

Az awk

| | |
|---|----|
| 1. Bevezető..... | 1 |
| 2. Rekordok és mezők | 2 |
| 3. Az awk programozási nyelv | 3 |
| Az awk minta | 3 |
| Az utasítások | 4 |
| A változók..... | 5 |
| Belső változók | 5 |
| Konstansok..... | 6 |
| Operátorok..... | 7 |
| Programszervező utasítások | 8 |
| Függvények..... | 9 |
| A fontosabb numerikus függvények..... | 9 |
| Fontosabb sztring függvények..... | 9 |
| Fontosabb ki és bemeneti függvények | 10 |
| Egysoros példaprogramok..... | 11 |
| Több állomány egyidejű feldolgozása és a getline függvény | 11 |
| Külső parancsok futtatása..... | 12 |
| Saját függvények | 12 |
| Tömbök az awk-ban..... | 14 |
| Tömbök létrehozása a split függvénnyel..... | 16 |
| Bibliográfia..... | 17 |

1. Bevezető

Az awk egy programozási nyelv, szerzői "*awk, a pattern scanning and processing language*": minta elemző és feldolgozó nyelv nevet adták neki. A nyelv tulajdonképpen bármilyen felépítésű szövegből minták alapján sorokat vagy mezőket tud kiemelni, és azokkal műveleteket tud végezni. Így akár a UNIX programok kimenetét (pl. az `ls -l` parancs kimenetét) akár adatokat tartalmazó mezőkre bontható állományok (pl. a `.csv` típusú állományok) soraiból ki lehet vele hámozni a reá bízott feladat megoldásához szükséges mezőket. Ezekkel különböző matematikai illetve szöveges műveleteket tud végezni. A nyelv programozási szerkezetei a C nyelv utasításaira épülnek. Nevét a szerzők nevének kezdőbetűiből állították össze: Aho – Weinberger – Kernighan. Az awk rendkívüli módon megkönnyíti a szöveges állományok feldolgozását. A mérnöki munkában nyers adatok elő feldolgozását végezhetjük vele könnyen. A UNIX/Linux alatti rendszerprogramozásban a parancsok kimenetét (pl. az `ls -l` parancs szöveges kimenetét) dolgozhatjuk fel könnyen.

Mivel szkript nyelv és igen tömör, kis mindennapi feladatok megoldását végezhetjük el vele gyorsan. A nyelv szerzői nem ajánlják a nyelvet nagy és bonyolult feladatok megoldásához: ennek ellenére, elég sok típusú feladatot lehet vele megoldani, különösen azokat, amelyek rendszerprogramozáshoz közeliek, és pl. valós számokkal is kell számolni.

A segédlet az awk fontosabb elemeit mutatja be. Kimerítő tanulmányozásához a felhasználói kézikönyv vagy egyéb dokumentációk tanulmányozása szükséges (bibliográfia a dokumentum végén. A felhasználói kézikönyvnek van magyar fordítása). Ebben a rövid bemutatóban a hangsúly a feldolgozóprogram elvének megértésén van. Az awk-nak több változata van, az alábbiak a GNU awk-ra érvényesek Linux alatt.

A program indítása hasonló a `sed` -éhez, a program végigolvassa a bemenetét (a standard input jelenti az implicit bemenetet), feldolgozza azt és a kimenetre ír. A `-f` kapcsoló ugyanúgy a szkript állományra hivatkozik, de a szkript megjelenhet az első paraméterben is idézőjelek között.

Az általunk használt kapcsolók:

`-F c` *Field separator*: megállapítja a mezőelválasztó karaktert. Ha nem adjuk meg, akkor az implicit elválasztó a szóköz és a TAB, illetve ezek ismétlődése (tulajdonképpen egy reguláris kifejezés minta: `[\t]+`).

`-f file` a szkript állomány megadása

`-v var=val` a `var` változónak `val` értéket ad indulás előtt

Az `awk` nyelven írt hosszabb programokat írhatjuk

```
#!/usr/bin/awk -f
```

sorral kezdődően külön állományba, melynek jó ha a `.awk` kiterjesztést adjuk megkülönböztetés végett.

2. Rekordok és mezők

Működési elv

Az `awk` a következő feldolgozást alkalmazza a bemeneti sorokra: úgy tekinti a sorokat, mint egy elválasztó karakterrel (vagy karakter szekvenciával) határolt, **mezőkből** álló sor. Egy sornak a neve **rekord** (ez a feldolgozási egység). Pl. az alábbi sor (vagy rekord) esetén:

```
112 LinkinPark Numb numb.txt 35
```

a sornak 5 mezője van, az elválasztó egy szóköz. A sor feldolgozása közben a nyelv a mezőkre a `$1`, `$2`, ... változókkal hivatkozik. Pl. a `$3` értéke `Numb`. A teljes sorra a `$0` változóval lehet hivatkozni.

Ugyanakkor a mezőkre az alábbi szerkezetű változókon keresztül is lehet hivatkozni, a:

```
k=3
print $k
```

a harmadik mező értékét fogja kinyomtatni (lásd alább hogyan definiáljuk a változókat).

A sorok számát az `awk` az `NR` (Number of Rows - az összes eddig beolvasott sor) és `FNR` (az aktuális bemeneti állomány, ha több van) változóban követi.

A mezők száma az `FN` változóban van, amely az utolsó mező címzésére is használható: `$FN`. Az alábbi példa:

```
print $FN
```

az utolsó mezőt nyomtatja, függetlenül attól, hogy hány van. A mezők címzésére kifejezések is használhatóak:

```
print $(2+2)
```

a negyedik mezőt fogja nyomtatni.

Használat közben az `awk` által előállított mezőknek meg lehet változtatni az értékét:

```
echo 'a b' | awk '{$1="b"; print $0}'
```

3. Az awk programozási nyelv

Az awk programsorok szabályokból állanak: egy szabály egy mintából (*pattern*) és a hozzá tartozó tevékenységből (*action*) vagy utasításokból (*statement*) áll, amely {} közé zárt blokkban van:

```
minta {utasítások}
```

A {} blokkban utasítások követik egymást.

Az awk függvényei közül a legegyszerűbb a `print` : kinyomtatja a \$0 változó tartalmát egy sort.

```
{print}
```

Ha argumentumokkal használjuk, a `print` függvény a felsorolt változókat kiírja a kimenetre.

```
$ echo '112 LinkinPark Numb numb.txt 35' | awk '{print $1, $4}'
112 numb.txt
```

Akár a `sed`, normális menetben az `awk` egy-egy sort olvas a \$0 változóba, feldolgozza azt: azaz alkalmazza reá szabályait ha a szabályhoz rendelt cím kiválaszta a sort. Ezt a műveletet ismétli amíg a bemenetről elfogynak a sorok.

Az `awk` mintái bővített reguláris kifejezéseket használnak, pl. a:

```
/^a/
```

azokat a sorokat fogja kiválasztani amelyek 'a' -val kezdődnek.

Az `awk` szabályban a mintának vagy tevékenységnek meg kell jelennie. Ha a minta elmarad, a tevékenységet minden bemeneti sorra alkalmazza az `awk`, ha a tevékenység marad el, akkor az implicit tevékenység a `print $0` , azaz a bemeneti sor nyomtatása.

Így:

```
$ echo alma | awk '/^a/{print $0}'
alma
```

kinyomtatja az egyetlen bemeneti sort, mert a minta illeszkedik a sorra,

```
$ echo alma | awk '/^a/'
alma
```

ugyanazt teszi, mert elhagytuk a tevékenységet. Ugyanakkor:

```
$ echo alma | awk '{print $0}'
alma
```

is kinyomtatja a bemeneti sort, mert nincs minta, tehát minden sorra illeszkedik.

Az awk minta

A minta felépítése bonyolultabb mint a `sed` által használté. Az alábbi táblázat tartalmazza lehetséges felépítését:

| <i>Minták</i> | <i>Feldolgozás</i> |
|---------------|--|
| BEGIN | A BEGIN és END speciálisak: az utánuk következő utasításokat a sorok feldolgozása előtt (BEGIN) illetve minden sor feldolgozása után (END) |
| END | hajtja végre az <code>awk</code> , egyetlen egyszer. Ezek a program előkészítő utasításai (pl. változók kezdeti értékének megadása) illetve befejező utasításai (pl. végső eredmény nyomtatása). |

Minták**Feldolgozás**

A program futtatható bemenet nélkül is, ha csak a BEGIN mintát használjuk. Ilyenkor végrehajtja utasításait és utána kilép.

/regex/

Reguláris kifejezés: a awk bővített (extended) kifejezéseket használ, **ugyanazokat mint az egrep**. Nincs tehát szükség külön kapcsolóra, hogy ezt bejelöljük.

minta && minta

2 minta és, vagy illetve nem logikai relációjából képzett feltétel. Az && esetében pl. mindkét mintának illeszkednie kell a feldolgozandó sorra, az || esetében csak az egyiknek. A ! -al jelölt minták esetében a blokk csak akkor hajtódik végre, ha a minta nem válassza ki a sort.

minta || minta

! minta

minta1, minta2

tartomány címezése két mintával

minta ? minta1 :
minta2

A C-hez hasonló feltételes teszt. Ha a minta talál, akkor a minta1 teszt lesz döntő, ha nem akkor a minta2.

relációs kifejezés

Az awk változóira alkalmazott C-szerű relációs kifejezés,pl:

\$1 > 2

tehát ha az első mező numerikusan, számmá alakítva nagyobb mint 2, akkor arra a sorra végrehajtódik az utasítássorozat. A kifejezésben a C nyelv relációs kifejezéseit lehet használni (>, <, <=, >=, ==, !=).

A C nyelvhez képest van két operátor ami különbözik azoktól: a ~ és a ~! (match és no match). Használatuk így történik:

\$1 ~ /^a/

jelentése: ha az első mező kis a betűvel kezdődik, azaz illeszkedik a megadott reguláris kifejezésre.

A kifejezés akkor választja ki a tevékenységet, ha az értéke 0-tól különbözik amennyiben valós számra lehet konvertálni, vagy nem az üres sztring amennyiben sztringre.

Tehát a kifejezéseket sztringekre is lehet alkalmazni: ilyenkor az awk karakterenként hasonlítja össze a sztringeket. ha két sztring megegyező karakterei ugyanazok, akkor a hosszabbik tekintendő nagyobbnak. Pl. "10" kisebb mint "9", mert első karaktereik összehasonlításakor a 1 < 9. De 10 nagyobb mint 9: az awk az adattípustól függően végzi el az összehasonlítást.

Az utasítások

Az awk nyelv elemei { } zárójelek közé kerülnek és szabályokat határoznak meg. A C nyelvhez hasonlóan, változó értékadást, programszervező utasításokat, kifejezéseket tartalmaznak. Az operátorok, utasítások a C-hez hasonlóak (a C nyelvet vették mintának a tervezésüknél).

Pl.

```
{a=2; print a}
```

Az utasítások közt az elválasztó az újsor karakter lehet, amennyiben egy sorba több kerül akkor a ; .

Az awk program 3 részt tartalmazhat, egy bevezető (BEGIN), egy szövegsoronként végrehajtható és egy záró (END) programrészt, tehát általánosan így néz ki:

```
BEGIN {bevezető utasítások}
      minta { a kiválasztott sorra végrehajtott utasítások}
      END   {záró utasítások }
```

A minta {utasítások} szerkezet egymás után többször is előfordulhat, és ez a rész ismétlődik minden egyes feldolgozott sorra.

pl.:

```
$ echo '1 2'| awk 'BEGIN{a=1} {a=a+$1+$2} END{print a}'
4
```

Akárcsak a sed esetében a # jel magyarázat beillesztését jelenti a sor végéig.

A változók

Az awk változóneveket ugyanúgy jelöljük mint a C nyelv változóit. A deklaráció pillanatában jönnek létre és típusukat nem kell explicit deklarálni (tehát nem szükséges őket még csak a BEGIN részben sem deklarálni), és a környezettől függően lesznek valósak vagy karakterláncok. A szkript nyelvekre jellemzően ezek dinamikus változók. A nyelv két típussal rendelkezik: valós illetve sztring. Pl.

```
s="alma"
```

egy sztringet hoz létre,

```
x=2.2
```

pedig egy valós változót.

```
y=$10
```

Az `y` a `$10` mezőtől függően lesz valós vagy karakterlánc.

Belső változók

A program tartalmaz belső változókat, ezek mindig létrejönnek, nem kell őket deklarálni. Közülük az alábbi fontosabbakat fogjuk használni:

| <i>Változó</i> | <i>Angol neve</i> | <i>Mit tartalmaz</i> | <i>Implicit érték</i> |
|----------------|------------------------------------|--|--|
| ARGC | <i>Command line argument count</i> | Hány argumentum van az awk parancssorán | nincs |
| FILENAME | <i>Filename</i> | A bemeneti állomány neve | |
| FS | <i>Input Field Separator</i> | A bementi mezőelválasztó karakter vagy karakterek. Az FS reguláris kifejezés is lehet, tulajdonképpen az implicit érték is az, pontosan: <code>[\t]+</code> , tehát szóköz vagy TAB legalább egyszer. Ha az FS értéke az üres sztring (FS="") akkor a bemenet minden egyes karaktere | szóköz és tabulátor, azaz: <code>[\t]+</code> |

| <i>Változó</i> | <i>Angol neve</i> | <i>Mit tartalmaz</i> | <i>Implicit érték</i> |
|----------------|---|--|-----------------------|
| | | külön mezőnek számít. | |
| RS | <i>Record separator</i> | Ez a rekord (azaz bemeneti sor) elválasztó. Ez is reguláris kifejezés, megváltoztatható. | \n egy újsor karakter |
| IGNORECASE | <i>Ignore case</i> | Ha nem zéró, akkor a reguláris kifejezéseknél kis nagybetű nem számít. | 0 |
| NF | <i>Fields in the current input record</i> | Hány mező van a bemeneti rekordban | nincs |
| NR | <i>Number of records seen so far</i> | Hányadik sornál tart a feldolgozás | nincs |
| OFMT | <i>The output format for numbers</i> | Hogyan formátálja a valós számokat ha a kimenetre írjuk azokat. Látható az implicit értékből, hogy a formátumokat a C nyelv printf függvényének formátáló sztringjeivel állítja elő. | "%.6g" |
| OFS | <i>output field separator</i> | A kimeneti mezőelválasztó karakter. | szóköz |
| ORS | <i>output record separator</i> | A kimeneti rekord (azaz sor) elválasztó karakter. Implicit újsor, akár más is lehet, amennyiben pedig az üres sztring, akkor a kimeneten nem lesznek szétválasztva a sorok. | újsor |

Az ARGV illetve ENVIRON változók a parancssor argumentumait illetve a környezeti változókat tartalmazzák és ezek awk tömbök (lásd alább).

Konstansok

Sztring konstansok

A sztring konstansok kettős idézőjellel körülvevett sztringek. A sztringeken belül használhatóak a C nyelv escape szekvenciával megadott speciális karakterei, a:

```

\\    backslash
\b    backspace
\f    form-feed
\n    newline - újsor
\r    carriage return - kocsi vissza
\t    horizontal tab - tabulátor
\v    vertical tab - függőleges tabulátor
\xhex számjegyek, pl: \xod
\c    literálisan a c karaktert jelenti, pl. \? a ? -t

```

Példa sztring definíciókra:

```

$ echo ' ' | awk '\{s="abc"; t="def\nitt ujsor jon"}\
END{print s,t}'

```

```
abc
def
itt ujsor jon
```

A tízes számrendszerben, nyolcasban és tizenhatosban megadott konstansokat ugyanúgy kezeli mint a C nyelv. Pl:

```
echo '' | awk '{print 10.2,011,0x22}'
10.2 9 34
```

Operátorok

Az awk a C nyelv operátorait használja, ezek változóira és konstansaira alkalmazhatók. Az operátorok:

| | |
|---------|--|
| () | zárójel, csoportosít |
| \$ | mező referencia, pl.: \$1 |
| ++ -- | prefix és postfix inkrementálás, pl.: i++ |
| ^ | hatvány (** is használható) |
| + - ! | + - egyo perandussal, és tagadás |
| * / % | szorzás, osztás, moduló |
| + - | összeadás, kivonás |
| < > | relációsak |
| <= >= | |
| != == | |
| ~ !~ | Reguláris kifejezés tesztelése (illesztés, match) annak tagadása. Ha a reguláris kifejezés konstans, azt a művelet jobb oldalán kell használni. pl.: \$1 ~ /abc/ |
| && | logikai ÉS |
| | logikai VAGY |
| ?: | feltételes teszt |
| = += -= | hozzárendelés és művelet |
| *= /= | |
| %= ^= | |

Az operátorok precedenciája és asszociativitása ugyanaz mint a C-nél.

Különbségek a C nyelvhez képest:

-a sztringek összefűzésének operátora a szóköz karakter, tehát:

```
{print "a" "b"}
```

ab -t fog kiírni

-a nyelv nem rendelkezik a vessző operátorral, tehát nem használhatunk ilyen szerkezeteket mint:

a=2, b=5

Pl:

```
#inkrementálás
$ echo '' | awk '{i=0; i++; print i}'
1
```

```
#reguláris kifejezés illesztés
$ echo '112' | awk '\
{if ($1 ~ /[0-9]+)/\
{print "az elso mezo szamokbol all"}}'
az elso mezo szamokbol all

$ echo 'abc' | awk '{if ($1 !~ /[0-9]+)/\
{print "az elso mezo nem all szamokbol"}}'
az elso mezo nem all szamokbol

#pelda sztring összefűzésre
$ echo '1 2 3 abc def ' | awk '{print $4 $5}'
abcdef
```

Programszervező utasítások

Ezeket a C nyelvből kölcsönzi az awk, az alábbiak használhatóak:

```
if (feltétel) utasítás [ else utasítás ]

while (feltétel) utasítás

do utasítás while (feltétel)

for (expr1; expr2; expr3) utasítás

break

continue

exit [ kifejezés ]

{ utasítás }
```

Pl. az if használata:

```
$ echo '2.2' | awk '{ if ($1==1) {print "igaz"} else\
{print "hamis"}}'
hamis
```

A modern awk nyelv vezérlési utasításnak tekinti a `next` utasítást (függvényt) is (lásd be/ki függvények).

A `next` végrehajtásakor az awk abbahagyja a kurrens rekord feldolgozását, újat vesz a bemenetről és újraindítja vele az awk programot. Pl. ha arra számítunk, hogy a bemenet minden sorában 4 mező van, és nem szeretnénk hibás feldolgozást végezni, akkor olyan sorok érkezésekor amelyekben nincs 4 mező könnyen kikerülhetjük ezeket:

```
#!/usr/bin/awk -f
{
    if (NF != 3) {
        print "Rossz mezőszám, sor: " NR " : " $0
        next
    }
}
```

```

    }
}
#... feldolgozások itt kezdődnek

```

Függvények

A függvényeknek 2 kategóriája van, a beépített és a felhasználó saját függvényei. A függvények fontosabb csoportjai a numerikus, a sztring és a be/ki függvények.

A fontosabb numerikus függvények

Mindazok a függvények amelyek egy kis tudományos kézi számológépben megtalálhatóak, megtalálhatóak az awk-ban is. A függvényeket a C nyelvhez hasonló szintaxissal lehet meghívni.

Ezek ismertek a C nyelv standard könyvtárából, ugyanúgy kell meghívni őket.

```

atan2(y, x)           # atan x/y -t számol
cos(expr)
exp(expr)
int(expr)             #egész részt ad vissza
log(expr)
sin(expr)
sqrt(expr)
rand() , srand ( [ szám ] ) #random szám generálása

```

Pl.

```

$ echo '' | awk '{print sin(1)}'
0.841471

```

Fontosabb sztring függvények

| | |
|--|---|
| <code>index(hol, mit)</code> | Visszatéríti a <code>mit</code> sztring pozícióját az <code>hol</code> sztringben, 0-t ha nincs benne. A sztring indexeket 1-től számolja a nyelv. |
| <code>length (s)</code> | A sztring hosszát téríti vissza. |
| <code>substr(s, i [, n])</code> | Szubsztringet térít vissza az <code>i</code> -edik pozíciótól kezdve, legtöbb <code>n</code> karaktert. |
| <code>tolower(str)</code> | Kisbetűssé konvertál. |
| <code>toupper(str)</code> | Nagybetűssé konvertál. |
| <code>sprintf (format, kifejezés-lista)</code> | A C <code>sprintf</code> -jének megfelelő formátáló függvény. |
| <code>sub(regex, mivel, [miben])</code> | Az <code>miben</code> sztringben behelyettesíti a <code>regex</code> -re illeszkedő részt a <code>mivel</code> sztringgel. Ha a <code>miben</code> paraméter hiányzik, akkor a <code>\$0</code> -ban helyettesít. |
| <code>gsub (regex, mivel, [miben])</code> | A <code>sub</code> egyszer helyettesít, a <code>gsub</code> minden találatot átír. Ha a <code>miben</code> paramétert nem adom meg, akkor a <code>\$0</code> -n dolgozik. Sikertelen átírás után 0-val térnek vissza, sikeres után a <code>sub</code> 1-el, a <code>gsub</code> a cserék számával. Itt is használható az <code>&</code> karakter a találatra való visszautalásként, akár a |

sed s parancsában.

gensub (regex,
mivel,hogyan,
[miben])

Ez az általános helyettesítő függvény. Ugyanúgy működik mint a sub, de a hogyan mezőben meg lehet adni ugyanazokat a kapcsolókat mint a sed helyettesítésénél (n: hányadik találatot illetve g). Ugyanakkor a mivel sztringen használhatóak a \1, \2 stb. visszautalások. Ez a függvény a gawk kiterjesztése.

match (s, regexp)

Illeszti a regexp-et a sztringre: visszatérít egy számot, ami 0 ha nincs illesztés illetve egy pozitív egész szám ha van: ez pontosan az illesztés indexe a sztringben.

Pl.:

```
$ echo 'első masodik' | awk '\
{s=substr($2,5,3); s1=sprintf("%s%s", $1,s);\
s2=substr($2,0,4); print s1,s2}'
elsődik maso
```

Fontosabb ki és bemeneti függvények

next

Abbahagyja a kurrens rekord feldolgozását, veszi a következőt. A feldolgozás újraindul az első awk szabálytól a BEGIN után.

Vigyázni kell használatánál, például az alábbi szkript semmit sem nyomtat a bemenetéről:

```
'{next;print $0}'
```

print

kiírja a kurrens rekordot (a \$0 értékét)

print kifejezés-lista

kiírja egymás mellé a kifejezések értékét

print kifejezés-lista >
file

a kiírást állományba írja. A > jel ekvivalens a shell által használt jellel, akár a >> is használható

printf (fmt, kifejezés-
lista)

A C nyelv printf függvényének megfelelő függvény. Részletes leírását lásd a kézikönyvben.

Pl.:

```
$ echo '12.2261564 94.56256521' | awk '{\
printf("%2.2f %2.4f", $1,$2)}'
12.23 94.5626
```

A print függvény nem csak a kimenetre, hanem csővezetékbe, állományba is tud írni:

```
$ echo '' | awk '{print "abc" > "file.txt"}'
$ echo '' | awk '{print "abc" | "grep -o 'abc'";\
close("grep -o 'abc')}'
```

Az awk kimenete átírányítható, akár a shell-é:

```
{print "a" > "szoveg.txt"}
```

Hasonlóan használható a >> jel is.

Egysoros példaprogramok

A `book.csv` állomány könyvei összárának kiszámítása az `awk`-val:

```
$ cat book.csv | awk -F '\t' 'BEGIN{sum=0}{sum=sum+$5}END{print sum}'
21561.6
```

Állományok összméretének kiszámítása:

```
$ ls -l
-rw-rw-r-- 1 lszabo lszabo    3 Nov 13 23:59 book6.txt
-rw-rw-r-- 1 lszabo lszabo 1794 Nov 14 01:26 hc.txt
-rw-rw-r-- 1 lszabo lszabo  920 Nov 14 01:26 numb1.txt

$ ls -l *.txt | awk '{ x += $5 } \
END {print "osszes byte-ok szama: " x }'
osszes byte-ok szama: 2717
```

Egy állomány sorainak számát így kapom meg:

```
$ cat szoveg.txt | awk 'END { print NR }'
22
```

Az üres sorok kiszűrése egy állományból:

```
$ awk 'NF > 0' szoveg.txt
```

Több állomány egyidejű feldolgozása és a `getline` függvény

A `getline` függvény használata nem ajánlott kezdőknek. Ennek ellenére röviden foglalkozunk vele, mert a fontos lehetőségeit mutatja meg az `awk`-nak. Részletek a kézikönyvben.

| | |
|--|--|
| <code>getline</code> | új sort olvas a bemeneti állományból a <code>\$0</code> -ba beállítja az <code>NF</code> , <code>NRF</code> és <code>NR</code> változókat. A visszatérített érték: 1 - ha sikerült olvasni. 0 - állomány vége -1 - hiba |
| <code>getline változó</code> | új sort olvas a bemenetről a változó változóba, beállítja: <code>NF</code> , <code>NR</code> , <code>FNR</code> változókat. Annak ellenére, hogy a <code>getline</code> függvény, úgy kell meghívni mint egy parancsot (a <code>getline (line)</code> hívás nem működik). |
| <code>getline < file</code> | új sort olvas a file állományból a <code>\$0</code> -ba beállítja az <code>NF</code> változót. Ha sikerült olvasnia, 1 -et térít vissza. |
| <code>getline változó < file</code> | új sort olvas a file állományból a változó változóba. A <code>getline sor < "/dev/tty"</code> mindig a terminálról olvas |
| <code>close (file)</code> | Olvasás után a bemeneti állományt le kell zárni. |
| <code>flush (file)</code> | Kíírja a kimeneti állomány pufferét, ha azt <code>></code> vagy <code>>></code> írással nyitottuk meg. |

Az alábbi program beolvassa a `words` állomány első 3 sorát:

```
BEGIN {
  sorok=0;
  while ( sorok < 4 ) {
    getline < "words"
    print $0
    sorok++
  }
  close("words")
}
```

Külső parancsok futtatása

| | |
|------------------------------|--|
| parancs getline | végrehajt egy UNIX parancsot és annak első sorát olvass |
| parancs getline változó | ugyanaz mint az előbbi, csak egy változóba olvas |
| system(parancs) | lefuttat egy shell parancssort: ennek be és kimenetével nem kommunikál direkt az awk |

Pl. az alábbi parancs lefuttatja az ls parancsot és annak első sorát olvassa.

```
$ awk 'BEGIN{"ls" | getline; print $0}'
array1.awk
```

Saját függvények

A függvények, mint más nyelvben, számításokat csoportosítanak, és név szerinti meghívásukat teszik lehetővé. Szintaktikailag rendelkeznek néhány furcsasággal, amire oda kell figyelni.

Saját függvényeket az alábbi szintaxissal hozhatunk létre:

```
function név (arg1, arg2, . . .)
{
  utasítások;
}
```

A létrehozás helye az awk szabályokon kívül kell történjen a programban (programszervezési szempontból mindegy, hogy hol vannak a szabályokon kívül: nem szükséges őket használatuk előtt definiálni. Ajánlott a program végére tenni őket.

Fontos: saját függvény hívásakor ne írjunk szóközt a függvény neve és a zárójel közé, a:
 függvénynev (
 szerkezetben a szóközt a kontextustól függően a nyelv értelkelheti úgy is, mint egy sztring összefűzés operátort, ezért jó megszokni, hogy a név és zárójel közé ne tegyünk szóközt.

Az argumentumok lokális változók lesznek a függvény testében. Ezzel ellentétben, ha a függvény

belsejében változókat hozunk létre, azok globálisan viselkednek, tehát alkalmasak a főprogrammal való kommunikációra.

Azért, hogy ezt (a függvényben létrehozott változók láthatósága a főprogramból) elkerüljük, a függvényben használt belső változókat is felsoroljuk az argumentumlistában, de értéküket nem adjuk meg híváskor, így üres változókká inicializálódnak. Pl.:

```
function hatvany(i ,j, hat) {
    hat=i**j
    return hat
}
```

Híváskor így hívjuk meg:

```
x=hatvany(2,3)
```

és akkor a `hat` nem lesz látható a főprogramból.

Az `awk` programból az alábbiak szerint használjuk a függvényeket:

```
név(kifejezés1, kifejezés2, ... )
```

vagy

```
változó=név(kifejezés1, kifejezés2, ... )
```

amennyiben a függvény `return kifejezés ;` utasítással lép ki, és egy értéket rendel hozzá.

A függvényeket lehet rekurzív módon használni.

Az alábbi program számok négyzetét listázza. Itt, bár nem használtuk fel, a `negy` változó is létrejön és látható lesz a főprogramban mivel nevét nem adtuk meg a függvény argumentumai között.

```
#!/usr/bin/awk -f
BEGIN {
    for (i=0; i< 10; i++) {
        printf "%4d\t%4d\n", i,negyzet(i)
    }
    print "\na főprogramból látható negy:" negy
}

function negyzet(i) {
    negy=i*i
    return negy
}
```

A program futtatásakor ezt látjuk:

```
$echo '' | awk -f negy.awk
0      0
1      1
2      4
3      9
4     16
5     25
6     36
7     49
8     64
9     81
```

```
a főprogramból látható negy:81
```

Tömbök az awk-ban

Az awk-ban használt tömbök asszociatív tömbök, tehát egy kulcshoz egy értéket rendelnek. Mivel az awk által használt "alap" adattípus a sztring, ezért a kulcsok mindig sztringek.

```
{
  a[1]="első"
  a["masodik"]="masodik"
  a[2]=122.3
}
END {
  print a[1]; print a["masodik"] ; print a[2]
}

első
masodik
122.3
```

A kulcsoknak:

- nem kell szekvenciálisan egymás után jönniük. Így könnyen elő lehet állítani úgynevezett "szórt" tömböket (*sparse array*).
- tulajdonképpen akkor is sztringek, ha számokat használunk kulcsként
- az értékek awk változók vagy konstansok

A tömbök értékeinek elérését megtehetjük ha ismerjük a kulcsokat illetve ha egy speciális `for` ciklust használunk. A kulcsok ismeretében ez egyszerű mint a fenti példa `print` utasításai esetében, vagy:

```
s=a["masodik"]
```

Amennyiben nemlétező kulccsal hivatkozunk egy tömb elemeire, az awk nem ad hibát, a visszaadott érték viszont az üres sztring lesz.

A tömbökre alkalmazható `for` ciklus szintaxisa pedig az alábbi:

```
for (kulcs_valtozo in tömb_valtozo) utasítás
```

A fenti tömböt tehát így listázzuk:

```
{
  a[1]="első"
  a["masodik"]="masodik"
  a[2]=122.3
}
END {
  for ( ix in a ) {
    print "kulcs: ", ix, "érték: ", a[ix];
  }
}
```

Kimenet:

```
kulcs: masodik érték: masodik
kulcs: 1 érték: elso
kulcs: 2 érték: 122.3
```

Látható, hogy a kilistázott elemek nem olyan sorrendben jelennek meg, ahogy deklaráltuk őket, hanem ahogyan az `awk` tárolja az elemeket. Annyit tudunk, hogy a `for` ciklus végigjár minden elemet, de hogy milyen sorrendben azt nem.

Elemeket a tömbből a `delete` függvénnyel lehet törölni:

```
delete tömb_változó[kulcs]
```

Tudnunk kell, hogy a `for` ciklus esetében az iteráció a kulcsokon lépked végig, tehát az értékek listázásához mindig használnunk kell a kulcsokat.

Ha valaki mégis egy bizonyos sorrendben akarja végigjárni a tömböt, akkor saját magának kell azt megszerveznie. Ezt úgy lehet a legkönnyebben elérni, ha numerikus kulcsokat használ. Ilyenkor a tömb létrehozásánál elő kell állítani az egymás utáni kulcsokat, majd ugyanúgy hivatkozni rájuk (lásd alább a példát).

Az elemek kapcsán a döntéshelyzeteket egy speciális `if` utasítással lehet megoldani, amelynek szintaxisa az alábbi:

```
if ( kulcs_változó in tömb_változó )
    utasítás
```

```
{
  a[1]="elso"
  a[3]="masodik"

  if ( 1 in a )
    print "van a[1] elem a tombben"
  if ( "1" in a )
    print "van a[2] elem a tombben"
  if ( 3 in a )
    print "van a[3] elem a tombben"
  if ( 2 in a )
    print "van a[2] elem a tombben"
  else
    print "nincs a[2] elem a tombben"
}
```

Kimenet:

```
van a[1] elem a tombben
van a[2] elem a tombben
van a[3] elem a tombben
nincs a[2] elem a tombben
```

Térjünk vissza az elemek sorban való végigjárhatóságához. Az alábbi példában tudjuk, hogy a

tömbnek csak numerikus kulcsai vannak, megnézzük először, hogy melyik a maximális és minimális kulcs, utána ezek közt iterálunk, és csak azokkal a kulcsokkal dolgozunk, amelyek ténylegesen léteznek a tömbben.

Figyeljük meg a legutolsó if szerkezetet, amely ténylegesen csak azokat a tömbelemeket választja ki, amelyek léteznek.

```
{
  #letrahozom az elemet, figyeljük meg, hogy a tömb hezagos
  a[3]="első"
  a[2]="második"
  a[7]=55

  #megpróbálom kideríteni a maximális és minimális indexet
  # a +0 szerkezetet azért kell használni az indexnél,
  #hogy rakenszerítsuk az awk-t, h. számként kezelje
  # az indexet
  min=0; max=0;
  # ez itt "tomb" for
  for (x in a) {
    if ( x+0 < min )
      min=x
    if ( x+0 > max )
      max=x
  }

  #ezek után tudjuk a határokat, és normal for ciklussal
  #iterálhatunk
  for ( i =min; i<=max; i++) { # ez itt klasszikus for
    if ( i in a ) {
      print "a tömb kulcsa=" i " az értéke pedig=" a[i]
    }
  }
}
```

Kimenet:

```
a tömb kulcsa=2 az értéke pedig=második
a tömb kulcsa=3 az értéke pedig=első
a tömb kulcsa=7 az értéke pedig=55
```

Tömbök létrehozása a `split` függvénnyel

A `split` függvényt egy sztring feldarabolására használjuk. Az eredményt egy tömbben kapjuk meg.

```
szám = split ( sztring , tömb_változó , regex )
```

A következőt végzi: a `regex` reguláris kifejezést használva mező határnak, feldarabolja a sztringet, és a darabokat tömbértékként hozzárendeli `tömb_változó` tömbhöz. A tömb kulcsai egész számok lesznek, 1-től kezdődően. A `szám` visszatérített érték az elemek létrejött elemek számát adja.

```
$ echo 'abc xyz pqgr' | awk '{ n=split($0, a, / /); \
> print "a kapott elemek száma:" n " , az első elem:" a[1]}'
a kapott elemek száma:3, az első elem:abc
```

vagy, ha a szavak végén pontuációs karakterek vannak:

```
$ echo 'abc, xyz. pqgr!' | awk '{ n=split($0, a, /[[:~punct:]]/); \
print "a kapott elemek szama:" n " , az elemek: " a[1] " " a[2] "
" a[3]}'
a kapott elemek szama:3, az elemek: abc xyz pqgr!
```

Bibliográfia

1. Büki András: UNIX/Linux héjprogramozás, Kiskapu, 2002, Az awk c. fejezet
2. A GNU Awk felhasználói kézikönyve, <http://www.ms.sapientia.ro/~lszabo/oprendszer1/gawk/>
letölteni az alábbi címről lehet: <http://hexahedron.hu/personal/peteri/gawk/index.html>
3. Awk, HUP Wiki, <http://wiki.hup.hu/index.php/Awk>