régi értékét. Például, a '`+=`' operátor a változó régi értékéhez hozzáadja a jobb oldal értékét, majd az így kapott új értéket tárolja el a változóban. Így az alábbi kifejezés ötöt ad a `foo` értékéhez:

```
foo += 5
```

ami megegyezik ezzel:

```
foo = foo + 5
```

Azt használd, amelyik jobban olvasható/érthető számodra.

Vannak olyan esetek, amikor a '`+=`' operátor (vagy bármilyen más értékadó operátor) *nem* ugyanazt csinálja mint amikor a bal oldali változó a jobb oldalon is szerepel, például:

```
# Köszönet Pat Rankin-nak ezért a példáért
BEGIN  {
    foo[rand()] += 5
    for (x in foo)
       print x, foo[x]

    bar[rand()] = bar[rand()] + 5
    for (x in bar)
       print x, bar[x]
}
```

A `bar` indexei garantáltan különbözőek lesznek, mivel a `rand` minden alkalommal más értékkel tér vissza. (A tömböket és a `rand` függvényt eddig még nem tárgyaltuk, lásd 11. Fejezet [Tömbök az `awk`-ban], 123. oldal, és lásd még 12.2. Bekezdés [Numerikus beépített függvények], 134. oldal, további információkért).

Ez a példa az értékadás operátorok egy másik fontos tulajdonságát is demonstrálja: a bal oldali kifejezés csak *egyszer* lesz kiértékelve.

Az implementációtól függ, hogy melyik kifejezés értékelődik ki először, a jobb vagy a bal oldali kifejezés, például:

```
i = 1
a[i += 2] = i + 1
```

Az `a[3]` értéke kettő vagy négy is lehet.

Az alábbi táblázat összefoglalja az értékadó operátorokat. Mindegyik esetben a jobb oldali kifejezést kiértékeli és ha szükséges számmá konvertálja az `awk`.

lvalue `+=` *increment*
: Hozzáadja az *increment*-et az *lvalue* régi értékéhez majd ezt az új értéket tárolja az `lvalue`-ban.

lvalue `-=` *decrement*
: Kivonja a *decrement*-et az *lvalue*-ból.

lvalue `*=` *coefficient*
: Az *lvalue*-t megszorozza a *coefficient*-el.

lvalue `/=` *divisor*
: Az *lvalue*-t elosztja a *divisor*-al.

lvalue `%=` *modulus*
: Az *lvalue* és a *modulus* osztásának maradékát számítja ki.

lvalue `^=` *power*
lvalue `**=` *power*
: Az *lvalue*-t a *power* hatványra emeli. (Csak a '`^=`' operátor POSIX kompatíbilis.)

Maximális hordozhatóság érdekében ne használd a '`**=`' operátort.

7.8. Növelő és csökkentő operátorok

A *növelő* és a *csökkentő* operátorok eggyel növelik vagy csökkentik a változó értékét. Ugyanezt megteheted értékadó operátorral is, így ez a két új operátor nem ad semmi újat az `awk` nyelvhez, de kényelmes rövidítése egy gyakran használt műveletnek.

A növelő operátor a '`++`', használható mielőtt vagy miután a kifejezés értékét megkaptuk.

Ha a '`++`*v*'-t használjuk, akkor egyet ad a változóhoz és ez lesz a kifejezés értéke is. Ez teljesen azonos a '*v* `+= 1`' kifejezéssel.

Ha a '`++`'-t a változó után írjuk, akkor bár ez is eggyel növeli a változó értékét, de az a különbség, hogy a kifejezés értéke a változó *régi* értéke lesz. Így, ha a `foo` értéke négy, akkor a '`foo++`' kifejezés értéke is négy, de a '`foo`' változó ötre változik.

A '`foo++`' majdnem azonos a '`(foo += 1) - 1`' kifejezéssel. Nem egészen azonos, mivel az `awk`-ban minden szám lebegőpontos (valós): így '`foo + 1 -`

1' nem biztos, hogy tökéletesen megegyezik 'foo'-val. Ez a különbség nem vehető észre ha "kis" számokat használsz (kisebb mint 10e12).

Bármilyen 'lvalue' növelhető. Mezőket és tömb elemeit pontosan úgy növel mint változókat. (Ha egy mezőre akarsz hivatkozni és ugyanakkor a változó értékét növelni a '$(i++)' kifejezést használd. A zárójelek fontosak a '$' precedenciája miatt.)

A csökkentő operátor '--' ugyanúgy viselkedik mint a növelő, csak kivon egyet a változóból. Mint a '++', használható az 'lvalue' előtt vagy után.

Az alábbiakban összefoglaljuk a növelő és csökkentő operátorok használatát.

++*lvalue* Ez a kifejezés megnöveli az 'lvalue' értékét és az új érték lesz a teljes kifejezés értéke is.

lvalue++ Ez a kifejezés megnöveli az 'lvalue' értékét és a *régi* érték lesz a teljes kifejezés értéke.

--*lvalue* Ez a kifejezés lecsökkenti az 'lvalue' értékét és az új érték lesz a teljes kifejezés értéke is.

lvalue-- Ez a kifejezés lecsökkenti az 'lvalue' értékét és az 'lvalue' *régi* értéke lesz a teljes kifejezés értéke.

7.9. Igaz és hamis az awk-ban

Sok programozási nyelvben speciális reprezentációja van az "igaz" és a "hamis" értékeknek. Ezek a nyelvek általában speciális konstanst használnak, például `true` és `false` vagy `TRUE` és `FALSE`.

Az `awk` ettől különböző, ugyanazt az egyszerű megoldást használja mint a C programozási nyelv. Az `awk`-ban, bármely nem zérus szám vagy nem üres szöveg igaz értéket képvisel. A zérus szám és az üres szöveg `""` hamis. Az alábbi program háromszor írja ki a '`Egy furcsa igaz érték`' szöveget:

```
BEGIN {
   if (3.1415927)
       print "Egy furcsa igaz érték"
   if ("Four Score And Seven Years Ago")
       print "Egy furcsa igaz érték"
   if (j = 57)
       print "Egy furcsa igaz érték"
}
```

A "nem zérus vagy nem üres szöveg" szabálynak van egy érdekes következménye: A `"0"` szöveg konstans valójában igaz, mivel nem üres szöveg (s.s.).

7.10. Változó típusok és az összehasonlító kifejezések

Az útikönyv pontos. A valóság gyakran nem egzakt.
Galaxis útikönyv stopposoknak

Más programozási nyelvekkel szemben az `awk` változóknak nincs fix típusuk, lehetnek számok vagy szövegek, attól függően, hogy mi az értékük.

Az 1992-es POSIX szabvány bevezette a *szám-szöveg* (strnum) koncepciót; ez egyszerűen egy szöveg ami úgy néz ki mint egy szám, például `" +2"`. E koncepció segítségével lehet meghatározni a változó típusát.

A változó típusa azért fontos, mert a típus határozza meg, hogy két változó hogyan lesz összehasonlítva.

A `gawk`-ban az alábbi szabályok érvényesek.

1. Egy számnak vagy egy matematikai műveletnek *szám* attribútuma van.
2. Egy szövegnek vagy egy szöveges műveletnek *szöveg* attribútuma van.
3. Mezőknek, a `getline` input-nak, a `FILENAME`-nek, az `ARGV` elemeinek, az `ENVIRON` elemeinek és a `split` által készített tömbök olyan elemeinek aminek szám-szöveg értéke van az attribútuma *strnum*. Minden egyéb esetben az attribútum *szöveg*. Nem inicializált változók attribútuma is *strnum*.
4. Az attribútum átadódik az értékadással, de a változó egyszerű használatával nem változik meg.

Az utolsó szabály különösen fontos. A következő programban `a`-nak szám értéke van, még akkor is ha később egy szöveges műveletben használjuk.

```
BEGIN {
          a = 12.345
          b = a " is a cute number"
          print b
}
```

Amikor két operandust hasonlítunk össze vagy szöveges vagy szám összehasonlítás hajtódik végre, az operandusok típusától függően, az alábbi szimmetrikus táblázat szerint:

	SZÖVEG	SZÁM	STRNUM
SZÖVEG	szöveg	szöveg	szöveg
SZÁM	szöveg	szám	szám
STRNUM	szöveg	szám	szám

Az alapkoncepció az, hogy a felhasználó által megadott bemenetet ami számnak néz ki, és *csak* a felhasználói bemenetet, számként kell kezelni még akkor is ha karakterekből áll, és ezért szöveg lenne.

Az *összehasonlító kifejezések* a szövegek és a számok közötti kapcsolatot ellenőrzik, például egyenlőségüket. Az összehasonlító kifejezéseket egy *összehasonlító operátorral* írjuk le, amelyek a C nyelvben található operátorokkal felülről kompatíbilisek. Íme az összehasonlító operátorok táblázata:

x `<` *y* Igaz, ha *x* kisebb mint *y*.

x `<=` *y* Igaz, ha *x* kisebb vagy egyenlő mint *y*.

x `>` *y* Igaz, ha *x* nagyobb mint *y*.

x `>=` *y* Igaz, ha *x* nagyobb vagy egyenlő mint *y*.

x `==` *y* Igaz, ha *x* egyenlő *y*-al.

x `!=` *y* Igaz, ha *x* nem egyenlő *y*-al.

x `~` *y* Igaz, ha *x* illeszkedik az *y* által megadott reguláris kifejezésre.

x `!~` *y* Igaz, ha *x* nem illeszkedik az *y* által megadott reguláris kifejezésre.

subscript `in` *array*
Igaz, ha az *array* tömbnek van *subscript* indexű eleme.

Az összehasonlító kifejezések értéke egy ha igaz, és zérus ha hamis.

Amikor különböző típusú komponenseket hasonlítunk össze, akkor a szám értékeket átalakítja szöveggé a `CONVFMT` változó értékét használva. (lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal).

A szövegek összehasonlításánál először az első karaktereket hasonlítja össze, majd a második karaktereket és így tovább. Így `"10"` kisebb mint `"9"`. Két olyan szöveg esetén, amikor az egyik szöveg eleje teljesen megegyezik a második szöveggel, akkor a rövidebb szöveg számít kisebbnek. Így `"abc"` kisebb mint `"abcd"`.

Nagyon könnyü véletlenül elgépelni a '`==`' operátort és az egyik ('`=`') egyenlőség jelet elhagyni. Az eredmény szintén érvényes `awk` kód, de a program nem azt fogja csinálni mint amit szeretnél.

```
if (a = b)   # hoppá ! a == b kellene
    ...
else
    ...
```

Hacsak `b` nem zérus vagy üres szöveg, az `if` feltételes kifejezés mindig igaz lesz. Az ilyen hibát sajnos nagyon nehéz észrevenni a forráskód átnézése során.

Alább bemutatunk néhány kifejezést, ami bemutatja, hogy a `gawk` hogyan végzi az összehasonlítást és milyen eredményt kapunk:

`1.5 <= 2.0`
numerikus összehasonlítás (igaz)

`"abc" >= "xyz"`
szöveg összehasonlítás (hamis)

`1.5 != " +2"`
szöveg összehasonlítás (igaz)

`"1e2" < "3"`
szöveg összehasonlítás (igaz)

`a = 2; b = "2"`
`a == b` szöveg összehasonlítás (igaz)

`a = 2; b = " +2"`
`a == b` szöveg összehasonlítás (hamis)

Ebben a példában,

```
$ echo 1e2 3 | awk '{ print ($1 < $2) ? "true" : "false" }'
⊣ false
```

az eredmény hamis, mivel `$1` és `$2` szám-szövegek, így mindkettő típusa *strnum*, ami szám összehasonlítást eredményez.

Az összehasonlítási szabályok és a szám-szövegek célja, hogy a program a "lehető legkisebb meglepetést" okozva a felhasználó által "elvárt, jó dolgot csinálja".

A szöveg és reguláris kifejezések összehasonlítása teljesen különböző. Például:

```
x == "foo"
```

értéke egy, vagyis igaz, ha az `x` változó értéke pontosan '`foo`'. Ezzel ellentétben, az

```
x ~ /foo/
```

értéke egy, ha az `x` változó tartalmazza a '`foo`' szöveget, úgy mint `"Oh, what a fool am I!"`.

A '`~`' és '`!~`' operátorok jobb oldalán álló kifejezés lehet egy regexp konstans (`/.../`) vagy egy általános kifejezés amikor is a kifejezés értékét mint szöveget használja egy dinamikus reguláris kifejezésként (lásd 4.1. Bekezdés [Hogyan használjuk a reguláris kifejezéseket], 23. oldal; és lásd 4.7. Bekezdés [Dinamikus reguláris kifejezések használata], 35. oldal).

Az `awk` jelenlegi implementációjában egy konstans reguláris kifejezés a '`/`' jelek között szintén általános kifejezésnek számít. A `/`*regexp*`/` csak egy rövidítése az összehasonlító kifejezésnek:

`$0 ~ /`*regexp*`/`

Egy speciális eset, amikor a `/foo/` *nem* a '`$0 ~ /foo/`' kifejezés rövidítése, ha a reguláris kifejezés a '`~`' vagy a '`!~`' operátor jobb oldalán áll! Lásd 7.2. Bekezdés [Reguláris kifejezés konstansok használata], 78. oldal, ahol ezt az esetet részletesen tárgyaltuk.

7.11. Logikai kifejezések

A *logikai kifejezések* összehasonlító vagy illesztő kifejezések kombinációja a "vagy" ('`||`'), az "és" ('`&&`') és a "negálás" ('`!`') operátorokkal illetve a zárójelek segítségével. A logikai kifejezések igazság értéke a komponens kifejezések igazság értékének összekombinálásával kapható meg. A logikai kifejezéseket *boolean kifejezésnek* is hívják. A két név teljesen egyenértékű.

Logikai kifejezés használható minden olyan helyen, ahol összehasonlító és illesztő kifejezés állhat. Szerepelhetnek az `if`, a `while`, a `do` és a `for` kifejezésekben (lásd 9. Fejezet [Vezérlésátadó kifejezések a tevékenységekben], 105. oldal). A logikai kifejzés értéke szám (egy ha igaz és zérus ha hamis), ami akkor játszik fontos szerepet ha a logikai kifejezés eredményét egy változóban tároljuk vagy aritmetikai kifejezésben használjuk.

Ráadásul minden logikai kifejezés egy érvényes minta is, így használható mint minta és így képes a szabályok végrehajtását is befolyásolni.

A három logikai operátor leírását néhány példával alább közöljük:

boolean1 `&&` *boolean2*
: Igaz, ha a *boolean1* és a *boolean2* igaz. Például az alábbi kifejezés kinyomtatja az aktuális rekordot, ha az tartalmaza a '`2400`' és a '`foo`' szövegeket.

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

A második kifejezés (*boolean2*) csak akkor értékelődik ki, ha a *boolean1* igaz. Ez akkor lehet fontos, ha a *boolean2* kiértékelésének van mellékhatása: a '`$0 ~ /foo/ && ($2 == bar++)`' kifejezés esetén a `bar` változó nem növelődik meg ha a `foo` szöveg nem szerepel a rekordban.

boolean1 `||` *boolean2*
: Igaz, ha vagy a *boolean1* vagy a *boolean2* igaz. Például az alábbi kifejezés kinyomtat minden olyan rekordot ami *vagy* a '`2400`', *vagy* a '`foo`', *vagy* mind a két szöveget tartalmazza.

```
if ($0 ~ /2400/ || $0 ~ /foo/) print
```

A második kifejezés (*boolean2*) csak akkor értékelődik ki, ha a *boolean1* hamis. Ez akkor lesz fontos ha a *boolean2* kifejezés egy olyan kifejezést tartalmaz aminek mellékhatása van.

! *boolean*
: Igaz, ha a *boolean* kifejezés hamis. Például az alábbi program a '`BBS-file`' file-ból kinyomtat minden olyan rekordot, ami *nem* tartalmazza a '`foo`' szöveget.

```
awk '{ if (! ($0 ~ /foo/)) print }' BBS-list
```

A '`&&`' és a '`||`' operátorokat *rövidre zárható* operátoroknak is nevezik. A teljes kifejezés kiértékelése "rövidre záródik", befejeződik, ha az eredmény a kifejezés egy részének kiértékelésével már meghatározható.

A logikai kifejezések sorokra bonthatók a '`&&`' és a '`||`' operátorok után beszúrt új sor karakterrel. Ugyanakkor az új sor karakter nem használható az operátorok előtt a '`\`' karakter nélkül (lásd 2.6. Bekezdés [`awk` kifejezések és sorok], 16. oldal).

A '!' operátort tartalmazó kifejezések értéke vagy egy vagy zérus lehet, attól függően, hogy milyen igazságértékű kifejezésre lett alkalmazva az operátor.

A '!' operátor gyakran használható kapcsoló változók átállítására igazról hamisra és fordítva. Például, az alábbi program az egyik módja annak, hogy speciális zárójelek közti sorokat kinyomtassunk:

```
$1 == "START"   { interested = ! interested }
interested == 1 { print }
$1 == "END"     { interested = ! interested }
```

Az `interested` változó, mint minden `awk` változó, kezdetben zérus értékű, ami hamis értékű. Amikor egy '`START`' szöveggel kezdődő sort olvas be, akkor az `interested` változót átállítja igazra a '`!`' operátorral. A következő szabály addig nyomtatja ki a sorokat amíg a `interested` változó igaz. Amikor egy '`END`' kezdetű sort olvas be, a `interested` változót visszaállítja hamisra.

7.12. Feltételes kifejezések

A *feltételes kifejezések* olyan speciális kifejezések aminek három operandusa van. Ennek segítségével egy kifejezés értékétől függően az egyik vagy egy másik kifejezés hajtódik végre.

A feltételes kifejezés ugyanolyan mint a C nyelvben:

választás ? *ha-igaz-kif* : *ha-hamis-kif*

Három alkifejezésből áll. Az első, a *választás*, értékelődik ki legelőször. Ha az értéke "igaz" (nem zérus és nem üres szöveg) akkor a *ha-igaz-kif* kifejezés lesz kiértékelve és ennek az értéke lesz a teljes kifejezés értéke. Más esetben a *ha-hamis-kif* lesz kiértékelve és ennek az értéke adja a teljes kifejezés értékét.

Például, ez a kifejezés az `x` változó abszolút értékét adja meg:

```
x > 0 ? x : -x
```

Minden alkalommal, amikor egy feltételes kifejezés kiértékelődik, pontosan egy kifejezés, vagy a *ha-igaz-kif* vagy a *ha-hamis-kif* értékelődik ki; a másikat nem veszi figyelembe. Ez akkor fontos, ha a kifejezéseknek mellékhatásuk van. Például ez a feltételes kifejezés vagy az `a` vagy a `b` tömb `i`-edik indexét vizsgálja meg és az `i`-t megnöveli.

```
x == y ? a[i++] : b[i++]
```

Ez a kifejezés garantáltan csak egyszer növeli meg a `i` értékét, mivel minden alkalommal a két növelő kifejezés közül csak az egyik hajtódik végre, és a másik nem. Lásd 11. Fejezet [Tömbök az `awk`-ban], 123. oldal, további információ a tömbökről.

Egy apró `gawk` módosítás, hogy egy új sor karakter beszúrható a '`?:`' karakterek bármelyike után, és így több sorba is írható a kifejezés. Ugyanakkor új sor karakter nem használható előttük kivéve ha a '\' karaktert használjuk (lásd 2.6. Bekezdés [`awk` kifejezések és sorok], 16. oldal). Ha a '`--posix`' opció meg van adva (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor ez a kiegészítés nem használható.

7.13. Függvényhívások

A *függvény* egy nevet rendel egy adott számításhoz. Mivel névvel rendelkezik, ezért bármikor meghívható a programból. Például az `sqrt` függvény egy szám négyzetgyökét számítja ki.

Egy adott számú *beépített* függvény áll rendelkezésre minden `awk` programban. Az `sqrt` függvény ezek egyike. Lásd 12. Fejezet [Beépített függvények], 133. oldal, ami a beépített függvények listáját és leírását tartalmazza. Ráadásul saját függvényeket is lehet definiálni. Lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal, hogy hogyan kell definiálni új függvényeket.

A függvényeket *függvényhíváson* keresztül lehet használni, ami a függvény nevét és a közvetlenül utána álló *argument* listát tartalmazza zárójelekben. Az argumentumok olyan kifejezések, amelyek extra adatot biztosítanak a függvénybeli számításhoz. Ha egynél több argumentum van, akkor az argumentumokat vesszővel kell elválasztani. Ha a függvénynek nincs argumentuma, akkor egy üres zárójelet kell írni, '`()`'. Íme néhány példa:

`sqrt(x^2 + y^2)`	*egy argumentum*
`atan2(y, x)`	*két argumentum*
`rand()`	*nincs argumentum*

Nem szabad szóköz karaktert tenni a függvény neve és a nyitó zárójel közé! A felhasználó által definiált függvény nevek pont úgy néznek ki mint a változók nevei és ezért a szóközzel úgy lenne értelmezve mintha egy változót összefűznénk a zárójelek közti kifejezéssel. Beépített függvények esetén egy szóköz teljesen ártalmatlan a zárójel előtt, de a legjobb ha nem szoksz hozzá, mert így elkerülheted ezt a hibát az általad definiált függvények esetén.

Minden függvény egy adott számú argumentumot kíván. Például az `sqrt` függvényt csak egyetlen argumentummal kell meghívni, azt a számot kell megadni, aminek a négyzetgyökét ki akarjuk számolni:

`sqrt(`*argument*`)`

Néhány beépített függvény esetén az utolsó argumentum elhagyható. Ha az utolsó argumentumot nem adod meg, akkor egy ésszerű alapbeállítást használ a program. Lásd 12. Fejezet [Beépített függvények], 133. oldal, a teljes részletekről. Ha egy a felhasználó által definiált függvény argumentumaiból elhagyunk néhányat, akkor azok az argumentumok, mint lokális változók lesznek kezelve és üres szöveg lesz a kezdő értékük (lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal).

Mint minden más kifejezésnek, a függvényhívásnak is van értéke, amit a függvény számol ki az argumentumok alapján. Ebben a példában, a '`sqrt(`*argument*`)`' kifejezés kiszámolja az *argument* négyzetgyökét. A függvénynek is lehet mellékhatása, mint például bizonyos változókhoz értéket rendelni vagy I/O végrehajtása.

Itt egy parancs számok beolvasására, egy szám soronként majd kinyomtatja mindegyik beolvasott szám négyzetgyökét:

```
$ awk '{ print "The square root of", $1, "is", sqrt($1) }'
1
⊣ The square root of 1 is 1
3
⊣ The square root of 3 is 1.73205
5
⊣ The square root of 5 is 2.23607
Control-d
```

7.14. Operátorok precedenciája (Hogyan ágyazhatók operátorok egymásba)

Az *operátorok precedenciája* azt határozza meg, hogy az operátorokat hogyan lehet csoportosítani egy kifejezésben. Például a '`*`' operátornak magasabb a precedenciája mint a '`+`' operátornak; így, a '`a + b * c`' kifejezés esetén a `b` és a `c` változókat összeszorozza, majd azután hozzáadja az `a` változót (például: '`a + (b * c)`').

Az operátorok precedenciáját felül lehet bírálni a zárójelek használatával. Úgy is gondolhatsz a precedenciaszabályokra, mint amelyek meghatározzák azt, hogy a zárójeleknek hol kellene lenniük, ha nem írod ki őket. Valójában hasznos a zárójeleket mindig használni, ha az operátorok valamilyen szokatlan kombinációját használjuk, mivel más emberek, akik a programodat olvassák, később lehet hogy nem emlékeznének, hogy mi is a helyes precedencia az adott esetben. Lehet, hogy Te is elfelejted; így te is hibázhatsz. A zárójelek használatával az ilyen hibák elkerülhetők.

Amikor azonos precedenciájú operátorokat használunk, a legbaloldalibb operátor alapján lesz először csoportosítva a kifejezés, kivéve persze az értékadás, a feltételes és az exponenciális operátorok, amelyeknél a sorrend fordított. Így a '`a - b + c`' csoportosítása '`(a - b) + c`' és a '`a = b = c`' csoportosítása '`a = (b = c)`'.

Az unáris operátorok esetén a precedencia nem számít egészen addig amig csak unáris operátorokat használunk, mivel csak egyféleképpen lehet értelmezni őket – a legbelsőt először. Így a '`$++i`' azt jelenti, hogy '`$(++i)`' és a '`++$x`' jelentése '`++($x)`'. Ugyanakkor, amikor egy másik operátor áll az operandus után, akkor az unáris operátor precedenciája is fontos. Így '`$x^2`' jelentése '`($x)^2`', de '`-x^2`' azt jelenti, hogy '`-(x^2)`', mivel a '`-`' operátornak alacsonyabb precedenciája van mint a '`^`' operátornak, míg a '`$`' operátornak magasabb a precedenciája.

Az alábbi táblázat tartalmazza az `awk` operátorok listáját a legmagasabb precedenciától az alacsonyabb felé:

`(...)`
Csoportosítás.

`$`
Mező.

`++ --`
Növelés, csökkentés.

`^ **`
Exponenciális operátor. Ezek az operátorok jobbról balra csoportosíthatók. (A '`**`' operátor nem POSIX kompatíbilis.)

`+ - !`
unáris plusz, mínusz, logikai "negálás".

`* / %`
Szorzás, osztás, maradékképzés.

`+ -`
Összeadás, kivonás.

Összefűzés
nincs speciális jel az összefűzés jelölésére. Az operátorokat egyszerűen egymás után kell írni.

`< <= == !=`
`> >= >> |`
Összehasonlítás és átirányítás. Az összehasonlítás operátornak és az átirányításnak azonos precedenciája van. A '`>`' karakter mind összehasonlítás operátorként, mind pedig átirányításként szolgál; az adott helyzet határozza meg a jelentését.

Az átirányítás operátor nem generál egy újabb kifejezést, amit mint operandus használhatunk egy másik operátorral. Az eredmény az, hogy nincs értelme az átirányítás operátort használni egy alacsonyabb precedenciájú operátor közelében zárójelek nélkül. Az ilyen kombináció, mint például a '`print foo > a ? b : c`' kifejezés szintaktikus hibát generál. A kifejezés helyes írásmódja '`print foo > (a ? b : c)`'.

`~ !~`
Illeszkedés, nem-illeszkedés.

`in`
Tömbhöz tartozás kifejezése (Tömb eleme).

`&&`
Logikai "és".

`||`
Logikai "vagy".

`?:`
Feltételes kifejezés. Ez az operátor jobbról balra csoportosítható.

`= += -= *=`
`/= %= ^= **=`
Értékadás. Ezek az operátorok jobbról balra csoportosíthatók. (A '`**=`' operátor nem POSIX kompatíbilis.)

8. Minták és tevékenységek

Mint azt már láttuk, minden `awk` szabály tartalmaz egy mintát és egy ahhoz kapcsolódó tevékenységet. Ez a fejezet azt mutatja be, hogy hogyan kell mintákat és tevékenységeket összeállítani.

8.1. Minta elemek

Az `awk`-ban a minták határozzák meg a szabályok végrehajtási sorrendjét: a szabály végrehajtódik, ha a minta illeszkedik az adott bemeneti rekordra. Ez a bekezdés elmagyarázza, hogy hogyan írjunk mintákat.

8.1.1. Minta típusok

Az alábbi táblázat felsorolja az `awk`-ban használható minták típusát.

/*reguláris kifejezés*/
: Egy reguláris kifejezés, mint minta, illeszkedik minden olyan bemeneti rekordra ami illeszkedik a reguláris kifejezés által megadott szövegre. (Lásd 4. Fejezet [Reguláris kifejezések], 23. oldal.)

kifejezés
: Egy kifejezés illeszkedik ha az értéke nem zérus (ha egy szám) vagy nem üres szöveg. (Lásd 8.1.3. Bekezdés [Kifejezések mint minták], 98. oldal.)

pat1, *pat2*
: Egy minta pár, vesszővel elválasztva egy rekord tartományt határoz meg. A tartomány tartalmazza a kezdeti rekordot ami illeszkedik a *pat1*-re és a végső rekordot ami illeszkedik a *pat2*-re. (Lásd 8.1.4. Bekezdés [Rekord tartományok definiálása mintákkal], 99. oldal.)

`BEGIN`
`END`
: Speciális minták kezdeti vagy végső tevékenységek definiálásához `awk` programokban. (Lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal.)

üres
: Az üres minta illeszkedik minden bemeneti rekordra. (Lásd 8.1.6. Bekezdés [Az üres minta], 102. oldal.)

8.1.2. Reguláris kifejezések mint minták.

Már eddig is használtunk reguláris kifejezéseket mintaként a példákban. Ez a típusú minta egy egyszerű regexp konstans a szabály minta pozíciójában, és jelentése megfelel a '`$0 ~ /`*pattern*`/`' kifejezésnek. A minta illeszkedik, amikor a bemeneti rekordra illeszkedik a regexp. Például:

```
/foo|bar|baz/  { buzzwords++ }
END            { print buzzwords, "buzzwords seen" }
```

8.1.3. Kifejezések mint minták

Bármilyen `awk` kifejezés érvényes `awk` minta és a minta illeszkedik ha a kifejezés értéke nem zérus (ha egy szám) vagy nem üres szöveg.

A kifejezések minden alkalommal kiértékelődnek, amikor a szabályt teszteli egy új bemeneti rekorddal. Ha a kifejezés mezőket használ mint `$1`, az érték közvetlenül függ az új bemeneti rekord szövegétől; másképpen attól függ, hogy mi történt eddig az `awk` program végrehajtása során.

Egy nagyon gyakori kifejezés mint minta az összehasonlító kifejezés, ami összehasonlító operátort használ, lásd 7.10. Bekezdés [Változó típusok és az összehasonlító kifejezések], 88. oldal.

A regexp illeszkedés és nem illeszkedés is gyakori kifejezések. A '`~`' és a '`!~`' operátorok bal oldali operandusa egy szöveg. A jobb oldali operandus vagy egy konstans reguláris kifejezés a '`/`' karakterek közé zárva (`/`*regexp*`/`) vagy bármilyen kifejezés aminek a szöveg értéke mint dinamikus reguláris kifejezés használható (lásd 4.7. Bekezdés [Dinamikus reguláris kifejezések használata], 35. oldal).

A következő példa minden olyan bemeneti rekord második rekordját kinyomtatja aminek az első mezője pontosan a '`foo`' szöveg.

```
$ awk '$1 == "foo" { print $2 }' BBS-list
```

(Nincs kimenet, mivel nincs "foo" nevű BBS állomás.) Ezzel ellentétben nézzük az alábbi reguláris kifejezést, ami bármilyen rekordra illeszkedik aminek az első mezője '`foo`' szöveget tartalmaz.

```
$ awk '$1 ~ /foo/ { print $2 }' BBS-list
⊣ 555-1234
⊣ 555-6699
⊣ 555-6480
⊣ 555-2127
```

A logikai kifejezések szintén gyakran használt minták, és hogy egy minta illeszkedik egy bemeneti rekordra attól függ, hogy az alkifejezés illeszkedik-e.

Például az alábbi parancs kinyomtat minden olyan rekordot a '`BBS-list`' file-ból, ami tartalmazza a '`2400`' és a '`foo`' szövegeket.

```
$ awk '/2400/ && /foo/' BBS-list
⊣ fooey        555-1234     2400/1200/300     B
```

Az alábbi parancs kinyomtat minden olyan rekordot a '`BBS-list`' file-ból, ami *vagy* a '`2400`' *vagy* a '`foo`' *vagy* mindkettő szöveget tartalmazza.

```
$ awk '/2400/ || /foo/' BBS-list
⊣ alpo-net     555-3412     2400/1200/300     A
⊣ bites        555-1675     2400/1200/300     A
⊣ fooey        555-1234     2400/1200/300     B
⊣ foot         555-6699     1200/300          B
⊣ macfoo       555-6480     1200/300          A
⊣ sdace        555-3430     2400/1200/300     A
⊣ sabafoo      555-2127     1200/300          C
```

Az alábbi parancs kinyomtat minden olyan rekordot a '`BBS-list`' file-ból, ami *nem* tartalmazza a '`foo`' szöveget.

```
$ awk '! /foo/' BBS-list
⊣ aardvark     555-5553     1200/300          B
⊣ alpo-net     555-3412     2400/1200/300     A
⊣ barfly       555-7685     1200/300          A
⊣ bites        555-1675     2400/1200/300     A
⊣ camelot      555-0542     300               C
⊣ core         555-2912     1200/300          C
⊣ sdace        555-3430     2400/1200/300     A
```

Egy mintában előforduló logikai operátor alkifejezései lehetnek konstans reguláris kifejezések, összehasonlító vagy bármilyen `awk` kifejezések. A tartomány minták nem kifejezések, így nem szerepelhetnek logikai kifejezésekben. A `BEGIN` és az `END` speciális minták soha nem illeszkednek bemeneti rekordra, nem kifejezések és nem szerepelhetnek logikai mintában.

Egy regexp konstans mint minta szintén egy speciális kifejezés minta. A `/foo/` mint kifejezésnek egy az értéke ha a '`foo`' szöveg előfordul a jelenlegi bemeneti rekordban; így, mint minta, a `/foo/` illeszkedik minden olyan rekordra ami tartalmazza a '`foo`' szöveget.

8.1.4. Rekord tartományok definiálása mintákkal

A *tartomány minta* két vesszővel elválasztott mintából áll az alábbi formában '*kezdminta*`,` *végminta*', és a bemeneti rekordok egymás utáni tartományára illeszkedik. Az első minta, *kezdminta*, adja meg a tartomány kezdetét és a második, *végminta*, határozza meg a végét. Például:

```
awk '$1 == "on", $1 == "off"'
```

kinyomtat minden rekordot az '`on`'/'`off`' pár között, a kezdő és a záró sorokat is beleértve.

Tartomány minta esetén először a *kezdminta* mintát keresi meg minden bemeneti rekord között. Amikor sikerül megtalálni a *kezdminta* mintát, a tartomány minta *bekapcsol*. A tartomány minta illeszkedik az adott rekordra is, és addig amíg bekapcsolva marad, automatikusan illeszkedik minden beolvasott bemeneti rekordra. Ezen kívül minden rekordot megpróbál illeszteni az *végminta* mintával is; amikor ez sikerül a tartomány minta kikapcsol a következő rekordra. Ekkor ismét a *kezdminta* mintát keresi a rekordok között.

Az a rekord, ami be- illetve kikapcsolja a tartomány mintát szintén illeszkedik a tartomány mintára. Ha ezeken a rekordokon nem akarunk dolgozni, akkor egy `if` kifejezést kell használni a szabályok tevékenység részében, hogy kigyüjtsük azokat a rekordokat amik érdekelnek.

Lehetséges, hogy egy minta be- és ki is kapcsolódik ugyanazon rekord által, ha a rekord mindkét feltételt teljesíti. Ekkor a tevékenység csak az adott rekordra hajtódik végre.

Például, tegyük fel hogy két azonos jel (mondjuk a '`%`' szimbólum) közötti szöveged van, amit szeretnél elhagyni egy másik szövegből. Összekombinálhatod a tartomány mintát a `next` kifejezéssel (eddig nem tárgyaltuk, lásd 9.7. Bekezdés [A `next` kifejezés], 111. oldal), aminek hatására az `awk` átugorja a jelenlegi rekord minden további feldolgozását, és újrakezdi a következő rekorddal. Egy ilyen program így nézne ki:

```
/^%$/,/^%$/    { next }
               { print }
```

Ez a program nem működik, mivel a tartomány mintát be- és ki is kapcsolja az első sor ami a '`%`' szimbólumot tartalmazza. Ahhoz, hogy valóban működjön a program, valahogy így kellene kinéznie:

```
/^%$/     { skip = ! skip; next }
skip == 1 { next } # skip lines with 'skip' set
```

Érdemes megjegyezni, hogy a tartomány mintában a vesszőnek ('`,`') van a legalacsonyabb precedenciája és ezért utolsóként értékelődik ki az operátorok közül. Így, például, a következő program megpróbál egy tartomány mintát és egy egyszerűbb tesztet összekombinálni.

```
echo Yes | awk '/1/,/2/ || /Yes/'
```

Ennek a programnak a szerzője úgy értette, hogy '`(/1/,/2/) || /Yes/`', de az `awk` ezt úgy értelmezi, hogy '`/1/, (/2/ || /Yes/)`'. Ezt nem lehet megváltoztatni vagy valami trükkel elkerülni; tartomány minták nem kombinálhatók más mintákkal.

8.1.5. A `BEGIN` és az `END` speciális minták

A `BEGIN` és az `END` speciális minták. Ezek a minták nem illeszkednek semmilyen bemeneti rekordra, ezzel szemben kezdő és lezáró tevékenységet biztosítanak egy `awk` program számára.

8.1.5.1. Kezdő és lezáró tevékenységek

A `BEGIN` szabály csak egyszer hajtódik végre, mielőtt beolvasná az első bemeneti rekordot. Az `END` szabály is csak egyszer hajtódik végre, miután beolvasott minden bemenetet. Például:

```
$ awk '
> BEGIN { print "Analysis of \"foo\"" }
> /foo/ { ++n }
> END   { print "\"foo\" appears " n " times." }' BBS-list
 ⊣ Analysis of "foo"
 ⊣ "foo" appears 4 times.
```

Ez a program megszámolja azokat a rekordokat a 'BBS-list' file-ban, amelyek tartalmazzák a 'foo' szöveget. A BEGIN szabály kinyomtatja a report fejlécét, ugyanakkor nincs arra szükség, hogy a BEGIN szabályban az n számlálót zérus értékkel inicializáljuk, mivel az awk megteszi ezt automatikusan (lásd 7.3. Bekezdés [Változók], 79. oldal).

A második szabály megnöveli az n változót minden alkalommal, amikor a rekord tartalmazza a 'foo' mintát. Az END szabály kinyomtatja az n változó értékét a futás végén.

A BEGIN és az END speciális szabályokat nem lehet sem tartomány mintában, sem logikai operátorral használni (valójában semmilyen operátorral nem lehet kombinálni).

Egy awk program tartalmazhat több BEGIN és/vagy END szabályt is. A megjelenési sorrendben hajtódnak végre, a BEGIN szabályok kezdetben és az END szabályok a program legvégén. A BEGIN és az END szabályok összekeverhetők más szabályokkal. Ezt a lehetőséget 1987-ben adták az awk-hoz és bekerült a POSIX szabványba. Az awk eredeti (1987-es) verziója megkívánta, hogy a BEGIN szabály a program elején álljon és az END szabály a legvégén, ezenkívül csak egy BEGIN és csak egy END minta volt használható. Ez ma már nincs így, de jó ötlet ha a program olvashatóságát vagy szervezését tekintjük fő szempontnak.

A többszörös BEGIN és END szabályok hasznosak könyvtár függvények írásánál, mivel minden könyvtári file-nak lehet saját BEGIN és/vagy END szabálya, ami elvégzi a szükséges inicializálást és/vagy takarítást. Fontos figyelembe venni, hogy amilyen sorrendben a könyvtári függvények megjelennek a parancssorban az meghatározza a BEGIN és az END szabályok végrehajtási sorrendjét is. Ezért fontos, hogy óvatosan írjunk ilyen szabályokat a könyvtár file-okba, mivel így a végrehajtási sorrend nem számít. Lásd 15. Fejezet [A Library of awk Functions], 169. oldal, ami bemutat néhány hasznos könyvtári függvényt.

Ha egy awk programban csak egy BEGIN szabály van és semmilyen más szabály nincs akkor a program kilép a BEGIN szabály végrehajtása után. (Az awk eredeti verizója folyamatosan olvasott, és minden bemenetet eldobott addig, amíg egy file vége jelet nem kapott.) Ugyanakkor ha van egy END szabály a programban, akkor a bemenetet mindenképpen olvasni fogja, még akkor is, ha nincs semmilyen más szabály a programban. Ez szükséges abban az esetben ha az END szabály használná a FNR és a NR változókat (s.s.).

A BEGIN és az END szabályoknak kell legyen tevékenység része; ezeknél a szabályoknál nincs alaptevékenység.

8.1.5.2. Bemenet/kimenet a BEGIN és az END szabályokban

Van néhány (néha csak apró), de fontos kérdés az I/O-val kapcsolatban, amikor a `BEGIN` vagy az `END` szabályokon belül használjuk.

Az első kérdés, hogy mi lesz a `$0` értéke a `BEGIN` szabályon belül. Mivel a `BEGIN` szabály a bemenet beolvasása előtt hajtódik végre, ezért nincs semmilyen bemeneti rekord, és nincsennek mezők a `BEGIN` szabály végrehajtása során. Ha a `$0`-ra vagy bármilyen mezőre hivatkozunk, akkor egy üres szöveget vagy zérus értéket kapunk, az adott helyzettől függően. Az egyik lehetőség, hogy a `$0`-nak valódi értéket adjunk a `getline` parancs használatával (lásd 5.8. Bekezdés [Explicit beolvasás `getline`-al], 53. oldal). A másik lehetőség, hogy értéket rendelünk hozzá.

A második kérdés hasonló az elsőhöz, de a másik irányból közelíti meg a problémát. Az `END` szabályon belül mi a `$0` és az `NF` értéke? Hagyományosan, főleg az implementációknak köszönhetően, a `$0` és a `NF` értéke *nem definiált* az `END` szabályon belül. A POSIX szabvány azt definiálja, hogy az `NF` elérhető az `END` szabályon belül, és az utolsó bemeneti rekordban előforduló mezők számát tartalmazza. Valószínűleg csak tévedésből a szabvány nem rendelkezik a `$0` értékének megőrzéséről, de ez lenne logikus. Valójában a `gawk` így is tesz és megőrzi a `$0` értékét az `END` szabályon belül, de jegyezd meg, hogy a UNIX `awk` és valószínűleg más implementációk nem így viselkednek.

A harmadik kérdés következik az első kettőből. Mit jelent a '`print`' parancs a `BEGIN` és az `END` szabályon belül? A jelentése persze ugyanaz, '`print $0`'. Ha a `$0` egy üres szöveg, akkor egy üres sort nyomtat ki. Régen az `awk` programozók a `BEGIN` és az `END` szabályokon belül a '`print`' parancsot használták a '`print ""`' helyett, abban bízva, hogy a `$0` egy üres szöveg. Bár ez gyakran igaz a `BEGIN` szabályon belül, legalább is a `gawk`-ban, de nagyon rossz ötlet az `END` szabályon belül. Ugyanakkor rossz programozói stílus is, mivel ha üres sort akarunk kinyomtatni, akkor adjuk azt meg a programnak.

8.1.6. Az üres minta

Az üres (vagyis nem létező) minta illeszkedik *minden* bemeneti rekordra. Például, a program:

```
awk '{ print $1 }' BBS-list
```

kinyomtatja minden rekord első mezőjét.

8.2. Tevékenységek áttekintése

Egy `awk` program "script"-szabályok és függvénydefiníciók keveréke. (A függvényeket később tárgyaljuk, lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal.)

Egy szabály egy mintát és egy tevékenységet tartalmaz, bármelyik (de egyszerre mind a kettő nem) hagyható el. A *tevékenység* mondja meg az `awk`-

nak, hogy mit kell csinálni, ha megtalálta a keresett mintát. Úgy nagyjából, egy `awk` program így néz ki:

```
[minta] [{ tevékenység }]
[minta] [{ tevékenység }]
...
function név(argumentumok) { ... }
...
```

Egy tevékenység egy vagy több `awk` *kifejezésből* áll kapcsos zárójelek között (‘{’ és ‘}’). Minden kifejezés meghatároz egy dolgot. A kifejezéseket új sor vagy pontosvessző karakterek választják el egymástól.

A kapcsos zárójelekre mindig szükség van a tevékenységek körül még akkor is, ha csak egy kifejezésből áll, vagy nincs is benne kifejezés. Ugyanakkor ha a tevékenységet elhagyjuk, akkor a kapcsos zárójeleket is el lehet hagyni. Az elhagyott tevékenység megegyezik a ‘`{ print $0 }`’ kifejezéssel.

```
/foo/  { }  # foo-ra illeszkedik, nem csinál semmit
/foo/       # foo-ra illeszkedik, kinyomtatja a rekordot
```

Az `awk`-ban használható kifejezéseket alább soroljuk fel:

- Kifejezések, amelyek függvényeket hívnak vagy értéket rendelnek egy változóhoz (lásd 7. Fejezet [Kifejezések], 77. oldal). Egy ilyen típusú kifejezés végrehajtása egyszerűen kiszámolja a kifejezés értékét. Ez akkor fontos, amikor a kifejezésnek mellékhatásai vannak (lásd 7.7. Bekezdés [Értékadó kifejezések], 84. oldal).
- Vezérlésátadó kifejezések, amelyek az `awk` program futásának folyamatát határozzák meg. Az `awk` nyelv a C programozási nyelvben használható utasításokat biztosítja (`if`, `for`, `while` és `do`) és néhány speciálisat (lásd 9. Fejezet [Vezérlésátadó kifejezések a tevékenységekben], 105. oldal).
- Összetett kifejezések, amelyek egy vagy több kifejezést fognak össze kapcsos zárójelek között. Az összetett kifejezések arra valók, hogy több kifejezést összefogjanak az `if`, a `while`, a `do` vagy a `for` kifejezések testében.
- Beolvasó (bemeneti) kifejezések, a `getline` parancs (lásd 5.8. Bekezdés [Explicit beolvasás `getline`-al], 53. oldal), a `next` parancs (lásd 9.7. Bekezdés [A `next` kifejezés], 111. oldal), és a `nextfile` parancs (lásd 9.8. Bekezdés [A `nextfile` kifejezés], 112. oldal).
- Nyomtató (kimeneti) kifejezések, `print` és `printf`. Lásd 6. Fejezet [Kimenet megjelenítése], 61. oldal.
- Törlő kifejezések, elemek törlése tömbből. Lásd 11.6. Bekezdés [A `delete` kifejezés], 127. oldal.

A következő fejezet a vezérlésátadó kifejezéseket tárgyalja részletesen.

9. Vezérlésátadó kifejezések a tevékenységekben

A *vezérlésátadó kifejezések*, mint az `if`, a `while` és így tovább, az `awk` program végrehajtásának folyamatát befolyásolják. Az `awk` vezérlésátadó kifejezései a C programozási nyelv kifejezésein alapulnak.

Minden vezérlésátadó kifejezés egy speciális kulcsszóval kezdődik, mint az `if` és a `while`, azért hogy megkülönböztethetők legyenek az egyszerű kifejezésektől.

Sok vezérlésátadó kifejezés magába foglal más kifejezéseket is; például az `if` kifejezés olyan kifejezéseket is tartalmaz, amiket vagy végrehajt vagy nem. Ezek a kifejezések alkotják a vezérlésátadó kifejezés *testét*. Ha egynél több kifejezést akarunk összefogni a vezérlésátadó kifejezés testében akkor ezt egy *összetett kifejezéssel*, kapcsos zárójelek között lehet megtenni. A kifejezéseket új sorral vagy a pontos vessző karakterrel lehet elválasztani egymástól.

9.1. Az `if-else` kifejezés

Az `if-else` kifejezés az `awk` döntéshozó kifejezése és így néz ki:

`if (`*feltétel*`)` *then-test* [`else` *else-test*]

A *feltétel* kifejezés befolyásolja, hogy a kifejezés további része mit csinál. Ha a *feltétel* igaz, akkor a *then-test* hajtódik végre; egyébként a *else-test* fut le. A kifejezés `else` része opcionális. A feltétel hamis ha az értéke zérus vagy üres szöveg, egyébként igaz.

Íme egy példa:

```
if (x % 2 == 0)
    print "x páros"
else
    print "x páratlan"
```

Ebben a példában, ha a '`x % 2 == 0`' kifejezés igaz (vagyis az `x` értéke osztható kettővel), akkor az első `print` parancs hajtódik végre, más esetben a második `print` fog lefutni.

Ha az `else` a *then-test*el egy sorban jelenik meg és a *then-test* nem összetett kifejezés (pl. nincs kapcsos zárójelek között), akkor egy pontos vesszőnek kell elválasztania a *then-test*et az `else`-től. Ennek bemutatására, írjuk át az előző példát:

```
if (x % 2 == 0) print "x páros"; else
        print "x páratlan"
```

Ha elfelejtjük a '`;`' -t kitenni, akkor az `awk` nem képes értelmezni a kifejezést és szintaktikai hibát fog jelezni.

Valójában nem érdemes így írni a példát, mivel egy emberi olvasó lehet, hogy nem veszi észre az `else` kifejezést, ha az nem az első szó a sorban.

9.2. A `while` kifejezés

A programozásban a *hurok* vagy a *ciklus* azt jelenti, hogy a program egymás után kétszer vagy többször hajtja végre ugyanazt a részt.

A `while` kifejezés a legegyszerűbb hurokképző kifejezés az `awk`-ban. A `while` addig hajt végre más kifejezéseket, amíg a feltétel igaz. A kifejezés formája:

```
while (feltétel)
  test
```

A *test* tartalmazza a végrehajtandó kifejezéseket, és a *feltétel* az a kifejezés, ami szabályozza, hogy a hurok hányszor fusson le.

A `while` kifejezés először kiértékeli a *feltétel* kifejezést. Ha a *feltétel* igaz, akkor végrehajtja a *test* kifejezést. Miután a *test* végrehajtódott, a *feltétel*t újra kiértékeli, és ha még mindig igaz, akkor a *test* ismét lefut. Ezt az eljárást addig ismétli, amíg a *feltétel* hamis nem lesz. Ha a *feltétel* kezdetben hamis, akkor a hurok teste soha nem hajtódik végre, és az `awk` a hurok utáni kifejezéssel folytatja a program végrehajtását.

Ez a példa minden rekord első három mezőjét nyomtatja ki, egyet egy sorba.

```
awk '{ i = 1
       while (i <= 3) {
           print $i
           i++
       }
}' inventory-shipped
```

Itt a hurok teste egy összetett kifejezés kapcsos zárójelek között, ami két kifejezésből áll.

A hurok valahogy így működik: először az `i` értéke egy lesz. Ezután a `while` ellenőrzi, hogy az `i` kisebb-e mint három vagy egyenlő-e hárommal. Ez igaz, hiszen az `i` értéke egy, így kinyomtatja az `i`-edik mezőt. Ezek után a '`i++`' kifejezés megnöveli az `i` értékét, majd a hurok megismétli ezt a folyamatot. A hurok véget ér amikor az `i` értéke négy lesz.

Mint látható, új sor nem szükséges a feltétel és kifejezés teste között, de a kifejezés több sorba szedése jobban olvashatóvá teszi a programot. Kivéve persze ha összetett kifejezést használunk vagy nagyon egyszerű kifejezést. Az új sor a nyitó zárójel után, ami az összetett kifejezést elkezdi szintén nem szükséges, de akkor a programot nehezebb olvasni.

9.3. A `do-while` kifejezés

A `do` hurok a `while` hurok kifejezés egy változata. A `do` hurok kifejezés egyszer végrehajtja a *test*et, és ezt addig ismétli amíg a *feltétel* igaz. A formája az alábbi:

`do`
  test
`while (`*feltétel*`)`

Még ha a *feltétel* kezdetben hamis is, a *test* legalább egyszer végrehajtódik (és csak egyszer, hacsak a *test* igazzá nem teszi a *feltétel*t. Érdemes ezt összehasonlítani az alábbi `while` kifejezéssel:

`while (`*feltétel*`)`
  test

Ez a kifejezés nem hajtja végre a *test* kifejezést egyszer sem, ha a *feltétel* hamis kezdetben.

Itt egy példa a `do` kifejezésre:

```
awk '{ i = 1
       do {
          print $0
          i++
       } while (i <= 10)
}'
```

Ez a program minden rekordot tízszer nyomtat ki. Ez nem egy valós példa, mivel egy közönséges `while` hurok is megtenné. Ugyanakkor egy tapasztalatot is tükröz, hogy nagyon ritkán van valós indok a `do` kifejezés használatára.

9.4. A `for` kifejezés

A `for` kifejezés kényelmessé teszi iterációs ciklusok készítését. A `for` kifejezés általános formája:

`for (`*inicializálás*`;` *feltétel*`;` *növelés*`)`
  test

Az *inicializálás*, a *feltétel* és a *növelés* tetszőleges `awk` kifejezések és a *test* az ismételten végrehajtandó `awk` kifejezés.

A `for` kifejezés először az *inicializálás* kifejezést hajtja végre. Ezután, amíg a *feltétel* igaz, ismételten végrehajtja a *test*et majd az *növelés* kifejezést. Tipikusan az *inicializálás* a változót egyre vagy zérusra állítja, a *növelés* eggyel növeli azt és a *feltétel* ellenőrzi, hogy az iterációk száma elérte-e a kívánt értéket.

Itt egy példa a `for` kifejezésre:

```
awk '{ for (i = 1; i <= 3; i++)
          print $i
}' inventory-shipped
```

A program minden bemeneti rekord első három mezőjét kinyomtatja, egy mezőt soronként.

Egynél több változót nem lehet beállítani az *inicializálás* részben kivéve a többszörös értékadás, mint '`x = y = 0`', ami csak akkor használható ha minden változó kezdeti értéke egyenlő. (Persze a pótlólagos változóknak külön-külön is értéket adhatunk a `for` ciklus előtt.)

Ugyanez igaz a *növelés* részre is; ahhoz, hogy egynél több változót növeljünk meg, a ciklus végén kell ezt elvégezni külön kifejezésekkel. A C programozási nyelvben az összetett kifejezések kialakítására használható vessző operátor az ilyen esetekben hasznos lehet, de az `awk`-ban nem támogatott.

Leggyakrabban a *növelés* egy növelő kifejezés, mint a fenti példában. De ez nem kötelező; ez bármilyen kifejezés lehet. Például, ez a kifejezés kinyomtatja a kettes szám egy és száz közé eső hatványait:

```
for (i = 1; i <= 100; i *= 2)
  print i
```

A `for` utáni zárójelek közé eső három paraméter közül bármelyik elhagyható, ha abban a pozícióban nincs mit csinálni. Így a '`for (; x > 0;)`' kifejezés egyenértékű a '`while (x > 0)`' kifejezéssel. Ha a *feltétel* nincs megadva, akkor a feltétel mindig *true*, vagyis igaz, ami egy *végtelen ciklust* eredményez (pl. a ciklus soha nem ér véget).

A legtöbb esetben, egy `for` ciklus egy `while` hurok rövidítése, ahogy ezt alább bemutatjuk:

```
inicializálás
while (feltétel) {
  test
  növelés
}
```

Az egyetlen kivétel, amikor a `continue` kifejezés (lásd 9.6. Bekezdés [A `continue` kifejezés], 110. oldal) szerepel a ciklusban. Ebben az esetben ha a `for` kifejezést átalakítjuk `while` kifejezésre, akkor a `continue` kifejezés hatását is könnyen megváltoztathatjuk akaratlanul.

A `for` ciklusnak van egy alternatív formája, ami egy tömb elemein megy végig:

```
for (i in tömb)
    csinálj valamit az elemmel array[i]
```

A 11.5. Bekezdés [Egy tömb elemeinek ellenőrzése], 126. oldal, további információt tartalmaz a `for` ciklus ezen verziójáról.

Az `awk` nyelv a `while` hurokképző kifejezésen kívül a `for` kifejezéssel is rendelkezik, mivel egy `for` ciklus begépelése kevesebb időt vesz igénybe, és egy természetesebb gondolkodást támogat. Az iterációs lépések számolása egy természetes feladat a ciklusokban. Egyszerűbb erről úgy gondolkodni, hogy ez a számolás a ciklus része, mint hogy a cikluson belül kelljen ezt elvégezni.

A következő fejezetben bonyolultabb `for` ciklusokat mutatunk be.

9.5. A `break` kifejezés

A `break` kifejezés a legbelsőbb `for`, `while` vagy `do` hurokból lép ki. A következő példa egy egész szám legkisebb osztóját keresi meg, és azonosítja a prímszámokat is:

```
awk '# legkisebb osztó keresése
     { num = $1
       for (div = 2; div*div <= num; div++)
         if (num % div == 0)
           break
       if (num % div == 0)
         printf "A %d legkisebb osztója a %d\n", num, div
       else
         printf "%d prím\n", num
     }'
```

Ha a maradék zérus az első `if` kifejezésben, az `awk` azonnal *kilép* a `for` ciklusból. Ez azt jelenti, hogy az `awk` közvetlenül a ciklus utáni kifejezéssel folytatódik. (Ez teljesen különböző az `exit` kifejezéstől, ami az egész `awk` programból lép ki. Lásd 9.9. Bekezdés [Az `exit` kifejezés], 112. oldal.)

Itt van még egy program, ami teljesen azonos az előzővel és bemutatja, hogy hogyan lehet a `for` vagy a `while` kifejezés `feltétel` részét lecserélni egy `if` és egy `break` kifejezéssel:

```
awk '# find smallest divisor of num
     { num = $1
       for (div = 2; ; div++) {
         if (num % div == 0) {
           printf "Smallest divisor of %d is %d\n", num, div
           break
         }
         if (div*div > num) {
           printf "%d is prime\n", num
           break
         }
       }
}'
```

Mint azt már korábban elmondtuk, a `break` kifejezésnek nincs semmi jelentése egy ciklus testén kívül. Ugyanakkor, bár ez soha nem volt dokumentálva, az `awk` régebbi implementációi a `break` kifejezést egy cikluson kívül úgy kezelték mint egy `next` kifejezés (lásd 9.7. Bekezdés [A `next` kifejezés], 111. oldal). Az `awk` jelenlegi implementációi többé nem támogatják ezt a viselkedést. A `gawk` támogatja a `break` ilyen viselkedését ha a parancssorban a '`--traditional`' opció is meg van adva (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). Más esetekben hibát eredményez, mivel a POSIX szabvány a `break` kifejezést úgy definiálja, hogy csak cikluson belül használható (s.s.).

9.6. A `continue` kifejezés

A `continue` kifejezés, mint a `break` kifejezés csak a `for`, a `while` és a `do` cikluson belül használható. A `continue` kifejezés végrehajtása a ciklus további részét átugorja, aminek hatására az újabb ciklus elkezdődik. Ezzel ellentétben a `break` kilép a ciklusból.

Egy `for` ciklusbeli `continue` kifejezés arra utasítja az `awk`-ot, hogy ugorja át a ciklus további részét és a `for` kifejezés növelő kifejezés részével folytassa a futtatást. Az alábbi program ezt a tényt illusztrálja:

```
awk 'BEGIN {
     for (x = 0; x <= 20; x++) {
         if (x == 5)
             continue
         printf "%d ", x
     }
     print ""
}'
```

Ez a program kinyomtatja a számokat zérustól 20-ig, kivéve az ötös számot, amikor a `printf` kifejezést átugorja. Mivel a növelő '`x++`' kifejezést nem ugorja át, az `x` változónak nem öt lesz az értéke ezek után. A fenti `for` ciklust érdemes összehasonlítani az alábbi `while` ciklussal:

```
awk 'BEGIN {
     x = 0
     while (x <= 20) {
         if (x == 5)
             continue
         printf "%d ", x
         x++
     }
     print ""
}'
```

Ez a program végtelen ciklusba kerül miután az `x` változó öt értéket kap.

Mint azt már korábban elmagyaráztuk, a `continue` kifejezésnek nincs értelme egy cikluson kívül. Ugyanakkor, bár eddig nem volt dokumentálva, az `awk` régi implementációjában a `continue` kifejezés cikluson kívüli használata a `next` kifejezésnek felelt meg (lásd 9.7. Bekezdés [A `next` kifejezés], 111. oldal). A Unix `awk` jelenlegi implementációi nem támogatják ezt a viselkedést. A `gawk` lehetővé teszi ezt viselkedést ha a '`--traditional`' opció szerepel a parancssorban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). Minden más esetben ez hibát jelent, mivel a POSIX szabvány a `continue` ilyen viselkedését nem definiálja (s.s.).

9.7. A next kifejezés

A `next` kifejezés arra utasítja az `awk`-ot, hogy azonnal fejezze be a jelenlegi rekord feldolgozását és folytassa a következő rekorddal. A jelenlegi szabály további tevékenység részét sem hajtja végre.

Érdemes ezt összehasonlítani a `getline` függvény hatásával (lásd 5.8. Bekezdés [Explicit beolvasás `getline`-al], 53. oldal). A `getline` kifejezés hatására is az `awk` azonnal beolvassa a következő rekordot, de nem változtatja meg a program futási folyamatát semmilyen módon. Így az aktuális tevékenység az új rekorddal folytatja a feldolgozást.

A legmagasabb szinten egy `awk` program végrehajtása egy olyan ciklus, ami beolvassa a bemeneti rekordokat és minden szabállyal szemben teszteli is azokat. Ha erre a ciklusra úgy gondolsz, mint egy `for` kifejezésre, aminek a testét a szabályok alkotják, akkor a `next` kifejezés megfelel a `continue` kifejezésnek: átugorja a ciklus testének végét és a növelés kifejezést hajtja végre (ami ebben az esetben beolvassa a következő rekordot).

Például, ha az `awk` programod csak négy mezőből álló rekordokon dolgozik és nem akarod, hogy rossz bemenet esetén felmondja a szolgálatot akkor egy ilyen szabályt használhatsz a program elején:

```
NF != 4 {
  err = sprintf("%s:%d: skipped: NF != 4\n", FILENAME, FNR)
  print err > "/dev/stderr"
  next
}
```

így a következő szabályok nem kapják meg a rossz rekordot. A hibaüzenet át van irányítva a szabványos hibakimenetre, mint ahogy minden hiba esetén lennie kell. Lásd 6.7. Bekezdés [Speciális file nevek `gawk`-ban], 71. oldal.

A POSIX szabvány szerint, a `next` kifejezés viselkedése nem definiált a `BEGIN` és az `END` szabályokban. A `gawk` szintaktikai hibát fog jelezni. Bár a POSIX szabvány megengedi, de néhány `awk` implementáció nem engedi meg a `next` kifejezés használatát egy függvényben (lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal). Mint minden más `next` kifejezés, a függvény belsejében elhelyezett `next` kifejezés is beolvassa a következő rekordot és elkezdi feldolgozni a program első szabályával.

Ha a `next` kifejezés használatával eléri a bemenet végét, akkor az `END` szabályon belüli kódot hajtja végre. Lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal.

Figyelem: Néhány `awk` implementáció futási időben hibát generál, ha egy a felhasználó által definiált függvényen belül a `next` kifejezést használjuk (lásd 13. Fejezet [User-defined Functions], 153. oldal). A `gawk`-nak nincs ilyen problémája.

9.8. A `nextfile` kifejezés

A `gawk` a `next` kifejezéshez hasonlóan egy `nextfile` kifejezést is biztosít. Ugyanakkor az aktuális rekord eldobása helyett a `nextfile` kifejezés arra utasítja a `gawk`-ot, hogy fejezze be az adott adat file feldolgozását.

A `nextfile` kifejezés hatására a parancssorbeli következő file nevével tölti fel a `FILENAME` változót, az `FNR` változót egyre állítja, az `ARGIND` változót megnöveli és elkezdi a feldolgozást a program első szabályától. Lásd 10. Fejezet [Beépített változók], 115. oldal.

Ha a `nextfile` kifejezés hatására eléri a bemenet végét, akkor az `END` szabály kódja hajtódik végre. Lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal.

A `nextfile` kifejezés egy `gawk` kiegészítés; (jelenleg) nem használható más `awk` implementációkban. Lásd 15.2. Bekezdés [Implementing `nextfile` as a Function], 170. oldal, ahol bemutatjuk, hogy a felhasználó által definiált függvénnyel is szimulálható a `nextfile` kifejezés.

A `nextfile` kifejezés hasznos lehet ha több adatfile-t kell feldolgozni, és nem akarod minden file minden rekordját feldolgozni. Normális esetben ahhoz, hogy a következő file feldogozását elkezdhesd, a szükségtelen rekordokat is át kellene nézni. A `nextfile` kifejezés ezt kerüli el hatékonyan.

Figyelem: A `gawk` 3.0-ás verziója előtt két azonos szó állt rendelkezésre ugyanarra a kifejezésre, a '`next file`' és a `nextfile`. Ez megváltozott a 3.0-ás verziótól kezdve, mivel a `file` kulcsszó kezelése így nem volt következetes. Ha a `next` után szerepelt akkor kulcsszó volt, egyébként meg egy egyszerű azonosító. A régi használat még mindig elfogadott. Ugyanakkor a `gawk` figyelmeztető üzenetet generál és a `gawk` jövőbeli verzióiban a `next file` kifejezés nem támogatott.

9.9. Az `exit` kifejezés

Az `exit` kifejezés hatására az `awk` azonnal befejezi az aktuális szabály végrehajtását, és abbahagyja a bemenet feldolgozását is; minden további bemenetet figyelmen kívül hagy. A formája az alábbi:

`exit` [*visszatérési érték*]

Ha egy `exit` kifejezést hajt végre az `awk` a `BEGIN` szabályban akkor azonnal befejez minden működést. Semmilyen bemenetet nem olvas be. Ugyanakkor ha egy `END` szabály is van a programban, akkor azt is végrehajtja (lásd 8.1.5. Bekezdés [A `BEGIN` és az `END` speciális minták], 100. oldal).

Ha egy `exit` kifejezés hajtódik végre az `END` szabályban, akkor a program azonnal leáll.

Egy `exit` kifejezés ami nem része sem egy `BEGIN` sem egy `END` szabálynak azonnal befejezi a további szabályok végrehajtását az adott rekordra, átugorja a további bemeneti rekordokat és végrehajtja az `END` szabályt, ha van ilyen.

Ha azt akarod, hogy az `END` szabály ne fusson le ebben az esetben, akkor egy változót be kell állítani egy nem zérus értékre az `exit` kifejezés előtt, és ezt a változót ellenőrizni kell az `END` szabályon belül. Lásd 15.3. Bekezdés [Assertions], 172. oldal, ami egy példát tartalmaz erre.

Ha egy argumentumot is megadunk az `exit` kifejezésnek, akkor ez az érték lesz az `awk` kimeneti státusz kódja, visszatérési értéke. Ha nincs argumentum megadva, akkor az `exit` státusza zérus lesz (siker). Abban az esetben amikor először az `exit` kifejezésnek egy argumentumot adunk meg, majd a második esetben nem argumentummal hívjuk meg, akkor az előző kimeneti értéket fogja használni (s.s.).

Például, tegyük fel, hogy egy olyan esetet fedezel fel, amit nem tudsz hogyan kellene kezelni. Hagyományosan a program ezt úgy jelenti neked, hogy nem zérus státusszal lép ki. Az `awk` programod is megteheti ezt, ha az `exit` kifejezést egy nem zérus argumentummal használod. Íme egy példa erre:

```
BEGIN {
       if (("date" | getline date_now) <= 0) {
         print "Can't get system date" > "/dev/stderr"
         exit 1
       }
       print "current date is", date_now
       close("date")
}
```

10. Beépített változók

A legtöbb `awk` változó saját használatra mindig elérhető; az értékük soha nem változik meg, kivétel ha a program értéket rendel hozzá, és soha nem befolyásol semmit, amikor az értékét lekérdézzük. Ugyanakkor néhány `awk` változónak speciális előre definiált jelentése van. Ezek közül néhányat az `awk` automatikusan többször megvizsgál, így ezek segítségével az `awk` működését befolyásolni lehet. Másokat az `awk` automatikusan beállít, így információt továbbít az `awk` belső struktúrájából a programod felé.

Ez a fejezet dokumentálja az `gawk` összes beépített változóját. A legtöbbjük dokumentálva van még a hatásukat leíró fejezetekben is.

10.1. Az `awk`-ot befolyásoló beépített változók

Az alábbiakban azoknak a változóknak az ábécé sorrendbe szedett listáját közöljük, amelyek segítségével az `awk` működése befolyásolható. Azok a változók, amelyek csak a `gawk`-ban találhatók meg egy csillaggal '`*`' vannak megjelölve.

`CONVFMT`
: Ez a szöveg befolyásolja a számok konverzióját szöveggé (lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal). A változó értéke a `sprintf` függvény első argumentuma lesz (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Az alapértéke `"%.6g"`. A `CONVFMT` változót a POSIX szabvány vezette be.

`FIELDWIDTHS *`
: Egy szóközökkel elválasztott lista, ami megmondja a `gawk`-nak, hogy a bemenetet hogyan kell feldarabolni fix szélességű oszlopokra. Ez csak egy kísérleti megoldás. Ha a `FIELDWIDTHS`-nek értéket adunk akkor az `FS` nem érvényes többé a mezők darabolásánál. Lásd 5.6. Bekezdés [Meghatározott szélességű adatok beolvasása], 49. oldal, további információért.

  Ha a `gawk` "compatibility" módban van (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor a `FIELDWIDTHS` változónak nincs speciális jelentése, és a mezőkre darabolás kizárólag az `FS` változó alapján történik.

`FS`
: Az `FS` változó adja meg a bemeneti mezőelválasztót (lásd 5.5. Bekezdés [Hogyan történik a mezőelválasztás], 44. oldal). A változó értéke egy egy-karakteres szöveg vagy több karakterből álló reguláris kifejezés ami illeszkedik a mezőket elválasztó szövegre. Ha az értéke egy üres szöveg (`""`) akkor a rekordon belül minden karakter egy különálló mező lesz.

  A változó alapértéke a `" "`, egy szóközből álló szöveg. Ennek a speciális jelentésa az, hogy szóközök, tab és/vagy új sor karakterek sorozata alkotja a mezőelválasztót.[1] Egy másik speciális tulajdonság, hogy ebben az esetben a kezdő és végső szóközök, tab és új sor karaktereket nem veszi figyelembe.

Az `FS` változó értékét a parancssorból is be lehet állítani a '`-F`' opcióval:

`awk -F, '`*program*`'` *input-files*

Ha a `gawk` a `FIELDWIDTHS` változót használja a mezők feldarabolására, akkor ha egy értéket rendelünk az `FS` változóhoz a `gawk` visszakapcsol az alap elválasztó módba. Ennek egy egyszerű módja az '`FS = FS`' kifejezés.

`IGNORECASE *`

Ha az `IGNORECASE` nem zérus vagy nem üres szöveg, akkor minden szöveg összehasonlításnál és reguláris kifejezés illesztésnél nem tesz különbséget a kis- és nagybetűk között. Így a reguláris kifejezések illesztése a '`~`' és a '`!~`' operátorokkal és a `gensub` a `gsub`, az `index`, a `match`, a `split` és a `sub` függvények, az `RS` rekordelválasztó és az `FS` mezőelválasztó nem tesz különbséget a kis- és a nagybetűk között. Az `IGNORECASE` változó értéke *nem* befolyásolja a tömbök elemeire való hivatkozásnál használt szövegeket. Lásd 4.5. Bekezdés [Kis- és nagybetűk az illesztésekben], 33. oldal.

Ha a `gawk` "compatibility" módban van (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor az `IGNORECASE` változónak nincs speciális jelentése és a szöveg és a regexp műveletekben megkülönbözteti a kis- és nagybetűket.

`OFMT` Ez a változó befolyásolja a számok szöveggé konvertálását (lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal) amit a `print` kifejezés használ. Lényegében mintha a `sprintf` függvénynek az első argumentumát adnánk meg (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Alapesetben az értéke: `"%.6g"`. Az `awk` korábbi verzióiban minden általános kifejezés esetén amikor egy számot szöveggé kellett konvertálni az `awk` a `OFMT` változót használta; ma már a `CONVFMT` változót használja erre.

`OFS` Ez a kimeneti mezőelválasztót határozza meg (lásd 6.3. Bekezdés [Kimeneti elválasztó], 63. oldal). Amikor egy `print` kifejezés mezőket nyomtat ki, akkor a változó értékét nyomtatja ki a mezők közé. Az alapérték `" "`, egy egyetlen szóközből álló szöveg.

`ORS` Ez a kimeneti rekordelválasztó. A `print` kifejezés által kinyomtatott rekordok között nyomtatja ki a változó értékét. Az

[1] POSIX `awk`-ban az új sor karakter nem mezőelválasztó.

alapértéke: `"\n"`. (Lásd 6.3. Bekezdés [Kimeneti elválasztó], 63. oldal.)

`RS` Ez a változó az `awk` bemeneti rekordelválasztója. Az alapértéke egy olyan szöveg, ami csak egy új sor karaktert tartalmaz. Ez azt jelenti, hogy a bemeneti rekord egyetlen sorból áll. Az értéke lehet üres szöveg is, így a rekordokat vagy üres sorok vagy regexp kifejezések illeszkedései választják el egymástól. (Lásd 5.1. Bekezdés [Hogyan történik a feldarabolás rekordokra], 37. oldal.)

`SUBSEP` A `SUBSEP` változó értéke választja el egy több dimenziós tömb indexeit és az alapértéke `"\034"`. Így ez a kifejezés `foo["A", "B"]` valójában `foo["A\034B"]` kifejezéssel egyezik meg (lásd 11.9. Bekezdés [Többdimenziós tömbök], 130. oldal).

10.2. Információt hordozó beépített változók

Az alábbiakban azoknak a változóknak az ábécé sorrendbe szedett listáját közöljük, amelyeket az `awk` automatikusan állít be, és így információt biztosít a programod számára. Azok a változók, amelyek csak a `gawk`-ban használhatók egy csillaggal '`*`' vannak megjelölve.

`ARGC`
`ARGV` Az `awk` program számára elérhető parancssori argumentumokat az `ARGV` tömb tárolja. Az `ARGC` változó a parancssori argumentumok számát tárolja. Lásd 14.2. Bekezdés [Other Command Line Arguments], 165. oldal. Más `awk` tömböktől eltérően az `ARGV` elemeinek az indexe zérustól (`ARGC` − 1) -ig terjed. Például:

```
$ awk 'BEGIN {
>        for (i = 0; i < ARGC; i++)
>            print ARGV[i]
>      }' inventory-shipped BBS-list
⊣ awk
⊣ inventory-shipped
⊣ BBS-list
```

Ebben a példában, az `ARGV[0]` értéke `"awk"`, az `ARGV[1]` értéke `"inventory-shipped"` és az `ARGV[2]` tartalma a `"BBS-list"` szöveg. Az `ARGC` értéke három, eggyel több mint az `ARGV` tömb utolsó elemének indexe, mivel a tömb elemeit zérustól számozza.

Az `ARGC` és az `ARGV` nevek, a zérustól (`ARGC` − 1)-ig indexelés a C programozási nyelvből származnak Lásd 10.3. Bekezdés [Az `ARGC` és az `ARGV` változók használata], 120. oldal, ami további információt ad arról, hogy az `awk` hogyan használja ezeket a változókat.

`ARGIND *` A feldolgozás alatt álló jelenlegi file indexét adja meg az `ARGV` tömbben. Minden alkalommal, amikor a `gawk` megnyit egy file-t feldolgozásra, beállítja az `ARGIND` változót az `ARGV` tömb elemének indexére. Amikor a `gawk` egy bemeneti file-t dolgoz fel a '`FILENAME == ARGV[ARGIND]`' kifejezés mindig igaz.

Ez a változó a file-ok feldolgozásában hasznos, mivel ez a változó mondja meg, hogy hol tartunk a file-ok feldolgozásában, és így meg lehet különböztetni a parancssorban megadott azonos file neveket.

Bár az `ARGIND` változó beállítható az `awk` programból, a `gawk` ezt a változót automatikusan beállítja az új értékre amikor elkezd feldolgozni egy új file-t.

Ez a változó egy `gawk` kiegészítés. Más `awk` implementációkban vagy ha a `gawk` "compatibility" módban van (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor a változónak nincs speciális jelentése.

`ENVIRON` Ez egy asszociatív tömb, ami a környezet értékeit tartalmazza. A tömb elemei a környezeti változók neveit tartalmazzák; az értékük az adott környezeti változó értékének felel meg. Például a `ENVIRON["HOME"]` értéke a `/home/arnold` lehet. Ha a tömb elemeit megváltoztatjuk, az nem befolyásolja azt a környezetet, amit az `awk` átad egy általa átirányítással vagy a `system` függvénnyel indított programnak. (A `gawk` jövőbeli verzióiban ez lehet, hogy megváltozik.)

Néhány operációs rendszernek nincs környezeti változója. Ezeken a rendszereken a `ENVIRON` változó üres (kivéve a `ENVIRON["AWKPATH"]` értékét).

`ERRNO *` Ha a `getline`-nál alkalmazott átirányítás, vagy a `getline` olvasás során, vagy egy `close` művelet alatt rendszer hiba lép fel, akkor az `ERRNO` változó egy a hibát leíró szöveget fog tartalmazni.

Ez a változó egy `gawk` kiegészítés. Más `awk` implementációkban vagy ha a `gawk` "compatibility" módban van (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor a változónak nincs speciális jelentése.

`FILENAME` Ez a változó tartalmazza az `awk` által éppen feldolgozás alatt álló file nevét. Amikor a parancssorban nincs file név megadva, és az `awk` a szabványos bemenetről olvas be, a `FILENAME` értéke `"-"`. A `FILENAME` értéke megváltozik minden alkalommal, amikor egy új file-t kezd el feldolgozni (lásd 5. Fejezet [Bemeneti file-ok olvasása], 37. oldal). A `BEGIN` szabályon belül a `FILENAME` értéke `""`, mivel ebben a szabályban még nem dolgoz fel file-okat.[2] (s.s.)

`FNR`
: Az aktuális file-ban az aktuális rekord számát tartalmazza az `FNR` változó. Minden alkalommal, amikor az `awk` beolvas egy új rekordot a változó értéke megnövelődik (lásd 5.8. Bekezdés [Explicit beolvasás `getline`-al], 53. oldal). Ha egy új file-t kezd el feldolgozni, akkor a változó értékét zérusra állítja.

`NF`
: Az `NF` változó az aktuális bemeneti rekordban található mezők számát tartalmazza. Az `NF` változó értékét az `awk` mindig beállítja, amikor egy új rekordot olvas be, vagy egy új mezőt hozunk létre,, vagy a `$0` megváltozik (lásd 5.2. Bekezdés [Mezők elérése], 40. oldal).

`NR`
: Az `awk` által a program futásának kezdetétől eddig feldolgozott bemeneti rekordok számát tartalmazza (lásd 5.1. Bekezdés [Hogyan történik a feldarabolás rekordokra], 37. oldal). Minden rekord beolvasása során az `awk` beállítja az `NR` értékét.

`RLENGTH`
: A `match` függvény által illesztett részszöveg hosszát tartalmazza az `RLENGTH` változó (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Az `RLENGTH` változót a `match` függvény hívása állítja be. Az értéke az illeszkedő szöveg hossza vagy -1, ha nem volt illeszkedés.

`RSTART`
: A `match` függvény által illesztett részszöveg kezdeti pozícióját adja meg az `RSTART` változó (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Az `RSTART` változót a `match` függvény hívása állítja be. Az értéke az illeszkedő szöveg kezdeti pozíciója vagy zérus, ha nem volt illeszkedés.

`RT *`
: Minden rekord beolvasása során az `awk` beállítja az `RT` változó értékét. A változó értéke a rekordelválasztóra, `RS`, illeszkedő bemeneti szöveget tartalmazza.

  Ez egy `gawk` kiegészítés. Más `awk` implementációkban vagy ha a `gawk` "compatibility" módban van (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), a változónak nincs speciális jelentése.

Egy megjegyzés az `NR` és az `FNR` változókról. Az `awk` egyszerűen megnöveli ezeket a változókat, amikor egy új rekordot olvas be, ahelyett, hogy a beolvasott rekordok számának abszolút értékét tárolná a változókban. Ez azt jelenti, hogy ha a programod megváltoztatja valamelyik változó értékét, akkor ezt az új értéket fogja megnövelni, amikor egy új rekordot olvas be (s.s.). Például:

---

[2] A Unix `awk` kezdeti implementációi a `FILENAME` értékét kezdetben `"-"` szövegre állították. Ez a viselkedés hibás, így nem szabad abban bízni, hogy ez lesz a változó kezdeti értéke.

```
$ echo '1
> 2
> 3
> 4' | awk 'NR == 2 { NR = 17 }
> { print NR }'
⊣ 1
⊣ 17
⊣ 18
⊣ 19
```

Mielőtt az `FNR` változó bekerült az `awk` nyelvbe (lásd 17.1. Bekezdés [Major Changes between V7 and SVR3.1], 251. oldal), sok `awk` program használta ezt a lehetőséget arra, hogy számon tartsa az egy file-ból beolvasott rekordok számát; amikor a `FILENAME` változó értéke megváltozott az `NR` értékét zérusra állították.

10.3. Az ARGC és az ARGV változók használata

A 10.2. Bekezdés [Információt hordozó beépített változók], 117. oldal, alatt ezt a programot láthattad, ami bemutatja, hogyan használható az `ARGC` és az `ARGV` változókban tárolt információ:

```
$ awk 'BEGIN {
>        for (i = 0; i < ARGC; i++)
>            print ARGV[i]
>      }' inventory-shipped BBS-list
⊣ awk
⊣ inventory-shipped
⊣ BBS-list
```

Ebben a példában az `ARGV[0]` értéke `"awk"`, az `ARGV[1]` értéke `"inventory-shipped"` és az `ARGV[2]` értéke `"BBS-list"`.

Érdemes megfigyelni, hogy maga az `awk` program nem került bele az `ARGV` tömbbe. Más speciális parancssori opciók, az argumentumaikkal együtt, szintén nem kerülnek be a tömbbe. Ilyenek például az értékadó utasítások a '`-v`' opcióval (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). Normális parancssori értékadó utasítások az argumentum *részei* és megjelennek az `ARGV` tömbben.

```
$ cat showargs.awk
⊣ BEGIN {
⊣     printf "A=%d, B=%d\n", A, B
⊣     for (i = 0; i < ARGC; i++)
⊣         printf "\tARGV[%d] = %s\n", i, ARGV[i]
⊣ }
⊣ END   { printf "A=%d, B=%d\n", A, B }
$ awk -v A=1 -f showargs.awk B=2 /dev/null
```

```
⊣ A=1, B=0
⊣  ARGV[0] = awk
⊣  ARGV[1] = B=2
⊣  ARGV[2] = /dev/null
⊣ A=1, B=2
```

A programod megváltoztathatja az `ARGC` változó értékét és az `ARGV` tömb elemeit. Minden alkalommal, amikor az `awk` eléri egy file végét, a következő file nevét az `ARGV` tömb következő eleméből veszi. Ha ezt az értéket a programod megváltoztatja, akkor a programod az általad megadott file-t fogja feldolgozni. A `"-"` szöveg használata esetén a szabványos bemenetről fog olvasni. Extra elemek tárolásával és az `ARGC` megnövelésével további file-ok feldolgozására lehet utasítani az `awk`-ot.

Ha az `ARGC` értékét csökkentjük, akkor bemeneti file-t távolíthatunk el a parancssori listából. Ha az `ARGC` régi értékét elmentjük egy másik változóba, akkor az eldobott argumentumokat a program felhasználhatja.

Ahhoz, hogy a lista közepéről távolítsunk el egy file nevet, az `ARGV` adott elemét üres szöveggel (`""`) kell feltölteni a file neve helyett. Ez egy speciális lehetőség, mivel az `awk` nem veszi figyelembe az üres szöveggel megadott file neveket. Másik lehetőség a `delete` kifejezés használata, ami eltávolít egy elemet az `ARGV` tömbből (lásd 11.6. Bekezdés [A `delete` kifejezés], 127. oldal).

Tipikusan ezt a feldolgozást a `BEGIN` szabályon belül érdemes elvégezni mielőtt a bemeneti file-ok feldolgozása megkezdődik. Lásd 16.1.4. Bekezdés [Splitting a Large File Into Pieces], 217. oldal, és lásd még 16.1.5. Bekezdés [Duplicating Output Into Multiple Files], 218. oldal, ami mindkét módszert bemutatja az `ARGV` egy elemének eltávolítására.

Az alábbi programrészlet az `ARGV` tömböt dolgozza fel és eltávolítja a parancssori opciókat.

```
BEGIN {
    for (i = 1; i < ARGC; i++) {
        if (ARGV[i] == "-v")
            verbose = 1
        else if (ARGV[i] == "-d")
            debug = 1
        else if (ARGV[i] ~ /^-?/) {
            e = sprintf("%s: unrecognized option -- %c",
                    ARGV[0], substr(ARGV[i], 1, ,1))
            print e > "/dev/stderr"
        } else
            break
        delete ARGV[i]
    }
}
```

Ahhoz, hogy tényleg csak az opciókat vegye figyelembe, az `awk` opciókat egy '`--`' kifejezéssel kell lezárni és azután kell megadni az opcióidat, például így:

```
awk -f myprog -- -v -d file1 file2 ...
```

Erre nincs szükség a `gawk` esetén: Hacsak a '`--posix`' opció nem lett megadva, a `gawk` minden, fel nem ismert opciót az `ARGV` tömbben helyez el, hogy az `awk` program dolgozza fel őket.

Amint a `gawk` nem ismer fel egy opciót, azonnal befejezi az opciók feldolgozását. A fenti példa így nézne ki `gawk`-ban:

```
gawk -f myprog -d -v file1 file2 ...
```

Mivel a '`-d`' nem egy érvényes `gawk` opció ezért a további '`-v`' opció átadódik az `awk` programnak.

11. Tömbök az awk-ban

Egy *tömb* az értékek, azaz *elemek*, táblája. A tömbök elemeit az indexek különböztetik meg. Az *indexek* lehetnek számok vagy szövegek. Mivel az `awk` egyetlen halmazt tart fenn a változók, tömbök és függvények neveire (lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal), ezért nem használható ugyanaz a név változóként és tömbként ugyanabban az `awk` programban.

11.1. Bevezetés a tömbökbe

Az `awk` nyelv egydimenziós *tömböket* biztosít összetartozó szövegek és számok csoportban tárolására.

Minden `awk` tömbnek kell legyen neve. A tömbök nevének ugyanaz a formája, mint a változók neveinek; bármilyen érvényes változónév egy érvényes tömbnév lehet. De nem használhatod ugyanazt a nevet változónak és tömbnek is ugyanabban az `awk` programban.

Az `awk` tömbök felületesen hasonlítanak más programozási nyelvek tömbjeire; de alapvető különbségek vannak. Az `awk`-ban nem kell megadni a tömb méretét mielőtt használnánk. Ráadásul bármilyen szöveg vagy szám használható indexként, nem csak egymás utáni egész számok.

Más programozási nyelvekben egy tömböt *definiálni* kell, és meg kell adni, hogy hány elemet fog tartalmazni. Ezekben a nyelvekben, a definiálás egy folyamatos memóriatömböt foglal le az elemek számára. A tömb indexei pozitív, egész számok kell legyenek; például, az első elemre a zérus indexel lehet hivatkozni, ami a memóriablokk legelején helyezkedik el. A második elemre az egyes indexel lehet hivatkozni, ami közvetlenül az első mellett helyezkedik el, és így tovább. Lehetetlen több elemet adni a tömbhöz, mivel csak az adott méretű hely lett lefoglalva. (Néhány programozási nyelv megenged tetszőleges kezdő vagy vég indexet, pl. '`15 .. 27`', de a tömb mérete csak adott méretű lehet.)

Koncepcionálisan egy négy elemű tömb így néz ki, ha az elemek a 8, a `"foo"`, a `""` és a 30:

`8`	`"foo"`	`""`	`30`	érték
0	1	2	3	index

Csak az elemek tárolódnak; az indexek implicit módon következnek az értékekből. A 8 a zérus indexű elem értéke, mivel a 8 a zérus pozíciójú elemben található.

Az `awk` tömbjei *asszociatívak*. Ez azt jelenti, hogy a tömb index és érték párok gyűjteménye:

Elem `4`	Érték `30`
Elem `2`	Érték `"foo"`
Elem `1`	Érték `8`

Elem `3` Érték `""`

A párokat kevert sorrendben közöltük mivel a sorrend nem számít.

Az asszociatív tömbök egy nagy előnye, hogy új elemeket bármikor hozzá lehet adni. Például tegyük fel, hogy a fenti tömbhöz egy tizedik elemet kell hozzáadni aminek az értéke `"number ten"`. Az eredmény:

Elem `10` Érték `"number ten"`
Elem `4` Érték `30`
Elem `2` Érték `"foo"`
Elem `1` Érték `8`
Elem `3` Érték `""`

most a tömb *szórt*, "sparse", ami azt jelenti, hogy bizonyos indexű elemek hiányoznak: a tömbnek megvannak az 1–4 -ig terjedő elemei és a 10. eleme, de nincs 5, 6, 7, 8 vagy 9 -es eleme.

Az asszociatív tömbök egy másik következménye, hogy az indexeknek nem kell pozitív egész számoknak lenniük. Bármilyen szám vagy szöveg indexként használható. Például itt közlünk egy tömböt, ami lefordítja az angol szavakat franciára:

Elem `"dog"` Érték `"chien"`
Elem `"cat"` Érték `"chat"`
Elem `"one"` Érték `"un"`
Elem `1` Érték `"un"`

Itt az egynek nem csak a szöveg formája hanem a szám formája is szerepel a szótárban – így illusztrálható, hogy egy tömbben szám és szöveg index is lehet. (Valójában a tömb indexe mindig szöveg; ezt részletesen tárgyaljuk a 11.7. Bekezdés [Számok használata mint tömb index], 128. oldal alatt.)

Az `IGNORECASE` értéke nincs hatással a tömb indexére. Pontosan ugyanazt a szöveget kell használni a tömb eléréséhez, mint amit az érték tárolásánál használtunk.

Amikor az `awk` hoz létre egy tömböt, pl. a `split` beépített függvénnyel, akkor az indexek egymás utáni egész számok lesznek egytől kezdve. (Lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal.)

11.2. Egy tömb elemeinek elérése

Egy tömb alapvető használati módja az elemeinek az elérése. Az elemek elérhetőek az alábbi formában:

array`[`*index*`]`

Itt az *array* a tömb neve. Az *index* kifejezés a tömb elemének eléréséhez szükséges indexet adja meg.

Egy tömb elemére való hivatkozás értéke a tömb elemének aktuális értéke. Például a `foo[4.3]` kifejezés egy hivatkozás a `foo` tömb '`4.3`' indexű elemére.

Ha egy olyan elemre hivatkozunk, aminek nincs "rögzített" értéke, akkor az értéke az üres szöveg, `""`, vagyis a változónak nem adtunk korábban értéket, vagy az elem törölve lett (lásd 11.6. Bekezdés [A `delete` kifejezés], 127. oldal). Egy ilyen hivatkozás automatikusan létrehozza az elemet egy üres szöveg értékkel. (Néhány esetben ez nem túl szerencsés, mivel ezzel `awk` memóriát pazarlunk el.)

Könnyen kideríthető, hogy egy elem az adott indexszel a tömb eleme-e:

index `in` *array*

Ez a kifejezés ellenőrzi, hogy az adott index létezik-e a tömbben, és a kifejezésnek nincs mellékhatása, nem hozza létre az elemet. A kifejezésnek egy (igaz) az értéke, ha az *array*`[`*index*`]` kifejezés létezik, és zérus (hamis), ha nem létezik.

Például ha ellenőrizni akarjuk, hogy a `frequencies` tömbnek van-e '2'-es indexű eleme, akkor az alábbi kifejezést használhatjuk:

```
if (2 in frequencies)
    print "Subscript 2 is present."
```

Figyelem, ez a kifejezés *nem* azt ellenőrzi, hogy a `frequencies` tömbnek van-e olyan eleme aminek az *értéke* kettő. (Ezt csak egyféle képpen lehet kideríteni, ha az összes elemet átnézzük.) Ugyanakkor a kifejezés *nem* hozza létre a `frequencies[2]` elemet, míg a következő (helytelen) kifejezés létrehozza az elemet:

```
if (frequencies[2] != "")
    print "Subscript 2 is present."
```

11.3. Tömb elemek feltöltése

Egy tömb elemei 'lvalue' kifejezések: így értéket adhatunk nekik, mint az `awk` változóknak:

array`[`*subscript*`] =` *value*

Itt az *array* a tömb neve. A *subscript* kifejezés a tömb elemének az indexe. A *value* kifejezés az az érték ami a tömb elemének az új értéke lesz.

11.4. Alapvető példák tömbökkel

Az alábbi program egy olyan szöveget olvas be, amiben minden sor egy számmal kezdődik, majd a számok emelkedő sorrendjében kinyomtatja a sorokat. Kezdetben a számok nincsennek sorrendben. A program sorbarendezi a sorokat. A sorszámokat tömbindexként használja. Ezután kinyomtatja a sorokat a rendezett sorrendben. Ez egy nagyon egyszerű program, hibásan működik, ha egy szám kétszer szerepel, nincs megadva az összes index, vagy egy sor nem számmal kezdődik.

```
{
  if ($1 > max)
    max = $1
  arr[$1] = $0
```

```
}

END {
  for (x = 1; x <= max; x++)
    print arr[x]
}
```

Az első szabály a legnagyobb számot tárolja, illetve minden sort eltárol az `arr` tömbben a számmal megadott indexű elemben.

A második szabály a bemenet beolvasása után fut le, és kinyomtatja az összes sort.

Ha a programot az alábbi bemenettel futtatjuk:

```
5  I am the Five man
2  Who are you?  The new number two!
4  . . . And four on the floor
1  Who is number one?
3  I three you.
```

a kimenet az alábbi lesz:

```
1  Who is number one?
2  Who are you?  The new number two!
3  I three you.
4  . . . And four on the floor
5  I am the Five man
```

Ha egy szám többször szerepel, akkor az adott számmal kezdődő utolsó sor lesz eltárolva a tömbben.

Az a probléma, amikor nincs megadva minden index, könnyen elkerülhető az `END` szabályban:

```
END {
  for (x = 1; x <= max; x++)
    if (x in arr)
      print arr[x]
}
```

11.5. Egy tömb elemeinek ellenőrzése

Az olyan programokban, amelyekben tömböket használunk, gyakran van szükség olyan ciklusra, ami a tömb minden elemén végigmegy. Más programozási nyelvekben, ahol a tömbök folyamatosak, és az indexek pozítiv egész számok ez nagyon könnyű: az érvényes indexeket megkaphatjuk, ha elszámolunk a legalacsonyabb számtól kezdve a legmagasabbig. Ez a módszer nem működik az `awk`-ban, mivel bármilyen szöveg vagy szám lehet index. Így a `for` kifejezésnek van egy speciális alakja, amivel egy tömb összes eleme ellenőrizhető:

`for (`*var* `in` *array*`)`

body

Ez a ciklus végrehajtja a *body* kifejezést az *array* tömb minden elemére egyszer, és a *var* változó megkapja a korábban használt indexeket.

Alább közlünk egy programot, ami a `for` kifejezésnek ezt a formáját használja. Az első szabály végignézi a bemeneti rekordokat, és minden legalább egyszer előforduló szót eltárol a `used` tömbben, ahol az adott szó szolgál indexként. A második szabály végigmegy a `used` tömb összes elemén, és minden 10 karakternél hosszabb szót kinyomtat, majd kinyomtatja az ilyen szavak számát. A 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal, további információt add a `length` függvényről.

```
# Minden használt szó esetén legyen az elem értéke egy.
{
    for (i = 1; i <= NF; i++)
        used[$i] = 1
}

# 10 karakternél hosszabb szavakat keresünk.
END {
    for (x in used)
        if (length(x) > 10) {
            ++num_long_words
            print x
        }
    print num_long_words, "words longer than 10 characters"
}
```

Lásd még a 16.2.5. Bekezdés [Generating Word Usage Counts], 235. oldal, ahol hasonló, részletesebb példák találhatók.

Egy tömb elemeinek elérési sorrendje az `awk` belső struktúrájától függ, és nem lehet szabályozni vagy megváltoztatni. Ez problémákhoz vezet ha az `array` tömbhöz elemeket adunk a cikluson belül; nem lehet megjósólni, hogy a `for` ciklus eléri-e az új elemeket vagy sem. Ehhez hasonló problémák lehetnek, ha a *var* változót a `for` cikluson belül megváltoztatjuk. A legjobb az ilyen változtatásokat elkerülni.

11.6. A `delete` kifejezés

Egy tömb elemet el lehet távolítani, törölni lehet a `delete` kifejezéssel:

`delete` *array*`[`*index*`]`

Ha egyszer egy elem törölve lett, az értéke többé nem érhető el. Lényegében olyan, mintha soha nem hivatkoztunk volna rá, és soha nem adtunk neki értéket.

Itt egy példa a törlésre:

```
for (i in frequencies)
```

```
delete frequencies[i]
```

Ez a példa kitörli az összes elemet a `frequencies` tömbből.

Ha törlünk egy elemet, akkor sem egy `for` ciklus során nem lehet az elemet elérni, és az `in` operátor, ami ellenőrzi, hogy az elem létezik-e a tömbben, szintén zérussal tér vissza (pl. hamis értékkel):

```
delete foo[4]
if (4 in foo)
    print "Soha nem nyomtatja ezt ki"
```

Fontos megjegyezni, hogy egy elem törlése *nem* ugyanaz, mintha egy üres szöveg, `""`, értéket rendelünk az elemhez.

```
foo[4] = ""
if (4 in foo)
  print "Ezt kinyomtatja, még ha a foo[4] üres is"
```

Egy nem létező elem törlése nem jelent hibát.

Egy tömb minden eleme törölhető egy kifejezéssel, ha elhagyjuk az indexet a `delete` kifejezésben.

`delete` *array*

Ez a lehetőség egy `gawk` kiegészítés; nem érhető el "compatibility" módban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

Ha a `delete` kifejezést így használjuk, az kb. háromszor olyan hatékony, mintha az elemeket egyenként törölnénk egy ciklussal.

Az alábbi kifejezés egy hordozható, de nem magától értetődő megoldást mutat egy tömb törlésére.

```
# köszönet Michael Brennan -nak ezért a példáért
split("", array)
```

A `split` függvény (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal) törli a céltömböt, mivel egy üres szöveget darabol fel. Mivel nincs mit felvágni (nincs adat), ezért egyszerűen törli a tömböt, és visszatér.

Figyelem: Egy elem törlése nem változtatja meg a típusát, így a törlés után nem használható mint skaláris szám. Például ez nem működik:

```
a[1] = 3; delete a; a = 3
```

11.7. Számok használata mint tömb index

Tömbök esetén fontos arra emlékezni, hogy a *tömbök indexei mindig szövegek*. Ha egy számot használunk indexként, át lesz konvertálva szöveggé mielőtt mint index lesz használva (lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal).

Ez azt jelenti, hogy a `CONVFMT` beépített változó befolyásolja azt, hogy a tömbök elemeit hogyan lehet elérni. Például:

```
xyz = 12.153
```

```
data[xyz] = 1
CONVFMT = "%2.2f"
if (xyz in data)
    printf "%s is in data\n", xyz
else
    printf "%s is not in data\n", xyz
```

Ez kinyomtatja: '`12.15 is not in data`'. Az első kifejezés numerikus értéket ad az `xyz` változónak. Ezután a `data[xyz]` kifejezésben lényegében az index a `"12.153"` szöveg lesz (az alap konverziót a `CONVFMT` alapján végzi el, `"%.6g"`), és egyet rendel a `data["12.153"]` tömb elemhez. A program ezután megváltoztatja a `CONVFMT` értékét. A '`xyz in data`' kifejezés egy új szöveget generál a `xyz` változóból, ez alkalommal `"12.15"`, mivel a `CONVFMT` csak két értékes jegyet enged meg. A teszt nem lesz sikeres, mivel a `"12.15"` szöveg nem egyezik meg a `"12.153"` szöveggel.

A konverzió szabályai szerint (lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal), az egész számokat mindig mint egész szám konvertálja szöveggé, attól függetlenül, hogy mi a `CONVFMT` értéke. Így:

```
for (i = 1; i <= maxsub; i++)
    valami történik itt az array[i] -vel
```

a program mindig működik, attól függetlenül hogy mi a `CONVFMT` értéke.

Mint a legtöbb dolog az `awk`-ban, a legtöbb esetben úgy működik, ahogy azt elvárjuk. De fontos, hogy pontosan tudjuk, hogy milyen szabályok érvényesek, mivel gyakran egy apró részlet nagy hatással lehet a programodra.

11.8. Nem inicializált változók használata mint index

Tegyük fel, hogy a bemeneti sorokat fordított sorrendben akarjuk kinyomtatni. Egy ésszerű kísérlet ennek megvalósítására az alábbi program, ami valahogy így nézne ki:

```
$ echo 'line 1
> line 2
> line 3' | awk '{ l[lines] = $0; ++lines }
> END {
>     for (i = lines-1; i >= 0; --i)
>         print l[i]
> }'
 ⊣ line 3
 ⊣ line 2
```

Sajnos a legelső bemeneti sort nem nyomtatta ki a program!

Első pillantásra a programnak működnie kellene. A `lines` változó nincs inicializálva, és egy nem inicializált változónak zérus szám értéke van. Így az `l[0]` értéket kellene kinyomtatnia.

De a lényeg, hogy az `awk` tömbök indexei **mindig** szövegek. Egy nem inicializált változó, amikor szöveg az értéke `""` és nem zérus. Így a '`line 1`' az `l[""]` -ben lesz tárolva.

A program alábbi verziója helyesen működik:

```
{ l[lines++] = $0 }
END {
    for (i = lines - 1; i >= 0; --i)
       print l[i]
}
```

Itt a '`++`' operátor arra kényszeríti a `lines` változót, hogy szám típusú legyen, így a régi érték numerikus zérus lesz, ami `"0"` szöveggé lesz konvertálva mint tömb index.

Amint az láttuk, bár egy kicsit szokatlan, de az üres szöveg (`""`) is használható mint tömb index (s.s.). Ha a '`--lint`' opció szerepel a parancssorban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), a `gawk` figyelmeztet arra, hogy egy üres szöveget akartunk indexként használni.

11.9. Többdimenziós tömbök

Egy többdimenziós tömb egy olyan tömb, amiben az elemeket indexek sorozata azonosít, és nem csak egy index. Például egy kétdimenziós tömb esetén két indexre van szükség. Általában (más programozási nyelvekben, beleértve az `awk`-ot is) egy kétdimenziós tömböt `mátrixnak` is szoktak nevezni, pl. `matrix[`*x*`,`*y*`]`.

Az `awk` támogatja a többdimenziós tömbök használatát, indexek egy szövegbe kapcsolásával. Lényegében az történik, hogy az `awk` az indexeket szöveggé alakítja (lásd 7.4. Bekezdés [Szövegek és számok konverziója], 81. oldal), és összekapcsolja őket egy speciális elválasztóval. Így egyetlen szöveget kapunk. Ezt a kombinált szöveget használja mint egy index egy egydimenziós tömbben. Az elválasztót a `SUBSEP` beépített változó értéke adja meg.

Például tegyük fel, hogy a '`foo[5,12] = "value"`' kifejezést kiértékeljük, amikor a `SUBSEP` változó értéke `"@"`. Az ötös és a 12-es számot átalakítja szövegekké és összekapcsolja őket egy '`@`' jellel közöttük, aminek az eredménye: `"5@12"`. Így a `foo["5@12"]` tömb eleme `"value"` lesz.

Ha egyszer egy tömb elemének értéket adtunk, attól kezdve nincs arra lehetőség, hogy kiderítsük, hogy egy indexet vagy több indexet használtunk. A '`foo[5,12]`' és a '`foo[5 SUBSEP 12]`' kifejezések mindig teljesen azonosak.

A `SUBSEP` alapértéke a `"\034"` szöveg, ami egy nem nyomtatható karakter, és így nem valószínű, hogy egy `awk` programban vagy adatfile-ban előfordulhat.

Egy valószínűtlen karakter használatának hasznossága abból fakad, hogy egy index értéke ami tartalmazza a `SUPSEP` szöveget olyan kombinált szöveghez vezet ami kétértelmű. Tegyük fel, hogy a `SUBSEP` értéke `"@"`, akkor a '`foo["a@b", "c"]`' és a '`foo["a", "b@c"]`' kifejezések teljesen megkülönböztethetetlenek mivel végül is a '`foo["a@b@c"]`' kifejezéssel egyeznek meg.

Ugyanazzal az '`in`' operátorral lehet ellenőrizni egy index létezését egy többdimenziós tömbben, mint egy egyszerű egydimenziós tömbben. Egyetlen index helyett indexek sorozatát kell leírni, vesszővel elválasztva zárójelekben:

(*subscript1*, *subscript2*, ...) `in` *array*

A következő példa a bemenetet mint egy két dimenziós tömb kezeli; elforgatja ezt a tömböt 90 fokkal az óramutató járásával egyező irányban, és kinyomtatja az eredményt. A program feltételezi, hogy minden sor azonos számú elemet tartalmaz.

```
awk '{
     if (max_nf < NF)
          max_nf = NF
     max_nr = NR
     for (x = 1; x <= NF; x++)
          vector[x, NR] = $x
}

END {
     for (x = 1; x <= max_nf; x++) {
          for (y = max_nr; y >= 1; --y)
               printf("%s ", vector[x, y])
          printf("\n")
     }
}'
```

Amikor az alábbi az input:

```
1 2 3 4 5 6
2 3 4 5 6 1
3 4 5 6 1 2
4 5 6 1 2 3
```

a végeredmény:

```
4 3 2 1
5 4 3 2
6 5 4 3
1 6 5 4
2 1 6 5
3 2 1 6
```

11.10. Többdimenziós tömbök elérése

Nincs speciális `for` kifejezés "többdimenziós" tömbök eléréséhez; nem is lehet, hiszen igazából nincsenek is többdimenziós tömbök; csak többdimenziós tömb *elérési mód* van.

Ugyanakkor, ha a programod mindig mint több dimenziós tömb használja a tömböt, akkor azt a hatást kelthetjük ha a `for` kifejezést (lásd 11.5. Bekezdés [Egy tömb elemeinek ellenőrzése], 126. oldal) kombináljuk a `split` beépített függvénnyel (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal). Így működik:

```
for (combined in array) {
  split(combined, separate, SUBSEP)
  ...
}
```

Ez beállítja a `combined` változót minden összefűzőtt indexre, és feldarabolja egyedi indexekre, ott ahol a `SUBSEP` értékének megfelelő szöveg található. A feldarabolt indexek a `separate` tömb elemei lesznek.

Így, ha korábban egy értéket tároltunk az `array[1, "foo"]` -ban, akkor létezik egy elem az `array` tömbben `"1\034foo"` indexel. (A `SUBSEP` alapértéke egy 034 -es kódú karakter.) Előbb vagy utóbb a `for` ciklus megtalálja ezt az elemet, és `combined` értéke a `"1\034foo"` lesz. Ezután a `split` függvényt hívja meg:

```
split("1\034foo", separate, "\034")
```

Ennek az eredménye, hogy a `separate[1]` értéke `"1"` lesz és a `separate[2]` értéke `"foo"`. Vagyis visszakaptuk az eredeti indexsorozatot.

12. Beépített függvények

A *beépített* függvények olyan függvények, amelyek mindig az `awk` programod rendelkezésére állnak. Ez a fejezet definiálja az `awk` összes beépített függvényét; néhányat más bekezdésekben is megemlítettünk már. (Te is definiálhatsz függvényeket. Lásd 13. Fejezet [Felhasználó által definiált függvények], 153. oldal.)

12.1. Beépített függvények hívása

Ahhoz, hogy egy beépített függvényt meghívhassunk, a nevét kell leírni, majd az argumentumait zárójelek között. Például, '`atan2(y + z, 1)`' a `atan2` függvényt fogja meghívni két argumentummal.

A függvénynév és a nyitó zárójel közötti szóközöket nem veszi figyelembe az `awk`, de azt tanácsoljuk, hogy ne használj szóközöket itt. A felhasználó által definiálható függvények hívásánál a szóközök nem megengedettek, és ha nem használsz szóköz karaktert a függvénynév és a nyitó zárójel között, akkor könnyebb ezt a hibát elkerülni.

Minden beépített függvény elfogad egy adott számú argumentumot. Bizonyos esetekben, az argumentumok elhagyhatók. Hogy mi történik akkor, amikor egy függvénynek nem adunk meg argumentumot, az függvényről függvényre változik, és az adott függvénynél ezt bemutatjuk. Néhány `awk` implementáció a függvénynek átadott extra argumentumokat nem veszi figyelembe. Ugyanakkor a `gawk` esetén, ez az eset egy végzetes hibát fog generálni.

Amikor egy függvény meghívásra kerül, akkor az argumentumait adó kifejezések először teljesen kiértékelődnek, és csak utána történik meg függvényhívás. Például az alábbi programrészletben:

```
i = 4
j = sqrt(i++)
```

a `i` változó értéke öt lesz mielőtt az `sqrt` függvény meghívódik. A függvény argumentuma négy lesz.

A függvény argumentumainak kiértékelési sorrendje nem definiált. Így, nem szabad arra számítani, hogy az argumentumok balról jobbra vagy jobbról balra értékelődnek ki. Például:

```
i = 5
j = atan2(i++, i *= 2)
```

Ha a kiértékelés balról jobbra halad, akkor az `i` értéke először hat lesz, majd 12 és az `atan2` argumentumai hat és 12 lesz. De ha a kiértékelés jobbról balra halad, akkor az `i` értéke először 10 lesz, majd 11, és az `atan2` argumentumai 11 és 10 lesz.

12.2. Numerikus beépített függvények

Az alábbiakban megadjuk a számokkal dolgozó beépített függvények listáját. Az opcionális argumentumokat szögletes zárójelek veszik körül ("[" és "]").

`int(`*x*`)`
: Ez a függvény visszaadja az *x*-hez legközelebbi egész számot, ami *x* és zérus közé esik, zérus felé csonkolva.

  Például, `int(3)` eredménye három, `int(3.9)` eredménye három, `int(-3.9)` eredménye -3 és `int(-3)` eredménye szintén -3.

`sqrt(`*x*`)`
: Ez a függvény visszaadja az *x* pozitív négyzetgyökét. Hibát jelez ha *x* negatív. Így `sqrt(4)` eredménye kettő.

`exp(`*x*`)`
: Ez a függvény visszaadja az *x* természetes alapú hatványát (`e ^` *x*), vagy hibát jelez ha *x* az értelmezési tartományon kívül esik. Az *x* értelmezési tartománya a géped lebegőpontos ábrázoló képességétől függ.

`log(`*x*`)`
: Ez a függvény visszaadja az *x* természetes alapú logaritmusát, ha *x* pozitív; egyébként hibát jelez.

`sin(`*x*`)`
: Ez a függvény visszaadja az *x* (radián) szinuszát.

`cos(`*x*`)`
: Ez a függvény visszaadja az *x* (radián) koszinuszát.

`atan2(`*y*`,` *x*`)`
: Ez a függvény visszaadja az *y* `/` *x* (radián) inverz tangensét (arctan).

`rand()`
: Ez a függvény egy véletlen számot ad vissza. A `rand` által visszaadott értékek egyenletesen oszlanak el egy és zérus között. Az érték viszont soha nem zérus és nem egy.

  Gyakran véletlen egész számokra van szükséged. Itt egy felhasználó által definiálható függvény, ami véletlen, nem negatív, *n*-nél kisebb egész számokat generál:

```
function randint(n) {
     return int(n * rand())
}
```

  A szorzás egy zérusnál nagyobb, és *n*-nél kisebb véletlen valós számot generál. Ezután egész számmá konvertáljuk (az `int` függvényt használva), ami zérus és `n` $-$ 1 közé esik, a határokat is beleértve.

  Itt egy másik példa, ahol hasonló függvényt használunk egy és *n* közé eső számok generálására. Ez a program minden bemeneti rekordnál egy új véletlen számot nyomtat ki.

```
awk '
# Szimulált kockadobás
function roll(n) { return 1 + int(rand() * n) }
```

```
# 3 db hat-oldalú kockával dobunk és
# a dobott pontok összegét nyomtatjuk ki.
{
      printf("%d points\n",
             roll(6)+roll(6)+roll(6))
}'
```

Figyelem: A legtöbb awk implementációban, a gawk-ot is beleértve, a rand függvény ugyanattól a számtól (*seed*) kezd el véletlen számokat generálni, amikor az awk újraindul. Így a program minden alkalommal ugyanazt az eredményt fogja produkálni. A számok véletlenszerűek egy awk futtatás alatt, de megjósolhatók egymás utáni futtatások során. Ez hasznos lehet tesztelésnél, de ha azt akarod, hogy a programod minden futtatás során más számot generáljon, akkor más kezdő számot (seed) kell megadni minden alkalommal. Ezt az srand függvénnyel lehet elérni.

srand([*x*])

Az srand függvény a véletlen szám generálás kezdőpontját (seed) állítja be az x által megadott értékkel.

Minden kezdőpont más véletlen számsorozathoz vezet.[1] Így ha a kezdőpont ugyanaz egy második alkalommal, akkor ugyanazt a sorozatot fogod megkapni, mint az első alkalommal.

Ha nem adsz meg argumentumot és csak az srand() formát használod, akkor az aktuális dátumot és időpontot használja kezdőpontként. Ez a módja annak, hogy igazi véletlen számokat generáljunk.

Az srand függvény visszatérési értéke az előző kezdőpont értéke. Ez lehetővé teszi a megadott kezdőpontok nyilvántartását és ugyanazon véletlen szám sorozatok generálását.

12.3. Szövegmanipuláló beépített függvények

Ebben a bekezdésben található függvények szövegeket vizsgálnak vagy változtatnak meg. Az opcionális argumentumok szögletes zárójelek között ("[" and "]") jelennek meg.

[1] A számítógépek által generált véletlen számok nem igazán véletlenszerűek. Technikai kifejezéssel élve, ezeket "pszeudo-véletlen" számoknak nevezik. Ez azt jelenti, hogy bár a sorozatban az egymás után következő számok véletlenszerűek, de valójában ugyanazt a sorozatot többször tudod generálni.

`index(`*in*`,` *find*`)`

Ez a függvény az *in* szövegben a *find* szöveg első előfordulását keresi meg, és azt a karakterpozíciót adja vissza az *in* szövegen belül, ahol a *find* szöveg kezdődik. Például:

```
$ awk 'BEGIN { print index("peanut", "an") }'
⊣ 3
```

Ha nem találja a *find* szöveget, akkor az `index` függvény zérussal tér vissza. (Az `awk` szövegek első karaktere az egyes pozíciónál található.)

`length(`[*string*]`)`

Ez a függvény a *string* szövegben található karakterek számát adja meg. Ha a *string* egy szám, akkor a szám ábrázolásához szüséges számjegyek számával tér vissza. Például, `length("abcde")` visszatérési értéke öt. Ezzel ellentétben, az `length(15 * 35)` eredménye három lesz. Miért? Nos, 15 * 35 = 525, amit szöveggé konvertál `"525"`, amiben három karakter van.

Ha nincs argumentum megadva, akkor a `length` függvény a `$0` hosszát adja meg.

Az `awk` régebbi verzióiban a `length` függvényt zárójelek nélkül is meg lehetett hívni. A POSIX szabvány ezt "helyteleníti". Ez azt jelenti, hogy ez a lehetőség végül is el fog tűnni a szabványból. Így a maximális hordozhatóság érdekében a legjobb a zárójeleket kitenni.

`match(`*string*`,` *regexp*`)`

A `match` függvény a *string* szövegben a leghosszabb, legbaloldalibb szövegrészletet keresi meg, ami illeszkedik a *regexp* reguláris kifejezésre. Azt a karakterpozíciót adja vissza, vagy *indexet*, ahol a szövegrészlet kezdődik (ez egy, ha a *string* elején kezdődik). Ha nem volt illeszkedés, akkor a visszatérési érték zérus.

A `match` függvény az `RSTART` beépített változónak az indexet adja meg, mint új érték. Ezen kívül az `RLENGTH` beépített változó az illeszkedő szöveg hosszát adja meg. Ha nem volt illeszkedés, az `RSTART` értéke zérus, az `RLENGTH` értéke −1 lesz.

Például:

```
awk '{
       if ($1 == "FIND")
         regex = $2
       else {
         where = match($0, regex)
         if (where != 0)
           print "Match of", regex, "found at", \
                     where, "in", $0
       }
}'
```

Ez a program olyan sorokat keres, amelyek illeszkednek a `regex` változóban tárolt kifejezésre. Ez a reguláris kifejezés megváltoztatható, így ha a sorban az első szó a '`FIND`', akkor a `regex` új értéke a sor második szava lesz. Így ha ez a bemenet:

```
FIND ru+n
My program runs
but not very quickly
FIND Melvin
JF+KM
This line is property of Reality Engineering Co.
Melvin was here.
```

az `awk` ezt fogja kinyomtatni:

```
Match of ru+n found at 12 in My program runs
Match of Melvin found at 1 in Melvin was here.
```

`split(`*string*, *array* [`,` *fieldsep*]`)`

Ez a függvény a *string* szöveget vágja fel olyan darabokra, amelyeket *fieldsep* szöveg választ el, és a darabokat az *array* tömbben tárolja. Az első darabot a *array*`[1]` -ba teszi, a másodikat a *array*`[2]`-ba és így tovább. A harmadik argumentum, a *fieldsep*, egy reguláris kifejezés ami azt adja meg, hogy hol kell a *string* szöveget felvágni (ugyanúgy mint ahogy az `FS` azt adja meg, hogy hol kell a mezőket feldarabolni). Ha a *fieldsep* nincs megadva, akkor az `FS` értékét használja. A `split` függvény visszatérési értéke a generált szöveg darabok száma.

A `split` függvény a bemeneti rekord mezőkre darabolásához hasonló módon vágja fel a szöveget. Például:

```
split("cul-de-sac", a, "-")
```

felvágja a '`cul-de-sac`' szöveget három darabba a '`-`' jel mentén. Az `a` tömb tartalma a következő lesz:

```
a[1] = "cul"
a[2] = "de"
a[3] = "sac"
```

A `split` függvény visszatérési értéke három lesz.

Ugyanúgy mint a meződarabolásnál, amikor a *fieldsep* értéke `" "`, akkor a kezdő és záró szóközöket és tab karaktereket nem veszi figyelembe és az elemeket szóközök és tab karakterek (sorozatai) választhatják el.

Szintén igaz, hogy ha a *fieldsep* értéke egy üres szöveg, akkor a szöveg minden karaktere egy külön tömb elembe kerül. (Ez egy `gawk` specifikus kiterjesztés.)

Az `awk` újabb implementációi, a `gawk` is, megengedi, hogy a harmadik argumentum egy regexp konstans (`/abc/`) vagy egy szöveg legyen (s.s.). A POSIX szabvány szintén megengedi ezt.

Mielőtt a szöveget felvágná, a `split` függvény kitörli az *array* tömb minden létező elemét (s.s.).

Ha a *string* szöveg egyik részlete sem illeszkedik a *fieldsep* szövegre, akkor az *array* tömbnek csak egy eleme lesz. Az elem értéke maga az eredeti szöveg, a *string*, lesz.

`sprintf(`*format*`,` *expression1*`,...)`

Ez a függvény azzal a szöveggel tér vissza (anélkül, hogy kinyomtatná) amit a `printf` függvény nyomtatna ki ugyanazokkal az argumentumokkal (lásd 6.5. Bekezdés [Nyomtatás a `printf` kifejezéssel], 64. oldal). Például:

```
sprintf("pi = %.2f (approx.)", 22/7)
```

a visszatérési értéke a `"pi = 3.14 (approx.)"` lesz.

`sub(`*regexp*`,` *replacement* [`,` *target*]`)`

A `sub` megváltoztatja a *target* értékét. A függvény a *regexp* reguláris kifejezésre illeszkedő legbaloldalibb, leghosszabb illeszkedést keresi a *target*-en belül. Ezután az egész szöveget megváltoztatja úgy, hogy az illeszkedő szöveg részletet lecseréli a *replacement* szövegre. A módosított szöveg lesz a *target* új értéke.

Ez a függvény különleges, mert a *target* értékét nem csak egy új érték "kiszámításához" használja, hanem meg is változtatja. Csak bizonyos kifejezések állhatnak a *target* helyén, úgy mint változók, mezők vagy tömb elemek, ahova a `sub` el tudja tárolni a módosított értéket. Ha ez az argumentum nincs megadva, akkor az alapérték a `$0`.

Például:

```
str = "water, water, everywhere"
sub(/at/, "ith", str)
```

az `str` új értéke a `"wither, water, everywhere"` lesz, mivel lecseréli az '`at`' legbaloldalibb, leghosszabb előfordulását a '`ith`' szövegre.

A `sub` függvény az elvégzett behelyettesítések száma lesz (vagy egy vagy zérus).

Ha a '`&`' speciális karakter előfordul a *replacement*-ben, akkor ez a *regexp*-re illeszkedő szöveg részletet jelenti. (Ha regexp egynél több szövegre illeszkedik, akkor ez a szöveg részlet változhat.) Például:

```
awk '{ sub(/candidate/, "& and his wife"); print }'
```

a '`candidate`' első előfordulását lecseréli a '`candidate and his wife`' szövegre a bemeneti file minden sorában.

Itt egy másik példa:

```
awk 'BEGIN {
        str = "daabaaa"
        sub(/a*/, "c&c", str)
        print str
}'
⊣ dcaacbaaa
```

Ez azt mutatja be, hogy a '`&`' hogyan képes nem konstans szöveget reprezentálni, és bemutatja, hogy mit jelent a "legbaloldalibb, leghosszabb" szabály a reguláris kifejezések illesztésében (lásd 4.6. Bekezdés [Mennyi szöveg illeszkedik?], 35. oldal).

A '`&`' speciális karakter hatása kikapcsolható, ha egy '\' karaktert teszünk eléje. Mint mindig, ha egy szövegbe akarsz egy '\' karaktert írni, akkor kettőt kell belőle leírni. Tehát a '`\\&`' karaktereket kell egy szöveg konstansba beírni, hogy maga a '`&`' karakter jelenjen meg a csere után. Például, itt bemutatjuk, hogy hogyan lehet az első '`|`' karaktert lecserélni egy '`&`' karakterre minden sorban:

```
awk '{ sub(/\|/, "\\&"); print }'
```

Megjegyzés: Ahogy azt korábban írtuk a `sub` harmadik argumentuma egy változó, egy mező vagy egy tömb eleme kell legyen. Az `awk` néhány verziója megengedi, hogy a harmadik argumentum ne egy lvalue legyen. Ebben az esetben a `sub` ugyanúgy viselkedik, de a csere eredményét eldobja, mivel nincs hol tárolnia. Ezek az `awk` verziók elfogadnák az alábbi kifejezést is:

```
sub(/USA/, "United States", "the USA and Canada")
```

A történelmi kompatibilitás miatt a `gawk` elfogadja a fenti hibás kódot is, de bármilyen "nem megváltoztatható" objektum, mint harmadik argumentum esetén fatális hibával leáll.

Végül, ha a *regexp* nem egy reguláris kifejezés, akkor egy szöveggé konvertálja majd a szöveg értékét használja regexpként az illesztésben.

`gsub(`*regexp*`,` *replacement* [`,` *target*]`)`
: Ez a függvény hasonló a `sub` függvényhez, kivéve, hogy ez a függvény *minden* legbaloldalibb, leghosszabb, *nem átfedő*, illesz-

kedő szöveg részletet lecserél. A '`g`' a `gsub` névben a "globálist" jelenti, vagyis cseréljen mindenhol. Például:

```
awk '{ gsub(/Britain/, "United Kingdom"); print }'
```

a '`Britain`' szöveg minden előfordulását lecseréli a '`United Kingdom`' szövegre a bemeneti rekordokban.

A `gsub` függvény az elvégzett cserék számával tér vissza. Ha a harmadik argumentum nincs megadva, akkor a keresést és a cserét a bemeneti rekordon, `$0`, végzi el.

Ugyanúgy mint a `sub`-nál a '`&`' és '`\`' karakterek speciálisak, és a harmadik argumentum egy lvalue kell legyen.

`gensub(`*regexp*`,` *replacement*`,` *how* [`,` *target*]`)`

A `gensub` egy általános helyettesítő függvény. Mint a `sub` és a `gsub` a *target* szövegben a *regexp* kifejezésre illeszkedő szöveg darabot keres. Ugyanakkor a `sub` és a `gsub` függvényekkel ellentétben a módosított szöveg a függvény visszatérési értéke lesz, és az eredeti *target* szöveg *nem* lesz módosítva. Ha a *how* egy olyan szöveg ami a '`g`' vagy '`G`' karakterrel kezdődik, akkor az összes illeszkedő szövegdarabot lecseréli a *replacement* szövegre. Minden más esetben, a *how* egy szám kell legyen, ami azt adja meg, hogy hanyadik illeszkedést kell lecserélni. Ha a *target* nincs megadva, akkor a `$0`-t használja.

A `gensub` függvénynek van egy olyan szolgáltatása is, ami nem használható sem a `sub` sem a `gsub` függvényekkel: a reguláris kifejezés komponenseinek felhasználása a csereként szolgáló szövegben. Ezt úgy lehet elérni, hogy zárójelek közé kell tenni a regexp adott komponensét és a csereként szolgáló szövegben a '`\`*n*' kifejezést kell használni, ahol az *n* egy egy és kilenc közé eső szám. Például:

```
$ gawk '
> BEGIN {
>      a = "abc def"
>      b = gensub(/(.+) (.+)/, "\\2 \\1", "g", a)
>      print b
> }'
 ⊣ def abc
```

Ahogy azt a `sub` függvénynél elmagyaráztuk, két '`\`' karaktert kell leírni ahhoz, hogy egy '`\`' karakter bekerüljön a szövegbe.

A csereként szolgáló szövegben a '`\0`' karakter sorozat a teljes illeszkedő szöveget jelenti, ugyanúgy mint a '`&`' karakter.

Ez a példa bemutatja, hogy hogyan szabályozza a harmadik argumentum azt, hogy melyik illeszkedés lesz lecserélve.

```
$ echo a b c a b c |
> gawk '{ print gensub(/a/, "AA", 2) }'
```

```
⊣ a b c AA b c
```

Ebben az esetben a `$0` lesz a negyedik argumentum. A `gensub` az új, csere utáni szöveggel tér vissza, amit átad a `print`-nek, hogy nyomtassa ki.

Ha a *how* argumentum egy olyan szöveg, ami nem '`g`' vagy '`G`' karakterekkel kezdődik, vagy egy zérusnál kisebb szám, akkor csak egy helyettesítés hajtódik végre.

Ha a *regexp* nem illeszkedik a *target* egyik részére sem, akkor a `gensub` visszatérési értéke az eredeti, változatlan *target* szöveg lesz.

A `gensub` egy `gawk` kiegészítés és "compatibility" módban nem használható (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

`substr(`*string*`, `*start* `[, `*length*`])`

Ez a függvény a *string* szöveg egy a *start* karakterpozíciónál kezdődő és *length* karakternyi részszövegével tér vissza. A szöveg első karakterpozíciója az egy. Például, `substr("washington", 5, 3)` eredménye az `"ing"`.

Ha a *length* nincs megadva, akkor a függvény a *string* szövegnek a *start* karakterpozíciónál kezdődő és a szöveg végéig tartó részszövegével tér vissza. Például, `substr("washington", 5)` eredménye az `"ington"`. Ugyanez történik ha a *length* nagyobb hosszat ad meg, mint ami a *start* karakterpozíciótól a szöveg végéig tart.

Megjegyzés: A `substr` függvény által visszaadott szövegnek *nem* lehet értéket adni. Így az alábbi kísérlet a szöveg egy részének megváltoztatására hibás:

```
string = "abcdef"
# ezt szeretnénk "abCDEf", de nem fog menni !
substr(string, 3, 3) = "CDE"
```

A `substr` függvényt nem lehet használni a `sub` vagy a `gsub` függvények harmadik argumentumában sem:

```
gsub(/xyz/, "pdq", substr($0, 5, 20))  # HIBÁS
```

`tolower(`*string*`)`

Ez a függvény a *string* szöveg egy másolatával tér vissza, amiben minden nagybetűt lecserél az azonos kisbetűre. Azokat a karaktereket, amelyek nem betűk, nem változtatja meg. Például, `tolower("MiXeD cAsE 123")` eredménye a `"mixed case 123"`.

`toupper(`*string*`)`

Ez a függvény a *string* szöveg egy másolatával tér vissza, amiben minden kisbetűt lecserél az azonos nagybetűre. Azokat a karaktereket, amelyek nem betűk, nem változtatja meg. Például, `tolower("MiXeD cAsE 123")` eredménye a `"MIXED CASE 123"`.

További információ a '\' és a '&' karakterek használatáról a `sub`, a `gsub` és a `gensub` függvényekben

Amikor a `sub`, a `gsub` vagy a `gensub` függvényen belül a '\' vagy a '&' karaktert szeretnéd használni, akkor fontos arra emlékezni, hogy az *escape karakterek feldolgozásának* több szintje van.

Először is van a *lexikális* szint, amikor is az `awk` beolvassa a programodat és felépíti a programod belső reprezentációját, amit majd végre tud hajtani.

Ezután van a futtatási szint, amikor az `awk` azt ellenőrzi a csereként szolgáló szövegekben, hogy mit is kell csinálnia.

Mindkét szinten az `awk` ellenőrzi, hogy milyen karakter következik a '\' karakter után. A lexikális szinten escape szekvenciákat keres (4.2. Bekezdés [Escape szekvenciák], 24. oldal). Így minden '\' karakterért, amit az `awk` majd a futtatási szinten vesz figyelembe két '\' karaktert kell megadni a lexikális szinten. Ha egy olyan karakter követi az '\' karaktert, ami nem egy érvényes escape szekvencia, akkor a Unix `awk` és a `gawk` is egyszerűen eltávolítja a '\' karaktert, és csak a követő karaktert tartja meg a szövegben. Így például, a `"a\qb"` szöveg a `"aqb"` szövegnek felel meg.

A futtatási szinten a különböző függvények különböző módon kezelik a '\' és a '&' karakterek használatát. A helyzet (sajnos) elég bonyolult.

Történelmileg a `sub` és a `gsub` függvények a '\&' karaktersorozatot speciálisan kezelték; ezt a sorozatot a csereként használt szövegben lecserélte a '&' karakterre. Minden más '\' karakter a csereként szolgáló szövegen belül, amit nem a '&' karakter követett, változatlan maradt. Ezt az alábbi táblázat illusztrálja:

Te gépeled	a `sub` ezt látja	a `sub` ezt generálja
`\&`	`&`	az illeszkedő szöveg
`\\&`	`\&`	maga a '&' karakter
`\\\&`	`\&`	maga a '&' karakter
`\\\\&`	`\\&`	maga a '\&' karaktersorozat
`\\\\\&`	`\\&`	maga a '\&' karaktersorozat
`\\\\\\&`	`\\\&`	maga a '\\&' karaktersorozat
`\\q`	`\q`	maga a '\q' karaktersorozat

Ez a táblázat bemutatja mind a lexikális feldolgozást, ahol a páratlan számú '\' karakterek páros számúvá válnak a futtatási szinten, és a `sub` által végrehajtott feldolgozást a futtatási szinten. (Az egyszerűség kedvéért a későbbi táblázatokban csak a lexikális szinten páros számú '\' karakterrel megadott eseteket mutatjuk be.)

A probléma a történelmi megközelítéssel az, hogy lehetetlen azt az esetet megadni, amikor az illeszkedő szöveget egy '\' karakter előzi meg.

Az 1992-es POSIX szabvány megpróbálta ezt a problémát megoldani. A szabvány azt mondja ki, hogy a `sub` és a `gsub` függvények a '\' karakter után vagy egy '\' vagy egy '&' karaktert keresnek. Ha bármelyik eset előfordul, akkor a '\' karaktert követő karakter érintetlenül kerül be a szövegbe. Tehát a '\' és a '&' karakterek értelmezése a következő:

Te gépeled	a `sub` ezt látja	a `sub` ezt generálja
`&`	`&`	az illeszkedő szöveg
`\\&`	`\&`	maga a '&' karakter
`\\\\&`	`\\&`	a '\' karakter és az illeszkedő szöveg
`\\\\\\&`	`\\\&`	maga a '\&' karaktersorozat

Ez úgy tűnik megoldja a problémát. Sajnos, a szabvány szövege szokatlanul fogalmazza ezt meg. Azt mondja, hogy a '\' karakter kikapcsolja bármely követő karakter speciális jelentését, de a speciális jelentés kivéve a '\' és a '&' karaktereket nem definiált. Ez a fogalmazás két problémához vezet.

1. A '\' karaktert kétszer kell leírni a *replacement* szövegben, aminek hatására a régi `awk` programok nem fognak helyesen működni.
2. Ha biztosak akarunk abban lenni, hogy az `awk` program hordozható, akkor a *replacement* szövegben *minden* karakter elé egy '\' karaktert kell tenni.[2]

A POSIX szabvány átdolgozás alatt áll.[3] A fenti problémák miatt, az új, módosított változat olyan szabályokat ad meg, amelyek jobban hasonlítanak a régiekhez. Az új szabály speciális eseteket definiál arra az esetre, ha egy '\' karakternek kell megelőznie az illeszkedő szöveget.

Te gépeled	a `sub` ezt látja	a `sub` ezt generálja
`\\\\\\&`	`\\\&`	maga a '\&' karaktersorozat
`\\\\&`	`\\&`	a '\'karakter és az illeszkedő szöveg
`\\&`	`\&`	maga a '&' karakter
`\\q`	`\q`	maga a '\q' karaktersorozat

Röviden, a futtatási szinten most három speciális eset van, a '\\\&', a '\\&' és a '\&'. Ezzel ellentétben régen csak egy volt. Ugyanakkor, mint a régi szabályban, bármely '\' karakter, ami nem része a három kivételnek nem speciális, és maga a karakter jelenik meg.

A `gawk` 3.0-ás verziója az új POSIX ajánlást követi a `sub` és a `gsub` esetén. Jelenleg még nem lehet tudni, hogy az új ajánlás végleg bekerül-e

[2] Ez biztos, hogy nem volt szándékos.

[3] Jelenleg, 1999 április, végső jóváhagyás előtt áll és remélhetőleg valamikor 1999-ben megjelenik az új változat.

a szabványba. A `gawk` jövőbeli verziói a szabványt fogják követni, bármi is legyen a végleges szabványban; ekkor ezt a könyvet is frissíteni fogjuk.

A szabályok a `gensub` esetén sokkal egyszerűbbek. A futtatási szinten, amikor a `gawk` egy '\' karaktert vesz észre, és a követő karakter egy számjegy, akkor a reguláris kifejezésben elhelyezett zárójelek a közötti illeszkedő szöveg fog megjelenni a kimenetben. Minden más esetben, akármi is legyen a '\' karakter után, maga a karakter fog megjelenni és a '\' karakter nem.

Te gépeled	a `gensub` ezt látja	a `gensub` ezt generálja
`&`	`&`	az illeszkedő szöveg
`\\&`	`\&`	maga a '`&`' karakter
`\\\\`	`\\`	maga a '\' karakter
`\\\\&`	`\\&`	a '\' karakter és az illeszkedő szöveg
`\\\\\\&`	`\\\&`	maga a '`\&`' karaktersorozat
`\\q`	`\q`	maga a '`q`' karakter

A lexikális és a futtatási szint bonyolultsága és a `sub` és a `gsub` függvények speciális esetei miatt azt tanácsoljuk, hogy ha `gawk`-ot használsz, akkor a `gensub` függvényt használd helyettesítésre.

12.4. Bemenettel és kimenettel (I/O) kapcsolatos beépített függvények

Az alábbi függvények a bemenettel vagy a kimenettel kapcsolatosak. Az opcionális argumentumok szögletes zárójelek ("[" and "]") között jelennek meg.

`close(`*filename*`)`
: Lezárja a *filename* file-t, akár bemenetre vagy kimenetre lett megnyitva. Az argumentum lehet egy shell parancs is, ami az eredményt egy file-ba irányította át; ekkor a csövet zárja le. Lásd 6.8. Bekezdés [Bemeneti és kimeneti file-ok és csövek lezárása], 74. oldal.

`fflush(`[*filename*]`)`
: Ez a függvény kiüríti a *filename*-hez tartozó buffer-t, és kinyomtatja a buffer tartalmát. A *filename* lehet egy file neve, de egy shell parancs is lehet.

  Sok segédprogram *bufferelі* a kimenetét; ami azt jelenti, hogy egy file-ba vagy a terminálra kiirandó adatokat a memóriában tartja addig, amikor is érdemes az egész adatot a kimeneti eszköznek átadni. Ez a megoldás sokkal hatékonyabb, mint amikor minden kis adatot azonnal megjelenítünk. Ugyanakkor alkalmanként fontos lehet a bufferek tartalmát kiíratni, üríteni az adott pillanatban, még akkor is ha a buffer nincs tele. Erre való az `fflush` függvény. A `gawk` is buffereli a kimenetét, és az `fflush` függvénnyel lehet arra kényszeríteni, hogy az adott pillanatban a buffer tartalmát kiírja.

Az `fflush` függvényt 1994-ben adták hozzá a Bell laboratóriumban fejlesztett `awk` verzióhoz; a függvény nem része a POSIX szabványnak, és nem használható ha a '`--posix`' opció szerepel a parancssorban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

A `gawk` kétféle módon fejlesztette tovább az `fflush` függvényt. Az első, hogy argumentum nélkül is meg lehet hívni a függvényt. Ebben az esetben a szabványos kimenet lesz ürítve. A második, hogy az üres szöveg (`""`) is megadható argumentumként, amikor is *minden* megnyitott file vagy cső buffere ürítve lesz.

Az `fflush` függvény zérus értékkel tér vissza ha sikeres volt az ürítés, és egy nem zérus értékkel minden más esetben.

`system(`*command*`)`

A `system` függvény teszi lehetővé, hogy a felhasználó bármilyen az operációs rendszerben engedélyezett parancsot végrehajthasson, majd az `awk` program futása folytatódjon. A `system` függvény a *command* szövegben adott parancsot hajtja végre. A parancs végrehajtása által visszaadott érték lesz a függvény visszatérési értéke.

Például, ha az alábbi programrészletet teszed be az `awk` programodba:

```
END {
     system("date | mail -s 'awk run done' root")
}
```

akkor a rendszeradminisztrátor egy levelet fog kapni, amikor az `awk` program befejezte a bemenete feldolgozását.

A `print` vagy a `printf` átirányítása egy csőbe gyakran elég a feladat elvégzéséhez. Ha sok parancsot kell végrehajtani, akkor hatékonyabb egy csőbe nyomtatni, ami egy shell-be van átirányítva:

```
while (amíg van mit csinálni)
    print command | "/bin/sh"
close("/bin/sh")
```

Ugyanakkor ha az `awk` programod a felhasználótól is vár bemenetet (interaktív), akkor a `system` függvény jól használható nagyobb programok indítására, mint egy shell vagy egy szövegszerkesztő.

Néhány operációs rendszer nem támogatja a `system` függvény használatát, ebben az esetben fatális hibát okoz a függvény hívása.

Interaktív és nem interaktív bufferelés

Érdemes itt megjegyezni, hogy a bufferelés zavarba ejtő is lehet ha a programod *interaktív*; pl. a program kommunikál a felhasználóval.[4]

Az interaktív programok általában *soronként bufferelik* a kimenetüket. A nem interaktív programok addig várnak amíg be nem tellik a bufferük, ami több sornyi kimenet is lehet, és csak ekkor írják ki a buffer tartalmát.

Itt egy példa a különbség bemutatására.

```
$ awk '{ print $1 + $2 }'
1 1
⊣ 2
2 3
⊣ 5
Control-d
```

Minden kimeneti sort azonnal kinyomtat. Hasonlítsd ezt össze a következő példával.

```
$ awk '{ print $1 + $2 }' | cat
1 1
2 3
Control-d
⊣ 2
⊣ 5
```

Itt addig nem ír ki semmit, amíg a *Control-d* billentyű kombinációt meg nem kapja, mivel minden bufferelt, és egy menetben küld el mindent a `cat` parancsnak a csövön keresztül.

A kimeneti bufferelés szabályozása a `system` függvénnyel

Az `fflush` függvénnyel közvetlen szabályozható az egyes file-ok vagy csövek bufferelése. Ugyanakkor sok más `awk` implementációban ez a függvény nem támogatott. Egy alternatív megoldás a bufferek kiürítésére, ha a `system` függvényt hívjuk meg egy üres szöveg argumentummal:

```
system("")   # kimenet ürítése
```

A `gawk` a `system` függvény ilyen használatát speciálisan kezeli, és van annyira okos, hogy nem futtat egy shell-t (vagy más parancsértelmezőt) egy üres paranccsal. Ezért a `gawk` esetén ez nem csak hasznos megoldás, de hatékony is. Bár ez a megoldás más `awk` implementációkban is működni fog, de nem biztos, hogy azok az `awk` implementációk nem indítanak el egy shell-t. (Más implementációk lehet hogy csak a szabványos kimenetet ürítik, és nem minden bufferelt kimenetet.)

---

[4] Egy program interaktív, ha a szabványos kimenet a terminál eszközhöz van kapcsolva.

Ha arra gondolsz, hogy egy programozó mit várna el, akkor egyértelmű, hogy a `system` függvény üríti a még ki nem nyomtatott buffereket. Az alábbi program:

```
BEGIN {
     print "first print"
     system("echo system echo")
     print "second print"
}
```

ezt kell, hogy nyomtassa

```
first print
system echo
second print
```

és nem ezt

```
system echo
first print
second print
```

Ha az `awk` nem üríti a buffereit a `system` függvény hívása előtt, akkor az utóbbi (nem kívánt) eredményt kapnánk.

12.5. Dátumfeldolgozó függvények

Az `awk` programok egyik gyakori felhasználása a log file-ok feldolgozása, amelyek dátum és idő információt is tartalmaznak. Sok program a `time` rendszerfüggvény által megadott időt használja a log file-okban. Ez az idő, a POSIX rendszereken, az 1970 január 1-e (UTC), éjfél óta eltelt másodpercek száma.

Azért hogy ezeket a log file-okat egyszerűbb legyen feldolgozni, a `gawk` két függvényt biztosít az időbélyegek kezelésére. Mind a két függvény `gawk` kiterjesztés; sem a POSIX szabvány nem tartalmazza, sem más `awk` implementációk.

Az opcionális paraméterek szögletes zárójelek ("[" and "]") között jelennek meg.

`systime()`
: Ez a függvény az aktuális időt adja meg. Ez az idő, a POSIX rendszereken, az 1970 január 1-e (UTC), éjfél óta eltelt másodpercek száma. Más rendszereken más lehet.

`strftime(`[*format* [`,` *timestamp*]]`)`
: Ez a függvény egy szöveggel tér vissza. Hasonló az ANSI C-ben található azonos nevű függvényhez. A *timestamp* által megadott időből egy szöveget készít a *format* mint formátumleíró alapján. A *timestamp* ugyanabban a formában van megadva, mint amit a `systime` függvény visszaadna. Ha nincs megadva

a *timestamp* argumentum, akkor a `gawk` az aktuális időt veszi alapul. Ha a *format* argumentum nincs megadva, akkor az `strftime` függvény a `"%a %b %d %H:%M:%S %Z %Y"` szöveget fogja használni. Ez a formátum (majdnem) ugyanaz mint amit a `date` segédprogram használ. (A `gawk` 3.0-ás verziója előtti verziókban a *format* argumentumot mindig meg kellett adni.)

A `systime` függvény lehetővé teszi, hogy időbélyegeket hasonlíts össze az aktuális időponttal. Például könnyű meghatározni, hogy mennyivel korábban lett az adott log bejegyzés rögzítve.

Az `strftime` függvény lehetővé teszi, hogy ember számárá is olvasható formában jelenjen meg az időbélyeg. Hasonló az `sprintf` függvényhez (lásd 12.3. Bekezdés [Szövegmanipuláló beépített függvények], 135. oldal), mivel minden nem formátum leíró karaktert ugyanúgy bemásol a szövegbe és csak a formátum leírók helyeire teszi be az adott időpont, adott komponensét.

Az ANSI C szabvány által garantált, hogy az `strftime` függvény az alábbi formátumleírókat támogatja:

`%a` A hét napjának a rövidített neve a helyi nyelven.

`%A` A hét napjának a neve a helyi nyelven.

`%b` A hónap rövidített neve a helyi nyelven.

`%B` A hónap neve a helyi nyelven.

`%c` A helyi szokásoknak "megfelelő" dátum és idő reprezentáció.

`%d` A hónap napja, mint decimális szám (01–31).

`%H` Az óra (24 órás beosztásban) mint decimális szám (00–23).

`%I` Az óra (12 órás beosztásban) mint decimális szám (01–12).

`%j` Az év napja, mint decimális szám (001–366).

`%m` A hónap, mint decimális szám (01–12).

`%M` A perc, mint decimális szám (00–59).

`%p` Az AM/PM jelölésnek megfelelő helyi rövidítés a 12 órás beosztáshoz.

`%S` A másodperc, mint decimális szám (00–60).[5]

`%U` A hét száma az évben (az első vasárnap, mint az első hét első napja) mint decimális szám (00–53).

`%w` A hét napja, mint decimális szám (0–6). Vasárnap a zérus.

[5] A Föld szabálytalan forgása miatt és hogy ma már több tizedes pontossággal tudjuk az időt mérni, előfordulhat, hogy az IERS elrendel egy szökőmásodpercet. Ezért alkalmanként a másodperc értéke 60 is lehet.

`%W` A hét száma az évben (az első hétfő, mint az első hét első napja) mint decimális szám (00–53).

`%x` A helyi szokásoknak "megfelelő" dátum reprezentáció.

`%X` A helyi szokásoknak "megfelelő" idő reprezentáció.

`%y` Az év az évszázadok nélkül, mint decimális szám (00–99).

`%Y` Az év az évszázadokat is megadva, mint decimális szám (pl. 1995).

`%Z` Az időzóna neve vagy rövidítése, vagy semmi ha az időzóna nem határozható meg.

`%%` Maga a '`%`' karakter.

Ha a formátum leíró nem a fentiek közül valamelyik, akkor a viselkedés nem definiált.[6]

Informálisan, a *helyi* azt a földrajzi helyet jelenti, ahol a programnak futnia kell. Például az 1991 szeptember 4-ei dátum egy általános rövidítése az Egyesült Államok területén a "9/4/91". Európa sok országában a "4.9.91" rövidítést használnák. Így a '`%x`' a '`9/4/91`' szöveget produkálja ha az `"US"` van definiálva, míg az `"EUROPE"` esetén az eredmény a '`4.9.91`' lenne. Az ANSI C a `"C"`-t definiálja mint helyi beállítás (locale), amit a legtöbb C programozó használ.

A `gawk` csomag a `strftime` függvény egy "public-domain" C verzióját is tartalmazza olyan rendszerek számára, amelyek nem teljesen ANSI kompatíbilisek. Ha a `gawk` ezzel a verzióval lett lefordítva (lásd B. Függelék [Installing `gawk`], 277. oldal), akkor az alábbi formátumleírókat is lehet használni:

`%D` Ugyanaz mint '`%m/%d/%y`'.

`%e` A hónap napja. Ha csak egy számjegyből áll, akkor egy szóközzel kiegészítve.

`%h` Ugyanaz mint '`%b`', fent.

`%n` Új sor karakter (ASCII LF).

`%r` Ugyanaz mint a '`%I:%M:%S %p`'.

`%R` Ugyanaz mint a '`%H:%M`'.

`%T` Ugyanaz mint a '`%H:%M:%S`'.

`%t` Egy tabulátor karakter.

---

[6] Ez azért van, mert az ANSI C az `strftime` C verziójának a viselkedését is így definiálja és a `gawk` a rendszer által megadott `strftime` függvényt fogja használni, ha létezik. Lényegében vagy megjelenik módosítás nélkül vagy nem fog szerepelni a szövegben.

`%k` Az óra (24 órás beosztásban) mint decimális szám (0-23). Ha csak egy számjegyből áll, akkor egy szóközzel kiegészítve.

`%l` Az óra (12 órás beosztásban) mint decimális szám (1-12). Ha csak egy számjegyből áll, akkor egy szóközzel kiegészítve.

`%C` Az évszázad, mint egy 00 és 99 közti szám.

`%u` A hét napja, mint decimális szám [1 (hétfő)–7].

`%V` A hét száma az évben (az első hétfő, mint az első hét első napja), mint decimális szám (01–53). A hét számának megállapítása a ISO 8601 szabványban definiált. (Ha a január elsejét tartalmazó hétből négy vagy több nap esik az új évre, akkor első hét, egyébként az előző év 53. hete és a következő hét lesz az első.)

`%G` Az év az évszázaddal együtt az ISO hét szám alapján, mint decimális szám.

Például, 1993 január 1, 1992 53. hetében van, így az ISO hét szám szerint az év 1992. Hasonlóan 1973 december 31, 1974 első hetében van, így az ISO hét szám szerint az év 1974, még akkor is ha a nap 1973-ra esik.

`%g` Az év az évszázad nélkül az ISO hét szám alapján, mint decimális szám (00–99).

`%Ec %EC %Ex %Ey %EY %Od %Oe %OH %OI`
`%Om %OM %OS %Ou %OU %OV %Ow %OW %Oy`
Ezek "alternatív reprezentációi" azoknak az eseteknek amelyik csak a második betűt használnák ('`%c`', '`%C`', és így tovább). Ezeket is felismeri, de a normális reprezentációt használja.[7] (Ez a lehetőség a POSIX `date` segédprogrammal kompatíbilis.)

`%v` A dátum VMS formátumban (pl. 20-JUN-1991).

`%z` Az időzóna eltolódás +HHMM formátumban (pl. ez a formátum a RFC-822/RFC-1036 által definiált dátum fejléchez kell).

Ez a példa a POSIX `date` segédprogram `awk` implementációja. Általában a `date` az aktuális dátumot és időt nyomtatja ki egy jól ismert formátumban. Ugyanakkor, ha olyan argumentumot adsz meg ami '`+`' karakterrel kezdődik, akkor a `date` a nem formátumleíró karaktereket közvetlenül kinyomtatja, míg az adott formátumnak megfelelően írja ki az aktuális időt. Például:

```
$ date '+Today is %A, %B %d, %Y.'
⊣ Today is Thursday, July 11, 1991.
```

Itt a `gawk` verziója a `date` segédprogramnak. Egy shell "wrapper" veszi körbe, hogy a '`-u`' opciót is kezelje, ami azt adja meg a `date` programnak, hogy UTC időzónában adja meg az időt.

[7] Ha az egészből semmit nem értesz, ne aggódj, nem olyan fontos, azért kerültek bele a programba, hogy könnyebben lehessen a helyi körülményekhez igazítani a programot.

```
#! /bin/sh
#
# date --- approximate the P1003.2 'date' command

case $1 in
-u)  TZ=GMT0     # use UTC
     export TZ
     shift ;;
esac

gawk 'BEGIN  {
    format = "%a %b %d %H:%M:%S %Z %Y"
    exitval = 0

    if (ARGC > 2)
        exitval = 1
    else if (ARGC == 2) {
        format = ARGV[1]
        if (format ~ /^\+/)
            format = substr(format, 2)   # remove leading +
    }
    print strftime(format)
    exit exitval
}' "$@"
```

13. Felhasználó által definiált függvények

Bonyolult `awk` programokat egyszerűsíteni lehet, ha saját függvényeket definiálsz. Az általad definiált függvényeket ugyanúgy kell meghívni, mint a beépített függvényeket (lásd 7.13. Bekezdés [Függvényhívások], 93. oldal), de neked kell definiálni a függvényeket – megmondani az `awk`-nak, hogy a függvénynek mit kell csinálnia.

13.1. A függvénydefiníció formája

A függvények definíciója bárhol előfordulhat az `awk` program szabályai között. Így egy `awk` program általános formája valójában szabályok *és* függvénydefiníciók sorozata. Nincs szükség arra, hogy a függvények definícióját a használatuk előtt adjuk meg, mivel az `awk` először beolvassa az egész programot, és csak utána kezdi el végrehajtani.

Egy *name* nevű függvény definíciója így néz ki:

```
function name(parameter-list)
{
     body-of-function
}
```

Egy érvényes függvény neve ugyanolyan, mint egy érvényes változó neve: betűk, számok és az aláhúzás karakterek sorozata, amelyek nem kezdődhetnek számmal. Egy `awk` programon belül egy adott név vagy változó, vagy tömb, vagy függvény.

A *parameter-list* a függvény argumentumainak és a lokális változóknak a listája, vesszővel elválasztva az elemeit. Amikor egy függvényt meghívunk, akkor az argumentumban adott nevek fogják a hívás során megadott értékeket tartalmazni. A lokális változók kezdeti értéke az üres szöveg. Egy függvénynek nem lehet két azonos nevű paramétere.

A *body-of-function* `awk` kifejezéseket tartalmaz. Ez a függvény legfontosabb része, mivel ez adja meg, hogy a függvénynek mit is kell *csinálnia*. Az argumentumok egy bizonyos fajta kommunikációt biztosítanak a program többi részével; a lokális változók ideiglenes tárolási helyet biztosítanak.

Az argumentumok és a lokális változók nem különböztethetők meg szintaktikailag; ehelyett a hívás során megadott argumentumok száma határozza meg, hogy mennyi argumentuma lesz a függvénynek. Így, ha három argumentummal hívjuk meg a függvényt, akkor az első három név a *parameter-list* listában argumentum lesz, a többi lokális változó.

Ebből következik, hogy ha nem mindig ugyanannyi argumentummal hívjuk meg a függvényt, akkor egyes nevek a *parameter-list* listában néha argumentumok lesznek, néha lokális változók. Egy másik felfogás szerint, a meg nem adott argumentumok kezdeti értéke az üres szöveg lesz.

Általában amikor a függvényt írod, akkor már tudod, hogy mennyi argumentumot akarsz használni és mennyi lokális változót, így az a szokás, hogy néhány extra szóköz karaktert kell az argumentumok és a lokális változók közé tenni, hogy ezzel dokumentáld a funkcióbeli különbséget.

A függvény futtatása során az argumentumok és lokális változók "eltakarják" a programod más részein használt azonos nevű változókat. Az "eltakart" változók nem elérhetőek a függvényben, mivel a nevük az adott függvényben valami mást jelentenek. Minden más változót, amit az `awk` programban használtál el lehet érni és megváltoztatni a függvényen belül.

Az argumentumok csak a függvény belsejében érvényesek. Ha a függvény befejezte a működését és kilépett, akkor újra el tudod érni a függvény által "eltakart" változókat.

A függvény maga is tartalmazhat olyan kifejezéseket, amelyek másik függvényt hívnak meg. A függvény még saját magát is meghívhatja, közvetlenül vagy indirekt módon. Ekkor a függvényt *rekurzívnak* hívjuk.

A legtöbb `awk` implementációban, a `gawk`-ban is, a `function` kulcsszót rövidíteni lehet a `func` szóval. Ugyanakkor a POSIX szabvány csak a `function` kulcsszót definiálja. Ennek az a praktikus következménye, hogy ha a `gawk` POSIX kompatíbilis módban fut (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor az alábbi kifejezés *nem* definiál egy függvényt:

```
func foo() { a = sqrt($1) ; print a }
```

Ehelyett egy szabályt generál minden rekordra, ami összefűzi a '`func`' változó tartalmát a '`foo`' függvény visszatérési értékével. Ha az eredmény szöveg nem üres, akkor végrehajtja a tevékenységet. Nem valószínű, hogy ezt akartad. (Az `awk` elfogadja ezt a programot, mint egy szintaktikailag helyes program, mivel a függvényt azelőtt lehet használni, mielőtt a függvény definiálva lett.)

Ha hordozható `awk` programot akarsz írni, akkor mindig a `function` kulcsszót használd a függvények definíciójánál.

13.2. Példák a függvénydefinícióra

Itt egy példa a felhasználó által definiálható függvényekre, `myprint`, ami egy számot kap argumentumként, és azt egy adott formában kinyomtatja.

```
function myprint(num)
{
     printf "%6.3g\n", num
}
```

Itt pedig bemutatjuk, hogy egy `awk` szabály hogyan használná a `myprint` függvényt:

```
$3 > 0     { myprint($3) }
```

Ez a program kinyomtatja minden bemeneti rekord harmadik mezőjét az általunk megadott formátumban, ha a mező pozitív számot tartalmaz. Így, ha a bemenet ez:

```
 1.2   3.4    5.6   7.8
 9.10 11.12 -13.14 15.16
17.18 19.20  21.22 23.24
```

akkor a program, az általunk megadott függvénnyel ezt fogja kinyomtatni:

```
  5.6
 21.2
```

Ez a függvény kitörli egy tömb minden elemét.

```
function delarray(a,    i)
{
    for (i in a)
       delete a[i]
}
```

Amikor tömbökkel dolgozunk, gyakran szükség lehet egy tömb minden elemének a kitörlésére, és új elemekkel újrakezdeni (lásd 11.6. Bekezdés [A `delete` kifejezés], 127. oldal). Ahelyett, hogy a törlő ciklust minden alkalommal le kellene írni, elég a fent definiált `delarray` függvényt meghívni.

Itt egy példa a rekurzív függvényre. A függvény egy szöveget vár argumentumként, és fordítva nyomtatja azt ki.

```
function rev(str, start)
{
    if (start == 0)
        return ""

    return (substr(str, start, 1) rev(str, start - 1))
}
```

Ha ez a függvény egy '`rev.awk`' nevű file-ban van, akkor így tesztelhetjük:

```
$ echo "Don't Panic!" |
> gawk --source '{ print rev($0, length($0)) }' -f rev.awk
 ⊣ !cinaP t'noD
```

Itt egy példa, ami az `strftime` beépített függvényt használja (lásd 12.5. Bekezdés [Dátumfeldolgozó függvények], 147. oldal). A `ctime` függvény a C nyelvben egy időbélyeget vár argumentumként, és egy jól ismert formában nyomtatja azt ki. Itt az `awk` verziója:

```
# ctime.awk
#
# awk version of C ctime(3) function

function ctime(ts,    format)
{
    format = "%a %b %d %H:%M:%S %Z %Y"
    if (ts == 0)
        ts = systime()       # use current time as default
    return strftime(format, ts)
}
```

13.3. A felhasználó által definiált függvények hívása

Egy *függvény hívása* azt jelenti, hogy a függvény lefut, és elvégzi a munkáját. A függvényhívás egy kifejezés, és az értéke a függvény visszatérési értéke.

A függvényhívás a függvény nevéből és az utána álló, zárójelek közti argumentumok listájából áll. Amit argumentumként megadsz, azok is `awk` kifejezések; minden függvényhívásnál az argumentumként adott kifejezések kiértékelődnek, és az értékük lesz az aktuális argumentum. Például itt a `foo` függvényt hívjuk meg három argumentummal (ahol az első egy szövegösszefűzés):

```
foo(x y, "lose", 4 * z)
```

Figyelem: szóköz és tabulátor karakterek nem megengedettek a függvény neve és a nyitó zárójel között. Ha mégis lenne szóköz vagy tabulátor karakter a függvény neve és a nyitó zárójel között, akkor az `awk` ezt úgy is értelmezheti, mintha összefűzést szeretnél csinálni a függvény nevével és a zárójelek között megadott kifejezéssel. Ugyanakkor észreveszi, hogy függvény nevet adtál meg, nem változót, és így hibát fog jelezni.

Amikor a függvény meghívódik, akkor az argumentumként megadott értékek egy *másolatát* kapja meg. Ezt a módszert úgy is hívják, hogy *hívás érték alapján*. A függvényt hívó kifejezés akár változót is használhat argumentumként, de a hívott függvény ezt nem tudja: csak annyit tud a meghívott függvény, hogy az argumentum értéke micsoda. Például, ha ezt a kódot írod:

```
foo = "bar"
z = myfunc(foo)
```

akkor nem szabad azt gondolnod, hogy a `myfunc` függvény argumentuma a “`foo` változó”, hanem hogy az argumentum a `"bar"` szöveg.

Ha a `myfunc` függvény megváltoztatja a lokális változók értékét, akkor az semmilyen hatással nincs más változókra. Így ha a `myfunc` függvény ezt csinálja:

```
function myfunc(str)
{
  print str
  str = "zzz"
  print str
}
```

akkor bár az első argumentumát, `str`, megváltoztatja, de a `foo` értéke nem fog megváltozni a hívó kifejezésben. A `foo` szerepe a `myfunc` függvény hívása során akkor ér véget, amikor az értékét, `"bar"`, kiszámolta. Ha az `str` a `myfunc` függvényen kívül is létezik, akkor a függvény ennek a külső változónak az értékét nem tudja megváltoztatni, mivel a `myfunc` függvény futtatása ezt a változót "eltakarja".

Ugyanakkor ha a függvény argumentuma egy tömb, akkor az *nem* másolódik. Ehelyett, a függvény magát az eredeti tömböt tudja megváltoztatni. Ezt úgy nevezik, hogy *hívás referencia alapján*. A függvény belsejében végrehajtott változások egy tömbön, a függvényen *kívül is* érvényben lesznek. *Ez nagyon veszélyes lehet, ha nem figyelsz oda, hogy mit csinálsz.* Például:

```
function changeit(array, ind, nvalue)
{
     array[ind] = nvalue
}

BEGIN {
    a[1] = 1; a[2] = 2; a[3] = 3
    changeit(a, 2, "two")
    printf "a[1] = %s, a[2] = %s, a[3] = %s\n",
            a[1], a[2], a[3]
}
```

Ennek a programnak az eredménye: '`a[1] = 1, a[2] = two, a[3] = 3`', mivel a `changeit` függvény a `"two"` szöveget tárolja el az `a` tömb második elemében.

Néhány `awk` implementáció megengedi nem definiált függvények hívását, és csak a futtatás során jeleznek hibát, amikor a függvényt megpróbálják lefuttatni. Például:

```
BEGIN {
    if (0)
        foo()
    else
        bar()
}
function bar() { ... }
# figyelem: 'foo' nem definiált
```

Mivel az '`if`' kifejezés soha nem igaz, ezért nem igazán probléma, hogy a `foo` függvény nem lett definiálva. Általában azért ez probléma, ha a program olyan függvényt akar meghívni, ami nem lett definiálva.

Ha a '`--lint`' opció szerepel a parancssorban (lásd 14.1. Bekezdés [Command Line Options], 161. oldal), akkor a `gawk` jelenteni fogja a nem definiált függvényeket.

Néhány `awk` implementáció hibát generál, ha a `next` kifejezést (lásd 9.7. Bekezdés [A `next` kifejezés], 111. oldal) használod egy általad definiált függvényben. A `gawk`-nak nincs ilyen problémája.

13.4. A `return` kifejezés

A felhasználó által definiált függvények tartalmazhatják a `return` kifejezést. Ennek hatására a függvényből kilép, és folytatja az `awk` program futtatását a függvény hívása után. Visszatérési érték is megadható. Így néz ki:

`return` [*expression*]

Az *expression* rész opcionális. Ha nincs megadva, akkor a visszatérési érték nem definiált, és ezért nem megjósolható.

Minden függvénydefiníció végén egy egyszerű, argumentumok nélküli `return` kifejezést lehet feltételezni. Így ha a program eléri a függvény végét, akkor a függvény egy megjósolhatatlan értékkel visszatér. Az `awk` *nem* fog figyelmeztetni, ha egy ilyen függvény visszatérési értékét használod.

Néha azért írsz egy függvényt, amit csinál, és nem azért amit visszaad. Ezek a C nyelvben a `void` függvények, a Pascal-ban a `procedure`-k. Ezért hasznos, hogy nem adsz meg semmilyen visszatérési értéket; egyszerűen csak tartsd fejben, hogy ha egy ilyen függvény visszatérési értékét használod, akkor azt a saját felelősségedre teszed.

Itt egy példa, amelyik egy tömbben található legnagyobb számmal tér vissza:

```
function maxelt(vec,   i, ret)
{
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}
```

A `maxelt` függvényt egy argumentummal kell meghívni, ami a tömb neve. Az `i` és a `ret` lokális változók, és nem argumentumok; bár semmi nincs ami meggátoljon abban, hogy egy második és egy harmadik argumentumot is megadj a függvénynek, de ekkor az eredmény furcsa is lehet. Az extra szóközök az `i` előtt jelzik, hogy ezután a lokális változókat definiálod. Ezt a konvenciót érdemes követned.

Itt egy program, amelyik a `maxelt` függvényt használja. A program betölt egy tömböt, meghívja a `maxelt` függvényt, majd kinyomtatja a tömb legnagyob elemét:

```
awk '
function maxelt(vec,   i, ret)
{
     for (i in vec) {
          if (ret == "" || vec[i] > ret)
               ret = vec[i]
     }
     return ret
}

# Load all fields of each record into nums.
{
     for(i = 1; i <= NF; i++)
          nums[NR, i] = $i
}

END {
     print maxelt(nums)
}'
```

Az alábbi bemenetre:

```
 1 5 23 8 16
44 3 5 2 8 26
256 291 1396 2962 100
-6 467 998 1101
99385 11 0 225
```

a program azt fogja mondani (megjósolható módon), hogy `99385` a legnagyobb szám a tömbben.

14. Running awk

There are two ways to run awk: with an explicit program, or with one or more program files. Here are templates for both of them; items enclosed in '[...]' in these templates are optional.

Besides traditional one-letter POSIX-style options, gawk also supports GNU long options.

awk [*options*] -f progfile [--] *file* ...
awk [*options*] [--] '*program*' *file* ...

It is possible to invoke awk with an empty program:

```
$ awk '' datafile1 datafile2
```

Doing so makes little sense though; awk will simply exit silently when given an empty program (d.c.). If '--lint' has been specified on the command line, gawk will issue a warning that the program is empty.

14.1. Command Line Options

Options begin with a dash, and consist of a single character. GNU style long options consist of two dashes and a keyword. The keyword can be abbreviated, as long the abbreviation allows the option to be uniquely identified. If the option takes an argument, then the keyword is either immediately followed by an equals sign ('=') and the argument's value, or the keyword and the argument's value are separated by whitespace. For brevity, the discussion below only refers to the traditional short options; however the long and short options are interchangeable in all contexts.

Each long option for gawk has a corresponding POSIX-style option. The options and their meanings are as follows:

-F *fs*
--field-separator *fs*
: Sets the FS variable to *fs* (lásd 5.5. Bekezdés [Specifying How Fields are Separated], 44. oldal).

-f *source-file*
--file *source-file*
: Indicates that the awk program is to be found in *source-file* instead of in the first non-option argument.

-v *var*=*val*
--assign *var*=*val*
: Sets the variable *var* to the value *val* **before** execution of the program begins. Such variable values are available inside the BEGIN rule (lásd 14.2. Bekezdés [Other Command Line Arguments], 165. oldal).

The '-v' option can only set one variable, but you can use it more than once, setting another variable each time, like this: 'awk -v foo=1 -v bar=2 ...'.

-mf *NNN*
-mr *NNN* Set various memory limits to the value *NNN*. The 'f' flag sets the maximum number of fields, and the 'r' flag sets the maximum record size. These two flags and the '-m' option are from the Bell Labs research version of Unix awk. They are provided for compatibility, but otherwise ignored by gawk, since gawk has no predefined limits.

-W *gawk-opt*
Following the POSIX standard, options that are implementation specific are supplied as arguments to the '-W' option. These options also have corresponding GNU style long options. See below.

-- Signals the end of the command line options. The following arguments are not treated as options even if they begin with '-'. This interpretation of '--' follows the POSIX argument parsing conventions.

This is useful if you have file names that start with '-', or in shell scripts, if you have file names that will be specified by the user which could start with '-'.

The following gawk-specific options are available:

-W traditional
-W compat
--traditional
--compat Specifies *compatibility mode*, in which the GNU extensions to the awk language are disabled, so that gawk behaves just like the Bell Labs research version of Unix awk. '--traditional' is the preferred form of this option. Lásd 17.5. Bekezdés [Extensions in gawk Not in POSIX awk], 253. oldal, which summarizes the extensions. Also see C.1. Bekezdés [Downward Compatibility and Debugging], 293. oldal.

-W copyleft
-W copyright
--copyleft
--copyright
Print the short version of the General Public License, and then exit. This option may disappear in a future version of gawk.

`-W help`
`-W usage`
`--help`
`--usage` Print a "usage" message summarizing the short and long style options that `gawk` accepts, and then exit.

`-W lint`
`--lint` Warn about constructs that are dubious or non-portable to other `awk` implementations. Some warnings are issued when `gawk` first reads your program. Others are issued at run-time, as your program executes.

`-W lint-old`
`--lint-old`
Warn about constructs that are not available in the original Version 7 Unix version of `awk` (lásd 17.1. Bekezdés [Major Changes between V7 and SVR3.1], 251. oldal).

`-W posix`
`--posix` Operate in strict POSIX mode. This disables all `gawk` extensions (just like '`--traditional`'), and adds the following additional restrictions:

- `\x` escape sequences are not recognized (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal).
- Newlines do not act as whitespace to separate fields when `FS` is equal to a single space.
- The synonym `func` for the keyword `function` is not recognized (lásd 13.1. Bekezdés [Function Definition Syntax], 153. oldal).
- The operators '`**`' and '`**=`' cannot be used in place of '`^`' and '`^=`' (lásd 7.5. Bekezdés [Arithmetic Operators], 82. oldal, and also lásd 7.7. Bekezdés [Assignment Expressions], 84. oldal).
- Specifying '`-Ft`' on the command line does not set the value of `FS` to be a single tab character (lásd 5.5. Bekezdés [Specifying How Fields are Separated], 44. oldal).
- The `fflush` built-in function is not supported (lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal).

If you supply both '`--traditional`' and '`--posix`' on the command line, '`--posix`' will take precedence. `gawk` will also issue a warning if both options are supplied.

`-W re-interval`
`--re-interval`
Allow interval expressions (lásd 4.3. Bekezdés [Regular Expression Operators], 26. oldal), in regexps. Because interval exp-

ressions were traditionally not available in `awk`, `gawk` does not provide them by default. This prevents old `awk` programs from breaking.

`-W source` *program-text*
`--source` *program-text*
: Program source code is taken from the *program-text*. This option allows you to mix source code in files with source code that you enter on the command line. This is particularly useful when you have library functions that you wish to use from your command line programs (lásd 14.3. Bekezdés [The `AWKPATH` Environment Variable], 166. oldal).

`-W version`
`--version`
: Prints version information for this particular copy of `gawk`. This allows you to determine if your copy of `gawk` is up to date with respect to whatever the Free Software Foundation is currently distributing. It is also useful for bug reports (lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal).

Any other options are flagged as invalid with a warning message, but are otherwise ignored.

In compatibility mode, as a special case, if the value of *fs* supplied to the '`-F`' option is '`t`', then `FS` is set to the tab character (`"\t"`). This is only true for '`--traditional`', and not for '`--posix`' (lásd 5.5. Bekezdés [Specifying How Fields are Separated], 44. oldal).

The '`-f`' option may be used more than once on the command line. If it is, `awk` reads its program source from all of the named files, as if they had been concatenated together into one big file. This is useful for creating libraries of `awk` functions. Useful functions can be written once, and then retrieved from a standard place, instead of having to be included into each individual program.

You can type in a program at the terminal and still use library functions, by specifying '`-f /dev/tty`'. `awk` will read a file from the terminal to use as part of the `awk` program. After typing your program, type *`Control-d`* (the end-of-file character) to terminate it. (You may also use '`-f -`' to read program source from the standard input, but then you will not be able to also use the standard input as a source of data.)

Because it is clumsy using the standard `awk` mechanisms to mix source file and command line `awk` programs, `gawk` provides the '`--source`' option. This does not require you to pre-empt the standard input for your source code, and allows you to easily mix command line and library source code (lásd 14.3. Bekezdés [The `AWKPATH` Environment Variable], 166. oldal).

If no '`-f`' or '`--source`' option is specified, then `gawk` will use the first non-option command line argument as the text of the program source code.

If the environment variable `POSIXLY_CORRECT` exists, then `gawk` will behave in strict POSIX mode, exactly as if you had supplied the '`--posix`' command line option. Many GNU programs look for this environment variable to turn on strict POSIX mode. If you supply '`--lint`' on the command line, and `gawk` turns on POSIX mode because of `POSIXLY_CORRECT`, then it will print a warning message indicating that POSIX mode is in effect.

You would typically set this variable in your shell's startup file. For a Bourne compatible shell (such as Bash), you would add these lines to the '`.profile`' file in your home directory.

```
POSIXLY_CORRECT=true
export POSIXLY_CORRECT
```

For a `csh` compatible shell,[1] you would add this line to the '`.login`' file in your home directory.

```
setenv POSIXLY_CORRECT true
```

14.2. Other Command Line Arguments

Any additional arguments on the command line are normally treated as input files to be processed in the order specified. However, an argument that has the form *var*`=`*value*, assigns the value *value* to the variable *var*—it does not specify a file at all.

All these arguments are made available to your `awk` program in the `ARGV` array (lásd 10. Fejezet [Built-in Variables], 115. oldal). Command line options and the program text (if present) are omitted from `ARGV`. All other arguments, including variable assignments, are included. As each element of `ARGV` is processed, `gawk` sets the variable `ARGIND` to the index in `ARGV` of the current element.

The distinction between file name arguments and variable-assignment arguments is made when `awk` is about to open the next input file. At that point in execution, it checks the "file name" to see whether it is really a variable assignment; if so, `awk` sets the variable instead of reading a file.

Therefore, the variables actually receive the given values after all previously specified files have been read. In particular, the values of variables assigned in this fashion are *not* available inside a `BEGIN` rule (lásd 8.1.5. Bekezdés [The `BEGIN` and `END` Special Patterns], 100. oldal), since such rules are run before `awk` begins scanning the argument list.

The variable values given on the command line are processed for escape sequences (d.c.) (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal).

In some earlier implementations of `awk`, when a variable assignment occurred before any file names, the assignment would happen *before* the `BEGIN` rule was executed. `awk`'s behavior was thus inconsistent; some command line assignments were available inside the `BEGIN` rule, while others were not.

[1] Not recommended.

However, some applications came to depend upon this "feature." When `awk` was changed to be more consistent, the '`-v`' option was added to accommodate applications that depended upon the old behavior.

The variable assignment feature is most useful for assigning to variables such as `RS`, `OFS`, and `ORS`, which control input and output formats, before scanning the data files. It is also useful for controlling state if multiple passes are needed over a data file. For example:

```
awk 'pass == 1  { pass 1 stuff }
     pass == 2  { pass 2 stuff }' pass=1 mydata pass=2 mydata
```

Given the variable assignment feature, the '`-F`' option for setting the value of `FS` is not strictly necessary. It remains for historical compatibility.

14.3. The `AWKPATH` Environment Variable

The previous section described how `awk` program files can be named on the command line with the '`-f`' option. In most `awk` implementations, you must supply a precise path name for each program file, unless the file is in the current directory.

But in `gawk`, if the file name supplied to the '`-f`' option does not contain a '`/`', then `gawk` searches a list of directories (called the *search path*), one by one, looking for a file with the specified name.

The search path is a string consisting of directory names separated by colons. `gawk` gets its search path from the `AWKPATH` environment variable. If that variable does not exist, `gawk` uses a default path, which is '`.:/usr/local/share/awk`'.[2] (Programs written for use by system administrators should use an `AWKPATH` variable that does not include the current directory, '`.`'.)

The search path feature is particularly useful for building up libraries of useful `awk` functions. The library files can be placed in a standard directory that is in the default path, and then specified on the command line with a short file name. Otherwise, the full file name would have to be typed for each file.

By using both the '`--source`' and '`-f`' options, your command line `awk` programs can use facilities in `awk` library files. Lásd 15. Fejezet [A Library of `awk` Functions], 169. oldal.

Path searching is not done if `gawk` is in compatibility mode. This is true for both '`--traditional`' and '`--posix`'. Lásd 14.1. Bekezdés [Command Line Options], 161. oldal.

Note: if you want files in the current directory to be found, you must include the current directory in the path, either by including '`.`' explicitly in

[2] Your version of `gawk` may use a directory that is different than '`/usr/local/share/awk`'; it will depend upon how `gawk` was built and installed. The actual directory will be the value of '`$(datadir)`' generated when `gawk` was configured. You probably don't need to worry about this though.

the path, or by writing a null entry in the path. (A null entry is indicated by starting or ending the path with a colon, or by placing two colons next to each other ('::').) If the current directory is not included in the path, then files cannot be found in the current directory. This path search mechanism is identical to the shell's.

Starting with version 3.0, if `AWKPATH` is not defined in the environment, `gawk` will place its default search path into `ENVIRON["AWKPATH"]`. This makes it easy to determine the actual search path `gawk` will use.

14.4. Obsolete Options and/or Features

This section describes features and/or command line options from previous releases of `gawk` that are either not available in the current version, or that are still supported but deprecated (meaning that they will *not* be in the next release).

For version 3.0.4 of `gawk`, there are no command line options or other deprecated features from the previous version of `gawk`. This section is thus essentially a place holder, in case some option becomes obsolete in a future version of `gawk`.

14.5. Undocumented Options and Features

Use the Source, Luke!
Obi-Wan

This section intentionally left blank.

14.6. Known Bugs in gawk

- The '`-F`' option for changing the value of `FS` (lásd 14.1. Bekezdés [Command Line Options], 161. oldal) is not necessary given the command line variable assignment feature; it remains only for backwards compatibility.
- If your system actually has support for '`/dev/fd`' and the associated '`/dev/stdin`', '`/dev/stdout`', and '`/dev/stderr`' files, you may get different output from `gawk` than you would get on a system without those files. When `gawk` interprets these files internally, it synchronizes output to the standard output with output to '`/dev/stdout`', while on a system with those files, the output is actually to different open files (lásd 6.7. Bekezdés [Special File Names in gawk], 71. oldal).
- Syntactically invalid single character programs tend to overflow the parse stack, generating a rather unhelpful message. Such programs are surprisingly difficult to diagnose in the completely general case, and the effort to do so really is not worth it.

15. A Library of awk Functions

This chapter presents a library of useful awk functions. The sample programs presented later (lásd 16. Fejezet [Practical awk Programs], 205. oldal) use these functions. The functions are presented here in a progression from simple to complex.

16.2.7. Bekezdés [Extracting Programs from Texinfo Source Files], 238. oldal, presents a program that you can use to extract the source code for these example library functions and programs from the Texinfo source for this könyv. (This has already been done as part of the gawk distribution.)

If you have written one or more useful, general purpose awk functions, and would like to contribute them for a subsequent edition of this könyv, please contact the author. Lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal, for information on doing this. Don't just send code, as you will be required to either place your code in the public domain, publish it under the GPL (lásd [GNU GENERAL PUBLIC LICENSE], 307. oldal), or assign the copyright in it to the Free Software Foundation.

15.1. Simulating gawk-specific Features

The programs in this chapter and in 16. Fejezet [Practical awk Programs], 205. oldal, freely use features that are specific to gawk. This section briefly discusses how you can rewrite these programs for different implementations of awk.

Diagnostic error messages are sent to '/dev/stderr'. Use '| "cat 1>&2"' instead of '> "/dev/stderr"', if your system does not have a '/dev/stderr', or if you cannot use gawk.

A number of programs use nextfile (lásd 9.8. Bekezdés [The nextfile Statement], 112. oldal), to skip any remaining input in the input file. 15.2. Bekezdés [Implementing nextfile as a Function], 170. oldal, shows you how to write a function that will do the same thing.

Finally, some of the programs choose to ignore upper-case and lower-case distinctions in their input. They do this by assigning one to IGNORECASE. You can achieve the same effect by adding the following rule to the beginning of the program:

```
# ignore case
{ $0 = tolower($0) }
```

Also, verify that all regexp and string constants used in comparisons only use lower-case letters.

15.2. Implementing `nextfile` as a Function

The `nextfile` statement presented in 9.8. Bekezdés [The `nextfile` Statement], 112. oldal, is a `gawk`-specific extension. It is not available in other implementations of `awk`. This section shows two versions of a `nextfile` function that you can use to simulate `gawk`'s `nextfile` statement if you cannot use `gawk`.

Here is a first attempt at writing a `nextfile` function.

```
# nextfile --- skip remaining records in current file

# this should be read in before the "main" awk program

function nextfile()    { _abandon_ = FILENAME; next }

_abandon_ == FILENAME  { next }
```

This file should be included before the main program, because it supplies a rule that must be executed first. This rule compares the current data file's name (which is always in the `FILENAME` variable) to a private variable named `_abandon_`. If the file name matches, then the action part of the rule executes a `next` statement, to go on to the next record. (The use of '`_`' in the variable name is a convention. It is discussed more fully in 15.13. Bekezdés [Naming Library Function Global Variables], 202. oldal.)

The use of the `next` statement effectively creates a loop that reads all the records from the current data file. Eventually, the end of the file is reached, and a new data file is opened, changing the value of `FILENAME`. Once this happens, the comparison of `_abandon_` to `FILENAME` fails, and execution continues with the first rule of the "real" program.

The `nextfile` function itself simply sets the value of `_abandon_` and then executes a `next` statement to start the loop going.[1]

This initial version has a subtle problem. What happens if the same data file is listed *twice* on the command line, one right after the other, or even with just a variable assignment between the two occurrences of the file name?

In such a case, this code will skip right through the file, a second time, even though it should stop when it gets to the end of the first occurrence. Here is a second version of `nextfile` that remedies this problem.

[1] Some implementations of `awk` do not allow you to execute `next` from within a function body. Some other work-around will be necessary if you use such a version.

```
# nextfile --- skip remaining records in current file
# correctly handle successive occurrences of the same file
# Arnold Robbins, arnold@gnu.org, Public Domain
# May, 1993

# this should be read in before the "main" awk program

function nextfile()   { _abandon_ = FILENAME; next }

_abandon_ == FILENAME {
      if (FNR == 1)
          _abandon_ = ""
      else
          next
}
```

The `nextfile` function has not changed. It sets `_abandon_` equal to the current file name and then executes a `next` satement. The `next` statement reads the next record and increments `FNR`, so `FNR` is guaranteed to have a value of at least two. However, if `nextfile` is called for the last record in the file, then `awk` will close the current data file and move on to the next one. Upon doing so, `FILENAME` will be set to the name of the new file, and `FNR` will be reset to one. If this next file is the same as the previous one, `_abandon_` will still be equal to `FILENAME`. However, `FNR` will be equal to one, telling us that this is a new occurrence of the file, and not the one we were reading when the `nextfile` function was executed. In that case, `_abandon_` is reset to the empty string, so that further executions of this rule will fail (until the next time that `nextfile` is called).

If `FNR` is not one, then we are still in the original data file, and the program executes a `next` statement to skip through it.

An important question to ask at this point is: "Given that the functionality of `nextfile` can be provided with a library file, why is it built into `gawk`?" This is an important question. Adding features for little reason leads to larger, slower programs that are harder to maintain.

The answer is that building `nextfile` into `gawk` provides significant gains in efficiency. If the `nextfile` function is executed at the beginning of a large data file, `awk` still has to scan the entire file, splitting it up into records, just to skip over it. The built-in `nextfile` can simply close the file immediately and proceed to the next one, saving a lot of time. This is particularly important in `awk`, since `awk` programs are generally I/O bound (i.e. they spend most of their time doing input and output, instead of performing computations).

15.3. Assertions

When writing large programs, it is often useful to be able to know that a condition or set of conditions is true. Before proceeding with a particular computation, you make a statement about what you believe to be the case. Such a statement is known as an "assertion." The C language provides an `<assert.h>` header file and corresponding `assert` macro that the programmer can use to make assertions. If an assertion fails, the `assert` macro arranges to print a diagnostic message describing the condition that should have been true but was not, and then it kills the program. In C, using `assert` looks this:

```
#include <assert.h>

int myfunc(int a, double b)
{
     assert(a <= 5 && b >= 17);
     ...
}
```

If the assertion failed, the program would print a message similar to this:

```
prog.c:5: assertion failed: a <= 5 && b >= 17
```

The ANSI C language makes it possible to turn the condition into a string for use in printing the diagnostic message. This is not possible in `awk`, so this `assert` function also requires a string version of the condition that is being tested.

```
# assert --- assert that a condition is true. Otherwise exit.
# Arnold Robbins, arnold@gnu.org, Public Domain
# May, 1993

function assert(condition, string)
{
    if (! condition) {
        printf("%s:%d: assertion failed: %s\n",
            FILENAME, FNR, string) > "/dev/stderr"
        _assert_exit = 1
        exit 1
    }
}

END {
    if (_assert_exit)
        exit 1
}
```

The `assert` function tests the `condition` parameter. If it is false, it prints a message to standard error, using the `string` parameter to describe the failed condition. It then sets the variable `_assert_exit` to one, and executes the `exit` statement. The `exit` statement jumps to the `END` rule. If the `END` rules finds `_assert_exit` to be true, then it exits immediately.

The purpose of the `END` rule with its test is to keep any other `END` rules from running. When an assertion fails, the program should exit immediately. If no assertions fail, then `_assert_exit` will still be false when the `END` rule is run normally, and the rest of the program's `END` rules will execute. For all of this to work correctly, '`assert.awk`' must be the first source file read by `awk`.

You would use this function in your programs this way:

```
function myfunc(a, b)
{
     assert(a <= 5 && b >= 17, "a <= 5 && b >= 17")
     ...
}
```

If the assertion failed, you would see a message like this:

```
mydata:1357: assertion failed: a <= 5 && b >= 17
```

There is a problem with this version of `assert`, that it may not be possible to work around with standard `awk`. An `END` rule is automatically added to the program calling `assert`. Normally, if a program consists of just a `BEGIN` rule, the input files and/or standard input are not read. However, now that the program has an `END` rule, `awk` will attempt to read the input data files, or standard input (lásd 8.1.5.1. Bekezdés [Startup and Cleanup Actions], 100. oldal), most likely causing the program to hang, waiting for input.

15.4. Rounding Numbers

The way `printf` and `sprintf` (lásd 6.5. Bekezdés [Using `printf` Statements for Fancier Printing], 64. oldal) do rounding will often depend upon the system's C `sprintf` subroutine. On many machines, `sprintf` rounding is "unbiased," which means it doesn't always round a trailing '`.5`' up, contrary to naive expectations. In unbiased rounding, '`.5`' rounds to even, rather than always up, so 1.5 rounds to 2 but 4.5 rounds to 4. The result is that if you are using a format that does rounding (e.g., `"%.0f"`) you should check what your system does. The following function does traditional rounding; it might be useful if your awk's `printf` does unbiased rounding.

```
# round --- do normal rounding
#
# Arnold Robbins, arnold@gnu.org, August, 1996
# Public Domain

function round(x,   ival, aval, fraction)
{
   ival = int(x)    # integer part, int() truncates
```

```
    # see if fractional part
    if (ival == x)   # no fraction
       return x

    if (x < 0) {
       aval = -x     # absolute value
       ival = int(aval)
       fraction = aval - ival
       if (fraction >= .5)
          return int(x) - 1   # -2.5 --> -3
       else
          return int(x)       # -2.3 --> -2
    } else {
       fraction = x - ival
       if (fraction >= .5)
          return ival + 1
       else
          return ival
    }
}

# test harness
{ print $0, round($0) }
```

15.5. Translating Between Characters and Numbers

One commercial implementation of `awk` supplies a built-in function, `ord`, which takes a character and returns the numeric value for that character in the machine's character set. If the string passed to `ord` has more than one character, only the first one is used.

The inverse of this function is `chr` (from the function of the same name in Pascal), which takes a number and returns the corresponding character.

Both functions can be written very nicely in `awk`; there is no real reason to build them into the `awk` interpreter.

```
# ord.awk --- do ord and chr
#
# Global identifiers:
#    _ord_:        numerical values indexed by characters
#    _ord_init:    function to initialize _ord_
#
# Arnold Robbins
# arnold@gnu.org
# Public Domain
# 16 January, 1992
# 20 July, 1992, revised

BEGIN    { _ord_init() }

function _ord_init(    low, high, i, t)
{
    low = sprintf("%c", 7) # BEL is ascii 7
    if (low == "\a") {    # regular ascii
        low = 0
        high = 127
    } else if (sprintf("%c", 128 + 7) == "\a") {
        # ascii, mark parity
        low = 128
        high = 255
    } else {        # ebcdic(!)
        low = 0
        high = 255
    }

    for (i = low; i <= high; i++) {
        t = sprintf("%c", i)
        _ord_[t] = i
    }
}
```

Some explanation of the numbers used by `chr` is worthwhile. The most prominent character set in use today is ASCII. Although an eight-bit byte can hold 256 distinct values (from zero to 255), ASCII only defines characters that use the values from zero to 127.[2] At least one computer manufacturer that we know of uses ASCII, but with mark parity, meaning that the leftmost bit in the byte is always one. What this means is that on those systems, characters have numeric values from 128 to 255. Finally, large mainframe

[2] ASCII has been extended in many countries to use the values from 128 to 255 for country-specific characters. If your system uses these extensions, you can simplify `_ord_init` to simply loop from zero to 255.

systems use the EBCDIC character set, which uses all 256 values. While there are other character sets in use on some older systems, they are not really worth worrying about.

```
function ord(str,    c)
{
    # only first character is of interest
    c = substr(str, 1, 1)
    return _ord_[c]
}

function chr(c)
{
    # force c to be numeric by adding 0
    return sprintf("%c", c + 0)
}

#### test code ####
# BEGIN    \
# {
#    for (;;) {
#        printf("enter a character: ")
#        if (getline var <= 0)
#            break
#        printf("ord(%s) = %d\n", var, ord(var))
#    }
# }
```

An obvious improvement to these functions would be to move the code for the `_ord_init` function into the body of the `BEGIN` rule. It was written this way initially for ease of development.

There is a "test program" in a `BEGIN` rule, for testing the function. It is commented out for production use.

15.6. Merging an Array Into a String

When doing string processing, it is often useful to be able to join all the strings in an array into one long string. The following function, `join`, accomplishes this task. It is used later in several of the application programs (lásd 16. Fejezet [Practical `awk` Programs], 205. oldal).

Good function design is important; this function needs to be general, but it should also have a reasonable default behavior. It is called with an array and the beginning and ending indices of the elements in the array to be merged. This assumes that the array indices are numeric—a reasonable assumption since the array was likely created with `split` (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).

```
# join.awk --- join an array into a string
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

function join(array, start, end, sep,    result, i)
{
    if (sep == "")
       sep = " "
    else if (sep == SUBSEP) # magic value
       sep = ""
    result = array[start]
    for (i = start + 1; i <= end; i++)
        result = result sep array[i]
    return result
}
```

An optional additional argument is the separator to use when joining the strings back together. If the caller supplies a non-empty value, `join` uses it. If it is not supplied, it will have a null value. In this case, `join` uses a single blank as a default separator for the strings. If the value is equal to `SUBSEP`, then `join` joins the strings with no separator between them. `SUBSEP` serves as a "magic" value to indicate that there should be no separation between the component strings.

It would be nice if `awk` had an assignment operator for concatenation. The lack of an explicit operator for concatenation makes string operations more difficult than they really need to be.

15.7. Turning Dates Into Timestamps

The `systime` function built in to `gawk` returns the current time of day as a timestamp in "seconds since the Epoch." This timestamp can be converted into a printable date of almost infinitely variable format using the built-in `strftime` function. (For more information on `systime` and `strftime`, lásd 12.5. Bekezdés [Functions for Dealing with Time Stamps], 147. oldal.)

An interesting but difficult problem is to convert a readable representation of a date back into a timestamp. The ANSI C library provides a `mktime` function that does the basic job, converting a canonical representation of a date into a timestamp.

It would appear at first glance that `gawk` would have to supply a `mktime` built-in function that was simply a "hook" to the C language version. In fact though, `mktime` can be implemented entirely in `awk`.

Here is a version of `mktime` for `awk`. It takes a simple representation of the date and time, and converts it into a timestamp.

The code is presented here intermixed with explanatory prose. In 16.2.7. Bekezdés [Extracting Programs from Texinfo Source Files], 238. oldal, you will see how the Texinfo source file for this könyv can be processed to extract the code into a single source file.

The program begins with a descriptive comment and a `BEGIN` rule that initializes a table `_tm_months`. This table is a two-dimensional array that has the lengths of the months. The first index is zero for regular years, and one for leap years. The values are the same for all the months in both kinds of years, except for February; thus the use of multiple assignment.

```
# mktime.awk --- convert a canonical date representation
#                into a timestamp
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

BEGIN    \
{
    # Initialize table of month lengths
    _tm_months[0,1] = _tm_months[1,1] = 31
    _tm_months[0,2] = 28; _tm_months[1,2] = 29
    _tm_months[0,3] = _tm_months[1,3] = 31
    _tm_months[0,4] = _tm_months[1,4] = 30
    _tm_months[0,5] = _tm_months[1,5] = 31
    _tm_months[0,6] = _tm_months[1,6] = 30
    _tm_months[0,7] = _tm_months[1,7] = 31
    _tm_months[0,8] = _tm_months[1,8] = 31
    _tm_months[0,9] = _tm_months[1,9] = 30
    _tm_months[0,10] = _tm_months[1,10] = 31
    _tm_months[0,11] = _tm_months[1,11] = 30
    _tm_months[0,12] = _tm_months[1,12] = 31
}
```

The benefit of merging multiple `BEGIN` rules (lásd 8.1.5. Bekezdés [The `BEGIN` and `END` Special Patterns], 100. oldal) is particularly clear when writing library files. Functions in library files can cleanly initialize their own private data and also provide clean-up actions in private `END` rules.

The next function is a simple one that computes whether a given year is or is not a leap year. If a year is evenly divisible by four, but not evenly divisible by 100, or if it is evenly divisible by 400, then it is a leap year. Thus, 1904 was a leap year, 1900 was not, but 2000 will be.

```
# decide if a year is a leap year
function _tm_isleap(year,    ret)
{
    ret = (year % 4 == 0 && year % 100 != 0) ||
            (year % 400 == 0)

    return ret
}
```

This function is only used a few times in this file, and its computation could have been written *in-line* (at the point where it's used). Making it a separate function made the original development easier, and also avoids the possibility of typing errors when duplicating the code in multiple places.

The next function is more interesting. It does most of the work of generating a timestamp, which is converting a date and time into some number of seconds since the Epoch. The caller passes an array (rather imaginatively named `a`) containing six values: the year including century, the month as a number between one and 12, the day of the month, the hour as a number between zero and 23, the minute in the hour, and the seconds within the minute.

The function uses several local variables to precompute the number of seconds in an hour, seconds in a day, and seconds in a year. Often, similar C code simply writes out the expression in-line, expecting the compiler to do *constant folding*. E.g., most C compilers would turn '`60 * 60`' into '`3600`' at compile time, instead of recomputing it every time at run time. Precomputing these values makes the function more efficient.

```
# convert a date into seconds
function _tm_addup(a,    total, yearsecs, daysecs,
                         hoursecs, i, j)
{
    hoursecs = 60 * 60
    daysecs = 24 * hoursecs
    yearsecs = 365 * daysecs

    total = (a[1] - 1970) * yearsecs

    # extra day for leap years
    for (i = 1970; i < a[1]; i++)
        if (_tm_isleap(i))
            total += daysecs

    j = _tm_isleap(a[1])
    for (i = 1; i < a[2]; i++)
        total += _tm_months[j, i] * daysecs

    total += (a[3] - 1) * daysecs
    total += a[4] * hoursecs
    total += a[5] * 60
    total += a[6]

    return total
}
```

The function starts with a first approximation of all the seconds between Midnight, January 1, 1970,[3] and the beginning of the current year. It then goes through all those years, and for every leap year, adds an additional day's worth of seconds.

The variable `j` holds either one or zero, if the current year is or is not a leap year. For every month in the current year prior to the current month, it adds the number of seconds in the month, using the appropriate entry in the `_tm_months` array.

Finally, it adds in the seconds for the number of days prior to the current day, and the number of hours, minutes, and seconds in the current day.

The result is a count of seconds since January 1, 1970. This value is not yet what is needed though. The reason why is described shortly.

The main `mktime` function takes a single character string argument. This string is a representation of a date and time in a "canonical" (fixed) form. This string should be "*year month day hour minute second*".

```
# mktime --- convert a date into seconds,
#            compensate for time zone

function mktime(str,    res1, res2, a, b, i, j, t, diff)
{
    i = split(str, a, " ")    # don't rely on FS

    if (i != 6)
        return -1

    # force numeric
    for (j in a)
        a[j] += 0

    # validate
    if (a[1] < 1970 ||
        a[2] < 1 || a[2] > 12 ||
        a[3] < 1 || a[3] > 31 ||
        a[4] < 0 || a[4] > 23 ||
        a[5] < 0 || a[5] > 59 ||
        a[6] < 0 || a[6] > 60 )
            return -1
    res1 = _tm_addup(a)
    t = strftime("%Y %m %d %H %M %S", res1)

    if (_tm_debug)
        printf("(%s) -> (%s)\n", str, t) > "/dev/stderr"
```

[3] This is the Epoch on POSIX systems. It may be different on other systems.

```
        split(t, b, " ")
        res2 = _tm_addup(b)

        diff = res1 - res2

        if (_tm_debug)
            printf("diff = %d seconds\n", diff) > "/dev/stderr"

        res1 += diff

        return res1
    }
```

The function first splits the string into an array, using spaces and tabs as separators. If there are not six elements in the array, it returns an error, signaled as the value -1. Next, it forces each element of the array to be numeric, by adding zero to it. The following '`if`' statement then makes sure that each element is within an allowable range. (This checking could be extended further, e.g., to make sure that the day of the month is within the correct range for the particular month supplied.) All of this is essentially preliminary set-up and error checking.

Recall that `_tm_addup` generated a value in seconds since Midnight, January 1, 1970. This value is not directly usable as the result we want, *since the calculation does not account for the local timezone*. In other words, the value represents the count in seconds since the Epoch, but only for UTC (Universal Coordinated Time). If the local timezone is east or west of UTC, then some number of hours should be either added to, or subtracted from the resulting timestamp.

For example, 6:23 p.m. in Atlanta, Georgia (USA), is normally five hours west of (behind) UTC. It is only four hours behind UTC if daylight savings time is in effect. If you are calling `mktime` in Atlanta, with the argument `"1993 5 23 18 23 12"`, the result from `_tm_addup` will be for 6:23 p.m. UTC, which is only 2:23 p.m. in Atlanta. It is necessary to add another four hours worth of seconds to the result.

How can `mktime` determine how far away it is from UTC? This is surprisingly easy. The returned timestamp represents the time passed to `mktime` *as UTC*. This timestamp can be fed back to `strftime`, which will format it as a *local* time; i.e. as if it already had the UTC difference added in to it. This is done by giving `"%Y %m %d %H %M %S"` to `strftime` as the format argument. It returns the computed timestamp in the original string format. The result represents a time that accounts for the UTC difference. When the new time is converted back to a timestamp, the difference between the two timestamps is the difference (in seconds) between the local timezone and UTC. This difference is then added back to the original result. An example demonstrating this is presented below.

Finally, there is a "main" program for testing the function.

```
BEGIN  {
    if (_tm_test) {
        printf "Enter date as yyyy mm dd hh mm ss: "
        getline _tm_test_date

        t = mktime(_tm_test_date)
        r = strftime("%Y %m %d %H %M %S", t)
        printf "Got back (%s)\n", r
    }
}
```

The entire program uses two variables that can be set on the command line to control debugging output and to enable the test in the final BEGIN rule. Here is the result of a test run. (Note that debugging output is to standard error, and test output is to standard output.)

```
$ gawk -f mktime.awk -v _tm_test=1 -v _tm_debug=1
-| Enter date as yyyy mm dd hh mm ss: 1993 5 23 15 35 10
error  (1993 5 23 15 35 10) -> (1993 05 23 11 35 10)
error  diff = 14400 seconds
-| Got back (1993 05 23 15 35 10)
```

The time entered was 3:35 p.m. (15:35 on a 24-hour clock), on May 23, 1993. The first line of debugging output shows the resulting time as UTC—four hours ahead of the local time zone. The second line shows that the difference is 14400 seconds, which is four hours. (The difference is only four hours, since daylight savings time is in effect during May.) The final line of test output shows that the timezone compensation algorithm works; the returned time is the same as the entered time.

This program does not solve the general problem of turning an arbitrary date representation into a timestamp. That problem is very involved. However, the `mktime` function provides a foundation upon which to build. Other software can convert month names into numeric months, and AM/PM times into 24-hour clocks, to generate the "canonical" format that `mktime` requires.

15.8. Managing the Time of Day

The `systime` and `strftime` functions described in 12.5. Bekezdés [Functions for Dealing with Time Stamps], 147. oldal, provide the minimum functionality necessary for dealing with the time of day in human readable form. While `strftime` is extensive, the control formats are not necessarily easy to remember or intuitively obvious when reading a program.

The following function, `gettimeofday`, populates a user-supplied array with pre-formatted time information. It returns a string with the current time formatted in the same way as the `date` utility.

```
# gettimeofday --- get the time of day in a usable format
# Arnold Robbins, arnold@gnu.org, Public Domain, May 1993
#
# Returns a string in the format of output of date(1)
# Populates the array argument time with individual values:
#    time["second"]       -- seconds (0 - 59)
#    time["minute"]       -- minutes (0 - 59)
#    time["hour"]         -- hours (0 - 23)
#    time["althour"]      -- hours (0 - 12)
#    time["monthday"]     -- day of month (1 - 31)
#    time["month"]        -- month of year (1 - 12)
#    time["monthname"]    -- name of the month
#    time["shortmonth"]   -- short name of the month
#    time["year"]         -- year within century (0 - 99)
#    time["fullyear"]     -- year with century (19xx or 20xx)
#    time["weekday"]      -- day of week (Sunday = 0)
#    time["altweekday"]   -- day of week (Monday = 0)
#    time["weeknum"]      -- week number, Sunday first day
#    time["altweeknum"]   -- week number, Monday first day
#    time["dayname"]      -- name of weekday
#    time["shortdayname"] -- short name of weekday
#    time["yearday"]      -- day of year (0 - 365)
#    time["timezone"]     -- abbreviation of timezone name
#    time["ampm"]         -- AM or PM designation

function gettimeofday(time,    ret, now, i)
{
    # get time once, avoids unnecessary system calls
    now = systime()

    # return date(1)-style output
    ret = strftime("%a %b %d %H:%M:%S %Z %Y", now)

    # clear out target array
    for (i in time)
        delete time[i]
```

```
    # fill in values, force numeric values to be
    # numeric by adding 0
    time["second"]       = strftime("%S", now) + 0
    time["minute"]       = strftime("%M", now) + 0
    time["hour"]         = strftime("%H", now) + 0
    time["althour"]      = strftime("%I", now) + 0
    time["monthday"]     = strftime("%d", now) + 0
    time["month"]        = strftime("%m", now) + 0
    time["monthname"]    = strftime("%B", now)
    time["shortmonth"]   = strftime("%b", now)
    time["year"]         = strftime("%y", now) + 0
    time["fullyear"]     = strftime("%Y", now) + 0
    time["weekday"]      = strftime("%w", now) + 0
    time["altweekday"]   = strftime("%u", now) + 0
    time["dayname"]      = strftime("%A", now)
    time["shortdayname"] = strftime("%a", now)
    time["yearday"]      = strftime("%j", now) + 0
    time["timezone"]     = strftime("%Z", now)
    time["ampm"]         = strftime("%p", now)
    time["weeknum"]      = strftime("%U", now) + 0
    time["altweeknum"]   = strftime("%W", now) + 0

    return ret
}
```

The string indices are easier to use and read than the various formats required by `strftime`. The `alarm` program presented in 16.2.2. Bekezdés [An Alarm Clock Program], 228. oldal, uses this function.

The `gettimeofday` function is presented above as it was written. A more general design for this function would have allowed the user to supply an optional timestamp value that would have been used instead of the current time.

15.9. Noting Data File Boundaries

The `BEGIN` and `END` rules are each executed exactly once, at the beginning and end respectively of your `awk` program (lásd 8.1.5. Bekezdés [The `BEGIN` and `END` Special Patterns], 100. oldal). We (the `gawk` authors) once had a user who mistakenly thought that the `BEGIN` rule was executed at the beginning of each data file and the `END` rule was executed at the end of each data file. When informed that this was not the case, the user requested that we add new special patterns to `gawk`, named `BEGIN_FILE` and `END_FILE`, that would have the desired behavior. He even supplied us the code to do so.

However, after a little thought, I came up with the following library program. It arranges to call two user-supplied functions, `beginfile` and `endfile`, at the beginning and end of each data file. Besides solving the problem in only nine(!) lines of code, it does so *portably*; this will work with any implementation of `awk`.

```
# transfile.awk
#
# Give the user a hook for filename transitions
#
# The user must supply functions beginfile() and endfile()
# that each take the name of the file being started or
# finished, respectively.
#
# Arnold Robbins, arnold@gnu.org, January 1992
# Public Domain

FILENAME != _oldfilename \
{
    if (_oldfilename != "")
        endfile(_oldfilename)
    _oldfilename = FILENAME
    beginfile(FILENAME)
}

END   { endfile(FILENAME) }
```

This file must be loaded before the user's "main" program, so that the rule it supplies will be executed first.

This rule relies on `awk`'s `FILENAME` variable that automatically changes for each new data file. The current file name is saved in a private variable, `_oldfilename`. If `FILENAME` does not equal `_oldfilename`, then a new data file is being processed, and it is necessary to call `endfile` for the old file. Since `endfile` should only be called if a file has been processed, the program first checks to make sure that `_oldfilename` is not the null string. The program then assigns the current file name to `_oldfilename`, and calls `beginfile` for the file. Since, like all `awk` variables, `_oldfilename` will be initialized to the null string, this rule executes correctly even for the first data file.

The program also supplies an `END` rule, to do the final processing for the last file. Since this `END` rule comes before any `END` rules supplied in the "main" program, `endfile` will be called first. Once again the value of multiple `BEGIN` and `END` rules should be clear.

This version has same problem as the first version of `nextfile` (lásd 15.2. Bekezdés [Implementing `nextfile` as a Function], 170. oldal). If the same data file occurs twice in a row on command line, then `endfile` and `beginfile` will not be executed at the end of the first pass and at the beginning of the second pass. This version solves the problem.

```
# ftrans.awk --- handle data file transitions
#
# user supplies beginfile() and endfile() functions
#
# Arnold Robbins, arnold@gnu.org, November 1992
# Public Domain

FNR == 1 {
    if (_filename_ != "")
        endfile(_filename_)
    _filename_ = FILENAME
    beginfile(FILENAME)
}

END  { endfile(_filename_) }
```

In 16.1.7. Bekezdés [Counting Things], 224. oldal, you will see how this library function can be used, and how it simplifies writing the main program.

15.10. Processing Command Line Options

Most utilities on POSIX compatible systems take options or "switches" on the command line that can be used to change the way a program behaves. `awk` is an example of such a program (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). Often, options take *arguments*, data that the program needs to correctly obey the command line option. For example, `awk`'s '`-F`' option requires a string to use as the field separator. The first occurrence on the command line of either '`--`' or a string that does not begin with '`-`' ends the options.

Most Unix systems provide a C function named `getopt` for processing command line arguments. The programmer provides a string describing the one letter options. If an option requires an argument, it is followed in the string with a colon. `getopt` is also passed the count and values of the command line arguments, and is called in a loop. `getopt` processes the command line arguments for option letters. Each time around the loop, it returns a single character representing the next option letter that it found, or '`?`' if it found an invalid option. When it returns −1, there are no options left on the command line.

When using `getopt`, options that do not take arguments can be grouped together. Furthermore, options that take arguments require that the argument be present. The argument can immediately follow the option letter, or it can be a separate command line argument.

Given a hypothetical program that takes three command line options, '-a', '-b', and '-c', and '-b' requires an argument, all of the following are valid ways of invoking the program:

```
prog -a -b foo -c data1 data2 data3
prog -ac -bfoo -- data1 data2 data3
prog -acbfoo data1 data2 data3
```

Notice that when the argument is grouped with its option, the rest of the command line argument is considered to be the option's argument. In the above example, '-acbfoo' indicates that all of the '-a', '-b', and '-c' options were supplied, and that 'foo' is the argument to the '-b' option.

`getopt` provides four external variables that the programmer can use.

`optind` The index in the argument value array (`argv`) where the first non-option command line argument can be found.

`optarg` The string value of the argument to an option.

`opterr` Usually `getopt` prints an error message when it finds an invalid option. Setting `opterr` to zero disables this feature. (An application might wish to print its own error message.)

`optopt` The letter representing the command line option. While not usually documented, most versions supply this variable.

The following C fragment shows how `getopt` might process command line arguments for `awk`.

```
int
main(int argc, char *argv[])
{
    ...
    /* print our own message */
    opterr = 0;
```

```
    while ((c = getopt(argc, argv, "v:f:F:W:")) != -1) {
        switch (c) {
        case 'f':    /* file */
            ...
            break;
        case 'F':    /* field separator */
            ...
            break;
        case 'v':    /* variable assignment */
            ...
            break;
        case 'W':    /* extension */
            ...
            break;
        case '?':
        default:
            usage();
            break;
        }
    }
    ...
}
```

As a side point, `gawk` actually uses the GNU `getopt_long` function to process both normal and GNU-style long options (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

The abstraction provided by `getopt` is very useful, and would be quite handy in `awk` programs as well. Here is an `awk` version of `getopt`. This function highlights one of the greatest weaknesses in `awk`, which is that it is very poor at manipulating single characters. Repeated calls to `substr` are necessary for accessing individual characters (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).

The discussion walks through the code a bit at a time.

```
# getopt --- do C library getopt(3) function in awk
#
# arnold@gnu.org
# Public domain
#
# Initial version: March, 1991
# Revised: May, 1993
```

```
# External variables:
#    Optind -- index of ARGV for first non-option argument
#    Optarg -- string value of argument to current option
#    Opterr -- if non-zero, print our own diagnostic
#    Optopt -- current option letter

# Returns
#    -1     at end of options
#    ?      for unrecognized option
#    <c>    a character representing the current option

# Private Data
#    _opti  index in multi-flag option, e.g., -abc
```

The function starts out with some documentation: who wrote the code, and when it was revised, followed by a list of the global variables it uses, what the return values are and what they mean, and any global variables that are "private" to this library function. Such documentation is essential for any program, and particularly for library functions.

```
function getopt(argc, argv, options,    optl, thisopt, i)
{
    optl = length(options)
    if (optl == 0)        # no options given
        return -1

    if (argv[Optind] == "--") {  # all done
        Optind++
        _opti = 0
        return -1
    } else if (argv[Optind] !~ /^-[^: \t\n\f\r\v\b]/) {
        _opti = 0
        return -1
    }
```

The function first checks that it was indeed called with a string of options (the `options` parameter). If `options` has a zero length, `getopt` immediately returns −1.

The next thing to check for is the end of the options. A '`--`' ends the command line options, as does any command line argument that does not begin with a '`-`'. `Optind` is used to step through the array of command line arguments; it retains its value across calls to `getopt`, since it is a global variable.

The regexp used, `/^-[^: \t\n\f\r\v\b]/`, is perhaps a bit of overkill; it checks for a '`-`' followed by anything that is not whitespace and not a colon. If the current command line argument does not match this pattern, it is not an option, and it ends option processing.

```
        if (_opti == 0)
            _opti = 2
        thisopt = substr(argv[Optind], _opti, 1)
        Optopt = thisopt
        i = index(options, thisopt)
        if (i == 0) {
            if (Opterr)
                printf("%c -- invalid option\n",
                                      thisopt) > "/dev/stderr"
            if (_opti >= length(argv[Optind])) {
                Optind++
                _opti = 0
            } else
                _opti++
            return "?"
        }
```

The `_opti` variable tracks the position in the current command line argument (`argv[Optind]`). In the case that multiple options were grouped together with one '`-`' (e.g., '`-abx`'), it is necessary to return them to the user one at a time.

If `_opti` is equal to zero, it is set to two, the index in the string of the next character to look at (we skip the '`-`', which is at position one). The variable `thisopt` holds the character, obtained with `substr`. It is saved in `Optopt` for the main program to use.

If `thisopt` is not in the `options` string, then it is an invalid option. If `Opterr` is non-zero, `getopt` prints an error message on the standard error that is similar to the message from the C version of `getopt`.

Since the option is invalid, it is necessary to skip it and move on to the next option character. If `_opti` is greater than or equal to the length of the current command line argument, then it is necessary to move on to the next one, so `Optind` is incremented and `_opti` is reset to zero. Otherwise, `Optind` is left alone and `_opti` is merely incremented.

In any case, since the option was invalid, `getopt` returns '`?`'. The main program can examine `Optopt` if it needs to know what the invalid option letter actually was.

```
        if (substr(options, i + 1, 1) == ":") {
            # get option argument
            if (length(substr(argv[Optind], _opti + 1)) > 0)
                Optarg = substr(argv[Optind], _opti + 1)
            else
                Optarg = argv[++Optind]
            _opti = 0
        } else
            Optarg = ""
```

If the option requires an argument, the option letter is followed by a colon in the `options` string. If there are remaining characters in the current command line argument (`argv[Optind]`), then the rest of that string is assigned to `Optarg`. Otherwise, the next command line argument is used (‘`-xFOO`’ vs. ‘`-x FOO`’). In either case, `_opti` is reset to zero, since there are no more characters left to examine in the current command line argument.

```
        if (_opti == 0 || _opti >= length(argv[Optind])) {
            Optind++
            _opti = 0
        } else
            _opti++
        return thisopt
    }
```

Finally, if `_opti` is either zero or greater than the length of the current command line argument, it means this element in `argv` is through being processed, so `Optind` is incremented to point to the next element in `argv`. If neither condition is true, then only `_opti` is incremented, so that the next option letter can be processed on the next call to `getopt`.

```
    BEGIN {
        Opterr = 1    # default is to diagnose
        Optind = 1    # skip ARGV[0]

        # test program
        if (_getopt_test) {
            while ((_go_c = getopt(ARGC, ARGV, "ab:cd")) != -1)
                printf("c = <%c>, optarg = <%s>\n",
                                               _go_c, Optarg)
            printf("non-option arguments:\n")
            for (; Optind < ARGC; Optind++)
                printf("\tARGV[%d] = <%s>\n",
                                        Optind, ARGV[Optind])
        }
    }
```

The `BEGIN` rule initializes both `Opterr` and `Optind` to one. `Opterr` is set to one, since the default behavior is for `getopt` to print a diagnostic message upon seeing an invalid option. `Optind` is set to one, since there's no reason to look at the program name, which is in `ARGV[0]`.

The rest of the `BEGIN` rule is a simple test program. Here is the result of two sample runs of the test program.

```
$ awk -f getopt.awk -v _getopt_test=1 -- -a -cbARG bax -x
⊣ c = <a>, optarg = <>
⊣ c = <c>, optarg = <>
⊣ c = <b>, optarg = <ARG>
⊣ non-option arguments:
⊣         ARGV[3] = <bax>
⊣         ARGV[4] = <-x>

$ awk -f getopt.awk -v _getopt_test=1 -- -a -x -- xyz abc
⊣ c = <a>, optarg = <>
error x -- invalid option
⊣ c = <?>, optarg = <>
⊣ non-option arguments:
⊣         ARGV[4] = <xyz>
⊣         ARGV[5] = <abc>
```

The first '`--`' terminates the arguments to `awk`, so that it does not try to interpret the '`-a`' etc. as its own options.

Several of the sample programs presented in 16. Fejezet [Practical `awk` Programs], 205. oldal, use `getopt` to process their arguments.

15.11. Reading the User Database

The '`/dev/user`' special file (lásd 6.7. Bekezdés [Special File Names in `gawk`], 71. oldal) provides access to the current user's real and effective user and group id numbers, and if available, the user's supplementary group set. However, since these are numbers, they do not provide very useful information to the average user. There needs to be some way to find the user information associated with the user and group numbers. This section presents a suite of functions for retrieving information from the user database. Lásd 15.12. Bekezdés [Reading the Group Database], 197. oldal, for a similar suite that retrieves information from the group database.

The POSIX standard does not define the file where user information is kept. Instead, it provides the `<pwd.h>` header file and several C language subroutines for obtaining user information. The primary function is `getpwent`, for "get password entry." The "password" comes from the original user database file, '`/etc/passwd`', which kept user information, along with the encrypted passwords (hence the name).

While an `awk` program could simply read '`/etc/passwd`' directly (the format is well known), because of the way password files are handled on networked systems, this file may not contain complete information about the system's set of users.

To be sure of being able to produce a readable, complete version of the user database, it is necessary to write a small C program that calls `getpwent`. `getpwent` is defined to return a pointer to a `struct passwd`. Each time it is called, it returns the next entry in the database. When there are no more entries, it returns `NULL`, the null pointer. When this happens, the C program should call `endpwent` to close the database. Here is `pwcat`, a C program that "cats" the password database.

```
/*
 * pwcat.c
 *
 * Generate a printable version of the password database
 *
 * Arnold Robbins
 * arnold@gnu.org
 * May 1993
 * Public Domain
 */

#include <stdio.h>
#include <pwd.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct passwd *p;

    while ((p = getpwent()) != NULL)
        printf("%s:%s:%d:%d:%s:%s:%s\n",
            p->pw_name, p->pw_passwd, p->pw_uid,
            p->pw_gid, p->pw_gecos, p->pw_dir, p->pw_shell);

    endpwent();
    exit(0);
}
```

If you don't understand C, don't worry about it. The output from `pwcat` is the user database, in the traditional '`/etc/passwd`' format of colon-separated fields. The fields are:

Login name
: The user's login name.

Encrypted password
: The user's encrypted password. This may not be available on some systems.

User-ID
: The user's numeric user-id number.

Group-ID
: The user's numeric group-id number.

Full name
: The user's full name, and perhaps other information associated with the user.

Home directory
: The user's login, or "home" directory (familiar to shell programmers as `$HOME`).

Login shell
: The program that will be run when the user logs in. This is usually a shell, such as Bash (the Gnu Bourne-Again shell).

Here are a few lines representative of `pwcat`'s output.

```
$ pwcat
⊣ root:3Ov02d5VaUPB6:0:1:Operator:/:/bin/sh
⊣ nobody:*:65534:65534::/:
⊣ daemon:*:1:1::/:
⊣ sys:*:2:2::/:/bin/csh
⊣ bin:*:3:3::/bin:
⊣ arnold:xyzzy:2076:10:Arnold Robbins:/home/arnold:/bin/sh
⊣ miriam:yxaay:112:10:Miriam Robbins:/home/miriam:/bin/sh
⊣ andy:abcca2:113:10:Andy Jacobs:/home/andy:/bin/sh
...
```

With that introduction, here is a group of functions for getting user information. There are several functions here, corresponding to the C functions of the same name.

```
# passwd.awk --- access password file information
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

BEGIN {
    # tailor this to suit your system
    _pw_awklib = "/usr/local/libexec/awk/"
}
```

```
function _pw_init(    oldfs, oldrs, olddol0, pwcat)
{
    if (_pw_inited)
        return
    oldfs = FS
    oldrs = RS
    olddol0 = $0
    FS = ":"
    RS = "\n"
    pwcat = _pw_awklib "pwcat"
    while ((pwcat | getline) > 0) {
        _pw_byname[$1] = $0
        _pw_byuid[$3] = $0
        _pw_bycount[++_pw_total] = $0
    }
    close(pwcat)
    _pw_count = 0
    _pw_inited = 1
    FS = oldfs
    RS = oldrs
    $0 = olddol0
}
```

The `BEGIN` rule sets a private variable to the directory where `pwcat` is stored. Since it is used to help out an `awk` library routine, we have chosen to put it in '`/usr/local/libexec/awk`'. You might want it to be in a different directory on your system.

The function `_pw_init` keeps three copies of the user information in three associative arrays. The arrays are indexed by user name (`_pw_byname`), by user-id number (`_pw_byuid`), and by order of occurrence (`_pw_bycount`).

The variable `_pw_inited` is used for efficiency; `_pw_init` only needs to be called once.

Since this function uses `getline` to read information from `pwcat`, it first saves the values of `FS`, `RS`, and `$0`. Doing so is necessary, since these functions could be called from anywhere within a user's program, and the user may have his or her own values for `FS` and `RS`.

The main part of the function uses a loop to read database lines, split the line into fields, and then store the line into each array as necessary. When the loop is done, `_pw_init` cleans up by closing the pipeline, setting `_pw_inited` to one, and restoring `FS`, `RS`, and `$0`. The use of `_pw_count` will be explained below.

```
function getpwnam(name)
{
    _pw_init()
    if (name in _pw_byname)
        return _pw_byname[name]
    return ""
}
```

The getpwnam function takes a user name as a string argument. If that user is in the database, it returns the appropriate line. Otherwise it returns the null string.

```
function getpwuid(uid)
{
    _pw_init()
    if (uid in _pw_byuid)
        return _pw_byuid[uid]
    return ""
}
```

Similarly, the getpwuid function takes a user-id number argument. If that user number is in the database, it returns the appropriate line. Otherwise it returns the null string.

```
function getpwent()
{
    _pw_init()
    if (_pw_count < _pw_total)
        return _pw_bycount[++_pw_count]
    return ""
}
```

The getpwent function simply steps through the database, one entry at a time. It uses _pw_count to track its current position in the _pw_bycount array.

```
function endpwent()
{
    _pw_count = 0
}
```

The endpwent function resets _pw_count to zero, so that subsequent calls to getpwent will start over again.

A conscious design decision in this suite is that each subroutine calls _pw_init to initialize the database arrays. The overhead of running a separate process to generate the user database, and the I/O to scan it, will only be incurred if the user's main program actually calls one of these functions. If this library file is loaded along with a user's program, but none of the routines are ever called, then there is no extra run-time overhead. (The alternative would be to move the body of _pw_init into a BEGIN rule, which would always run pwcat. This simplifies the code but runs an extra process that may never be needed.)

In turn, calling _pw_init is not too expensive, since the _pw_inited variable keeps the program from reading the data more than once. If you are worried about squeezing every last cycle out of your awk program, the check of _pw_inited could be moved out of _pw_init and duplicated in all the other functions. In practice, this is not necessary, since most awk programs are I/O bound, and it would clutter up the code.

The id program in 16.1.3. Bekezdés [Printing Out User Information], 214. oldal, uses these functions.

15.12. Reading the Group Database

Much of the discussion presented in 15.11. Bekezdés [Reading the User Database], 192. oldal, applies to the group database as well. Although there has traditionally been a well known file, '/etc/group', in a well known format, the POSIX standard only provides a set of C library routines (<grp.h> and getgrent) for accessing the information. Even though this file may exist, it likely does not have complete information. Therefore, as with the user database, it is necessary to have a small C program that generates the group database as its output.

Here is grcat, a C program that "cats" the group database.

```
/*
 * grcat.c
 *
 * Generate a printable version of the group database
 *
 * Arnold Robbins, arnold@gnu.org
 * May 1993
 * Public Domain
 */

#include <stdio.h>
#include <grp.h>

int
main(argc, argv)
int argc;
char **argv;
{
    struct group *g;
    int i;
    while ((g = getgrent()) != NULL) {
        printf("%s:%s:%d:", g->gr_name, g->gr_passwd,
```

```
                                                g->gr_gid);
            for (i = 0; g->gr_mem[i] != NULL; i++) {
                printf("%s", g->gr_mem[i]);
                if (g->gr_mem[i+1] != NULL)
                    putchar(',');
            }
            putchar('\n');
        }
        endgrent();
        exit(0);
    }
```

Each line in the group database represent one group. The fields are separated with colons, and represent the following information.

Group Name
: The name of the group.

Group Password
: The encrypted group password. In practice, this field is never used. It is usually empty, or set to '*'.

Group ID Number
: The numeric group-id number. This number should be unique within the file.

Group Member List
: A comma-separated list of user names. These users are members of the group. Most Unix systems allow users to be members of several groups simultaneously. If your system does, then reading '/dev/user' will return those group-id numbers in `$5` through `$NF`. (Note that '/dev/user' is a `gawk` extension; lásd 6.7. Bekezdés [Special File Names in `gawk`], 71. oldal.)

Here is what running `grcat` might produce:

```
$ grcat
⊣ wheel:*:0:arnold
⊣ nogroup:*:65534:
⊣ daemon:*:1:
⊣ kmem:*:2:
⊣ staff:*:10:arnold,miriam,andy
⊣ other:*:20:
...
```

Here are the functions for obtaining information from the group database. There are several, modeled after the C library functions of the same names.

```
# group.awk --- functions for dealing with the group file
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

BEGIN    \
{
    # Change to suit your system
    _gr_awklib = "/usr/local/libexec/awk/"
}
function _gr_init(    oldfs, oldrs, olddol0, grcat, n, a, i)
{
    if (_gr_inited)
        return
    oldfs = FS
    oldrs = RS
    olddol0 = $0
    FS = ":"
    RS = "\n"
    grcat = _gr_awklib "grcat"
    while ((grcat | getline) > 0) {
        if ($1 in _gr_byname)
            _gr_byname[$1] = _gr_byname[$1] "," $4
        else
            _gr_byname[$1] = $0
        if ($3 in _gr_bygid)
            _gr_bygid[$3] = _gr_bygid[$3] "," $4
        else
            _gr_bygid[$3] = $0

        n = split($4, a, "[ \t]*,[ \t]*")
        for (i = 1; i <= n; i++)
            if (a[i] in _gr_groupsbyuser)
                _gr_groupsbyuser[a[i]] = \
                    _gr_groupsbyuser[a[i]] " " $1
            else
                _gr_groupsbyuser[a[i]] = $1
        _gr_bycount[++_gr_count] = $0
    }
```

```
        close(grcat)
        _gr_count = 0
        _gr_inited++
        FS = oldfs
        RS = oldrs
        $0 = olddol0
    }
```

The `BEGIN` rule sets a private variable to the directory where `grcat` is stored. Since it is used to help out an `awk` library routine, we have chosen to put it in '`/usr/local/libexec/awk`'. You might want it to be in a different directory on your system.

These routines follow the same general outline as the user database routines (lásd 15.11. Bekezdés [Reading the User Database], 192. oldal). The `_gr_inited` variable is used to ensure that the database is scanned no more than once. The `_gr_init` function first saves `FS`, `RS`, and `$0`, and then sets `FS` and `RS` to the correct values for scanning the group information.

The group information is stored is several associative arrays. The arrays are indexed by group name (`_gr_byname`), by group-id number (`_gr_bygid`), and by position in the database (`_gr_bycount`). There is an additional array indexed by user name (`_gr_groupsbyuser`), that is a space separated list of groups that each user belongs to.

Unlike the user database, it is possible to have multiple records in the database for the same group. This is common when a group has a large number of members. Such a pair of entries might look like:

```
tvpeople:*:101:johny,jay,arsenio
tvpeople:*:101:david,conan,tom,joan
```

For this reason, `_gr_init` looks to see if a group name or group-id number has already been seen. If it has, then the user names are simply concatenated onto the previous list of users. (There is actually a subtle problem with the code presented above. Suppose that the first time there were no names. This code adds the names with a leading comma. It also doesn't check that there is a `$4`.)

Finally, `_gr_init` closes the pipeline to `grcat`, restores `FS`, `RS`, and `$0`, initializes `_gr_count` to zero (it is used later), and makes `_gr_inited` non-zero.

```
function getgrnam(group)
{
    _gr_init()
    if (group in _gr_byname)
        return _gr_byname[group]
    return ""
}
```

The `getgrnam` function takes a group name as its argument, and if that group exists, it is returned. Otherwise, `getgrnam` returns the null string.

```
function getgrgid(gid)
{
    _gr_init()
    if (gid in _gr_bygid)
        return _gr_bygid[gid]
    return ""
}
```

The `getgrgid` function is similar, it takes a numeric group-id, and looks up the information associated with that group-id.

```
function getgruser(user)
{
    _gr_init()
    if (user in _gr_groupsbyuser)
        return _gr_groupsbyuser[user]
    return ""
}
```

The `getgruser` function does not have a C counterpart. It takes a user name, and returns the list of groups that have the user as a member.

```
function getgrent()
{
    _gr_init()
    if (++_gr_count in _gr_bycount)
        return _gr_bycount[_gr_count]
    return ""
}
```

The `getgrent` function steps through the database one entry at a time. It uses `_gr_count` to track its position in the list.

```
function endgrent()
{
    _gr_count = 0
}
```

`endgrent` resets `_gr_count` to zero so that `getgrent` can start over again.

As with the user database routines, each function calls `_gr_init` to initialize the arrays. Doing so only incurs the extra overhead of running `grcat` if these functions are used (as opposed to moving the body of `_gr_init` into a `BEGIN` rule).

Most of the work is in scanning the database and building the various associative arrays. The functions that the user calls are themselves very simple, relying on `awk`'s associative arrays to do work.

The `id` program in 16.1.3. Bekezdés [Printing Out User Information], 214. oldal, uses these functions.

15.13. Naming Library Function Global Variables

Due to the way the `awk` language evolved, variables are either *global* (usable by the entire program), or *local* (usable just by a specific function). There is no intermediate state analogous to `static` variables in C.

Library functions often need to have global variables that they can use to preserve state information between calls to the function. For example, `getopt`'s variable `_opti` (lásd 15.10. Bekezdés [Processing Command Line Options], 186. oldal), and the `_tm_months` array used by `mktime` (lásd 15.7. Bekezdés [Turning Dates Into Timestamps], 177. oldal). Such variables are called *private*, since the only functions that need to use them are the ones in the library.

When writing a library function, you should try to choose names for your private variables so that they will not conflict with any variables used by either another library function or a user's main program. For example, a name like '`i`' or '`j`' is not a good choice, since user programs often use variable names like these for their own purposes.

The example programs shown in this chapter all start the names of their private variables with an underscore ('`_`'). Users generally don't use leading underscores in their variable names, so this convention immediately decreases the chances that the variable name will be accidentally shared with the user's program.

In addition, several of the library functions use a prefix that helps indicate what function or set of functions uses the variables. For example, `_tm_months` in `mktime` (lásd 15.7. Bekezdés [Turning Dates Into Timestamps], 177. oldal), and `_pw_byname` in the user data base routines (lásd 15.11. Bekezdés [Reading the User Database], 192. oldal). This convention is recommended, since it even further decreases the chance of inadvertent conflict among variable names. Note that this convention can be used equally well both for variable names and for private function names too.

While I could have re-written all the library routines to use this convention, I did not do so, in order to show how my own `awk` programming style has evolved, and to provide some basis for this discussion.

As a final note on variable naming, if a function makes global variables available for use by a main program, it is a good convention to start that variable's name with a capital letter. For example, `getopt`'s `Opterr` and `Optind` variables (lásd 15.10. Bekezdés [Processing Command Line Options], 186. oldal). The leading capital letter indicates that it is global, while the fact that the variable name is not all capital letters indicates that the variable is not one of `awk`'s built-in variables, like `FS`.

It is also important that *all* variables in library functions that do not need to save state are in fact declared local. If this is not done, the variable could accidentally be used in the user's program, leading to bugs that are very difficult to track down.

```
function lib_func(x, y,    l1, l2)
{
    ...
    use variable some_var  # some_var could be local
    ...                    # but is not by oversight
}
```

A different convention, common in the Tcl community, is to use a single associative array to hold the values needed by the library function(s), or "package." This significantly decreases the number of actual global names in use. For example, the functions described in 15.11. Bekezdés [Reading the User Database], 192. oldal, might have used `PW_data["inited"]`, `PW_data["total"]`, `PW_data["count"]` and `PW_data["awklib"]`, instead of `_pw_inited`, `_pw_awklib`, `_pw_total`, and `_pw_count`.

The conventions presented in this section are exactly that, conventions. You are not required to write your programs this way, we merely recommend that you do so.

16. Practical awk Programs

This chapter presents a potpourri of `awk` programs for your reading enjoyment. There are two sections. The first presents `awk` versions of several common POSIX utilities. The second is a grab-bag of interesting programs.

Many of these programs use the library functions presented in 15. Fejezet [A Library of awk Functions], 169. oldal.

16.1. Re-inventing Wheels for Fun and Profit

This section presents a number of POSIX utilities that are implemented in `awk`. Re-inventing these programs in `awk` is often enjoyable, since the algorithms can be very clearly expressed, and usually the code is very concise and simple. This is true because `awk` does so much for you.

It should be noted that these programs are not necessarily intended to replace the installed versions on your system. Instead, their purpose is to illustrate `awk` language programming for "real world" tasks.

The programs are presented in alphabetical order.

16.1.1. Cutting Out Fields and Columns

The `cut` utility selects, or "cuts," either characters or fields from its standard input and sends them to its standard output. `cut` can cut out either a list of characters, or a list of fields. By default, fields are separated by tabs, but you may supply a command line option to change the field *delimiter*, i.e. the field separator character. `cut`'s definition of fields is less general than `awk`'s.

A common use of `cut` might be to pull out just the login name of logged-on users from the output of `who`. For example, the following pipeline generates a sorted, unique list of the logged on users:

```
who | cut -c1-8 | sort | uniq
```

The options for `cut` are:

`-c` *list* Use *list* as the list of characters to cut out. Items within the list may be separated by commas, and ranges of characters can be separated with dashes. The list '`1-8,15,22-35`' specifies characters one through eight, 15, and 22 through 35.

`-f` *list* Use *list* as the list of fields to cut out.

`-d` *delim* Use *delim* as the field separator character instead of the tab character.

`-s` Suppress printing of lines that do not contain the field delimiter.

The `awk` implementation of `cut` uses the `getopt` library function (lásd 15.10. Bekezdés [Processing Command Line Options], 186. oldal), and the join library function (lásd 15.6. Bekezdés [Merging an Array Into a String], 176. oldal).

The program begins with a comment describing the options and a usage function which prints out a usage message and exits. usage is called if invalid arguments are supplied.

```
# cut.awk --- implement cut in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Options:
#    -f list        Cut fields
#    -d c           Field delimiter character
#    -c list        Cut characters
#
#    -s        Suppress lines without the delimiter character

function usage(    e1, e2)
{
    e1 = "usage: cut [-f list] [-d c] [-s] [files...]"
    e2 = "usage: cut [-c list] [files...]"
    print e1 > "/dev/stderr"
    print e2 > "/dev/stderr"
    exit 1
}
```

The variables e1 and e2 are used so that the function fits nicely on the page.

Next comes a BEGIN rule that parses the command line options. It sets FS to a single tab character, since that is cut's default field separator. The output field separator is also set to be the same as the input field separator. Then getopt is used to step through the command line options. One or the other of the variables by_fields or by_chars is set to true, to indicate that processing should be done by fields or by characters respectively. When cutting by characters, the output field separator is set to the null string.

```
BEGIN    \
{
    FS = "\t"    # default
    OFS = FS
    while ((c = getopt(ARGC, ARGV, "sf:c:d:")) != -1) {
        if (c == "f") {
            by_fields = 1
            fieldlist = Optarg
```

```
    } else if (c == "c") {
        by_chars = 1
        fieldlist = Optarg
        OFS = ""
    } else if (c == "d") {
        if (length(Optarg) > 1) {
            printf("Using first character of %s" \
            " for delimiter\n", Optarg) > "/dev/stderr"
            Optarg = substr(Optarg, 1, 1)
        }
        FS = Optarg
        OFS = FS
        if (FS == " ")    # defeat awk semantics
            FS = "[ ]"
    } else if (c == "s")
        suppress++
    else
        usage()
}
for (i = 1; i < Optind; i++)
    ARGV[i] = ""
```

Special care is taken when the field delimiter is a space. Using `" "` (a single space) for the value of `FS` is incorrect—`awk` would separate fields with runs of spaces, tabs and/or newlines, and we want them to be separated with individual spaces. Also, note that after `getopt` is through, we have to clear out all the elements of `ARGV` from one to `Optind`, so that `awk` will not try to process the command line options as file names.

After dealing with the command line options, the program verifies that the options make sense. Only one or the other of '`-c`' and '`-f`' should be used, and both require a field list. Then either `set_fieldlist` or `set_charlist` is called to pull apart the list of fields or characters.

```
if (by_fields && by_chars)
    usage()

if (by_fields == 0 && by_chars == 0)
    by_fields = 1    # default

if (fieldlist == "") {
    print "cut: needs list for -c or -f" > "/dev/stderr"
    exit 1
}
```

```
    if (by_fields)
        set_fieldlist()
    else
        set_charlist()
}
```

Here is `set_fieldlist`. It first splits the field list apart at the commas, into an array. Then, for each element of the array, it looks to see if it is actually a range, and if so splits it apart. The range is verified to make sure the first number is smaller than the second. Each number in the list is added to the `flist` array, which simply lists the fields that will be printed. Normal field splitting is used. The program lets `awk` handle the job of doing the field splitting.

```
function set_fieldlist(        n, m, i, j, k, f, g)
{
    n = split(fieldlist, f, ",")
    j = 1    # index in flist
    for (i = 1; i <= n; i++) {
        if (index(f[i], "-") != 0) { # a range
            m = split(f[i], g, "-")
            if (m != 2 || g[1] >= g[2]) {
                printf("bad field list: %s\n",
                                  f[i]) > "/dev/stderr"
                exit 1
            }
            for (k = g[1]; k <= g[2]; k++)
                flist[j++] = k
        } else
            flist[j++] = f[i]
    }
    nfields = j - 1
}
```

The `set_charlist` function is more complicated than `set_fieldlist`. The idea here is to use `gawk`'s `FIELDWIDTHS` variable (lásd 5.6. Bekezdés [Reading Fixed-width Data], 49. oldal), which describes constant width input. When using a character list, that is exactly what we have.

Setting up `FIELDWIDTHS` is more complicated than simply listing the fields that need to be printed. We have to keep track of the fields to be printed, and also the intervening characters that have to be skipped. For example, suppose you wanted characters one through eight, 15, and 22 through 35. You would use '`-c 1-8,15,22-35`'. The necessary value for `FIELDWIDTHS` would be `"8 6 1 6 14"`. This gives us five fields, and what should be printed are `$1`, `$3`, and `$5`. The intermediate fields are "filler," stuff in between the desired data.

`flist` lists the fields to be printed, and `t` tracks the complete field list, including filler fields.

```
function set_charlist(    field, i, j, f, g, t,
                          filler, last, len)
{
    field = 1   # count total fields
    n = split(fieldlist, f, ",")
    j = 1       # index in flist
    for (i = 1; i <= n; i++) {
        if (index(f[i], "-") != 0) { # range
            m = split(f[i], g, "-")
            if (m != 2 || g[1] >= g[2]) {
                printf("bad character list: %s\n",
                               f[i]) > "/dev/stderr"
                exit 1
            }
            len = g[2] - g[1] + 1
            if (g[1] > 1)  # compute length of filler
                filler = g[1] - last - 1
            else
                filler = 0
            if (filler)
                t[field++] = filler
            t[field++] = len  # length of field
            last = g[2]
            flist[j++] = field - 1
        } else {
            if (f[i] > 1)
                filler = f[i] - last - 1
            else
                filler = 0
            if (filler)
                t[field++] = filler
            t[field++] = 1
            last = f[i]
            flist[j++] = field - 1
        }
    }
    FIELDWIDTHS = join(t, 1, field - 1)
    nfields = j - 1
}
```

Here is the rule that actually processes the data. If the '`-s`' option was given, then `suppress` will be true. The first `if` statement makes sure that the input record does have the field separator. If `cut` is processing fields, `suppress` is true, and the field separator character is not in the record, then the record is skipped.

If the record is valid, then at this point, `gawk` has split the data into fields, either using the character in `FS` or using fixed-length fields and `FIELDWIDTHS`. The loop goes through the list of fields that should be printed. If the corresponding field has data in it, it is printed. If the next field also has data, then the separator character is written out in between the fields.

```
{
    if (by_fields && suppress && $0 !~ FS)
        next

    for (i = 1; i <= nfields; i++) {
        if ($flist[i] != "") {
            printf "%s", $flist[i]
            if (i < nfields && $flist[i+1] != "")
                printf "%s", OFS
        }
    }
    print ""
}
```

This version of `cut` relies on `gawk`'s `FIELDWIDTHS` variable to do the character-based cutting. While it would be possible in other `awk` implementations to use `substr` (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal), it would also be extremely painful to do so. The `FIELDWIDTHS` variable supplies an elegant solution to the problem of picking the input line apart by characters.

16.1.2. Searching for Regular Expressions in Files

The `egrep` utility searches files for patterns. It uses regular expressions that are almost identical to those available in `awk` (lásd 7.1.2. Bekezdés [Regular Expression Constants], 77. oldal). It is used this way:

`egrep` [*options*] `'`*pattern*`'` *files* ...

The *pattern* is a regexp. In typical usage, the regexp is quoted to prevent the shell from expanding any of the special characters as file name wildcards. Normally, `egrep` prints the lines that matched. If multiple file names are provided on the command line, each output line is preceded by the name of the file and a colon.

The options are:

`-c` Print out a count of the lines that matched the pattern, instead of the lines themselves.

`-s` Be silent. No output is produced, and the exit value indicates whether or not the pattern was matched.

`-v` Invert the sense of the test. `egrep` prints the lines that do *not* match the pattern, and exits successfully if the pattern was not matched.

`-i` Ignore case distinctions in both the pattern and the input data.

`-l` Only print the names of the files that matched, not the lines that matched.

`-e` *pattern* Use *pattern* as the regexp to match. The purpose of the '`-e`' option is to allow patterns that start with a '`-`'.

This version uses the `getopt` library function (lásd 15.10. Bekezdés [Processing Command Line Options], 186. oldal), and the file transition library program (lásd 15.9. Bekezdés [Noting Data File Boundaries], 184. oldal).

The program begins with a descriptive comment, and then a `BEGIN` rule that processes the command line arguments with `getopt`. The '`-i`' (ignore case) option is particularly easy with `gawk`; we just use the `IGNORECASE` built in variable (lásd 10. Fejezet [Built-in Variables], 115. oldal).

```
# egrep.awk --- simulate egrep in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Options:
#    -c    count of lines
#    -s    silent - use exit value
#    -v    invert test, success if no match
#    -i    ignore case
#    -l    print filenames only
#    -e    argument is pattern

BEGIN {
    while ((c = getopt(ARGC, ARGV, "ce:svil")) != -1) {
        if (c == "c")
            count_only++
        else if (c == "s")
            no_print++
        else if (c == "v")
            invert++
        else if (c == "i")
```

```
                IGNORECASE = 1
            else if (c == "l")
                filenames_only++
            else if (c == "e")
                pattern = Optarg
            else
                usage()
        }
```

Next comes the code that handles the `egrep` specific behavior. If no pattern was supplied with '`-e`', the first non-option on the command line is used. The `awk` command line arguments up to `ARGV[Optind]` are cleared, so that `awk` won't try to process them as files. If no files were specified, the standard input is used, and if multiple files were specified, we make sure to note this so that the file names can precede the matched lines in the output.

The last two lines are commented out, since they are not needed in `gawk`. They should be uncommented if you have to use another version of `awk`.

```
        if (pattern == "")
            pattern = ARGV[Optind++]

        for (i = 1; i < Optind; i++)
            ARGV[i] = ""
        if (Optind >= ARGC) {
            ARGV[1] = "-"
            ARGC = 2
        } else if (ARGC - Optind > 1)
            do_filenames++

    #    if (IGNORECASE)
    #        pattern = tolower(pattern)
    }
```

The next set of lines should be uncommented if you are not using `gawk`. This rule translates all the characters in the input line into lower-case if the '`-i`' option was specified. The rule is commented out since it is not necessary with `gawk`.

```
    #{
    #    if (IGNORECASE)
    #        $0 = tolower($0)
    #}
```

The `beginfile` function is called by the rule in '`ftrans.awk`' when each new file is processed. In this case, it is very simple; all it does is initialize a variable `fcount` to zero. `fcount` tracks how many lines in the current file matched the pattern.

```
function beginfile(junk)
{
    fcount = 0
}
```

The `endfile` function is called after each file has been processed. It is used only when the user wants a count of the number of lines that matched. `no_print` will be true only if the exit status is desired. `count_only` will be true if line counts are desired. `egrep` will therefore only print line counts if printing and counting are enabled. The output format must be adjusted depending upon the number of files to be processed. Finally, `fcount` is added to `total`, so that we know how many lines altogether matched the pattern.

```
function endfile(file)
{
    if (! no_print && count_only)
        if (do_filenames)
            print file ":" fcount
        else
            print fcount

    total += fcount
}
```

This rule does most of the work of matching lines. The variable `matches` will be true if the line matched the pattern. If the user wants lines that did not match, the sense of the `matches` is inverted using the '`!`' operator. `fcount` is incremented with the value of `matches`, which will be either one or zero, depending upon a successful or unsuccessful match. If the line did not match, the `next` statement just moves on to the next record.

There are several optimizations for performance in the following few lines of code. If the user only wants exit status (`no_print` is true), and we don't have to count lines, then it is enough to know that one line in this file matched, and we can skip on to the next file with `nextfile`. Along similar lines, if we are only printing file names, and we don't need to count lines, we can print the file name, and then skip to the next file with `nextfile`.

Finally, each line is printed, with a leading filename and colon if necessary.

```
{
    matches = ($0 ~ pattern)
    if (invert)
        matches = ! matches

    fcount += matches    # 1 or 0

    if (! matches)
        next
```

```
        if (no_print && ! count_only)
            nextfile

        if (filenames_only && ! count_only) {
            print FILENAME
            nextfile
        }

        if (do_filenames && ! count_only)
            print FILENAME ":" $0
        else if (! count_only)
            print
    }
```

The `END` rule takes care of producing the correct exit status. If there were no matches, the exit status is one, otherwise it is zero.

```
    END    \
    {
        if (total == 0)
            exit 1
        exit 0
    }
```

The `usage` function prints a usage message in case of invalid options and then exits.

```
    function usage(    e)
    {
        e = "Usage: egrep [-csvil] [-e pat] [files ...]"
        print e > "/dev/stderr"
        exit 1
    }
```

The variable `e` is used so that the function fits nicely on the printed page.

Just a note on programming style. You may have noticed that the `END` rule uses backslash continuation, with the open brace on a line by itself. This is so that it more closely resembles the way functions are written. Many of the examples in this chapter use this style. You can decide for yourself if you like writing your `BEGIN` and `END` rules this way, or not.

16.1.3. Printing Out User Information

The `id` utility lists a user's real and effective user-id numbers, real and effective group-id numbers, and the user's group set, if any. `id` will only print the effective user-id and group-id if they are different from the real ones. If possible, `id` will also supply the corresponding user and group names. The output might look like this:

```
    $ id
```

```
⊣ uid=2076(arnold) gid=10(staff) groups=10(staff),4(tty)
```

This information is exactly what is provided by gawk's '/dev/user' special file (lásd 6.7. Bekezdés [Special File Names in gawk], 71. oldal). However, the id utility provides a more palatable output than just a string of numbers.

Here is a simple version of id written in awk. It uses the user database library functions (lásd 15.11. Bekezdés [Reading the User Database], 192. oldal), and the group database library functions (lásd 15.12. Bekezdés [Reading the Group Database], 197. oldal).

The program is fairly straightforward. All the work is done in the BEGIN rule. The user and group id numbers are obtained from '/dev/user'. If there is no support for '/dev/user', the program gives up.

The code is repetitive. The entry in the user database for the real user-id number is split into parts at the ':'. The name is the first field. Similar code is used for the effective user-id number, and the group numbers.

```
# id.awk --- implement id in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# output is:
# uid=12(foo) euid=34(bar) gid=3(baz) \
#             egid=5(blat) groups=9(nine),2(two),1(one)

BEGIN    \
{
    if ((getline < "/dev/user") < 0) {
        err = "id: no /dev/user support - cannot run"
        print err > "/dev/stderr"
        exit 1
    }
    close("/dev/user")

    uid = $1
    euid = $2
    gid = $3
    egid = $4

    printf("uid=%d", uid)
    pw = getpwuid(uid)
    if (pw != "") {
        split(pw, a, ":")
        printf("(%s)", a[1])
    }
```

```
    if (euid != uid) {
        printf(" euid=%d", euid)
        pw = getpwuid(euid)
        if (pw != "") {
            split(pw, a, ":")
            printf("(%s)", a[1])
        }
    }

    printf(" gid=%d", gid)
    pw = getgrgid(gid)
    if (pw != "") {
        split(pw, a, ":")
        printf("(%s)", a[1])
    }

    if (egid != gid) {
        printf(" egid=%d", egid)
        pw = getgrgid(egid)
        if (pw != "") {
            split(pw, a, ":")
            printf("(%s)", a[1])
        }
    }

    if (NF > 4) {
        printf(" groups=");
        for (i = 5; i <= NF; i++) {
            printf("%d", $i)
            pw = getgrgid($i)
            if (pw != "") {
                split(pw, a, ":")
                printf("(%s)", a[1])
            }
            if (i < NF)
                printf(",")
        }
    }
    print ""
}
```

16.1.4. Splitting a Large File Into Pieces

The `split` program splits large text files into smaller pieces. By default, the output files are named '`xaa`', '`xab`', and so on. Each file has 1000 lines in it, with the likely exception of the last file. To change the number of lines in each file, you supply a number on the command line preceded with a minus, e.g., '`-500`' for files with 500 lines in them instead of 1000. To change the name of the output files to something like '`myfileaa`', '`myfileab`', and so on, you supply an additional argument that specifies the filename.

Here is a version of `split` in `awk`. It uses the `ord` and `chr` functions presented in 15.5. Bekezdés [Translating Between Characters and Numbers], 174. oldal.

The program first sets its defaults, and then tests to make sure there are not too many arguments. It then looks at each argument in turn. The first argument could be a minus followed by a number. If it is, this happens to look like a negative number, so it is made positive, and that is the count of lines. The data file name is skipped over, and the final argument is used as the prefix for the output file names.

```
# split.awk --- do split in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# usage: split [-num] [file] [outname]

BEGIN {
    outfile = "x"    # default
    count = 1000
    if (ARGC > 4)
        usage()

    i = 1
    if (ARGV[i] ~ /^-[0-9]+$/) {
        count = -ARGV[i]
        ARGV[i] = ""
        i++
    }
    # test argv in case reading from stdin instead of file
    if (i in ARGV)
        i++    # skip data file name
    if (i in ARGV) {
        outfile = ARGV[i]
        ARGV[i] = ""
    }
```

```
        s1 = s2 = "a"
        out = (outfile s1 s2)
    }
```

The next rule does most of the work. `tcount` (temporary count) tracks how many lines have been printed to the output file so far. If it is greater than `count`, it is time to close the current file and start a new one. `s1` and `s2` track the current suffixes for the file name. If they are both '`z`', the file is just too big. Otherwise, `s1` moves to the next letter in the alphabet and `s2` starts over again at '`a`'.

```
    {
        if (++tcount > count) {
            close(out)
            if (s2 == "z") {
                if (s1 == "z") {
                    printf("split: %s is too large to split\n", \
                           FILENAME) > "/dev/stderr"
                    exit 1
                }
                s1 = chr(ord(s1) + 1)
                s2 = "a"
            } else
                s2 = chr(ord(s2) + 1)
            out = (outfile s1 s2)
            tcount = 1
        }
        print > out
    }
```

The `usage` function simply prints an error message and exits.

```
    function usage(   e)
    {
        e = "usage: split [-num] [file] [outname]"
        print e > "/dev/stderr"
        exit 1
    }
```

The variable `e` is used so that the function fits nicely on the page.

This program is a bit sloppy; it relies on `awk` to close the last file for it automatically, instead of doing it in an `END` rule.

16.1.5. Duplicating Output Into Multiple Files

The `tee` program is known as a "pipe fitting." `tee` copies its standard input to its standard output, and also duplicates it to the files named on the command line. Its usage is:

```
    tee [-a] file ...
```

The '-a' option tells tee to append to the named files, instead of truncating them and starting over.

The BEGIN rule first makes a copy of all the command line arguments, into an array named copy. ARGV[0] is not copied, since it is not needed. tee cannot use ARGV directly, since awk will attempt to process each file named in ARGV as input data.

If the first argument is '-a', then the flag variable append is set to true, and both ARGV[1] and copy[1] are deleted. If ARGC is less than two, then no file names were supplied, and tee prints a usage message and exits. Finally, awk is forced to read the standard input by setting ARGV[1] to "-", and ARGC to two.

```
# tee.awk --- tee in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993
# Revised December 1995
BEGIN    \
{
    for (i = 1; i < ARGC; i++)
        copy[i] = ARGV[i]
    if (ARGV[1] == "-a") {
        append = 1
        delete ARGV[1]
        delete copy[1]
        ARGC--
    }
    if (ARGC < 2) {
        print "usage: tee [-a] file ..." > "/dev/stderr"
        exit 1
    }
    ARGV[1] = "-"
    ARGC = 2
}
```

The single rule does all the work. Since there is no pattern, it is executed for each line of input. The body of the rule simply prints the line into each file on the command line, and then to the standard output.

```
{
    # moving the if outside the loop makes it run faster
    if (append)
        for (i in copy)
            print >> copy[i]
    else
        for (i in copy)
            print > copy[i]
    print
}
```

It would have been possible to code the loop this way:

```
for (i in copy)
    if (append)
        print >> copy[i]
    else
        print > copy[i]
```

This is more concise, but it is also less efficient. The '`if`' is tested for each record and for each output file. By duplicating the loop body, the '`if`' is only tested once for each input record. If there are N input records and M input files, the first method only executes N '`if`' statements, while the second would execute $N*M$ '`if`' statements.

Finally, the `END` rule cleans up, by closing all the output files.

```
END    \
{
    for (i in copy)
        close(copy[i])
}
```

16.1.6. Printing Non-duplicated Lines of Text

The `uniq` utility reads sorted lines of data on its standard input, and (by default) removes duplicate lines. In other words, only unique lines are printed, hence the name. `uniq` has a number of options. The usage is:

`uniq` [`-udc` [`-`*n*]] [`+`*n*] [*input file* [*output file*]]

The option meanings are:

`-d` Only print repeated lines.

`-u` Only print non-repeated lines.

`-c` Count lines. This option overrides '`-d`' and '`-u`'. Both repeated and non-repeated lines are counted.

`-`*n* Skip *n* fields before comparing lines. The definition of fields is similar to `awk`'s default: non-whitespace characters separated by runs of spaces and/or tabs.

`+n` Skip *n* characters before comparing lines. Any fields specified with '-*n*' are skipped first.

input file Data is read from the input file named on the command line, instead of from the standard input.

output file The generated output is sent to the named output file, instead of to the standard output.

Normally `uniq` behaves as if both the '-d' and '-u' options had been provided.

Here is an `awk` implementation of `uniq`. It uses the `getopt` library function (lásd 15.10. Bekezdés [Processing Command Line Options], 186. oldal), and the `join` library function (lásd 15.6. Bekezdés [Merging an Array Into a String], 176. oldal).

The program begins with a `usage` function and then a brief outline of the options and their meanings in a comment.

The `BEGIN` rule deals with the command line arguments and options. It uses a trick to get `getopt` to handle options of the form '-25', treating such an option as the option letter '2' with an argument of '5'. If indeed two or more digits were supplied (`Optarg` looks like a number), `Optarg` is concatenated with the option digit, and then result is added to zero to make it into a number. If there is only one digit in the option, then `Optarg` is not needed, and `Optind` must be decremented so that `getopt` will process it next time. This code is admittedly a bit tricky.

If no options were supplied, then the default is taken, to print both repeated and non-repeated lines. The output file, if provided, is assigned to `outputfile`. Earlier, `outputfile` was initialized to the standard output, '/dev/stdout'.

```
# uniq.awk --- do uniq in awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

function usage(    e)
{
    e = "Usage: uniq [-udc [-n]] [+n] [ in [ out ]]"
    print e > "/dev/stderr"
    exit 1
}

# -c    count lines. overrides -d and -u
# -d    only repeated lines
# -u    only non-repeated lines
# -n    skip n fields
# +n    skip n characters, skip fields first

BEGIN    \
```

```
{
    count = 1
    outputfile = "/dev/stdout"
    opts = "udc0:1:2:3:4:5:6:7:8:9:"
    while ((c = getopt(ARGC, ARGV, opts)) != -1) {
        if (c == "u")
            non_repeated_only++
        else if (c == "d")
            repeated_only++
        else if (c == "c")
            do_count++
        else if (index("0123456789", c) != 0) {
            # getopt requires args to options
            # this messes us up for things like -5
            if (Optarg ~ /^[0-9]+$/)
                fcount = (c Optarg) + 0
            else {
                fcount = c + 0
                Optind--
            }
        } else
            usage()
    }

    if (ARGV[Optind] ~ /^\+[0-9]+$/) {
        charcount = substr(ARGV[Optind], 2) + 0
        Optind++
    }

    for (i = 1; i < Optind; i++)
        ARGV[i] = ""

    if (repeated_only == 0 && non_repeated_only == 0)
        repeated_only = non_repeated_only = 1

    if (ARGC - Optind == 2) {
        outputfile = ARGV[ARGC - 1]
        ARGV[ARGC - 1] = ""
    }
}
```

The following function, `are_equal`, compares the current line, `$0`, to the previous line, `last`. It handles skipping fields and characters.

If no field count and no character count were specified, `are_equal` simply returns one or zero depending upon the result of a simple string comparison of `last` and `$0`. Otherwise, things get more complicated.

If fields have to be skipped, each line is broken into an array using `split` (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal), and then the desired fields are joined back into a line using `join`. The joined lines are stored in `clast` and `cline`. If no fields are skipped, `clast` and `cline` are set to `last` and `$0` respectively.

Finally, if characters are skipped, `substr` is used to strip off the leading `charcount` characters in `clast` and `cline`. The two strings are then compared, and `are_equal` returns the result.

```
function are_equal(    n, m, clast, cline, alast, aline)
{
    if (fcount == 0 && charcount == 0)
        return (last == $0)

    if (fcount > 0) {
        n = split(last, alast)
        m = split($0, aline)
        clast = join(alast, fcount+1, n)
        cline = join(aline, fcount+1, m)
    } else {
        clast = last
        cline = $0
    }
    if (charcount) {
        clast = substr(clast, charcount + 1)
        cline = substr(cline, charcount + 1)
    }

    return (clast == cline)
}
```

The following two rules are the body of the program. The first one is executed only for the very first line of data. It sets `last` equal to `$0`, so that subsequent lines of text have something to be compared to.

The second rule does the work. The variable `equal` will be one or zero depending upon the results of `are_equal`'s comparison. If `uniq` is counting repeated lines, then the `count` variable is incremented if the lines are equal. Otherwise the line is printed and `count` is reset, since the two lines are not equal.

If `uniq` is not counting, `count` is incremented if the lines are equal. Otherwise, if `uniq` is counting repeated lines, and more than one line has been seen, or if `uniq` is counting non-repeated lines, and only one line has been seen, then the line is printed, and `count` is reset.

Finally, similar logic is used in the `END` rule to print the final line of input data.

```
NR == 1 {
    last = $0
    next
}

{
    equal = are_equal()

    if (do_count) {    # overrides -d and -u
        if (equal)
            count++
        else {
            printf("%4d %s\n", count, last) > outputfile
            last = $0
            count = 1    # reset
        }
        next
    }

    if (equal)
        count++
    else {
        if ((repeated_only && count > 1) ||
            (non_repeated_only && count == 1))
                print last > outputfile
        last = $0
        count = 1
    }
}

END {
    if (do_count)
        printf("%4d %s\n", count, last) > outputfile
    else if ((repeated_only && count > 1) ||
            (non_repeated_only && count == 1))
        print last > outputfile
}
```

16.1.7. Counting Things

The `wc` (word count) utility counts lines, words, and characters in one or more input files. Its usage is:

`wc` [`-lwc`] [*files* ...]

If no files are specified on the command line, wc reads its standard input. If there are multiple files, it will also print total counts for all the files. The options and their meanings are:

`-l` Only count lines.

`-w` Only count words. A “word” is a contiguous sequence of non-whitespace characters, separated by spaces and/or tabs. Happily, this is the normal way awk separates fields in its input data.

`-c` Only count characters.

Implementing wc in awk is particularly elegant, since awk does a lot of the work for us; it splits lines into words (i.e. fields) and counts them, it counts lines (i.e. records) for us, and it can easily tell us how long a line is.

This version uses the getopt library function (lásd 15.10. Bekezdés [Processing Command Line Options], 186. oldal), and the file transition functions (lásd 15.9. Bekezdés [Noting Data File Boundaries], 184. oldal).

This version has one major difference from traditional versions of wc. Our version always prints the counts in the order lines, words, and characters. Traditional versions note the order of the ‘-l’, ‘-w’, and ‘-c’ options on the command line, and print the counts in that order.

The BEGIN rule does the argument processing. The variable print_total will be true if more than one file was named on the command line.

```
# wc.awk --- count lines, words, characters
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Options:
#    -l    only count lines
#    -w    only count words
#    -c    only count characters
#
# Default is to count lines, words, characters

BEGIN {
    # let getopt print a message about
    # invalid options. we ignore them
    while ((c = getopt(ARGC, ARGV, "lwc")) != -1) {
        if (c == "l")
            do_lines = 1
        else if (c == "w")
            do_words = 1
        else if (c == "c")
            do_chars = 1
    }
    for (i = 1; i < Optind; i++)
```

```
            ARGV[i] = ""

        # if no options, do all
        if (! do_lines && ! do_words && ! do_chars)
            do_lines = do_words = do_chars = 1

        print_total = (ARGC - i > 2)
    }
```

The `beginfile` function is simple; it just resets the counts of lines, words, and characters to zero, and saves the current file name in `fname`.

The `endfile` function adds the current file's numbers to the running totals of lines, words, and characters. It then prints out those numbers for the file that was just read. It relies on `beginfile` to reset the numbers for the following data file.

```
function beginfile(file)
{
    chars = lines = words = 0
    fname = FILENAME
}

function endfile(file)
{
    tchars += chars
    tlines += lines
    twords += words
    if (do_lines)
        printf "\t%d", lines
    if (do_words)
        printf "\t%d", words
    if (do_chars)
        printf "\t%d", chars
    printf "\t%s\n", fname
}
```

There is one rule that is executed for each line. It adds the length of the record to `chars`. It has to add one, since the newline character separating records (the value of `RS`) is not part of the record itself. `lines` is incremented for each line read, and `words` is incremented by the value of `NF`, the number of "words" on this line.[1]

Finally, the `END` rule simply prints the totals for all the files.

```
# do per line
{
```

[1] Examine the code in 15.9. Bekezdés [Noting Data File Boundaries], 184. oldal. Why must `wc` use a separate `lines` variable, instead of using the value of `FNR` in `endfile`?

```
    chars += length($0) + 1    # get newline
    lines++
    words += NF
}

END {
    if (print_total) {
        if (do_lines)
            printf "\t%d", tlines
        if (do_words)
            printf "\t%d", twords
        if (do_chars)
            printf "\t%d", tchars
        print "\ttotal"
    }
}
```

16.2. A Grab Bag of awk Programs

This section is a large "grab bag" of miscellaneous programs. We hope you find them both interesting and enjoyable.

16.2.1. Finding Duplicated Words in a Document

A common error when writing large amounts of prose is to accidentally duplicate words. Often you will see this in text as something like "the the program does the following ...." When the text is on-line, often the duplicated words occur at the end of one line and the beginning of another, making them very difficult to spot.

This program, '`dupword.awk`', scans through a file one line at a time, and looks for adjacent occurrences of the same word. It also saves the last word on a line (in the variable `prev`) for comparison with the first word on the next line.

The first two statements make sure that the line is all lower-case, so that, for example, "The" and "the" compare equal to each other. The second statement removes all non-alphanumeric and non-whitespace characters from the line, so that punctuation does not affect the comparison either. This sometimes leads to reports of duplicated words that really are different, but this is unusual.

```
# dupword --- find duplicate words in text
# Arnold Robbins, arnold@gnu.org, Public Domain
# December 1991

{
    $0 = tolower($0)
    gsub(/[^A-Za-z0-9 \t]/, "");
    if ($1 == prev)
        printf("%s:%d: duplicate %s\n",
            FILENAME, FNR, $1)
    for (i = 2; i <= NF; i++)
        if ($i == $(i-1))
            printf("%s:%d: duplicate %s\n",
                FILENAME, FNR, $i)
    prev = $NF
}
```

16.2.2. An Alarm Clock Program

The following program is a simple "alarm clock" program. You give it a time of day, and an optional message. At the given time, it prints the message on the standard output. In addition, you can give it the number of times to repeat the message, and also a delay between repetitions.

This program uses the `gettimeofday` function from 15.8. Bekezdés [Managing the Time of Day], 182. oldal.

All the work is done in the `BEGIN` rule. The first part is argument checking and setting of defaults; the delay, the count, and the message to print. If the user supplied a message, but it does not contain the ASCII BEL character (known as the "alert" character, '`\a`'), then it is added to the message. (On many systems, printing the ASCII BEL generates some sort of audible alert. Thus, when the alarm goes off, the system calls attention to itself, in case the user is not looking at their computer or terminal.)

```
# alarm --- set an alarm
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# usage: alarm time [ "message" [ count [ delay ] ] ]

BEGIN    \
{
    # Initial argument sanity checking
    usage1 = "usage: alarm time ['message' [count [delay]]]"
    usage2 = sprintf("\t(%s) time ::= hh:mm", ARGV[1])
```

```
    if (ARGC < 2) {
        print usage > "/dev/stderr"
        exit 1
    } else if (ARGC == 5) {
        delay = ARGV[4] + 0
        count = ARGV[3] + 0
        message = ARGV[2]
    } else if (ARGC == 4) {
        count = ARGV[3] + 0
        message = ARGV[2]
    } else if (ARGC == 3) {
        message = ARGV[2]
    } else if (ARGV[1] !~ /[0-9]?[0-9]:[0-9][0-9]/) {
        print usage1 > "/dev/stderr"
        print usage2 > "/dev/stderr"
        exit 1
    }

    # set defaults for once we reach the desired time
    if (delay == 0)
        delay = 180    # 3 minutes
    if (count == 0)
        count = 5
    if (message == "")
        message = sprintf("\aIt is now %s!\a", ARGV[1])
    else if (index(message, "\a") == 0)
        message = "\a" message "\a"
```

The next section of code turns the alarm time into hours and minutes, and converts it if necessary to a 24-hour clock. Then it turns that time into a count of the seconds since midnight. Next it turns the current time into a count of seconds since midnight. The difference between the two is how long to wait before setting off the alarm.

```
    # split up dest time
    split(ARGV[1], atime, ":")
    hour = atime[1] + 0    # force numeric
    minute = atime[2] + 0  # force numeric

    # get current broken down time
    gettimeofday(now)

    # if time given is 12-hour hours and it's after that
    # hour, e.g., `alarm 5:30' at 9 a.m. means 5:30 p.m.,
    # then add 12 to real hour
    if (hour < 12 && now["hour"] > hour)
```

```
            hour += 12

        # set target time in seconds since midnight
        target = (hour * 60 * 60) + (minute * 60)

        # get current time in seconds since midnight
        current = (now["hour"] * 60 * 60) + \
                   (now["minute"] * 60) + now["second"]

        # how long to sleep for
        naptime = target - current
        if (naptime <= 0) {
            print "time is in the past!" > "/dev/stderr"
            exit 1
        }
```

Finally, the program uses the `system` function (lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal) to call the `sleep` utility. The `sleep` utility simply pauses for the given number of seconds. If the exit status is not zero, the program assumes that `sleep` was interrupted, and exits. If `sleep` exited with an OK status (zero), then the program prints the message in a loop, again using `sleep` to delay for however many seconds are necessary.

```
        # zzzzzz..... go away if interrupted
        if (system(sprintf("sleep %d", naptime)) != 0)
            exit 1

        # time to notify!
        command = sprintf("sleep %d", delay)
        for (i = 1; i <= count; i++) {
            print message
            # if sleep command interrupted, go away
            if (system(command) != 0)
                break
        }

        exit 0
    }
```

16.2.3. Transliterating Characters

The system `tr` utility transliterates characters. For example, it is often used to map upper-case letters into lower-case, for further processing.

generate data `| tr '[A-Z]' '[a-z]' |` *process data* ...

You give tr two lists of characters enclosed in square brackets. Usually, the lists are quoted to keep the shell from attempting to do a filename expansion.[2] When processing the input, the first character in the first list is replaced with the first character in the second list, the second character in the first list is replaced with the second character in the second list, and so on. If there are more characters in the "from" list than in the "to" list, the last character of the "to" list is used for the remaining characters in the "from" list.

Some time ago, a user proposed to us that we add a transliteration function to gawk. Being opposed to "creeping featurism," I wrote the following program to prove that character transliteration could be done with a user-level function. This program is not as complete as the system tr utility, but it will do most of the job.

The translate program demonstrates one of the few weaknesses of standard awk: dealing with individual characters is very painful, requiring repeated use of the substr, index, and gsub built-in functions (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).[3]

There are two functions. The first, stranslate, takes three arguments.

from
: A list of characters to translate from.

to
: A list of characters to translate to.

target
: The string to do the translation on.

Associative arrays make the translation part fairly easy. t_ar holds the "to" characters, indexed by the "from" characters. Then a simple loop goes through from, one character at a time. For each character in from, if the character appears in target, gsub is used to change it to the corresponding to character.

The translate function simply calls stranslate using $0 as the target. The main program sets two global variables, FROM and TO, from the command line, and then changes ARGV so that awk will read from the standard input.

Finally, the processing rule simply calls translate for each record.

```
# translate --- do tr like stuff
# Arnold Robbins, arnold@gnu.org, Public Domain
# August 1989

# bugs: does not handle things like: tr A-Z a-z, it has
# to be spelled out. However, if ‘to’ is shorter than ‘from’,
# the last character in ‘to’ is used for the rest of ‘from’.
```

[2] On older, non-POSIX systems, tr often does not require that the lists be enclosed in square brackets and quoted. This is a feature.

[3] This program was written before gawk acquired the ability to split each character in a string into separate array elements. How might this ability simplify the program?

```
function stranslate(from, to, target,     lf, lt, t_ar, i, c)
{
    lf = length(from)
    lt = length(to)
    for (i = 1; i <= lt; i++)
        t_ar[substr(from, i, 1)] = substr(to, i, 1)
    if (lt < lf)
        for (; i <= lf; i++)
            t_ar[substr(from, i, 1)] = substr(to, lt, 1)
    for (i = 1; i <= lf; i++) {
        c = substr(from, i, 1)
        if (index(target, c) > 0)
            gsub(c, t_ar[c], target)
    }
    return target
}

function translate(from, to)
{
    return $0 = stranslate(from, to, $0)
}

# main program
BEGIN {
    if (ARGC < 3) {
        print "usage: translate from to" > "/dev/stderr"
        exit
    }
    FROM = ARGV[1]
    TO = ARGV[2]
    ARGC = 2
    ARGV[1] = "-"
}

{
    translate(FROM, TO)
    print
}
```

While it is possible to do character transliteration in a user-level function, it is not necessarily efficient, and we started to consider adding a built-in function. However, shortly after writing this program, we learned that the System V Release 4 `awk` had added the `toupper` and `tolower` functions. These functions handle the vast majority of the cases where character transliteration is necessary, and so we chose to simply add those functions to `gawk` as well, and then leave well enough alone.

An obvious improvement to this program would be to set up the `t_ar` array only once, in a `BEGIN` rule. However, this assumes that the "from" and "to" lists will never change throughout the lifetime of the program.

16.2.4. Printing Mailing Labels

Here is a "real world"[4] program. This script reads lists of names and addresses, and generates mailing labels. Each page of labels has 20 labels on it, two across and ten down. The addresses are guaranteed to be no more than five lines of data. Each address is separated from the next by a blank line.

The basic idea is to read 20 labels worth of data. Each line of each label is stored in the `line` array. The single rule takes care of filling the `line` array and printing the page when 20 labels have been read.

The `BEGIN` rule simply sets `RS` to the empty string, so that `awk` will split records at blank lines (lásd 5.1. Bekezdés [How Input is Split into Records], 37. oldal). It sets `MAXLINES` to 100, since `MAXLINE` is the maximum number of lines on the page (20 * 5 = 100).

Most of the work is done in the `printpage` function. The label lines are stored sequentially in the `line` array. But they have to be printed horizontally; `line[1]` next to `line[6]`, `line[2]` next to `line[7]`, and so on. Two loops are used to accomplish this. The outer loop, controlled by `i`, steps through every 10 lines of data; this is each row of labels. The inner loop, controlled by `j`, goes through the lines within the row. As `j` goes from zero to four, '`i+j`' is the `j`'th line in the row, and '`i+j+5`' is the entry next to it. The output ends up looking something like this:

```
line 1          line 6
line 2          line 7
line 3          line 8
line 4          line 9
line 5          line 10
```

As a final note, at lines 21 and 61, an extra blank line is printed, to keep the output lined up on the labels. This is dependent on the particular brand of labels in use when the program was written. You will also note that there are two blank lines at the top and two blank lines at the bottom.

The `END` rule arranges to flush the final page of labels; there may not have been an even multiple of 20 labels in the data.

```
# labels.awk
# Arnold Robbins, arnold@gnu.org, Public Domain
# June 1992

# Program to print labels.  Each label is 5 lines of data
```

[4] "Real world" is defined as "a program actually used to get something done."

```
# that may have blank lines.  The label sheets have 2
# blank lines at the top and 2 at the bottom.

BEGIN    { RS = "" ; MAXLINES = 100 }

function printpage(    i, j)
{
    if (Nlines <= 0)
        return

    printf "\n\n"        # header

    for (i = 1; i <= Nlines; i += 10) {
        if (i == 21 || i == 61)
            print ""
        for (j = 0; j < 5; j++) {
            if (i + j > MAXLINES)
                break
            printf "   %-41s %s\n", line[i+j], line[i+j+5]
        }
        print ""
    }

    printf "\n\n"        # footer

    for (i in line)
        line[i] = ""
}

# main rule
{
    if (Count >= 20) {
        printpage()
        Count = 0
        Nlines = 0
    }
    n = split($0, a, "\n")
    for (i = 1; i <= n; i++)
        line[++Nlines] = a[i]
    for (; i <= 5; i++)
        line[++Nlines] = ""
    Count++
}
```

```
END    \
{
    printpage()
}
```

16.2.5. Generating Word Usage Counts

The following `awk` program prints the number of occurrences of each word in its input. It illustrates the associative nature of `awk` arrays by using strings as subscripts. It also demonstrates the '`for` *x* `in` *array*' construction. Finally, it shows how `awk` can be used in conjunction with other utility programs to do a useful task of some complexity with a minimum of effort. Some explanations follow the program listing.

```
awk '
# Print list of word frequencies
{
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}'
```

The first thing to notice about this program is that it has two rules. The first rule, because it has an empty pattern, is executed on every line of the input. It uses `awk`'s field-accessing mechanism (lásd 5.2. Bekezdés [Examining Fields], 40. oldal) to pick out the individual words from the line, and the built-in variable `NF` (lásd 10. Fejezet [Built-in Variables], 115. oldal) to know how many fields are available.

For each input word, an element of the array `freq` is incremented to reflect that the word has been seen an additional time.

The second rule, because it has the pattern `END`, is not executed until the input has been exhausted. It prints out the contents of the `freq` table that has been built up inside the first action.

This program has several problems that would prevent it from being useful by itself on real text files:

- Words are detected using the `awk` convention that fields are separated by whitespace and that other characters in the input (except newlines) don't have any special meaning to `awk`. This means that punctuation characters count as part of words.
- The `awk` language considers upper- and lower-case characters to be distinct. Therefore, '`bartender`' and '`Bartender`' are not treated as the same word. This is undesirable since, in normal text, words are capi-

talized if they begin sentences, and a frequency analyzer should not be sensitive to capitalization.

- The output does not come out in any useful order. You're more likely to be interested in which words occur most frequently, or having an alphabetized table of how frequently each word occurs.

The way to solve these problems is to use some of the more advanced features of the `awk` language. First, we use `tolower` to remove case distinctions. Next, we use `gsub` to remove punctuation characters. Finally, we use the system `sort` utility to process the output of the `awk` script. Here is the new version of the program:

```
# Print list of word frequencies
{
    $0 = tolower($0)    # remove case distinctions
    gsub(/[^a-z0-9_ \t]/, "", $0)  # remove punctuation
    for (i = 1; i <= NF; i++)
        freq[$i]++
}

END {
    for (word in freq)
        printf "%s\t%d\n", word, freq[word]
}
```

Assuming we have saved this program in a file named '`wordfreq.awk`', and that the data is in '`file1`', the following pipeline

```
awk -f wordfreq.awk file1 | sort +1 -nr
```

produces a table of the words appearing in '`file1`' in order of decreasing frequency.

The `awk` program suitably massages the data and produces a word frequency table, which is not ordered.

The `awk` script's output is then sorted by the `sort` utility and printed on the terminal. The options given to `sort` in this example specify to sort using the second field of each input line (skipping one field), that the sort keys should be treated as numeric quantities (otherwise '`15`' would come before '`5`'), and that the sorting should be done in descending (reverse) order.

We could have even done the `sort` from within the program, by changing the `END` action to:

```
END {
    sort = "sort +1 -nr"
    for (word in freq)
        printf "%s\t%d\n", word, freq[word] | sort
    close(sort)
}
```

You would have to use this way of sorting on systems that do not have true pipes.

See the general operating system documentation for more information on how to use the `sort` program.

16.2.6. Removing Duplicates from Unsorted Text

The `uniq` program (lásd 16.1.6. Bekezdés [Printing Non-duplicated Lines of Text], 220. oldal), removes duplicate lines from *sorted* data.

Suppose, however, you need to remove duplicate lines from a data file, but that you wish to preserve the order the lines are in? A good example of this might be a shell history file. The history file keeps a copy of all the commands you have entered, and it is not unusual to repeat a command several times in a row. Occasionally you might wish to compact the history by removing duplicate entries. Yet it is desirable to maintain the order of the original commands.

This simple program does the job. It uses two arrays. The `data` array is indexed by the text of each line. For each line, `data[$0]` is incremented.

If a particular line has not been seen before, then `data[$0]` will be zero. In that case, the text of the line is stored in `lines[count]`. Each element of `lines` is a unique command, and the indices of `lines` indicate the order in which those lines were encountered. The `END` rule simply prints out the lines, in order.

```
# histsort.awk --- compact a shell history file
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

# Thanks to Byron Rakitzis for the general idea
{
    if (data[$0]++ == 0)
        lines[++count] = $0
}

END {
    for (i = 1; i <= count; i++)
        print lines[i]
}
```

This program also provides a foundation for generating other useful information. For example, using the following `print` satement in the `END` rule would indicate how often a particular command was used.

```
print data[lines[i]], lines[i]
```

This works because `data[$0]` was incremented each time a line was seen.

16.2.7. Extracting Programs from Texinfo Source Files

Both this chapter and the previous chapter (15. Fejezet [A Library of `awk` Functions], 169. oldal), present a large number of `awk` programs. If you wish to experiment with these programs, it is tedious to have to type them in by hand. Here we present a program that can extract parts of a Texinfo input file into separate files.

This könyv is written in Texinfo, the GNU project's document formatting language. A single Texinfo source file can be used to produce both printed and on-line documentation. Texinfo is fully documented in *Texinfo—The GNU Documentation Format*, available from the Free Software Foundation.

For our purposes, it is enough to know three things about Texinfo input files.

- The "at" symbol, '`@`', is special in Texinfo, much like '`\`' in C or `awk`. Literal '`@`' symbols are represented in Texinfo source files as '`@@`'.
- Comments start with either '`@c`' or '`@comment`'. The file extraction program will work by using special comments that start at the beginning of a line.
- Example text that should not be split across a page boundary is bracketed between lines containing '`@group`' and '`@end group`' commands.

The following program, '`extract.awk`', reads through a Texinfo source file, and does two things, based on the special comments. Upon seeing '`@c system ...`', it runs a command, by extracting the command text from the control line and passing it on to the `system` function (lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal). Upon seeing '`@c file` *filename*', each subsequent line is sent to the file *filename*, until '`@c endfile`' is encountered. The rules in '`extract.awk`' will match either '`@c`' or '`@comment`' by letting the '`omment`' part be optional. Lines containing '`@group`' and '`@end group`' are simply removed. '`extract.awk`' uses the `join` library function (lásd 15.6. Bekezdés [Merging an Array Into a String], 176. oldal).

The example programs in the on-line Texinfo source for *Hatékony AWK programozás* ('`gawk.texi`') have all been bracketed inside '`file`', and '`endfile`' lines. The `gawk` distribution uses a copy of '`extract.awk`' to extract the sample programs and install many of them in a standard directory, where `gawk` can find them. The Texinfo file looks something like this:

```
...
This program has a @code{BEGIN} block,
which prints a nice message:

@example
@c file examples/messages.awk
BEGIN @{ print "Don't panic!" @}
@c end file
```

```
@end example

It also prints some final advice:

@example
@c file examples/messages.awk
END @{ print "Always avoid bored archeologists!" @}
@c end file
@end example
...
```

'extract.awk' begins by setting IGNORECASE to one, so that mixed upper-case and lower-case letters in the directives won't matter.

The first rule handles calling system, checking that a command was given (NF is at least three), and also checking that the command exited with a zero exit status, signifying OK.

```
# extract.awk --- extract files and run programs
#                 from texinfo files
# Arnold Robbins, arnold@gnu.org, Public Domain
# May 1993

BEGIN    { IGNORECASE = 1 }

/^@c(omment)?[ \t]+system/    \
{
    if (NF < 3) {
        e = (FILENAME ":" FNR)
        e = (e  ": badly formed `system' line")
        print e > "/dev/stderr"
        next
    }
    $1 = ""
    $2 = ""
    stat = system($0)
    if (stat != 0) {
        e = (FILENAME ":" FNR)
        e = (e ": warning: system returned " stat)
        print e > "/dev/stderr"
    }
}
```

The variable e is used so that the function fits nicely on the page.

The second rule handles moving data into files. It verifies that a file name was given in the directive. If the file named is not the current file, then the current file is closed. This means that an '@c endfile' was not given for that file. (We should probably print a diagnostic in this case, although at the moment we do not.)

The '`for`' loop does the work. It reads lines using `getline` (lásd 5.8. Bekezdés [Explicit Input with `getline`], 53. oldal). For an unexpected end of file, it calls the `unexpected_eof` function. If the line is an "endfile" line, then it breaks out of the loop. If the line is an '`@group`' or '`@end group`' line, then it ignores it, and goes on to the next line. (These Texinfo control lines keep blocks of code together on one page; unfortunately, TEX isn't always smart enough to do things exactly right, and we have to give it some advice.)

Most of the work is in the following few lines. If the line has no '`@`' symbols, it can be printed directly. Otherwise, each leading '`@`' must be stripped off.

To remove the '`@`' symbols, the line is split into separate elements of the array `a`, using the `split` function (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal). Each element of `a` that is empty indicates two successive '`@`' symbols in the original line. For each two empty elements ('`@@`' in the original file), we have to add back in a single '`@`' symbol.

When the processing of the array is finished, `join` is called with the value of `SUBSEP`, to rejoin the pieces back into a single line. That line is then printed to the output file.

```
/^@c(omment)?[ \t]+file/    \
{
    if (NF != 3) {
        e = (FILENAME ":" FNR ": badly formed `file' line")
        print e > "/dev/stderr"
        next
    }
    if ($3 != curfile) {
        if (curfile != "")
            close(curfile)
        curfile = $3
    }

    for (;;) {
        if ((getline line) <= 0)
            unexpected_eof()
        if (line ~ /^@c(omment)?[ \t]+endfile/)
            break
        else if (line ~ /^@(end[ \t]+)?group/)
            continue
        if (index(line, "@") == 0) {
            print line > curfile
            continue
        }
```

```
        n = split(line, a, "@")
        # if a[1] == "", means leading @,
        # don't add one back in.
        for (i = 2; i <= n; i++) {
            if (a[i] == "") { # was an @@
                a[i] = "@"
                if (a[i+1] == "")
                    i++
            }
        }
        print join(a, 1, n, SUBSEP) > curfile
    }
}
```

An important thing to note is the use of the '>' redirection. Output done with '>' only opens the file once; it stays open and subsequent output is appended to the file (lásd 6.6. Bekezdés [Redirecting Output of print and printf], 70. oldal). This allows us to easily mix program text and explanatory prose for the same sample source file (as has been done here!) without any hassle. The file is only closed when a new data file name is encountered, or at the end of the input file.

Finally, the function unexpected_eof prints an appropriate error message and then exits.

The END rule handles the final cleanup, closing the open file.

```
function unexpected_eof()
{
    printf("%s:%d: unexpected EOF or error\n", \
        FILENAME, FNR) > "/dev/stderr"
    exit 1
}

END {
    if (curfile)
        close(curfile)
}
```

16.2.8. A Simple Stream Editor

The sed utility is a "stream editor," a program that reads a stream of data, makes changes to it, and passes the modified data on. It is often used to make global changes to a large file, or to a stream of data generated by a pipeline of commands.

While sed is a complicated program in its own right, its most common use is to perform global substitutions in the middle of a pipeline:

```
command1 < orig.data | sed 's/old/new/g' | command2 > result
```

Here, the 's/old/new/g' tells sed to look for the regexp 'old' on each input line, and replace it with the text 'new', globally (i.e. all the occurrences on a line). This is similar to awk's gsub function (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).

The following program, 'awksed.awk', accepts at least two command line arguments; the pattern to look for and the text to replace it with. Any additional arguments are treated as data file names to process. If none are provided, the standard input is used.

```
# awksed.awk --- do s/foo/bar/g using just print
#    Thanks to Michael Brennan for the idea

# Arnold Robbins, arnold@gnu.org, Public Domain
# August 1995

function usage()
{
    print "usage: awksed pat repl [files...]" > "/dev/stderr"
    exit 1
}
BEGIN {
    # validate arguments
    if (ARGC < 3)
        usage()

    RS = ARGV[1]
    ORS = ARGV[2]

    # don't use arguments as files
    ARGV[1] = ARGV[2] = ""
}

# look ma, no hands!
{
    if (RT == "")
        printf "%s", $0
    else
        print
}
```

The program relies on gawk's ability to have RS be a regexp and on the setting of RT to the actual text that terminated the record (lásd 5.1. Bekezdés [How Input is Split into Records], 37. oldal).

The idea is to have RS be the pattern to look for. gawk will automatically set $0 to the text between matches of the pattern. This is text that we wish to keep, unmodified. Then, by setting `ORS` to the replacement text, a simple `print` statement will output the text we wish to keep, followed by the replacement text.

There is one wrinkle to this scheme, which is what to do if the last record doesn't end with text that matches `RS`? Using a `print` statement unconditionally prints the replacement text, which is not correct.

However, if the file did not end in text that matches `RS`, `RT` will be set to the null string. In this case, we can print `$0` using `printf` (lásd 6.5. Bekezdés [Using `printf` Statements for Fancier Printing], 64. oldal).

The `BEGIN` rule handles the setup, checking for the right number of arguments, and calling `usage` if there is a problem. Then it sets `RS` and `ORS` from the command line arguments, and sets `ARGV[1]` and `ARGV[2]` to the null string, so that they will not be treated as file names (lásd 10.3. Bekezdés [Using `ARGC` and `ARGV`], 120. oldal).

The `usage` function prints an error message and exits.

Finally, the single rule handles the printing scheme outlined above, using `print` or `printf` as appropriate, depending upon the value of `RT`.

16.2.9. An Easy Way to Use Library Functions

Using library functions in `awk` can be very beneficial. It encourages code re-use and the writing of general functions. Programs are smaller, and therefore clearer. However, using library functions is only easy when writing `awk` programs; it is painful when running them, requiring multiple '`-f`' options. If `gawk` is unavailable, then so too is the `AWKPATH` environment variable and the ability to put `awk` functions into a library directory (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

It would be nice to be able to write programs like so:

```
# library functions
@include getopt.awk
@include join.awk
...

# main program
BEGIN {
    while ((c = getopt(ARGC, ARGV, "a:b:cde")) != -1)
        ...
    ...
}
```

The following program, '`igawk.sh`', provides this service. It simulates `gawk`'s searching of the `AWKPATH` variable, and also allows *nested* includes; i.e. a file that has been included with '`@include`' can contain further '`@include`' statements. `igawk` will make an effort to only include files once, so that nested includes don't accidentally include a library function twice.

`igawk` should behave externally just like `gawk`. This means it should accept all of `gawk`'s command line arguments, including the ability to have multiple source files specified via '`-f`', and the ability to mix command line and library source files.

The program is written using the POSIX Shell (`sh`) command language. The way the program works is as follows:

1. Loop through the arguments, saving anything that doesn't represent `awk` source code for later, when the expanded program is run.
2. For any arguments that do represent `awk` text, put the arguments into a temporary file that will be expanded. There are two cases.
   a. Literal text, provided with '`--source`' or '`--source=`'. This text is just echoed directly. The `echo` program will automatically supply a trailing newline.
   b. File names provided with '`-f`'. We use a neat trick, and echo '`@include` *filename*' into the temporary file. Since the file inclusion program will work the way `gawk` does, this will get the text of the file included into the program at the correct point.
3. Run an `awk` program (naturally) over the temporary file to expand '`@include`' statements. The expanded program is placed in a second temporary file.
4. Run the expanded program with `gawk` and any other original command line arguments that the user supplied (such as the data file names).

The initial part of the program turns on shell tracing if the first argument was '`debug`'. Otherwise, a shell `trap` statement arranges to clean up any temporary files on program exit or upon an interrupt.

The next part loops through all the command line arguments. There are several cases of interest.

`--`
: This ends the arguments to `igawk`. Anything else should be passed on to the user's `awk` program without being evaluated.

`-W`
: This indicates that the next option is specific to `gawk`. To make argument processing easier, the '`-W`' is appended to the front of the remaining arguments and the loop continues. (This is an `sh` programming trick. Don't worry about it if you are not familiar with `sh`.)

`-v`
`-F`
: These are saved and passed on to `gawk`.

`-f`
`--file`
`--file=`
`-Wfile=`
: The file name is saved to the temporary file '`/tmp/ig.s.$$`' with an '`@include`' statement. The `sed` utility is used to remove the leading option part of the argument (e.g., '`--file=`').

`--source`
`--source=`
`-Wsource=`
: The source text is echoed into '`/tmp/ig.s.$$`'.

`--version`
`--version`
`-Wversion`
: `igawk` prints its version number, and runs '`gawk --version`' to get the `gawk` version information, and then exits.

If none of '`-f`', '`--file`', '`-Wfile`', '`--source`', or '`-Wsource`', were supplied, then the first non-option argument should be the `awk` program. If there are no command line arguments left, `igawk` prints an error message and exits. Otherwise, the first argument is echoed into '`/tmp/ig.s.$$`'.

In any case, after the arguments have been processed, '`/tmp/ig.s.$$`' contains the complete text of the original `awk` program.

The '`$$`' in `sh` represents the current process ID number. It is often used in shell programs to generate unique temporary file names. This allows multiple users to run `igawk` without worrying that the temporary file names will clash.

Here's the program:

```
#! /bin/sh

# igawk --- like gawk but do @include processing
# Arnold Robbins, arnold@gnu.org, Public Domain
# July 1993

if [ "$1" = debug ]
then
    set -x
    shift
else
    # cleanup on exit, hangup, interrupt, quit, termination
    trap 'rm -f /tmp/ig.[se].$$' 0 1 2 3 15
fi

while [ $# -ne 0 ] # loop over arguments
do
    case $1 in
    --)     shift; break;;

    -W)     shift
            set -- -W"$@"
            continue;;
```

```
    -[vF])  opts="$opts $1 '$2'"
            shift;;

    -[vF]*) opts="$opts '$1'" ;;

    -f)     echo @include "$2" >> /tmp/ig.s.$$
            shift;;

    -f*)    f=`echo "$1" | sed 's/-f//'`
            echo @include "$f" >> /tmp/ig.s.$$ ;;
    -?file=*)    # -Wfile or --file
            f=`echo "$1" | sed 's/-.file=//'`
            echo @include "$f" >> /tmp/ig.s.$$ ;;

    -?file)    # get arg, $2
            echo @include "$2" >> /tmp/ig.s.$$
            shift;;

    -?source=*)    # -Wsource or --source
            t=`echo "$1" | sed 's/-.source=//'`
            echo "$t" >> /tmp/ig.s.$$ ;;

    -?source)  # get arg, $2
            echo "$2" >> /tmp/ig.s.$$
            shift;;

    -?version)
            echo igawk: version 1.0 1>&2
            gawk --version
            exit 0 ;;

    -[W-]*)    opts="$opts '$1'" ;;

    *)      break;;
    esac
    shift
done

if [ ! -s /tmp/ig.s.$$ ]
then
    if [ -z "$1" ]
    then
```

```
        echo igawk: no program! 1>&2
        exit 1
    else
        echo "$1" > /tmp/ig.s.$$
        shift
    fi
fi

# at this point, /tmp/ig.s.$$ has the program
```

The `awk` program to process '`@include`' directives reads through the program, one line at a time using `getline` (lásd 5.8. Bekezdés [Explicit Input with `getline`], 53. oldal). The input file names and '`@include`' statements are managed using a stack. As each '`@include`' is encountered, the current file name is "pushed" onto the stack, and the file named in the '`@include`' directive becomes the current file name. As each file is finished, the stack is "popped," and the previous input file becomes the current input file again. The process is started by making the original file the first one on the stack.

The `pathto` function does the work of finding the full path to a file. It simulates `gawk`'s behavior when searching the `AWKPATH` environment variable (lásd 14.3. Bekezdés [The `AWKPATH` Environment Variable], 166. oldal). If a file name has a '`/`' in it, no path search is done. Otherwise, the file name is concatenated with the name of each directory in the path, and an attempt is made to open the generated file name. The only way in `awk` to test if a file can be read is to go ahead and try to read it with `getline`; that is what `pathto` does.[5] If the file can be read, it is closed, and the file name is returned.

```
gawk -- '
# process @include directives

function pathto(file,    i, t, junk)
{
    if (index(file, "/") != 0)
        return file

    for (i = 1; i <= ndirs; i++) {
        t = (pathlist[i] "/" file)
        if ((getline junk < t) > 0) {
            # found it
            close(t)
            return t
        }
    }
```

[5] On some very old versions of `awk`, the test '`getline junk < t`' can loop forever if the file exists but is empty. Caveat Emptor.

```
        return ""
    }
```

The main program is contained inside one `BEGIN` rule. The first thing it does is set up the `pathlist` array that `pathto` uses. After splitting the path on ':', null elements are replaced with ".", which represents the current directory.

```
    BEGIN {
        path = ENVIRON["AWKPATH"]
        ndirs = split(path, pathlist, ":")
        for (i = 1; i <= ndirs; i++) {
            if (pathlist[i] == "")
                pathlist[i] = "."
        }
```

The stack is initialized with `ARGV[1]`, which will be '/tmp/ig.s.$$'. The main loop comes next. Input lines are read in succession. Lines that do not start with '@include' are printed verbatim.

If the line does start with '@include', the file name is in `$2`. `pathto` is called to generate the full path. If it could not, then we print an error message and continue.

The next thing to check is if the file has been included already. The `processed` array is indexed by the full file name of each included file, and it tracks this information for us. If the file has been seen, a warning message is printed. Otherwise, the new file name is pushed onto the stack and processing continues.

Finally, when `getline` encounters the end of the input file, the file is closed and the stack is popped. When `stackptr` is less than zero, the program is done.

```
        stackptr = 0
        input[stackptr] = ARGV[1] # ARGV[1] is first file

        for (; stackptr >= 0; stackptr--) {
            while ((getline < input[stackptr]) > 0) {
                if (tolower($1) != "@include") {
                    print
                    continue
                }
                fpath = pathto($2)
                if (fpath == "") {
                    printf("igawk:%s:%d: cannot find %s\n", \
                        input[stackptr], FNR, $2) > "/dev/stderr"
                    continue
                }
```

```
            if (! (fpath in processed)) {
                processed[fpath] = input[stackptr]
                input[++stackptr] = fpath
            } else
                print $2, "included in", input[stackptr], \
                    "already included in", \
                    processed[fpath] > "/dev/stderr"
        }
        close(input[stackptr])
    }
}' /tmp/ig.s.$$ > /tmp/ig.e.$$
```

The last step is to call `gawk` with the expanded program and the original options and command line arguments that the user supplied. `gawk`'s exit status is passed back on to `igawk`'s calling program.

```
eval gawk -f /tmp/ig.e.$$ $opts -- "$@"

exit $?
```

This version of `igawk` represents my third attempt at this program. There are three key simplifications that made the program work better.

1. Using '`@include`' even for the files named with '`-f`' makes building the initial collected `awk` program much simpler; all the '`@include`' processing can be done once.
2. The `pathto` function doesn't try to save the line read with `getline` when testing for the file's accessibility. Trying to save this line for use with the main program complicates things considerably.
3. Using a `getline` loop in the `BEGIN` rule does it all in one place. It is not necessary to call out to a separate loop for processing nested '`@include`' statements.

Also, this program illustrates that it is often worthwhile to combine `sh` and `awk` programming together. You can usually accomplish quite a lot, without having to resort to low-level programming in C or C++, and it is frequently easier to do certain kinds of string and argument manipulation using the shell than it is in `awk`.

Finally, `igawk` shows that it is not always necessary to add new features to a program; they can often be layered on top. With `igawk`, there is no real reason to build '`@include`' processing into `gawk` itself.

As an additional example of this, consider the idea of having two files in a directory in the search path.

'`default.awk`'
: This file would contain a set of default library functions, such as `getopt` and `assert`.

'`site.awk`'
: This file would contain library functions that are specific to a site or installation, i.e. locally developed functions. Having a separate file allows '`default.awk`' to change with new `gawk` releases, without requiring the system administrator to update it each time by adding the local functions.

One user suggested that `gawk` be modified to automatically read these files upon startup. Instead, it would be very simple to modify `igawk` to do this. Since `igawk` can process nested '`@include`' directives, '`default.awk`' could simply contain '`@include`' statements for the desired library functions.

17. The Evolution of the awk Language

This könyv describes the GNU implementation of `awk`, which follows the POSIX specification. Many `awk` users are only familiar with the original `awk` implementation in Version 7 Unix. (This implementation was the basis for `awk` in Berkeley Unix, through 4.3–Reno. The 4.4 release of Berkeley Unix uses `gawk` 2.15.2 for its version of `awk`.) This chapter briefly describes the evolution of the `awk` language, with cross references to other parts of the könyv where you can find more information.

17.1. Major Changes between V7 and SVR3.1

The `awk` language evolved considerably between the release of Version 7 Unix (1978) and the new version first made generally available in System V Release 3.1 (1987). This section summarizes the changes, with cross-references to further details.

- The requirement for '`;`' to separate rules on a line (lásd 2.6. Bekezdés [`awk` Statements Versus Lines], 16. oldal).
- User-defined functions, and the `return` statement (lásd 13. Fejezet [User-defined Functions], 153. oldal).
- The `delete` statement (lásd 11.6. Bekezdés [The `delete` Statement], 127. oldal).
- The `do-while` statement (lásd 9.3. Bekezdés [The `do-while` Statement], 106. oldal).
- The built-in functions `atan2`, `cos`, `sin`, `rand` and `srand` (lásd 12.2. Bekezdés [Numeric Built-in Functions], 134. oldal).
- The built-in functions `gsub`, `sub`, and `match` (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).
- The built-in functions `close`, and `system` (lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal).
- The `ARGC`, `ARGV`, `FNR`, `RLENGTH`, `RSTART`, and `SUBSEP` built-in variables (lásd 10. Fejezet [Built-in Variables], 115. oldal).
- The conditional expression using the ternary operator '`?:`' (lásd 7.12. Bekezdés [Conditional Expressions], 92. oldal).
- The exponentiation operator '`^`' (lásd 7.5. Bekezdés [Arithmetic Operators], 82. oldal) and its assignment operator form '`^=`' (lásd 7.7. Bekezdés [Assignment Expressions], 84. oldal).
- C-compatible operator precedence, which breaks some old `awk` programs (lásd 7.14. Bekezdés [Operator Precedence (How Operators Nest)], 94. oldal).
- Regexps as the value of `FS` (lásd 5.5. Bekezdés [Specifying How Fields are Separated], 44. oldal), and as the third argument to the `split` function (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).

- Dynamic regexps as operands of the '~' and '!~' operators (lásd 4.1. Bekezdés [How to Use Regular Expressions], 23. oldal).
- The escape sequences '\b', '\f', and '\r' (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal). (Some vendors have updated their old versions of `awk` to recognize '\r', '\b', and '\f', but this is not something you can rely on.)
- Redirection of input for the `getline` function (lásd 5.8. Bekezdés [Explicit Input with `getline`], 53. oldal).
- Multiple `BEGIN` and `END` rules (lásd 8.1.5. Bekezdés [The `BEGIN` and `END` Special Patterns], 100. oldal).
- Multi-dimensional arrays (lásd 11.9. Bekezdés [Multi-dimensional Arrays], 130. oldal).

17.2. Changes between SVR3.1 and SVR4

The System V Release 4 version of Unix `awk` added these features (some of which originated in `gawk`):

- The `ENVIRON` variable (lásd 10. Fejezet [Built-in Variables], 115. oldal).
- Multiple '-f' options on the command line (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The '-v' option for assigning variables before program execution begins (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The '--' option for terminating command line options.
- The '\a', '\v', and '\x' escape sequences (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal).
- A defined return value for the `srand` built-in function (lásd 12.2. Bekezdés [Numeric Built-in Functions], 134. oldal).
- The `toupper` and `tolower` built-in string functions for case translation (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).
- A cleaner specification for the '%c' format-control letter in the `printf` function (lásd 6.5.2. Bekezdés [Format-Control Letters], 65. oldal).
- The ability to dynamically pass the field width and precision (`"%*.*d"`) in the argument list of the `printf` function (lásd 6.5.2. Bekezdés [Format-Control Letters], 65. oldal).
- The use of regexp constants such as `/foo/` as expressions, where they are equivalent to using the matching operator, as in '`$0 ~ /foo/`' (lásd 7.2. Bekezdés [Using Regular Expression Constants], 78. oldal).

17.3. Changes between SVR4 and POSIX `awk`

The POSIX Command Language and Utilities standard for `awk` introduced the following changes into the language:

- The use of '-W' for implementation-specific options.
- The use of CONVFMT for controlling the conversion of numbers to strings (lásd 7.4. Bekezdés [Conversion of Strings and Numbers], 81. oldal).
- The concept of a numeric string, and tighter comparison rules to go with it (lásd 7.10. Bekezdés [Variable Typing and Comparison Expressions], 88. oldal).
- More complete documentation of many of the previously undocumented features of the language.

The following common extensions are not permitted by the POSIX standard:

- \x escape sequences are not recognized (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal).
- Newlines do not act as whitespace to separate fields when FS is equal to a single space.
- The synonym func for the keyword function is not recognized (lásd 13.1. Bekezdés [Function Definition Syntax], 153. oldal).
- The operators '**' and '**=' cannot be used in place of '^' and '^=' (lásd 7.5. Bekezdés [Arithmetic Operators], 82. oldal, and also lásd 7.7. Bekezdés [Assignment Expressions], 84. oldal).
- Specifying '-Ft' on the command line does not set the value of FS to be a single tab character (lásd 5.5. Bekezdés [Specifying How Fields are Separated], 44. oldal).
- The fflush built-in function is not supported (lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal).

17.4. Extensions in the Bell Laboratories awk

Brian Kernighan, one of the original designers of Unix awk, has made his version available via anonymous ftp (lásd B.8. Bekezdés [Other Freely Available awk Implementations], 291. oldal). This section describes extensions in his version of awk that are not in POSIX awk.

- The '-mf *NNN*' and '-mr *NNN*' command line options to set the maximum number of fields, and the maximum record size, respectively (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The fflush built-in function for flushing buffered output (lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal).

17.5. Extensions in gawk Not in POSIX awk

The GNU implementation, gawk, adds a number of features. This sections lists them in the order they were added to gawk. They can all be disabled with either the '--traditional' or '--posix' options (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

Version 2.10 of `gawk` introduced these features:

- The `AWKPATH` environment variable for specifying a path search for the '`-f`' command line option (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The `IGNORECASE` variable and its effects (lásd 4.5. Bekezdés [Case-sensitivity in Matching], 33. oldal).
- The '`/dev/stdin`', '`/dev/stdout`', '`/dev/stderr`', and '`/dev/fd/`*n*' file name interpretation (lásd 6.7. Bekezdés [Special File Names in `gawk`], 71. oldal).

Version 2.13 of `gawk` introduced these features:

- The `FIELDWIDTHS` variable and its effects (lásd 5.6. Bekezdés [Reading Fixed-width Data], 49. oldal).
- The `systime` and `strftime` built-in functions for obtaining and printing time stamps (lásd 12.5. Bekezdés [Functions for Dealing with Time Stamps], 147. oldal).
- The '`-W lint`' option to provide source code and run time error and portability checking (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The '`-W compat`' option to turn off these extensions (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The '`-W posix`' option for full POSIX compliance (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).

Version 2.14 of `gawk` introduced these features:

- The `next file` statement for skipping to the next data file (lásd 9.8. Bekezdés [The `nextfile` Statement], 112. oldal).

Version 2.15 of `gawk` introduced these features:

- The `ARGIND` variable, that tracks the movement of `FILENAME` through `ARGV` (lásd 10. Fejezet [Built-in Variables], 115. oldal).
- The `ERRNO` variable, that contains the system error message when `getline` returns -1, or when `close` fails (lásd 10. Fejezet [Built-in Variables], 115. oldal).
- The ability to use GNU-style long named options that start with '`--`' (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The '`--source`' option for mixing command line and library file source code (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The '`/dev/pid`', '`/dev/ppid`', '`/dev/pgrpid`', and '`/dev/user`' file name interpretation (lásd 6.7. Bekezdés [Special File Names in `gawk`], 71. oldal).

Version 3.0 of `gawk` introduced these features:

- The `next file` statement became `nextfile` (lásd 9.8. Bekezdés [The `nextfile` Statement], 112. oldal).

- The '--lint-old' option to warn about constructs that are not available in the original Version 7 Unix version of awk (lásd 17.1. Bekezdés [Major Changes between V7 and SVR3.1], 251. oldal).
- The '--traditional' option was added as a better name for '--compat' (lásd 14.1. Bekezdés [Command Line Options], 161. oldal).
- The ability for FS to be a null string, and for the third argument to split to be the null string (lásd 5.5.3. Bekezdés [Making Each Character a Separate Field], 46. oldal).
- The ability for RS to be a regexp (lásd 5.1. Bekezdés [How Input is Split into Records], 37. oldal).
- The RT variable (lásd 5.1. Bekezdés [How Input is Split into Records], 37. oldal).
- The gensub function for more powerful text manipulation (lásd 12.3. Bekezdés [Built-in Functions for String Manipulation], 135. oldal).
- The strftime function acquired a default time format, allowing it to be called with no arguments (lásd 12.5. Bekezdés [Functions for Dealing with Time Stamps], 147. oldal).
- Full support for both POSIX and GNU regexps (lásd 4. Fejezet [Regular Expressions], 23. oldal).
- The '--re-interval' option to provide interval expressions in regexps (lásd 4.3. Bekezdés [Regular Expression Operators], 26. oldal).
- IGNORECASE changed, now applying to string comparison as well as regexp operations (lásd 4.5. Bekezdés [Case-sensitivity in Matching], 33. oldal).
- The '-m' option and the fflush function from the Bell Labs research version of awk (lásd 14.1. Bekezdés [Command Line Options], 161. oldal; also lásd 12.4. Bekezdés [Built-in Functions for Input/Output], 144. oldal).
- The use of GNU Autoconf to control the configuration process (lásd B.2.1. Bekezdés [Compiling gawk for Unix], 282. oldal).
- Amiga support (lásd B.6. Bekezdés [Installing gawk on an Amiga], 289. oldal).

A. Függelék gawk Summary

This appendix provides a brief summary of the `gawk` command line and the `awk` language. It is designed to serve as "quick reference." It is therefore terse, but complete.

A.1. Command Line Options Summary

The command line consists of options to `gawk` itself, the `awk` program text (if not supplied via the '`-f`' option), and values to be made available in the `ARGC` and `ARGV` predefined `awk` variables:

`gawk` [*POSIX or GNU style options*] `-f` *source-file* [`--`] *file* ...
`gawk` [*POSIX or GNU style options*] [`--`] `'`*program*`'` *file* ...

The options that `gawk` accepts are:

`-F` *fs*
`--field-separator` *fs*
: Use *fs* for the input field separator (the value of the `FS` predefined variable).

`-f` *program-file*
`--file` *program-file*
: Read the `awk` program source from the file *program-file*, instead of from the first command line argument.

`-mf` *NNN*
`-mr` *NNN*
: The '`f`' flag sets the maximum number of fields, and the '`r`' flag sets the maximum record size. These options are ignored by `gawk`, since `gawk` has no predefined limits; they are only for compatibility with the Bell Labs research version of Unix `awk`.

`-v` *var*`=`*val*
`--assign` *var*`=`*val*
: Assign the variable *var* the value *val* before program execution begins.

`-W traditional`
`-W compat`
`--traditional`
`--compat`
: Use compatibility mode, in which `gawk` extensions are turned off.

`-W copyleft`
`-W copyright`
`--copyleft`
`--copyright`
: Print the short version of the General Public License on the standard output, and exit. This option may disappear in a future version of `gawk`.

`-W help`
`-W usage`
`--help`
`--usage` Print a relatively short summary of the available options on the standard output, and exit.

`-W lint`
`--lint` Give warnings about dubious or non-portable `awk` constructs.

`-W lint-old`
`--lint-old`
Warn about constructs that are not available in the original Version 7 Unix version of `awk`.

`-W posix`
`--posix` Use POSIX compatibility mode, in which `gawk` extensions are turned off and additional restrictions apply.

`-W re-interval`
`--re-interval`
Allow interval expressions (lásd 4.3. Bekezdés [Regular Expression Operators], 26. oldal), in regexps.

`-W source=`*program-text*
`--source` *program-text*
Use *program-text* as `awk` program source code. This option allows mixing command line source code with source code from files, and is particularly useful for mixing command line programs with library functions.

`-W version`
`--version`
Print version information for this particular copy of `gawk` on the error output.

`--` Signal the end of options. This is useful to allow further arguments to the `awk` program itself to start with a '`-`'. This is mainly for consistency with POSIX argument parsing conventions.

Any other options are flagged as invalid, but are otherwise ignored. Lásd 14.1. Bekezdés [Command Line Options], 161. oldal, for more details.

A.2. Language Summary

An `awk` program consists of a sequence of zero or more pattern-action statements and optional function definitions. One or the other of the pattern and action may be omitted.

pattern `{` *action statements* `}`
pattern
`{` *action statements* `}`

`function name(parameter list)     { action statements }`

gawk first reads the program source from the *program-file*(s), if specified, or from the first non-option argument on the command line. The '-f' option may be used multiple times on the command line. gawk reads the program text from all the *program-file* files, effectively concatenating them in the order they are specified. This is useful for building libraries of awk functions, without having to include them in each new awk program that uses them. To use a library function in a file from a program typed in on the command line, specify '--source '*program*'', and type your program in between the single quotes. Lásd 14.1. Bekezdés [Command Line Options], 161. oldal.

The environment variable AWKPATH specifies a search path to use when finding source files named with the '-f' option. The default path, which is '.:/usr/local/share/awk'[1] is used if AWKPATH is not set. If a file name given to the '-f' option contains a '/' character, no path search is performed. Lásd 14.3. Bekezdés [The AWKPATH Environment Variable], 166. oldal.

gawk compiles the program into an internal form, and then proceeds to read each file named in the ARGV array. The initial values of ARGV come from the command line arguments. If there are no files named on the command line, gawk reads the standard input.

If a "file" named on the command line has the form '*var*=*val*', it is treated as a variable assignment: the variable *var* is assigned the value *val*. If any of the files have a value that is the null string, that element in the list is skipped.

For each record in the input, gawk tests to see if it matches any *pattern* in the awk program. For each pattern that the record matches, the associated *action* is executed.

A.3. Variables and Fields

awk variables are not declared; they come into existence when they are first used. Their values are either floating-point numbers or strings. awk also has one-dimensional arrays; multiple-dimensional arrays may be simulated. There are several predefined variables that awk sets as a program runs; these are summarized below.

A.3.1. Fields

As each input line is read, gawk splits the line into *fields*, using the value of the FS variable as the field separator. If FS is a single character, fields are separated by that character. Otherwise, FS is expected to be a full regular expression. In the special case that FS is a single space, fields are separated

[1] The path may use a directory other than '/usr/local/share/awk', depending upon how gawk was built and installed.

by runs of spaces, tabs and/or newlines.[2] If `FS` is the null string (`""`), then each individual character in the record becomes a separate field. Note that the value of `IGNORECASE` (lásd 4.5. Bekezdés [Case-sensitivity in Matching], 33. oldal) also affects how fields are split when `FS` is a regular expression.

Each field in the input line may be referenced by its position, `$1`, `$2`, and so on. `$0` is the whole line. The value of a field may be assigned to as well. Field numbers need not be constants:

```
n = 5
print $n
```

prints the fifth field in the input line. The variable `NF` is set to the total number of fields in the input line.

References to non-existent fields (i.e. fields after `$NF`) return the null string. However, assigning to a non-existent field (e.g., `$(NF+2) = 5`) increases the value of `NF`, creates any intervening fields with the null string as their value, and causes the value of `$0` to be recomputed, with the fields being separated by the value of `OFS`. Decrementing `NF` causes the values of fields past the new value to be lost, and the value of `$0` to be recomputed, with the fields being separated by the value of `OFS`. Lásd 5. Fejezet [Reading Input Files], 37. oldal.

A.3.2. Built-in Variables

`gawk`'s built-in variables are:

`ARGC` The number of elements in `ARGV`. See below for what is actually included in `ARGV`.

`ARGIND` The index in `ARGV` of the current file being processed. When `gawk` is processing the input data files, it is always true that '`FILENAME == ARGV[ARGIND]`'.

`ARGV` The array of command line arguments. The array is indexed from zero to `ARGC` − 1. Dynamically changing `ARGC` and the contents of `ARGV` can control the files used for data. A null-valued element in `ARGV` is ignored. `ARGV` does not include the options to `awk` or the text of the `awk` program itself.

`CONVFMT` The conversion format to use when converting numbers to strings.

`FIELDWIDTHS`
A space separated list of numbers describing the fixed-width input data.

`ENVIRON` An array of environment variable values. The array is indexed by variable name, each element being the value of that variable.

[2] In POSIX `awk`, newline does not separate fields.

Thus, the environment variable `HOME` is `ENVIRON["HOME"]`. One possible value might be '`/home/arnold`'.

Changing this array does not affect the environment seen by programs which `gawk` spawns via redirection or the `system` function. (This may change in a future version of `gawk`.)

Some operating systems do not have environment variables. The `ENVIRON` array is empty when running on these systems.

`ERRNO` The system error message when an error occurs using `getline` or `close`.

`FILENAME` The name of the current input file. If no files are specified on the command line, the value of `FILENAME` is the null string.

`FNR` The input record number in the current input file.

`FS` The input field separator, a space by default.

`IGNORECASE`
The case-sensitivity flag for string comparisons and regular expression operations. If `IGNORECASE` has a non-zero value, then pattern matching in rules, record separating with `RS`, field splitting with `FS`, regular expression matching with '`~`' and '`!~`', and the `gensub`, `gsub`, `index`, `match`, `split` and `sub` built-in functions all ignore case when doing regular expression operations, and all string comparisons are done ignoring case. The value of `IGNORECASE` does *not* affect array subscripting.

`NF` The number of fields in the current input record.

`NR` The total number of input records seen so far.

`OFMT` The output format for numbers for the `print` statement, `"%.6g"` by default.

`OFS` The output field separator, a space by default.

`ORS` The output record separator, by default a newline.

`RS` The input record separator, by default a newline. If `RS` is set to the null string, then records are separated by blank lines. When `RS` is set to the null string, then the newline character always acts as a field separator, in addition to whatever value `FS` may have. If `RS` is set to a multi-character string, it denotes a regexp; input text matching the regexp separates records.

`RT` The input text that matched the text denoted by `RS`, the record separator.

`RSTART` The index of the first character last matched by `match`; zero if no match.

`RLENGTH` The length of the string last matched by `match`; −1 if no match.

`SUBSEP` The string used to separate multiple subscripts in array elements, by default `"\034"`.

Lásd 10. Fejezet [Built-in Variables], 115. oldal, for more information.

A.3.3. Arrays

Arrays are subscripted with an expression between square brackets (‘`[`’ and ‘`]`’). Array subscripts are *always* strings; numbers are converted to strings as necessary, following the standard conversion rules (lásd 7.4. Bekezdés [Conversion of Strings and Numbers], 81. oldal).

If you use multiple expressions separated by commas inside the square brackets, then the array subscript is a string consisting of the concatenation of the individual subscript values, converted to strings, separated by the subscript separator (the value of `SUBSEP`).

The special operator `in` may be used in a conditional context to see if an array has an index consisting of a particular value.

```
if (val in array)
        print array[val]
```

If the array has multiple subscripts, use ‘`(i, j, ...) in` *array*’ to test for existence of an element.

The `in` construct may also be used in a `for` loop to iterate over all the elements of an array. Lásd 11.5. Bekezdés [Scanning All Elements of an Array], 126. oldal.

You can remove an element from an array using the `delete` statement.

You can clear an entire array using ‘`delete` *array*’.

Lásd 11. Fejezet [Arrays in `awk`], 123. oldal.

A.3.4. Data Types

The value of an `awk` expression is always either a number or a string.

Some contexts (such as arithmetic operators) require numeric values. They convert strings to numbers by interpreting the text of the string as a number. If the string does not look like a number, it converts to zero.

Other contexts (such as concatenation) require string values. They convert numbers to strings by effectively printing them with `sprintf`. Lásd 7.4. Bekezdés [Conversion of Strings and Numbers], 81. oldal, for the details.

To force conversion of a string value to a number, simply add zero to it. If the value you start with is already a number, this does not change it.

To force conversion of a numeric value to a string, concatenate it with the null string.

Comparisons are done numerically if both operands are numeric, or if one is numeric and the other is a numeric string. Otherwise one or both operands are converted to strings and a string comparison is performed. Fields, `getline` input, `FILENAME`, `ARGV` elements, `ENVIRON` elements and the elements of an array created by `split` are the only items that can be numeric strings. String constants, such as `"3.1415927"` are not numeric strings, they are string constants. The full rules for comparisons are described in 7.10. Bekezdés [Variable Typing and Comparison Expressions], 88. oldal.

Uninitialized variables have the string value `""` (the null, or empty, string). In contexts where a number is required, this is equivalent to zero.

Lásd 7.3. Bekezdés [Variables], 79. oldal, for more information on variable naming and initialization; lásd 7.4. Bekezdés [Conversion of Strings and Numbers], 81. oldal, for more information on how variable values are interpreted.

A.4. Patterns

An `awk` program is mostly composed of rules, each consisting of a pattern followed by an action. The action is enclosed in '`{`' and '`}`'. Either the pattern may be missing, or the action may be missing, but not both. If the pattern is missing, the action is executed for every input record. A missing action is equivalent to '`{ print }`', which prints the entire line.

Comments begin with the '`#`' character, and continue until the end of the line. Blank lines may be used to separate statements. Statements normally end with a newline; however, this is not the case for lines ending in a '`,`', '`{`', '`?`', '`:`', '`&&`', or '`||`'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '`\`', in which case the newline is ignored.

Multiple statements may be put on one line by separating each one with a '`;`'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

Lásd 2.2.5. Bekezdés [Comments in awk Programs], 13. oldal, for information on `awk`'s commenting convention; lásd 2.6. Bekezdés [awk Statements Versus Lines], 16. oldal, for a description of the line continuation mechanism in `awk`.

A.4.1. Pattern Summary

`awk` patterns may be one of the following:

`/`*regular expression*`/`
relational expression
pattern `&&` *pattern*
pattern `||` *pattern*
pattern `?` *pattern* `:` *pattern*
`(`*pattern*`)`
`!` *pattern*

pattern1, *pattern2*
`BEGIN`
`END`

`BEGIN` and `END` are two special kinds of patterns that are not tested against the input. The action parts of all `BEGIN` rules are concatenated as if all the statements had been written in a single `BEGIN` rule. They are executed before any of the input is read. Similarly, all the `END` rules are concatenated, and executed when all the input is exhausted (or when an `exit` statement is executed). `BEGIN` and `END` patterns cannot be combined with other patterns in pattern expressions. `BEGIN` and `END` rules cannot have missing action parts.

For */regular-expression/* patterns, the associated statement is executed for each input record that matches the regular expression. Regular expressions are summarized below.

A *relational expression* may use any of the operators defined below in the section on actions. These generally test whether certain fields match certain regular expressions.

The '`&&`', '`||`', and '`!`' operators are logical "and," logical "or," and logical "not," respectively, as in C. They do short-circuit evaluation, also as in C, and are used for combining more primitive pattern expressions. As in most languages, parentheses may be used to change the order of evaluation.

The '`?:`' operator is like the same operator in C. If the first pattern matches, then the second pattern is matched against the input record; otherwise, the third is matched. Only one of the second and third patterns is matched.

The '*pattern1*, *pattern2*' form of a pattern is called a range pattern. It matches all input lines starting with a line that matches *pattern1*, and continuing until a line that matches *pattern2*, inclusive. A range pattern cannot be used as an operand of any of the pattern operators.

Lásd 8.1. Bekezdés [Pattern Elements], 97. oldal.

A.4.2. Regular Expressions

Regular expressions are based on POSIX EREs (extended regular expressions). The escape sequences allowed in string constants are also valid in regular expressions (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal). Regexps are composed of characters as follows:

c matches the character *c* (assuming *c* is none of the characters listed below).

c matches the literal character *c*.

. matches any character, *including* newline. In strict POSIX mode, '`.`' does not match the NUL character, which is a character with all bits equal to zero.

^ matches the beginning of a string.

`$` matches the end of a string.

`[`*abc*`...]` matches any of the characters *abc*... (character list).

`[[:`*class*`:]]`
matches any character in the character class *class*. Allowable classes are `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`.

`[[.`*symbol*`.]]`
matches the multi-character collating symbol *symbol*. `gawk` does not currently support collating symbols.

`[[=`*classname*`=]]`
matches any of the equivalent characters in the current locale named by the equivalence class *classname*. `gawk` does not currently support equivalence classes.

`[^`*abc*`...]`
matches any character except *abc*... (negated character list).

r1`|`*r2* matches either *r1* or *r2* (alternation).

r1r2 matches *r1*, and then *r2* (concatenation).

r`+` matches one or more *r*'s.

r`*` matches zero or more *r*'s.

r`?` matches zero or one *r*'s.

`(`*r*`)` matches *r* (grouping).

r`{`*n*`}`
r`{`*n*`,}`
r`{`*n*`,`*m*`}` matches at least *n*, *n* to any number, or *n* to *m* occurrences of *r* (interval expressions).

`\y` matches the empty string at either the beginning or the end of a word.

`\B` matches the empty string within a word.

`\<` matches the empty string at the beginning of a word.

`\>` matches the empty string at the end of a word.

`\w` matches any word-constituent character (alphanumeric characters and the underscore).

`\W` matches any character that is not word-constituent.

`` \` `` matches the empty string at the beginning of a buffer (same as a string in `gawk`).

`\'` matches the empty string at the end of a buffer.

The various command line options control how `gawk` interprets characters in regexps.

No options
: In the default case, `gawk` provide all the facilities of POSIX regexps and the GNU regexp operators described above. However, interval expressions are not supported.

`--posix`
: Only POSIX regexps are supported, the GNU operators are not special (e.g., '`\w`' matches a literal '`w`'). Interval expressions are allowed.

`--traditional`
: Traditional Unix `awk` regexps are matched. The GNU operators are not special, interval expressions are not available, and neither are the POSIX character classes (`[[:alnum:]]` and so on). Characters described by octal and hexadecimal escape sequences are treated literally, even if they represent regexp metacharacters.

`--re-interval`
: Allow interval expressions in regexps, even if '`--traditional`' has been provided.

Lásd 4. Fejezet [Regular Expressions], 23. oldal.

A.5. Actions

Action statements are enclosed in braces, '`{`' and '`}`'. A missing action statement is equivalent to '`{ print }`'.

Action statements consist of the usual assignment, conditional, and looping statements found in most languages. The operators, control statements, and Input/Output statements available are similar to those in C.

Comments begin with the '`#`' character, and continue until the end of the line. Blank lines may be used to separate statements. Statements normally end with a newline; however, this is not the case for lines ending in a '`,`', '`{`', '`?`', '`:`', '`&&`', or '`||`'. Lines ending in `do` or `else` also have their statements automatically continued on the following line. In other cases, a line can be continued by ending it with a '`\`', in which case the newline is ignored.

Multiple statements may be put on one line by separating each one with a '`;`'. This applies to both the statements within the action part of a rule (the usual case), and to the rule statements.

Lásd 2.2.5. Bekezdés [Comments in `awk` Programs], 13. oldal, for information on `awk`'s commenting convention; lásd 2.6. Bekezdés [`awk` Statements Versus Lines], 16. oldal, for a description of the line continuation mechanism in `awk`.

A.5.1. Operators

The operators in `awk`, in order of decreasing precedence, are:

`(...)`
: Grouping.

`$`
: Field reference.

`++ --`
: Increment and decrement, both prefix and postfix.

`^`
: Exponentiation ('`**`' may also be used, and '`**=`' for the assignment operator, but they are not specified in the POSIX standard).

`+ - !`
: Unary plus, unary minus, and logical negation.

`* / %`
: Multiplication, division, and modulus.

`+ -`
: Addition and subtraction.

space
: String concatenation.

`< <= > >= != ==`
: The usual relational operators.

`~ !~`
: Regular expression match, negated match.

`in`
: Array membership.

`&&`
: Logical "and".

`||`
: Logical "or".

`?:`
: A conditional expression. This has the form '*expr1* `?` *expr2* `:` *expr3*'. If *expr1* is true, the value of the expression is *expr2*; otherwise it is *expr3*. Only one of *expr2* and *expr3* is evaluated.

`= += -= *= /= %= ^=`
: Assignment. Both absolute assignment (*var*`=`*value*) and operator assignment (the other forms) are supported.

Lásd 7. Fejezet [Expressions], 77. oldal.

A.5.2. Control Statements

The control statements are as follows:

```
if (condition) statement [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
for (var in array) statement
break
continue
delete array[index]
```

`delete` *array*
`exit` [*expression*]
`{` *statements* `}`

Lásd 9. Fejezet [Control Statements in Actions], 105. oldal.

A.5.3. I/O Statements

The Input/Output statements are as follows:

`getline`
: Set `$0` from next input record; set `NF`, `NR`, `FNR`. Lásd 5.8. Bekezdés [Explicit Input with `getline`], 53. oldal.

`getline <`*file*
: Set `$0` from next record of *file*; set `NF`.

`getline` *var*
: Set *var* from next input record; set `NR`, `FNR`.

`getline` *var* `<`*file*
: Set *var* from next record of *file*.

command `| getline`
: Run *command*, piping its output into `getline`; sets `$0`, `NF`, `NR`.

command `| getline var`
: Run *command*, piping its output into `getline`; sets *var*.

`next`
: Stop processing the current input record. The next input record is read and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed. Lásd 9.7. Bekezdés [The `next` Statement], 111. oldal.

`nextfile`
: Stop processing the current input file. The next input record read comes from the next input file. `FILENAME` is updated, `FNR` is set to one, `ARGIND` is incremented, and processing starts over with the first pattern in the `awk` program. If the end of the input data is reached, the `END` rule(s), if any, are executed. Earlier versions of `gawk` used '`next file`'; this usage is still supported, but is considered to be deprecated. Lásd 9.8. Bekezdés [The `nextfile` Statement], 112. oldal.

`print`
: Prints the current record. Lásd 6. Fejezet [Printing Output], 61. oldal.

`print` *expr-list*
: Prints expressions.

`print` *expr-list* `>` *file*
: Prints expressions to *file*. If *file* does not exist, it is created. If it does exist, its contents are deleted the first time the `print` is executed.

`print` *expr-list* `>>` *file*
: Prints expressions to *file*. The previous contents of *file* are retained, and the output of `print` is appended to the file.

`print` *expr-list* `|` *command*
: Prints expressions, sending the output down a pipe to *command*. The pipeline to the command stays open until the `close` function is called.

`printf` *fmt, expr-list*
: Format and print.

`printf` *fmt, expr-list* `> file`
: Format and print to *file*. If *file* does not exist, it is created. If it does exist, its contents are deleted the first time the `printf` is executed.

`printf` *fmt, expr-list* `>>` *file*
: Format and print to *file*. The previous contents of *file* are retained, and the output of `printf` is appended to the file.

`printf` *fmt, expr-list* `|` *command*
: Format and print, sending the output down a pipe to *command*. The pipeline to the command stays open until the `close` function is called.

`getline` returns zero on end of file, and −1 on an error. In the event of an error, `getline` will set `ERRNO` to the value of a system-dependent string that describes the error.

A.5.4. printf Summary

Conversion specification have the form `%`[*flag*][*width*][`.`*prec*]*format*. Items in brackets are optional.

The `awk` `printf` statement and `sprintf` function accept the following conversion specification formats:

`%c`
: An ASCII character. If the argument used for '`%c`' is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

`%d`
`%i`
: A decimal number (the integer part).

`%e`
`%E`
: A floating point number of the form '[`-`]`d.dddddde`[`+-`]`dd`'. The '`%E`' format uses '`E`' instead of '`e`'.

`%f`
: A floating point number of the form [`-`]`ddd.dddddd`.

`%g`
`%G` Use either the '`%e`' or '`%f`' formats, whichever produces a shorter string, with non-significant zeros suppressed. '`%G`' will use '`%E`' instead of '`%e`'.

`%o` An unsigned octal number (again, an integer).

`%s` A character string.

`%x`
`%X` An unsigned hexadecimal number (an integer). The '`%X`' format uses '`A`' through '`F`' instead of '`a`' through '`f`' for decimal 10 through 15.

`%%` A single '`%`' character; no argument is converted.

There are optional, additional parameters that may lie between the '`%`' and the control letter:

`-` The expression should be left-justified within its field.

space For numeric conversions, prefix positive values with a space, and negative values with a minus sign.

`+` The plus sign, used before the width modifier (see below), says to always supply a sign for numeric conversions, even if the data to be formatted is positive. The '`+`' overrides the space modifier.

`#` Use an "alternate form" for certain control letters. For '`o`', supply a leading zero. For '`x`', and '`X`', supply a leading '`0x`' or '`0X`' for a non-zero result. For '`e`', '`E`', and '`f`', the result will always contain a decimal point. For '`g`', and '`G`', trailing zeros are not removed from the result.

`0` A leading '`0`' (zero) acts as a flag, that indicates output should be padded with zeros instead of spaces. This applies even to non-numeric output formats. This flag only has an effect when the field width is wider than the value to be printed.

width The field should be padded to this width. The field is normally padded with spaces. If the '`0`' flag has been used, it is padded with zeros.

`.`*prec* A number that specifies the precision to use when printing. For the '`e`', '`E`', and '`f`' formats, this specifies the number of digits you want printed to the right of the decimal point. For the '`g`', and '`G`' formats, it specifies the maximum number of significant digits. For the '`d`', '`o`', '`i`', '`u`', '`x`', and '`X`' formats, it specifies the minimum number of digits to print. For the '`s`' format, it specifies the maximum number of characters from the string that should be printed.

Either or both of the *width* and *prec* values may be specified as '`*`'. In that case, the particular value is taken from the argument list.

Lásd 6.5. Bekezdés [Using `printf` Statements for Fancier Printing], 64. oldal.

A.5.5. Special File Names

When doing I/O redirection from either `print` or `printf` into a file, or via `getline` from a file, `gawk` recognizes certain special file names internally. These file names allow access to open file descriptors inherited from `gawk`'s parent process (usually the shell). The file names are:

'`/dev/stdin`'
: The standard input.

'`/dev/stdout`'
: The standard output.

'`/dev/stderr`'
: The standard error output.

'`/dev/fd/`*n*'
: The file denoted by the open file descriptor *n*.

In addition, reading the following files provides process related information about the running `gawk` program. All returned records are terminated with a newline.

'`/dev/pid`'
: Returns the process ID of the current process.

'`/dev/ppid`'
: Returns the parent process ID of the current process.

'`/dev/pgrpid`'
: Returns the process group ID of the current process.

'`/dev/user`'
: At least four space-separated fields, containing the return values of the `getuid`, `geteuid`, `getgid`, and `getegid` system calls. If there are any additional fields, they are the group IDs returned by `getgroups` system call. (Multiple groups may not be supported on all systems.)

These file names may also be used on the command line to name data files. These file names are only recognized internally if you do not actually have files with these names on your system.

Lásd 6.7. Bekezdés [Special File Names in `gawk`], 71. oldal, for a longer description that provides the motivation for this feature.

A.5.6. Built-in Functions

`awk` provides a number of built-in functions for performing numeric operations, string related operations, and I/O related operations.

The built-in arithmetic functions are:

`atan2(`*y*`, `*x*`)`
: the arctangent of *y/x* in radians.

`cos(`*expr*`)` the cosine of *expr*, which is in radians.

`exp(`*expr*`)` the exponential function (`e ^` *expr*).

`int(`*expr*`)` truncates to integer.

`log(`*expr*`)` the natural logarithm of `expr`.

`rand()` a random number between zero and one.

`sin(`*expr*`)` the sine of *expr*, which is in radians.

`sqrt(`*expr*`)`
: the square root function.

`srand(`[*expr*]`)`
: use *expr* as a new seed for the random number generator. If no *expr* is provided, the time of day is used. The return value is the previous seed for the random number generator.

`awk` has the following built-in string functions:

`gensub(`*regex*`, `*subst*`, `*how* [`, `*target*]`)`
: If *how* is a string beginning with '`g`' or '`G`', then replace each match of *regex* in *target* with *subst*. Otherwise, replace the *how*'th occurrence. If *target* is not supplied, use `$0`. The return value is the changed string; the original *target* is not modified. Within *subst*, '*n*', where *n* is a digit from one to nine, can be used to indicate the text that matched the *n*'th parenthesized subexpression. This function is `gawk`-specific.

`gsub(`*regex*`, `*subst* [`, `*target*]`)`
: for each substring matching the regular expression *regex* in the string *target*, substitute the string *subst*, and return the number of substitutions. If *target* is not supplied, use `$0`.

`index(`*str*`, `*search*`)`
: returns the index of the string *search* in the string *str*, or zero if *search* is not present.

`length(`[*str*]`)`
: returns the length of the string *str*. The length of `$0` is returned if no argument is supplied.

`match(`*str*`, `*regex*`)`
: returns the position in *str* where the regular expression *regex* occurs, or zero if *regex* is not present, and sets the values of `RSTART` and `RLENGTH`.

`split(`*str*`, `*arr* [`, `*regex*]`)`
: splits the string *str* into the array *arr* on the regular expression *regex*, and returns the number of elements. If *regex* is omitted, `FS` is used instead. *regex* can be the null string, causing each character to be placed into its own array element. The array *arr* is cleared first.

`sprintf(`*fmt*`, `*expr-list*`)`
: prints *expr-list* according to *fmt*, and returns the resulting string.

`sub(`*regex*`, `*subst* [`, `*target*]`)`
: just like `gsub`, but only the first matching substring is replaced.

`substr(`*str*`, `*index* [`, `*len*]`)`
: returns the *len*-character substring of *str* starting at *index*. If *len* is omitted, the rest of *str* is used.

`tolower(`*str*`)`
: returns a copy of the string *str*, with all the upper-case characters in *str* translated to their corresponding lower-case counterparts. Non-alphabetic characters are left unchanged.

`toupper(`*str*`)`
: returns a copy of the string *str*, with all the lower-case characters in *str* translated to their corresponding upper-case counterparts. Non-alphabetic characters are left unchanged.

The I/O related functions are:

`close(`*expr*`)`
: Close the open file or pipe denoted by *expr*.

`fflush(`[*expr*]`)`
: Flush any buffered output for the output file or pipe denoted by *expr*. If *expr* is omitted, standard output is flushed. If *expr* is the null string (`""`), all output buffers are flushed.

`system(`*cmd-line*`)`
: Execute the command *cmd-line*, and return the exit status. If your operating system does not support `system`, calling it will generate a fatal error.

: '`system("")`' can be used to force `awk` to flush any pending output. This is more portable, but less obvious, than calling `fflush`.

A.5.7. Time Functions

The following two functions are available for getting the current time of day, and for formatting time stamps. They are specific to `gawk`.

`systime()`
: returns the current time of day as the number of seconds since a particular epoch (Midnight, January 1, 1970 UTC, on POSIX systems).

`strftime([`*format*`[, `*timestamp*`]])`
: formats *timestamp* according to the specification in *format*. The current time of day is used if no *timestamp* is supplied. A default format equivalent to the output of the `date` utility is used if no *format* is supplied. Lásd 12.5. Bekezdés [Functions for Dealing with Time Stamps], 147. oldal, for the details on the conversion specifiers that `strftime` accepts.

Lásd 12. Fejezet [Built-in Functions], 133. oldal, for a description of all of `awk`'s built-in functions.

A.5.8. String Constants

String constants in `awk` are sequences of characters enclosed in double quotes (`"`). Within strings, certain *escape sequences* are recognized, as in C. These are:

`\\` A literal backslash.

`\a` The "alert" character; usually the ASCII BEL character.

`\b` Backspace.

`\f` Formfeed.

`\n` Newline.

`\r` Carriage return.

`\t` Horizontal tab.

`\v` Vertical tab.

`\x`*hex digits*
: The character represented by the string of hexadecimal digits following the '`\x`'. As in ANSI C, all following hexadecimal digits are considered part of the escape sequence. E.g., `"\x1B"` is a string containing the ASCII ESC (escape) character. (The '`\x`' escape sequence is not in POSIX `awk`.)

`\`*ddd* The character represented by the one, two, or three digit sequence of octal digits. Thus, `"\033"` is also a string containing the ASCII ESC (escape) character.

`\c` The literal character *c*, if *c* is not one of the above.

The escape sequences may also be used inside constant regular expressions (e.g., the regexp `/[ \t\f\n\r\v]/` matches whitespace characters).

Lásd 4.2. Bekezdés [Escape Sequences], 24. oldal.

A.6. User-defined Functions

Functions in `awk` are defined as follows:

`function` *name*`(`*parameter list*`) {` *statements* `}`

Actual parameters supplied in the function call are used to instantiate the formal parameters declared in the function. Arrays are passed by reference, other variables are passed by value.

If there are fewer arguments passed than there are names in *parameter-list*, the extra names are given the null string as their value. Extra names have the effect of local variables.

The open-parenthesis in a function call of a user-defined function must immediately follow the function name, without any intervening white space. This is to avoid a syntactic ambiguity with the concatenation operator.

The word `func` may be used in place of `function` (but not in POSIX `awk`).

Use the `return` statement to return a value from a function.

Lásd 13. Fejezet [User-defined Functions], 153. oldal.

A.7. Historical Features

There are two features of historical `awk` implementations that `gawk` supports.

First, it is possible to call the `length` built-in function not only with no arguments, but even without parentheses!

```
a = length
```

is the same as either of

```
a = length()
a = length($0)
```

For example:

```
$ echo abcdef | awk '{ print length }'
⊣ 6
```

This feature is marked as “deprecated” in the POSIX standard, and `gawk` will issue a warning about its use if ‘`--lint`’ is specified on the command line. (The ability to use `length` this way was actually an accident of the original Unix `awk` implementation. If any built-in function used `$0` as its default argument, it was possible to call that function without the parentheses. In particular, it was common practice to use the `length` function in this fashion, and this usage was documented in the `awk` manual page.)

The other historical feature is the use of either the `break` statement, or the `continue` statement outside the body of a `while`, `for`, or `do` loop. Traditional `awk` implementations have treated such usage as equivalent to the `next` statement. More recent versions of Unix `awk` do not allow it. `gawk` supports this usage if '`--traditional`' has been specified.

Lásd 14.1. Bekezdés [Command Line Options], 161. oldal, for more information about the '`--posix`' and '`--lint`' options.

B. Függelék Installing gawk

This appendix provides instructions for installing gawk on the various platforms that are supported by the developers. The primary developers support Unix (and one day, GNU), while the other ports were contributed. The file 'ACKNOWLEDGMENT' in the gawk distribution lists the electronic mail addresses of the people who did the respective ports, and they are also provided in B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal.

B.1. The gawk Distribution

This section first describes how to get the gawk distribution, how to extract it, and then what is in the various files and subdirectories.

B.1.1. Getting the gawk Distribution

There are three ways you can get GNU software.

1. You can copy it from someone else who already has it.
2. You can order gawk directly from the Free Software Foundation. Software distributions are available for Unix, MS-DOS, and VMS, on tape and CD-ROM. The address is:

   Free Software Foundation
   59 Temple Place—Suite 330
   Boston, MA 02111-1307 USA
   Phone: +1-617-542-5942
   Fax (including Japan): +1-617-542-2652
   E-mail: gnu@gnu.org

   Ordering from the FSF directly contributes to the support of the foundation and to the production of more free software.
3. You can get gawk by using anonymous ftp to the Internet host gnudist.gnu.org, in the directory '/gnu/gawk'.

   Here is a list of alternate ftp sites from which you can obtain GNU software. When a site is listed as "*site*:*directory*" the *directory* indicates the directory where GNU software is kept. You should use a site that is geographically close to you.

   Asia:

   ```
   cair-archive.kaist.ac.kr:/pub/gnu
   ftp.cs.titech.ac.jp
   ftp.nectec.or.th:/pub/mirrors/gnu
   utsun.s.u-tokyo.ac.jp:/ftpsync/prep
   ```

   Australia:

archie.au:/gnu
(archie.oz or archie.oz.au for ACSnet)

Africa:

ftp.sun.ac.za:/pub/gnu

Middle East:

ftp.technion.ac.il:/pub/unsupported/gnu

Europe:

archive.eu.net
ftp.denet.dk
ftp.eunet.ch
ftp.funet.fi:/pub/gnu
ftp.ieunet.ie:pub/gnu
ftp.informatik.rwth-aachen.de:/pub/gnu
ftp.informatik.tu-muenchen.de
ftp.luth.se:/pub/unix/gnu
ftp.mcc.ac.uk
ftp.stacken.kth.se
ftp.sunet.se:/pub/gnu
ftp.univ-lyon1.fr:pub/gnu
ftp.win.tue.nl:/pub/gnu
irisa.irisa.fr:/pub/gnu
isy.liu.se
nic.switch.ch:/mirror/gnu
src.doc.ic.ac.uk:/gnu
unix.hensa.ac.uk:/pub/uunet/systems/gnu

South America:

ftp.inf.utfsm.cl:/pub/gnu
ftp.unicamp.br:/pub/gnu

Western Canada:

ftp.cs.ubc.ca:/mirror2/gnu

USA:

col.hp.com:/mirrors/gnu
f.ms.uky.edu:/pub3/gnu
ftp.cc.gatech.edu:/pub/gnu
ftp.cs.columbia.edu:/archives/gnu/prep
ftp.digex.net:/pub/gnu
ftp.hawaii.edu:/mirrors/gnu
ftp.kpc.com:/pub/mirror/gnu

USA (continued):

```
ftp.uu.net:/systems/gnu
gatekeeper.dec.com:/pub/GNU
jaguar.utah.edu:/gnustuff
labrea.stanford.edu
mrcnext.cso.uiuc.edu:/pub/gnu
vixen.cso.uiuc.edu:/gnu
wuarchive.wustl.edu:/systems/gnu
```

B.1.2. Extracting the Distribution

`gawk` is distributed as a `tar` file compressed with the GNU Zip program, `gzip`.

Once you have the distribution (for example, '`gawk-3.0.4.tar.gz`'), first use `gzip` to expand the file, and then use `tar` to extract it. You can use the following pipeline to produce the `gawk` distribution:

```
# Under System V, add 'o' to the tar flags
gzip -d -c gawk-3.0.4.tar.gz | tar -xvpf -
```

This will create a directory named '`gawk-3.0.4`' in the current directory.

The distribution file name is of the form '`gawk-`*V*`.`*R*`.`*n*`.tar.gz`'. The *V* represents the major version of `gawk`, the *R* represents the current release of version *V*, and the *n* represents a *patch level*, meaning that minor bugs have been fixed in the release. The current patch level is 4, but when retrieving distributions, you should get the version with the highest version, release, and patch level. (Note that release levels greater than or equal to 90 denote "beta," or non-production software; you may not wish to retrieve such a version unless you don't mind experimenting.)

If you are not on a Unix system, you will need to make other arrangements for getting and extracting the `gawk` distribution. You should consult a local expert.

B.1.3. Contents of the gawk Distribution

The `gawk` distribution has a number of C source files, documentation files, subdirectories and files related to the configuration process (lásd B.2. Bekezdés [Compiling and Installing gawk on Unix], 282. oldal), and several subdirectories related to different, non-Unix, operating systems.

various '`.c`', '`.y`', and '`.h`' files
: These files are the actual `gawk` source code.

'`README`'
'`README_d/README.*`'
: Descriptive files: '`README`' for `gawk` under Unix, and the rest for the various hardware and software combinations.

'INSTALL' A file providing an overview of the configuration and installation process.

'PORTS' A list of systems to which gawk has been ported, and which have successfully run the test suite.

'ACKNOWLEDGMENT'
A list of the people who contributed major parts of the code or documentation.

'ChangeLog'
A detailed list of source code changes as bugs are fixed or improvements made.

'NEWS' A list of changes to gawk since the last release or patch.

'COPYING' The GNU General Public License.

'FUTURES' A brief list of features and/or changes being contemplated for future releases, with some indication of the time frame for the feature, based on its difficulty.

'LIMITATIONS'
A list of those factors that limit gawk's performance. Most of these depend on the hardware or operating system software, and are not limits in gawk itself.

'POSIX.STD'
A description of one area where the POSIX standard for awk is incorrect, and how gawk handles the problem.

'PROBLEMS'
A file describing known problems with the current release.

'doc/awkforai.txt'
A short article describing why gawk is a good language for AI (Artificial Intelligence) programming.

'doc/README.card'
'doc/ad.block'
'doc/awkcard.in'
'doc/cardfonts'
'doc/colors'
'doc/macros'
'doc/no.colors'
'doc/setter.outline'
The troff source for a five-color awk reference card. A modern version of troff, such as GNU Troff (groff) is needed to produce the color version. See the file 'README.card' for instructions if you have an older troff.

'doc/gawk.1'
: The troff source for a manual page describing gawk. This is distributed for the convenience of Unix users.

'doc/gawk.texi'
: The Texinfo source file for this könyv. It should be processed with TeX to produce a printed document, and with makeinfo to produce an Info file.

'doc/gawk.info'
: The generated Info file for this könyv.

'doc/igawk.1'
: The troff source for a manual page describing the igawk program presented in 16.2.9. Bekezdés [An Easy Way to Use Library Functions], 243. oldal.

'doc/Makefile.in'
: The input file used during the configuration process to generate the actual 'Makefile' for creating the documentation.

'Makefile.in'
'acconfig.h'
'aclocal.m4'
'configh.in'
'configure.in'
'configure'
'custom.h'
'missing/*'
: These files and subdirectory are used when configuring gawk for various Unix systems. They are explained in detail in B.2. Bekezdés [Compiling and Installing gawk on Unix], 282. oldal.

'awklib/extract.awk'
'awklib/Makefile.in'
: The 'awklib' directory contains a copy of 'extract.awk' (lásd 16.2.7. Bekezdés [Extracting Programs from Texinfo Source Files], 238. oldal), which can be used to extract the sample programs from the Texinfo source file for this könyv, and a 'Makefile.in' file, which configure uses to generate a 'Makefile'. As part of the process of building gawk, the library functions from 15. Fejezet [A Library of awk Functions], 169. oldal, and the igawk program from 16.2.9. Bekezdés [An Easy Way to Use Library Functions], 243. oldal, are extracted into ready to use files. They are installed as part of the installation process.

'atari/*'
: Files needed for building gawk on an Atari ST. Lásd B.5. Bekezdés [Installing gawk on the Atari ST], 287. oldal, for details.

'pc/*'
: Files needed for building gawk under MS-DOS and OS/2. Lásd B.4. Bekezdés [MS-DOS and OS/2 Installation and Compilation], 286. oldal, for details.

'vms/*'
: Files needed for building gawk under VMS. Lásd B.3. Bekezdés [How to Compile and Install gawk on VMS], 283. oldal, for details.

'test/*'
: A test suite for gawk. You can use 'make check' from the top level gawk directory to run your version of gawk against the test suite. If gawk successfully passes 'make check' then you can be confident of a successful port.

B.2. Compiling and Installing gawk on Unix

Usually, you can compile and install gawk by typing only two commands. However, if you do use an unusual system, you may need to configure gawk for your system yourself.

B.2.1. Compiling gawk for Unix

After you have extracted the gawk distribution, cd to 'gawk-3.0.4'. Like most GNU software, gawk is configured automatically for your Unix system by running the configure program. This program is a Bourne shell script that was generated automatically using GNU autoconf. (The autoconf software is described fully in *Autoconf—Generating Automatic Configuration Scripts*, which is available from the Free Software Foundation.)

To configure gawk, simply run configure:

```
sh ./configure
```

This produces a 'Makefile' and 'config.h' tailored to your system. The 'config.h' file describes various facts about your system. You may wish to edit the 'Makefile' to change the CFLAGS variable, which controls the command line options that are passed to the C compiler (such as optimization levels, or compiling for debugging).

Alternatively, you can add your own values for most make variables, such as CC and CFLAGS, on the command line when running configure:

```
CC=cc CFLAGS=-g sh ./configure
```

See the file 'INSTALL' in the gawk distribution for all the details.

After you have run configure, and possibly edited the 'Makefile', type:

```
make
```

and shortly thereafter, you should have an executable version of gawk. That's all there is to it! (If these steps do not work, please send in a bug report; lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal.)

B.2.2. The Configuration Process

(This section is of interest only if you know something about using the C language and the Unix operating system.)

The source code for gawk generally attempts to adhere to formal standards wherever possible. This means that gawk uses library routines that are specified by the ANSI C standard and by the POSIX operating system interface standard. When using an ANSI C compiler, function prototypes are used to help improve the compile-time checking.

Many Unix systems do not support all of either the ANSI or the POSIX standards. The 'missing' subdirectory in the gawk distribution contains replacement versions of those subroutines that are most likely to be missing.

The 'config.h' file that is created by the configure program contains definitions that describe features of the particular operating system where you are attempting to compile gawk. The three things described by this file are what header files are available, so that they can be correctly included, what (supposedly) standard functions are actually available in your C libraries, and other miscellaneous facts about your variant of Unix. For example, there may not be an st_blksize element in the stat structure. In this case 'HAVE_ST_BLKSIZE' would be undefined.

It is possible for your C compiler to lie to configure. It may do so by not exiting with an error when a library function is not available. To get around this, you can edit the file 'custom.h'. Use an '#ifdef' that is appropriate for your system, and either #define any constants that configure should have defined but didn't, or #undef any constants that configure defined and should not have. 'custom.h' is automatically included by 'config.h'.

It is also possible that the configure program generated by autoconf will not work on your system in some other fashion. If you do have a problem, the file 'configure.in' is the input for autoconf. You may be able to change this file, and generate a new version of configure that will work on your system. Lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal, for information on how to report problems in configuring gawk. The same mechanism may be used to send in updates to 'configure.in' and/or 'custom.h'.

B.3. How to Compile and Install gawk on VMS

This section describes how to compile and install gawk under VMS.

B.3.1. Compiling gawk on VMS

To compile gawk under VMS, there is a DCL command procedure that will issue all the necessary CC and LINK commands, and there is also a 'Makefile' for use with the MMS utility. From the source directory, use either

```
$ @[.VMS]VMSBUILD.COM
```

or

```
$ MMS/DESCRIPTION=[.VMS]DESCRIP.MMS GAWK
```

Depending upon which C compiler you are using, follow one of the sets of instructions in this table:

VAX C V3.x
: Use either 'vmsbuild.com' or 'descrip.mms' as is. These use CC/OPTIMIZE=NOLINE, which is essential for Version 3.0.

VAX C V2.x
: You must have Version 2.3 or 2.4; older ones won't work. Edit either 'vmsbuild.com' or 'descrip.mms' according to the comments in them. For 'vmsbuild.com', this just entails removing two '!' delimiters. Also edit 'config.h' (which is a copy of file '[.config]vms-conf.h') and comment out or delete the two lines '#define __STDC__ 0' and '#define VAXC_BUILTINS' near the end.

GNU C
: Edit 'vmsbuild.com' or 'descrip.mms'; the changes are different from those for VAX C V2.x, but equally straightforward. No changes to 'config.h' should be needed.

DEC C
: Edit 'vmsbuild.com' or 'descrip.mms' according to their comments. No changes to 'config.h' should be needed.

gawk has been tested under VAX/VMS 5.5-1 using VAX C V3.2, GNU C 1.40 and 2.3. It should work without modifications for VMS V4.6 and up.

B.3.2. Installing gawk on VMS

To install gawk, all you need is a "foreign" command, which is a DCL symbol whose value begins with a dollar sign. For example:

```
$ GAWK :== $disk1:[gnubin]GAWK
```

(Substitute the actual location of gawk.exe for '$disk1:[gnubin]'.) The symbol should be placed in the 'login.com' of any user who wishes to run gawk, so that it will be defined every time the user logs on. Alternatively, the symbol may be placed in the system-wide 'sylogin.com' procedure, which will allow all users to run gawk.

Optionally, the help entry can be loaded into a VMS help library:

```
$ LIBRARY/HELP SYS$HELP:HELPLIB [.VMS]GAWK.HLP
```

(You may want to substitute a site-specific help library rather than the standard VMS library 'HELPLIB'.) After loading the help text,

```
$ HELP GAWK
```

will provide information about both the gawk implementation and the awk programming language.

The logical name 'AWK_LIBRARY' can designate a default location for awk program files. For the '-f' option, if the specified filename has no device or directory path information in it, gawk will look in the current directory first, then in the directory specified by the translation of 'AWK_LIBRARY' if the file was not found. If after searching in both directories, the file still is not found, then gawk appends the suffix '.awk' to the filename and the file search will be re-tried. If 'AWK_LIBRARY' is not defined, that portion of the file search will fail benignly.

B.3.3. Running gawk on VMS

Command line parsing and quoting conventions are significantly different on VMS, so examples in this könyv or from other sources often need minor changes. They *are* minor though, and all awk programs should run correctly.

Here are a couple of trivial tests:

```
$ gawk -- "BEGIN {print ""Hello, World!""}"
$ gawk -"W" version
! could also be -"W version" or "-W version"
```

Note that upper-case and mixed-case text must be quoted.

The VMS port of gawk includes a DCL-style interface in addition to the original shell-style interface (see the help entry for details). One side-effect of dual command line parsing is that if there is only a single parameter (as in the quoted string program above), the command becomes ambiguous. To work around this, the normally optional '--' flag is required to force Unix style rather than DCL parsing. If any other dash-type options (or multiple parameters such as data files to be processed) are present, there is no ambiguity and '--' can be omitted.

The default search path when looking for awk program files specified by the '-f' option is "SYS$DISK:[],AWK_LIBRARY:". The logical name 'AWKPATH' can be used to override this default. The format of 'AWKPATH' is a comma-separated list of directory specifications. When defining it, the value should be quoted so that it retains a single translation, and not a multi-translation RMS searchlist.

B.3.4. Building and Using gawk on VMS POSIX

Ignore the instructions above, although 'vms/gawk.hlp' should still be made available in a help library. The source tree should be unpacked into a container file subsystem rather than into the ordinary VMS file system. Make sure that the two scripts, 'configure' and 'vms/posix-cc.sh', are executable; use 'chmod +x' on them if necessary. Then execute the following two commands:

```
psx> CC=vms/posix-cc.sh configure
psx> make CC=c89 gawk
```

The first command will construct files 'config.h' and 'Makefile' out of templates, using a script to make the C compiler fit configure's expectations. The second command will compile and link gawk using the C compiler directly; ignore any warnings from `make` about being unable to redefine `CC`. `configure` will take a very long time to execute, but at least it provides incremental feedback as it runs.

This has been tested with VAX/VMS V6.2, VMS POSIX V2.0, and DEC C V5.2.

Once built, `gawk` will work like any other shell utility. Unlike the normal VMS port of `gawk`, no special command line manipulation is needed in the VMS POSIX environment.

B.4. MS-DOS and OS/2 Installation and Compilation

If you have received a binary distribution prepared by the DOS maintainers, then `gawk` and the necessary support files will appear under the '`gnu`' directory, with executables in '`gnu/bin`', libraries in '`gnu/lib/awk`', and manual pages under '`gnu/man`'. This is designed for easy installation to a '`/gnu`' directory on your drive, but the files can be installed anywhere provided `AWKPATH` is set properly. Regardless of the installation directory, the first line of '`igawk.cmd`' and '`igawk.bat`' (in '`gnu/bin`') may need to be edited.

The binary distribution will contain a separate file describing the contents. In particular, it may include more than one version of the `gawk` executable. OS/2 binary distributions may have a different arrangement, but installation is similar.

The OS/2 and MS-DOS versions of `gawk` search for program files as described in 14.3. Bekezdés [The `AWKPATH` Environment Variable], 166. oldal. However, semicolons (rather than colons) separate elements in the `AWKPATH` variable. If `AWKPATH` is not set or is empty, then the default search path is `".;c:/lib/awk;c:/gnu/lib/awk"`.

An `sh`-like shell (as opposed to `command.com` under MS-DOS or `cmd.exe` under OS/2) may be useful for `awk` programming. Ian Stewartson has written an excellent shell for MS-DOS and OS/2, and a `ksh` clone and GNU Bash are available for OS/2. The file '`README_d/README.pc`' in the `gawk` distribution contains information on these shells. Users of Stewartson's shell on DOS should examine its documentation on handling of command-lines. In particular, the setting for `gawk` in the shell configuration may need to be changed, and the `ignoretype` option may also be of interest.

`gawk` can be compiled for MS-DOS and OS/2 using the GNU development tools from DJ Delorie (DJGPP, MS-DOS-only) or Eberhard Mattes (EMX, MS-DOS and OS/2). Microsoft C can be used to build 16-bit versions for MS-DOS and OS/2. The file '`README_d/README.pc`' in the `gawk` distribution contains additional notes, and '`pc/Makefile`' contains important notes on compilation options.

To build gawk, copy the files in the 'pc' directory (*except* for 'ChangeLog') to the directory with the rest of the gawk sources. The 'Makefile' contains a configuration section with comments, and may need to be edited in order to work with your make utility.

The 'Makefile' contains a number of targets for building various MS-DOS and OS/2 versions. A list of targets will be printed if the make command is given without a target. As an example, to build gawk using the DJGPP tools, enter 'make djgpp'.

Using make to run the standard tests and to install gawk requires additional Unix-like tools, including sh, sed, and cp. In order to run the tests, the 'test/*.ok' files may need to be converted so that they have the usual DOS-style end-of-line markers. Most of the tests will work properly with Stewartson's shell along with the companion utilities or appropriate GNU utilities. However, some editing of 'test/Makefile' is required. It is recommended that the file 'pc/Makefile.tst' be copied to 'test/Makefile' as a replacement. Details can be found in 'README_d/README.pc'.

B.5. Installing gawk on the Atari ST

There are no substantial differences when installing gawk on various Atari models. Compiled gawk executables do not require a large amount of memory with most awk programs and should run on all Motorola processor based models (called further ST, even if that is not exactly right).

In order to use gawk, you need to have a shell, either text or graphics, that does not map all the characters of a command line to upper-case. Maintaining case distinction in option flags is very important (lásd 14.1. Bekezdés [Command Line Options], 161. oldal). These days this is the default, and it may only be a problem for some very old machines. If your system does not preserve the case of option flags, you will need to upgrade your tools. Support for I/O redirection is necessary to make it easy to import awk programs from other environments. Pipes are nice to have, but not vital.

B.5.1. Compiling gawk on the Atari ST

A proper compilation of gawk sources when sizeof(int) differs from sizeof(void *) requires an ANSI C compiler. An initial port was done with gcc. You may actually prefer executables where ints are four bytes wide, but the other variant works as well.

You may need quite a bit of memory when trying to recompile the gawk sources, as some source files ('regex.c' in particular) are quite big. If you run out of memory compiling such a file, try reducing the optimization level for this particular file; this may help.

With a reasonable shell (Bash will do), and in particular if you run Linux, MiNT or a similar operating system, you have a pretty good chance that the configure utility will succeed. Otherwise sample versions of 'config.h' and 'Makefile.st' are given in the 'atari' subdirectory and can be edited and copied to the corresponding files in the main source directory. Even if `configure` produced something, it might be advisable to compare its results with the sample versions and possibly make adjustments.

Some `gawk` source code fragments depend on a preprocessor define 'atarist'. This basically assumes the TOS environment with `gcc`. Modify these sections as appropriate if they are not right for your environment. Also see the remarks about `AWKPATH` and `envsep` in B.5.2. Bekezdés [Running gawk on the Atari ST], 288. oldal.

As shipped, the sample 'config.h' claims that the `system` function is missing from the libraries, which is not true, and an alternative implementation of this function is provided in 'atari/system.c'. Depending upon your particular combination of shell and operating system, you may wish to change the file to indicate that `system` is available.

B.5.2. Running gawk on the Atari ST

An executable version of `gawk` should be placed, as usual, anywhere in your `PATH` where your shell can find it.

While executing, `gawk` creates a number of temporary files. When using `gcc` libraries for TOS, `gawk` looks for either of the environment variables `TEMP` or `TMPDIR`, in that order. If either one is found, its value is assumed to be a directory for temporary files. This directory must exist, and if you can spare the memory, it is a good idea to put it on a RAM drive. If neither `TEMP` nor `TMPDIR` are found, then `gawk` uses the current directory for its temporary files.

The ST version of `gawk` searches for its program files as described in 14.3. Bekezdés [The AWKPATH Environment Variable], 166. oldal. The default value for the `AWKPATH` variable is taken from `DEFPATH` defined in 'Makefile'. The sample `gcc`/TOS 'Makefile' for the ST in the distribution sets `DEFPATH` to `".,c:\lib\awk,c:\gnu\lib\awk"`. The search path can be modified by explicitly setting `AWKPATH` to whatever you wish. Note that colons cannot be used on the ST to separate elements in the `AWKPATH` variable, since they have another, reserved, meaning. Instead, you must use a comma to separate elements in the path. When recompiling, the separating character can be modified by initializing the `envsep` variable in 'atari/gawkmisc.atr' to another value.

Although `awk` allows great flexibility in doing I/O redirections from within a program, this facility should be used with care on the ST running under TOS. In some circumstances the OS routines for file handle pool processing lose track of certain events, causing the computer to crash, and requiring a reboot. Often a warm reboot is sufficient. Fortunately, this happens infrequently, and in rather esoteric situations. In particular, avoid having one part of an `awk` program using `print` statements explicitly redirected to "/dev/stdout", while other print statements use the default standard output, and a calling shell has redirected standard output to a file.

When gawk is compiled with the ST version of gcc and its usual libraries, it will accept both '/' and '\' as path separators. While this is convenient, it should be remembered that this removes one, technically valid, character ('/') from your file names, and that it may create problems for external programs, called via the system function, which may not support this convention. Whenever it is possible that a file created by gawk will be used by some other program, use only backslashes. Also remember that in awk, backslashes in strings have to be doubled in order to get literal backslashes (lásd 4.2. Bekezdés [Escape Sequences], 24. oldal).

B.6. Installing gawk on an Amiga

You can install gawk on an Amiga system using a Unix emulation environment available via anonymous ftp from ftp.ninemoons.com in the directory 'pub/ade/current'. This includes a shell based on pdksh. The primary component of this environment is a Unix emulation library, 'ixemul.lib'.

A more complete distribution for the Amiga is available on the Geek Gadgets CD-ROM from:

CRONUS
1840 E. Warner Road #105-265
Tempe, AZ 85284 USA
US Toll Free: (800) 804-0833
Phone: +1-602-491-0442
FAX: +1-602-491-0048
Email: info@ninemoons.com
WWW: http://www.ninemoons.com
Anonymous ftp site: ftp.ninemoons.com

Once you have the distribution, you can configure gawk simply by running configure:

```
configure -v m68k-amigaos
```

Then run make, and you should be all set! (If these steps do not work, please send in a bug report; lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal.)

B.7. Reporting Problems and Bugs

There is nothing more dangerous than a bored archeologist.
The Hitchhiker's Guide to the Galaxy

If you have problems with `gawk` or think that you have found a bug, please report it to the developers; we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you have actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible `awk` program and input data file that reproduces the problem. Then send us the program and data file, some idea of what kind of Unix system you're using, and the exact results `gawk` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you have a precise problem, there are two e-mail addresses you can send mail to.

Internet: '`bug-gnu-utils@gnu.org`'

UUCP: '`uunet!gnu.org!bug-gnu-utils`'

Please include the version number of `gawk` you are using. You can get this information with the command '`gawk --version`'. You should send a carbon copy of your mail to Arnold Robbins, who can be reached at '`arnold@gnu.org`'.

Important! Do *not* try to report bugs in `gawk` by posting to the Usenet/Internet newsgroup `comp.lang.awk`. While the `gawk` developers do occasionally read this newsgroup, there is no guarantee that we will see your posting. The steps described above are the official, recognized ways for reporting bugs.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, ask Arnold Robbins; he will try to help you out, although he may not have the time to fix the problem. You can send him electronic mail at the Internet address above.

If you find bugs in one of the non-Unix ports of `gawk`, please send an electronic mail message to the person who maintains that port. They are listed below, and also in the '`README`' file in the `gawk` distribution. Information in the '`README`' file should be considered authoritative if it conflicts with this könyv.

The people maintaining the non-Unix ports of gawk are:

MS-DOS
: Scott Deifik, 'scottd@amgen.com', and Darrel Hankerson, 'hankedr@mail.auburn.edu'.

OS/2
: Kai Uwe Rommel, 'rommel@ars.de'.

VMS
: Pat Rankin, 'rankin@eql.caltech.edu'.

Atari ST
: Michal Jaegermann, 'michal@gortel.phys.ualberta.ca'.

Amiga
: Fred Fish, 'fnf@ninemoons.com'.

If your bug is also reproducible under Unix, please send copies of your report to the general GNU bug list, as well as to Arnold Robbins, at the addresses listed above.

B.8. Other Freely Available awk Implementations

It's kind of fun to put comments like this in your awk code.
    `// Do C++ comments work? answer: yes! of course`
Michael Brennan

There are two other freely available awk implementations. This section briefly describes where to get them.

Unix awk
: Brian Kernighan has been able to make his implementation of awk freely available. You can get it via anonymous ftp to the host netlib.bell-labs.com. Change directory to '/netlib/research'. Use "binary" or "image" mode, and retrieve 'awk.bundle.gz'.

  This is a shell archive that has been compressed with the GNU gzip utility. It can be uncompressed with the gunzip utility.

  You can also retrieve this version via the World Wide Web from Brian Kernighan's home page (http://cm.bell-labs.com/who/bwk).

  This version requires an ANSI C compiler; GCC (the GNU C compiler) works quite nicely.

mawk
: Michael Brennan has written an independent implementation of awk, called mawk. It is available under the GPL (lásd [GNU GENERAL PUBLIC LICENSE], 307. oldal), just as gawk is.

  You can get it via anonymous ftp to the host ftp.whidbey.net. Change directory to '/pub/brennan'. Use "binary" or "image" mode, and retrieve 'mawk1.3.3.tar.gz' (or the latest version that is there).

  gunzip may be used to decompress this file. Installation is similar to gawk's (lásd B.2. Bekezdés [Compiling and Installing gawk on Unix], 282. oldal).

C. Függelék Implementation Notes

This appendix contains information mainly of interest to implementors and maintainers of `gawk`. Everything in it applies specifically to `gawk`, and not to other implementations.

C.1. Downward Compatibility and Debugging

Lásd 17.5. Bekezdés [Extensions in `gawk` Not in POSIX `awk`], 253. oldal, for a summary of the GNU extensions to the `awk` language and program. All of these features can be turned off by invoking `gawk` with the '`--traditional`' option, or with the '`--posix`' option.

If `gawk` is compiled for debugging with '`-DDEBUG`', then there is one more option available on the command line:

`-W parsedebug`
`--parsedebug`
: Print out the parse stack information as the program is being parsed.

This option is intended only for serious `gawk` developers, and not for the casual user. It probably has not even been compiled into your version of `gawk`, since it slows down execution.

C.2. Making Additions to `gawk`

If you should find that you wish to enhance `gawk` in a significant fashion, you are perfectly free to do so. That is the point of having free software; the source code is available, and you are free to change it as you wish (lásd [GNU GENERAL PUBLIC LICENSE], 307. oldal).

This section discusses the ways you might wish to change `gawk`, and any considerations you should bear in mind.

C.2.1. Adding New Features

You are free to add any new features you like to `gawk`. However, if you want your changes to be incorporated into the `gawk` distribution, there are several steps that you need to take in order to make it possible for me to include your changes.

1. Get the latest version. It is much easier for me to integrate changes if they are relative to the most recent distributed version of `gawk`. If your version of `gawk` is very old, I may not be able to integrate them at all. Lásd B.1.1. Bekezdés [Getting the `gawk` Distribution], 277. oldal, for information on getting the latest version of `gawk`.

2. Follow the *GNU Coding Standards*. This document describes how GNU software should be written. If you haven't read it, please do so, preferably *before* starting to modify `gawk`. (The *GNU Coding Standards* are available as part of the Autoconf distribution, from the FSF.)
3. Use the `gawk` coding style. The C code for `gawk` follows the instructions in the *GNU Coding Standards*, with minor exceptions. The code is formatted using the traditional "K&R" style, particularly as regards the placement of braces and the use of tabs. In brief, the coding rules for `gawk` are:
   - Use old style (non-prototype) function headers when defining functions.
   - Put the name of the function at the beginning of its own line.
   - Put the return type of the function, even if it is `int`, on the line above the line with the name and arguments of the function.
   - The declarations for the function arguments should not be indented.
   - Put spaces around parentheses used in control structures (`if`, `while`, `for`, `do`, `switch` and `return`).
   - Do not put spaces in front of parentheses used in function calls.
   - Put spaces around all C operators, and after commas in function calls.
   - Do not use the comma operator to produce multiple side-effects, except in `for` loop initialization and increment parts, and in macro bodies.
   - Use real tabs for indenting, not spaces.
   - Use the "K&R" brace layout style.
   - Use comparisons against `NULL` and `'\0'` in the conditions of `if`, `while` and `for` statements, and in the `case`s of `switch` statements, instead of just the plain pointer or character value.
   - Use the `TRUE`, `FALSE`, and `NULL` symbolic constants, and the character constant `'\0'` where appropriate, instead of `1` and `0`.
   - Provide one-line descriptive comments for each function.
   - Do not use '`#elif`'. Many older Unix C compilers cannot handle it.
   - Do not use the `alloca` function for allocating memory off the stack. Its use causes more portability trouble than the minor benefit of not having to free the storage. Instead, use `malloc` and `free`.

   If I have to reformat your code to follow the coding style used in `gawk`, I may not bother.
4. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your changes, you must either place those changes in the public domain, and submit a signed statement to that effect, or assign

the copyright in your changes to the FSF. Both of these actions are easy to do, and *many* people have done so already. If you have questions, please contact me (lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal), or `gnu@gnu.org`.

5. Update the documentation. Along with your new code, please supply new sections and or chapters for this könyv. If at all possible, please use real Texinfo, instead of just supplying unformatted ASCII text (although even that is better than no documentation at all). Conventions to be followed in *Hatékony AWK programozás* are provided after the '`@bye`' at the end of the Texinfo source file. If possible, please update the man page as well.

   You will also have to sign paperwork for your documentation changes.

6. Submit changes as context diffs or unified diffs. Use '`diff -c -r -N`' or '`diff -u -r -N`' to compare the original `gawk` source tree with your version. (I find context diffs to be more readable, but unified diffs are more compact.) I recommend using the GNU version of `diff`. Send the output produced by either run of `diff` to me when you submit your changes. Lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal, for the electronic mail information.

   Using this format makes it easy for me to apply your changes to the master version of the `gawk` source code (using `patch`). If I have to apply the changes manually, using a text editor, I may not do so, particularly if there are lots of changes.

Although this sounds like a lot of work, please remember that while you may write the new code, I have to maintain it and support it, and if it isn't possible for me to do that with a minimum of extra work, then I probably will not.

C.2.2. Porting `gawk` to a New Operating System

If you wish to port `gawk` to a new operating system, there are several steps to follow.

1. Follow the guidelines in C.2.1. Bekezdés [Adding New Features], 293. oldal, concerning coding style, submission of diffs, and so on.

2. When doing a port, bear in mind that your code must co-exist peacefully with the rest of `gawk`, and the other ports. Avoid gratuitous changes to the system-independent parts of the code. If at all possible, avoid sprinkling '`#ifdef`'s just for your port throughout the code.

   If the changes needed for a particular system affect too much of the code, I probably will not accept them. In such a case, you will, of course, be able to distribute your changes on your own, as long as you comply with the GPL (lásd [GNU GENERAL PUBLIC LICENSE], 307. oldal).

3. A number of the files that come with `gawk` are maintained by other people at the Free Software Foundation. Thus, you should not

change them unless it is for a very good reason. I.e. changes are not out of the question, but changes to these files will be scrutinized extra carefully. The files are '`alloca.c`', '`getopt.h`', '`getopt.c`', '`getopt1.c`', '`regex.h`', '`regex.c`', '`dfa.h`', '`dfa.c`', '`install-sh`', and '`mkinstalldirs`'.

4. Be willing to continue to maintain the port. Non-Unix operating systems are supported by volunteers who maintain the code needed to compile and run `gawk` on their systems. If no-one volunteers to maintain a port, that port becomes unsupported, and it may be necessary to remove it from the distribution.
5. Supply an appropriate '`gawkmisc.???`' file. Each port has its own '`gawkmisc.???`' that implements certain operating system specific functions. This is cleaner than a plethora of '`#ifdef`'s scattered throughout the code. The '`gawkmisc.c`' in the main source directory includes the appropriate '`gawkmisc.???`' file from each subdirectory. Be sure to update it as well.

   Each port's '`gawkmisc.???`' file has a suffix reminiscent of the machine or operating system for the port. For example, '`pc/gawkmisc.pc`' and '`vms/gawkmisc.vms`'. The use of separate suffixes, instead of plain '`gawkmisc.c`', makes it possible to move files from a port's subdirectory into the main subdirectory, without accidentally destroying the real '`gawkmisc.c`' file. (Currently, this is only an issue for the MS-DOS and OS/2 ports.)
6. Supply a '`Makefile`' and any other C source and header files that are necessary for your operating system. All your code should be in a separate subdirectory, with a name that is the same as, or reminiscent of, either your operating system or the computer system. If possible, try to structure things so that it is not necessary to move files out of the subdirectory into the main source directory. If that is not possible, then be sure to avoid using names for your files that duplicate the names of files in the main source directory.
7. Update the documentation. Please write a section (or sections) for this könyv describing the installation and compilation steps needed to install and/or compile `gawk` for your system.
8. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your code, you must either place your code in the public domain, and submit a signed statement to that effect, or assign the copyright in your code to the FSF.

Following these steps will make it much easier to integrate your changes into `gawk`, and have them co-exist happily with the code for other operating systems that is already there.

In the code that you supply, and that you maintain, feel free to use a coding style and brace layout that suits your taste.

C.3. Probable Future Extensions

AWK is a language similar to PERL, only considerably more elegant.
Arnold Robbins

Hey!
Larry Wall

This section briefly lists extensions and possible improvements that indicate the directions we are currently considering for `gawk`. The file '`FUTURES`' in the `gawk` distributions lists these extensions as well.

This is a list of probable future changes that will be usable by the `awk` language programmer.

Localization
: The GNU project is starting to support multiple languages. It will at least be possible to make `gawk` print its warnings and error messages in languages other than English. It may be possible for `awk` programs to also use the multiple language facilities, separate from `gawk` itself.

Databases
: It may be possible to map a GDBM/NDBM/SDBM file into an `awk` array.

A `PROCINFO` Array
: The special files that provide process-related information (lásd 6.7. Bekezdés [Special File Names in gawk], 71. oldal) may be superseded by a `PROCINFO` array that would provide the same information, in an easier to access fashion.

More `lint` warnings
: There are more things that could be checked for portability.

Control of subprocess environment
: Changes made in `gawk` to the array `ENVIRON` may be propagated to subprocesses run by `gawk`.

This is a list of probable improvements that will make `gawk` perform better.

An Improved Version of `dfa`
: The `dfa` pattern matcher from GNU `grep` has some problems. Either a new version or a fixed one will deal with some important regexp matching issues.

Use of GNU `malloc`
: The GNU version of `malloc` could potentially speed up `gawk`, since it relies heavily on the use of dynamic memory allocation.

C.4. Suggestions for Improvements

Here are some projects that would-be `gawk` hackers might like to take on. They vary in size from a few days to a few weeks of programming, depending on which one you choose and how fast a programmer you are. Please send any improvements you write to the maintainers at the GNU project. Lásd C.2.1. Bekezdés [Adding New Features], 293. oldal, for guidelines to follow when adding new features to `gawk`. Lásd B.7. Bekezdés [Reporting Problems and Bugs], 289. oldal, for information on contacting the maintainers.

1. Compilation of `awk` programs: `gawk` uses a Bison (YACC-like) parser to convert the script given it into a syntax tree; the syntax tree is then executed by a simple recursive evaluator. This method incurs a lot of overhead, since the recursive evaluator performs many procedure calls to do even the simplest things.

   It should be possible for `gawk` to convert the script's parse tree into a C program which the user would then compile, using the normal C compiler and a special `gawk` library to provide all the needed functions (regexps, fields, associative arrays, type coercion, and so on).

   An easier possibility might be for an intermediate phase of `awk` to convert the parse tree into a linear byte code form like the one used in GNU Emacs Lisp. The recursive evaluator would then be replaced by a straight line byte code interpreter that would be intermediate in speed between running a compiled program and doing what `gawk` does now.
2. The programs in the test suite could use documenting in this könyv.
3. See the '`FUTURES`' file for more ideas. Contact us if you would seriously like to tackle any of the items listed there.

D. Függelék Glossary

Action
: A series of `awk` statements attached to a rule. If the rule's pattern matches an input record, `awk` executes the rule's action. Actions are always enclosed in curly braces. Lásd 8.2. Bekezdés [Overview of Actions], 102. oldal.

Amazing `awk` Assembler
: Henry Spencer at the University of Toronto wrote a retargetable assembler completely as `awk` scripts. It is thousands of lines long, including machine descriptions for several eight-bit microcomputers. It is a good example of a program that would have been better written in another language.

Amazingly Workable Formatter (`awf`)
: Henry Spencer at the University of Toronto wrote a formatter that accepts a large subset of the '`nroff -ms`' and '`nroff -man`' formatting commands, using `awk` and `sh`.

ANSI
: The American National Standards Institute. This organization produces many standards, among them the standards for the C and C++ programming languages.

Assignment
: An `awk` expression that changes the value of some `awk` variable or data object. An object that you can assign to is called an *lvalue*. The assigned values are called *rvalues*. Lásd 7.7. Bekezdés [Assignment Expressions], 84. oldal.

`awk` Language
: The language in which `awk` programs are written.

`awk` Program
: An `awk` program consists of a series of *patterns* and *actions*, collectively known as *rules*. For each input record given to the program, the program's rules are all processed in turn. `awk` programs may also contain function definitions.

`awk` Script
: Another name for an `awk` program.

Bash
: The GNU version of the standard shell (the Bourne-Again shell). See "Bourne Shell."

BBS
: See "Bulletin Board System."

Boolean Expression
: Named after the English mathematician Boole. See "Logical Expression."

Bourne Shell
: The standard shell ('`/bin/sh`') on Unix and Unix-like systems, originally written by Steven R. Bourne. Many shells (Bash, `ksh`,

pdksh, zsh) are generally upwardly compatible with the Bourne shell.

Built-in Function
: The `awk` language provides built-in functions that perform various numerical, time stamp related, and string computations. Examples are `sqrt` (for the square root of a number) and `substr` (for a substring of a string). Lásd 12. Fejezet [Built-in Functions], 133. oldal.

Built-in Variable
: `ARGC`, `ARGIND`, `ARGV`, `CONVFMT`, `ENVIRON`, `ERRNO`, `FIELDWIDTHS`, `FILENAME`, `FNR`, `FS`, `IGNORECASE`, `NF`, `NR`, `OFMT`, `OFS`, `ORS`, `RLENGTH`, `RSTART`, `RS`, `RT`, and `SUBSEP`, are the variables that have special meaning to `awk`. Changing some of them affects `awk`'s running environment. Several of these variables are specific to `gawk`. Lásd 10. Fejezet [Built-in Variables], 115. oldal.

Braces
: See "Curly Braces."

Bulletin Board System
: A computer system allowing users to log in and read and/or leave messages for other users of the system, much like leaving paper notes on a bulletin board.

C
: The system programming language that most GNU software is written in. The `awk` programming language has C-like syntax, and this könyv points out similarities between `awk` and C when appropriate.

Character Set
: The set of numeric codes used by a computer system to represent the characters (letters, numbers, punctuation, etc.) of a particular country or place. The most common character set in use today is ASCII (American Standard Code for Information Interchange). Many European countries use an extension of ASCII known as ISO-8859-1 (ISO Latin-1).

CHEM
: A preprocessor for `pic` that reads descriptions of molecules and produces `pic` input for drawing them. It was written in `awk` by Brian Kernighan and Jon Bentley, and is available from `netlib@research.bell-labs.com`.

Compound Statement
: A series of `awk` statements, enclosed in curly braces. Compound statements may be nested. Lásd 9. Fejezet [Control Statements in Actions], 105. oldal.

Concatenation
: Concatenating two strings means sticking them together, one after another, giving a new string. For example, the string '`foo`'

concatenated with the string '`bar`' gives the string '`foobar`'. Lásd 7.6. Bekezdés [String Concatenation], 83. oldal.

Conditional Expression
: An expression using the '`?:`' ternary operator, such as '*expr1* `?` *expr2* `:` *expr3*'. The expression *expr1* is evaluated; if the result is true, the value of the whole expression is the value of *expr2*, otherwise the value is *expr3*. In either case, only one of *expr2* and *expr3* is evaluated. Lásd 7.12. Bekezdés [Conditional Expressions], 92. oldal.

Comparison Expression
: A relation that is either true or false, such as '`(a < b)`'. Comparison expressions are used in `if`, `while`, `do`, and `for` statements, and in patterns to select which input records to process. Lásd 7.10. Bekezdés [Variable Typing and Comparison Expressions], 88. oldal.

Curly Braces
: The characters '`{`' and '`}`'. Curly braces are used in `awk` for delimiting actions, compound statements, and function bodies.

Dark Corner
: An area in the language where specifications often were (or still are) not clear, leading to unexpected or undesirable behavior. Such areas are marked in this könyv with "(d.c.)" in the text, and are indexed under the heading "dark corner."

Data Objects
: These are numbers and strings of characters. Numbers are converted into strings and vice versa, as needed. Lásd 7.4. Bekezdés [Conversion of Strings and Numbers], 81. oldal.

Double Precision
: An internal representation of numbers that can have fractional parts. Double precision numbers keep track of more digits than do single precision numbers, but operations on them are more expensive. This is the way `awk` stores numeric values. It is the C type `double`.

Dynamic Regular Expression
: A dynamic regular expression is a regular expression written as an ordinary expression. It could be a string constant, such as `"foo"`, but it may also be an expression whose value can vary. Lásd 4.7. Bekezdés [Using Dynamic Regexps], 35. oldal.

Environment
: A collection of strings, of the form *name*`=`*val*, that each program has available to it. Users generally place values into the environment in order to provide information to various programs. Typical examples are the environment variables `HOME` and `PATH`.

Empty String
: See “Null String.”

Escape Sequences
: A special sequence of characters used for describing non-printing characters, such as ‘`\n`’ for newline, or ‘`\033`’ for the ASCII ESC (escape) character. Lásd 4.2. Bekezdés [Escape Sequences], 24. oldal.

Field
: When `awk` reads an input record, it splits the record into pieces separated by whitespace (or by a separator regexp which you can change by setting the built-in variable `FS`). Such pieces are called fields. If the pieces are of fixed length, you can use the built-in variable `FIELDWIDTHS` to describe their lengths. Lásd 5.5. Bekezdés [Specifying How Fields are Separated], 44. oldal, and also see Lásd 5.6. Bekezdés [Reading Fixed-width Data], 49. oldal.

Floating Point Number
: Often referred to in mathematical terms as a “rational” number, this is just a number that can have a fractional part. See “Double Precision” and “Single Precision.”

Format
: Format strings are used to control the appearance of output in the `printf` statement. Also, data conversions from numbers to strings are controlled by the format string contained in the built-in variable `CONVFMT`. Lásd 6.5.2. Bekezdés [Format-Control Letters], 65. oldal.

Function
: A specialized group of statements used to encapsulate general or program-specific tasks. `awk` has a number of built-in functions, and also allows you to define your own. Lásd 12. Fejezet [Built-in Functions], 133. oldal, and 13. Fejezet [User-defined Functions], 153. oldal.

FSF
: See “Free Software Foundation.”

Free Software Foundation
: A non-profit organization dedicated to the production and distribution of freely distributable software. It was founded by Richard M. Stallman, the author of the original Emacs editor. GNU Emacs is the most widely used version of Emacs today.

`gawk`
: The GNU implementation of `awk`.

General Public License
: This document describes the terms under which `gawk` and its source code may be distributed. (lásd [GNU GENERAL PUBLIC LICENSE], 307. oldal)

GNU
: "GNU's not Unix". An on-going project of the Free Software Foundation to create a complete, freely distributable, POSIX-compliant computing environment.

GPL
: See "General Public License."

Hexadecimal
: Base 16 notation, where the digits are `0-9` and `A-F`, with '`A`' representing 10, '`B`' representing 11, and so on up to '`F`' for 15. Hexadecimal numbers are written in C using a leading '`0x`', to indicate their base. Thus, `0x12` is 18 (one times 16 plus 2).

I/O
: Abbreviation for "Input/Output," the act of moving data into and/or out of a running program.

Input Record
: A single chunk of data read in by `awk`. Usually, an `awk` input record consists of one line of text. Lásd 5.1. Bekezdés [How Input is Split into Records], 37. oldal.

Integer
: A whole number, i.e. a number that does not have a fractional part.

Keyword
: In the `awk` language, a keyword is a word that has special meaning. Keywords are reserved and may not be used as variable names.

  `gawk`'s keywords are: `BEGIN`, `END`, `if`, `else`, `while`, `do...while`, `for`, `for...in`, `break`, `continue`, `delete`, `next`, `nextfile`, `function`, `func`, and `exit`.

Logical Expression
: An expression using the operators for logic, AND, OR, and NOT, written '`&&`', '`||`', and '`!`' in `awk`. Often called Boolean expressions, after the mathematician who pioneered this kind of mathematical logic.

Lvalue
: An expression that can appear on the left side of an assignment operator. In most languages, lvalues can be variables or array elements. In `awk`, a field designator can also be used as an lvalue.

Null String
: A string with no characters in it. It is represented explicitly in `awk` programs by placing two double-quote characters next to each other (`""`). It can appear in input data by having two successive occurrences of the field separator appear next to each other.

Number
: A numeric valued data object. The `gawk` implementation uses double precision floating point to represent numbers. Very old `awk` implementations use single precision floating point.

Octal
: Base-eight notation, where the digits are `0-7`. Octal numbers are written in C using a leading '`0`', to indicate their base. Thus, `013` is 11 (one times 8 plus 3).

Pattern
: Patterns tell `awk` which input records are interesting to which rules.

  A pattern is an arbitrary conditional expression against which input is tested. If the condition is satisfied, the pattern is said to *match* the input record. A typical pattern might compare the input record against a regular expression. Lásd 8.1. Bekezdés [Pattern Elements], 97. oldal.

POSIX
: The name for a series of standards being developed by the IEEE that specify a Portable Operating System interface. The "IX" denotes the Unix heritage of these standards. The main standard of interest for `awk` users is *IEEE Standard for Information Technology, Standard 1003.2-1992, Portable Operating System Interface (POSIX) Part 2: Shell and Utilities*. Informally, this standard is often referred to as simply "P1003.2."

Private
: Variables and/or functions that are meant for use exclusively by library functions, and not for the main `awk` program. Special care must be taken when naming such variables and functions. Lásd 15.13. Bekezdés [Naming Library Function Global Variables], 202. oldal.

Range (of input lines)
: A sequence of consecutive lines from the input file. A pattern can specify ranges of input lines for `awk` to process, or it can specify single lines. Lásd 8.1. Bekezdés [Pattern Elements], 97. oldal.

Recursion
: When a function calls itself, either directly or indirectly. If this isn't clear, refer to the entry for "recursion."

Redirection
: Redirection means performing input from other than the standard input stream, or output to other than the standard output stream.

  You can redirect the output of the `print` and `printf` statements to a file or a system command, using the '`>`', '`>>`', and '`|`' operators. You can redirect input to the `getline` statement using the '`<`' and '`|`' operators. Lásd 6.6. Bekezdés [Redirecting Output of `print` and `printf`], 70. oldal, and 5.8. Bekezdés [Explicit Input with `getline`], 53. oldal.

Regexp
: Short for *regular expression*. A regexp is a pattern that denotes a set of strings, possibly an infinite set. For example, the regexp '`R.*xp`' matches any string starting with the letter '`R`' and ending with the letters '`xp`'. In `awk`, regexps are used in patterns and in

conditional expressions. Regexps may contain escape sequences. Lásd 4. Fejezet [Regular Expressions], 23. oldal.

Regular Expression
: See "regexp."

Regular Expression Constant
: A regular expression constant is a regular expression written within slashes, such as `/foo/`. This regular expression is chosen when you write the `awk` program, and cannot be changed doing its execution. Lásd 4.1. Bekezdés [How to Use Regular Expressions], 23. oldal.

Rule
: A segment of an `awk` program that specifies how to process single input records. A rule consists of a *pattern* and an *action*. `awk` reads an input record; then, for each rule, if the input record satisfies the rule's pattern, `awk` executes the rule's action. Otherwise, the rule does nothing for that input record.

Rvalue
: A value that can appear on the right side of an assignment operator. In `awk`, essentially every expression has a value. These values are rvalues.

`sed`
: See "Stream Editor."

Short-Circuit
: The nature of the `awk` logical operators '`&&`' and '`||`'. If the value of the entire expression can be deduced from evaluating just the left-hand side of these operators, the right-hand side will not be evaluated (lásd 7.11. Bekezdés [Boolean Expressions], 90. oldal).

Side Effect
: A side effect occurs when an expression has an effect aside from merely producing a value. Assignment expressions, increment and decrement expressions and function calls have side effects. Lásd 7.7. Bekezdés [Assignment Expressions], 84. oldal.

Single Precision
: An internal representation of numbers that can have fractional parts. Single precision numbers keep track of fewer digits than do double precision numbers, but operations on them are less expensive in terms of CPU time. This is the type used by some very old versions of `awk` to store numeric values. It is the C type `float`.

Space
: The character generated by hitting the space bar on the keyboard.

Special File
: A file name interpreted internally by `gawk`, instead of being handed directly to the underlying operating system. For example,

'/dev/stderr'. Lásd 6.7. Bekezdés [Special File Names in gawk], 71. oldal.

Stream Editor
: A program that reads records from an input stream and processes them one or more at a time. This is in contrast with batch programs, which may expect to read their input files in entirety before starting to do anything, and with interactive programs, which require input from the user.

String
: A datum consisting of a sequence of characters, such as 'I am a string'. Constant strings are written with double-quotes in the awk language, and may contain escape sequences. Lásd 4.2. Bekezdés [Escape Sequences], 24. oldal.

Tab
: The character generated by hitting the *TAB* key on the keyboard. It usually expands to up to eight spaces upon output.

Unix
: A computer operating system originally developed in the early 1970's at AT&T Bell Laboratories. It initially became popular in universities around the world, and later moved into commercial evnironments as a software development system and network server system. There are many commercial versions of Unix, as well as several work-alike systems whose source code is freely available (such as Linux, NetBSD, and FreeBSD).

Whitespace
: A sequence of space, tab, or newline characters occurring inside an input record or a string.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary.

To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for non-commercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EX-

TENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place --- Suite 330, Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Tárgymutató

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

U

V

W

Rövid tartalomjegyzék

Tartalomjegyzék