# Sisältö

# 1 Johdanto

## 1.1 Kurssin tavoite

Kurssi on tarkoitettu lähinnä Unix-järjestelmään sovellusohjelmia kirjoittaville ohjelmoijille sekä järjestelmän ylläpitäjille (tai sellaisiksi aikoville), ensisijaisena päämääränä oppia ymmärtään shellien toimintaa, käyttämään perustyökaluohjelmia sekä kirjoittamaan toimivia, luotettavia, tehokkaita ja kohtuullisen siirrettäviä shell-skriptejä.

## 1.2 Esitiedot

Pohjatiedoiksi oletetaan Unixin peruskäsitteiden tuntemusta (tiedostorakenne, hakemistot, prosessit) ja kokemusta sen interaktiivisesta käytöstä (editorit, tiedostojen ja hakemistojen luonti jne), perusohjelmointitaitoa jollakin sopivalla ohjelmointikielellä (vähintään kurssia `TIE120` *Ohjelmointi* vastaava) sekä C-kielen perusteita (ymmärtämistä).

## 1.3 Kirjallisuutta

- IEEE 1003.2 standard, 1992 ("POSIX.2")
- Bill Rosenblatt: Learning Korn shell (1st ed., O'Reilly 1993)
- Ellie Quigley: Unix shells by example (2nd ed., Prentice Hall 1999)
- Dale Dougherty & Arnold Robbins: sed & awk (2nd ed., O'Reilly 1997)

Kurssis kotisivu löytyy osoitteesta

`http://www.mit.jyu.fi/opiskelu/kurssit/unixshell01/`

## 2 Eri shelleistä ja standardeista

Erilaisia shellejä on Unix-maailmassa lukuisia, ja shell-ohjelmien eli *skriptien* pahin ongelma onkin heikko siirrettävyys.

Jonkinlaisena yhteisenä tekijänä voidaan pitää POSIXin (*Portable Operating System Interface* -komitea) vuonna 1992 hyväksyttyä standardia (*IEEE 1003.2*, yleisesti tunnettu nimellä *POSIX.2*; jatkossa "POSIX-yhteensopiva" tarkoittaa ko. standardin mukaista): Jokseenkin jokaisesta modernista Unixintapaisesta (ja aika monesta muustakin) käyttöjärjestelmästä löytyy jokin ainakin melkein POSIX-yhteensopiva shell. Samassa standardissa määriteltiin myös yleisten työkaluohjelmien ominaisuudet, ja niidenkin osalta useimmat Unixit alkavat olla aika standardinmukaisia, joskin eroja vielä löytyy. Toisaalta POSIX-yhteensopivia shellejä yms saa myös moniin hyvinkin vähän Unixmaisiin käyttöjärjestelmiin eri MS-Windowseista alkaen.

POSIXin shell-standardi pohjautuu lähinnä *Bourne* ja *Korn* shelleihin, ennen kaikkea jälkimmäisen versioon `ksh88`. Mitään "puhdasta POSIX-shelliä" ei sinänsä ole olemassa, esim. HP-UX:n "POSIX sh" sisältää joukon ksh88:n piirteitä jotka POSIX jätti standardista pois.

Ehkä tärkein POSIX-tyyppinen shell nykyisin onkin juuri Korn shell eri versioineen, etenkin `ksh88`, joka on nykyisin vakiona lähes kaikissa kaupallisissa Unixeissa (uudempi `ksh93` on harvinaisempi, vaikka se onkin ilmaiseksi saatavissa ei-kaupalliseen käyttöön); toinen erityismaininnan arvoinen on *Gnu-projektin* tuottama `bash`, jota etenkin Linux on levittänyt tehokkaasti.

Eri ympäristöissä POSIX-yhteensopivimpia shellejä ovat: HP-UX:ssä (versiosta 10 alkaen) `/usr/bin/sh`, SunOS:ssä `/usr/bin/ksh`, Linuxissa `/bin/ksh` tai sen puuttuessa `/bin/zsh` tai `/bin/bash`. (Erityisesti SunOS:ssä `/bin/sh` *ei* ole POSIX-yhteensopiva vaan klassinen Bourne sh. Muualla yleensä `/bin/sh` on linkki johonkin sopivaan shelliin.)

Olemassa on myös iso joukko shellejä, jotka eivät yritäkään olla POSIX-yhteensopivia, erityisesti *C-shell* (`csh`) variantteineen (kuten `tcsh`). Tällä kurssilla niihin ei juuri kajota, vaan keskitytään Bourne-tyyppisiin shelleihin ja erityisesti POSIXin standardoimiin piirteisiin; lisäksi lähinnä `ksh88`:n ja `bash`in tärkeimpiä lisäominaisuuksia käsitellään jonkin verran.

Samoin työkaluohjelmista käytetään ensisijaisesti POSIX-versioita, mutta siirrettävyyden kannalta tärkeimpiä eroja etenkin BSD:n (*Berkeley Standard Distribution*) versioihin otetaan myös esille.

(Mainittakoon erityisesti että SunOS:n `/bin/awk` *ei* ole POSIX-yhteensopiva, sen sijaan siellä on `/usr/xpg4/bin/awk` joka on sama kuin `/usr/bin/nawk` ja se on ainakin suunnilleen POSIXin mukainen.)

# 3 Mitä shell yleensä tekee?

Shellin historiallisesti ensimmäinen ja edelleen ensisijainen tehtävä on toimia laukaisualustana muille ohjelmille. Sen lisäksi nykyaikaiset shellit osaavat tehdä itsekin melkoisen määrän asioita, ja ovat sellaisenaankin monipuolisia ohjelmointikieliä. Silti shell-skriptit ovat suurelta osin muiden ohjelmien kutsumista ja niiden tulosten yhdistelemistä, ja olennainen osa shell-ohjelmointitaitoa onkin yleisten apuohjelmien hallitseminen.

Shell tulkitsee tiedostosta tai käyttäjältä lukemiaan komentoja normaalisti riveittäin. Interaktiivisessa käytössä välissä on komentorivin editointiin liittyviä toimintoja; niihin ei tässä puututa vaan tutkitaan mitä shell tekee saadessaan valmiin rivin käsiteltäväkseen.

## 3.1 Esimerkki minimalistisesta shellistä

Ohessa on yksinkertaisen shellin C-kielinen lähdekoodi. Sen kääntämiseen riittää
`cc -o minish minish.c`
(Olettaen että `cc` on ANSI-C kääntäjä; esim. HP-UX:ssä siihen tarvitaan optio `-Aa`.)

*Minish* on nimensä mukaisesti minimalistinen: Se ei ymmärrä mitään optioita, argumenttina sille voi antaa yhden tiedostonimen jolloin se lukee komentoja sieltä, muussa tapauksessa se lukee niitä `stdin`istä. Sen "kieli" on äärimmäisen yksinkertainen: se tunnistaa välilyönnit argumenttien erottimiksi, tulkitsee `#`:lla alkavat rivit kommenteiksi, ja tuntee yhden sisäisen komennon (`exit`) — ja osaa kutsua ulkoisia ohjelmia ja välittää niille parametreja.

Kaikki shellit osaavat samat asiat (jos kohta useimmat paljon muutakin), erityisesti viimeksimainittu on jokaisen shellin pääsilmukan ydin.

Moiseen minishellin tarvittava koodi on yllättävän lyhyt ja yksinkertainen, mutta se kannattanee kuitenkin käydä tarkemmin läpi:

Luettuaan komentorivin `minish` tekee seuraavat toimenpiteet:

- tarkistaa ettei rivi ole liian pitkä ja poistaa rivinvaihtomerkin sen lopusta
- hyppää alkutyhjien ohi
- tarkistaa onko ensimmäinen ei-tyhjä merkki risuaita (`#`) ja jos on, tulkitsee rivin kommentiksi ja lopettaa sen käsittelyn siihen
- paloittelee rivin välilyöntien kohdalta argumenttitaulukkoon `execvp()`:n haluamaan muotoon
- tutkii onko komento "`exit`" ja jos on, lopettaa ohjelman (paluuarvona joko nolla tai ensimmäisenä argumenttina mahdollisesti ollut kokonaisluku)

- käynnistää uuden (lapsi)prosessin joka yrittää suorittaa annetun komennon (sen nimisen ulkoisen ohjelman)
- odottaa lapsiprosessin päättymistä ennen seuraavan rivin lukemista.

Erityisesti lapsiprosessin käynnistysvaihe (`fork()`) on Unixille tyypillinen, ja sitä kannattaa miettiä niin että varmasti ymmärtää asian.

Tällä shellillä voidaan kirjoittaa ihan oikeita shell-skriptejä:

```
echo Hello, world
```

Jos tuo talletetaan tiedostoon `hello.sh`, se voidaan nyt suorittaa komennolla `minish hello.sh` (olettaen että `minish` on `$PATH`issa). Useimmissa Unixeissa siitä voidaan myös tehdä sellaisenaan suoritettava lisäämällä alkuun shell-määritys tyyliin `#!/home/t/tt/minish` ja asettamalla suoritusbitti päälle (`chmod +x hello.sh`), ja sitten sen voi suorittaa suoraan nimellä:

```
./hello.sh
```

Huomaa että `echo` on `minish`:lle ulkoinen komento, ja tuon skriptin toimivuus edellyttää, että se löytyy `$PATH`ista. Näin vaikka `minish` ei itse `$PATH`ista mitään ymmärräkään: se kuitenkin säilyttää sen arvon ja välittää sen edelleen `execvp()`:lle joka osaa sitä käyttää.

Joskus on vaikea tietää mitä shell tekee ja mitä shellin kutsuma komento tekee. Tällöin voi olla hyödyllistä katsoa mitä tapahtuu kun ko. komentoa kutsutaan `minish`:lla. Kokeile esim. seuraavia sillä ja jollain muulla shellillä:

```
echo hello,   world!
echo "hello,   world!"
echo *
echo \\\\
echo "\\\\"
echo '\\\\'
ls *
```

# 4 POSIX-shell-ohjelmoinnin perusteet

## 4.1 Shell-skriptien suoritustavat

Shell-skriptin voi suorittaa kolmella perustavalla:

1. Antamalla sen shellille argumenttina:
   `sh [`*options*`]` *file* `[`*argument...*`]`
   Tällöin tiedoston pitää olla oletushakemistossa tai `$PATH`issa olevassa hakemistossa, tai sille pitää antaa (absoluuttinen tai suhteellinen) hakemistopolku. (Huom. kaikki shellit eivät kunnioita `$PATH`ia tässä yhteydessä. Täydellinen polku toimii aina.)

   Tämä käynnistää uuden shellin, joka suorittaa skriptin. Uusi shell ei välttämättä ole sama kuin käynnissä oleva: tässä se voidaan (ja täytyy) nimetä suoraan komentorivillä (ja normaalit hakusäännöt määräävät mistä se löytyy, yleensä `$PATH`ista).

   Uusi shell perii nykyisen ympäristömuuttujat muttei sisäisiä muuttujia eikä kaikkia muitakaan asetuksia (tästä enemmän myöhemmin).

2. Kirjoittamalla tiedoston nimen sellaisenaan (mahdollisten argumenttien kanssa), on oltava `$PATH`issa, tai hakemistopolun kanssa. Huom. nyt sitä ei etsitä oletushakemistosta (ellei tämä ole `$PATH`issa). Lisäksi tiedoston suoritusbitin on oltava päällä (`chmod +x`). Siis esim.
   `./file.sh arg1 arg2`

   Tässäkin käynnistyy uusi shell suorittamaan skriptiä, mutta mikä, riippuu tiedoston sisällöstä hieman monimutkaisella tavalla, joka vielä on systeemiriippuvainen ja standardoimaton (POSIX nimenomaan jätti asian auki), mutta yleensä se käy näin:

   - Jos tiedoston ensimmäinen rivi alkaa `#!`, sen perässä olevaa tiedostoa käytetään skriptin suoritukseen (se käynnistetään niin että se saa skriptitiedoston argumentikseen). Ensimmäisen rivin pituudella ja siinä mahdollisesti olevien lisäargumenttien määrällä on usein systeemiriippuvaisia rajoituksia; yksi optio kuitenkin menee yleensä läpi.
   - Jos tiedoston alussa *ei* ole em. riviä, POSIX-shellit käynnistävät uuden kopion itsestään sitä suorittamaan. Vanhat shellit saattavat käynnistää (1) kopion itsestään, (2) Bourne sh:n tai (3) csh:n riippuen mm. siitä mitä muuta tiedoston alussa on.

   Pääsääntöisesti skriptitiedoston alkuun kannattaa aina laittaa `#!` -rivi, vaikkei POSIX sitä vaadikaan. (Huom. jotkin systeemit vaativat välilyönnin !:n perään, eikä se missään tiettävästi haittaa, joten sitäkin kannattaa käyttää.) Huomaa kuitenkin että polkua voi joutua muuttamaan siirrettäessä skriptiä koneesta toiseen (sh:n jne sijainti vaihtelee).

   Tämä tapa on itse asiassa sama jolla yleensäkin suoritetaan ulkoisia komentoja.

3. Kirjoittamalla `.` *file*
   ts. piste ja tiedostonimi; tiedoston hakuehdot kuten edellisessä (`$PATH` tai polku), mutta suoritusbittiä ei tarvita.

   Tämä eroaa edellisistä sillä ratkaisevalla tavalla, että shell suorittaa tiedoston sisältämät komennot itse: uutta prosessia ei käynnistetä, sisäiset muuttujat näkyvät (myös parametrit, komentorivillä tiedostonimen perässä mahdollisesti annetut argumentit eivät välity), kaikki asetukset säilyvät ja skriptin mahdollisesti tekemät muutokset muuttujiin yms jäävät voimaan. Tiedoston alussa mahdollisesti olevalla `#!`-rivillä ei ole merkitystä.

   Huom. kaksi edellämainittua toimii kaikissa shelleissä, tämä vain Bourne-tyyppisissä (mukaanlukien kaikki POSIX-yhteensopivat). Muissa shelleissä on kyllä yleensä omia vastaavia komentojaan (esim. csh:ssä `source`, mutta niillä voi aina kutsua vain saman shellin skriptejä).

## 4.2 Funktiot

Shellissä voi määritellä funktioita tyyliin

```
funcname() {
   ...
}
```

Funktiot toimivat kuin skriptit skriptin sisällä: Olennaisesti saman vaikutuksen voi saada kirjoittamalla uuden, erillisen skriptin, mutta funktiot ovat nopeampia jos niitä kutsutaan toistuvasti (jäävät muistiin) ja helpottavat koodin ylläpitoa, kun loogisesti yhteenkuuluvia kokonaisuuksia voi sijoittaa samaan tiedostoon.

Funktionmäärittelyn sisällä mahdollisesti olevia erikoismerkkejä *ei* käsitellä määrittelyvaiheessa, vaan vasta funktiota suoritettaessa.

Funktio perii kutsuvan skriptin ympäristön, trap-asetukset, optiot jne, mukaanlukien muuttujat *paitsi* parametrejä `$1, $2`..., jotka saavat arvoikseen funktion kutsussa annetut argumentit, sekä `$#`, joka on niiden lukumäärä. (Joissakin shelleissä myös `$0` muuttuu, mutta POSIXin mukaan sen ei pitäisi.)

Kaikki funktion käyttäjät muuttujat (em. parametreja lukuunottamatta) ovat globaaleja ja vaikuttavat kutsuvaan skriptiin.

Funktion suoritus loppuu sen lopettavaan }-merkkiin tai `return`-käskyyn, ja paluuarvo on sama kuin sen viimeksi suorittaman komennon. (Funktion *määrittelyn* paluuarvo on 0 jos se onnistui, muuten jotain positiivista.)

Funktiomäärittelyn voi poistaa komennolla `unset -f` *funcname*.

Funktiot eivät periydy alishelleille.

### 4.2.1 Funktiot ksh:ssa

Ksh88 eroaa POSIXista funktioiden käsittelyssä muutamalla tärkeällä tavalla:

- optiot (`set -o`) ja `trap`it ovat lokaaleja funktioissa
- funktioissa voi määritellä lokaaleja muuttujia komennolla `typedef`
- funktioita voi periyttää alishelleille (`export`)
- jos muuttuja `$FPATH` on määritelty, funktioita haetaan sen osoittamista hakemistoista
- funktio voidaan määritellä ladattavaksi muistiin vain tarvittaessa määreellä `autoload` (tai `typeset -fu`)
- funktioiden määrittelyyn on tarjolla toinenkin syntaksi:
  `function` *funcname* `{...`
  tai `typedef -f` *funcname* `{...`
  Alkuperäisessä ksh88:ssa eri funktiomäärittelytavat ovat identtisiä, mutta suunnitelmissa on niiden eriyttäminen siten, että POSIX-tyylinen määrittely toimisi POSIX-tyyliin (erityisesti optioiden ja trappien kanssa) ja `function`-tyylinen kuten ennenkin. Tämän varaan ei voi oikein laskea mutta kannattanee silti käyttää jälkimmäistä tyyliä jos kirjoittaa ksh-spesifisiä funktioita.

### 4.2.2 Aliakset ksh:ssa

POSIX ei tunne aliaksia, mutta jokseenkin kaikki POSIX-yhteensopivat shellit tuntevat. Ksh:ssa ne määritellään näin:

`alias` *name=definition*

Huom. =:n ympärillä ei saa olla tyhjää.

Aliaksille ei voi määritellä argumentteja, mutta aliaksen perässä oleva teksti jää sinne kun aliaskorvaus on tehty ja ajaa saman asian monessa yhteydessä (monimutkaisemmissa tarpeissa käytä funktiota).

Alias korvataan määrittelyllään vain rivin alussa, *paitsi* jos aliasmäärittely loppuu välilyöntiin, jolloin seuraavallekin sanalle tehdään aliaskorvaus tarvittaessa.

Aliaksia voi ketjuttaa, ts. aliaksen määrittely voi olla toinen alias.

Aliaksen voi poistaa komennolla `unalias` *name*. (`unalias` ei tunne mitään jokerimerkkejä, aliakset on poistettava yksitellen.)

Aliakset eivät periydy alishelleille.

Ksh määrittelee automaattisesti pysyvät aliakset joukolle yleisiä komentoja (sekä niiden löytämisen nopeuttamiseksi että *Troijan hevosten* teon vaikeuttamiseksi).

Yleensä skripteissä aliaksille on vähän tarvetta, funktiot toimivat paremmin; aliakset onkin ajateltu lähinnä interaktiiviseen käyttöön. Silti ne toimivat myös skripteissä ja saattavat aiheuttaa yllätyksiä ellei niitä ota huomioon.

## 4.3 Erikoismerkit

Shell tulkitsee joukon merkkejä omalla tavallaan, antaa niille erikoismerkityksen joka on poistettava (*suojattava* ne shellin tulkinnalta) jos sen haluaa estää.

Seuraavat erikoismerkit pitää *aina* suojata, jos niiden haluaa esittävän itseään:

```
| & ; < > ( ) $ ‘ \ " ’
```

sekä välilyönti, tabulaattori ja rivinvaihto.

Seuraavat pitää suojata joissakin yhteyksissä muttei aina (kannattaa suojata varmuuden vuoksi ellei ole varma):

```
* ? [ # ~ = %
```

Shell tunnistaa muitakin erikoismerkkejä joissakin yhteyksissä, mutta niitä ei tarvitse suojata koska niillä on erikoismerkitys vain ko. yhteydessä.

Alkuperäisessä Bourne sh:ssa oli erikoismerkitys myös hattumerkillä ^ (se tarkoitti samaa kuin putkimerkki | nykyisin), joten joskus näkee sitäkin suojattavan; nykyisissä shelleissä se ei ole tarpeen.

### 4.3.1 Suojauskeinot

Erikoismerkkien suojaukseen on kolme tapaa:

- Kenoviiva \ suojaa seuraavan erikoismerkin. Huom. Jos seuraava merkki *ei* ole erikoismerkki eikä siis tarvitse suojausta, kenoviiva menettää erikoismerkityksensä ja tulostuu. (Varoitus: Eri shellit ovat varsin epäjohdonmukaisia tässä. Kannattaa varoa suojaamasta kenoviivalla epävarmoja tapauksia.)
- Yksinkertaiset lainausmerkit `'...'` suojaavat kaikki välissään olevat erikoismerkit (niitä on aina käytettävä parittain), paitsi yksinkertaista lainausmerkkiä itseään (jota ei saa niiden väliin mitenkään).

- Tuplalainausmerkit `"..."` suojaavat välissään olevat erikoismerkit lukuunottamatta dollaria ($), korvausmerkkiä ‘ ja kenoviivaa (jolla suojattuna väliin saa tuplalainausmerkin). Jos tuplalainausmerkkien välissä on `$(...)`, lainausmerkit eivät vaikuta siellä välissä oleviin merkkeihin ennen niiden tulkintaa.

Huom. nämä suojaavat erikoismerkit *shellin* tulkinnoilta, eri komennot saattavat tulkita niitä tai muita merkkejä kuitenkin, ja jos sama merkki on erikoinen sekä shellin että jonkin komennon mielestä, se voi olla tarpeen suojata kahdesti, esim.
`tr '\\' /`
tai
`tr \\\\ /`
Kummassakin `tr` saa argumentikseen kaksi kenoviivaa jotka se puolestaan tulkitsee yhdeksi.
Samoin esim. option alun merkkinä toimivaa viivaa ei voi näin suojata, vaan siihen tarvitaan komennosta riippuva optio, yleensä `--`:
`rm -- -viivafile`
tai muu vastaava keino:
`rm ./-viivafile`

### 4.3.2 Erikoismerkkien tulkintajärjestys

Shell laajentaa tekstikorvauksia aiheuttavat erikoismerkit komentoriviltä seuraavan toimintajärjestyksen mukaisesti:

1. Tilde ~, muuttujat ja parametrit ja komentokorvaus tehdään (erikoismerkit ~, $ ja ‘ tulkitaan) vasemmalta oikealle (tai oikealta vasemmalle jos käytössä oleva kieli niin määrää)
2. Rivi pilkotaan $IFS:in sisällön määräämällä tavalla kentiksi
3. Tiedostonimet (polut) levitetään (*, ? jne)
4. Lainausmerkit poistetaan (\, ‘ ja ")

Operaattorit ja uudelleensuuntausmerkit käsitellään myöhemmässä vaiheessa.

## 4.4 Muuttujat

### 4.4.1 Sisäiset muuttujat

Yksinkertaisia muuttujia luodaan ja niille asetetaan arvo tyyliin

`varname=value`

Huom. yhtäsuuruusmerkin ympärillä ei saa olla välilyöntejä.

Muuttujanimessä sallittuja merkkejä ovat kirjaimet, numerot ja alaviiva; ensimmäisen on oltava kirjain.

Muuttujan arvoon viitataan syntaksilla `$varname` tai `${varname}`; ts. jos komentorivillä esiintyy `ă$varname`, se korvataan muuttujan `varname` arvolla.

Muuttujanimi on suljettava aaltosulkuihin silloin, kun seuraavakin merkki voisi olla osa muuttujanimeä (se on kirjain, numero tai alaviiva). Vertaa:

```
K=a
Kalle=n
echo $Kalle
echo ${K}alle
```

Muuttujan voi hävittää komennolla `unset` *varname* (yksitellen — POSIXin `unset` ei ymmärrä mitään jokerimerkkejä).

Muuttujasta voi tehdä vakion komennolla `readonly` *varname*`[=`*value*`]`. Kaikki vakiot voi listata komennolla `readonly -p`. Vakion arvoa ei voi muuttaa eikä sitä voi hävittää `unset`illa.

### 4.4.2 Ympäristömuuttujat

Ympäristömuuttujat eroavat sisäisistä siinä, että ne periytyvät aliprosesseille. Muuttujan voi määritellä ympäristömuuttujaksi komennolla. Käänteistä toimintoa ei ole mutta ympäristömuuttujankin voi hävittää `unset`illa.

`export` *varname...*

Lisäksi jos optio `-a` on voimassa, kaikista luotavista muuttujista tulee automaattisesti ympäristömuuttujia. Ympäristömuuttuja voi olla vakio (*readonly*), mutta *readonly*-attribuutti ei välttämättä periydy (riippuu shellistä).

### 4.4.3 Erikoisparametrit

Edellisten lisäksi shell käyttää joukkoa erikoismuuttujia, jotka tunnistaa siitä, että ne eivät ala kirjaimella. Niitä ei voi `export`ata ja ne käyttäytyvät joissakin tilanteissa muutenkin eri tavoin kuin normaalit muuttujat.

Jos `sh`-skriptille annetaan argumentteja (optioiden lisäksi), ne talletetaan ns. positioparametreihin (*positional parameters*) `$1, $2,`... (Funktion sisällä positioparametrit viittaavat funktion kutsussa annettuihin argumentteihin, kuten edellä on todettu.)

Sen lisäksi `sh` määrittelee automaattisesti seuraavat parametrit:

`*` Kaikki positioparametrit yhtenä merkkijonona

`@` Kaikki positioparametrit erillisinä

`0` skriptin nimi

`#` Positioparametrien lukumäärä

`?` Viimeksi suoritetun komennon palauttama status

`-` Voimassa olevat optiot

`$` Shellin process id

`!` Viimeksi käynnistetyn taustaprosessin id

Kaksi ensimmäistä tarvitsevat lisäselitystä. Ne eroavat toisistaan ainoastaan siinä, miten ne käyttäytyvät tuplalainausmerkeissä `"`: `"$*"` on yksi merkkijono joka sisältää kaikki parametrit välilyönnillä eroteltuina (tai oikeammin `$IFS`:n ensimmäisellä merkillä — enemmän siitä myöhemmin). Ts. `echo "$*"` käyttäytyy kuin
`echo "$1 $2 $3 ..."`
`"$@"` taas toimii kuin kaikki parametrit olisi yksitellen suljettu lainausmerkkeihin:
`echo "$1" "$2 "$3" ....`

### 4.4.4 Systeemimuuttujat

POSIX määrittelee seuraavat yleiset ympäristömuuttujat — ei vaadi että niiden pitäisi olla olemassa mutta jos ne ovat, niiden merkityksen pitää olla tämä:

`HOME` Käyttäjän kotihakemisto

`LANG` Oletuskieli (*locale*) niille, joille sitä ei ole erikseen määrätty `LC_*`-muuttujalla

`LC_ALL` Kieliasetus - ohittaa `LANG`- ja muut `LC_*`-muuttujat

LC_COLLATE Kieliasetus aakkosjärjestyksen määräämiseen (lajittelu, regexp-luokat)

LC_CTYPE Kieliasetus merkkien luokitteluun (määrää mitkä ovat numeroita ja kirjaimia jne)

LC_MESSAGES Kieliasetus systeemin viesteille

LC_MONETARY Kieliasetus rahamäärien muotoilulle

LC_NUMERIC Kieliasetus lukujen muotoilulle (esim. desimaalipilkku vs. -piste)

LC_TIME Kieliasetus päivämäärän ja kellonajan muotoilulle

LOGNAME Käyttäjätunnus

PATH Hakupolku, lista tiedostoista joista komentoja etsitään, eroteltuina kaksoispisteillä. Tyhjä (kaksi peräkkäistä kaksoispistettä tai kaksoispiste alussa tai lopussa) vastaa oletushakemistoa (`.`) — tätä ei kuitenkaan pitäisi käyttää vaan mieluummin eksplisiittistä pistettä jos sitä halutaan.

SHELL Käyttäjän oletusshell

TMPDIR Käyttäjien kirjoitettavissa oleva tilapäinen hakemisto

TERM Päätetyyppi

TZ Aikavyöhyke

Nuo ovat siis periaatteessa yhteisiä kaikille komennoille. Lisäksi yksittäiset POSIXin tuntemat komennot käyttävät seuraavia muuttujia:

CDPATH (cd), COLUMNS (ls), DEAD, MAILRC (mailx), IFS (sh), LPDEST, PRINTER (lp), MAKEFLAGS (make), OPTARG, OPTIND (getopts).

Näihin palataan tarkemmin myöhemmin.

### 4.4.5 sh:n toimintaan vaikuttavat ympäristömuuttujat

POSIX määrittelee sh:n ei-interaktiiviseen toimintaan vaikuttaviksi vain muuttujat HOME, PATH, LANG, LC_ALL, LC_COLLATE, LC_CTYPE ja LC_MESSAGES, jotka on kuvattu edellä, sekä sh-spesifin muuttujan IFS.

IFS (*Input Field Separators*) määrittelee erotinmerkit syöttörivien lukemiseen ja parametrien ja komentokorvauksen tulosten kenttien erotteluun (siitä enemmän näiden yhteydessä). Oletusarvo on välilyönti + tab + newline (myös jos sitä ei ole asetettu lainkaan, mutta ei jos se on tyhjä).

POSIX.2 välttää ottamasta kantaa sh:n interaktiiviseen käyttöön, mutta luettelee kuitenkin seuraavat siihen mahdollisesti vaikuttavia muuttujia:

`ENV FCEDIT HISTFILE HISTSIZE LINENO MAIL PPID PS1 PS2 PS4`

Osa näistä vaikuttaa ainakin epäsuorasti skripteihinkin, joten palaamme niihin myöhemmin. (Ksh käyttää muuttujaa `PS3` `select`-komennossaan.)

### 4.4.6 Muuttujien käsittely

Muuttujien (ja parametrien) arvoon viitatessa niille voi samalla tehdä joitakin testejä ja muokkaustoimintoja. Yleinen syntaksi on `${`*varname oper [arg]*`}` eli ${}:n sisällä muuttujan nimen perään lisätään operaattorimerkki ja sen argumentti.

Seuraavat testaavat muuttujien olemassaoloa:

`${varname:-word}` Jos nimetty muuttuja on olemassa ja epätyhjä, palauttaa sen arvon, muuten tekstin *word*.

`${varname:=word}` Kuten yllä, mutta lisäksi asettaa muuttujan arvoksi *word*

`${varname:+word}` Jos nimetty muuttuja on olemassa ja epätyhjä, palauttaa tekstin *word*, muuten tyhjän.

`${varname:?message}` Jos nimetty muuttuja on olemassa ja epätyhjä, palauttaa sen arvon, muuten aiheuttaa virheen viestinä *varname: message* (tai *varname*: `parameter null or not found` jos *message* on tyhjä)

Kaikissa noissa kaksoispisteen voi jättää pois, jolloin ne hyväksyvät myös tyhjän muuttujan.

Seuraavilla voi muokata (tutkia) muuttujan arvoa, tai palauttaa vain osan siitä:

`${#varname}` Palauttaa muuttujan pituuden (merkkeinä)

`${varname%pattern}` Poistaa muuttujan lopusta lyhimmän *pattern*ia vastaavan merkkijonon, jos mahdollista, ja palauttaa tuloksen (ei muuta itse muuttujaa)

`${varname%%pattern}` Kuten yllä, mutta poistaa lopusta *pisimmän mahdollisen* merkkijonon

`${varname#pattern}` Kuten yllä, mutta poistaa *alusta* lyhimmän merkkijonon

`${varname##pattern}` Kuten yllä, mutta poistaa alusta *pisimmän mahdollisen* merkkijonon

### 4.4.7 Positioparametrien käsittely: shift ja set

Positioparametrien käsittelyyn on kaksi erityiskomentoa (funktion sisällä nevaikuttavat funktion parametreihin):

`shift` *[n]*

hukkaa `n` ensimmäistä parametria (`$0`:aa ei lueta mukaan) ja siirtää seuraavia vastaavasti alkuun päin (`$`$n$:stä tulee `$1` jne).

`set` *[--] par1 ...*

asettaa kerralla kaikki positioparametrit uusiksi. Rajoitin `--` tarvitaan jos ensimmäinen parametri alkaa viivalla. (Näin siksi että set-komennolla on toinenkin tehtävä: sillä voi asettaa shellin optioita.)

### 4.4.8 Muuttujan tilapäinen sijoitus

Muuttujalle voi antaa arvon vain yhden komennon suorituksen ajaksi:

```
var=value command
```

Tällöin muuttuja asetetaan suoritettavan komennon ympäristöön mutta asetus ei jää voimaan komennon päätyttyä. (Poikkeuksena erityiset sisäiset komennot, joista myöhemmin enemmän.)

### 4.4.9 Ksh: tyypitetyt muuttujat ja taulukot

Bourne sh ja POSIX sen myötä tuntee vain yhdentyyppisiä muuttujia, merkkijonoja, joita sitten voidaan tarvittaessa tulkita luvuiksikin. Ksh88 tuntee erillisen kokonaislukutyypin, taulukot sekä joukon muita ominaisuuksia joita muuttujille voi määritellä. Ksh93 tuntee liukuluvutkin.

Muuttujille annetaan tyyppi komennolla `typeset`, jolla on melkoinen liuta optioita, seuraavassa tärkeimmät:

- `-i` kokonaislukumuuttuja
- `-l` merkkijono jossa isot kirjaimet muutetaan automaattisesti pieniksi
- `-u` merkkijono jossa pienet kirjaimet muutetaan automaattisesti isoiksi
- `-r` read-only muuttuja, jota ei voi sen jälkeen muuttaa

Taulukota puolestaan voi luoda komennolla
`set -A` *nimi alkio1 alkio2 ...*
tai alkioittain tyyliin `x[0]=a`, ja taulukon alkioihin viitataan tyyliin `${x[a]}`. Alkioiden indeksointi alkaa nollasta ja yläraja voi olla suhteellisen pieni (ksh88:ssa 1023).

## 4.5 Syöttö ja tulostus

### 4.5.1 Uudelleensuuntaus

Komentojen (mukaanlukien funktiot) syöttöä ja tulostusta voi ohjata seuraavilla tavoilla:

*n*>*file* Ohjaa tiedostokahvan *n* (tai sen puuttuessa 0:n eli standard outputin) tiedostoon *file*; jos tiedosto on ennestään olemassa se tuhoutuu, tai jos `noclobber` -optio on voimassa, aiheutuu virhe. tulostuksen tiedostoon *file*

*n*<*file* Ohjaa tiedostokahvan *n* (tai sen puuttuessa 1:n standard inputin) lukemaan tiedostosta *file*

*n*>>*file* Kuten > mutta kirjoittaa mahdollisesti olemassaolevan tiedoston loppuun.

*n*> | *file* Kuten > mutta tuhoaa olemassaolevan tiedoston optiosta `noclobber` riippumatta

*n*<>*file* Kuten > mutta suuntaa sekä syötön että tulostuksen

*n*<<*word* Kääntää syötön lukemaan perässä seuraavaa tekstiä (ns. *here-document*) sanaan *word* saakka

*n*>&*m* Kääntää tiedostokahvan *n* (tai stdoutin) tulostuksen ennestään auki olevaan tiedostokahvaan *m*, tai jos *m*:n paikalla on `-`, sulkee tiedostokahvan *n*

*n*<&*m* Kääntää tiedostokahvan *n* (tai stdinin) lukemaan auki olevasta tiedostokahvasta *m*, tai jos *m*:n paikalla on `-`, sulkee tiedostokahvan *n*

*cmd1*| *cmd2* Ohjaa *cmd1*:n stdoutin *cmd2*:n stdiniin. Huom. *cmd2* saatetaan suorittaa samassa (ksh tekee näin) tai alishellissä; yleensä siihen ei voi luottaa kumminkaan päin.

Ksh tuntee lisäksi seuraavat:

*cmd*|& Käynnistää *co-prosessin*: siirtää komennon suorituksen taustalle mutta jättää sen I/O:n sidotuksi emoprosessiin

*n*>&`p` ohjaa tulostuksen *co-prosessille* *n*<&p ohjaa syötön *co-prosessille*

Huom. Tiedostokahvanumero *n* **pitää** erottaa edeltävästä tekstistä välilyönnillä tms, ja toisaalta sen ja siihen kuuluvan uudelleensuuntausmerkin välissä ei **saa** olla välilyöntiä. (Sen jäljessä, ennen tiedostonimeä, tyhjää saa olla muttei tarvita.) Vertaa seuraavia:

```
cat file22>foo
cat file2 2>bar
echo 1 2>foo
echo 1 2 >bar
```

Tiedostonumeron yläraja on toteutuskohtainen, kuitenkin ainakin 0-9 toimivat aina, ja 0 (stdin), 1 (stdout) ja 2 (stderr) ovat valmiiksi auki.

Uudelleensuuntausmerkit tulkitaan ennen komentoa, niinpä ne voi sijoitella komentoriville melko vapaasti. Vertaa:

```
echo kala >fisu
>fisu echo kala
echo >fisu kala
echo 4 + 2 > 5 - 3
echo 4 + 2> 5 - 3
echo 4 +2> 5 - 3
echo 4+2>5-3
```

Huom. erityiset sisäiset komennot ja varatut sanat voivat käyttäytyä poikkeavasti tässä suhteessa. Pääsääntöisesti uudelleensuuntaukset kannattanee sijoittaa komennon jälkeen.

### 4.5.2 Here-dokumenteista

Ns. *here-document* on erikoistapaus uudelleensuuntauksesta: se ohjaa syötön tulemaan skriptissä itsessään seuraavasta tekstistä rajasanaan saakka. Jos sana tai sen osakin on suojattu, teksti luetaan sellaisenaan, muussa tapauksessa tekstilaajennukset tehdään kuten tuplalainausmerkeissä. Lisäksi sanan alussa mahdollisesti oleva tavuviiva tarkoittaa sisennystä tab-merkeillä (ei välilyönnein).

Esim. Tulostetaan muuttujien `X`, `Y` ja `Z` nimet ja arvot:

```
cat <<EOF
X=$X
Y=$Y
```

```
Z=$Z
EOF
```

### 4.5.3 Command substitution

Komennon tulostuksen voi muuttaa merkkijonoksi komentorivillä kahdella vaihtoehtoisella syntaksilla:

1. takaheittomerkillä ‘ komennon molemmin puolin:
   `x=‘date‘`
2. dollaria seuraavilla suluilla:
   `x=$(date)`

Jälkimmäinen on jokseenkin aina parempi vaihtoehto, se on selkeämpi erityisesti useampia sisäkkäisiä korvauksia käytettäessä ja erikoismerkkien suojaussääntö sille on parempi.

Ksh (mutta ei POSIX) tuntee erikoistapauksena syntaksin `$(<`*file*`)`, joka tekee olennaisesti saman kuin `$(cat` *file*`)` käynnistämättä uutta prosessia.

### 4.5.4 read

Komento

`read [-r]` *muuttujalista*

lukee yhden rivin *stdin*istä (jonka tietty voi uudelleensuunnata), ja purkaa sen `$IFS`:n sisällön rajaamiin kenttiin ja tallettaa tulokset listan muuttujiin. Jos muuttujia on liian vähän, kaikki loput kentät talletetaan viimeiseen, jos liikaa, viimeiset jäävät tyhjiksi. Jos rivi loppuu kenoviivaan, lukee seuraavankin rivin paitsi jos `-r` optio on annettu.

Esim. Mitä seuraava tekee? Miksi? Vertaa ksh:ta ja bashia.

```
echo a b | read x
echo $x
```

### 4.5.5 IFS

Muuttuja `IFS` vaikuttaa syöttöön sen verran monimutkaisella tavalla, että se ansaitsee oman kappaleensa.

POSIXin mukaisissa shelleissä `IFS` vaikuttaa vain muuttuja- ja komentokorvauksen tulokseen, ei kuitenkaan tuplalainausmerkkien sisällä, sekä `read`-komentoon. Vanhemmissa shelleissä se saattaa vaikuttaa komentorivin tulkintaan muutenkin.

`IFS`:n eri arvot vaikuttavat seuraavasti:

1. Jos IFS on (oletusarvo) välilyönti+tab+rivinvaihto, tai sitä ei ole lainkaan, ko. merkit syötön alussa ohitetaan ja mielivaltainen ei-tyhjä joukko niitä toimii yhtenä erottimena.
2. Jos IFS on tyhjä, kenttiä ei pilkota lainkaan.
3. Muussa tapauksessa sovelletaan järjestyksessä seuraavia:
   (a) `IFS`:n mahdollisesti sisältämät tyhjämerkit ohitetaan rivin alussa ja lopussa
   (b) jokainen ei-tyhjä `IFS`:n merkki ja sen ympärillä mahdollisesti olevat tyhjät `IFS`:n merkit tulkitaan yhdeksi erottimeksi
   (c) ei-tyhjä joukko `IFS`:ssä olevia tyhjiä tulkitaan yhdeksi erottimeksi

Esimerkki:

```
echo x , y   z |(IFS=' ,' read a b c; echo "a=$a,b=$b,c=$c")
a=x,b=y,c=z
```

Huom. Tämä tuntuu olevan vaikea asia, etenkin viimeinen kohta; esim. `bash` ei osaa sitä ainakaan vielä versiossa 2.03. Yleensä ei kannattane käyttää `IFS`:ssä oletuksen lisäksi kuin yksimerkkisiä arvoja (ja sittenkin varautua siihen että tyhjiä hukkuu kuitenkin).

Lisää esimerkkejä:

```
x='1 + 2'
expr $x
IFS=:
expr $x
expr 1 + 2
```

#### 4.5.6 echo

Echo tulostaa merkkijonon stdout'iin. Sillä ei ole mitään siirrettäviä optioita: SysV:n ja BSD:n versiot olivat liian erilaisia että POSIXkaan olisi saanut niitä yhdistettyä.

BSD:n `echo` tuntee yhden option: `-n`, joka jättää rivinvaihdon tulostuksesta pois.

SysV:n `echo` ei tunne optioita mutta sen sijaan tulkitsee argumenteissaan olevia kenoviivoja seuraavasti:

\\`b` backspace

\\`c` jättää rivinvaihdon pois

\\`f` formfeed (sivunvaihto)

\\`n` ylimääräinen rivinvaihdon

\\`r` cr (telanpalautus)

\\`t` tab

\\`v` vertical tab

\\\\ \\

\\`0`*num* oktaalilukua *num* vastaava merkki

Gnu-projektin echo toimii kuten BSD:n mutta tunnistaa lisäksi option `-e`, joka saa sen tulkitsemaan kenoviivan sysV:n tyyliin. (Gnu echo:n uudemmat versiot tuntevat muitakin optioita.)

Yleisesti `echo`a ei voi käyttää siirrettäväksi aiotussa skriptissä jos pitää tulostaa kenoviivoja, viivalla alkava rivi tai jotakin ilman rivinvaihtoa.

### 4.5.7 Ksh: print

Ksh tarjoaa `echo`n korvikkeeksi komentoa `print`. Se ymmärtää sekä BSD:n `echo`:n `-n` -option että samat kenoviivakonventiot kuin SysV:n `echo`, mutta ne saa pois päältä optioilla `-r`. Muista optioista mainittakoon `-u`*n*, joka ohjaa tulostuksen tiedostokahvaan *n*, sekä `-p`, joka ohjaa sen `co-prosessille`.

### 4.5.8 printf

`printf` *format [arguments. . . ]*

tarjoaa muotoillun tulostuksen suunnilleen samalla tavalla kuin C:n *printf()* (liukulukutulostusta lukuunottamatta). Lisäformaattina `%b` ymmärtää samat kenoviivakonventiot kuin sysV:n echo. Formaateilla `b`, `c` ja `s` argumentti tulkitaan aina merkkijonoksi, muuten C:n mukaisesti.

Jos argumentteja on enemmän kuin formaatissa kenttämäärityksiä, formaattia käytetään uudelleen alusta.

Haluttaessa vähänkään monimutkaisempaa tulostusta, `printf` on ainoa edes kohtuullisen siirrettävä keino.

### 4.6 Shell pattern matching (globbing)

Shell käyttää tiedostonimien tulkintaan ja joihinkin testikomentoihin (`case`) omaa, yksinkertaista merkkijonolausekesyntaksiaan. Erityisesti sitä käytetään tiedostonimien generoimiseen (sekä ulkoisten komentojen nimissä että argumenteissa). Myös jotkin ulkoiset komennot käyttävät samaa syntaksia (erityisesti `find`).

#### 4.6.1 Tavalliset jokerimerkit

POSIX määrittelee seuraavat jokerimerkit:

? Vastaa yhtä mielivaltaista merkkiä

* Vastaa mielivaltaista (myös tyhjää) merkkijonoa

[. . . ] Vastaa yhtä merkkiä hakasulkulausekkeen määräämästä joukosta (ks. alla).

Muut merkit vastaavat itseään (huomaa kuitenkin että kohdassa 4.3 luetellut shellin erikoismerkit on tarvittaessa suojattava).

Huomaa että jokerilausekkeen on vastattava verrattavaa merkkijonoa kokonaisuudessaan (`a*` vastaa vain a:lla *alkavia*), toisin kuin esim. `grep`in kanssa.

#### 4.6.2 Hakasulkulausekkeet

Hakasulkulauseke voi sisältää

- Yksittäisiä merkkejä; erikoismerkitys on merkeillä ], [, !, -, ja mahdollisesti ˆ (sen sijaan * ja ? käyttäytyvät siellä normaaleina merkkeinä). Ne voi tarvittaessa suojata normaaliin tapaan kenoviivalla \ tai lainausmerkeillä.
- Merkkivälejä tyyliin `[a-z]` tai `[0-9a-fA-F]` jne. Merkkien on oltava oikeassa järjestyksessä (`[z-a]` on laiton). Huom. merkkivälin merkitys riippuu käytettävästä kielestä joten niitä tulisi välttää; vain väli `[0-9]` osaväleineen toimii aina.
- Jos ensimmäinen merkki [:n jälkeen on huutomerkki (!), lausekkeen merkitys kääntyy päinvastaiseksi: "Mitä tahansa *paitsi*. . . ". Muualla kuin heti [:n jäljessä ! käyttäytyy normaalisti. (ˆ *voi* myös käyttäytyä näin, POSIX kehottaa varautumaan siihen mutta ei edellytä sitä.)
- Jos oikea hakasulku ] esiintyy heti alku-[:n tai sitä seuraavan !:n perässä, se edustaa itseään eikä lopeta lauseketta.

- `[:`*class*`:]` edustaa merkkiluokkaa, *class* voi olla jokin seuraavista:

  `[:alnum:]` kirjain tai numero

  `[:alpha:]` kirjain

  `[:blank:]` tyhjä (välilyönti, tab, mutta ei rivinvaihto)

  `[:cntrl:]` kontrollimerkki

  `[:digit:]` numero

  `[:graph:]` näkyvä merkki (ei välilyönti)

  `[:lower:]` pieni kirjain

  `[:print:]` tulostuva merkki (myös välilyönti)

  `[:punct:]` välimerkki

  `[:space:]` tyhjä (myös rivinvaihdot jne)

  `[:upper:]` iso kirjain

  `[:xdigit:]` heksadesimaalinumero

  Näiden merkitys riippuu kielivalinnasta, semantiikka on aina sama; esim. `[:lower:]` vastaa myös kirjaimia “äöå”, jos kieli on valittu sopivasti.

- `[.`*symb*`.]` edustaa lajittelusymbolia (*collating symbol*). Tarvitaan lähinnä kielissä joissa useamman merkin ryhmä käyttäytyy kuin yksi merkki (esim. ch tai ll espanjassa).

- `[=`*symb*`=]` edustaa ekvivalenssiluokkaa, merkkejä jotka ovat lajitteluarvoltaan samoja (esim. [=a=] vastaa ranskassa mm. merkkejä a, á, à).

### 4.6.3 Tiedostonimien lisärajoitukset

Tiedostonimiä laajennettaessa on voimassa seuraavat rajoitukset:

- Kauttaviivat `/` on annettava eksplisiittisesti, mikään em. jokereista ei vastaa sitä (ei edes `[/]`).

- Piste tiedostonimen alussa on annettava eksplisiittisesti. Erikseen lueteltuna hakasuluissa (esim. `[.]*`) piste saattaa toimia nimen alussakin tai sitten ei (”unspecified”).

- Jokaiselle täsmättävän tiedostopolun osalle on oltava lukuoikeus sen hakemistoon, tai hakuoikeus niille osille joissa ei ole jokerimerkkejä.

- Tiedostonimet laajennetaan aakkosjärjestyksessä voimassa olevan kieliympäristön mukaisesti.

### 4.6.4 Ksh:n lisäjokerit

Ksh tuntee edellisten lisäksi seuraavat listaoperaattorit, kun lista on joukko pystyviivalla | erotettuja merkkijonoja (joissa voi itsessään olla jokereita):

`?(lista)` Vastaa nollaa tai yhtä listan alkioista

`*(lista)` Vastaa nollaa tai useampaa listan alkioista

`+(lista)` Vastaa yhtä tai useampaa listan alkioista

`@(lista)` Vastaa tasan yhtä listan alkioista

`!(lista)` Vastaa mitä tahansa *paitsi* listan alkioita

Esim. `?(a|bb)+([xyz]|kala)` vastaa merkkijonoja x, y, z, kala, ax, ay, az, akala, bbx, bby, bbz ja bbkala.

### 4.6.5 Tilde

Tilde ˜ on tavallaan jokerimerkki, mutta se käsitellään eri tavalla ja eri vaiheessa kuin edellämainitut. Se korvataan vain rivin alussa tai tyhjän perässä ja muuttujasijoituksessa yhtäsuuruusmerkin sekä kaksoispisteen jäljessä, sitä seuraavan käyttäjätunnuksen tai jos sitä seuraa välittömästi / tai tyhjää, `$HOME`:n arvolla.

Sen aikaisesta käsittelystä seuraa erityisesti, että sitä voi käyttää muuttujasijoituksessa (vaikkapa `PATH`issa). Sen sijaan se ei toimi sijoituksennäköisessä argumentissa, esim.
`make -k kala DIR=~/xyzzy`
*ei* toimi.

## 4.7 Erilaisista komennoista

Termiä “komento” käytetään yleisesti kaikista sanoista, jotka shell ymmärtää rivin alussa. Näitä on kuitenkin useampaa tyyppiä, jotka käyttäytyvät hieman eri tavoin. Jotkin komennot myös muistuttavat operaattoreita, joten nämäkin esitellään tässä yhteydessä. (Myöskään “operaattori” ei ole mitenkään yksikäsitteinen sana yleisessä kielenkäytössä.)

### 4.7.1 Operaattorit

POSIX luokittelee operaattoreiksi (*control operators*) seuraavat:

`&` `&&` `(` `)` `;` `;;` | || sekä rivinvaihto ja EOF.

Näitä ei tarvitse erottaa sanoista tyhjällä eikä välimerkeillä (toisistaan ja muista erikoismerkeistä joskus, jos muuten olisi sekaannuksen vaara).

### 4.7.2 Varatut sanat

Nämä ovat aina shellin sisäisesti tulkitsemia, niitä ei saa suojata (`'if'` ei toimi),ja ne toimivat vain tietyissä yhteyksissä. Niitä ei myöskään voi käyttää ulkoisen komennon nimenä (eivät löydy `PATH`ista, joskin eksplisiittisen polun kanssa samannimisen komennon voisi suorittaa), ja mahdollisen uudelleensuuntaus voi toimia tavanomaisesta poikkeavalla tavalla.

POSIXin määrittelemät varatut sanat ovat:

`! case do done elif else esac fi for if in then until while { }`

Standardi mainitsee myös (ksh:n tuntemat) sanat

`function select [[ ]]`

sekä kaksoispisteellä alkavat sanat (“unspecified results”).

Varattujen sanojen käsittely eroaa operaattoreista mm. siinä, että ne pitää erottaa yhteydestä tyhjällä tai muulla erotinmerkillä, ja toisaalta varsinaisista komennoista syntaktisten rajoitteidensa takia (erilaisia eri sanoille, esim. `in` saa esiintyä vain `for`- ja `case`-lauseiden yhteydessä).

### 4.7.3 Erityiset sisäiset komennot

POSIX määrittelee seuraavat komennot erityisiksi ("special built-in commands"): ne ovat aina sisäisiä (shell suorittaa ne itse), mahdollinen syntaksivirhe saattaa aiheuttaa shellin suorituksen keskeytyksen, niillä voi olla erityisiä syntaktisia ominaisuuksia (voivat vaikuttaa muun komentorivin tulkintaan), ja niiden yhteydessä mahdollisesti tehtävä muuttujasijoitus jää voimaan komennon jälkeen:

`break continue : . eval exec exit export readonly return set shift trap unset`

Näitä ei ole olemassa ulkoisina eikä niitä voi suorittaa `exec`illä (epäsuorastikaan, vrt. `find`).

### 4.7.4 Tavalliset sisäiset komennot

Tavalliset sisäiset komennot ("regular built-in utilities") ovat komentoja, jotka joidenkin erikoispiirteiden takia lähes aina toteutetaan sisäisinä. Esimerkiksi shellin ympäristöön vaikuttavat komennot ovat tällaisia. Niistä pitää kuitenkin olla ulkoinenkin versio, ts. ne pitää pystyä suorittamaan `execv()`:llä. POSIXin luettelee erikseen seuraavat tavalliset sisäiset komennot:

`cd false kill true wait command getopts read umask`

Näille on määritelty normaalista poikkeava suoritusjärjestys. Mahdolliset muut (toteutuskohtaiset) sisäiset komennot käyttäytyvät siinä suhteessa kuten ulkoiset.

### 4.7.5 Ulkoiset komennot

Ulkoiset komennot löytyvät erillisinä tiedostoina ja ne voi suorittaa `execv()`:llä jne.

Kaikki edellä luettelemattomat POSIXin määrittelemät komennot ovat ulkoisia, tai niiden pitää toimia kuin ne olisivat ulkoisia ohjelmia vaikka ne olisikin toteutettu shellin sisällä (kuten esim. `echo` yleensä on, samoin `test`).

## 4.8 Komentorivin tulkinnasta

### 4.8.1 Komentojen erotinmerkit

Komentoja voi yhdistellä erottamalla ne jollakin seuraavista:

`;` Puolipiste: komennot suoritetaan peräkkäin, jälkimmäinen käynnistyy edellisen loputtua (vrt. *newline*).

`&` Komennot suoritetaan yhtaikaa, &:ta edeltävä jää taustaprosessiksi. Eksplisiittisen uudelleensuuntauksen puuttuessa stdin suunnataan `/dev/null`ista (interaktiivisessa shellissä voi tapahtua muutakin). Jälkimmäinen komento voi puuttua.

`|` Putki: komennot suoritetaan yhtaikaa, edeltävän komennon tulostus ohjautuu seuraavan syötteeksi.

`&&`: Ja: komennot suoritetaan peräkkäin mutta jälkimmäinen vain jos edellinen *onnistuu* (palauttaa statuksen 0).

`||` Tai: komennot suoritetaan peräkkäin mutta jälkimmäinen vain jos edellinen *epäonnistuu* (palauttaa nollasta eroavan statuksen).

Näiden tulkintajärjestys on (1) |, (2) || ja &&, (3) ; ja &.
Huomaa erityisesti että || ja && ovat samanarvoisia:
`prog1 || prog2 && prog3`
on sama kuin
`{ prog1 || prog2 ;} && prog3`

Yhdistelmän palauttama status on viimeisen suoritetun komennon status, paitsi `&` palauttaa aina nollan.

Huom. |, || ja && vaativat kaksi komentoa, esiintyessään rivin lopussa ne jäävät odottamaan seuraavaa riviä ennen suoritusta.

Joukon komentoja voi suorittaa alishellissä kirjoittamalla ne sulkuihin:
`(`*komento...*`)`. Sulut kannattaa selvyyden vuoksi erottaa yhteydestään välilyönnein ainakin jos niitä on useita peräkkäin (joissain tilanteissa kahdella peräkkäisellä sululla on eri merkitys). Tällaista alishelliä kohdellaan esim. I/O:n uudelleensuuntauksen kannalta kuten yhtä komentoa; esim.

```
(cat file1 file2; echo LOPPU) | grep L
```

Komentoja voi yhdistellä myös aaltosuluilla {}. Tällöin ne suoritetaan samassa shellissä käynnistämättä uutta prosessia. Huom. aaltosulut *eivät* ole shellin mielestä erikoismerkkejä tässä yhteydessä vaan varattuja sanoja, niinpä ne **täytyy** erottaa yhteydestään kuten komennot. Esim.

```
{ cat file1 file2; echo LOPPU ;} | grep L
```

### 4.8.2 Komentorivin käsittelyjärjestys

Shellin pääsilmukka toimii pääpiirteissään näin: Shell

1. lukee komentorivin (yleensä tiedostosta)
2. jakaa komentorivin sanoihin ja operaattoreihin
3. jäsentää rivin joukoksi komentoja ja niiden argumentteja
4. tulkitsee tekstikorvauksia aiheuttavat erikoismerkit
5. käsittelee syötön ja tulostuksen uudelleensuuntauksen
6. etsii ja suorittaa komennon
7. mahdollisesti odottaa komennon päättymistä

Tätä prosessia tehdään rekursiivisesti siltä osin kuin erikoismerkkien tulkinta sitä edellyttää. Mm. tekstikorvauksen tulos puretaan edelleen, ellei sitä ole suojattu. Esim. vertaa seuraavia:

```
x=echo
$x kala
"$x" kala
x='echo kala'
$x
"$x"
```

Tuplalainausmerkkien sisälläkin oleva komentokorvaus ($((...)) tai `...`) aiheuttaa kuitenkin sisällä olevan tekstin purkamisen (rekursiivisesti koko yo. prosessi läpi).

Kun komento on saatu eristetyksi, sitä etsitään seuraavalla tavalla:

Jos komennossa on kauttaviiva, se suoritetaan `execve()`:llä tai vastaavalla tavalla. Jos tämä epäonnistuu, käynnistetään `sh` argumentteina ko. komentonimi argumentteineen (ts. skriptin pitäisi toimia ilman `#!`-alkuakin POSIX-yhteensopivissa järjestelmissä).

Jos siinä ei ole kauttaviivoja, yritetään järjestyksessä seuraavia:

1. Erityiset sisäiset komennot
2. Funktiot
3. Kohdassa 4.7.4 luetellut tavalliset sisäiset komennot

4. Etsitään `PATH`ista. Jos se löytyy, mutta se on toteutettu funktiona tai sisäisenä komentona tämä suoritetaan, muussa tapauksessa se suoritetaan ulkoisena ohjelmana.

(Huom. mahdolliset varatut sanat on käsitelty jo aikaisemmin.)

Niinpä shell voi tietää, että `/bin/ls` on sen sisäinen komento, mutta sen pitäisi silti löytää `/home/tt/ls` ilman polkua jos `/home/tt` on `PATH`issa ennen kuin `/bin`. Sen sijaan omatekoinen `set` tai `cd` ei löydy vaikka sellainen kuinka olisi `PATH`issa; `cd`:n voi kyllä korvata funktiolla, `set`iä ei edes sillä.

*Aliakset* tulkitaan ksh:ssa (ja useimmissa muissakin aliaksia käyttävissä shelleissä) ennen mitään sisäisiäkään komentoja, mutta kuitenkin varattujen sanojen jälkeen (jälkimmäisessä on jo eroja, esim. bash sallii aliakset niillekin).

Huom. Lähes kaikki shellit sotkevat tämän tavalla tai toisella. Erityisen yleistä on, että kaikki sisäiset komennot suoritetaan funktioiden jälkeen ennen ulkoisia komentoja `PATH`ista riippumatta. Niinpä minkään yleisen komennon nimen käyttäminen oman skriptin, funktion tai ulkoisen ohjelman nimenä ei ole turvallista ilman eksplisiittistä polkua. Mahdollisuudesta ohittaa erityiset sisäiset komennot funktioilla aiheutuu myös turvallisuusongelma; ksh:ssa se on ratkaistu tekemällä tärkeimmille komennoille automaattisesti aliakset.

### 4.9 Ohjausrakenteet

Sh tarjoaa useimmista proseduraalisista ohjelmointikielistä tutut silmukka- ja ehtokäskyt, mutta omine erikoispiirteineen. Huomaa erityisesti testien luonne ja uudelleensuuntauksen käyttötapa.

#### 4.9.1 If

`if` *komentolista*
`then` *komentolista*
`[ elif` *komentolista* `]`
`...`
`[ else` *komentolista* `]`
`fi`

Testattava ehto on mielivaltainen lista komentorivejä ja `then`-osa suoritetaan jos listan exit status on nolla (viimeinen komento onnistui).

Esimerkki:

```
if cd "$DIR" ;then
  ...
else
   echo "Hakemisto $DIR puuttuu tai on rikki" >&2
   exit 1
fi
```

Yleisin testattava komento on `test`, josta enemmän alla.

#### 4.9.2 While

`while` *komentolista*
`do` *komentolista*
`...`
`done`

Silmukan käskyjä toistetaan niin kauan kuin ehto on tosi, ts. niin kauan kuin ko. komentolistan status on nolla.

Esimerkkejä:

Tulostetaan kaksi tiedostoa rinnakkain:

```
while
  read sana1 &&
  read sana2 <&3
do
  printf "%s\t%s\n" "$sana1" "$sana2"
done <file1 3<file2
```

Etsitään `/etc/passwd`:stä käyttäjiä, joiden *uid* on nolla:

```
while IFS=: read id pw uid junk
do case "$id:$uid" in
  root:0) : ;;
  *:0)   echo "Found an extra root: $id!!" ;;
  esac
done </etc/passwd
```

Uudelleensuuntaus on ohjausrakenteissa laitettava rakenteen loppusanan jälkeen, jolloin se vaikuttaa koko rakenteeseen. (Huom. jotkin shellit suorittavat tässä tilanteessa koko rakenteen alishellinä. Jos tällöin esim. uudelleensuunnatussa silmukassa halutaan asettaa muuttujia, täytyy uudelleensuuntaus tehdä `exec`illä.)

### 4.9.3 Until

`until` *komentolista*
`do` *komentolista*
`...`
`done`

Silmukan käskyjä toistetaan niin kauan kuin komentolista palauttaa nollasta eroavan statuksen, ts. olennaisesti kuin `while !`... Harvoin käytetty, yleensä `while` on havainnollisempi.

### 4.9.4 For

For-lause eroaa syntaksiltaan ja käytöltään jo enemmän "perinteisistä" ohjelmointikielistä:

`for` *name* `[in` *lista*`]`
`do`
`...`
`done`

Silmukka toistetaan muuttujan `name` saadessa kaikki *lista*n arvot. (Jos *lista* puuttuu, oletus on `"$@"`.)

### 4.9.5 Break, continue

`While`-, `until`- ja `for`- silmukoista voi myös poistua kesken komennolla `break`. Jos silmukoita on useita sisäkkäin, `break`ille voi antaa kokonaislukuargumentin joka kertoo monestako sisimmästä hypätään ulos.

Vastaavasti `continue [`*n*`]` hyppää silmukan alkuun.

### 4.9.6 : ja !

Kaksoispiste on ns. tyhjä komento (*null command*): se ei tee mitään (mutta sen argumentit puretaan ja niiden mahdolliset sivuvaikutukset suoritetaan).

Varattu sana `!` *komento* suorittaa komennon ja palauttaa statuksen 1 jos sen sen status oli 0, muuten 1. Sitä voi käyttää etenkin `if`-, `while`- ja `until`-lauseissa ehdon merkityksen kääntämiseen.

Huutomerkki shellin toimintona on suhteellisen uusi lisäys eikä sitä kaikista shelleistä vieläkään löydy (ei edes ksh88:sta), niinpä usein näkee tällaista:

```
if prog1 ;then : ;else
  ...
fi
```

mikä on sama kuin

```
if ! prog1 ;then
 ...
fi
```

### 4.9.7 Case

Hiukan erikoisempi on `case`-lause:

`case` *merkkijono* `in`
*pattern1*) *komentolista1* `;;`
*pattern2*) *komentolista2* `;;`
`...`
`esac`

Toisin kuin `if`, `case` tekee testin itse: se vertaa annettua merkkijonoa järjestyksessä annettuihin merkkijonolausekkeisiin, löydettyään täsmäävän se suorittaa sitä seuraavat komennot kaksoispuolipisteeseen saakka ja jatkaa suoritusta `esac`in jäljestä.

Viimeisen `;;`-parin voi jättää pois, mutta mitään tapaa jatkaa suoritusta seuraavan vaihtoehdon komennoista ei ole. (Ksh tarjoaa tähän dokumentoimattoman (?) keinon: `;;`:n paikalle `;&`. Sen käyttöä lienee kuitenkin syytä välttää.)

Huomaa erikoinen parittomien sulkujen käyttö. POSIX edellyttää kyllä, että niille saa kirjoittaa parinkin lausekkeen vasemmalle puolelle, mutta yleensä sitä ei tehdä.

Merkkijonolausekkeissa käytettävä syntaksi on sama kuin tiedostonimilaajennuksissa edellä (mutta ilman kauttaviivaa ja pistettä koskevia rajoituksia).

Esimerkki: Olkoon tiedostossa `koneet` koneiden nimiä ja tyyppejä tähän tapaan:

```
tarzan HP-UX 11.0
jane HP-UX 10.20
josef Solaris 5.7
korak HP-UX 10.20
volsung Linux RedHat 7.0
```

Halutaan tutkia koneiden tyypit suorittamalla niissä `ssh`:lla (*Secure Shell*; oletetaan että se on konfiguroitu niin että 'ssh kone komento' toimii ko. koneille ilman salasanaa) eri käyttöjärjestelmissä sopiva komento:

```
while read hostname type junk
do
  case "$type" in
    HP*)   ssh $hostname model ;;
    Solaris|Sun*) ssh $hostname uname -p ;;
    Linux) ssh $hostname uname -m ;;
  esac </dev/null
done <koneet
```

### 4.9.8 test, [

Komentoa `test` käytetään yleensä `if`-, `while`- ja `until`-rakenteissa (sekä `||`:n ja `&&`:n kanssa) ehtotestinä. Sillä on kaksi syntaksia, riippuen siitä kutsutaanko sitä nimellä `test` vain `[`:

`test` *ehtolauseke*

`[` *ehtolauseke* `]`

Vaikutus on sama, ainoa ero on että `[` vaatii loppuun `]`:n, ja jos sitä halutaan käyttää ulkoisena tarvitaan `test` (esim. `find ... -exec test ...`). Huomaa että hakasulut on erotettava lausekkeesta tyhjällä.

Ehtolausekkeessa voi olla seuraavanlaisia testejä:

`-b` *tiedosto* : tosi jos *tiedosto* ja blokkilaitetiedosto (*block special file*)

`-c` *tiedosto* : tosi jos *tiedosto* ja merkkilaitetiedosto (*character special file*)

`-d` *tiedosto* : tosi jos *tiedosto* on hakemisto

`-e` *tiedosto* : tosi jos *tiedosto* on olemassa

`-f` *tiedosto* : tosi jos *tiedosto* on normaali tiedosto (BSD: on olemassa mutta ei ole hakemisto)

`-g` *tiedosto* : tosi jos *tiedosto*n *setgid*-bitti on päällä

`-n` *merkkijono* : tosi jos *merkkijono* on epätyhjä

`-p` *tiedosto* : tosi jos *tiedosto* on nimetty putki (*FIFO*)

`-r` *tiedosto* : tosi jos *tiedosto* on luettavissa

`-s` *tiedosto* : tosi jos *tiedosto*n koko on suurempi kuin nolla

`-t` *tiedostokahva* : tosi jos  on auki ja päätelaite

`-u` *tiedosto* : tosi jos *tiedosto*n *setuid*-bitti on päällä

`-w` *tiedosto* : tosi jos *tiedosto* on kirjoitettavissa (w-bitti päällä – read-only tiedostojärjestelmässä tämä voi olla tosi vaikkei tiedostoon voikaan kirjoittaa)

`-x` *tiedosto* : tosi jos *tiedosto* on suoritettavissa (x-bitti päällä, myös hakemistoille)

`-z` *merkkijono* : tosi jos *merkkijono* on tyhjä

*merkkijono* : tosi jos *merkkijono* ei ole tyhjä (vaarallinen jos merkkijono voi alkaa `-` :lla!)

*s1* = *s2* : tosi jos merkkijonot *s1* ja *s2* ovat identtisiä

*s1* != *s2* : tosi jos merkkijonot *s1* ja *s2* ovat erilaisia

*n1* `-eq` *n2* : tosi jos kokonaisluvut *n1* ja *n2* ovat yhtäsuuria

*n1* `-ne` *n2* : tosi jos kokonaisluvut *n1* ja *n2* ovat erisuuria

*n1* `-gt` *n2* : tosi jos kokonaisluku *n1* on suurempi kuin *n2*

*n1* `-ge` *n2* : tosi jos kokonaisluku *n1* on suurempi tai yhtäsuuri kuin *n2*

*n1* `-lt` *n2* : tosi jos kokonaisluku *n1* on pienempi kuin *n2*

*n1* `-le` *n2* : tosi jos kokonaisluku *n1* on pienempi tai yhtäsuuri kuin *n2*

Kaikki tiedostotestit palauttavat epätoden jos tiedostoa ei ole olemassa.

Kaikkien edellä lueteltujen merkityksen voi kääntää negaatiokseen lisäämällä eteen huutomerkin (vrt. huutomerkkiin sh:n varattuna sanana).

POSIX.2 ei edellytä mitään muuta, erityisesti ei sulkuja eikä muutakaan testien yhdistelytapaa. Sekä sulut että seuraavat toimivat kuitenkin sekä BSD:ssä että SysV:ssä (ja jokseenkin kaikkialla) mutta joissakin tilanteissa yhteensopimattomilla tavoilla:

*expr1* `-a` *expr2* : tosi jos molemmat lausekkeet ovat tosia

*expr1* `-o` *expr2* : tosi jos jompikumpi lauseke on tosi

Näiden prioriteetti on alhaisempi kuin edellämainittujen (myös `!`:n) ja `-o`:n alhaisempi kuin `-a`:n. Järjestystä voi muuttaa suluilla normaaliin tapaan (ne ovat sh:n erikoismerkkejä ja suojattava).

Olennainen siirrettävyysongelma näiden kanssa on se, että BSD:ssä yksiargumenttisilla optioilla on korkeampi prioriteetti kuin merkkijonovertailuilla, sysV:ssä päinvastoin. Merkkijonot voivat muutenkin aiheuttaa yllätyksiä monimutkaisissa lausekkeissa, jos ne saattavat alkaa erikoismerkeillä `- ! = ( )` (sysV:ssä jopa `test -n "$x"` aiheuttaa virheen jos `$x` on `'='`). Tämän takia merkkijonojen vertailukin usein tehdään tyyliin `test X"$a" = X"$b"`, vaikka POSIX-säännöillä moista ei tarvitakaan. Sen sijaan `test "$x"` voi aiheuttaa virheen POSIX-yhteensopivassakin shellissä (jos `$x` on esim. `"-f"`).

Siirrettävyysongelmien takia `-a`:ta ja `-o`:ta on syytä välttää vähänkään monimutkaisemmissa lausekkeissa. Esim. `test` *expr1* `-o` *expr2* on korvattavissa siirrettävällä muodolla `test` *expr1* `||` `test` *expr2* (`-a` samaan tapaan `&&`:lla). Huomaa kuitenkin että `test`-komennossa `-a`:lla on korkeampi prioriteetti kuin `-o`:lla mutta shellin `||` ja `&&` ovat samanarvoisia.

### 4.9.9 Ksh: [[...]]

Korn shellissä on `test`in lisäksi sisäänrakennettuna testisyntaksi `[[` *ehtolauseke* `]]` . Hakasulkuparit ovat varattuja sanoja ja vaativat ympärilleen välilyönnit tms. Ehtolauseke on samanlainen kuin `test`-komennossa, `-b`, `-c`, `-d`, `-f`, `-g`, `-n`, `-p`, `-r`, `-s`, `-t`, `-u`, `-w`, `-x`, `-z` `-eq`, `-ne`, `-lt`, `-gt`, `-le` ja `-ge` toimivat samalla tavoin, ja sen lisäksi:

`-a` *tiedosto* : tosi jos tiedosto on olemassa (sama kuin `-e` – vanhemmissa ksh:ssa `-e` puuttuu)

`-h` *tiedosto* : tosi jos tiedosto on symbolinen linkki

`-k` *tiedosto* : tosi jos tiedoston *sticky bit* on päällä

`-o` *optio* : tosi jos optio on päällä

`-L` *tiedosto* : tosi jos tiedosto on symbolinen linkki (sama kuin `-h`)

`-O` *tiedosto* : tosi jos tiedosto on oma (omistaja shell-prosessin effective UID)

`-G` *tiedosto* : tosi jos tiedosto on omassa ryhmässä (ryhmä shell-prosessin effective GID)

`-S` *tiedosto* : tosi jos tiedosto on *socket*

*tiedosto1* -nt *tiedosto2* : tosi jos *tiedosto1* on uudempi kuin *tiedosto2*

*tiedosto1* -ot *tiedosto2* : tosi jos *tiedosto1* on vanhempi kuin *tiedosto2*

*tiedosto1* -ef *tiedosto2* : tosi jos *tiedosto1* ja *tiedosto2* ovat sama tiedosto

*string* = *pattern* : tosi jos merkkijono *string* vastaa mallia *pattern*

*string* ! = *pattern* : tosi jos merkkijono *string* ei vastaa mallia *pattern*

*s1* < *s2* : tosi jos merkkijono *s1* on pienempi kuin *s2* (aakkosjärjestyksessä)

*s1* > *s2* : tosi jos merkkijono *s1* on suurempi kuin *s2*

Lisäksi ehtoja voi yhdistellä seuraavilla, prioriteettijärjestyksessä:

1. sulut `()`
2. `!` (negaatio)
3. `&&` JA
4. || TAI

Tuplahakasulkujen sisällä ei levitetä jokerimerkkejä eikä erotella sanoja IFS:llä, ja ehtojen yhdistelysäännöt on määritelty täsmällisesti joten niitä voi käyttää siirrettävästi (shelleissä joissa `[[ ]]` on). Huomaa myös merkkijonovertailun ero `test`iin.

#### 4.9.10 exit

`exit` *n*

lopettaa shellin suorituksen, palauttaen statuksena argumenttinsa (tai sen puuttuessa nollan).

### 4.9.11 exec

`exec` *komento* [*argumentteja...*]

suorittaa nimetyn komennon niin, että se korvaa käynnissä olevan shellin luomatta uutta prosessia (vrt. *execv(3)*). (Huom. se ei siitä huolimatta peri shellin sisäisiä muuttujia, optioita jne.)

Komentorivillä mahdollisesti annetut syötön tai tulostuksen uudelleensuuntaukset tehdään ensin. Jos `exec` suoritetaan ilman komentoa, se ei mitään muuta teekään kun suuntaa shellin I/O:n uudelleen.

Esimerkki: Alkuperäisessä Bourne sh:ssa ohjausrakenteen I/O:n uudelleensuuntaus aiheutti sen suorittamisen alishellissä, ja jotkin sh:t tekevät niin yhä. Niinpä haluttaessa asettaa muuttuja uudelleensuunnatussa silmukassa tarvittiin tällaista:

```
exec 3<&0
exec <file
while read key value comment ;do
  case "$key" in
    SECRET)  x=$value ;;
  esac
done
exec 0<&3
```

### 4.9.12 eval

Komento `eval` suorittaa merkkijonon sisältämän komennon. Esim.

```
x='y=5'
eval $x
echo $y
```

`Eval` edellyttää erityistä huolellisuutta erikoismerkkien suojauksen kanssa, ne kun tulevat käsitellyksi kahteen kertaan.

Esim. simuloidaan taulukkoa:

```
tau() {
# tau taulukko indeksi [value]
# asettaa tai palauttaa taulukon alkion
  case $# in
     2)  eval 'printf "%s\n" "$'"$1_$2"\" ;;
```

```
    3)  eval "$1_$2=\"\$3\"" ;;
 # eval "$1_$2='$3'" ei toimi - miksei?
    *)  printf "usage: tau arrayname index [value]\n"
 return 1 ;;
  esac
}
```

Esim. kuvitellaan esihistoriallinen shell jossa ei ole `eval`-komentoa. Sitä voisi silloin matkia tähän tapaan:

```
myeval() {
  printf "%s\n" "$*" > /tmp/.myeval_temp_file_$$
  . /tmp/.myeval_temp_file_$$
  rm /tmp/.myeval_temp_file_$$
}
```

## 4.10 Aritmetiikka

### 4.10.1 Expr

`expr` *lauseke*

laskee arvon lausekkeelle, joka koostuu erillisinä argumentteina annetuista kokonaisluvuista ja merkkijonoista ja seuraavista operaatioista:

`( ) | & = > >= < <= != + - * / % :`

Huomaa että monet noista ovat shellin erikoismerkkejä jotka on suojattava.

Laskutoimitukset `+ - * / %` on määritelty kuten C:ssä mutta vain kokonaisluvuille.

Vertailut `= > >= < <= !=` toimivat sekä kokonaislukujen että merkkijonojen kanssa (jälkimmäisillä aakkosjärjestysvertailu, kieliriippuvainen): tulos on 0 (epätosi) tai 1 (tosi).

Erikoisempia ovat seuraavat:

- `|` TAI: *expr1* `|`ă*expr2* palauttaa ensimmäisen lausekkeen jos se ei ole nolla eikä tyhjä, muuten jälkimmäisen.
- `&` JA: *expr1* `&`ă*expr2* palauttaa ensimmäisen lausekkeen jos kumpikaan lausekkeista ei ole nolla eikä tyhjä, muuten nollan.
- `:` merkkijonovertailu: *string : regexp* vertaa merkkijonoa malliin (merkkijonolauseke, syntaksi kuten `grep`issä mutta implisiittisesti ankkuroitu alusta (mutta ei

lopusta), ^ lausekkeen alussa ei ole tarpeen (se saattaa toimia tai sitten ei, POSIX.2 jättää asian auki). Tulos on nolla jos vastinetta ei löydy, muuten täsmäävien merkkien lukumäärä tai jos mallissa on ryhmittelysulkeita `\(...\)`, ensimmäisen sisältöä vastaava merkkijono.

Sulkuja voi käyttää ryhmittelyyn tavalliseen tapaan, muuten prioriteettijärjestys on seuraava:

1. :
2. * / %
3. + -
4. = > >= < <= !=
5. &
6. |

Exit status on 0 jos tulos oli jotain muuta kuin nolla tai tyhjä, mikä helpottaa `expr`in käyttöä testeissä.

Argumenttien erillisyysvaatimus, useiden shellin erikoismerkkien suojaustarve sekä merkkijonojen sekoittuminen operaattoreihin tekee `expr`istä hankalan käyttää, eikä sitä nykyisin juurikaan tarvita laskutoimituksiin ellei haluta yhteensopivuutta vanhojen shellien kanssa. Ainoastaan sen merkkijonovertailuoperaattori on selvästi voimakkaampi kuin shellin muuttujankäsittelytoiminnot ja usein kätevämpi kuin esim. `sed`.

### 4.10.2 $((...))

POSIX shellit osaavat sisäisesti laskea aritmeettisia lausekkeita tällaisella syntaksilla:

`$$((`*lauseke* `))`

Lauseke puretaan samassa vaiheessa kuin muutkin `$`-alkuiset (muuttujat jne), ja sen sisällä olevat erikoismerkit suojataan kuten tuplalainausmerkkien sisällä paitsi että " itse ei ole siellä erikoismerkki. Lausekkeen sisällä muuttujien arvoihin viitataan ilman dollarimerkkiä. Sallittuja laskutoimituksia ovat C:n tuntemat kokonaislukuoperaatiot (poislukien `++`, `--` ja `sizeof`), samoilla prioriteettisäännöillä:

1. `+` ja `-` *etumerkkeinä*, ! (negaatio)
2. `*`, `/` : kerto- ja jakolasku

3. `+` ja `-` *laskutoimituksina*
4. `«`, `»` : bittisiirrot
5. `<`, `<=`, `>`, `>=` : erisuuruusvertailut
6. `=`, `!=` : yhtäsuuruusvertailut
7. `&` : bitti-JA
8. `^` : bitti-XOR
9. `|` : bitti-TAI
10. `&&` : looginen JA
11. `||` : looginen TAI
12. *expr* `?` *expr* `:` *expr* : ehtolauseke
13. `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `«=`, `»=`, `&=`, `^=`, `|=` : sijoitukset

Huom. Kyseessä ei ole komento, haluttaessa laskea lauseke käyttämättä sen tulosta täytyy käyttää apuna esim. *null*-komentoa (kaksoispiste), seuraavat tekevät saman asian:

```
: $((a+=2))
a=$((a+2))
```

Sitä ei myöskään suoraan voi käyttää ehtotestissä (`case`:ssa kylläkin), vaan tarvitaan jotain tällaista:

```
if [ $((x>y)) -ne 0 ]
then ...
```

### 4.10.3 Ksh: ((. . . ))

Ksh tuntee myös komentomuotoisen aritmetiikan kahdella syntaksilla:

`((`*lauseke*`))`
`let` *lauseke* . . .

Erona noilla on että edellinen suojaa sulkujen sisällön automaattisesti samoin kuin tuplalainausmerkit (ts. `((...))` on sama kuin `let "..."`), ja `let`ille voi antaa monta lauseketta.

Kummassakin tapauksessa exit status on nolla jos (viimeisen) lausekkeen arvo on nollasta eroava, muussa tapauksessa yksi. Tätä voi siis käyttää suoraan esim. `if`-lauseessa.

Ksh tulkitsee tuplasulut operaattorimerkeiksi joten niitä ei tarvitse erottaa välilyönnein (mistä toisaalta seuraa tarve erottaa kaksi peräkkäistä alishelliä merkitsevää sulkua toisistaan).

## 4.11 Shellin optiot ja set

Kohdassa 4.4.7 esiteltiin `set`-komennon käyttö positioparametrien muuttamiseen. Sillä voidaan myös tulostaa kaikki shellin muuttujat (ilman mitään optioita ja argumentteja), sekä muuttaa shellin optioita, jotka on lueteltu alla.

Kaikissa seuraavissa voi alkuviivan korvata plus-merkillä, jolloin merkitys kääntyy (kytkee option pois päältä).

- `-a` Muuttujasijoituksista seuraa automaattinen `export`.
- `-C` Estää tulostuksen uudelleensuuntauksen olemassaolevan tiedoston päälle (*noclobber*); vrt. 4.5.1.
- `-e` Virhe yksinkertaisesta komennosta (jota ei oteta kiinni ohjausrakenteella) aiheuttaa shellistä poistumisen (`exit`in).
- `-f` Estää tiedostonimien laajennuksen.
- `-n` Shell lukee komennot muttei suorita niitä (syntaksitarkistus).
- `-u` Yritys käyttää muuttujaa jota ei ole aiheuttaa virheen. Ilman tätä (`set +u`) sellaisesta tulee tyhjä.
- `-v` Shell tulostaa stdininsä stderriin sitä lukiessaan.
- `-x` Komennot tulostetaan stderriin ennen suoritusta.

Kaikki nämä voi myös antaa shellin käynnistyksessä optioina.

Esimerkki: Lukkotiedoston käyttö edellyttää atomaarista tiedoston olemassaolon testausta ja luontia. Se voidaan tehdä `-C`-option avulla tähän tapaan:

```
LOCK=Lukko
reset="set +C"
case $- in *C*) reset="" ;; esac
set -C
until 2>/dev/null >$LOCK
do sleep 10
printf "."
done
$reset
echo "got it!"
```

### 4.12 Trap

`trap` [*toiminta ehto* ...]

määrää miten shell reagoi signaaleihin (vrt. `kill`). Jos *toiminta* on `-`, palautetaan oletusasetus, jos se on tyhjä merkkijono, ko. signaaliin ei reagoida. Muussa tapauksessa sen pitää olla shellin ymmärtämä komentorivi (kuin jos tehtäisiin `eval "\$toiminta"`) ja se suoritetaan samassa ympäristössä kuin signaalia edeltänyt komento, mutta `$?`:n arvo palautetaan sen jälkeen ennalleen.

*Ehto* voi olla `EXIT`, `0` (sama kuin `EXIT`), signaalinimi (ilman `SIG`-etuliitettä) tai numero (lailliset nimet ja numerot kuten `kill`-komennolla).

Esim. tulostetaan ”heippa” shellin päättyessä (normaalisti tai ei):
`trap "echo heippa" EXIT`

Ilman argumentteja `trap` tulostaa voimassa olevat asetuksensa sellaisessa muodossa että sen voi suorittaa `eval`-komennolla asetuksen palauttamiseksi:

```
oldtraps=$(trap)
...
eval "$oldtraps"
```

# 5 Standardikomennot

Seuraavassa esitellään kaikki POSIX.2:n edellyttämät komennot paitsi erityisiä sisäisiä komentoja (4.7.3). Osa on esitelty jo aikaisemmin ja osa esitellään perusteellisemmin myöhemmin ja sivuutetaan nyt maininnalla, mutta täydellisyyden vuoksi ne kuitenkin luetellaan tässä. Vain ohjelmankehitysympäristössä vaadittavia komentoja `ar`, `make` ja `strip` ei esitellä tässä.

## 5.1 awk - Pattern scanning and processing language

Awk on täydellinen ohjelmointikieli ja ansaitsee oman lukunsa (ks. 8).

## 5.2 basename - Return nondirectory portion of pathname

`basename` *string* [*suffix*]

tulkitsee argumenttinsa tiedostonimeksi, jossa voi olla polku ja/tai pisteellä erotettu tarkennin, ja palauttaa sen ilman polkua, ja ilman tarkenninta jos tarkennin on annettu argumenttina ja se vastaa nimen lopussa olevaa. Jos tarkenninargumentti ei vastaa nimeä, sillä ei ole vaikutusta (ei myöskään virheilmoitusta).

Jos argumentin lopussa on /, se poistetaan ensin ja palautetaan sen vasemmalla puolella oleva osa. Tulokseen voi jäädä / vain jos argumentissa on vain niitä.

## 5.3 bc - Arbitrary-precision arithmetic language

`bc [-l]`

Yleinen laskinohjelma. Ymmärtää suurin piirtein samoja liukulukulausekkeita kuin C (ei kaikkia kirjastofunktioita), mutta laskee mielivaltaisella tarkkuudella. Tuntee myös muuttujat, ohjausrakenteet `for`, `while` ja `if`, funktiomäärittelyt (omalla syntaksillaan) jne.

Optio `-l` asettaa näyttötarkkuudeksi alussa 20 numeroa (ilman sitä oletus on 0; asetettavissa muuten muuttujalla `scale`) ja määrittelee joitakin matemaattisia funktioita.

(Historiallisesti aikaisempi laskinohjelma `dc`, jolla `bc` usein on toteutettu, ei ole POSIX-standardissa.)

## 5.4 cat - Concatenate and print files

`cat [-u]` [*file(s)*]

Tulostaa argumenttina annetut tiedostot peräjälkeen (tai tiedostoargumenttien puutteessa stdinin) stdout'iin. Optiolla `-u` tulostus tapahtuu merkki kerrallaan ilman blokkausta (tarpeen lähinnä luettaessa laitetiedostoa).

Epästandardeja mutta yleisiä optioita ovat

-b ei tulosta tyhjiä rivejä

-e tulostaa jokaisen rivin loppuun $

-n tulostaa rivinumerot

-r tulostaa monesta peräkkäisestä tyhjästä rivistä vain yhden

-s estää virheilmoitukset

-t tulostaa tabulaattorin muodossa `^I`

-v (*verbose*): tulostaa kontrollimerkit näkyvässä muodossa (ei tabulaattoria)

Siirrettävissä skripteissä ei näitä optioita voi käyttää (samat asiat voi tehdä mm. `sed`illä), interaktiivisesti etenkin `cat -et` on usein kätevä.

## 5.5 cd - Change working directory

Vaihtaa työhakemiston annetuksi tai ilman argumenttia kotihakemistoksi (`$HOME`).

Jos annettu hakemisto ei ala /:lla, sitä etsitään muuttujan `CDPATH` sisältämistä hakemistoista (kaksoispisteellä eroteltu lista) tai jos se puuttuu, oletushakemistosta.

Ksh asettaa lisäksi `$PWD`:hen valitun hakemiston ja `$OLDPWD`:hen edellisen hakemiston, vaihtaa takaisin edelliseen jos argumenttina on pelkkä `-`, ja vaihtaa osan nykyisestä oletushakemistosta syntaksilla `cd` *old new*.

## 5.6 chgrp - Change file group ownership

`chgrp [-R]` *group file(s)*

Vaihtaa tiedostojen ryhmän, optiolla `-R` rekursiivisesti hakemistopuulle.

Yleensä sallittu vain superkäyttäjälle tai erikseen määritellyille käyttäjäryhmille.

### 5.7 chmod - Change file modes

`chmod [-R]` *mode file(2)*

Muuttaa tiedostojen suojausbittejä. *Mode* voi olla joko oktaaliluku tai symbolinen esitys muotoa

`[augo][+-=][rwxXsugo][,...]`

eli lista määrityksiä joissa ensimmäinen kirjainyhdistelmä kertoo kenen oikeuksia muutetaan, a=all, u=user, g=group, o=others (oletus `umask`in mukainen), toinen miten ('=' asettaa, '+' lisää, '-' poistaa) ja kolmas mitä oikeutta muutetaan, r=read, w=write, x=execute/search, s=setuid/setgid, X=execute-if-relevant (sama kuin x jos kyseessä on hakemisto tai tiedosto jolla on jo ainakin yksi x-bitti päällä).

Oikeutena u, g ja o tarkoittavat vastaavaa oikeutta ko. joukolla (esim. `chmod g=u` asettaa ryhmälle samat oikeudet kuin käyttäjällä).

Huom. jos `mode`-argumentti alkaa `-`:lla, sen eteen tarvitaan `--` ettei sitä tulkittaisi optioksi. (Käytännössä lähes kaikki toteutukset toimivat kyllä ilmankin.)

Oktaalimuotoinen `mode` asettaa kaikki oikeudet kerralla, oktaaliluvun bitit vastaavat seuraavia oikeuksia: 01: u=x, 02: u=w, 04: u=r, 010: g=x, 020: g=w, 04: g=r, 0100: o=x, 0200: o=w, 0400: o=r, 02000: g=s, 04000: u=s.

Huom. 01000 on standardissa jätetty auki, mutta yleisesti se on ns. *sticky bit*, ja sen voi asettaa symbolisessakin muodossa kirjaimena t.

Optio `-R` tekee muutoksen rekursiivisesti hakemistopuulle.

Joissakin järjestelmissä on lisäksi optio `-A`, joka säilyttää mahdolliset ACL:t (*access control list*), oletuksena ne poistetaan.

Jos tiedosto johon viitataan on symbolinen linkki, muutokset tehdään sen osoittamalle tiedostolle, ei linkille.

### 5.8 chown - Change file ownership

`chown [-R]` *owner*`[:`*group*`]` *file(s)*

Vaihtaa tiedostojen omistajaa (ja mahdollisesti ryhmää), optiolla `-R` rekursiivisesti koko hakemistopuulle.

Yleensä sallittu vain superkäyttäjälle tai erikseen määritellyille käyttäjäryhmille.

### 5.9 cksum - Write file checksums and sizes

Laskee tiedostoargumenteilleen tarkistussumman (ISO 8802-3:n määrittelemä CRC).

Historiallisesti samaan tarkoitukseen käytetty `sum` jätettiin pois standardista eikä sitä ole syytä käyttää mihinkään.

### 5.10 cmp - Compare two files

Vertaa kahta tiedostoa ja jos ne eroavat, tulostaa ensimmäisen eron paikan.

Optiolla `-l` tulostaa kaikki erot, optiolla `-s` ei mitään.

Exit status on nolla jos tiedostot ovat identtisiä, muutoin yksi.

### 5.11 comm - Select or reject lines common to two files

`comm [-123]` *file1 file2*

Valikoi yhteisiä rivejä kahdesta tiedostosta, joiden oletetaan olevan lajiteltuna, ensin vain tiedostossa *file1* olevat rivit, sitten vain tiedostossa *file2* olevat ja lopuksi yhteiset. Optiolla `-1` jätetään vain ensimmäisessä olevat pois, `-2`:lla vain toisessa olevat ja `-3`:lla yhteiset.

Stdiniä käytetään argumenttina vain jos tiedostonimeksi annetaan `-`.

### 5.12 command - Execute a simple command

`command [-p]` *command_name...*

Suorittaa nimetyn komennon, ohittaen mahdollisen samannimisen funktion. Optiolla `-p` ohittaa myös `PATH`in ja etsii komentoa `PATH`in sen oletusarvon mukaisesti.

### 5.13 cp - Copy files

`cp [-R|-r] [-fip]` *tiedosto(t) kohde*

kopioi tiedostoja, optiolla `-r` tai `-R` myös hakemistopuita rekursiivisesti. Optio `-R` kopioi myös nimettyjä putkia yms erikoistiedostoja; `-r`:n käytös niiden kanssa on määrittelemätön, joten kannattanee aina käyttää `-R`:ää.

Jos kopioitavia tiedostoja on useita, kohteen pitää olla hakemisto, joka on valmiiksi olemassa.

Hakemistoa kopioitaessa luodaan aina uusi samanniminen hakemisto (vaikka kohde olisikin valmiiksi sellainen).

Jos kopioitava tiedosto on sama kuin kohde tai `.` tai `..`, sille ei tehdä mitään.

Uuden tiedoston suojausbitit ovat samat kuin alkuperäisen *XOR* `umask`.

Jos kohdetiedosto on jo olemassa, uusi kirjoitetaan oletuksena sen päälle, paitsi jos optio `-i` on annettu, jolloin asiaa kysytään käyttäjältä (viesti stderriin, vastaus stdinistä). Jos se on suojattu tapahtuu virhe, paitsi optiolla `-f` se yritetään ensin hävittää (`unlink()`-kutsulla — ei onnistu hakemistolle).

Jos `-p`-optio on annettu, tiedoston aikaleimat (mtime ja atime muttei ctime), suojausbitit, omistaja ja ryhmä säilytetään (viimeiset kaksi toimivat vain superkäyttäjälle tai yleensä niille joille myös `chown`, ja jos omistaja tai ryhmä muuttuu myös suid- ja setgid-bitit nollataan).

## 5.14 cut - Cut out selected fields of each line of a file

`cut [-b]` *list* `[-n] [`*file(s)*`]`
`cut [-c]` *list* `[`*file(s)*`]`
`cut [-f]` *list* `[-d` *delim*`] [-s] [`*file(s)*`]`

valikoi tiedostoista tai stdinistä kenttiä kolmella eri tavalla:

- `-b` valitsee tavuja; valitun tavun puuttuminen on virhe. Optiolla `-n` ei katkaise monitavuisia merkkejä keskeltä.
- `-c` valitsee merkkejä (jotka voivat olla monitavuisia joissakin kielissä); valitun puuttuminen ei ole virhe.
- `-f` valitsee erotinmerkin (oletus tab, vaihdettavissa optiolla `-d`) rajaamia kenttiä. Rivit joilla ei ole yhtään erotinmerkkiä tulostetaan sellaisenaan, paitsi jos `-s` on annettu jolloin niitä ei tulosteta lainkaan.

Kaikissa valintalista on pilkuilla ja tavuviivoilla eroteltu kokonaislukulista: $n - m$ tarkoittaa positioita $n$:stä $m$:ään, $n-$ $n$:stä eteenpäin ja $-m$ on sama kuin $1 - m$. Tällaisia elementtejä voi olla mielivaltainen määrä, ne voivat mennä päällekkäin ja olla missä järjestyksessä tahansa. Valittuja kenttiä ei kuitenkaan tulosteta moneen kertaan tai eri järjestyksessä.

## 5.15 date - Write the date and time

`date [-u] [+`*format*`]`

Tulostaa päiväyksen ja kellonajan halutussa formaatissa. Optiolla `-u` käyttää aikavyöhykkeenä UTC0:aa (GMT) riippumatta ympäristömuuttujan `TZ` arvosta.

Jos muotoiluformaatti on annettu, siitä korvataan prosenttialkuiset merkkiparit seuraavan listan mukaisesti.

`%a` Lyhennetty viikonpäivä[1]

`%A` Täydellinen viikonpäivä[1]

`%b` Lyhennetty kuukausi[1]

`%B` Täydellinen kuukausi[1]

`%c` Päivämäärä ja aika oletusformaatissa[1,2]

`%C` Vuosisata numerona 00-99[2]

`%d` Kuukaudenpäivä numerona 01-31[3]

`%D` Päivämäärä muodossa mm/dd/yy[4]

`%e` Päivämäärä numerona 1-31 (yksinumeroisissa alkutyhjä)[3]

`%h` Sama kuin %b[4]

`%H` Tunti 24h kellon mukaan 00-23[3]

`%I` Tunti 12h kellon mukaan 01-12[3]

`%j` Vuoden päivänumero (001-366)

`%m` Kuukausi desimaalilukuna 01-12[3]

`%M` Minuutti desimaalilukuna 00-59[3]

`%n` Rivinvaihto (newline)

`%p` Aamu/iltapäiväsymboli (AM/PM)[1]

`%r` Kellonaika 12h-muodossa[1,4]

`%S` Sekunnit desimaalilukuna 00-61[3]

`%t` Tabulaattori

`%T` Kellonaika 24h-muodossa HH:MM:SS[4]

`%U` Viikkonumero (sunnuntaialkuinen viikko) 00-53[3]

`%w` Viikonpäivä numerona 0-6 (0=su))[3]

`%W` Viikkonumero (maanantaialkuinen viikko) 00-53[3]

`%x` Päivämäärä oletusformaatissa[1,2]

`%X` Kellonaika oletusformaatissa[1]

`%y` Vuosi kaksinumeroisena[2 3]

`%Y` Vuosi nelinumeroisena[2]

`%Z` Aikavyöhyke

`%%` Prosenttimerkki

[1] Riippuu kielestä (*locale*)
[2] `E` prosentin ja formaattikirjaimen välissä valitsee mahdollisen vaihtoehtoisen vuosiluvun
[3] `O` prosentin ja formaattikirjaimen välissä valitsee mahdolliset vaihtoehtoiset numerosymbolit
[4] Vanhentunut, pitäisi välttää

Esim. Saudi-Arabiassa `+%OH` voisi tuottaa kellonajan sikäläisillä numerosymboleilla (jotka eroavat meikäläisistä ns. arabialaisista numeroista!), kun taas `+%H`:n pitäisi aina käyttää länsimaisia numeroita. Sen sijaan tarkennin `E` on tarkoitettu vain kalentereille jotka poikkeavat gregoriaanisesta vain alkamisvuoden mukaan, kuten Thaimaassa ja Japanissa, ja laajennus kalenterihin joissa vuosi vaihtuukin eri aikaan on jätetty auki ("implementation-specific extension"). Muutkin päivämääräkentät on yleensä määritelty vain gregoriaanista kalenteria silmällä pitäen, vain `%x` ja `%c` on tarkoitettu yleisemmiksi.

Tarkentimet E ja O eivät aiheuta virhettä vaan jätetään huomiotta jos ko. vaihtoehtoista esitystä ei ole.

Moderneissa järjestelmissä näkee lisäksi mm. seuraavia:

`%V` Viikkonumero ISO-standardin mukaan

`%E` Aikakauden/keisarin nimi ja vuosiluku

`%N` Aikakauden/keisarin nimi

`%o` Vuosi japanilaisen kalenterin mukaan

`%u` Viikonpäivä numerona 1-7 (ma=1)

Huom. `%U` ja `%W` laskevat viikkonumeron väärin (suomalaisittain ja ISO-standardin mukaan)! Jos järjestelmä ei tunne `%V`:tä, sitä ei saa oikein mitenkään (muuten kuin erikseen laskemalla).

Joskus näkee myös `%F`:n, joka on sama kuin `%B` (paitsi ettei se ehkä välitä kieliasetuksesta).

Joissakin järjestelmissä voi lisäksi prosenttimerkin ja seuraavan kirjaimen väliin lisätä kentän pituus- ja tarkkuusmäärittelyn kuten `printf`issä.

Esim. Halutaan viikon- ja kuukaudenpäivä, kuukausi ja vuosi eri muuttujiin:

```
eval `date '+MONTH=%b DOW=%a DAY=%d YEAR=%Y'`
```

POSIX.2 ei tunne `date`-komennon toista yleistä käyttötapaa, joka kuitenkin toimii lähes kaikissa koneissa:

`date [-u]` *mmddhhmm*`[[`*cc*`]`*yy*`]`

Tällä asetetaan systeemikello. Yleensä sallittu vain superkäyttäjälle. (Huom. formaatti vaihtelee eri koneissa, tarkista ensin!)

## 5.16 dd - Convert and copy a file

`dd` on historiallisista syistä syntaksiltaan sangen erikoinen. Se tuntee seuraavanlaiset argumentit:

`if=`*file* syöttötiedosto, oletus stdin

`of=`*file* tulostustiedosto, oletus stdout; jos `seek=`-määritystä ei ole, tiedosto nollataan aluksi

`ibs=`*n* blokkikoko luettaessa, tavuja, oletus 512

`obs=`*n* blokkikoko kirjoitettaessa, tavuja, oletus 512

`bs=`*n* blokkikoko sekä luettaessa että kirjoitettaessa

`cbs=`*n* blokkikoko optioita `block` ja `unblock` varten

`skip=`*n* ohita *n* tavua syöttötiedoston alusta

`seek=`*n* ohita *n* tavua tulostustiedoston alusta

`count=`*n* kopioi vain *n* syöttötiedoston blokkia

`conv=`*optio*`[,...]` , missä *optiot* voivat olla seuraavia:

> `block` lue vaihtelevanmittaisia rivinvaihtoon (\n) loppuvia tietueita ja muunna ne `cbs`:n mittaisiksi kiinteiksi (poistaen rivinvaihdot)
>
> `unblock` tulosta vaihtelevanmittaisia rivinvaihdolla rajattuja tietueita

`lcase` muuta isot kirjaimet pieniksi

`ucase` muuta pienet kirjaimet isoiksi

`swab` vaihda aina kaksi peräkkäistä tavua keskenään

`noerror` älä keskeytä virheeseen

`notrunc` älä nollaa tiedostoa aluksi

`sync` täydennä syöttöblokit `ibs=`:n mittaisiksi nullilla

Yleisiä epästandardeja `conv`-optioita ovat lisäksi `EBCDIC` (muunna ASCII EBCDICiksi) ja `ASCII` (muunna EBCDIC ASCIIksi).

## 5.17 diff - Compare two files

`diff [-c|-e|-C` *n*`] [-br]` *file1 file2*

vertaa kahta tiedostoa ja tulostaa listan muutoksista joita ensimmäiselle olisi tehtävä että siitä saisi toisen. Jos tiedostot ovat identtisiä ei tulosta mitään.

Optiot:

`-b` älä ota huomioon alkutyhjiä ja tulkitse peräkkäisten tyhjien jonot samoiksi

`-c` tulosta kolme riviä kontekstia eroille

`-C` *n* tulosta *n* riviä kontekstia

`-e` `ed`-komennon syötöksi sopiva tulosformaatti

`-r` vertaa kahta hakemistoa rekursiivisesti

Exit status on 0 jos eroja ei löytynyt, 1 jos löytyi, suurempi jos tapahtui virhe.

## 5.18 dirname - Return directory portion of pathname

`dirname` *string*

Palauttaa tiedostonimen hakemisto-osan. Mahdollinen lopussa oleva / poistetaan ensin, sen jälkeen lopusta viimeiseen jäljelläolevaan kauttaviivaan saakka. Jos argumentissa ei ole yhtään kauttaviivaa, palauttaa pisteen, ja jos kauttaviivoja on vain alussa peräkkäin tai jos siinä on pelkkiä kauttaviivoja palauttaa `/`:n. (Historiallisista syistä `dirname //` saattaa palauttaa myös `//`:n.)

### 5.19 echo - Write arguments to standard output

Ks. 4.5.6.

### 5.20 ed - Edit text

Rivipohjainen editori, jota ei tässä esitellä sen tarkemmin (mutta vrt. `sed`).

### 5.21 env - Set environment for command invocation

`env [-i] [`*muuttuja*`=`*arvo*`...] [komento...]`

alustaa ympäristömuuttujat annetuilla arvoilla komennon suorituksen ajaksi. Ilman `-i` optiota se on olennaisesti sama kuin

`[`*muuttuja*`=`*arvo*`...] [komento...]`

(vrt. 4.4.8), paitsi että `env` on ulkoinen komento (toimii esim. `find`in kanssa).

Jos `-i` on annettu, ympäristö tyhjennetään ensin, jolloin suoritettava komento saa tasan `env`in asettamat ympäristömuuttujat.

Exit status on 0 jos komento onnistui, 1–125 jos `env` epäonnistui, 126 jos komento löytyi mutta sitä ei voitu suorittaa, ja 127 jos sitä ei löutynyt.

### 5.22 expr - Evaluate arguments as an expression

Ks. 4.10.1, merkkijonovertailun osalta vrt. `grep` (6).

### 5.23 false - Return false value

Palauttaa nollasta eroavan statuksen.

### 5.24 find - Find files

`find` *polku* `[`*optiot...*`]`

etsii tiedostoja hakemistopuusta rekursiivisesti.

Hakua voi ohjata seuraavilla optioilla.

**-atime** *n* tosi jos tiedoston viimeinen käyttöaika (*access time*) on *n* päivää.

**-ctime** *n* tosi jos tiedoston hakemistotiedon (*inode*n) muutosaika on *n* päivää.

**-depth** aina tosi mutta muuttaa hakujärjestystä niin, että alihakemistot käydään läpi ensin.

**-exec** *komento* ... ] ; suorittaa ulkoisen komennon, tosi jos tämän exit status on nolla. Käsiteltävänä oleva polku voidaan välittää argumenttina `{}` (joihin ei yleensä saa liittää muuta).

**-group** *gname* tosi jos tiedoston ryhmä on `gname`.

**-links** *n* tosi jos tiedostoon on *n* linkkiä (ei laske symbolisia linkkejä!).

**-mtime** *n* tosi jos tiedoston muutosaika on *n* päivää.

**-name** *pattern* tosi jos tiedoston nimi vastaa annettua merkkijonolauseketta (*glob pattern*, ts. sama syntaksi kuin shellin tiedostonimilaajennuksissa, eri kuin `grep`issä).

**-newer** *file* tosi jos tiedosto on uudempi kuin *file*.

**-nogroup** tosi jos tiedoston ryhmää ei ole (se puuttuu `/etc/group`-tiedostosta).

**-nouser** tosi jos tiedoston omistajaa ei ole (se puuttuu `/etc/passwd`-tiedostosta).

**-ok** *komento* ... ; kuten `-exec` mutta kysyy interaktiivisesti suoritetaanko komento.

**-perm** [-]*mode* tosi jos tiedoston suojaukset vastaavat *mode*a, joka on annettu samalla syntaksilla kuin `chmod`-komennossa paitsi että `-`:lla alkavia ei saa käyttää, sen sijaan jos alussa on `-` tutkitaan ovatko *ainakin* annetut suojaukset päällä, ilman sitä ovatko ne *täsmälleen* annetut. Kuten chmodissa tässäkin voi sekä symbolista että oktaalimuotoa (jälkimmäistä ei suositella).

**-print** aina tosi, tulostaa tiedostonimen (polun).

**-prune** aina tosi, estää rekursion kohdalla olevan hakemiston sisään. Ei toimi jos myös `-depth` on annettu.

**-size** *n*[c] tosi jos tiedoston koko on *n*/ lohkoa (512 tavun) tai tavua (jos koon lopussa on `c`).

**-type** *c* tosi jos tiedoston tyyppi on *c*, missä *c*:lle tunnistetaan arvot b = lohkolaitetiedosto (*block special file*), c = merkkilaitetiedosto (*character special file*), d = hakemisto, p = nimetty putki ja f = tavallinen tiedosto. Epästandardeja mutta yleisesti toimivia ovat myös l (symbolinen linkki), s (socket), n (*network special file*) ja M (*mount point*).

**-user** *uname* tosi jos tiedoston omistaja on *uname*.

**-xdev** aina tosi, estää rekursion eri laitteelle (*mount point*in yli).

Näitä voi yhdistellä seuraavilla:

**( *lauseke* )** ryhmittely.

**! *lauseke*** negaatio.

***lauseke1* [-a] *lauseke2*** AND; toista lauseketta ei suoriteta jos ensimmäinen on epätosi.

***lauseke1* -o *lauseke2*** OR; toista lauseketta ei suoriteta jos ensimmäinen on tosi.

Jos mitään optioista `-print`, `-exec` tai `-ok` ei ole annettu, optiolauseke tulkitaan kuin `(` *lauseke* `) -print`.

Huomaa että sulut ja puolipiste ovat shellin erikoismerkkejä ja on suojattava.

Yleisiä epästandardeja optioita on mm.

**-follow** seuraa symbolisia linkkejä

**-fstype *fstype*** tosi jos tiedostojärjestelmän tyyppi on *fstype* (esim. `NFS`)

**-fsonly** estää rekursion muihin tiedostojärjestelmätyyppeihin (globaalisti)

**-path *pattern*** kuten `-name` mutta vertaa koko polkua (ja kauttaviivat käyttäytyvät kuten muutkin merkit)

**-inum *n*** tosi jos tiedoston *inode* on *n*.

**-linkedto *file*** tosi jos tiedostosta on linkki *file*:een

**-only** "käänteinen `-prune`": hakemisto käsitellään rekursiivisesti vain jos... Ei toimi yhtaikaa `-depth`'in kanssa.

Joissakin uusissa toteutuksissa voi myös käyttää `-exec`in loppumerkkinä puolipisteen vaihtoehtona plus-merkkiä, sillä toimintaerolla että tiedostoja kootaan useita jokaiselle komennon kutsulle (vrt. `xargs`).

## 5.25 fold - Fold lines

`fold [-bs] [-w` *width*`] [`*file(s)*`]`

jakaa syöttötiedostojen tai stdinin rivit enintään annetun mittaisiksi. Leveys annetaan optiolla `-w` merkkeinä (oletus 80) tai tavuina jos `-b`-optio on annettu. Jos optio `-s` on annettu, yrittää katkaista rivin tyhjän kohdalta.

### 5.26 getconf - Get configuration values

`getconf` *system_var*
`getconf` *path_var pathname*

palauttaa systeemi- tai polkumuuttujan arvon, millä tässä tarkoitetaan järjestelmäriippuvia (raja-)arvoja ja vakioita, jälkimmäiset lisäksi tiedosto(järjestelmä)riippuvia.

Jos kysytty muuttuja on sallittu muttei määritelty tässä järjestelmässä, tulos on merkkijono "`undefined`".

Exit status on nolla jos muuttuja on sallittu, muutoin yksi.

POSIX määrittelee seuraavat systeemimuuttujat:

`ARG_MAX` maximum length of arguments for running a program, in bytes, including environment data.

`CHILD_MAX` maximum number of simultaneous processes allowed per real user.

`CLK_TCK` number of intervals per second in machine clock (optional).

`NGROUPS_MAX` number of simultaneous group IDs per process.

`OPEN_MAX` maximum number of open files at any time per process.

`STREAM_MAX` maximum number of streams that one process can have open at one time.

`TZNAME_MAX` maximum number of bytes supported for the name of a time zone (not of the `TZ` variable).

`_POSIX_ARG_MAX` minimum conforming value for `ARG_MAX`.

`_POSIX_CHILD_MAX` minimum conforming value for `CHILD_MAX`.

`_POSIX_JOB_CONTROL` POSIX job control supported.

`_POSIX_LINK_MAX` minimum conforming value for `LINK_MAX`.

`_POSIX_MAX_CANON` minimum conforming value for `MAX_CANON`.

`_POSIX_MAX_INPUT` minimum conforming value for `MAX_INPUT`.

`_POSIX_NAME_MAX` minimum conforming value for `NAME_MAX`.

`_POSIX_NGROUPS_MAX` minimum conforming value for `NGROUPS_MAX`.

`_POSIX_OPEN_MAX` minimum conforming value for `OPEN_MAX`.

`_POSIX_PATH_MAX` minimum conforming value for `PATH_MAX`.

`_POSIX_PIPE_BUF` minimum conforming value for `PIPE_BUF`.

`_POSIX_SAVED_IDS` processes have saved set-user-ID and saved set-group-ID.

`_POSIX_SSIZE_MAX` value that can be stored in an object of type `ssize_t`.

`_POSIX_STREAM_MAX` minimum conforming value for `STREAM_MAX`.

`_POSIX_TZNAME_MAX` minimum conforming value for `TZNAME_MAX`.

`_POSIX_VERSION` gives version of POSIX adhered to in this release.

`BC_BASE_MAX` maximum `ibase` and `obase` values for the `bc` command.

`BC_DIM_MAX` maximum number of elements permitted in a `bc` array.

`BC_SCALE_MAX` maximum `scale` size allowed in `bc`.

`BC_STRING_MAX` maximum number of characters in a string in `bc`.

`COLL_WEIGHTS_MAX` maximum number of weights assignable to an entry of the `LC_COLLATE order` keyword.

`EXPR_NEST_MAX` maximum number of expressions that you can nest inside parentheses in an expression evaluated by `expr`.

`LINE_MAX` maximum number of characters that a utility can accept as an input line (either from the standard input or a text file) when the utility takes text files as input. This number includes the trailing newline.

`PATH` standard `PATH` setting.

`RE_DUP_MAX` maximum number of repeated occurrences of a regular expression when using the interval notation {m,n} (see `regexp`).

`POSIX2_C_BIND` indicates whether the system supports the C Language Bindings Option.

`POSIX2_C_DEV` indicates whether the system supports the C Language Development Utilities Option.

`POSIX2_FORT_DEV` indicates whether the system supports the FORTRAN Development Utilities Option.

`POSIX2_FORT_RUN` indicates whether the system supports the FORTRAN Runtime Utilities Option.

`POSIX2_LOCALEDEF` indicates whether the system supports the creation of locales.

`POSIX2_SW_DEV` indicates whether the system supports the Software Development Utilities Option.

`POSIX2_CHAR_TERM` indicates whether the system supports at least one terminal type capable of all operations necessary for the User Portability Utilities. Only on if `POSIX2_UPE` is on.

`POSIX2_UPE` indicates whether the system supports the User Portability Utilities Option.

POSIX2_VERSION gives the version of POSIX.2 adhered to in this release.

POSIX2_BC_BASE_MAX minimum conforming value for BC_BASE_MAX.

POSIX2_BC_DIM_MAX minimum conforming value for BC_DIM_MAX.

POSIX2_BC_SCALE_MAX minimum conforming value for BC_SCALE_MAX.

POSIX2_BC_STRING_MAX minimum conforming value for BC_STRING_MAX.

POSIX2_COLL_WEIGHTS_MAX minimum conforming value for EQUIV_CLASS_MAX.

POSIX2_EXPR_NEST_MAX minimum conforming value for EXPR_NEST_MAX.

POSIX2_LINE_MAX minimum conforming value for LINE_MAX.

POSIX2_RE_DUP_MAX minimum conforming value for RE_DUP_MAX.

ja seuraavat polkumuuttujat:

LINK_MAX maximum number of links that this file can have.

MAX_CANON maximum number of bytes in the terminal's canonical input queue (before line editing).

MAX_INPUT space available in terminal input queue.

NAME_MAX largest file name size.

PATH_MAX maximum number of bytes in a path name.

PIPE_BUF largest atomic write to a pipe.

_POSIX_CHOWN_RESTRICTED restrictions apply to file ownership changes.

_POSIX_NO_TRUNC if set, it is an error for any path name component to be longer than NAME_MAX bytes.

_POSIX_VDISABLE processes are allowed to disable terminal special characters.

Eri järjestelmissä `getconf` voi tuntea muitakin muuttujia, mutta sellaisia käytettäessä on syytä varautua sekä tulokseen `"undefined"` että virheeseen. Itse asiassa siihen on syytä varautua yllämainittujenkin kanssa, erityisesti SSIZE_MAX, _POSIX_SSIZE_MAX, STREAM_MAX, _POSIX_STREAM_MAX, TZNAME_MAX ja _POSIX_TZNAME_MAX eivät olleet mukana alkuperäisessä POSIX.1-standardissa ja saattavat puuttua joistakin järjestelmistä.

Muutenkaan tuloksiin ei aina kannata sokeasti luottaa, vaan "sanity check" on aina paikallaan.

## 5.27 getopts - Parse utility options

Yleinen optionkäsittelijä, jolla saa standardinmukaiset optiot shell-skripteihin.

Useimmat standardikomennot noudattavat seuraavia sääntöjä optioiden ja argumenttien käsittelyssä:

- Optionimet ovat yksikirjaimisia, isot ja pienet kirjaimet erillisiä, myös numerot sallittuja mutta muut merkit eivät.
- Optiot alkavat tavuviivalla (`-`).
- Argumentittomia optioita saa yhdistellä yhden tavuviivan perään.
- Argumentillisten optioiden pitää olla erillisiä ja sen argumentin pitää olla pakollinen. Optio pitäisi erottaa argumentistaan tyhjällä. (Tähän on historiallisista syistä paljon poikkeuksia. Tyhjä on kuitenkin aina sallittu, vaikka monet komennot eivät sitä vaadikaan.)
- Jos optio tarvitsee useita argumentteja ne pitää antaa yhtenä merkkijonona pilkuilla tai välilyönnillä erotettuina.
- Optiot pitää antaa komentorivillä ennen komennon argumentteja.
- Optiolistan voi lopettaa argumentilla ”`--`”.
- Optioiden keskinäisellä järjestyksellä ei pitäisi olla merkitystä.
- Argumenttien keskinäisellä järjestyksellä voi olla merkitys.
- Tiedostoargumenttina ”`-`” tarkoittaa standard inputia tai standard outputia.

Näiden sääntöjen mukaisen optiokäsittelyn saa tehtyä `getopts`illa seuraavasti: joka kerta kutsuttaessa

`getopts` *optstring varname* `[`*arg...*`]`

tutkii onko seuraava optiokirjain merkkijonossa *optstring* ja jos on, asettaa sen muuttujaan *varname*. Sen lisäksi se asettaa muuttujaan `OPTIND` seuraavan option järjestysluvun (shellin käynnistyessä `OPTIND` alustetaan 1:ksi) ja jos optiokirjaimen perässä *optstring*issä on kaksoispiste, option argumentin muuttujaan `OPTARG`.

Kun optiot loppuvat, `getopts` palauttaa nollasta eroavan statuksen, ja `OPTIND` osoittaa ensimmäiseen ei-optio-argumenttiin komentorivillä (mahdollisen `--`:n ohi).

Jos optiokirjainta ei ole *optstring*issä, `getopts` asettaa *varname*:een kysymysmerkin. Jos lisäksi *optstring* alkaa kaksoispisteellä, tallettaa optiokirjaimen muuttujaan `OPTARG`, muutoin aiheutuu virheilmoitus (mutta `getopts`in exit status on silti nolla).

Jos optiokirjaimen perässä *optstring*issä on kaksoispiste mutta optiolle ei ole annettu argumenttia, aiheuttaa virheilmoituksen paitsi jos *optstring*in alussa on kaksoispiste, jolloin *varname*en sijoitetaan kaksoispiste ja optiokirjain muuttujaan `OPTARG`.

Jos `getopts`ille annetaan argumentteja *optstring*in ja *varname*n lisäksi, se tutkii niitä positioparametrien asemesta.

Esimerkki: skripti ymmärtää optiot `-a`, `-b` ja `-f` *tiedosto*, ja haluaa niiden perään yhden argumentin:

```
opta=
optb=
file=
while getopts :abf: opt ;do
  case $opt in
    a) opta=1 ;;
    b) optb=1 ;;
    f) file="$OPTARG" ;;
    ?) printf "%s: invalid option %s\n" "$0" "$OPTARG"
       exit 1 ;;
  esac
done
shift $((OPTIND - 1))
case $# in
  0) printf "%s: argument missing\n", "$0"
     exit 2 ;;
  1) arg="$1" ;;
  *) printf "%s: too many arguments\n", "$0"
     exit 3 ;;
esac
...
```

### 5.28 grep - File pattern searcher

Esitellään tarkemmin myöhemmin (6).

### 5.29 head - Copy the first part of files

`head [-n` *number*`] [`*file(s)*`]`
`head [-`*number*`] [`*file(s)*`]`

tulostaa tiedostoista tai stdinistä *number* (oletus 10) ensimmäistä riviä. Alempi syntaksi on vanha ja virallisesti epäsuosiossa.

Jos argumentteja on useita, tulostaa alkurivit kustakin tiedostosta erikseen tiedostonimien kera.

## 5.30 id - Return user identity

`id [-G|-g|-u] [-nr] [`*user*`]`

tulostaa käyttäjän UID:n ja ryhmät. Optiot:

-G tulosta vain ryhmät

-g tulosta vain EGID

-n tulosta numeerisen UID:n asemesta käyttäjätunnus

-r tulosta todellinen (real) ID vaikuttavan (EID) asemesta

-u tulosta vain UID

## 5.31 join - Relational database operator

`join [-a` *n* | `-v` *n*`] [-e` *s*`] [-o` *list*`] [-t char] [-1` *f1*`] [-2` *f2*`]` *file1 file2*

yhdistää kaksi tiedostoa yhteisen avainsarakkeen perusteella. Tiedostojen on oltava valmiiksi lajiteltuja ko. sarakkeen mukaan (kuten `sort -b` tekisi). Oletuksena tulostaa ensin ensimmäisen avainsarakkeen, sitten ensimmäisen tiedoston vastaavan rivin muut sarakkeet ja sitten toisen tiedoston muut sarakkeet.

Sarakkeiden erottimena on oletuksena tyhjä (ja monta peräkkäistä tyhjää lasketaan yhdeksi), ja alkutyhjiä ei oteta huomioon.

Seuraavassa "vastineeton rivi" tarkoittaa riviä, jolle ei löydy vastinetta (riviä jolla olisi sama avainsarake) toisesta tiedostosta.

Optiot:

`-a` *n* tulosta vastineettomat rivit tiedostosta *n*, missä *n* on 1 tai 2. Jos sekä `-a 1` että `-a 2` on annettu, tulostaa kaikki vastineettomat rivit.

`-e` *string* korvaa tyhjät kentät merkkijonolla *string*.

`-o` *list* määrittää mitkä sarakkeet tulostetaan ja missä järjestyksessä. Sarakemääritykset ovat muotoa *n*.*f*, missä *n* on tiedostonumero (1 tai 2) ja *f* sarakenumero, ja niitä voi olla mielivaltainen määrä pilkulla tai tyhjällä erotettuna.

`-t` *char* käytä kentän erottimena merkkiä *char*, sekä syötössä että tulostuksessa. Tätä käytettäessä tiedostojen lajittelujärjestyksen pitäisi olla sama kuin `sort`in ilman `-b`-optiota.

`-v` *n* tulosta *vain* (normaalin tulostuksen asemesta) vastineettomat rivit tiedostosta *n*, missä *n* on 1 tai 2. Jos sekä `-v 1` että `-v 2` on annettu, tulostaa kaikki vastineettomat rivit.

`-1` *f* käytä avainsarakkeena ensimmäisestä tiedostosta saraketta *f* (kokonaisluku 1...).

`-2` *f* käytä avainsarakkeena toisesta tiedostosta saraketta *f*.

Seuraavat optiot ovat virallisesti ”obsolete” mutta käytännössä toimivat yhteensopivuussyistä lähes kaikkialla. Uusissa skripteissä niitä ei kuitenkaan tulisi käyttää:

`-j1` *f* sama kuin `-1` *f*.

`-j2` *f* sama kuin `-2` *f*.

`-j` *f* sama kuin `-1` *f* `-2` *f*.

## 5.32 kill - Terminate or signal processes

`kill -s` *signaalinimi pid...*
`kill -l [`*signaalinumero*`]`
`kill -[SIG`*signaalinimi*|*signaalinumero*`]` *pid...*

Lähettää prosess(e)ille halutun signaalin. Viimeinen syntaksi on virallisesti vanhentunut, mutta toimii käytännössä kaikkialla. Jos signaalia ei anneta komentorivillä, oletus on `SIGTERM`.

Optio `-l` tulostaa kaikki järjestelmän tuntemat signaalit, tai annettua numeroa vastaavan signaalin nimen.

Signaali voi olla jokin seuraavista (etuliitettä `SIG` ei tarvita option `-s` kanssa), suluissa numeroarvot niille joille ne on standardoitu (kaikille löytyy kyllä numero joka järjestelmästä mutta muut vaihtelevat):

`0` null signal[3]

`SIGHUP (1)` hangup[1]

`SIGINT (2)` terminal interrupt [1]

`SIGQUIT (3)` terminal quit[2]

`SIGABRT (6)` abort[2]

`SIGKILL (9)` kill (cannot be caught or ignored)[2]

`SIGALRM (14)` alarm clock[1]

`SIGTERM (15)` software termination[1]

`SIGILL` illegal instruction[1]

`SIGTRAP` trace/breakpoint trap[2]

`SIGFPE` floating point exception[2]

`SIGUSR1` user defined signal 1[1]

`SIGSEGV` segmentation violation[2]

`SIGUSR2` user defined signal 2[1]

`SIGPIPE` write on a pipe with no one to read it[1]

`SIGCHLD` child process terminated or stopped[3]

`SIGCONT` continue if stopped[5]

`SIGSTOP` stop signal (cannot be caught or ignored)[4]

`SIGTSTP` interactive stop signal[4]

`SIGTTIN` background process wants to read[4]

`SIGTTOU` background process wants to write[4]

Oletustoiminta jos prosessi ei ota signaalia kiinni:
[1] prosessi lopetetaan epänormaalisti, siivotaan kuten `_exit()`
[2] prosessi lopetetaan epänormaalisti, järjestelmäkohtaisin seurauksin (core dump)
[3] ei vaikutusta
[4] prosessi pysäytetään
[5] jos prosessi on pysähtynyt sen suoritusta jatketaan, muuten ei vaikutusta.

Exit status on nolla jos ainakin yksi prosessi käsitteli signaalin, muutoin > 0. Niinpä
`kill -0` *pid*
sopii prosessin olemassaolon testaamiseen.

Useimmista järjestelmistä löytyy muitakin, epästandardeja signaaleja, esim.

`SIGBUS` Bus error

`SIGPOLL` Pollable event

`SIGPROF` Profiling timer expired

`SIGPWR` Power fail/restart

`SIGSYS` Bad system call

`SIGURG` High bandwidth data is available at a socket

`SIGVTALRM` Virtual timer expired

`SIGWINCH` Window size change

`SIGXCPU` CPU time limit exceeded

`SIGXFSZ` File size limit exceeded

### 5.33 ln - Link files

`ln [-f] [-s]` *tiedosto(t) kohde*

"Linkittää" tiedostoja, ts. luo niille uuden hakemistotiedon (*link()*). Jos linkitettäviä tiedostoja on useampia, kohteen on oltava ennestään olemassa oleva hakemisto, ja linkki tehdään sen alle alkuperäisellä nimellä. Joka tapauksessa kohteen on oltava samassa tiedostojärjestelmässä eikä hakemistoa voi linkittää, ellei optiota `-s` ole käytetty.

Jos kohdetiedosto on olemassa, aiheutuu virhe paitsi jos optio `-f` on annettu, jolloin se yritetään ensin poistaa (*unlink()* — ei onnistu hakemistolle).

Hakemistolle ei voi tehdä linkkiä.

POSIX.2 ei itse asiassa määritellyt symbolisia linkkejä lainkaan, eikä niitä varsinkaan ei-Unixmaisissa POSIX-järjestelmissä useinkaan ole, mutta kaikissa moderneissa Unixeissa kylläkin.

Symbolinen linkki on osoitin tiedoston *nimeen* (tavallinen tai "hard" linkki osoittaa inodeen, eikä itse asiassa eroa alkuperäisestä nimestä mitenkään), ja voi siten osoittaa tiedostojärjestelmästä toiseen. Symbolinen linkki voi myös osoittaa hakemistoon tai vaikka tiedostoon jota ei edes ole olemassa, ja alkuperäisen tiedoston nimen vaihtuessa se hukkaa sen. Saadessaan argumenttina symbolisen linkin lähes kaikki komennot käyttävät tiedostoa johon se viittaa (poikkeuksena erityisesti `rm`).

### 5.34 locale - Get locale-specific information

`locale [-a | -m]`
`locale [-ck]` *name...*

palauttaa tietoja paikallisesta kieliympäristöstä (*locale*, kieli- ja maa-valinta).

Ilman optioita tulostaa voimassa olevat asetukset, ilman argumentteja yhteenvedon (ts. `LC_*`- ja `LANG`-muuttujien arvot), jos argumentteja on annettu, tulostaa niitä vastaavat asetukset. Argumentti voi olla kategoria (`LC_*`), jolloin tulostetaan kaikki siihen kuuluvat asetukset, tai avainsana joka vastaa jotakin asetusta.

Optiot:

`-a` tulosta lista järjestelmän tuntemista kieliympäristöistä

`-c` tulosta kategorianimet

`-k` tulosta avainsanat

`-m` tulosta lista järjestelmän tuntemista merkistöistä

Mahdolliset kategoriat ja tärkeimmät niihin liittyvät avainsanat ovat:

`LC_CTYPE`: Merkkien luokitteluun ja iso-pieni-muunnokseen liittyvät koodit, vaihtelevat järjestelmittäin. Käytännössä hyödyllinen voisi olla lähinnä `charmap`, joka kertoo käytössä olevan merkistön.

`LC_NUMERIC`: lukujen esitykseen liittyvät:

> `decimal_point` desimaalierotin (piste, pilkku)
>
> `thousands_sep` tuhansien (jne) erotinmerkki
>
> `grouping` erotteluvälit edelliselle

`LC_TIME`: ajan ja päivämäärien esitykseen liittyvät, mm:

> `abday` viikonpäivien lyhenteet
>
> `day` viikonpäivien nimet
>
> `abmon` kuukausien lyhenteet
>
> `mon` kuukausien nimet
>
> `am_pm` aamu- ja iltapäiväsymbolit
>
> `d_t_fmt` ajan ja päiväyksen oletusformaatti (mitä `date`:n `%c` vastaa)
>
> `d_fmt` päiväyksen oletusformaatti (`date`:n `%x`)
>
> `t_fmt` kellonajan oletusformaatti (`date`:n `%T`)
>
> `t_fmt_ampm` 12h-ajan oletusformaatti (`date`:n `%r`)

`LC_COLLATE`: lajittelualgoritmin määräävät avainsanat, vaihtelevat järjestelmittäin eikä niitä juurikaan tarvita suoraan.

`LC_MONETARY`: valuutan esittämiseen liittyvät:

> `int_curr_symbol` kansainvälinen valuuttasymboli (esim. FIM)
>
> `currency_symbol` paikallinen valuuttasymboli (esim. mk)
>
> `mon_decimal_point` desimaalierotin
>
> `mon_thousands_sep` tuhansien erotin

- `mon_grouping` em. ryhmittelyväli
- `positive_sign` positiivinen etumerkki
- `negative_sign` negatiivinen etumerkki
- `int_frac_digits` desimaalien määrä kv. yhteyksissä
- `frac_digits` desimaalien määrä paikallisesti
- `p_cs_precedes` tuleeko valuuttasymboli ennen (1) vai jälkeen (0) positiivisissa luvuissa
- `p_sep_by_space` erotetaanko valuuttasymboli välilyönnillä positiivisissa luvuissa (0=ei, 1=kyllä, 2=erotetaan myös etumerkistä)
- `n_cs_precedes` tuleeko valuuttasymboli ennen (1) vai jälkeen (0) negatiivisissa luvuissa
- `n_sep_by_space` erotetaanko valuuttasymboli välilyönnillä negatiivisissa luvuissa
- `p_sign_posn` positiivisen etumerkin paikka (0=luku suluissa, 1=ennen lukua ja valuuttasymbolia, 2=luvun ja valuuttasymbolin jälkeen, 3=ennen valuuttasymbolia, 4=valuuttasymbolin jälkeen)
- `n_sign_posn` negatiivisen etumerkin paikka

`LC_MESSAGES`: viestejä käyttäjälle ja -ltä:

- `yesexpr` ”kyllä” paikallisella kielellä
- `noexpr` ”ei” paikallisella kielellä
- `yesstr` ”kyllä”-vastaukseksi tunnistettava *regexp* (muodossa jonka `grep -E` tunnistaa)
- `nostr` ”ei”-vastaukseksi tunnistettava *regexp*

Näitä käyttävät soveltuvin osin kaikki standardikomennot, erityisesti kaikki jotka vertailevat tai lajittelevat merkkijonoja tai tulostavat kellonaikoja tai päivämääriä. Jos jotakin kategorianimeä vastaava ympäristömuuttuja ei ole asetettu käytetään sen asemesta muuttujaa `LANG`. Jos kuitenkin `LC_ALL` on asetettu käytetään sitä riippumatta muista `LC_*`- tai `LANG`-muuttujista. Jos arvo on virheellinen tai ko. järjestelmässä tuntematon, käytetään oletuksena arvoa `POSIX` tai `C` (jotka on määritelty kaikissa järjestelmissä).

Muuttujien `LC_*` ja `LANG` mahdolliset arvot vaihtelevat eri järjestelmissä, mutta yleensä ne ovat muotoa
*kieli*`[_`*alue*`][.`*merkistö*`]`
missä *kieli* on joko kaksikirjaiminen lyhenne tai kielen englanninkielinen nimi täydellisenä (mutta pienillä kirjaimilla) (esim. `fi` tai `finnish`), *alue* on maan tai alueen kaksikirjaiminen lyhenne (isoilla kirjaimilla, esim. `FI`), ja *merkistö* on käytetyn merkistön lyhenne (vaihtelevat suuresti). Suomessa voivat tulla kyseeseen esim. `fi_FI.iso88591`, `fi_FI.iso885915@euro`, `finnish.iso8859-1`, `finnish` jne.

Jos haluaa olla varma että kieliympäristö ei muuta jonkin komennon tulostusta voi asettaa `LC_ALL`-muuttujan tunnettuun arvoon, esim.

```
LC_ALL=C DOW=$(DATE +%a)
```

(Tuota **ei** pidä tehdä jos kyseessä on ihmisten luettavaksi tarkoitettu viesti — ne yleensä halutaan voida kääntää paikalliselle kielelle.)

Esim. `LC_MESSAGES`-kategorian avulla saa monikielisen kyllä/ei-kysymyksen:

```
printf "%s? " "$(locale yesstr)/$(locale nostr)"
read reply
if printf "%reply\n" "$query?" | grep -Eq "$(locale yesexpr)"
then echo "se on suomeksi 'joo'!"
else echo "se on suomeksi 'ei hitossa'!"
fi
```

Tämä ei kuitenkaan monessakaan järjestelmässä toimi kuten pitäisi, erityisesti joitakin arvoja ei ehkä ole määritelty ollenkaan joten oletusarvot on syytä laittaa, ja mm. HP-UX lisää ylimääräisiä lainausmerkkejä vastauksiinsa. Paranoidisiirrettävyyttä kaipaava voisi tehdä esim. näin:

```
# get locale-specific yes/no question & reply, defaults in English
nostr=$(locale nostr); nostr=${nostr:-no}
yesstr=$(locale yesstr); yesstr=${yesstr:-yes}
yesexpr=$(locale yesexpr); yesexpr=${yesexpr:-"^[yY]"}
# peel possible extra quotes away
for var in yesstr nostr yesexpr ;do
  eval value=\$$var
  case "$val" in \"*) eval $var=$value ;; esac
done
printf "%s? " "$yesstr/$nostr"
read reply
if printf "%s\n" "$reply" | grep -Eq "$yesexpr"
then echo "se on suomeksi 'joo'!"
else echo "se on suomeksi 'ei hitossa'!"
fi
```

Esim. Halutaan tutkia onko annettu merkkijono luku, jossa voi olla desimaalierotin (muttei muita koristeita):

```
case "${luku:-BAD}" in
  *[!0-9$(locale decimal_point)]*)
       echo "'$luku' is not a proper number"  ;;
esac
```

## 5.35 localedef - Define locale environment

`localedef [-c] [-f` *charmap*`] [-i` *sourcefile*`]` *name*

luo uuden kieliympäristön annetun määrityksen mukaisesti. Yksityiskohtiin ei tässä puututa sen enempää.

## 5.36 logger - Log messages

`logger` *tekstiä*

lähettää viestin systeemilokiin. Käyttökelpoinen lähinnä järjestelmän ylläpitäjälle. Lokiin menee yleensä automaattisesti myös aikaleima ja tieto siitä, kuka viestin lähetti.

## 5.37 logname - Return user's login name

Palauttaa käyttäjätunnuksen (kuten `getlogin()`, näkee `su`:n läpi jne).

## 5.38 lp - Send files to a printer

`lp [-c] [-d` *dest*`] [-o` *option...*`] [-n` *copies*`] [`*file(s)*`]`

lähettää tiedostoja (tai stdinin) kirjoittimelle. Optiot:

- `-c` Palaa vasta kun tiedostoja ei enää tarvita.
- `-d` *kirjoitin* Käytä nimettyä kirjoitinta. Ilman tätä oletus on ensisijaisesti `$LPDEST`, jos sitä ei ole `$PRINTER`, jos sitäkään ei ole systeemin oletuskirjoitin ja jos sitäkään ei ole tapahtuu virhe.
- `-n` *k* Tulosta *k* kopiota.
- `-o` *option* Tätä POSIX ei vaadi mutta suosittelee järjestelmäkohtaisiin optioihin.

Useimmissa järjestelmissä on muitakin optioita, mutta vain nuo ovat edes kohtuullisen siirrettäviä. BSD-pohjaisissa järjestelmissä vastaava komento on historiallisesti `lpr`; sen optioista mainittakoon `-P`*kirjoitin*, joka vastaa `lp`:n `-d`:tä.

## 5.39 ls - List directory contents

`ls [-CFRacdilqrtu1] [file ...]`

Tulostaa hakemiston sisällysluetteloa: muille kuin hakemistoargumenteille ko. tiedoston tiedot, hakemistoille ko. hakemiston sisältämien tiedostojen tiedot, jos argumentteja ei ole, työhakemiston sisällön.

Optiot:

- `-C` Monisarakkeinen tulostus aakkosjärjestyksessä. Ympäristömuuttuja `COLUMNS` määrää tulostusleveyden.
- `-F` Tulostaa kauttaviivan (`/`) hakemistonimien, tähden (`*`) suoritettavien ja pystyviivna (|) putkien perään. (Yleensä lisäksi `=` sockettien ja `@` symbolisten linkkien perään.)
- `-R` Tulostaa alihakemistot rekursiivisesti.
- `-a` Tulostaa myös pisteellä alkavat tiedostonimet. Ilman `-a`:ta niitä ei tulosteta ellei niitä ole erikseen annettu argumentteina.
- `-c` Käyttää hakemistotiedon muutosaikaa (`ctime`) itse tiedoston muutosajan asemesta lajittelussa (`-t`) ja tulostuksessa (`-l`).
- `-d` Tulostaa tiedostoistakin niiden omat tiedot niiden sisällön asemesta.
- `-i` Tulostaa tiedostojen *inode*-numerot (tai yleisemmin "järjestysnumero").
- `-l` Pitkä tulostusmuoto, yksi tiedosto per rivi.
- `-q` Tulostaa erikoismerkkien paikalle kysymysmerkin.
- `-r` Lajittelee käänteisessä järjestyksessä (viimeinen tai vanhin ensin).
- `-t` Lajittelee tiedostot ensin aikajärjestykseen (oletuksena tiedoston muutosajan mukaan), vasta sitten aakkosjärjestykseen.
- `-u` Käyttää tiedoston käyttöaikaa (*atime*) tiedoston muutosajan asemesta lajittelussa (`-t`) ja tulostuksessa (`-l`).
- `-1` Tulostaa vain yhden tiedosto per rivi. Oletus silloin kun tulostuslaite ei ole pääte.

Argumenttina annettu tiedostonimi tulostetaan sellaisena kuin se on annettu (polkuineen).

Optiolla `-l` tulostetaan seuraavat tiedot järjestyksessä:
*mode, links, owner, group, bytes, date and time, pathname*
Päivämäärän ja kellonajan muotoilu riippuu kieliympäristöstä (`LC_TIME`), `POSIX`-ympäristössä formaatti on kuten `date` "`+%b %e %H:%M`" tulostaisi jos tiedosto on alle 6 kuukautta vanha, muuten kuin "`+%b %e %Y`".

Laitetiedostoille voi koon asemesta tulostua jotain järjestelmäkohtaista.

*Mode* on muotoa *tuuuooogga*, missä

*t*: tiedoston tyyppi, d=hakemisto, b=lohkolaitetiedosto, c=merkkilaitetiedosto, p=putki, -=tavallinen tiedosto. (Yleensä lisäksi l=symbolinen linkki, s=socket, n=*network special file.*)

*uuu*: tiedoston omistajan oikeudet vasemmalta oikealle:

- `r` jos tiedosto on luettavissa, muutoin `-`
- `w` jos tiedosto on luettavissa, muutoin `-`
- `x` jos tiedosto on suoritettavissa, `s` jos tiedosto on suoritettavissa ja *setuid*-bitti on päällä, `S` jos tiedosto ei ole suoritettavissa ja *setuid*-bitti on päällä, muutoin `-`.

*ggg*: ryhmän oikeudet, kuten edellä mutta `s` ja `S` viittaavat *setgid*-bittiin.

*ooo*: muiden oikeudet, `rxw-` kuten edellä, muita ei määritelty. Epästandardi mutta yleinen on `t` = *sticky bit* on päällä (tai `T` jos suoritusoikeus samalla puuttuu).

*a*: *alternate access method flag*, jos epätyhjä (suositus: `+`) kertoo että tiedostolle on jokin muukin suojaus (yleensä *ACL* eli *Access Control List*).

Yleisiä epästandardeja optioita ovat

`-x` Riveittäin lajiteltu monipalstainen tulostus.

`-m` Tiivis pilkuilla eroteltu tulostus.

`-A` Kuten `-a` mutta jättää tiedostot `.` ja `..` tulostamatta.

`-b` Tulostaa erikoismerkit (enemmän tai vähemmän) tunnistettavassa muodossa (vrt. `cat -v`).

`-s` Tulostaa tiedoston koon lohkoina (yleensä 512 tavua).

`-L` Jos tiedosto on symbolinen linkki, näyttää sen osoittaman tiedoston tiedot itse linkin asemesta.

### 5.39.1 Epästandardien suojausbittien merkityksestä

Standardi ei määrittele mitä *setuid*- ja *setgid*-bitit tarkoittavat jakemistoille eikä *sticky bit*in merkitystä yleensä, mutta seuraavat ovat aika vakiintuneita:

- *Sticky bit tiedostolle* tarkoittaa, että suorituksen päätyttyä ohjelmakoodi jätetään muistiin (kunnes muisti loppuu tms). Historiallisesti yleinen, nykyisissä järjestelmissä harvinainen (ei yleensä vaikuta mitään, tai saattaa olla muussa käytössä).
- *Sticky bit hakemistolle* tarkoittaa, että siellä olevan tiedoston saa poistaa vain jos itse tiedostoon on kirjoitusoikeus (muuten riittää kirjoitusoikeus hakemistoon). Yleinen kaikissa `/tmp`:n tyyppisissä hakemistoissa.

- *Setgid hakemistolle* tarkoittaa, että sinne luodut uudet tiedostot perivät hakemiston ryhmän.

### 5.40 mailx - Process messages

`mailx [-s` *subject*`]` *address...*

lähettää standard inputin sähköpostina. (Huom. `mail` ei ole standardi, eikä ole mitään siirrettävää tapaa antaa sille otsikkotietoa.)

### 5.41 mkdir - Make directories

`mkdir [-p] [-m` *mode*`]` *dir...*

luo nimetyt hakemistot. Optiot:

`-p` Luo mahdollisesti puuttuvat komponentit polun varresta, ei aiheuta virhettä valmiiksi olemassa olevista hakemistoista.

`-m` *mode* Asettaa luodun hakemiston suojauksen, *mode* kuten `chmod`-komennossa. Oletussuojaus on `rwxrwxrwx` XOR *umask*. Option `-p` takia luotavien välihakemistojen suojaukseen tulee kuitenkin aina `u+wx`.

### 5.42 mkfifo - Make FIFO special files

`mkfifo [-m` *mode*`]` *file...*

luo nimetyn putken. Optiolla `-m` voi asettaa sen suojausbitit, *mode* kuten `chmod`issa.

Historiallisesti samaan tarkoitukseen on käytetty komentoa `mknod` *file* `p`.

### 5.43 mv - Move files

`mv [-fi]` *tiedosto(t) kohde*

siirtää tai uudelleennimeää tiedostoja, joko luomalla uuden hakemistotiedon (kuten `ln`) ja poistamalla vanhan (jos siirrettäviä on vain yksi ja kohde on samassa tiedostojärjestelmässä) tai kopiomalla tiedoston tai tiedostot (kuten `cp`) ja hävittämällä alkuperäiset. Jos siirrettäviä on useita, kohteen on oltava olemassa oleva hakemisto.

Jos kohde on ennestään olemassa eikä siihen ole kirjoitusoikeutta aiheutuu virhe, paitsi jos `-f` on annettu jolloin se yritetään ensin poistaa kyselemättä, tai jos stdin on pääte tai `-i` on annettu jolloin kysytään käyttäjältä (ts. interaktiivisesti `-i` on oletus). Jos on annettu sekä `-f` että `-i`, viimeinen ratkaisee.

Mahdollisen virheen sattuessa joko alkuperäinen tai uusi tiedosto voi olla korruptoitunut, mutta jompi kumpi on aina ehjä.

Historiallisesti `mv` ei ole osannut siirtää hakemistoa tiedostojärjestelmästä toiseen, mutta POSIX edellyttää että senkin pitäisi onnistua, kunhan kaikkiin siirrettäviin tiedostoihin on lukuoikeus (mitä ei tiedostojärjestelmän sisällä tarvita).

## 5.44 nohup - Invoke a utility immune to hangups

`nohup` *komento* `[`*argumentteja...*`]`

suorittaa ulkoisen komennon niin, että se ei keskeydy saadessaan `SIGHUP`-signaalin. Jos stdout on pääte se suunnataan tiedoston `./nohup.out` perään (appending) tai jos siihen ei voi kirjoittaa, `$HOME/nohup.out` (jos siihenkään ei voi kirjoittaa, aiheutuu virhe). Jos stderr on pääte se suunnataan samaan tiedostoon kuin stdout.

Exit status on 126 jos komento löytyi mutta sitä ei voinut suorittaa, 127 jos komentoa ei löytynyt tai `nohup` itse epäonnistui, muuten komennon palauttama status.

Huom. `nohup` suorittaa vain yhden komennon. Jos se halutaan tehdä putkelle tms, tarvitaan jotain tällaista:

```
nohup sh -c ’prog1 | prog2’
```

## 5.45 od - Dump files in various formats

`od [-v] [-A` *base*`] [-j` *skip*`] [ŋN` *count*`] [-t` *type*`]` *file(s)*

tulostaa tiedoston (tai stdinin) halutussa formaatissa. Optiot:

- `-A` *base* Kantaluku osoitteille: d=desimaali, o=oktaali, x=heksadesimaali, n=osoitteita ei tulosteta.
- -j *skip* Ohita *skip* tavua alusta. Oletuksena *skip* on desimaaliluku, jos se alkaa `0x` heksadesimaali, jos se muuten alkaa nollalla oktaaliluku. Suffikseilla b, k ja m se tulkitaan blokeiksi (512 tavua), kiloiksi (1024) tai megoiksi (1048576).
- `-N` *count* Tulosta enintään *count* tavua (*count* voi olla desimaali-, oktaali- tai heksadesimaaliluku kuten *skip*).
- `-t` *type* Tulostustyyppi: a=ASCII (7-bittinen, kontrollimerkit nimillä), c=merkki, d=desimaaliluku, f=liukuluku, o=oktaaliluku, u=etumerkitön kokonaisluku, x=heksadesimaaliluku. Kaikkien perässä voi olla luku joka määrää montako tavua tulostetaan; f:n perässä tarkkuus F=float, D=double tai L=long double; d:n, o:n, u:n ja x:n perässä voi olla C=char, S=short, I=int, L=long. Tyypit kuten C:ssä.
- `-v` Tulosta kaikki. Ilman `-v`:tä identtiset rivit korvataan tähdillä.

Optioiden puuttuessa oletus on `-t o2`.

## 5.46 paste - Merge corresponding or subsequent lines of files

`paste [-s] [-d` *lista*`]` *tiedosto...*

tulostaa tiedostoja rinnakkain, oletuksena erotettuna tabulaattorilla. Muita lyhyemmät tiedostot käsitellään kuin niiden lopussa olisi tyhjää, ellei `-s`-optiota ole annettu.

Optiot:

-d *lista* Käytä erottimena tabulaattorin asemesta *listan* merkkejä järjestyksessä (tarvittaessa toistuvasti). Kunkin rivin (tai tiedoston `-s`-option kanssa) jälkeen aloitetaan listan alusta, ja viimeisen tiedoston rivinvaihtoa (tai tiedoston viimeistä rivin vaihtoa `-s`:n kanssa) ei korvata. Seuraavilla merkkiyhdistelmillä on erikoismerkitys:

> `\n` = rivinvaihto
> `\t` = tabulaattori
> `\\` = kenoviiva (\)
> `\0` = tyhjä merkkijono (`-d ''` ei yleensä toimi)

`-s` Tulosta jokainen tiedosto omana rivinään, niin että vastinrivit tulevat alekkain samoihin sarakkeisiin.

Stdiniä luetaan vain jos - on annettu argumenttina. Jos se on annettu monta kertaa, stdiniä luetaan rivi kerrallaan jokaista listan alkiota kohti yhä uudestaan.
Esim. `ls | paste - - - -` tulostaa hakemiston neljään palstaan.

## 5.47 pathchk - Check pathnames

`pathchk [-p]` *tiedostonimi...*

tarkistaa että annetut tiedostonimet ovat laillisia, oletuksena käytettävässä järjestelmässä, eli että

- nimi ei ole pitempi kuin `PATH_MAX`,
- nimen yksikään komponentti ei ole pitempi kuin `NAME_MAX`,
- yksikään komponentti ei ole hakemistossa johon ei ole hakuoikeutta, ja
- missään komponentissa ei ole merkkejä jotka eivät olisi laillisia sen hakemistossa.

Komponenttien puuttuminen ei aiheuta virhettä jos ne olisi mahdollista luoda.

Jos optio `-p` on annettu tutkitaan nimen siirrettävyyttä, eli että

- nimi ei ole pitempi kuin `_POSIX_PATH_MAX`,
- nimen yksikään komponentti ei ole pitempi kuin `_POSIX_NAME_MAX`,
- nimessä ei ole merkkejä jotka eivät kuulu siirrettävien tiedostonimien merkistöön (*portable filename character set*)

### 5.48 pax - Portable archive interchange

```
pax  [-cdnv] [-f archive] [-s replstr] ...  [pattern ...]

pax  -r [-cdiknuv] [-f archive] [-o options] ...  [-p string] ...
      [-s replstr] ...  [pattern ...]

pax  -w [-dituvX] [-b blocksize] [ [-a] [-f archive] ] [-o options] ...
      [-s replstr] ...  [-x format] [file ...]

pax  -r -w [-diklntuvX] [-p string] ...  [-s replstr] ...  [file ...]
      directory
```

`Pax` lukee ja kirjoittaa tiedostoarkistoja (*file archive*):

- Jos kumpaakaan optioista `-r` ja `-w` ei ole annettu, tulostaa arkiston sisällysluettelon (vrt. `tar t`). Jos *pattern* (*glob pattern* eli shellin jokerimerkit) on annettu, tulostaa sitä vastaavat tiedostonimet.
- Jos `-r` on annettu mutta ei `-w`:tä, purkaa arkiston sisällön (*pattern*ia vastaavan osan) työhakemiston alle, rakentaen tarvittaessa vastaavan hakemistopuun.
- Jos `-w` on annettu mutta ei `-r`:ää, rakentaa arkiston nimetyistä tiedostoista.
- Jos sekä `-w` että `-r` on annettu, kopioi nimetyt tiedostot kohdehakemistoon, kuten jos ne olisi ensin kasattu arkistoksi ja sitten purettu.

Jos tiedostoargumentteja ei ole annettu, `pax` lukee tiedostolistan stdinistä. Hakemistot tulkitaan rekursiivisesti hakemistopuiksi.

Edellämainittujen `-r` ja `-w` lisäksi `pax` tuntee seuraavat optiot:

`-b` *blocksize* Lohkokoko

`-c` etsii tiedostonimiä jotka *eivät* vastaa annettua *pattern*ia

`-d` ei käy hakemistoja läpi rekursiivisesti

`-f` *archive* käyttää arkistona nimettyä tiedostoa stdinin tai stdoutin asemesta

- `-i` kysyy interaktiivisesti mitä tiedostoille tehdään (ohita, vaihda nimeä, käytä sellaisenaan)
- `-k` ei kirjoita olemassaolevien tiedostojen päälle
- `-n` valitse vain yksi (ensimmäinen) tiedostonimi per *pattern*
- `-q` *options* järjestelmäkohtaisia optioita
- `-p` *string* joukko seuraavia optioita:

  - `a` älä säilytä käyttöaikoja (*access time*)
  - `e` säilytä omistaja, ryhmä, aikaleimat ja suojausbitit
  - `m` älä säilytä muutosaikoja (*modification time*)
  - `o` säilytä omistaja ja ryhmä
  - `o` säilytä suojausbitit

  Jos omistaja tai ryhmä vaihtuu, *setuid*- ja *setgid*-bitit nollataan.
- `-s` *replstr* muuta tiedostonimiä `ed`-editorin 's'-syntaksin mukaisesti: `/`*old*`/`*new*`/[gp]`, missä *old* ja *new* ovat ed-tyylisiä regexp'ejä, `g` tarkoittaa että kaikki esiintymät korvataan, `p` että tehdyt muutokset tulostetaan stderr'iin.
- `-t` palauta tiedostojen käyttöajat samoiksi kuin ennen kuin `pax` luki niitä
- `-u` ohita tiedostot jotka ovat vanhempia kuin samannimiset kohteessa
- `-v` pitkä tulostusmuoto sisällysluettelolle
- `-x` *format* valitse tallennusformaatti: `cpio` tai `ustar` (*extended tar*)
- `-X` älä ohita mount pointteja (ts. pysy yhden laitteen sisällä)

`Pax` on tarkoitettu korvaamaan vanhoja `cpio`- ja `tar`-komentoja, niistä kun on paljon erilaisia ja keskenään yhteensopimattomia versioita. Se ei kuitenkaan ole saavuttanut suurtakaan suosiota eikä sitä esim. Linux-järjestelmissä useinkaan ole.

## 5.49 pr - Print files

```
pr [+page] [-column] [-adFmrt] [-e[char][gap]] [-h header] [-i[char][gap]]
[-l lines] [-n[char][width]] [-o offset] [-s[char]] [-w width] [file ...]
```

muotoilee tekstiä tulostusta varten, oletuksena 66-rivisiksi sivuiksi, joissa on 5-rivinen otsikko jossa on sivunumero, päivämäärä, kellonaika ja tiedostonimi, ja 5 riviä tyhjää sivun alalaidassa.

Optiot:

- +*page* Aloita sivulta *page*

-*column* Muotoile tulostus *column* palstaiseksi

`-a` Palstoita riveittäin

`-d` Tuplarivinvälit (kaksi rivinvaihtoa joka väliin)

`-e[`*char*`][`*gap*`]` Aseta tabulaattorimerkiksi syötössä `char` (mikä tahansa ei-numero) ja -positioiksi $n * gap + 1$, missä $n = 1, 2, \ldots$

`-F` Käytä *formfeed*-merkkiä sivujen välissä

`-h` *header* Otsikkoon tiedostonimen paikalle *header*

`-i[`*char*`][`*gap*`]` Aseta tabulaattorimerkiksi tulostuksessa `char` ja -positioiksi $n * gap + 1$, missä $n = 1, 2, \ldots$

`-l` *lines* Aseta sivun pituudeksi *lines*.

`-m` Yhdistä tiedostoja rinnakkain, ts. eri sarakkeet eri tiedostoista.

`-i[`*char*`][`*width*`]` Rivinumerointi *width* numerolla, erottimena *char* (oletus tab).

`-o` *offset* Joka rivin alkuun *offset* välilyöntiä.

`-p` (ei POSIX.2:ssa, vain interaktiivisessa käytössä) pysähdy joka sivun jälkeen odottamaan näppäilyä

`-r` Ei virhettä tiedostoista joita ei saada avatuksi.

`-s[`*char*`]` Palstojen erottimena vain yksi *char* (oletus tab)

`-t` Ei otsikkoa eikä tyhjiä rivejä sivun eikä tiedoston loppuun

`-w` *width* Rivin leveydeksi *width* (vaikuttaa vain palstaleveyteen monipalstatulostuksessa)

Nykyisin `pr`-komennolle on suhteellisen vähän tarvetta, vastaavat toiminnot tehdään erilaisilla ASCII-to-PostScript- ohjelmilla.

## 5.50 printf - Write formatted output

Ks. 4.5.8.

## 5.51 pwd - Return working directory name

Tulostaa työhakemiston nimen. (Ksh:ssa se on myös muuttujassa `PWD`.)

### 5.52 read - Read a line from standard input

Ks. 4.5.4.

### 5.53 rm - Remove directory entries

`rm [-fiRr]` *tiedosto...*

poistaa tiedostojen hakemistotietoja (tiedostoon osoittavia linkkejä). Tiedoston sisältö hukkuu (sen käyttämä levytila vapautuu) kun viimeinen siihen osoittava linkki poistetaan (mikä saattaa epäonnistua jos kyseessä on käynnissä oleva ohjelma).

Optiot:

- `-f` ei virhettä olemattomista tiedostoista, poista kyselemättä.
- `-i` kysy varmistus käyttäjältä (stdinistä, vaikkei se olisikaan pääte). Jos sekä `-f` että `-i` on annettu, viimeinen ratkaisee. Oletuksena `-i` on päällä jos stdin on pääte ja yritetään poistaa hakemistoa.
- `-R` poista hakemistopuu rekursiivisesti
- `-r` sama kuin `-R`.

Tiedostoja `.` ja `..` ei voi poistaa.

### 5.54 rmdir - Remove directories

`rmdir [-p]` *hakemisto...*

poistaa tyhjiä hakemistoja (niiden hakemistotietoja, vrt. `rm`). Optiolla `-p` poistaa kaikki polun varressa olevat hakemistot (vrt. `mkdir -p`).

### 5.55 sed - Stream editor

Esitellään tarkemmin myöhemmin (ks. 7).

### 5.56 sh - Shell, the standard command language interpreter

Komento `sh` käynnistää uuden shellin. Sille voi antaa `set`-komennon yhteydessä (4.11) lueteltuja optioita (myös +-merkin kanssa) sekä seuraavia:

- `-c` *command* Suorittaa argumenttina annetun merkkijonon sisältämän komennon sen sijaan että lukisi komentoja stdinistä. Positioparametrit seuraavat sen jälkeen.
- `-i` Kertoo että shell on interaktiivinen.
- `-s` Lukee komentoja stdinistä (oletus ellei argumentteja ole).

Optioiden `-c` tai `-s` puuttuessa ensimmäinen optioita seuraava argumentti tulkitaan komentoja sisältäväksi tiedostoksi. Viivaa (`-`) ei tässä tulkita stdiniksi vaan se jätetään huomiotta. Loput argumentit asetetaan positioparametreiksi.

## 5.57 sleep - Suspend execution for an interval

`sleep` *seconds*

pysäyttää shellin suorituksen määräajaksi.

Jos `sleep` saa `ALRM`-signaalin, se lopettaa normaalisti (exit status 0, shellin suoritus jatkuu).

## 5.58 sort - Sort, merge, or sequence check text files

Komennolla `sort` voi tehdä kolme asiaa:

1. lajitella tiedostoja haluttuun järjestykseen;
2. yhdistää valmiiksi lajiteltuja tiedostoja; tai
3. tutkia onko tiedosto oikein lajiteltu.

Syntaksi:

`sort [-m] [-o` *output*`] [-bdfinru] [-t` *char*`] [-k` *keydef*`]...[`*file...*`]`

`sort -c [-bdfinru] [-t` *char*`] [-k` *keydef*`]...[`*file*`]`

Optiot:

- `-c` Tutkii onko (yksi) tiedosto oikein lajiteltu. Ei tulosta mitään, exit status 0 jos tiedosto on lajiteltu, muuten 1.
- `-m` (*merge*) Yhdistää tiedostot joiden on oltava valmiiksi järjestyksessä.

-o *output* Kirjoittaa tuloksen tiedostoon *output* stdinin asemesta. Tulostiedosto saa olla sama kuin syöttötiedosto (tai yksi niistä).

-u Tulostaa vertailussa samanarvoisista riveistä vain yhden. Option -c kanssa tarkistaa että duplikaatteja ei ole.

-d Vain tyhjät ja aakkosnumeeriset merkit otetaan huomioon ([[:blank:][:alnum:]]).

-f Pienet kirjaimet samaistetaan isoihin.

-i Vain näkyvät merkit otetaan huomioon ([[:graph:]]).

-n Numeerinen avain: kentän alussa olevat numeromerkit (mahdollisesti tyhjän jälkeen) tulkitaan numeroksi (sallittuja numerot, alussa miinusmerkki, desimaalipilkku, tuhansien erotin, riippuu LC_NUMERIC:stä). Tyhjä (tai ei-numeerinen) kenttä tulkitaan nollaksi.

-r Lajittelee laskevaan järjestykseen.

-b Ohittaa alkutyhjät avainkentissä. Ennen ensimmäistä -k:ta vaikuttaa kaikkiin avaimiin, muuten voi liittää erikseen jokaiseen -k:hon.

-t *char* Merkkiä *char* käytetään kenttien erottimena (jokainen merkitsevä; oletuksena erotin on tyhjä ja peräkkäisiä ei lasketa erikseen).

-k *keydef* Lajitteluavain (voi olla monta, merkitsevyysjärjestyksessä), muotoa
*alku*[*tyyppi*][,*loppu*[*tyyppi*]]
missä *alku* ja *loppu* määräävät avaimen alku- ja loppukohdan rivillä ja ovat muotoa *n*[.*p*], mikä tarkoittaa kentän n merkkiä p, numerointi ykkösestä alkaen, paitsi lopussa p==0 tarkoittaa kentän viimeistä merkkiä. Merkkimääritysten puuttuessa oletukset ovat alussa 1 ja lopussa 0. Tarkennin *tyyppi* on jokin kirjaimista b, d, f, i, n ja r ja vaikuttaa kuten vastaava optio mutta vain ko. avaimeen (b erikseen alkuun ja loppuun, muut koko avaimeen). Jos *loppu* puuttuu avain jatkuu rivin loppuun. Jos yhtään -k-optiota ei ole annettu, avaimena käytetään koko riviä. Erotinmerkkiä ei lasketa mukaan avainkenttään jos -t on annettu, oletuserotin kyllä.

Huomaa että optioiden järjestys on rajoitettu, erityisesti kaikki -k:t on annettava viimeiseksi (joissakin toteutuksissa optiot -bdfinr vaikuttavat vain itsensä jälkeen tuleviin (kaikkiin tai vain seuraavaan) avainmäärityksiin, tai mahdollisesti juuri edelliseen).

Kieliympäristö vaikuttaa kaikkiin lajitteluavaimiin, erityisesti LC_COLLATE mutta myös LC_CTYPE ja LC_NUMERIC.

Aika usein näkee yhä vanhaa syntaksia, jossa option -k asemesta avaimen alku- ja loppukohta annetaan muodossa +*alku* ja -*loppu*. Tällöin sekä kentät että merkit niiden sisällä numeroidaan nollasta alkaen ja *loppu* osoittaa ensimmäistä merkkiä lajitteluavaimen jälkeen. Siis +w.x -y.z on sama kuin -k w+1.x+1,y+1.z jos z>0 tai -k w+1.x+1,y.0 jos z==0.

Seuraavat epästandardit optiot ovat aika yleisiä:

- `-A` ASCII-vertailu (kieliympäristöstä riippumatta);
- `-T` *dir* Käyttää tilapäisenä hakemistona *dir*iä oletuksen asemesta (joka on yleensä `$TMPDIR`, `/var/tmp` tai `tmp`).
- `M` Vertailee kuukausien nimiä (riippuu `LC_TIME`:stä, tai sitten ei. . . ).
- `-z` *n* Riville varataan tilaa *n* merkkiä (tarpeen vain optioiden `-m` ja `-c` kanssa, jos tiedostossa on ylipitkiä rivejä).
- `-y [`*kmem*`]` Varaa *kmem* kilotavua muistia aluksi. Pelkkä `-y` tarkoittaa maksimia, `-y 0` minimiä.

Esim. listataan työhakemiston tiedostot kokojärjestyksessä:
`ls -l | sort -k 5,5n`

Esim. lajitellaan `/etc/passwd` ryhmän mukaan ja ryhmien sisällä tunnuksen mukaan:
`sort -t: -k 3,3n -k 1,1 /etc/passwd`

Esim. tutkitaan onko `/etc/passwd` järjestetty ID:n mukaan numerojärjestykseen:

```
if sort -c -t: -k 3,3n /etc/passwd
then echo "passwd properly sorted"
else echo "passwd out of order!"
fi
```

Esim. lajitellaan kaksi `passwd`-tiedostoa erikseen ID:n mukaan ja yhdistetään ne sitten:

```
sort -o passwd1.sorted -t: -k 3,3n passwd1
sort -o passwd2.sorted -t: -k 3,3n passwd2
sort -m -o passwd-both.sorted  -t: -k 3,3n passwd[12].sorted
```

## 5.59 stty - Set the options for a terminal

`stty [-a|-g]`
`stty` *asetuksia. . .*

tulostaa tai asettaa päätteen tai I/O-laitteen linjan tilan.

Optio `-a` tulostaa asetukset ihmisten luettavaksi tarkoitetussa muodossa, `-g` muodossa jonka voi myöhemmin antaa `stty`:lle argumenttina samojen asetusten palauttamiseksi (ja jossa kaikki erikoismerkit esitetään sellaisessa muodossa että niitä ei tarvitse suojata; muoto on järjestelmäkohtainen).

Mahdollisia asetuksia ovat (alkumiinus kääntää merkityksen):

[-]parenb: pariteettibitin generointi ja testaus päälle (pois)

[-]parodd: *odd parity* (even)

cs[5678]: merkkikoko bitteinä

kokonaisluku $n$: linjanopeus (baud rate)

ispeed $n$: syötön nopeus

ospeed $n$: tulostuksen nopeus

[-]hupcl: modeemilinjat vapautetaan kun laite suljetaan

[-]hup: sama kuin [-]hupcl

[-]cstopb: kaksi (-=yksi) stop bittiä

[-]cread: (laitteen) vastaanotto päälle

[-]clocal: oleta ettei linjalla ole modeemiohjauslinjoja

[-]ignbrk: ohita BREAK-signaali

[-]brkint: generoi SIGINTR BREAKista

[-]ignpar: jätä pariteettivirheet huomiotta

[-]parmrk: merkitse pariteettivirheet

[-]inpck: tarkista priteetti syötöstä

[-]istrip: nollaa pariteettibitti syötöstä

[-]inlcr: muunna rivinvaihto telanpalautukseksi syötössä

[-]igncr: ohita telanpalautus syötössä

[-]icrnl: muunna telanpalautus rivinvaihdoksi syötössä

[-]ixon: reagoi STOP- ja START-merkkeihin

[-]ixoff: generoi STOP- ja START-merkkejä

[-]opost: käsittele tulostus (vs. päästä raakana läpi)

[-]isig: reagoi kontrollimerkkeihin INTR, QUIT, SUSP

[-]icanon: kanoninen syötön tulostus, tulkitsee ERASE ja KILL merkit

[-]iexten: käsittele järjestelmäriippuvat kontrollimerkit (muut kuin START, STOP, ERASE ja KILL)

[-]echo: kaiuta näppäillyt merkit

[-]echoe: kaiuta ERASE (pyyhi merkki näytöstä jos mahdollista)

[-]echok: kaiuta KILL (pyyhi koko rivi näytöstä)

`[-]echonl`: kaiuta `NL` (mahd. `-echo`:sta huolimatta)

`[-]noflsh`: älä tyhjennä näppäinpuskuria `INTR`in, `QUIT`in tai `SUSP`in jälkeen

`[-]tostop`: lähetä `SIGTTOU` taustatulostukselle

`evenp|parity`: sama kuin `parenb cs7 -parodd`

`oddp`: sama kuin `parenb cs7 parodd`

`nl`: sama kuin `icrnl`

`-nl`: sama kuin `-icrnl -inclr -igncr`

`ek`: `ERASE` ja `KILL` oletusarvoiksi

`sane`: kaikki arvot "järkeviksi" oletusarvoiksi

`min` *n*, `time` *n*: ohjaavat merkkien puskurointia ei-kanonisessa syötössä

*control-char string*: Asettaa arvon jollekin seuraavista kontrollimerkeistä:

- `eof`: end-of-file
- `eol`: end-of-line
- `erase`: peruutus
- `intr`: pehmeä keskeytys (`SIGNIT`)
- `kill`: kova keskeytys (`SIGKILL`)
- `quit`: lopetus (`SIGQUIT`)
- `susp`: prosessin pysäytys (`SIGTSTP`)
- `start`: tulostuksen jatko (`IXON`)
- `stop`: tulostuksen pysäytys (`IXOFF`)

ja mahdollisia arvoja ovat normaalit merkit sekä `^` ja jokin merkki vastaavan kontrollimerkin symbolina (esim. `^r` on control-r). Lisäksi erityisesti `^-` ja `undef` tarkoittavat ko. kontrollimerkin kytkemistä kokonaan pois.

Toiminta on sangen järjestelmäriippuvaista, osa on mielekkäitä vain tietyntyyppisille laitteille jne, ja useissa järjestelmissä on myös muita kuin edellä lueteltuja asetuksia.

Huomaa että `stty` vaikuttaa stdiniin, joten laite valitaan tarvittaessa sen uudelleensuuntauksella.

Esim. jos modeemin laitetiedosto on `/dev/modem` ja sille halutaan linjanopeudeksi 38400 bps:

```
stty 38400 </dev/modem
```

### 5.60 tail - Copy the last part of a file

`tail [-f] [-c` *num* `| -n` *num*`]` `[`*tiedosto*`]`

tulostaa oletuksena 10 viimeistä riviä tiedostosta (tai stdinistä).

Optiot:

- `-c` *num* Tulostuksen aloituskohta tavuina: `+`*num* tarkoittaa rivejä tiedoston alusta laskien, `-`*num* tai etumerkitön *num* tiedoston lopusta, numerointi alkaen ykkösestä.
- `-n` *num* Tulostuksen aloituskohta riveinä, *num* kuten `-c`:llä
- `-f` Älä lopeta tiedoston loppuessa vaan odota jatkoa. Ei toimi putkien kanssa.

Vanha syntaksi: `tail [+-][`*num*`][c|l][f]` `[`*tiedosto*`]`
`+`*num*`c` on sama kuin uudessa `-c +`*num*,
`+`*num*`[l]` sama kuin `-n +`*num*
ja vastaavasti miinukselle, suffiksi `f` = `-f`.

Esim. halutaan tiedostosta rivit 3–5:

```
tail -n +3 "$file" | head -n 3

head -n 5 "$file" | tail -n -3
```

Jälkimmäinen on teoriassa nopeampi (miksi?).

Vrt. myös `sed -n 3,5p "$file"` ja `awk 'NR==3,NR==5' "$file"`.

### 5.61 tee - Duplicate standard input

`tee [-ai]` `[`*tiedosto...*`]`

kopioi stdinin (mahdollisesti useaan) tiedostoon.

Optiot:

- `-a` Kirjoittaa tiedosto(je)n loppuun eikä päälle
- `-i` Ei reagoi `SIGINT`-signaaliin

Esim. halutaan `/etc/passwd`-tiedostosta tunnukset, UID:t ja nimet sekä alkuperäisessä järjestyksessä että UID:n mukaan lajiteltuina:

```
cut -d: -f 1,3,5 /etc/passwd |
 tee list1 |
 sort -t: -k2,2 -o list1.sorted
```

### 5.62 test - Evaluate expression

Ks. 4.9.8.

### 5.63 touch - Change file access and modification times

`touch [-acm] [ -r` *referenssi* `| -t` *time* `]` *tiedosto...*

muuttaa tiedostojen aikaleimoja. Optiot:

- `-a` muuttaa käyttöaikaa (*access time*).
- `-m` muuttaa muutosaikaa (*modification time*).
- `-c` ei luo tiedostoa ellei se ole ennestään olemassa (eikä aiheuta virhettä).
- `-r` *referenssi* käyttää aikana tiedoston *referenssi* aikaleimoja kellonajan asemesta.
- `-t` *[[CC]YY]MMDDhhmm[.SS]* käyttää annettua aikaa kellonajan asemesta. Vuosiluvun CCYY puuttuessa olettaa kuluvan vuoden, vuosisadan CC puuttuessa oletus on 19 jos $YY >= 69$ ja 20 jos $YY <= 68$.

Jos kumpaakaan optioista `-a` ja `-m` ei ole annettu, tulos on sama kuin jos ne molemmat olisi annettu. Jos kumpaakaan optioista `-t` ja `-r` ei ole annettu, aikana käytetään suoritushetken kellonaikaa.

Vanha syntaksi: `touch [-acm] [`*MMDDhhmm[yy]*`]` *tiedosto...*
Huom. jos tiedoston nimi on tulkittavissa vanhanmalliseksi aikaleimaksi eikä -t tai -r -optioita käytetä, on tiedostonimen eteen syytä laittaa `--`.

Esim. asetetaan tiedoston `/tmp/x` luontiajaksi 19.1.2038 kello 04:53:
`touch -t 203801190453 /tmp/x`

Esim. etsittävä hakemistopuusta tiedostot joita on muutettu tänään puolenyön jälkeen:

```
touch -t $(date +%Y%m%d0000) /tmp/.marker$$
find . -type f -newer /tmp/.marker$$
rm /tmp/.marker$$
```

## 5.64 tr - Translate characters

`tr [-cs]` *string1 string2*
`tr -s [-c]` *string1*
`tr -d [-c]` *string1*
`tr -ds [-c]` *string1 string2*

muuttaa tai poistaa merkkejä stdinistä. Oletuksena (ilman optiota `-d`), `tr` korvaa jokaisen *string1*:n merkin *string2*:n vastaavalla merkillä. Merkkijonojen *string1* ja *string2* on tällöin oltava yhtä pitkiä (mahdollisten erikoismerkki- ym tulkintojen jälkeen).

Optiot:

`-c` käännä *string1* komplementikseen (aakkosjärjestyksessä) (kaikki merkit joita siinä *ei* esiinny).

`-d` poista valitut merkit

`-s` korvaa monta peräkkäistä samaa merkkiä yhdellä

Jos sekä `-d` että `-s` on annettu, *string1*:tä käytetään `-d`:n ja *string2*:ta `-s`:n kanssa.

Erikoismerkkejä voi esittää seuraavilla tavoilla:

*nnn* missä *nnn* on enintään kolmenumeroinen oktaaliluku

*c* merkkiä *c* vastaava kontrollimerkki (ks. taulukko alla)

*c*-*c* merkkiväli (ks. 4.6.2).

`[:`*class*`:]` merkkiluokka (ks. 4.6.2), levitetään aakkosjärjestyksessä. Sallittu vain *string1*:ssä, paitsi että siinä olevaa `[:lower:]` luokkaa vastaavalla paikalla *string2*:ssa saa olla `[:upper:]` ja päinvastoin.

`[=`*equiv*`=]` lajitteluekvivalenssiluokka (ks. 4.6.2), levitetään määrittelemättömässä järjestyksessä (järkevä vain `-d`:n tai `-s`:n kanssa). Sallittu vain *string1*:ssä ellei ole annettu molempia optioita `-d` ja `-s`.

`[`*x*`*`*n*`]` merkki *x* toistettuna *n* kertaa. Sallittu vain *string2*:ssa. Jos *n* puuttuu tai on nolla, se tulkitaan niin isoksi että se venyttää *string2*:n yhtä pitkäksi kuin *string1*. Jos *n* alkaa nollalla se tulkitaan oktaaliluvuksi, muuten desimaali-.

Kontrollimerkit, joille on *c*-esitys:

| merkkiyhdistelmä | kontrollimerkki | merkitys |
|---|---|---|
| \\\\ | | \\ |
| \a | control-G | <*alert*> |
| \b | control-H | <*backspace*> |
| \f | control-L | <*formfeed*> |
| \n | control-J | <*newline*> |
| \r | control-M | <*carriage return*> |
| \t | control-I | <*tab*> |
| \v | control-K | <*vertical tab*> |

Huom. `tr` ei huoli tiedostoargumenttia, tarvittaessa käytä uudelleensuuntausta.

Esim. Muutetaan åäö-kirjaimet a:ksi ja o:ksi:
`tr åäöÅÄÖ aaoAAO`

Esim. Vaihdetaan a:t ja b:t keskenään:
`tr aAbB bBaA`

Muutetaan pienet kirjaimet isoiksi:
`tr '[:lower:]' '[:upper:]'`

Muutetaan pienet kirjaimet isoiksi ja isot pieniksi:
`tr '[:lower:][:upper:]' '[:upper:][:lower:]'`

Poistetaan tiedostosta kontrollimerkit:
`tr -d '[:cntrl:]'`

Poistetaan kaikki näkymättömät kontrollimerkit (mutta ei rivinvaihtoa eikä tabulaattoreita):
`tr -dc '[:print:][:space:]'`

Poistetaan kaikki muut merkit kuin kirjaimet, numerot ja 'normaalit' välimerkit:
`tr -dc '[:alnum:] .,:;?!()\n'`

Poistetaan kaikki muut merkit kuin kirjaimet, numerot ja käytetyssä kieliympäristössä määritellyt välimerkit ja tyhjät:
`tr -dc '[:alnum:][:punct:][:space:]'`

Muutetaan MS-DOS- tai Windows-ympäristöstä peräisin oleva tekstitiedosto Unix-muotoon eli poistetaan telanpalautukset:
`tr -d '\r'`

Poistetaan moninkertaiset välilyönnit (korvataan yhdellä):
`tr -s ' '`

Poistetaan aksentit niiden vokaalien päältä joissa se ei vaikuta aakkosjärjestykseen:
`tr '[=a=][=e=][=i=][=o=][=u=][=y=][=A=][=E=][=I=][=O=][=U=][=Y=]' 'aeiouyAEIOUY'`

Muutetaan kaikki kokonaisluvut X:ksi:
`tr -s '0-9' '[X*]'`

Tulostetaan tiedostossa olevat sanat (kirjaimista koostuvat) yksi per rivi:
`tr -cs '[:alpha:]' '[\n*]'`

Huom. SystemV:n `tr` vaatii hakasulut merkkivälin ympärille, ja monet nykyisetkin versiot tulkitsevat ne siinä syntaksiin kuuluviksi (eivätkä muuta hakasulkuja). Tästä seuraa ongelmia jos hakasulkuja pitää muuttaa, esim.
`tr '[a-z]' '{A-Z}'`
on vaarallinen; parempi vaihtoehto on
`tr '[]a-z' '{}A-Z'`

## 5.65 true - Return true value

Palauttaa exit statuksen nolla.

Yleensä : ajaa saman asian ja voi olla nopeampi, mutta `true` voi olla luettavampi.

## 5.66 tty - Return user's terminal name

`tty [-s]`

Tulostaa stdiniä vastaavan laitetiedoston nimen, jos se on pääte. Optiolla `-s` ei tulosta mitään.

Exit status on 0 jos stdin on pääte, muuten 1. (Vrt. `test -t 0`).

## 5.67 umask - Get or set the file mode creation mask

`umask [-S] [`*mask*`]`

asettaa tai näyttää tiedostojen luonnissa käytettävät oletussuojaukset. Optiolla `-S` tulostus on symbolisessa muodossa, muuten oktaalilukuna.

Asetettaessa symbolisessa muodossa syntaksi on kuten `chmod`-komennossa ja ilmoittaa sallitut oikeudet, muut poistetaan (mutta sellaisia joita ei muutenkaan tulisi ei lisätä).

Oktaalimuodossa puolestaan annetaan ne oikeudet jotka poistetaan (ts. argumenttia käytetään bittimaskina suojausbiteille).

Esim. asetetaan oletussuojaus niin että kaikki voivat lukea ja suorittaa tiedostoja mutta vain omistaja voi muuttaa niitä:

```
umask 022
umask u=rwx,og=rx
```

Tai niin että vain omistaja voi tehdä yhtään mitään:

```
umask 077
umask u=rwx,og=
```

## 5.68 uname - Return system name

`uname [-amnrsv]`

tulostaa tietoja käyttöjärjestelmästä, ilman optioita sen nimen.

Optiot:

- `-m`: tulostaa laitteiston tyypin
- `-n`: tulostaa koneen nimen
- `-r`: tulostaa käyttöjärjestelmän pääversion (*release level*)
- `-s`: tulostaa käyttöjärjestelmän nimen
- `-v`: tulostaa käyttöjärjestelmätoteutuksen version
- `-a`: sama kuin `-mnrsv`

Esim. Linux-järjestelmistä usein puuttuu standardikomento `mailx`, mutta `mail` tuntee sen optiot, joten voidaan tehdä näin:

```
case "$(uname -s)" in
Linux) mailx() { mail "$@" ; } ;;
esac
```

## 5.69 uniq - Report or filter out repeated lines in a file

`uniq [-c|-d|-u] [-f` *fields*`] [-s` *chars*`] [`*infile* `[`*outfile*`]]`

poistaa oletuksena duplikaattirivit (peräkkäisistä identtisistä riveistä jättää yhden).

Optiot:

- `-c`: tulosta joka rivin alkuun sen esiintymiskertojen määrä
- `-d`: tulosta vain duplikaattirivit (kerran)
- `-f` *fields*: ohita ensimmäiset *fields* kenttää joka riviltä vertailussa. Kenttien erottimina käytetään tyhjää.
- `-s` *chars*: ohita ensimmäiset *chars* (kokonaisluku) merkkiä vertailussa.
- `-u`: älä tulosta duplikaattirivejä lainkaan

Vanha syntaksi: `-`*n* on sama kuin `-f` *n*, `+`*n* sama kuin `-s` *n*.

### 5.70 wait - Await process completion

`wait [`*pid...*`]`

odottaa lapsiprosessi(e)n päättymistä, ilman argumentteja kaikkien, muuten vain niiden joiden PID on annettu.

Exit status sama kuin viimeisenä annetun lapsiprosessin, tai jos se keskeytyi signaalin takia $signaalinumero + 128$, tai 0 jos argumentteja ei annettu eikä lapsiprosesseja ollut, tai 127 jos viimeistä annettua PIDiä vastaavaa prosessia ei ollut.

### 5.71 wc - Word, line, and byte count

`wc [-clw] [`*tiedosto...*`]`

laskee tiedostossa olevien tavujen, sanojen ja rivien määrät. Optiot:

- `-c`: tulosta tavumäärä
- `-l`: tulosta rivimäärä (rivinvaihtojen määrä)
- `-w`: tulosta sanamäärä (tyhjällä rajatut)

Jos mitään optioita ei ole annettu, tulostaa sekä rivit, sanat että tavut.

Jos tiedostoargumentteja on useita, tulostaa sekä tiedostokohtaiset tulokset että yhteismäärät.

Esim. Etsittävä rekursiivisesti `$HOME`:n alta 10 rivimäärältään suurinta .c-loppuista tiedostoa:

```
find ~ -type f -name '*.c' -exec wc -l {} \; | sort -n | tail
```

## 5.72 xargs - Construct argument list(s) and invoke utility

`xargs [-t] [-n` *num*`] [-s` *size*`] [`*command* `[`*args*`]]`

lukee tyhjällä tai rivinvaihdoilla eroteltuja argumentteja stdinistä ja suorittaa annetun komennon niin monta kertaa kuin tarvitaan että se saa kerralla enintään halutun määrän argumentteja.

Optiot:

- `-n` *num* käytä *num* argumenttia kerralla (tai vähemmän jos `-s` tulee ensin vastaan tai viimeisellä kerralla on vähemmän jäljellä)
- `-s` *size* käytä kerralla niin monta argumenttia kuin mahtuu *size* tavun mittaiselle komentoriville (vähemmän jos `-n` tulee ensin vastaan tai viimeisellä kerralla on vähemmän jäljellä)
- `-t` tulosta suoritettavat komennot ennen suoritusta stderr'iin
- `-x` lopeta jos `-n`:ää argumenttia ei saada mahtumaan `-s`:n (tai järjestelmän maksimin) mittaiselle riville

Jos kumpaakaan optioista `-n` ja `-s` ei ole annettu, vaikutus on kuin `-s` *ARG_MAX - 2048*.

Jos komento palauttaa statuksen 255 tai kaatuu signaaliin, `xargs` lopettaa siihen.

Tyhjiä argumenteissa voi suojata lainausmerkeillä (ei rivinvaihtoja) ja mitä tahansa voi suojata kenoviivalla (eroaa shellistä tässä!).

Exit status on 126 jos komento löytyi mutta sitä ei voitu suorittaa, 127 jos sitä ei löytynyt, 1-125 jos tapahtui muu virhe.

Erityisen tarpeen `xargs` on siksi, että jotkin komennot eivät ymmärrä kuin tietyn määrän argumentteja ja shellin komentorivilläkin on rajallinen pituus.

Esim. poistetaan kaikki `core`-nimiset tiedostot `/home`'n alta:
`find /home -type f -name core | xargs rm -f`
on paljon nopeampi kuin
`find /home -type f -name core -exec rm -f {}`

Olennaisesti saman tekee myös
`rm -f $(find /home -type f -name core)`
mutta jos tiedostoja löytyy paljon, tämä kaatuu.

Yleensäkin `...| xargs` on parempi kuin `for...in $(...)` aina kun argumenttien määrä vaihtelee ennustamattomasti. Erityisesti `for i in $(find...)` tai `for i in $(cat...)` on yleensä aina huono idea.

Jos argumenttikokoelma on valmiina tiedostossa tai luettavissa stdin'istä, `xargs` tarjoaa helpon tavan yksinkertaisen silmukan tekoon. Vertaa seuraavia:

```
for i in $(cat "$file") ;do
  ./myprog "$i"
done

while read line ;do
  ./myprog "$line"
done <"$file"

xargs -n 1 ./myprog <"$file"
```

Huom. `xargs` tulkitsee siis sekä rivinvaihtot että suojaamattomat välilyönnit argumenttien erottimiksi. Jos syöte on epäilyttävästä lähteestä, kannattaa olla varovainen.

# 6 *grep & regular expressions

## 6.1 Yleistä

Monet Unix-komennot tarvitsevat yleistä merkkijonovertailua, keinoa kysyä "kuuluuko merkkijono $x$ joukkoon $X$?" Tähän tarkoitukseen käytetään merkkijonomalleja, erikoismerkkejä sisältäviä merkkijonoja, jotka vastaavat tiettyjen sääntöjen mukaisesti joukkoa merkkijonoja. Yksi yleinen tällainen on *regular expression*, formaalien kielten teoriasta lainattu termi jolle ei ole oikein hyvää suomennosta.

Historiallisista syistä eri komennot käyttävät erilaisia ja yhteensopimattomia merkkijonomalleja:

- *glob pattern* (jatkossa vain GP) on jo esitelty (4.6). Sitä käyttävät `sh` tiedostonimien levittämiseen, merkkijono-operaatioihin (`${var...}` ja `case`-lauseeseen, ksh:n `[[...]]` sekä ulkoisista komennoista `find`.
- *basic regular expression* (jatkossa BRE) on historiallisesti vanhin ja yhä yleisin merkkijonomalli. Sitä käyttävät standardikomennoista `grep`, `ed`, `sed`, `expr` ja `pax`.
- *extended regular expression* (jatkossa ERE) on edellisen laajennus tai oikeammin vain uusi versio siitä. Standardikomennoista EREjä käyttävät `grep -E` ja `awk`.

Jotkin komennot käyttävät vielä edellisistä poikkeavia malleja, yleensä edellisiä jotenkin rajoitettuina, esim. `tr`. Huomionarvoinen on myös `case`-lauseen syntaksiin kuuluva `|`, joka olennaisesti laajentaa sen ymmärtämää GP:tä, vaikka muodollisesti onkin eri asia.

Lisäksi monet epästandardit komennot käyttävät omia versioitaan, lähinnä erilaisia laajennuksia edellisiin. Erityisesti ksh tuntee omat laajennuksensa GP:hen (4.6.4) ja Gnu-versiot standardikomennoista tuntevat erinäisiä *regexp*-laajennuksia, samoin `perl`.

## 6.2 grep - File pattern searcher

Komento `grep` ja käsite *regular expression* liittyvät kiinteästi yhteen, sen nimikin on lyhenne `ed`-editorin syntaksista: *Global/RegExp/Print*. Sen syntaksi on seuraavanlainen:

`grep [-E|-F] [-c|-l|-q] [-insvx] [[-e]` *pattern_list*`...] [-f` *pattern_file*`...]` `[`*file*`...]`

ja se etsii tiedostosta annettua mallia (-eja) vastaavia rivejä. Oletuksena malli (*pattern*) on *basic regular expression* (BRE), mistä enemmän alla, ja (vain) mallia vastaavat rivit (rivit joilla esiintyy mallia vastaava merkkijono) tulostetaan.

Optiot:

- `-E` Käytä laajennettua regexp syntaksia (*extended regular expression*, ERE).
- `-F` Etsi vakiomerkkijonoa (ei regexp).
- `-c` Tulosta vain löydettyjen rivien lukumäärä.
- `-e` *pattern_list* Etsittävät mallit (komentoriviargumentteina).
- `-f` *pattern_file* Lue etsittävät mallit tiedostosta, yksi per rivi.
- `-i` Samaista isot ja pienet kirjaimet.
- `-l` Tulosta vain niiden tiedostojen nimet, joista malli löytyi.
- `-n` Tulosta rivinumerot löydetyille riveille (kunkin tiedoston alusta ykkösellä alkaen).
- `-q` Älä tulosta mitään.
- `-s` Ei virhettä tiedostoista joita ei voi lukea tai joita ei ole.
- `-v` Tulosta vain rivit jotka *eivät* vastaa mallia.
- `-x` Tulosta vain rivit jotka kokonaisuudessaan vastaavat mallia.

Optioita `-e` ja `-f` voi antaa useita. Jos kumpaakaan ei ole annettu, ensimmäinen ei-optio argumentti tulkitaan malliksi. (Siten `-e` on tarpeen vain jos malleja halutaan antaa useita tai malli alkaa viivalla (vrt. `--`).) Kussakin tapauksessa useita malleja voidaan antaa erottamalla ne rivinvaihdolla (`grep` on tiukasti rivipohjainen, rivinvaihtoja sisältäviä malleja ei voi käyttää).

Jos tiedostoargumentteja on useita, `grep` tulostaa tiedostonimet kustakin tiedostosta löytyvien rivien (tai löydettyjen rivien lukumäärän optiolla `-c`) alkuun kaksoispisteellä erotettuna (optiolla `-l` aina vain tiedostonimet).

Exit status on 0 jos ainakin yksi mallia vastaava rivi löytyi, 1 jos ei, > 1 jos tapahtui virhe (paitsi optiolla `-q` nolla myös virheen tapahtuessa jos rivi löytyi).

Komennosta on historiallisesti myös versiot `egrep` ja `fgrep`, jotka toimivat kuten `grep -E` ja `grep -F`.

### 6.3 Basic Regular Expressions

BRE:t tulkitaan seuraavien sääntöjen mukaisesti:

- Tavalliset merkit (muut paitsi `\ . [ * ^ $`) vastaavat itseään.
- Kenoviiva (\) ja jokin erikoismerkeistä `\ . [ * ^ $` vastaa ko. merkkiä.
- Piste (`.`) vastaa yhtä mielivaltaista merkkiä.
- `[...]` (hakasulkulauseke) vastaa yhtä lausekkeen sisällön määräämään joukkoon kuuluvaa merkkiä. Hakasulkujen välissä olevat merkit tulkitaan seuraavasti:
  - Jos vasenta hakasulkua seuraa `^`, joukko tulkitaan seuraavien merkkien määräämän joukon komplementiksi. Muualla kuin heti `[`:n perässä sillä ei ole erikoismerkitystä.
  - Merkeillä `. * [ \` ei ole erikoismerkitystä hakasulkujen sisällä.
  - *char1*-*char2* tarkoittaa merkkejä jotka ovat >= *char*1 ja <= *char*2 (lajittelujärjestyksessä, riippuu `LC_COLLATE`:sta).
  - Oikea hakasulku edustaa itseään heti vasemman hakasulun tai sitä mahdollisesti seuraavan `^`:n jäljessä.
  - Tavuviiva edustaa itseään heti vasemman hakasulun ja sitä mahdollisesti seuraavien `^`:n ja `]`:n jäljessä. (Siis jos halutaan sekä `]` että `-` mukaan, edellinen ensin.)
  - `[:`*class*`:]` tarkoittaa luokkaan *class* kuuluvaa merkkiä. Tunnetut luokat ovat `[:alnum:]`, `[:alpha:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]` ja `[:xdigit:]` (samat kuin kohdassa 4.6.2).
  - `[=`*char*`=]` tarkoittaa merkkejä jotka ovat aakkosjärjestyksessä samanarvoisia merkin *char* kanssa. (Esim. `[=e=]` vastaa ranskassa (ja suomessa!) merkkejä e, é, è ja ë).
  - `[.`*symb*`.]` tarkoittaa lajitteluelementtiä joka vastaa symbolia *symb*. Käytetään lähinnä silloin kun monesta merkistä muodostuva lajitteluelementti pitää erottaa vastaavista erillisistä merkeistä. (Esim. espanjassa `grep 'a[[.ch.]x]b'` löytäisi jonot `'achb'` ja `'axb'`.)
- Usean peräkkäisen BRE:n yhdistelmä vastaa niitä vastaavien merkkijonojen yhdistelmää.
- BRE:n osa voidaan rajoittaa symbolipareilla `\(` ja `\)`. Tällainen rajaus ei vaikuta siihen mitä ko. BRE:n osa vastaa (paitsi ankkureiden `^ $` toiminta niissä on standardin mukaan "optional").
- *n*, missä *n* on numero `1-9`, vastaa järjestyksessä *n*:ttä edeltävää sulkulauseketta `\(...\)`. Esim. `\([0-9]\)\1` tarkoittaa samaa numeroa kahdesti peräkkäin.

- Tähti `*` tarkoittaa edeltävän merkin tai suluilla rajoitetun osalausekkeen tai osalausekeviittauksen (\\$n$) toistamista nolla tai useampia kertoja. Rivin alussa `*` edustaa itseään.

- `\{`$m$`[,`$n$`]\}` tarkoittaa edeltävän merkin, osalausekkeen tai osalausekeviittauksen toistamista $m$–$n$ kertaa (`\{`$m$,`\}` vähintään $m$, `\{`$m$`\}` tasan $m$).

- `^` BRE:n alussa ankkuroi sen alun, niin että se vastaa vain verrattavan merkkijonon alkua. (Esim. "`^ab`" vastaa "`abc`":tä mutta ei "`aab`":tä.) Muualla kuin BRE:n alussa `^` vastaa itseään.

- `$` BRE:n lopussa ankkuroi vastaavasti sen lopun. Muualla kuin BRE:n lopussa `$` vastaa itseään. Jos molempia ankkureita on käytetty, koko merkkijonon pitää täsmätä.

Erikoismerkit tulkitaan seuraavan prioriteettijärjestyksen mukaisesti:

1. `[= =] [: :] [. .]`
2. \\<erikoismerkki>
3. `[...]`
4. `\( \)` \\$n$
5. `*` ja `\{`$m$`[,`$n$`]\}`
6. yhdistely
7. `^` ja `$` ankkureina

Useasta mahdollisesta osamerkkijonosta BRE vastaa ensimmäistä (vasemmanpuoleisinta), jos niitä on useita pisintä niistä. Tällä ei ole merkitystä `grep`in kanssa mutta esim. `sed`in ja `expr`in kanssa kylläkin.

Kenoviivan vaikutusta muiden kuin edellämainittujen merkkien edellä ei ole määritelty, ja se vaihtelee järjestelmästä toiseen.

Huom. Rivinvaihdolla ei ole mitään erityisasemaa RE:ssä sinänsä, vaikka monet komennot moisen rajoituksen asettavatkin (mutta esim. `sed` ei).

Kaiken mitä voi tehdä GP:llä voi tehdä BRE:lläkin, mutta ei päinvastoin. Seuraavassa taulukossa on esimerkkejä niiden eroista:

| GP | BRE |
|---|---|
| `?` | `.` |
| `*` | `.*` |
| `a*` | `a.*` |
| `a.*` | `a\..*` tai `a[.].*` |
| `*[!0-9]*` | `.*[^0-9].*` |
| `^?^` | `\^.^` |
| `\?` | `?` |
| `\*a` | `*a` |
| `[0-9][0-9][0-9][0-9]` | `[0-9]\{4\}` |

Seuraavassa taulukossa on BRE:llä ja ksh:n GP:llä esimerkkejä malleista joita ei perus-GP:llä voi tehdä:

<table>
<tr><th>BRE</th><th>ksh GP</th></tr>
<tr><td><code>[-+]\{0,1}[0-9][0-9]*</code></td><td><code>?([-+])+([0-9])</code></td></tr>
<tr><td><code>[A-Za-z][A-Za-z0-9_]*</code></td><td><code>[A-Za-z]*([A-Za-z0-9_])</code></td></tr>
<tr><td><code>[0-9]\{1,3\}</code></td><td><code>[0-9]?([0-9])?([0-9])</code></td></tr>
<tr><td><code>\([[:alpha:]]\)[[:blank:].,]\1</code></td><td>(ei onnistu)</td></tr>
<tr><td>(ei onnistu)</td><td><code>?(Kalle|Pekka|Tapani)</code></td></tr>
</table>

Viimemainittu ei siis onnistu BRE:llä lainkaan, seuraavassa esiteltävällä ERE:llä kylläkin. Myös `case`-lauseella se onnistuu helposti.

## 6.4 Extended Regular Expressions

ERE:t tulkitaan muuten samoin kuin BRE:t paitsi:

- Erikoismerkkejä jotka pitää suojata kenoviivalla jos haluaa niiden esittävän itseään ovat `. [ \ ( ) * + ? { | ^` ja `$`. Oikeaa sulkua ei tarvitse suojata ellei vastaavaa vasenta sulkua ole lausekkeessa, muut on suojattava aina paitsi hakasulkulausekkeen sisällä (erityisesti `$ ^` ja `*` sijainnista riippumatta, toisin kuin BRE:ssä).
- Ankkurit `^` ja `$` toimivat myös osalausekkeissa.
- Osalausekkeet rajataan suluilla ilman kenoviivoja.
- Aaltosulkuja käytetään toistoon syntaksilla `{`$m$`[,`$n$`]}` ilman kenoviivoja.
- Osalausekeviittauksia (\\$n$) ei ole.
- Plus-merkki `+` tarkoittaa edeltävän merkin tai osalausekkeen toistamista yhden tai useampia kertoja.
- Kysymysmerkki `?` tarkoittaa edeltävän merkin tai osalausekkeen toistamista nolla tai yksi kertaa.

- Pystyviiva (putkimerkki) `|` kahden ERE:n välissä tarkoittaa jompaa kumpaa niistä. (Esim. `a(b|c)d` vastaa ”`abd`”:tä ja ”`acd`”:tä; `Pe(ikko|kka)` sanoja ”`Pekka`” ja ’’`Peikko`”.)

Erikoismerkkien prioriteetti on soveltuvin osin sama:

1. `[= =] [: :] [. .]`
2. \<erikoismerkki>
3. `[...]`
4. `( )`
5. `* + ?` ja `{`$m$`[,`$n$`]}`
6. yhdistely
7. `^` ja `$` ankkureina
8. |

Olennainen lisä BRE:hen on *tai*-operaattori `|` ja olennainen heikkous osalausekeviittauksen (\\$n$) puuttuminen. Seuraavassa esimerkkejä eroista:

| BRE | ERE |
|---|---|
| `[-+]\{0,1\}[0-9][0-9]*` | `[-+]?[0-9]+` |
| `[-+]\{0,1\}[0-9]\{1,10\}` | `[-+]?[0-9]{1,10}` |
| `\(/[a-zA-Z0-9]\{1,14\}\)\{1,\}` | `(/[a-zA-Z0-9]{1,14})`+ |
| `\([[:alpha:]]\)[[:blank:].,]\1` | (ei onnistu) |
| (ei onnistu) | `Kalle\|Pekka\|Tapani` |

Siis joitakin asioita voi tehdä vain BRE:llä ja joitakin vain ERE:llä. (Tosin esim. Gnu `grep` kyllä tuntee osalausekeviittauksen myös ERE:n kanssa.) Teoriassa ksh:n GP on yhtä ilmaisuvoimainen kuin ERE, käytännössä usein paljon hankalampi (erityisesti toisto-operaattorin puutteen vuoksi).

## 6.5 Esimerkkejä

Toisinaan käytettävä komento ratkaisee millaista merkkijonomallia on käytettävä, mutta `grep` tuntee kaksi erilaista ja usein saman asian voi tehdä monella eri komennolla jotka käyttävät erilaisia malleja. Seuraavassa esimerkkejä valinnan avuksi.

- Jos testattavana on vain yksi merkkijono ja testiin riittää GP, helpoin ja yleensä myös nopein tapa on `case` tai ksh:ssa `[[...]]`. Esim. onko tiedoston tarkennin `.c`:

```
case "$name" in
      *.c) ... ;;
esac

if [[ $name = *.c ]] # ksh
then ...

if expr "$name" : '.*\.c$' >/dev/null
then ...

if printf "%s\n" "$name" | grep -q '\.c$'
then ...

if printf "%s\n" "$name" | grep -q '[.]c$'
then ...
```

- Jos ehto on hieman monimutkaisempi, sitä ei useinkaan pysty GP:llä esittämään. Ehdon kääntäminen tai jakaminen useammaksi onnistuu usein, mutta se saattaa mennä hankalaksi. Yksittäisen merkkijonon tutkiminen BRE:llä on usein helpointa komennolla `expr` (muista implisiittinen ankkuri ^). Esim. Onko sanassa iso alkukirjain ja loput pieniä ja vain kirjaimia a-z:

```
case "$sana" in
  [!A-Z]*|?*[!a-z]*) echo BAD ;;
  *) echo OK ;;
esac

if [[ $sana = [!A-Z]* || $sana = ?*[!a-z]* ]]
then echo BAD
else echo OK
fi

if expr "$sana" : '[A-Z][a-z]*$' >&-
then echo OK
else echo BAD
fi

if printf "%s\n" "$sana" | grep -q '^[A-Z][a-z]*$'
then echo OK
else echo BAD
fi

if printf "%s\n" "$sana" | grep -qx '[A-Z][a-z]*'
then echo OK
else echo BAD
fi
```

- Jos tekstiä etsitään tiedostosta, `grep` on yleensä helpoin ratkaisu, vaikka malli olisikin yksinkertainen. Esim. haettava tiedostosta A-kirjaimella alkavat rivit:

```
grep '^A' "$file"

while read line
do case "$line" in
A*) printf "%s\n" "$line" ;;
   esac
done <"$file"
```

- Esim. etsittävä tiedostosta epätyhjät rivit:

```
grep -v '^$' "$file"

grep . "$file"
```

- Jos etsittäviä vaihtoehtoja on useita, ne voi antaa `grep`ille yhtenä monelle riville jaettuna argumenttina mutta se vaikeuttaa luettavuutta (ja aiheuttaa vaikeuksia joidenkin ei-POSIX-shellien kanssa). Yleensä on parempi käyttää useaa `-e` optiota tai `-E`:tä jolla ne saa yhdeksi:

```
grep 'Kalle
Pekka
Jussi' "$file"

grep -e Kalle -e Pekka -e Jussi "$file"

grep -E 'Kalle|Pekka|Jussi' "$file"

while read line ;do
  case "$line" in
    Pekka|Kalle|Jussi) printf "%s\n" "$line" ;;
  esac
done <"$file"
```

- Jos halutaan etsiä dynaamisesti luotua (esim. käyttäjältä saatua) merkkijonoa tai jos vain et ole varma erikoismerkkien suojauksesta, `grep -F` on turvallinen ratkaisu. Esim. tutkittava esiintyykö tiedostossa merkkijonoja `**.?[a]` tai `a[2*3]+4`:

```
grep -e '*\*\.\[a]' -e 'a\[2\*3]+4' "$file"

grep -F -e '**.?[a]' -e 'a[2*3]+4' "$file"
```

- Esim. etsitään laskutoimitusmerkkejä `+-*/`:

```
grep '[-+*/]' "$file"

grep -F -e + -e - -e '*' -e / "$file"
```

Huomaa merkkien järjestys hakasuluissa!

- Jos halutaan erottaa merkkijonosta osa erilleen, `grep` ei auta. Jos GP riittää voi käyttää shellin merkkijono-operaatioita, muuten `expr`iä (kyseeseen voi tulla myös `cut` tai myöhemmin esiteltävät `sed` ja `awk`, erityisesti jos halutaan käsitellä kokonainen tiedosto). Esim. `dirname` (ilman erikoistapauksia):

```
dirname=${name%/*}

dirname=$(expr "$name" : '\(.*\)/[^/]*')
```

- Esim. halutaan merkkijonon toinen merkki:

```
x=${mjono#?}
x="${x%${x#?}}"

x=$(expr "$mjono" : '.\(.\)')

x=$(printf "%s\n" "$mjono" | cut -b2)
```

- Jos halutaan samaistaa isot ja pienet kirjaimet, `grep -i` on yleensä helpoin ratkaisu, joskin ksh:lla on tähänkin oma keinonsa:

```
case "$nimi" in
  [kK][aA][lL][lL][eE]) echo "kalle se on" ;;
esac

[[ "$nimi" = [kK][aA][lL][lL][eE] ]] &&
  echo "kalle se on"

typeset -l tmp="$nimi"
[[ "$tmp" = kalle ]] && echo "kalle se on"

expr "$nimi" : "[kK][aA][lL][lL][eE]$" &&
  echo "kalle se on"

printf "%s\n" "$sana" | grep -iq '^kalle$' &&
  echo "kalle se on"
```

- Jos halutaan monimutkaisempia toistorakenteita, `grep -E` on yleensä selkein. Esim. tutkittava onko annettu merkkijono mahdollisesti etumerkillinen kokonaisluku:

```
case "$luku" in
  [!-+0-9]*|?*[!0-9]*|'') echo "not a number" ;;
  *)  echo "good number!" ;;
esac

case "${luku#[-+]}" in
```

```
  *[!0-9]*|'') echo "not a number" ;;
  *)  echo "good number!" ;;
esac

if expr "$luku" '[-+]\{0,1\}[0-9][0-9]*$' >&-
then echo "good number!"
else echo "not a number"
fi

if printf "%s\n" "$luku" | grep -q '^[-+]\{0,1\}[0-9][0-9]*$'
then echo "good number!"
else echo "not a number"
fi

if printf "%s\n" "$luku" | grep -qx '[-+]\{0,1\}[0-9]\{1,\}'
then echo "good number!"
else echo "not a number"
fi

if printf "%s\n" "$luku" | grep -Eqx '[-+]?[0-9]+'
then echo "good number!"
else echo "not a number"
fi
```

- Esim. Tutkittava onko merkkijono laillinen liukuluku, jossa voi olla desimaalipilkku ja eksponentti merkittynä isolla tai pienellä E:llä (desimaalipilkun vasemmalla puolella on oltava ainakin yksi numero):

```
expr "$luku" :\
'[-+]\{0,1\}[0-9]{\1,}.\{0,1}[0-9]*\([Ee][-+]\{0,1\}[0-9]\{1,\}\)\{0,1\}$' &&
echo "looks OK floating point number to me"

printf "%s\n" "$luku" |
grep -qix '[-+]\{0,1\}[0-9]{\1,}.\{0,1}[0-9]*\(e[-+]\{0,1\}[0-9]\{1,\}\)\{0,1\}' &&
echo "looks OK floating point number to me"

printf "%s\n" "$luku" |
grep -Eqix '[-+]?[0-9]+.?[0-9]*(e[-+]?[0-9]+)?' &&
echo "looks OK floating point number to me"
```

- Jos pitää ilmaista monimerkkisiä vaihtoehtoja lausekkeen sisällä, `grep -E` on usein ainoa järkevä vaihtoehto. Esim. etsittävä tiedostosta sanoja *agoraphobia* ja *sesquipedalophobia* sekä em. englanninkielisessä muodossa että kokonaan tai osittain suomalaisittain translitteroituina (*qu->k* ja *ph->f*):

```
grep -i 'agoraphobia
agorafobia
sesquipedalophobia
```

```
sesquipedalofobia
seskipedalophobia
seskipedalofobia' "$file"

grep -Ei 'agora(ph|f)obia
ses(qu|k)ipedalo(ph|f)obia' "$file"

grep -Ei '(agora|ses(qu|k)ipedalo)(ph|f)obia' "$file"
```

- Esim. etsitään tiedostosta sanaa "ei" erillisenä sanana, ts. niin että sen molemmin puolin on joko rivinvaihto tai jokin ei-kirjain:

```
grep -i '[^[:alpha:]]ei[^[:alpha:]]
^ei[^[:alpha:]]
[^[:alpha:]]ei$
^ei$' "$file"

grep -iE '(^|[^[:alpha:]])ei([^[:alpha:]]|$)' "$file"
```

- Esim. etsittävä työhakemistosta .tex-tarkenteiset tiedostot, joissa on vähintään 250 riviä:

```
wc -l *.tex |
while read lines file
do if [ $lines -ge 250 ] printf "%s\n" "$lines $file"
done

wc -l *.tex |
while read lines file
do case "$lines" in [3-9][0-9][0-9]|2[5-9][0-9]|[0-9][0-9][0-9][0-9]*)
printf "%s\n" "$lines $file" ;;
   esac
done

wc -l *.tex | grep -E '^ *([3-9][0-9]{2}|2[5-9][0-9]|[0-9]{4,}) '

wc -l *.tex | grep -E '[3-9][0-9]{2}|2[5-9][0-9]|[0-9]{4,}'
```

Viimeinen vaihtoehto listaa myös tiedostot joiden nimessä on numero $> 250$, mutta kelpaa interaktiivisesti tai jos muuten tiedetään ettei sellaisia ole.

- Jos tarvitaan osalausekeviittauksia, tarvitaan BRE. `Grep` on parempi jos käsitellään tiedostoa tai halutaan käyttää sen optioita kuten `-i`, `expr` jos testataan yhtä merkkijonoa tai halutaan erottaa siitä osa. Esim. etsitään tiedostosta rivejä jotka ovat muotoa `x="x"` tai `x='x'` (muuttujan arvoksi sijoitetaan sen nimi jonkinlaisissa lainausmerkeissä):

```
grep -x '\([a-zA-Z][a-zA-Z0-9_]*\)=\(["'\'']\)\1\2' "$file"
```

```
while read line ;do
  expr "$line" : '\(\([a-zA-Z][a-zA-Z0-9_]*\)=\(["'\'']\)\2\3\)$'
done <"$file"
```

- Jos tarvittava komento ei osaa halutunlaisia malleja, voi ongelman usein ratkaista yhdistelemällä komentoja. Esim. etsittävä hakemistopuusta tiedostoja joilla on sama tarkennin kaksi kertaa peräkkäin ('`koe.c.c`' tms):

  ```
  find . -type f | grep '\(\..*\)\1$'

  find . -type f | grep '\(\.[^.][^.]*\)\1$'
  ```

  Jälkimmäinen jättää pois tyhjät tarkentimet (kahteen pisteeseen päättyvät nimet).

- Esim. halutaan poistaa tiedostonimestä siinä mahdollisesti oleva tuplatarkennin:

  ```
  FNAME=$(expr "$FNAME" : "\(.*\(\..*\)\)\2")
  ```

- Jos vaihtoehtoisia malleja on paljon tai ne generoidaan dynaamisesti, on usein helpointa käyttää optiota `-f`. Esim. Tarkistettava että jokainen `/etc/passwd`:ssä oleva ryhmä on myös `/etc/group`issa:

  ```
  cut -d: -f3 /etc/group >groups
  cut -d: -f4 /etc/passwd | grep -vx -f groups

  cut -d: -f4 /etc/passwd | grep -vx "$(cut -d: -f3 /etc/group)"
  ```

## 6.6 Gnu extensions

Gnu-projektin komennot tuntevat joukon laajennuksia BRE/ERE-syntaksiin:

\ < sanan alku

\ > sanan loppu

`\b` sanan raja (alku tai loppu)

`\B` muu kuin sanan raja

`\w` kirjain tai numero (`[[:alnum:]]`)

`\W` sama kuin `[^[:alnum:]]`

`\|` BRE:ssä sama kuin `|` ERE:ssä

\ <, \ >, `\b` ja `\B` vastaavat nollamittaista merkkijonoa ao. paikassa, ts. toimivat ankkureina `^`:n ja `$`:n tapaan. Lisäksi osalausekeviittaukset toimivat myös ERE:ssä. (Siis Gnu ERE ja BRE ovat toiminnallisesti identtisiä, ainoa ero on syntaksissa.)

Gnu `grep` tuntee myös joukon epästandardeja optioita ja pitkiä synonyymejä standardioptioille. Niihin ei tässä puututa sen enempää.

# 7 Sed

## 7.1 Johdanto

SED (*Stream EDitor*) on ei-interaktiivinen (eräajo) editori. Se on hyödyllisin komentojonoissa, erityisesti putkitettaessa kun ensimmäisen ohjelman tulostus on melkein sellaisenaan sopiva toisen syötöksi, formaatti vain on hiukan väärä (kentät väärässä järjestyksessä tms).

Sed on toiminnaltaan rivipohjainen: se lukee yhden rivin työmuistiin (*pattern space*), käsittelee sen (soveltaa siihen annettuja editointikomentoja) ja tulostaa mitä jäi jäljelle, lukee seuraavan rivin jne. (Useamman rivin käsittely yhtaikaa, rivien siirtely jne. on mahdollista mutta vaatii erityistoimenpiteitä.)

Sed on erityisen kätevä paikallisluontoisten muutosten teossa. Globaaleihin operaatioihin, kuten suurten lohkojen siirtelyyn paikasta toiseen se ei niinkään sovellu. Yleensäkin haluttaessa muuttaa tiedostoa "paikalla", saattaa paremmin soveltua normaali editori (esim. *ed*(1)), joka lukee tiedoston muistiin editoitavaksi ja lopuksi kirjoittaa sen takaisin. (Useimmat editorit osaavat lukea komentoja standard inputista, jonka voi uudelleensuunnata. Niiden käyttö filttereinä ei sen sijaan yleensä onnistu.)

Sed ei osaa laskea mitään, se tuntee vain kaikkein alkeellisimmat kontrollirakenteet jne. Jos sed-ohjelma uhkaa paisua satariviseksi kannattaa harkita työkalun vaihtoa: varsinaisten ohjelmointikielten ohella usein tulee kyseeseen *awk*(1) tai *perl* (joka ei valitettavasti ole UNIXin standardikomento).

Sedistä (kuten UNIXistakin) on valitettavasti olemassa useita ei-aivan-yhteensopivia versioita; seuraavassa on enimmäkseen rajoituttu niiden suurimpaan yhteiseen osajoukkoon (POSIX-versioon).

## 7.2 sed-komento

Sed-komennon syntaksi on:
`sed [-n] [-e script...] [-f sfile...] [file...]`

missä

- `-n` aiheuttaa sen, ettei sed tulosta kuin ne rivit, jotka erikseen käsketään tulostaa; muutoin se tulostaa kaikki rivit, joiden tulostamista ei erikseen kielletä. Saman vaikutuksen saa aikaan myös `#n` ensimmäisen komennon alussa. Esimerkeissä oletetaan ettei `-n`:ää ole käytetty, ellei erikseen toisin mainita.
- `-e script` ilmoittaa, että merkkijono *script* sisältää editointikomennon. (Jos `-e`-optioita on vain yksi (eikä yhtään `-f`:ää), voi "`-e`":n jättää pois, ts. `sed 'komento'` on sama kuin `sed -e 'komento'`.)

- `-f` *`sfile`* ilmoittaa, että tiedosto *sfile* sisältää yhden tai useampia editointikomentoja.
- Optioiden jälkeen seuraa lista tiedostonimiä. Jos se on tyhjä, syötöksi otetaan standard input (jota vastaa myös tiedostonimi "`-`"). Jos tiedostoja on useita, ne käsitellään niin kuin ne olisi katenoitu yhdeksi (erityisesti myös rivinumerointi jatkuu tiedostosta toiseen).

Jos `-e` ja `-f` optioita on useita, ne yhdistetään järjestyksessä yhdeksi komentojonoksi. Komentojen kokonaismäärä on rajoitettu, yläraja riippuu versiosta mutta sen pitäisi olla sata tai suurempi.

Optioita *ei* voi yhdistää, esim. `-nf` ei toimi (toimii Gnu sedissä).

Tulos tulee normaalisti stdoutiin.

### 7.3 Osoitteet

Useimmilla komennoilla voi olla nolla, yksi tai kaksi osoitetta, jotka määräävät mihin riveihin ko. komentoa sovelletaan. Osoitteita on kahdenlaisia:

- *Numeerisia*, ts. rivinumeroita (1:stä alkaen); erikoistapauksena $ tarkoittaa viimeistä riviä.
- *Sisältöosoitteita*, ts. viitataan riveihin jotka sisältävät tietynlaisen merkkijonon. Näiden syntaksi on `/`*`merkkijonolauseke`*`/` tai `\x`*`merkkijonolauseke`*`x`, missä `x` on mielivaltainen merkki. (Merkkijonolausekkeista myöhemmin enemmän.)

Osoitteet tulkitaan seuraavasti:

- Jos osoitetta ei ole lainkaan: Käskyä sovelletaan kaikkiin riveihin.
- Yksi osoite: Käsky viittaa riveihin joita ko. osoite vastaa, numeerinen osoite täsmälleen ko. riviin, sisältöosoite kaikkiin riveihin, joilla annetunlainen merkkijono esiintyy.
- Kaksi osoitetta: Käsky suoritetaan alkaen ensimmäisen osoitteen osoittamasta rivistä ja päättyen jälkimmäisen määräämään riviin (molemmat mukaanluettuna).

Jos osoitteita on kaksi, ne voivat molemmat olla numeerisia tai sisältöosoitteita. Eri tapaukset tulkitaan seuraavasti:

- $n_1, n_2$: Rivit $n_1$–$n_2$. Jos $n_1 > n_2$ käskyä ei suoriteta koskaan (ilman virheilmoitusta).
- $n_1$,*lauseke*$_2$: Kaikki rivit $n_1$:stä alkaen ensimmäiseen sitä seuraavaan, jolla *lauseke*$_2$ esiintyy, tai tiedoston loppuun ellei sellaista löydy.
- *lauseke*$_1$,$n_2$: Alkaen ensimmäisestä rivistä, jolla *lauseke*$_1$ esiintyy, riville $n_2$, tai tiedoston loppuun jos *lauseke*$_1$ esiintyy ensimmäisen kerran rivin $n_2$ jälkeen.
- *lauseke*$_1$,*lauseke*$_2$: Alkaen riviltä, jolla *lauseke*$_1$ esiintyy, sitä lähinnä seuraavaan jolla *lauseke*$_2$ esiintyy, ja jos sen jälkeen esiintyy taas *lauseke*$_1$, jälleen siitä seuraavaan *lauseke*$_2$-riviin, jne.

Huom. Jos toinen osoite on sisältöosoite, käsky kohdistuu aina vähintään kahteen riviin (ellei tiedosto lopu ensin).

Osoitteiden merkityksen voi kääntää niiden perään sijoitetulla huutomerkillä (käskyä sovelletaan kaikkiin riveihin *paitsi* ...).

## 7.4 Merkkijonolausekkeet (*regular expressions*)

Sisältöosoitteissa ja korvaa-komennossa käytettävät merkkijonolausekkeet tunnistavat joukon erikoismerkkejä; tarkalleen minkä joukon riippuu versiosta (POSIXissa `Basic Regular Expression`), mutta ainakin seuraavien pitäisi kelvata kaikille:

**.** Edustaa yhtä mielivaltaista merkkiä, myös rivinvaihtoa työmuistin sisällä (muttei sen lopussa), ja erityisesti `.*` ei ylitä rivinvaihtoa. (Joissakin versioissa ei vastaa rivinvaihtoa koskaan.)

**^** Edustaa rivin alkua (kuviteltua nollamittaista merkkiä työmuistin alussa — jos työmuistissa on useita rivejä, *ei* vastaa rivinvaihtoa sen keskellä) ollessaan lausekkeen alussa (muualla normaali merkki).

**$** Edustaa rivin loppua (kuviteltua nollamittaista merkkiä työmuistin lopussa — ei rivinvaihtoa työmuistin keskellä) ollessaan lausekkeen lopussa (muualla normaali merkki).

**\n** Vastaa rivinvaihtomerkkiä työmuistin keskellä (ei sen lopussa).

***** Vastaa nollaa tai useampaa kopiota edeltävästä yhden merkin osalausekkeesta. (Joissakin versioissa toimii myös \(*lauseke*\):n perässä.)

**[*merkkejä* ]** Vastaa mitä tahansa hakasulkujen sisällä olevista merkeistä (tasan yhtä), paitsi jos ensimmäinen merkki siellä on ^, jolloin vastaa mitä tahansa merkkiä jota *ei* ole siellä. Lisäksi - kahden merkin välissä tarkoittaa väliä, esim. `[a-z]` vastaa kaikkia kirjaimia a–z. Jos joukkoon halutaan ], se on sijoitettava heti alku-[:n jälkeen. Muilla erikoismerkeillä ei ole erikoismerkitystä hakasulkujen sisällä (paitsi `\n` toimii).

\(***lauseke***\) Rajoittaa osan lausekkeesta niin että siihen voi viitata *numero*:lla.

\\$n$ missä $n$ on numero 1–9. Vastaa $n$:nnen sulkulausekkeen arvoa. Esim. ”`\([Kk][a-zA-Z]\)* \1`” vastaa k-kirjaimella alkavaa sanaa, joka esiintyy kahdesti peräkkäin yhdellä välilyönnillä erotettuna (ja molemmilla kerroilla joko isolla tai pienellä alkukirjaimella).

\\ Jos jotakin erikoismerkkiä halutaan käyttää normaalina merkkinä, sen eteen on lisättävä \\ (tai sen voi sijoittaa hakasulkuihin).

Näiden lisäksi ainakin seuraavat esiintyvät joissakin versioissa:

// Tyhjä lauseke vastaa lähinnä edellistä lauseketta.

\\< Sanan alku, ts. kohta, jonka oikealla puolella on kirjain, numero tai alaviiva ja vasemmalla jotakin muuta (myös rivin alku).

\\> Sanan loppu, kohta, jonka vasemmalla puolella on kirjain, numero tai alaviiva ja oikealla jotakin muuta (myös rivin loppu).

\\{$m$**[**,**[**$n$ ]\\}] Toistaa edeltävää merkkiä (tai osalauseketta) tasan $m$, (\\{$m$\\}), vähintään $m$ (\\{$m$,\\}) tai $m$–$n$ (\\{$m$,$n$\\}) kertaa. \\{0,\\} on sama kuin `*`.

(seuraavat toimivat lähinnä Gnu-sedissä)

\\+ tai vain +: Toista edellistä merkkiä (tai osalauseketta) yksi tai useampia kertoja. Sama kuin \\{1,\\}.

**?** tai vain **?**: Toista edellistä merkkiä (tai osalauseketta) nolla tai yksi kertaa. Sama kuin \\{0,1\\}. (Huom. se, että `+` ja `?` saattavat toimia \\:n kanssa tai ilman aiheuttaa ongelmia kun niiden halutaan esiintyvän normaaleina merkkeinä: Silloin ei auta muu kuin selvittää miten ne toimivat käsillä olevassa versiossa, tai panna ne hakasulkuihin.)

\\| ”tai”; vastaa oikealla *tai* vasemmalla puolellaan olevaa osalauseketta. Esim. `kissa\|koira` ja `k\(iss\|oir\)a` vastaavat sanoja ”kissa” ja ”koira”.

**w** Sama kuin `[a-zA-Z0-9]`.

**W** Sama kuin `[^a-zA-Z0-9]`.

**b** Sanan raja, ts. kohta, jonka toisella puolella on kirjain tai numero ja toisella ei (kummin päin tahansa).

**B** Sanan sisällä tai ulkopuolella, ts. kohta, jonka molemmilla puolilla joko on kirjain tai numero tai kummallakaan ei ole.

Jos lauseke täsmää tekstiin useammalla tavalla, sääntö on se, että se vastaa niistä vasemmanpuoleisinta, ja jos niitäkin on useita, pisintä.

## 7.5 Editointikomennot

### 7.5.1 Yleistä

Komentojen syntaksi on seuraavaa muotoa:

[osoite1[,osoite2]]käsky[argumentit]

Seuraavassa osoitteita ei ole merkitty näkyviin; käskyt hyväksyvät 0–2 osoitetta ellei toisin ole mainittu (jotkut käskyt hyväksyvät vain yhden osoitteen, jotkut eivät yhtään).

Yleensä jokainen käsky on omalla rivillään; niitä voi myös sijoittaa samalle riville puolipisteellä erotettuna (ei välttämättä toimi kaikissa versioissa). Jokainen erillinen `-e`-optio vastaa yhtä riviä.

### 7.5.2 Kokorivikomennot

**d** (*delete*): Poista rivejä. Suoritus jatkuu uuden rivin lukemisen jälkeen ensimmäisestä komennosta.

**n** (*next line*): Tulosta työmuisti (paitsi jos `-n` optio on voimassa) ja lue seuraava rivi. Suoritus jatkuu ensimmäisestä komennosta.

**a** (*append*): Tulosta teksti käsiteltävän rivin jälkeen. Lisättävä teksti on komentoa seuraavilla riveillä, ja jos sitä on useampia rivejä on kaikkien paitsi viimeisen loppuun lisättävä \\. Enintään yksi osoite.

**i** (*insert*): Kuten **a**, mutta teksti lisätään *ennen* käsiteltävää riviä. Enintään yksi osoite.

**c** (*change*): Kuten **a** ja **i**, mutta teksti *korvaa* ko. rivit.

Jos osoitteita on kaksi, ko. rivit poistetaan ja korvataan yhdellä kopiolla tekstistä.

Jos osoitteita on yksi, jokainen vastaava rivi korvataan tekstillä.

Jos osoitteita ei ole yhtään, koko tiedosto korvataan tekstillä.

*Esimerkkejä:*

- Poista rivit, jotka alkavat sanalla `REM`, isoilla tai pienillä kirjaimilla kirjoitettuna:

  ```
  /^[Rr][Ee][Mm]/d
  ```

- Lisää tiedoston loppuun rivi `*** The End ***`:

```
$a\
*** The End ***
```

- Korvaa kolme ensimmäistä riviä rivillä `ALKU`:

```
1,3c\
ALKU
```

### 7.5.3 I/O-komennot

Nämä komennot eivät vaikuta työmuistin sisältöön, ainoastaan tulostukseen.

**p** (*print*): Tulosta työmuistin sisältö stdoutiin. Huom. työmuistin sisältö tulostetaan normaalisti automaattisesti käsittelyn lopuksi, ellei `-n`-optiota ole käytetty; `sed 'p'` tulostaisi jokaisen rivin kahdesti.

**l** (*list*): Kuten **p**, mutta kontrollimerkit korvataan *nnn* oktaali-notaatiolla ja yli 72 merkkiä pitkät rivit katkaistaan. Rivin sisällä olevat `\n`:t merkitään \\illa rivin lopussa.

**w** ⟨***tiedosto***⟩ (*write to file*): Kuten **p**, mutta tulosta tiedostoon ⟨*tiedosto*⟩. **w**:n ja tiedostonimen välissä on oltava tasan yksi välilyönti.

**r** ⟨***tiedosto***⟩ (*read from file*): Kuten **a**, mutta lisättävä teksti luetaan tiedostosta ⟨*tiedosto*⟩. **r**:n ja tiedostonimen välissä on oltava tasan yksi välilyönti. Enintään yksi osoite.

Jos komentojonossa on useita **r** ja **a** komentoja, niiden lisäämät tekstit tulostetaan samassa järjestyksessä kuin komennot esiintyvät, sen jälkeen kun kaikki muut nykyriviin vaikuttavat komennot on suoritettu.

### 7.5.4 Korvaa-komento

Korvaa (*substitute*) komennon syntaksi on seuraava:

`s/`⟨*`merkkijonolauseke`*⟩*`/korvaava teksti/`*⟨*`liput`*⟩

minkä edessä voi taas olla 0–2 osoitetta.

Korvattavan merkkijonon (⟨*merkkijonolauseke*⟩) rajoittimina voi käyttää kauttaviivan asemesta mitä tahansa merkkiä paitsi välilyöntiä tai `\n`:ää.

Korvaava teksti voi sisältää mitä tahansa. Merkkijonolausekkeissa käytetyillä erikoismerkeillä ei ole erikoismerkitystä, ainoastaan rajoittimena käytetty merkki, \ itse, rivinvaihto sekä & (ks. alla) on suojeltava edeltävällä \\:lla.

Korvaavassa tekstissä voi käyttää löydettyä tekstiä seuraavasti:

`&` edustaa merkkijonolauseketta vastaavaa tekstiä kokonaisuudessaan.

$\backslash n$ , missä $n$ on jokin numero 0–9, edustaa $n$:nnen sulkulausekkeen sisältöä merkkijonolausekkessa.

*Liput:* **s**-komento tuntee seuraavat liput:

**g** : Korvaa kaikki esiintymät (oletuksena vain ensimmäinen joka riviltä korvataan).

$n$ : Korvaa vain $n$:s esiintymä joka rivillä, kun $n$ on kokonaisluku 1–512. (POSIX: ei ylärajaa $n$:llä.)

**p** : Tulosta rivi stdoutiin jos ⟨merkkijonolauseke⟩ löytyi (ja korvattiin).

**w** ⟨***tiedosto***⟩ : Kuten **p**, mutta tulosta nimettyyn tiedostoon stdout:in asemesta.

Nämä voivat esiintyä millaisina yhdistelminä tahansa, paitsi **p** ja **w** eivät kaikissa versioissa voi esiintyä yhtaikaa (ei myöskään useita **w**-optioita).

*Esimerkkejä*

- Muuta jokainen ”tex”, ”TEX” jne. muotoon ”TeX”:

  ```
  s/[Tt][Ee][Xx]/TeX/g
  ```

- Vaihda jokaisen rivin kahden ensimmäisen sanan järjestys, olettaen että sanat muodostuvat kirjaimista a–z ja niiden välissä on mahdollisesti pilkku, piste tai puolipiste (enintään yksi näistä) sekä välilyöntejä:

  ```
  s/^\([a-zA-Z][a-zA-Z]*\)\([,.; ] *\)\([a-zA-Z][a-zA-Z]*\)/\3\2\1/
  ```

- Lisää jokaisen ”defined”-sanan sisältävän rivin alkuun kolme huutomerkkiä:

  ```
  s/^.*defined/!!!&/
  ```

  *tai*

  ```
  /defined/s/^/!!!/
  ```

### 7.5.5 Muunna-komento

Muunna (*transform*) komento muuttaa yksittäisiä merkkejä toisiksi. Sen syntaksi on seuraavanlainen:

`y/`*`merkkijono1`*`/`*`merkkijono2`*`/`

Jokainen *merkkijono1*:ssä esiintyvä merkki korvataan *merkkijono2*:ssa vastaavalla paikalla olevalla merkillä. *merkkijono1*:n ja *merkkijono2*:n on oltava yhtä pitkiä. Muunnosta sovelletaan kuhunkin merkkiin vain kerran.

Esim.

```
y/ab/ba/
```

vaihtaa kaikki a-kirjaimet b:ksi ja päinvastoin;

```
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

muuttaa kaikki pienet kirjaimet isoiksi.

### 7.5.6 Monen rivin käsittely yhtaikaa

**N** Lue seuraava rivi työmuistiin sen nykyisen sisällön perään. Väliin jää yksi rivinvaihto (`\n`).

**D** Tuhoa työmuistin ensimmäinen rivi (alkuosa ensimmäiseen `\n`:ään saakka. Suoritus jatkuu ensimmäisestä komennosta. Jos työmuisti tyhjenee (siellä on on vain yksi `\n` lopussa), sinne luetaan seuraava rivi (toimii tällöin täsmälleen kuten **d**). Ei hyväksy yhtään osoitetta.

**P** Tulosta työmuistin ensimmäinen rivi (alusta ensimmäiseen `\n`:ään saakka). Ei hyväksy yhtään osoitetta.

Näitä käytetään yleensä ryhmittely- ja hyppykomentojen kanssa.

Yksinkertaisia monen rivin kuvioita voi toki etsiä muutenkin: Esim. seuraava etsii paikat, joissa sama sana esiintyy rivin lopussa ja seuraavan alussa, ja poistaa jälkimmäisen esiintymän.

```
N
s/\([a-zA-Z][A-Za-z]*\) *\n *\1/\1\n/
P
D
```

### 7.5.7 Apumuistin käyttö

Editointioperaatioihin käytettävän työmuistin lisäksi on käytettävissä erillinen apumuisti (*hold area*). Suorituksen alkaessa se on aina tyhjä.

**h** Kopioi työmuisti apumuistiin. Apumuistin aiempi sisältö tuhoutuu.

**H** Kopioi työmuisti apumuistiin sen nykyisen sisällön perään.

**g** Kopioi apumuisti työmuistiin. Työmuistin aiempi sisältö tuhoutuu.

**G** Kopioi apumuisti työmuistiin sen nykyisen sisällön perään.

**x** Vaihda työmuistin ja apumuistin sisällöt keskenään.

Esim. Siirrä tiedoston ensimmäinen rivi viimeiseksi:

```
1h
1d
$G
```

Käännä rivit päinvastaiseen järjestykseen (tulosta viimeinen ensimmäiseksi jne):

```
#n
G
h
$p
```

Huom. Apumuistin koko on rajoitettu (raja saattaa jopa olla melko pieni); tämäkin esimerkki todennäköisesti kaatuu, jos tiedosto on suuri.

### 7.5.8 Suorituksen ohjauskomennot

**!** "Älä": Käännä osoitteen merkitys, ts. seuraavaa komentoa sovelletaan kaikkiin riveihin *paitsi* edeltävää osoitetta (osoiteparia) vastaaviin.

{ } Ryhmittely: Samaa osoitetta sovelletaan joukkoon komentoja. Käytetään seuraavasti:

⟨*osoite1*⟩[, ⟨*osoite2*⟩]{[*komento1*]
*komento2*
...
}

Ryhmiä voi olla sisäkkäin (enintään 100 tasoa).

**:⟨*nimiö*⟩** Merkitsee kohdan johon **b**- tai **t**-komennolla voi hypätä. Enintään 7 merkkiä. Ei osoitetta.

**b⟨*nimiö*⟩** Ehdoton hyppy. Jos ⟨*nimiö*⟩ puuttuu, hyppää komentojonon loppuun.

**t⟨*nimiö*⟩** Ehdollinen hyppy. Hyppää jos yksikin korvaa-komento on onnistuneesti korvannut jotakin rivin lukemisen tai edellisen **t**-komennon jälkeen; ellei, suoritus jatkuu seuraavasta komennosta.

Ryhmittelyä voi käyttää mm. erilaisten osoiteyhdistelmien tekoon:

```
1,10{/^REM/d
}
```

poistaa REM-alkuiset rivit 10:n ensimmäisen joukosta;

```
#n
1,30{11,20!p
}
```

tulostaa rivit 1–10 ja 21–30.

Sisältöosoitteet testaavat työmuistin senhetkistä sisältöä; **b**-komentokin voi siten itse asiassa olla ehdollinen hyppy.

### 7.5.9 Sekalaisia komentoja

= Tulosta käsiteltävänä olevan rivin numero. Hyväksyy enintään yhden osoitteen.

**q** (*quit*): Lopeta editointi välittömästi. Työmuistin sisältö käsitellään kuin olisi hypätty komentojonon loppuun ja lopetetaan siihen (uutta riviä ei enää lueta). Enintään yksi osoite.

\# Kommentti: `#`:lla alkavat rivit (samoin tyhjät rivit) eivät vaikuta mitään. Poikkeuksena `#n` ensimmäisen rivin alussa, joka saa aikaan saman vaikutuksen kuin `-n`-optio.

## 7.6 Rajoituksia

Eri sed-versiot asettavat erilaisia rajoituksia ohjelmien koolle yms. Seuraavat rajat edustavat jonkinlaista minimiä (näihin mahtuvan pitäisi mennä läpi kaikista).

- Komentojen määrä: Enintään 100. (POSIX: ei rajaa.)
- Nimiöiden määrä: Enintään 48. (POSIX: ei rajaa.)
- Nimiöiden pituus: Enintään 7 merkkiä. (POSIX: 8.)
- Ryhmittely: Enintään 100 sisäkkäistä ryhmää. (POSIX: ei rajaa.)
- Komentorivin pituus: Enintään 4000 merkkiä. (POSIX: `LINE_MAX`.)
- Työmuisti: Enintään 4000 merkkiä. (POSIX: 8192.)

- Apumuisti: Enintään 4000 merkkiä. (POSIX: 8192.)
- Tiedostot: **w**-komennoilla voi kirjoittaa enintään yhdeksään eri tiedostoon. (POSIX: myös 9.)
- Alkutyhjät: Jotkut versiot poistavat välilyönnit ja tabit rivin alusta. (POSIX: tyhjät poistetaan vain osoitteiden ja komentojen edestä.)
- Merkkijonolausekkeet: Ainoat erikoismerkit, joiden voi luottaa toimivan samalla tavalla, ovat `. * [] ^ $ \( \) \`*numero* ja `\n`. (POSIX: *Basic Regular Expressions*.)

Käytettävä shell asettaa lisäksi omat rajoituksensa (komentorivin pituus on rajallinen, rivinvaihdon syöttäminen ei onnistu tms), jotka voi yleensä kiertää käyttämällä `-f`-optiota.

## 7.7 Esimerkkejä

- Tulostetaan tiedoston jokaiselta riviltä toinen merkki:

  ```
  sed 's/.\(.\).*/\1/' "$file"
  ```

  ```
  cut -b2 "$file"
  ```

- Tulostetaan tiedostosta joka toinen merkki:

  ```
  sed 's/\(.\)./\1/g' "$file"
  ```

- Tulostetaan tiedosto `h a r v e n n e t t u n a`:

  ```
  sed 's/./& /g'
  ```

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

  ```
  grep Bingo "$file" | head -n 1
  ```

  ```
  sed -n '/Bingo{p' -e q -e } "$file"
  ```

- Halutaan sh-skripti, joka toimii kuten `grep` (ilman optioita), mutta tulostaa vain mallia vastaavan osan kustakin rivistä jolta se löytyi, ei koko riviä:

  ```
  #! /bin/sh
  PAT=$1
  shift
  sed -n 's/.*\('"$PAT"'\).*/\1/p' "$@"
  ```

  Tuo ei toimi jos mallissa on `/`-merkkejä. Miten sen voisi korjata?

- Muutetaan tekstitiedosto html:ksi lisäämällä alku- ja lopputagit ja muuttamalla tyhjät rivit $< p >$:ksi:

```
sed '1i\
<html><head>\
<title></title>\
</head>\
<body>
s/^$/<p>/
$a\
</body>\
</html>' "$file" >"$file.html"
```

- Generoidaan `/etc/passwd`-tiedoston nimikentästä *gcos* sähköpostialiaksia tyyliin `etunimi.sukunimi`. Oletetaan että nimikentässä voi olla lempinimiä lainausmerkeissä (ja poistetaan ne), ääkkösiä voi olla sekä 'oikein' (ISO-88591) että 7-bittisinä ja jos etunimiä on useita, tehdään alias sekä kaikilla että vain ensimmäisellä. Alias-tiedoston formaatti on *alias : osoite*.

```
#! /bin/sh
cut -d: -f1,5 /etc/passwd |
sed -n '
/[a-z][^a-z0-9]\{0,7\}:/!{w bad
d
}
s/,.*//
s/".*"//
s/(.*)//
s/<.*>//
s/'"'.*'"'//
y/åäöÅÄÖ[\]{|}/aaoAAOAOAaoa/
s/'\''//g
s/\./ /g
s/  */ /g
/:$/d
/[^-A-Za-z0-9 :]/{w bad
d
}
s/ /./g
s/\(.*\):\(.*\)/\2 : \1/
p
s/\..*\././p
' >dot-aliases
```

Miten voisit tehdä saman ilman `cut`-komento, siis vain `sed`illä? (Huom. tuo ei tarkista kahden samannimisen esiintymismahdollisuutta eikä aliaksen mahdollista ennestään olemista tms.)

- Etsitään tiedostosta kaikki sh-tyyliset sijoituslauseet joissa ei ole lainausmerkkejä ja lisätään niihin sellaiset:

```
sed 's/^\([a-zA-Z][a-zA-Z0-9_]*=\)\([^"'\'']*[^ ]*\)/\1"\2"/' "$file"
```

- Tarvitaan skripti, joka vaihtaa kaikki `$x` -tyyppiset muuttuja- ja parametriviittaukset muotoon `${x}`, ei kuitenkaan kommenttiriveiltä:

```
#! /bin/sed -f
/^[:blank:]*#/n
s/$\([a-zA-Z][a-zA-Z0-9_]*\)/${\1}/g
s/$\([-*@0-9#?$]\)/${\1}/g
```

  Tässäkin voi käydä huonosti jos muuttujaviittaus onkin heittomerkkien sisällä.

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

```
sed 's/\(.*\) \([^ ][^ ]*\)/\2, \1/' "$file"
```

- Tiedostossa on kokonaislukuja pilkulla erotettuna. Korvataan toisessa sarakkeessa olevat negatiiviset luvut XXX:llä:

```
sed 's/^\([^,]*\),-[0-9]*/\1,XXX/' "$file"
```

- Tiedostossa on kokonaislukuja pilkulla erotettuna. Jos rivillä on ainakin kaksi negatiivista lukua, korvataan toinen niistä X:llä.

```
sed 's/-[0-9]*,/X,/2' "$file"
```

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

```
while read line; do
  printf "%s\n\n" "$line"
done <"$file"

sed 's/$/\
/' "$file"

sed 'a\
' "$file"

sed G "$file"
```

- Tarkistetaan että jokaisen .sh-tarkenteisen tiedoston alussa on `#!` -rivi ja lisätään `#!/bin/sh` niihin joissa sitä ei ole:

```
for i in *.sh ;do
  cp $i "$i.bak" &&
  sed -e 1{ -e '/^#!/!i\
#!/bin/sh' -e } "$i.bak" >"$i" &&
  rm "$i.bak"
done
```

- Oletetaan että sh-skriptissä on vain "siististi" kirjoitettuja funktioita (alku-`fun()` aina rivin alussa, samalla rivillä enintään `{`, ja loppu-`}` omalla rivillään). Kerätään sieltä funktiomäärittelyt tiedostoon `funcs` ja sijoituslauseet (myös funktioiden sisällä olevat) tiedostoon `assigns`:

```
sed -n '/^ *[A-Za-z][A-Za-z0-9_]* *( *) *{\{0,1\} *$/,/^ *} *$/w funcs
/^ *[A-Za-z][A-Za-z0-9_]*=/w assigns' "$scriptfile"
```

- Kerätään sh-skriptistä ne muuttujat, joihin sijoitetaan jotakin (ei koko sijoituslauseita) tiedostoon `sijm`:

```
sed -n 's/^ *\([A-Za-z][A-Za-z0-9_]*\)=.*/\1/w sijm' "$scriptfile"
```

- Kerätään sh-skriptistä ne muuttujat, joihin sijoitetaan jotakin ja tulostetaan ne samalle riville välilyönnein eroteltuina:

```
#! /bin/sed -f
s/^ *\([A-Za-z][A-Za-z0-9_]*\)=.*/\1/
tfound
bend
:found
H
:end
${g
s/\n/ /g
s/^ //
p
}
d
```

- Olkoon tiedostossa useita sh-skriptiesimerkkejä, jokaisen alussa `#!`-rivi, ja tiedoston alussa kommenttirivi, joka pitäisi kopioida jokaisen esimerkin alkuun:

```
sed '1h
/^#!/{G
s/\(.*\)\n\(.*\)/\2\
\1/
}'
```

- Olkoon sh-skriptissä taas vain siistejä funktioita. Halutaan sed-skripti joka lisää jokaisen funktion loppu-`}`:n perään kommenttina funktion nimen.

```
#! /bin/sed -f
/^ *[A-Za-z][A-Za-z0-9_]* *( *)/h
/^ *} *$/{
s/ *$/ #/
G
s/ *(.*//
}
```

- Oletukset kuten edellä. Lisätään skriptin loppuun kommenttilohko jossa on lueteltuna kaikki siinä määritellyt funktiot:

  ```
  #! /bin/sed -f
  /^ *#/n
  /^ *[A-Za-z][A-Za-z0-9_]* *( *)/H
  ${p
  g
  s/^\n/# Functions defined:&/
  s/([^\n]*/)/g
  s/\n/&# /g
  }
  ```

- Työtiedostossa on joukko tiedostoja joilla on sama tarkennin kahteen kertaan. Ylimääräiset tarkentimet pitäisi poistaa (ts. vaihtaa ao. tiedostojen nimet).

  ```
  ls | grep '\(..*\)\2$' |
  while read f ;do
     mv "$f" "${f%.*}"
  done

  ls | sed -n 's/\(.*\)\(\..*\)\2$/mv \1\2\2 \1\2/p' | sh
  ```

- Korvataan C-tiedoston alussa oleva kommenttilohko (alusta ensimmäiseen `*/`:n sisältävään riviin) uudella:

  ```
  sed '1,\x\*/xc\
  /*\
   * uusi alkukommenttiblokki\
   */' "$file"
  ```

- Erotetaan tiedostosta `malli.c` alkukommenttilohko ja kopioidaan se kaikkien sellaisten `.c`-tarkenteisten tiedostojen alkuun, joiden alussa ei ole kommenttilohkoa (ensimmäinen rivi ei ala `/*`:lla):

  ```
  sed '1,\x\*/xw alku.tmp' malli.c
  for i in *.c ;do
    head -1 "$i" | grep -q '^/\*' && continue
    cp "$i" "$i".bak &&
    sed '1r alku.tmp' "$i.bak" >"$i" &&
    rm "$i.bak"
  done
  ```

- 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.

```
sed 's/ \([^ ][^ ]* *\)$/|\1/
N
s/\n/|/' "$file" |
sort -t'|' -k2,2 -k1,1 |
sed 's/|/ /
s/|/\
/' >"$file".sorted
```

- Tiedostossa on vaihtelevanmittaisia tietueita ihmisistä: ensin henkilötunnus omalla rivillään ilman välilyöntejä, sitten vaihteleva määrä muuta tietoa sisennettynä ainakin yhdellä välilyönnillä. Tiedosto pitäisi lajitella syntymäajan mukaan. Oletetaan että tiedostossa ei ole '`|`'-merkkejä. (Huom. suomalaisessa (ja ASCII) aakkosjärjestyksessä `"+" < "-" < "A"`.)

```
sed ':loop
$!N
s/\n\(  *\)/|\1/
tloop
P
D' "$file" |
sort -k1.7,1.7 -k1.5,1.6 -k1.3,1.4 -k1.1,1.2 |
sed 's/|/\
/g' >"$file".sorted
```

### 7.8 Esimerkki: C-prototyyppien muodostus

```
# PROTO.SED - build ANSI-style C function prototypes from K&R-style source
# Usage: sed -f proto.sed [file(s)]
#
# Assumes function definitions begin at column 1.
# Comment delimiters inside strings aren't handled properly,
# nor can this cope with variable argument lists,
# and of course #if's and macros can cause no end of trouble.
#
# Tapani Tarvainen 10/16/89
#
# junk initial comments
/\/\*/{
:COM
# try deleting /* ... */
s-/\*.*\*/--
tNOCOMM
# didn't work, get another line
N
# junk up to newline (comments can be loooong ...)
s-/\*.*\n-/\*-
# t below jumps always, but b wouldn't clear substitution flag
tCOM
:NOCOMM
}
# look for lines beginning with a letter ('_' counts as a letter)
/^[a-zA-Z_]/{
# look for left parenthesis; anything but letter, digit, * or
# whitespace before it means this can't be function definition
:PAREN
/^[^(]*[^a-zA-Z0-9_*    \n(]/bQUIT
/(/bBRACE
N
# check for comments within
\-/\*-bCOM2
bPAREN
:COM2
s-/\*.*\*/--
tPAREN
N
s-/\*.*\n-/\*-
tCOM2
# then read up to {
:BRACE
# ); means this is a declaration or ready prototype, don't touch it
/)[     \n]*;/bQUIT
```

```
s/{/;/
tGOTALL
N
# check for comments again
\-/\*-bCOM3
bBRACE
:COM3
\-/\*.*\*/-tBRACE
N
s-/\*.*\n-/\*-
tCOM3
:GOTALL
# we’ve got it all (and no comments left)
# convert newlines and tabs to spaces
s/[\n   ]/ /g
# convert "int m, n;" into "int m; int n;" etc.
:DUPTYPE
s/\([);] *\)\([a-zA-Z_][a-zA-Z0-9_ ]* \)\([^,;]*\),/\1\2\3; \2/
tDUPTYPE
# match names before and after ) and move type specifiers inside parens
:MATCH
s/\((.*\)\([a-zA-Z_][a-zA-Z0-9_]*\)\(.*\))\([a-zA-Z0-9_*# ]*[* ]\2\);/\1\4\3)/
tMATCH
# prepend "int " to those still typeless
s/\([(,] *\)\([a-zA-Z_][a-zA-Z0-9_]*\)\( *[),]\)/\1int \2\3/g
# if param list is empty, insert "void"
s/(\( *\))/(\1void)/
# if function type is missing, insert "int"
s/^[a-zA-Z_][a-zA-Z0-9_]* *(/int &/
s/^static \([a-zA-Z_][a-zA-Z0-9_]* *(\)/static int \1/
# final polish: discard extra spaces &c
# (this is the place to change if you want more spaces &c in output)
s/  */ /g
s/( /(/
s/ ;/;/
# uncomment the next two lines to get each parameter to a line of its own
#s/\(,\) */\1\
#       /g
p
:QUIT
}
d
#
```

# 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
```