8 Awk

Awk (nimi tulee tekijöistä: *Aho-Weinberger-Kernighan*) on erityisesti tekstimuotoisten, erotinmerkeillä järjestettyjen tiedostojen muokkaamiseen ja raporttien tekoon suunniteltu pieni ohjelmointikieli. Kuten *sed*, myös *awk* lukee tietoa oletuksena rivi kerrallaan, valitsee siitä rivit, jotka täsmäävät annettuun merkkijonomalliin ja suorittaa riveille halutut toimenpiteet.

Yksinkertaisissa editointitoiminnoissa *sed* on yleensä nopeampi ja usein helpompi, mutta *awk* osaa enemmän: se osaa laskea (liukuluvuillakin), siinä on monipuolisemmat ohjausrakenteet (olennaisesti kuten C:ssä), ja se osaa automaattisesti jakaa tietueita kenttiin.

Tässä kuvataan *awk* POSIXin määrittelemässä muodossa.

8.1 Awkin käynnistys

*Awk*in käynnistys komentoriviltä tapahtuu seuraavasti:

`awk [-v` *var=value*`...] [-F`*fs*`]` *program* `[`*arguments*`]`

tai

`awk -f` *progfile* `...[-v` *var=value*`...] [-F`*fs*`] [`*arguments*`]`

Optiot:

`-f` *tiedosto* lukee *awk*-ohjelman tiedostosta.

`-v` *var=value* alustaa muuttujan *var* arvon ennen ohjelman käynnistymistä (ennen `BEGIN`-osaa). Vrt. *var=value* argumenttina.

`-F`*fs* asettaa kenttäerottimen `FS` arvon.

Jos yhtään `-f` -optiota ei ole annettu, ensimmäinen ei-optio-argumentti tulkitaan *awk*-ohjelmaksi.

Muut argumentit voivat olla tiedostonimiä tai muuttujan alustuslauseita (muotoa *var=value*; senmuotoista tiedostonimeä ei voi käyttää!). Näin annetut asetukset tehdään mahdollisen `BEGIN`-osan jälkeen mutta ennen seuraavan tiedoston käsittelyä.

Esim. `awk -f prog x=1 file1 x=2 file2 x=3`
suorittaisi ensin `BEGIN`-osan, alustaisi sitten muuttujan `x` ykköseksi, käsittelisi tiedoston `file1`, asettaisi `x`:n kakkoseksi, käsittelisi tiedoston `file2`, asettaisi `x`:n kolmoseksi ja suorittaisi sitten `END`-osan.

*Awk*in exit status on 0 jos se sai kaikki tiedostot käsitellyksi, jotain muuta jos tapahtui virhe. Sen voi myös erikseen asettaa *awk*in omalla `exit`-lauseella (ks. 8.8.7).

8.2 Awk-ohjelman rakenne

Awk-ohjelma koostuu yhdestä tai useammasta ehto/toiminto (*pattern*/*action*) -parista (sekä mahdollisista funktiomäärittelyistä, ks. 8.11):

```
pattern {action}
pattern {action}
...
```

Seassa voi olla myös tyhjiä rivejä (jotka eivät vaikuta mitään), sekä `#`-alkuisia kommentteja (omina riveinään tai rivin lopussa). Näin ollen *awk*-ohjelmasta voi tehdä itsenäisen laittamalla tiedoston alkuun rivi `#! /bin/awk -f` (tai mikä *awk*in polku onkin). Kommentti toimii kuten pelkkä rivinvaihto.

Rivin lopussa oleva kenoviiva \ toimii jatkorivin merkkinä, ei kuitenkaan kommentin lopussa eikä merkkijonon sisällä.

Ohjelma suoritetaan siten, että *awk* lukee yhden tietueen (oletuksena rivi), käy järjestyksessä läpi ohjelman ja suorittaa kaikki ne toiminnot (*action*) joiden ehto (*pattern*) toteutuu. Sitten luetaan seuraava tietue ja aloitetaan ohjelman suoritus taas alusta, ja niin edelleen.

Ehto-osa voi olla merkkijonomalli (*Extended Regular Expression*) kauttaviivoilla rajattuna (ks. 6.4, 8.3) johon tietueen sisältöä verrataan, tai yleensä ehtolauseke. Jos malli tai lauseke täsmää, suoritetaan toiminto-osan sisältämät komennot.

Kaksi ehtoa voi yhdistää pilkulla, mikä tarkoittaa väliä ensimmäisen ehdon toteuttavasta tietueesta lähinnä seuraavaan toisen ehdon toteuttavaan (ne mukaanlukien).

Ehdon merkityksen voi kääntää päinvastaiseksi sen edessä olevalla huutomerkillä. (Välin tapauksessa vaikuttaa kumpaankin päähän erikseen, ei koko väliin.)

Lisäksi erikoistapauksina ehto `BEGIN` tarkoittaa ”ennen ensimmäisen syöttörivin lukemista” ja `END` ”viimeisen rivin jälkeen”. Ne suoritetaan vain kerran vaikka syöttötiedostoja olisi useita.

Jos ehto-osa jätetään pois suoritetaan toiminto-osan komennot kaikille tietueille.

Toiminto-osassa voi olla mielivaltainen määrä jäljempänä esiteltäviä komentoja rivinvaihdoilla tai puolipisteillä eroteltuina. Jos toiminto-osa puuttuu, oletustoiminto on `{print}` (tulostaa käsiteltävän olevan tietueen).

Useita ehto-toiminto -pareja voi olla samalla rivillä. Tällöin toiminto-osa ei saa puuttua kuin viimeisestä.

Esim. `awk '!/Pekka/ {print}' file`
tekisi saman kuin `grep -v 'Pekka'`, samoin yksinkertaisesti `awk '!/Pekka/' file`.

Esim. Tulostettava tiedostosta rivit 5–10: (`NR` on tietuenumero):

```
        awk 'NR==5, NR==10' file
tai     awk 'NR>=5 && NR<=10' file
```

Ehto-osassa voi testata muutakin kuin syöttötietuetta. Esim. kumpikin seuraavista tulostaisi jokaisen rivin jolla esiintyy `"Kalle"` ja viisi seuraavaa riviä:

```
awk '/Kalle/{x=NR+5}  NR<=x'
awk '/Kalle/,NR==x { if (!x) x=NR+5; print }'
```

`BEGIN`- ja `END`-ehtoja ei voi yhdistellä muiden kanssa. Ne sijoitetaan yleensä alkuun ja loppuun, mutta toiminnan kannalta niiden sijainti on yhdentekevä.

8.3 Merkkijonovakiot ja -mallit

Merkkijonovakiot rajataan lainausmerkeillä `"`, ja niiden sisällä kenoviivalla `\` voidaan esittää erikoismerkkejä seuraavasti (vrt. 5.64 `tr`):

merkkiyhdistelmä	merkitys
`\"`	lainausmerkki (`"`)
`\/`	kauttaviiva (`/`)
`\`*ddd*	merkki jonka oktaalikoodi on *ddd* (1–3 numeroa, ei pelkkiä nollia)
`\\`	kenoviiva (`\`)
`\a`	<*alert*> (control-G)
`\b`	<*backspace*> (control-H)
`\f`	<*formfeed*> (control-L)
`\n`	<*newline*> (control-J)
`\r`	<*carriage return*> (control-M)
`\t`	<*tab*> (control-I)
`\v`	<*vertical tab*> (control-K)

Merkkijonon sisällä ei saa olla rivinvaihtoa (muuten kuin `\n`:llä esitettynä).

Merkkijonomallit (*regexp*) ovat *Extended Regular Expression*-tyyppisiä (ks. 6.4) ja ne esitetään kuten merkkijonovakiot (ylläolevat erikoismerkkinotaatiot toimivat), paitsi että ne rajataan kauttaviivoilla.

Useimmissa yhteyksissä ERE:nä voi käyttää myös merkkijonoarvoista lauseketta, ja silloin lainausmerkitkin kelpaavat. Huom. kenoviiva on erikoismerkki sekä merkkijonovakiossa että ERE:ssä, ja jos merkkijonovakiota käytetään ERE:nä, kenoviivat tulkitaan kahdesti.

Esim. tulostetaan rivit joiden toisessa kentässä esiintyy kenoviiva:

```
awk '$2 ~ /\\/'
```
tai
```
awk '{ if ($2 ~ "\\\\") print }'
```

8.4 Tietueet ja kentät

Awk jakaa syöttötietueen (*record*) automaattisesti kenttiin (*fields*). Kenttiin voidaan viitata kenttämuuttujilla tyyliin `$1`, `$2`..., ja koko tietueeseen `$0`:lla. `$`:n perässä voi olla myös lauseke, jolloin sen arvoa käytetään kenttäindeksinä (esim. `n=20; print $n` tulostaisi saman kuin `print $20`). Kenttämuuttujiin voi myös sijoittaa, myös sellaiseen jota vastaavaa kenttää ei tietueessa ennestään ole (`$14="x"` jne); tällöin myös `$0` muuttuu (paitsi muutettu kenttä, myös kaikki kenttäerottimet vaihtuvat `OFS`-muuttujan mukaisiksi), samoin `NF` (ks. 8.5.1).

Olemattomaan kenttään viittaaminen ei aiheuta virhettä vaan palauttaa tyhjän merkkijonon (joka toimii laskutoimituksissa nollana).

Tietueet erotellaan normaalisti rivinvaihdolla (rivit ovat siis tietueita) ja kentät joko välilyönneillä tai tabulaattoreilla. Kenttäerotin voidaan vaihtaa komentorivioptiolla `-F`*fs*, missä *fs* tarkoittaa uutta kenttäerotinta, tai asettamalla muuttujalle `FS` uusi arvo (yleensä `BEGIN`-osassa), ja tietue-erotin muuttujalla `RS` (sille ei olekaan komentorivioptiota).

Kenttäerotin `FS` on *Extended Regexp*, `RS` sen sijaan vain yksi merkki (tosin mm. Gnu awk käyttää siinäkin regexp'iä). Erikoistapauksena tyhjä merkkijono `RS`:ssä tarkoittaa yhtä tai useampaa peräkkäistä tyhjää riviä.

Esim.
```
awk '{print $1, $3, $2}' tied
```
tulostaa tiedoston `tied` riveiltä kentät yksi, kolme ja kaksi.

```
awk -F: '!$3' /etc/passwd
```
etsii `/etc/passwd`:stä käyttäjät joiden UID on nolla.

```
awk -F'[^[:alpha:]]*' '{ print $3 }'
```
tulostaa jokaisen rivin kolmannen (kirjaimista koostuvan) sanan.

Olkoon tiedostossa sivutettua tekstiä, jokaisen sivun lopussa *formfeed* (control-L). Halutaan tulostaa jokaisen sivun ensimmäinen rivi:
```
awk 'BEGIN{ FS="\n"; RS="\f" } { print $1 }'
```

8.5 Muuttujat

Muuttujat voivat olla joko luku- tai merkkijonoarvoisia. Muuttujan tyyppiä ei määritellä erikseen vaan se määräytyy muuttujan arvon (sisällön) perusteella, ja voi muuttua aina kun muuttujan arvokin muuttuu. Muuttujia ei tarvitse alustaa erikseen, sillä kaikki *awk*issa esiintyvät muuttujat on oletusarvoisesti alustettu tyhjiksi (lukuna käytettäessä tyhjä toimii nollana). Kaikki lukumuuttujat ovat liukulukuja.

Muuttujaa voi käyttää lukuna vaikka sitä olisikin aikaisemmin käytetty merkkijonona (merkkijono jonka alusta ei lukua löydy tulkitaan nollaksi) ja päinvastoin (`CONVFMT`:n määräämässä formaatissa, ks. 8.5.1).

Muuttujien nimet saavat sisältää sarjan kirjaimia, numeroita ja alaviivoja (`_`). Muuttujan nimi ei kuitenkaan saa alkaa numerolla. Isoilla ja pienillä kirjaimilla on oma merkityksensä eli `a` ja `A` ovat eri muuttujia. (Huomaa että normaaleissa muuttujaviittauksissa ei käytetä dollarimerkkiä.)

Esim. tulostetaan rivit joiden ensimmäinen kenttä on sama kuin ensimmäisellä rivillä:

```
awk 'NR==1 { key=$1 } $1 == key'
```

8.5.1 Systeemimuuttujat

Awk tuntee joukon valmiiksi määriteltyjä ns. systeemimuuttujia, joilla voi toisaalta vaikuttaa syötön ja tulostuksen käsittelyyn ja joidenkin käskyjen toimintaan ja toisaalta saada tietoa mm. käsiteltävänä olevasta datasta. Osa systeemimuuttujista on tarkoitettu vain luettaviksi, ja vaikka niiden arvoja voikin muuttaa, se ei yleensä ole järkevää ja tulokset voivat yllättää.

POSIX määrittelee *awk*ille seuraavat systeemimuuttujat:

ARGC `ARGV`-taulukon koko.

ARGV Komentoriviargumentit taulukkona, poislukien optiot ja *awk*-ohjelma.

CONVFMT[2] Formaatti liukulukujen merkkijonomuunnoksiin muuten kuin tulostuskäskyissä, oletus `%.6g`. Ei sovelleta kokonaislukuihin.

ENVIRON[1] Ympäristömuuttujat taulukkona (indeksinä muuttujan nimi).

FILENAME[1] Käsiteltävänä olevan syöttötiedoston nimi. `BEGIN`-osassa arvo määrittelemätön, `END`-osassa viimeksi käsitellyn tiedoston nimi.

FNR[1] Tietuenumero käsiteltävänä olevassa tiedostossa. `BEGIN`-osassa nolla, `END`-osassa viimeksi käsitellyn tietueen numero.

FS[2] Kenttäerotin syötössä (field separator). Oletuksena välilyönti, jolla on erikoismerkitys: alkutyhjät ohitetaan ja erottimina käytetään yhtä tai useampaa peräkkäistä tyhjää (= välilyöntiä tai tabulaattoria). Muuten mikä tahansa yksittäinen merkki tarkoittaa sitä itseään, ja useampimerkkinen tulkitaan EREksi.

NF[1] Kenttien määrä nykyisessä syöttötietueessa. `BEGIN`-osassa määrittelemätön, `END`-osassa siihen viimeksi jäänyt arvo (ellei `getline`ä ole käytetty ilman muuttuja-argumenttia, ks. 8.7.3).

NR[1] Tietuenumero syötteen alusta laskien (kaikissa tiedostoissa yhteensä). `BEGIN`-osassa nolla, `END`-osassa viimeksi käsitellyn tietueen numero.

OFMT[2] Oletusformaatti liukulukujen merkkijonomuunnoksiin tulostuskäskyissä. Oletusarvo on `%.6g`. Ei sovelleta kokonaislukuihin.

OFS[2] Tulostuksessa käytettävä kenttäerotin. Oletusarvo on välilyönti.

ORS[2] Tulostuksessa käytettävä tietue-erotin. Oletusarvo on rivinvaihto.

RLENGTH[1] *match()*-funktion löytämän merkkijonon pituus (ks. 8.10).

RS[2] Syötössä käytettävä tietue-erotin. Oletusarvo on rivinvaihto (newline). Erikoistapauksena tyhjä merkkijono tarkoittaa yhtä tai useampaa peräkkäistä tyhjää riviä. `RS`:ssä saa standardi-*awk*issa olla enintään yksi merkki.

RSTART[1] *match()*-funktion löytämän merkkijono alkukohta (ks. 8.10).

SUBSEP[2] Indeksien erotin moniulotteisissa taulukoissa (ks. 8.5.3).

[1] Ei yleensä kannata muuttaa suoraan.
[2] Yleensä asetetaan vain kerran (`BEGIN`-osassa).

Esim. seuraavat ovat ekvivalentteja:

```
awk -F: '{ print $3 }'
awk 'BEGIN{FS=":"} { print $3 }
```

Esim. lisätään jokaisen .c-tarkenteisen tiedoston alkuun kommenttina tiedoston nimi:

```
for i in *.c ;do
  awk 'NR==1 { print "/* " FILENAME " */" }
       { print }' $i >$i.bak &&
  mv $i.bak $i
done
```

8.5.2 Taulukot

*Awk*in taulukot eroavat useimmista perinteisistä ohjelmointikielistä siinä, että indekseinä ei käytetä kokonaislukuja vaan merkkijonoja (usein puhutaan assosiatiivisista taulukoista).

Taulukkoa ei myöskään tarvitse erikseen luoda, riittää kun sijoittaa taulukon alkiolle arvon. Taulukon alkioihin viitataan syntaksilla *array*`[`*index*`]`.

Esim.
```
taulu["Ma"] = "Maanantai"
print taulu["Ma"]
```

Olemattomaan alkioon viittaaminen palauttaa tyhjän merkkijonon (ja luo taulukon jos sitä ei ole, paitsi `in`-operaattorin kanssa).

Operaattoria *in* voidaan käyttää testattaessa löytyykö tietylle indeksille arvoa taulukosta syntaksilla

index `in` *array*

joka palauttaa ykkösen jos *array*`[`*index*`]` löytyy ja nollan jos ei löydy.

Taulukon luomiseen voidaan käyttää myös merkkijonofunktiota *split()* (ks. 8.10).

Taulukosta voidaan poistaa arvoja komennolla `delete` *array*`[`*index*`]`.

Itse taulukkoa sen sijaan ei voi poistaa mitenkään (Gnu *awk*issa voi: `delete` *array*). Kerran taulukkona käytettyä nimeä ei voi sen jälkeen muulla tavoin käyttää.

Taulukkoa kokonaisuutena ei voi käyttää kuin `in`-operaattorin kanssa sekä funktion argumenttina. Koko taulukon kopioiminen edellyttää siten silmukkaa (ks. 8.8.4).

8.5.3 Moniulotteiset taulukot

Awk ei tunne "oikeita" moniulotteisia taulukoita, mutta jos taulukon indeksinä käytetään useaa pilkulla erotettua lauseketta, se tekee niistä indeksin joka käyttäytyy kuin ne olisivat erillisiä moniulotteisen taulukon indeksejä. Sisäisesti indeksi on ko. lausekkeet eroteltuna muuttujan `SUBSEP` arvolla.

Siis esim. `a[1,2]=5` toimii kuten voisi odottaakin. Sisäisesti se on ekvivalentti muodon `a[1 SUBSEP 2]=5` kanssa.

Moniulotteisen taulukon alkioiden olemassaoloa voi myös testata `in`-operaattorilla, paitsi rakentamalla indeksin yhdeksi em. tavalla myös laittamalla indeksit sulkuihin pilkuilla erotettuna, seuraavat tekevät saman asian:

```
"a" SUBSEP 1 in array
("a",1) in array
```

8.6 Laskutoimitukset, lausekkeet, operaattorit

Awk tuntee seuraavat, enimmäkseen C:stä tutut peruslaskutoimitus- ja sijoitusoperaattorit (ryhmiteltynä prioriteettijärjestyksessä):

operaattori	merkitys	assosiatiivisuus
`(`*expr*`)`	ryhmittely	–
`$`*expr*	kenttäviittaus	–
`++`*var*	*var*`=`*var*`+1`	–
var`++`	kuten yllä mutta palauttaa vanhan arvon	–
`--`*var*	*var*`=`*var*`-1`	–
var`--`	kuten yllä mutta palauttaa vanhan arvon	–
expr `^` *expr*	potenssiinkorotus	oikea
`!` *expr*	looginen EI	–
`+` *expr*	plus etumerkkinä	–
`-` *expr*	miinus etumerkkinä	–
expr `*` *expr*	kertolasku	vasen
expr `/` *expr*	jakolasku	vasen
expr `%` *expr*	modulus (jakojäännös)	vasen
expr `+` *expr*	yhteenlasku	vasen
expr `-` *expr*	vähennyslasku	vasen
string string	merkkijonojen katenointi	vasen
expr == *expr*	yhtäsuuri kuin	ei
expr != *expr*	erisuuri kuin	ei
expr < *expr*	pienempi kuin	ei
expr <= *expr*	pienempi tai yhtäsuuri kuin	ei
expr > *expr*	suurempi kuin	ei
expr >= *expr*	suurempi tai yhtäsuuri kuin	ei
string `~` *ERE*	merkkijono vastaa mallia	ei
string `!~` *ERE*	merkkijono ei vastaa mallia	ei
expr `in` *array*	taulukon jäsenyys	vasen
`(` *index* `)` `in` *array*	moniulotteisen taulukon jäsenyys	vasen
expr `&&` *expr*	looginen JA	vasen
expr `||` *expr*	looginen TAI	vasen
expr `?` *expr* `:` *expr*	ehdollinen lauseke	oikea
var `=` *expr*	sijoitus	oikea
var `+=` *expr*	yhteenlaskusijoitus	oikea
var `-=` *expr*	vähennyslaskusijoitus	oikea
var `*=` *expr*	kertolaskusijoitus	oikea
var `/=` *expr*	jakolaskusijoitus	oikea
var `%=` *expr*	modulussijoitus	oikea
var `^=` *expr*	potenssisijoitus	oikea

C:n bittisiirto-operaattorit `<<` ja `>>` puuttuvat, samoin bitti-JA, -TAI ja -XOR (ja niiden symboleille on eri merkityksiä, vrt. 8.7.4). Toisaalta C:stä poiketen kaikki vertailuoperaattorit toimivat myös merkkijonoille (merkkijonoa ja lukua vertailtaessa luku muunnetaan ensin merkkijonoksi), ja lisänä on taulukon jäsenyysoperaattori `in` sekä kolme uutta merkkijono-operaattoria:

Merkkijonojen katenointi: kaksi merkkijonoa peräkkäin (tarvittaessa tyhjällä erotettuna) yhdistetään (esim. `"ab" "cd"` antaa tulokseksi `"abcd"`).

`~` : RegExp -vertailu, tosi jos vasemmalla puolella oleva merkkijono vastaa oikealla puolella olevaa mallia (kauttaviivoilla rajattu merkkijono tai merkkijonoarvoinen lauseke). Esim. `$5 ~ /[0-9]/` on tosi, jos nykyisellä rivillä viides kenttä sisältää numeron.

`!~` Edellisen negaatio. Esim. `$5 !~ "[0-9]"` on tosi, jos nykyisellä rivillä viides kenttä ei sisällä numeroa.

Merkkijonoja ja lukuja saa vapaasti yhdistellä, ne muunnetaan automaattisesti tarvittaessa. Merkkijono muunnetaan luvuksi kuten C:n `atof()`-funktio (merkkijonon alusta ohitetaan ensin tyhjät, jos sittenkään ei löydy lukua se tulkitaan nollaksi), luku merkkijonoksi muuttujan `CONVFMT` määräämässä formaatissa, paitsi jos se on arvoltaan kokonaisluku: silloin käytetään formaattia `"\%d"`.

Esim. `print "a" 3 * 5}` tulostaisi `a15`.

Merkkijonoa ja lukua vertailtaessa luku muunnetaan merkkijonoksi (`"2" < 12` on epätosi). Tarvittaessa luku voidaan muuttaa merkkijonoksi katenoimalla siihen tyhjä merkkijono ja merkkijono luvuksi lisäämällä siihen nolla (`"2"+0 < 12` on tosi).

8.7 Syöttö ja tulostus

8.7.1 Print

Komento `print` tulostaa argumenttinsa (tai ilman argumentteja `$0`:n) `OFS`-muuttujan arvolla (alussa välilyönti) erotettuina. Tulostuksen muotoiluun vaikuttaa myös muuttuja `OFMT`, joka määrää liukulukujen tulostusformaatin (syntaksi kuten `printf`issä).

Haluttaessa tulostaa arvoja ilman erottimia voi paitsi muuttaa `OFS`:ää, käyttää hyväksi merkkijonojen katenointia, vertaa seuraavia:

```
{ print 1,2 }
{ OFS=""; print 1,2 }
{ print 1 2 }
```

8.7.2 Printf

Monimutkaisempaan tulostuksen muotoiluun on käytettävissä funktio `printf`:

printf formaatti **[**, argumentit **]**

Muotoilussa käytettävät kentät on lainattu suoraan C:stä:
`%[liput][kentän koko][tarkkuus]tyyppi`. Normaalit merkit tulostuvat itsenään.

Alla on taulukoitu *awk*issa käyttökelpoiset tyyppikirjaimet ja liput:

Tyyppikirjain	Merkitys
d,i	Etumerkillinen desimaalikokonaisluku.
u	Etumerkitön desimaalikokonaisluku.
o,O	Etumerkitön oktaalikokonaisluku.
x,X	Etumerkitön heksadesimaalikokonaisluku.
e,E	Liukuluku muodossa `[-]d.ddde[+-]dd`.
f	Liukuluku muodossa `[-]ddd.ddd`
g,G	Liukuluku f-muodossa jos mahtuu, muuten e, paitsi loppunollia desimaaliosasta ei tulosteta.
c	Merkki.
s	Merkkijono.
%	Prosenttimerkki itse.
Lippu	**Merkitys**
0	Kenttä täytetään nollilla eikä tyhjillä.
-	Kenttä tasataan vasemmalle (oletus oikealle).
' '	Positiivisen etumerkin paikalle tulostetaan välilyönti.
+	Positiivisen etumerkin paikalle tulostetaan '+'.

Kentän koko on haluttu merkkipositioiden minimimäärä, desimaaliluku tai `*` jolloin se otetaan argumenttilistasta.

Tarkkuus on kokonaisluvuille haluttu numeromäärä (täydennetään nollilla tarvittaessa), liukuluvuille haluttu desimaalimäärä.

Huomaa että *printf* ei automaattisesti tulosta rivinvaihtoa (toisin kuin *print*), vaan se on aina tarvittaessa lisättävä formaattiin ("*n*", ks. 8.3).

Esim. `{ printf ("%d+%d=%d\n", $1, $2, $1+$2 }`

8.7.3 Getline

Funktiolla `getline` voidaan lukea tietue ohi normaalin suoritusjärjestyksen.

Ilman argumentteja `getline` lukee seuraavan tietueen normaalista syötteestä ja asettaa kenttämuuttujat `$0` jne sekä muuttujat `NF`, `NR` ja `FNR`.

`getline` *var* lukee seuraavan tietueen muuttujaan *var*. Se ei muuta kenttämuuttujia mutta asettaa `FNR`:n ja `NR`:n.

`Getline` palauttaa arvon 1 jos luku onnistui, 0 jos tiedosto loppui ja -1 jos tapahtui virhe.

Esim. tulostettava jokainen rivi jolla esiintyy 'Pekka' ja seuraava rivi:
`awk '/Pekka/ { print; if (getline) print }'`

8.7.4 Uudelleensuuntaus

Sekä `print` että `printf` tulostavat oletuksena stdout'iin, mutta ne voi tarvittaessa suunnata tiedostoon tai ohjelmalle kuten shellissä:

```
print[f] ... > file
print[f] ... >> file
print[f] ... | command
```

Tiedostonimi tai komento voi olla mielivaltainen merkkijono, myös muuttuja (vakiomerkkijonon ympärille tarvitaan lainausmerkit). Muuttujaan tulostusta ei saa (mutta vrt. `sprintf`, ks. 8.10).

Komennon on oltava ulkoinen komento (se suoritetaan kuten *execv()* ilman shelliä).

Vastaavasti `getline`:n syötön voi uudelleensuunnata tiedostosta tai komennolta:

```
getline [var] < file
command | getline [var]
```

Tällöin `getline` ei aseta muuttujia `NR` ja `FNR` (`NF`:n ja `$0`:n kyllä jos muuttuja-argumentti puuttuu).

Sekä syötön että tulostuksen uudelleensuuntauksessa tiedosto jää auki tai komento käyntiin ja lukemista tai kirjoittamista voi jatkaa käyttämällä täsmälleen samanarvoista merkkijonoa tiedosto- tai komentonimenä uudelleensuuntauksessa. Niinpä muotoa `>>` tarvitaan vain jos halutaan kirjoittaa *awk*-ohjelman alkaessa olemassaolevan tiedoston perään (tai jos tiedosto välillä suljetaan).

Esim. tiedostossa `all.data` on rivin alussa käyttäjätunnuksia ja niiden perässä niihin liittyviä tilastotietoja. Kerätään kunkin tunnuksen tiedot tiedostoon nimeltä *tunnus*`.dat`:

```
awk '{ print > $1 ".dat" }' all.data
```

Standardi jättää avoimeksi mitä tapahtuu jos samaa putkea yritetään käyttää sekä syöttöön että tulostukseen, seuraava temppu saattaa toimia tai sitten ei:

```
print "2/3" | "bc -l"
"bc -l" | getline
```

Huom. *Awk*in uudelleensuuntaus on nimenomaan käskyjen `print`, `printf` ja `getline` ominaisuus, mielivaltaista käskyä ei voi uudelleensuunnata eikä uudelleensuuntauksia ketjuttaa (`print "2/3" | "bc" | getline` ei toimi).

8.7.5 Close()

Jos uudelleensuuntauksessa käytetty tiedosto tai komento halutaan sulkea sen voi tehdä näin:

`close(`*string*`)`

missä *string* on sama tiedostonimi tai komento jota käytettiin uudelleensuuntauksessa.

Tämä on tarpeen jos tiedosto tai komento halutaan aloittaa uudelleen alusta, tai jos tiedostokahvojen loppumisen välttämiseksi: yhtaikaa auki olevien tiedostojen ja putkien määrä voi on rajoitettu, joten ne on syytä sulkea kun niitä ei enää tarvita jos niitä käyttää vähänkin enemmän.

Esim. tiedostossa `all.data` on rivin alussa käyttäjätunnuksia ja niiden perässä niihin liittyviä tilastotietoja. Kerätään kunkin tunnuksen tiedot tiedostoon nimeltä *tunnus*`.dat`, ja varaudutaan siihen että tunnuksia on enemmän kuin käytettävissä olevia tiedostokahvoja:

```
rm -f *.dat
awk '{ print >> $1 ".dat" ; close($1 ".dat") }' all.data
```

8.7.6 System()

Funktio *system()* suorittaa parametrinä annetun komentorivin kuten C:n *system()*, eli olennaisesti välittää sen shellille. Arvonaan se palauttaa saamansa exit statuksen, komennon mahdollista tulostusta ei saa takaisin *awk*ille (mutta siihen voi sisältyä uudelleensuuntaus tiedostoon josta tuloksen voi lukea; yleensä kuitenkin `getline` on kätevämpi jos tulostus halutaan talteen). *Awk* odottaa komennon suorituksen päättymisen ennenkuin suoritusta jatketaan (tai oikeammin sen käynnistävän shellin päättymistä, komennon voi jättää taustalle &:lla).

Esim.

```
BEGIN { if (system ("mkdir tmp") != 0 ) {
        print "temporary directory creation failed!" ; exit 1
      }
}
```

8.8 Ohjausrakenteet

*Awk*in ohjausrakenteet on lainattu melko suoraan C:stä, switch tosin puuttuu. Huomaa kuitenkin ehtojen monipuolisuus, `for`-lauseen toinen muoto sekä ehto-osan käyttö.

8.8.1 If

if (ehto**)**
komento1
[else
komento2 **]**

Ehto on mielivaltainen luku- tai merkkijonoarvoinen lauseke. Jos ehto toteutuu (eli on eri kuin nolla tai ei-tyhjä), niin suoritetaan komento1. Mikäli *else* osa on määritelty, komento2 suoritetaan jos ehto on epätosi (nolla tai tyhjä). Mikäli komentoja on useampia on ne sijoitettava aaltosulkuihin { }.

Esim. Tulostetaan x ellei se ole nolla tai tyhjä:

```
if ( x ) print x
if ( x ) print x; else print "nietu"
if ( x ) { print x } else print "nietu"
```

Ehto voi olla myös pelkkä *regexp*, jolloin sitä verrataan `$0`:aan. Esim. tulostetaan riveistä 10-20 ne, joilla esiintyy sana "Jussi":

```
awk 'NR==10, NR==20 { if (/Jussi/) print }'
```

Esim. tulostetaan postikansiosta otsikot, ts. Subject: -rivit, mutta vain kuoriosista:

```
awk '/^From /,/^$/ { if (/^Subject:/) print }'
```

Jos `else` voisi liittyä useampaan eri `if`-sanaan (`if (...) if (...) else ...`), se liittyy viimeiseen. (Käytä aaltosulkuja moisissa rakenteissa!)

*Awk*issa `if`-lausetta tarvitaan vähemmän kuin monissa muissa ohjelmointikielissä, koska ehto-osalla voi suoraan testata samoja asioita. Esim. seuraavat tekevät saman asian:

```
awk '{ if ($2 > 0) print }'
awk '$2 > 0 { print }'
awk '$2 > 0'
```

Vastaavasti else-osan voi usein korvata `next`-komennolla (ks. 8.8.6).

C:stä tuttu *ehto* ? *kyllä-arvo* : *ei-arvo* on myös käytettävissä.

8.8.2 While

while (ehto **)**
komennot

Silmukan aluksi *ehto* tutkitaan ja mikäli se on tosi suoritetaan komennot. Mikäli ehto ei ole alussa tosi ei silmukan runkoa suoriteta lainkaan. Jos komentoja on useampia ne on suljettava aaltosulkuihin { }.

Esim.
```
i=1
while ( i <= 4 ) {
  print $i
  i++
}
```

8.8.3 Do

do
komennot
while (ehto **)**

Do-silmukan ero *while*-silmukkaan on siinä, että tutkittava ehto on silmukan lopussa. Tämä aiheuttaa sen että silmukassa määritellyt komennot suoritetaan ainakin kerran. Kuten *while*-silmukassakin, aaltosulkuja { } on käytettävä, jos suoritettavia komentoja on useampia.

8.8.4 For

For-lauseella on kaksi muotoa. Ensimmäinen on C:stä tuttu:

for (*alustus*; *ehto*; *askel* **)**
komennot

Silmukkaa suoritetaan niin kauan kuin *ehto* on tosi. Minkä tahansa osan saa jättää tyhjäksi (tyhjä ehto on aina tosi).

Esim.
```
for ( i = 0; ++i <= NF; )  print $i
```

Toinen muoto on taulukon läpikäyvä

for (*var* **in** *array***)**
komennot

joka suorittaa silmukan muuttujan *var* käydessä läpi kaikkien taulukon *array* alkioiden indeksit (määräämättömässä järjestyksessä).

Esim. tulostetaan taulukko indekseineen:

```
for (i in t) printf "%s[%s]=%s\n","t",i,t[i]
```

Taulukon kopiointi käy näin:

```
for (i in a) b[i]=a[i]
```

Moniulotteisten taulukoiden läpikäyntiin ei ole vastaavaa syntaksia, mutta koska ne ovat sisäisesti yksiulotteisia, ne voi käydä läpi yhdellä silmukkamuuttujalla. Esim.

```
BEGIN {
  SUBSEP=","
  n=3
  for (i=1; ++i<n; ) {
     for (j=1; ++j<=n; ) a[i,j]=0
  }
  ...
  for (k in a) printf "a[%s]=%s\n", k, a[k]
}
```

8.8.5 Break ja continue

Käsky `break` hyppää ulos sisimmästä `while`-, `do`- tai `for`-silmukasta, ja `continue` hyppää silmukan alkuun kuten C:ssä. Monesta sisäkkäisestä silmukasta ei pääse suoraan ulos.

8.8.6 Next

Käsky `next` lopettaa käsiteltävän olevan tietueen käsittelyn ja hyppää *awk*-ohjelman alkuun (*ei* `BEGIN`-osaan).

Usein `next`-komennolla voi välttää turhia ehtoja ja nopeuttaa ohjelmaa paljonkin. Esim. kerätään tiedoston `#`-alkuiset rivit tiedostoon `comments` ja muut rivit tiedostoon `action`:

```
awk '/^#/{ print > "comments" ; next }
         { print > "action" }'
```

8.8.7 Exit

Käsky `exit` lopettaa syötön käsittelyn ja hyppää mahdolliseen `END`-osaan, paitsi `END`-osan sisällä se lopettaa suorituksen välittömästi.

Argumenttina voidaan antaa exit status 0–255 (`END`-osan sisällä tapahtuva virhe tai toinen `exit` voi sen vielä muuttaa).

8.9 Matemaattiset funktiot

Awk tuntee seuraavan (varsin suppean) joukon matemaattisia perusfunktioita:

cos(x) kosini (x radiaaneissa).

sin(x) sini (x radiaaneissa).

exp(x) e^x (luonnollinen antilogaritmi).

log(x) luonnollinen logaritmi.

int(x) kokonaisosa.

sqrt(x) neliöjuuri.

atan2(y,x) y/x:n arcustangentti väliltä $[-\pi, \pi]$.

rand() Palauttaa pseudosatunnaisluvun r , missä $0 \leq r < 1$.

srand(x) Alustaa uuden siemenluvun *rand()*-funktiolle. Mikäli siemenlukua ei alusteta, käytetään siemenlukuna aikaa. Palauttaa vanhan siemenluvun.

Esim. piirretään sinikäyrä merkkigrafiikalla:

```
#! /bin/awk -f
BEGIN{ PI = 4 * atan2(1,1)
    for (i = 0; i < 2*PI; i += 0.1)
       printf "%*c\n", sin(i) * 38 + 40, "*"
}
```

8.10 Merkkijonofunktiot

Seuraavassa on lueteltu *awk*in merkkijonofunktiot. Huomaa että monilla on sivuvaikutuksia, ts. ne muuttavat argumenttejaan.

gsub(r,s,t) Vaihtaa jonon *s* kaikkiin merkkijonomallin *r* osumiin jonossa *t*. Palauttaa suoritettujen vaihtojen määrän. Korvausjonossa & tarkoittaa korvattavaa merkkijonoa. Mikäli parametri *t* puuttuu käytetään `$0`:aa.

index(s,t) Palauttaa merkkijonon *t* paikan merkkijonossa *s* tai nollan mikäli jonoa *t* ei löydy jonosta *s*.

length(s) Palauttaa jonon *s* pituuden, ilman parametriä palauttaa $0:n pituuden.

match(s,r) Palauttaa paikan jonossa *s* josta merkkijonomalli *r* alkaa tai nollan mikäli merkkijonomallia *r* ei löydy. Asettaa arvot systeemimuuttujille RSTART (sama kuin palautettu arvo) ja RLENGTH (löydetyn merkkijonnon pituus, tai -1).

split(s,a,sep) Jakaa merkkijonon *s* taulokoksi *a* käyttäen kenttäerottimena parametriä *sep*, jos parametri *sep* puuttuu käytetään muuttujan FS arvoa kenttä erottimena. Palauttaa taulukon alkioiden lukumäärän.

sprintf(fmt,expr,...) Formatoi *expr*-lauseet *fmt*n määräämään asuun. Palauttaa tuloksen merkkijonona.

sub(r,s,t) Vaihtaa jonon *s* ensimmäisen merkkijonomalli *r* osuman paikalle jonossa *t*. Mikäli vaihto onnistuu palauttaa ykkösen, muuten nollan. Korvausjonossa & tarkoittaa korvattavaa merkkijonoa. Mikäli parametri *t* puuttuu käytetään `$0`:aa.

substr(s,p,n) Palauttaa alimerkkijonon merkkijonosta *s* alkaen paikasta *p* maksimissaan pituuteen *n* asti. Mikäli parametri *n* puuttuu palautetaan loppujono.

tolower(s) Muuttaa kaikki isot kirjaimet pieniksi jonossa *s* ja palauttaa uuden jonon.

toupper(s) Muuttaa kaikki pienet kirjaimet isoiksi jonossa *s* ja palauttaa uuden jonon.

Esim. muutetaan merkkijonon (muuttujassa `name`) ensimmäinen kirjain isoksi ja loput pieniksi:

```
Name = toupper(substr(name,1,1)) tolower(substr(name,2))
```

Esim. muutetaan ä:t ae:ksi ja ö:t oe:ksi `$0`:ssa:

```
gsub("ä","ae"); gsub("ö","oe")
```

Esim. tutkitaan onko merkkijonossa mahdollisesti olevan yhtäsuuruusmerkin perässä lainausmerkki:

```
if (match(string,"=")) {
   if (substr(string, RSTART+1, 1) ~ /[\"']/) ...
```

Esim. lasketaan `PATH`in kussakin komponentissa olevien suorituskelpoisten ohjelmien lukumäärä, ja tulostetaan ne suuruusjärjestyksessä lukumäärän mukaan:

```
#! /bin/awk -f
BEGIN{
  split(ENVIRON["PATH"],dirs,":")
  for (i in dirs) {
    cmd="ls -l " dirs[i]
    while (cmd | getline)
       if (/^-..x/) progs[i]++
    close (cmd)
  }
  for (i in dirs) print dirs[i], progs[i] | "sort -k2n"
}
```

tai

```
#! /bin/awk -f
BEGIN{
  split(ENVIRON["PATH"],dirs,":")
  for (i in dirs) {
    cmd="ls -l " dirs[i]
    progs=0
    while (cmd | getline && /^-..x/) progs++
    close (cmd)
    print dirs[i], progs | "sort -k2n"
  }
}
```

8.11 Omien funktioiden määrittely

Funktioita voi *awk*issa määritellä seuraavasti:

`function` *funktion_nimi* (*parametrilista*) `{`
  `komentoja`
`}`

missä parametrilista on (mahdollisesti tyhjä) lista muodollisten parametrien nimiä.

Funktiota kutsutaan sitten tyyliin *funktion_nimi(args)*. Kutsussa funktion nimen ja vasemman sulun välissä ei saa olla tyhjää.

Funktiomäärittelyt pitää sijoittaa varsinaisten ehto/toiminto-osien ulkopuolelle (yleensä muttei välttämättä ohjelman alkuun tai loppuun).

Funktio voi sisältää *return*-lauseen: `return` *expr*
Tällöin funktio palauttaa arvon *expr*. Ilman *expr*-osaa tai ilman `return`-käskyä funktio palaa ilman ennustettavaa arvoa.

Funktion nimenä ei saa käyttää muuttujan tai sen oman parametrin nimeä. Funktiot voivat kutsua toisiaan ja itseäänkin rekursiivisesti.

Esim. määritellään puuttuvat trigonometriset funktiot:

```
function tan(x) { return sin(x)/cos(x) }
function asin(x) { return atan2(x, sqrt(1-x*x)) }
function acos(x) { return atan2(sqrt(1-x*x), x) }
function atan(x) { return atan2(x,1) }
```

Esim. kertoman laskeminen rekursiivisesti:

```
function fact(n) { if (n<2) return 1; else return n*fact(n-1) }
```

Jos funktion kutsussa on vähemmän argumentteja kuin parametrilistassa on parametrejä, loput täydennetään tyhjiksi (nolliksi) ilman virhettä. Jos argumentteja on liikaa, voi sen sijaan tulla virheilmoitus.

Parametrit ovat lokaaleja funktion sisällä, kaikki muut muuttujat ovat globaaleja. Jos halutaan lokaaleja muuttujia, sellaisina voidaan käyttää ylimääräisiä parametreja. Tällöin ne on tapana laittaa omalle rivilleen ja kommentoida, esim. kertoman laskeminen iteratiivisesti:

```
function fact(n,\
              i,result)   # local variables
{
    result=1
    for (i=1; ++i<=n; ) result *= i
    return result
}
```

Parametri voi olla taulukko, ilman eri määritystä: riittää kun sitä käyttää funktion sisällä taulukkona. Taulukkoparametria täytyy kutsussa vastata taulukkoargumentti (se voi myös puuttua, jolloin siihen automaattisesti täydennetään tyhjä taulukko). Taulukko välittyy funktiolle referenssinä, ts. taulukkoparametrin alkion muuttaminen funktiossa muuttaa vastaavaa globaalia taulukkoa. Taulukkoa ei voi palauttaa funktion arvona.

Esim. yksikkömatriisin luonti:

```
# create identity matrix of size n
function identity_matrix(a,n,\
                         i,j)     # local counters
{
  for (i=1,n) {
```

```
      for (j=1,n) {
        a[i,j] = (i==j ? 1 : 0)
      }
  }
}
```

8.12 *Awk*in versioista

Edellä kuvattu POSIXin standardoima *awk* on laajempi kuin alkuperäinen. Tunnetuimpia versioita ovat *nawk*, johon POSIX *awk* perustuu, ja Gnu *awk* (*gawk*).

Seuraavat puuttuvat alkuperäisestä mutta ovat jo *nawk*issa:

Delete do function return atan2 cos sin rand srand gsub sub match close system FNR ARGC ARGV RSTART RLENGTH SUBSEP ?: ^ sekä *ERE*:n käyttö FS-muuttujassa ja *split()*-funktiossa.

Seuraavat ovat useissa hiukan uudemmissa versioissa:

Usean `-f`-option käyttö, `-v`, ENVIRON, toupper, tolower, lisää formaatteja printf'iin.

POSIXin tuoma uudistus on CONVFMT, aikaisemmissa versioissa OFMT:tä käytettiin siihenkin. POSIX myös täsmensi joitakin aikaisemmin epämääräisiä asioita, kuten milloin muuttujan tyyppi vaihtuu. (Esim. joissakin versioissa muuttujan tyyppi muuttuu kun siihen viitataankin, mikä aiheuttaa joskus hämmästyttäviä sivuvaikutuksia.)

Gnu *awk* on olennaisesti POSIX *awk* johon on lisätty iso joukko laajennuksia. Niitä ei tässä käsitellä sen enempää, mutta mainittakoon optio `--posix`, joka yrittää tehdä *gawk*ista POSIX-yhteensopivan (poistaa yhteensopimattomat laajennukset).

Mainittakoon vielä että monissa vanhoissa *awk*eissa `-Ft` asettaa kenttäerottimeksi t-kirjaimen asemesta tabulaattorin.

Useimmissa moderneissa järjestelmissä *awk* on kutakuinkin POSIX-yhteensopiva. Erityisenä poikkeuksena mainittakoon Solaris, jossa *awk* on yhä alkuperäisen määrityksen mukainen, ja sen lisäksi on erikseen *nawk*, jota on laajennettu POSIXin mukaiseksi.

8.13 Esimerkkejä

- Lasketaan hakemistossa olevien tiedostojen kokonaistavumäärä:

  ```
  ls -l | awk '/^-/{ t+=$5 } END { printf "%d\n", t }'
  ```

- Tulostetaan tiedostosta ensimmäinen rivi, jolla esiintyy sana 'Bingo':

  ```
  awk '/Bingo/{ print; exit }' "$file"
  ```

- Tiedostossa on ihmisten nimiä etunimet ensin, sukunimi lopussa, välilyönneillä erotettuna. Siirretään sukunimi alkuun ja erotetaan se etunimistä pilkulla ja välilyönnillä:

  ```
  awk '{ printf  "%s,", $NF
         for (i=0; ++i<NF;) printf " %s", $i
         printf "\n" }' "$file"
  ```

  tai

  ```
  awk '{ printf "%s, ", $NF; $NF=""; print }' "$file"
  ```

- Tulostettava tiedosto "kakkosvälillä" ts. joka rivin jälkeen tyhjä rivi:

  ```
  awk '{ printf "%s\n\n", $0 }' "$file"
  ```

  tai

  ```
  awk '{ print }' ORS='\n\n' "$file"
  ```

- "Context-grep": tulostettava halutun sanan sisältävä rivi sekä edeltävä ja seuraava:

  ```
  awk '/Kalle/{ print last; print; getline; print } { last=$0 }'
  ```
  tai
  ```
  awk '/Kalle/{ print last"\n"$0; getline; print } { last=$0 }'
  ```

  Noissa on se vika, että jos merkkijono löytyy ensimmäiseltä tai viimeiseltä riviltä, ylimääräinen tyhjä rivin tulostuu sitä ennen tai sen jälkeen. Tämän voisi korjata esim. näin:

  ```
  awk '/Kalle/{ if (NR>1) print last; print; if (getline) print }
              { last=$0 }'
  ```

- Tiedostossa `file1` on lukupareja x,y pilkulla eroteltuina, tiedostossa `file2` kaksoispisteellä eroteltuina ja tiedostossa `file3` x ja y aina omalla rivillään ja tyhjällä rivillä seuraavasta erotettuina. Kaikki tiedostot pitäisi tulostaa perä perää x,y-parit puolipisteellä erotettuina:

  ```
  awk -F, '{ print $1,$2 }' OFS=\; file1 FS=: file2 FS='\n' RS='' file3
  ```

- Tiedostossa on koordinaattipareja suorakulmaisessa koordinaatistossa, x ja y pilkulla erotettuna. Tulostetaan vastaavat napakoordinaatit kenoviivalla erotettuina:

  ```
  awk -F, '{ print sqrt($1*$1+$2*$2) "\\" atan2($2,$1) }' "$file"
  ```
  tai
  ```
  awk -F, '{ print sqrt($1*$1+$2*$2), atan2($2,$1) }' OFS='\' "$file"
  ```

tai
```
awk -F, '{ printf "%s\/%s\n", sqrt($1*$1+$2*$2), atan2($2,$1) }' "$file"
```

- *Awk*ista puuttuu osalausekeviittausmahdollisuus, joten ehdot tyyliin "sama sana kaksi kertaa" ovat vaikeita. Usein ne kuitenkin voi tehdä helposti valitsemalla kenttäerotin sopivasti.

  Etsitään `/etc/passwd`:stä käyttäjät joiden UID ja GID ovat samat:

  ```
  grep '^[^:]*:[^:]*:\([^:]*\):\1:' /etc/passwd
  grep '^.*:.*:\(.*\):\1:.*:.*:' /etc/passwd
  awk -F: '$3 == $4' /etc/passwd
  ```

- Etsitään rivejä joiden alussa ja lopussa on sama sana:

  ```
  awk -F'[^[:alpha:]]+' 'NF>1 && $1 == $NF'
  grep '^\([[:alpha:]][[:alpha:]]*\)[^[:alpha:]].*[^[:alpha:]]\1$
   ^\([[:alpha:]][[:alpha:]]*\)[^[:alpha:]]\1$'
  ```

- Edellistä oli vaikea tehdä `grep`illä mutta helppo *awk*illa. Toisaalta jos pitää etsiä rivejä joilla esiintyy sama vähintään 5-kirjaiminen merkkijono kaksi kertaa niin että esiintymät eivät mene päällekkäin, se on *awk*illa erittäin työlästä mutta `grep`illä helppoa:

  ```
  grep '\(.\{5\}\).*\1'

  #! /bin/awk -f
  /...../{
    for (i=1; i<=length()-9; ++i) {
      for (j=i+5; j<=length()-4; ++j) {
        if (substr($0,i,5) == substr($0,j,5)) print
      }
    }
  }
  ```

- Olkoon tiedostossa tietoa ihmisistä siten, että kustakin on ensin yhdellä rivillä nimi "sukunimi, etunimet" -muodossa, sitten muuta tietoa tuntematon määrä rivejä ja tyhjä rivi ennen seuraavaa nimeä. Tiedosto pitää lajitella aakkosjärjestykseen. Oletetaan että tiedostossa ei esiinny merkkijonoa `"!@newline@!"`:

  ```
  NL="!@newline@!"
  awk -v NL="$NL" 'BEGIN{RS=""} {gsub("\n",NL); print }' "$file" |
  sort |
  awk '{gsub("'"$NL"'","\n"); if (NR>1) printf "\n"; print }'
  ```

  Huomaa kaksi erilaista tapaa välittää shell-muuttuja `$NL` *awk*ille.

- Tiedostossa on tietoa ihmisistä, kaksi riviä kustakin: ensimmäisellä etunimet ja sukunimi välilyönneillä eroteltuina, toisella osoite. Tiedosto pitäisi lajitella aakkosjärjestykseen sukunimen ja toissijaisesti etunimien mukaan. Oletetaan että nimissä ei ole '`|`'-merkkejä.

```
#! /bin/awk -f
BEGIN{ TMPFILE="/tmp/.secret" rand()
  SORT="sort -o " TMPFILE }
{ printf "%s|", $NF | SORT
  $NF=""; printf "%s|", $0 | SORT
  getline; print | SORT }
END{ close(SORT); FS="|"
  while (getline < TMPFILE) printf "%s %s\n%s\n", $2,$1,$3
  system("rm " TMPFILE) }
```

tai

```
#! /bin/awk -f
BEGIN{ TMPFILE="/tmp/.secret" rand() }
{ printf "%s|", $NF >TMPFILE
  $NF=""; printf "%s|", $0 >TMPFILE
  getline; print >TMPFILE }
END{ close(TMPFILE)
  FS="|"
  while ("sort " TMPFILE | getline) printf "%s %s\n%s\n", $2,$1,$3
  system("rm " TMPFILE) }
```

tai vielä helpommin kahdella *awk*-kutsulla:

```
#! /bin/sh
awk '{ printf "%s|", $NF; $NF=""; printf "%s|", $0
       getline; print }' "$file" |
sort |
awk -F '|' '{ printf "%s %s\n%s\n", $2,$1,$3 }'
```

- Lisätään jokaisen .c-tarkenteisen tiedoston alkuun kommenttina tiedoston nimi (vrt. 8.5.1):

```
awk '
function mv() {
   if (OldFILENAME) {
     close(OldFILENAME ".new")
     system("mv " OldFILENAME ".new " OldFILENAME)
   }
}
FILENAME != OldFILENAME {
   mv()
   OldFilename=FILENAME
   print "/* " FILENAME " */" > FILENAME ".new"
}
{ print > FILENAME ".new" }
END { mv() }
' *.c
```

- Etsitään tiedostosta pisin muuttujanimeltä näyttävä merkkijono (alkaa kirjaimella tai alaviivalla, sitten kirjaimia, numeroita ja alaviivoja):

  ```
  #! /bin/awk -f
  { while (match($0, /[a-zA-Z_][a-zA-Z0-9_]+/)) {
      if (RLENGTH > maxlen ) {
        maxlen=RLENGTH
        longest=substr($0, RSTART,RLENGTH)
      }
      $0=substr($0, RSTART+RLENGTH)
    }
  }
  END { print longest }
  ```

  tai

  ```
  #! /bin/sh
  awk '{ gsub("[^a-zA-Z0-9_]+", "\n"); print }' "$@" |
  awk '/^[a-zA-Z_]/{ print $0, length() }' |
  sort -k2nr |
  awk 'NR==1 { print $1 }'
  ```

  tai

  ```
  #! /bin/sh
  awk '{ gsub("[^a-zA-Z0-9_]+", "\n"); print }' "$@" |
  awk 'length() > maxlen { longest=$0; maxlen=length() } END { print longest }'
  ```

- Argumentteja voi lukea ja muuttaa muuttujien `ARGC` ja `ARGV` avulla. Niitä ei edes ole pakko tulkita tiedostonimiksi. Seuraava skripti laskee yhteen argumenttinsa:

  ```
  #! /bin/awk -f
  BEGIN {
    for (i=1; i<=ARGC; ++i)
      s+=ARGV[i]
    print s
    ARGC=1
  }
  ```

- `OFS`-muuttujaa käytetään paitsi `print`-käskyssä, myös `$0`:n uudelleenrakentamisessa kun jotain kenttämuuttujaa muutetaan. Muutetaan kenttäerottimet kaksoispisteistä pilkuiksi:

  ```
  awk -F: -v OFS=, '{ $1=$1; print }'
  ```

8.14 Awk vs. sed

Awk ja *sed* ovat samantyyppisiä ja joskus on vaikea päättää kumpaa käyttäisi. Kannattaa myös muistaa `grep`, `tr` ja `cut`. Seuraavassa valinta-apua.

- Yksinkertaisissa editointitehtävissä *Sed* on yleensä helpompi ja vaatii vähemmän lainausmerkkejä kuin *awk*:

```
sed s/Kalle/Pekka/g
awk '{gsub("Kalle","Pekka"} {print}'

sed 1d
awk 'NR>1'

sed 10,20d
awk 'NR<10 || NR>20'

sed y/öäåÖÄÅ/oaaOAA/
awk '{gsub("ö","o");gsub("[äå]","a");gsub("Ö","O");gsub("[ÄÅ]","A"} {print}'
tr öäåÖÄÅ oaaOAA

sed '\-////-s+x+y+2'
awk '/\/\/\/\//{gsub("x","ThisWasSmallX"); gsub("x","y");
                gsub("ThisWasSmallX","x")}
    {print}'
```

- *Awk* on kätevä jos tieto jakaantuu helposti tietueisiin. ERE:n käyttäminen kenttäerottimena tarjoaa monipuolisia mahdollisuuksia, mutta se myös hukkaa informaatiota: jos tietoa pitää muokata siten, että erottimien erot (esim. kuinka monta välilyöntiä tai mikä nimenomainen välimerkki kenttien välissä oli) pitää säilyttää, *sed* voi sittenkin olla helpompi.

  Seuraavissa vaihdetaan kaksoispisteellä erotetuista kentistä 1 ja 3 keskenään, ensin niin että tulostetaan vain 4 kenttää (ylimääräiset hukataan), sitten niin että tulostetaan kaikki, kolmannessa samoin mutta erotin voi olla kaksoispiste tai pilkku (ja ne pitää säilyttää ennallaan):

```
sed 's/\([^:]*\):\([^:]*\):\([^:]*\):\([^:]*\).*/\3:\2:\1:\4/'
awk -F: -v OFS=: '{ print $3,$2,$1,$4 }'

sed 's/\([^:]*\):\([^:]*\):\([^:]*\)/\3:\2:\1/'
awk -F: -v OFS=: '{ tmp=$3; $3=$1; $1=tmp; print }'

sed 's/\([^:,]*\)\([:,]\)\([^:,]*\)\([:,]\)\([^:,]*\)/\5\2\3\4\1/'
awk -F'[:,]' '{ print $3 substr($0,length($1)+1,1) $2 \
       substr($0,length($1 $2)+2,1) $1 substr($0,length($1 $2 $3)+3) }'
```

- Monirivisten tietueiden käsittelyssä *awk* on kätevä, *jos* tietueet on eroteltu jollain yksittäisellä merkillä tai tyhjillä riveillä. Rivin lopussa olevan jatkorivimerkin käsittely sujuu myös, mutta jatkorivin alussa oleva on jo vaikeampi.

  Esim. lajiteltava tiedosto jossa on nimiä ja niiden perässä osoitteita ja erottimina tyhjät rivit:

```
awk -v RS='' -F'\n' -v OFS='|' '{ $1=$1; print }' "$file" |
sort |
awk -v FS='|' -v OFS='\n' -v ORS='\n\n' '{$1=$1; print}'

awk -v RS='' -F'\n' -v OFS='|' '{ $(NF+1)=""; print }' "$1" |
sort |
tr '|' '\n'

sed '
:a
$!N
s/\n\(.\)/|\1/
ta
s/\n/|/
P
D' "$file" |
sort |
sed 's/|/\
/g'
```

  Esim. tiedostossa on pilkulla erotettuja kenttiä riveittäin, jatkorivin merkkinä on kenoviiva rivin lopussa (ja sellainen voi olla kentän keskellä). Tulostettava viides kenttä joka tietueesta:

```
awk -F, '
{ while (/\\$/) {
    old=substr($0,length()-1)
    getline
    $0=old $0
  }
  print $5
}'

sed ':test
s/\\$//
$bp
tmore
s/,[^,]*/\
&\
/5
s/.*\n\(.*\)\n.*\1/p
d
```

```
:more
N
btest'

sed -e :a -e '/\\$/!n' -e 's/.$//' -e '$!N' -e ba' |
cut -d, -f5
```

Esim. kuten edellä mutta jatkorivin merkkinä on tyhjä rivin alussa.

```
awk '/^ +/{ sub("^ *",""); printf %"s",$0; next }
     NR>1 { printf "\n" }
     { printf "%s", $0 }
     END { printf "\n" }' "$file" |
awk -F, '{ print $5 }'

awk 'NR==1 { last=$0; next }
     /^ / { sub("^ *",""); last=last $0; next }
     { print last; last=$0 }
     END { print last }' |
awk -F, '{ print $5 }'

awk 'NR==1 { last=$0; next }
     /^ / { sub("^ *",""); last=last $0; next }
     { split(last,a,","); print a[5]; last=$0 }
     END { split(last,a,","); print a[5] }'

sed ':a
$n
N
s/\n  *//
ta
P
D' | cut -d, -f5
```