Unix shell scripting

Daniel C. von Asmuth

Inhoudsopgave

Dit is een uitgebreide inleiding over het gebruik van de Unix shell en bijbehorende hulpmiddelen om scripts te schrijven. Het veronderstelt enige kennis van de Unix commandoregel, bijvoorbeeld de bash-prompt-howto. Het bevat wat meer voorbeelden dan de officiële handleidingen.

1. Unix en de shell

De shell is de Unix 'commandostip', die opdrachten van de gebruikers inleest van het toetsenbord, ze uitvoert en het resultaat op het scherm zet. In de simpelste vorm bestaat een opdracht uit de naam van een programma; de shell zal dat programma dan opstarten, dat de beschikking over het toetsenbord en beeldscherm krijgt, en laat een nieuwe prompt zien als het programma beëindigd is.

Het Unix besturingssysteem wordt wel eens voorgesteld als bestaande uit een harde kern oftewel kernel, die de hardware aanstuurt en hardware aanstuurt en dat de shell een schil erom heen is, die de kernel van de gebruikers afschermt. Het is echter de taak van de kernel om de hardware af te schermen, terwijl de shell dient om toegang tot het systeem te geven. Gebruikersprogramma's hoeven geen gebruik van de shell te maken; de C-bibliotheek heeft bijvoorbeeld wel de functie van een schil om de kernel.

1.1. inleiding

Dit verhaal gaat ervan uit dat u enige ervaring met Unix hebt en dus met het typen van commando's in de shell. Het kan nooit kwaad om de uitgebreide informatie over een besproken commando op te vragen met man commando of (op GNU systemen) info commando.

Er zijn verschillende varianten op het Unix besturingssysteem; alle besturingssystemen die de hier besproken commando's bezitten worden hier voor het gemak met Unix aangeduid, ook al zijn ze niet gebaseerd op de originele code van A T & T. Op een Unix systeem zijn vaak verschillende shells te vinden. Het is ook mogelijk om een shell te draaien op Windows NT™. De voorbeelden bij dit verhaal zijn getest onder Linux en zullen op andere systemen soms aanpassing behoeven.

De oorspronkelijke `/bin/sh` is de Bourne shell, en de andere zijn daarvan afgeleid. De nummer twee was de C-shell, die handige features had om processen te besturen, maar minder handig om ermee te programmeren, en dus niet verder aan de orde zal komen. De nieuwere Korn shell lijkt weer meer op de Bourne shell, met de features van de C-shell, plus een hele reeks eigen uitbreidingen voor interaktief gebruik en programmeren. De Bourne Again Shell van het GNU projekt heeft eveneens een wijde verbreiding gekregen.

1.2. Mijn eerste scriptje

Het bijzondere aan de shell is dat hij zowel een handige commandoverwerker is voor interaktief gebruik als een programmeertaal. Verschillende delen van een Unix besturingssysteem bestaan uit shell scripts, omdat deze gemakkelijk door de gebruiker aan diens behoefte kunnen worden aangepast, wat vooral systeembeheerders waarderen. Shell scripts worden ook vaak gebruikt als een schil om een complexe applicatie.

Het is een kleine moeite om een korte reeks veel gebruikte commando's samen te voegen tot een scriptje: maak met vi (c.q. uw favoriete editor) een file `voorbeeld` aan met de volgende, vrij willekeurige, inhoud:

```
date
uname -a
who
```

Vervolgens typt u chmod ugo+rx voorbeeld waarmee het tekstbestand tot programma wordt gepromoveerd, dat met ./voorbeeld kan worden uitgevoerd. De kernel is zelf in staat te bepalen of een uit te voeren file een script of binair programma bevat.

In het voorbeeld is de directory waar het programma zich bevindt expliciet opgegeven. Als die wordt weggelaten zal de shell de lijst van directory's doorzoeken die in de speciale variabele $PATH staat. Het is enigszins riskant om de aktuele directory (.) in het zoekpad op te nemen.

Een standaard vergissing is om eigen programma's de naam `test` te geven. Helaas is test een ingebouwd commando van de shell. De shell heeft niet zoveel ingebouwde commando's: het meeste werk wordt door externe programma's gedaan.

1.3. hallo

Als volgende voorbeeld de nederlandstalige versie van het onvermijdelijke nutteloze programma hello, bekend van Kernighan & Ritchie.

```
#!/bin/sh
# dit programma zegt vriendelijk goedendag

echo Hallo, allemaal! ; exit 0
```

Dit lijkt al iets meer op een programma. In de eerste regel staat achter #! aangegeven welk programma het script moet uitvoeren. Normaliter is dat de shell waarmee u inlogt; u kunt hiermee voorkomen dat een programma niet werkt als het bijvoorbeeld vanaf de C-shell wordt gestart. Aan de andere kant zal een script dat begint met #!/bin/bash niet werken op een systeem waar die shell in `/usr/local/bin` staat of afwezig is. Deze eigenschap is vooral handig voor scripts in andere talen, bijv.:

```
#!/usr/bin/perl
print "Hallo, allemaal!\n"
```

De programmeertaal Perl wordt hier niet verder behandeld. Voor de rest geeft een hekje aan dat de rest van de regel kommentaar is. Een lege commandoregel is toegestaan. Behalve het einde van de regel kan ook een puntkomma worden gebruikt om opdrachten te scheiden die na elkaar worden uitgevoerd.

De opdracht echo is een ingebouwde functie van de shell, net als trouwens exit. Echo zorgt ervoor dat de resterende woorden op het standaard uitvoerkanaal (bijv. het scherm) worden afgedrukt. Shell commando's, argumenten en vlaggen moeten atlijd van elkaar worden gescheiden door spaties of andere scheidingstekens.

De opdracht exit beëindigt het script, ook als er nog opdrachten volgen; het gebruik ervan is niet verplicht. Net als de exit functie in een C programma wordt er een foutcode geretourneerd, die 0 bedraagt als er geen fout is opgetreden. Typ maar eens de volgende regels in achter de prompt.

```
true
echo $?
```

```
false
echo $?
```

De variabele $? krijgt de exit status van het laatst uitgevoerde commando of script. Na exit of het einde van een script gaat de shell verder met het aanroepende script of vraagt de gebruiker om invoer. True en false zijn externe programma's die niets doen en alleen in shell programma's nut hebben. Het resultaat verschilt nogal van Boolean types in andere programmeertalen.

Als u inlogt zal de shell eerst het script `/etc/profile` uitvoeren, waarin zich algemene instellingen bevinden, gevolgd door `.profile` in uw eigen directory, waarin u uw persoonlijke instellingen kwijt kunt. Bash en ksh kennen ook initialisatie files `~/.bashrc` en `~/.kshrc` die iedere keer uitgevoerd worden als een interactieve shell start, bijvoorbeeld bij het openen van een xterm.

1.4. een herhalingsoefening

De rest van dit hoofdstuk wordt besteed aan Unix en de shell, zonder dat er verder programmeren bij komt kijken. De behandeling zal erg oppervlakkig blijven; in Paragraaf 3 komen we er uitgebreid op terug.

Het centrale concept in Unix is de file oftewel het bestand. Files worden voortdurend gebruikt waar informatie voor min of meer lange tijd moet worden bewaard. Een file is een reeks bytes, die meestal op een schijf wordt bewaard. De belangrijkste operaties zijn het lezen of schrijven van een aantal bytes. Daarbij wordt de file pointer of bladwijzer om het aantal gelezen of geschreven bytes verplaatst. Verder is het mogelijk om de file pointer naar een opgegeven punt te verplaatsen en de lengte van het bestand in te krimpen tot bijvoorbeeld nul bytes.

De inhoud van een bestand ligt niet vast. Unix maakt vaak gebruik van tekstbestanden. Een byte staat dan voor een letterteken (karakter) van het ASCII alfabet. Het teken <LF> oftewel 0xa oftwel ^J oftewel \n geeft het einde van een regel aan. Er is geen teken nodig om het einde van een bestand te markeren, maar ^D kan worden gebruikt om invoer van de terminal te beëindigen.

Veel tools gaan ervan uit dat een bestand tekst bevat. Bijvoorbeeld zal cat voorbeeld de inhoud van het bestand over het scherm laten lopen. Als het bestand echter geen tekst bevat, dan kan de terminal de kluts kwijt raken van de stuurcodes; doe dus geen /bin/cat /bin/cat. Met cat -v loopt u geen gevaar.

In sommige Unices kunt u de schade repareren met het commando reset. De oude methode is de terminal uit en weer aan te zetten. De schade is te voorkomen. Het kommando file hallo zou iets moeten geven als `hallo: Bourne shell script text`.

Als file een frase met het woord 'executable' erin retourneert, hebben we te maken met een binair programma. Een binair programma kun je alleen uitvoeren: dat werkt wel een stuk sneller dan een ingewikkeld shell script. File kan zich ondanks zijn 'magic' uiteraard vergissen.

Verder zijn er speciale files, waarmee bijvoorbeeld apparaten kunnen worden afgelezen of beschreven alsof het files waren; een tape wordt bijvoorbeeld bediend als een hele lange pseudo-file, waarvan de grootte aan een vast maximum is gebonden. Deze worden vaak in de `/dev` directory gevonden.

Tekst die naar `/dev/tty` wordt geschreven, wordt wel zichtbaar gemaakt, maar niet opgeslagen. Een poging om `/dev/tty` uit te lezen retourneert niet de geschreven tekst, maar van het toetsenbord ingevoerde tekens. Unix zal u laten wachten totdat er een regel is ingevoerd, c.q. op de return-toets is gedrukt.

Om een tekstbestand uit te printen zou je cat hallo >/dev/lp0 of iets dergelijks kunnen doen, maar Unix staat dat enkel aan de user root toe. De juiste manier om een bestand te printen is via het printer spoolprogramma met lpr hallo

Een ander bestand dat in scripts gebruikt kan worden is `/dev/null`, ook wel de bittenbak genoemd, omdat alle data die er naartoe worden geschreven direkt worden weggegooid in tegenstelling tot de vuilnisbak van de Macintosh™. Lezen uit `/dev/null` is toegestaan, maar er zal niets uit komen.

De directory's kunnen worden beschouwd als een speciaal soort bestand, dat alleen met speciale opdrachten als ls, cp, mv, rm en ln kan worden gemanipuleerd. Directory's bevatten weinig anders als de namen van files en directory's. Daarom is voor een opdracht als rm -f ./voorbeeld schrijfpermissie op de aktuele directory vereist, maar geen permissie op het bestand. In scripts gebruiken we vaak de `-f` optie, zodat Unix niet nog eens vraagt of we het wel zeker weten.

Een file kan meerdere namen of links hebben; met het commando ln wordt een nieuwe link naar een bestaand bestand gelegd. ln /bin/ls /bin/dir maakt een nieuw commando, dir geheten, dat hetzelfde doet als ls. Met rm wordt het aantal links met één verminderd. Pas wanneer dat tot nul is gedaald en het bestand niet in gebruik is, zal Unix het bestand daadwerkelijk verwijderen.

Het is niet mogelijk naar een link te maken naar een directory of een bestand op een andere schijf. Deze beperking wordt opgeheven door de symbolic link of snelkoppeling, die je maakt met ln -s. Een snelkoppeling loopt het gevaar dat het bestand waar hij naar verwijst niet bestaat of dat een circulaire keten van links ontstaat. Vergelijk de uitvoer van ls -lL eens met ls -l.

```
-rwxr-xr-x 1 daniel users 769 Mai 29 example
-rwxr-xr-x 1 daniel users 769 Mai 29 voorbeeld

lrwxrwxrwx 2 daniel users   3 Jun 6 example -> wie
lrwxrwxrwx 2 daniel users   3 Jun 6 voorbeeld -> wie
```

1.5. processen en hun in- en uitvoer

We zagen al dat een programma in Unix wordt opgeslagen in een bestand. Een Unix proces is een lopend programma, met zijn data en de context waarin het draait zoals de huidige directory. Moderne besturingssystemen kunnen een proces verdelen in zgn. threads; en programma kan daarmee meerdere taken tegelijk uitvoeren, die hun data delen, wat al snel in een chaos kan ontaarden.

De shell zal, met uitzondering van ingebouwde opdrachten, commando's uitvoeren door er aparte processen voor te starten. Shell scripts worden doorgaans in afzonderlijke processen (subshells) uitgevoerd. Ondanks dat Unix' multi-tasking efficiënter werkt dan huis-, tuin- en keuken besturingssystemen, maakt het voortdurende maken en opruimen van processen shell scripts een stuk langzamer dan andere programmeertalen.

Als een extern programma is opgestart zal de shell gewoon wachten totdat hij een seintje ontvangt van de kernel dat het kindproces is gestorven c.q. beëindigd.

Door een 'ampersand' achter een commando te zetten als in voorbeeld&, zal het 'in de achtergrond' worden verwerkt, en de shell onmiddellijk om de volgende opdracht vragen.

Een voorbeeld van een ingebouwde opdracht is het commando cd. Beginners verbazen zich er soms over dat een cd opdracht binnen een shell script wel wordt uitgevoerd, maar het effekt ervan vergeten is als het script beëindigd is en terug keert naar de interactieve shell.

Commando's in een script kunnen ook worden uitgevoerd met bijv. . ./voorbeeld. Hiermee worden de opdrachten in het bestand voorbeeld binnen de lopende shell uitgevoerd en werkt cd bijvoorbeeld wel. Een minder gebruikte mogelijkheid is om programma's (niet alleen scripts) te starten met de opdracht exec ervoor. Er wordt dan geen apart proces gemaakt, maar het nieuwe programma wordt in het lopende proces geladen en vervangt het, zodat het oorspronkelijke programma na afloop niet verder kan gaan.

Tot de context van een proces behoren ook de standaard invoer-, uitvoer- en error-kanalen. Voor een interactieve shell verwijst de standaard invoer naar het toetsenbord en de uitvoer en error kanalen naar het beeldscherm. Als u bent ingelogd via een modem of netwerk, dan verwijzen deze kanalen naar uw scherm en toetsenbord in plaats van dat van de computer waarop de shell draait.

Met ./voorbeeld <data >resultaat worden de kanalen omgeleid zodat de inhoud van het bestand `data` zal worden gelezen en de uitvoer in het bestand `resultaat` komt. Eventuele foutmeldingen komen nog op het scherm.

De notatie ./voorbeeld <data > >resultaat 2>&1 & zegt dat het script in de achtergrond moet draaien. We zien toch geen resultaten. Het dubbele 'groter dan' teken geeft aan dat de uitvoer achter de bestaande inhoud van `resultaat` moet komen in plaats van het bestand eerst te wissen.

De aanduiding 2>&1 geeft aan dat het standaard error-kanaal (de tweede file-descriptor) een kopie is van nummer 1 (standaard uitvoer), dus foutmeldingen en resultaten verschijnen door elkaar heen in het bestand `resultaat`; het standaard-invoerkanaal heeft file-descriptor nummer 0. Kind processen erven file-descriptors (open bestanden); de standaard kanalen zijn altijd open. Als de standaard kanalen zijn omgeleid naar bestanden zijn toetsenbord en beeldscherm toch te benaderen door om te leiden naar het pseudo-bestand `/dev/tty`.

Een here-document is een bijzonder geval van omleiding waarin de te verwerken gegevens in het script-bestand zelf staan.

```
fox=dog; dog=fox
cat <<woord
The quick brown $fox jumps over the lazy $dog.
...
woord
```

De rest van het script tot aan woord wordt dan als invoer gebruikt voor het commando (bijv. cat). Het woord dat als markering dient moet letterlijk worden herhaald op een aparte regel. De shell zal wel eventuele substituties uitvoeren op de data.

1.6. een les over 'ls': jokertekens

Het volgende commando is een eenvoudige vervanging voor ls.

```
echo * | tr  " " "\n" | column
```

Het illustreert hoe de shell jokertekens of wildcards behandelt. Als er in een woord een jokerteken voorkomt, dan zal de shell dat woord vervangen door alle bestandsnamen in de huidige directory die overeenkomen met dat zoekpatroon.

De `*` staat voor iedere reeks van nul of meer tekens, dus ls a*z geeft de lijst van alle bestanden waarvan de naam begint met een a en eindigt op z, zoals 'alcatraz' (onder DOS/Windows kunt u de * alleen aan het eind gebruiken). Als er geen corresponderende bestanden zijn dan blijft de asterisk staan, bijv. echo Wie*dit*leest*is*gek

Het is mogelijk te voorkomen dat jokertekens worden vervangen door ze tussen aanhalingstekens te zetten of door er een backslash (\) voor te zetten; ls * komt van pas, want een bestand met de naam `*` is niet uitgesloten.

Een . corrrespondeert met een willekeurig teken, zodat de shell `p?n` kan vervangen door `pan pen pin pon`. Een reeks van tekens tussen rechte haken kan worden vervangen door één van die tekens, bijv. ls fig-[123456789].jpg door `ls fig-1.jpg fig-2.jpg`

Let erop dat ls * ook de inhoud van subdirectory's weergeeft. ls geeft alleen files in de huidige directory en ls */ geeft de inhoud van alle subdirectory's.

1.7. pijpleidingen

Twee processen kunnen gemakkelijk via een pijp gekoppeld worden, zodat data die uit de standaard uitvoer van het eerste proces komen, door de standaard invoer van het tweede proces worden verwerkt, bijv.: ps -ef | sort

Het eerste commando geeft een lijst van alle processen die momenteel draaien en het sort commando sorteert ze (op de inlognaam van de gebruiker; werkt helaas niet op alle Unix versies gelijk). Als in een pijp een commando wordt gebruikt dat een filenaam als argument nodig heeft, dan kan meestal '-' worden gebruikt om de standaard invoer aan te duiden; veel commando's gebruiken automatisch de standaard invoer als ze geen bestandsnaam meekrijgen.

Een programma dat data van het standaard invoerkanaal leest en na een eenvoudige bewerking naar het standaard uitvoerkanaal schrijft noemen we een filter. Het eenvoudigste filter is cat, dat data onveranderd kopieert; sort is ingewikkelder.

Een ander filter is dd;. Voor details zie Paragraaf 3.18; hier volgt een voorbeeld hoe je een bestand kunt converteren naar hoofdletters: dd conv=ucase <voorbeeld >gesorteerd 2>/dev/null

Dd wordt onder Unix wel gebruikt om een floppy disk te kopiëren met dd if=/dev/fd0 of=image-file Vervolgens verwissel je de disks en kopiëert de image file terug naar floppy.

Vergelijk het resultaat van find /bin | sort eens met ls -1 /bin/* en u ziet dat ls de gewoonte heeft de uitvoer te sorteren. Bij gebruik van een pijp wordt de uitvoer van het eerste proces opgeslagen in een kleine hoeveelheid buffergeheugen. Als dat vol is, wordt de producent stilgezet totdat de consument deze data heeft verwerkt.

Om de uitvoer van een commando op het scherm te bekijken en tegelijk een kopie in een bestand te bewaren gebruikt men iets als ./voorbeeld 2>&1 | tee resultaat

Als u beschikt over het tooltje netcat (http://199.103.168.8:4984/web1/hak/netcat.html) dan kunt u ook op eenvoudige wijze gegevens over een netwerk versturen. In plaats van prog1 | prog2 start u op machine *`host`* netcat -v -l -p 1234 | `prog2` en op de andere machine doet u `prog1` | netcat host 1234 Hierin is `1234` de gebruikte IP poort: een min of meer willekeurig nummer, mits de betreffende poort nog ongebruikt is, en `prog1` en `prog2` zijn willekeurige commando's. Netcat fungeert hierin als een pijp tussen processen op verschillende computers.

Alle Unix versies kennen de bovengenoemde anonieme pijp; sommige kennen ook een pijp die als een speciale file in het bestandssysteem voorkomt. Een pijp kan worden aangemaakt met mkfifo pijp of mknod pijp p Het mknod commando wordt ook gebruikt om speciale files van het block of character type aan te maken. Probeer maar eens welke output het file commando op een speciale file geeft.

2. elementair shell programmeren

In dit artikel wordt de shell geprogrammeerd met behulp van script files. Moderne shells bezitten bovendien de mogelijkheid om korte macro's te definiëren met de alias opdracht bijv. alias l='ls -alg '

Aliassen kunnen ingewikkelde commando's vervangen door eenvoudiger te onthouden namen. Ze worden vaak gedefinieerd in `~/.profile`, een shell script dat elke keer als iemand inlogt wordt uitgevoerd. Met unalias wordt de definitie weer verwijderd.

2.1. variabelen

De werking van de shell is vrij ingewikkeld. Er zijn verschillende manieren waarop de shell reeksen tekens zal vervangen door andere. Een daarvan is de expansie van aliassen: het kommando l ~/*/a* kan worden uitgeschreven tot bijvoorbeeld **`/bin/ls -alg /home/gast/agf/aardappel /home/gast/agf/aardbei /home/gast/agf/appel /home/gast/autos/audi`**

De alias kwamen we zojuist tegen; aan 'ls' is hier het pad toegevoegd. De tilde staat voor de home directory, waarop de gebruiker 'gast' inlogt; de uitdrukking 'a*' wordt door de shell vervangen door een lijst van filenamen die met een a beginnen.

Een variabele is een naam, die door de shell aan een reeks tekens wordt gekoppeld met bijv. L='ls -alg '

Het kommando set zonder argumenten geeft een lijst van alle op dat moment gedefinieerde variabelen met hun waarden. De shell heeft zelf een aantal ingebouwde variabelen en andere variabelen worden bij het inloggen gedefinieerd, zoals $HOME, de home directory van de gebruiker en $PWD, de huidige directory.

We zijn ook al de variabele $PATH tegengekomen, ook wel het zoekpad genoemd. De waarde van $PATH is een lijst van directory's, gescheiden door dubbele punten. Als een commando dat geen slash (/) in de naam heeft, niet is ingebouwd in de shell of als alias is gedefinieerd, wordt gezocht of het commando overeenkomt met de naam van een file in een van deze directory's.

Hetzelfde kunt u doen met het which commando, dat echter niet op alle Unix systemen bestaat, of waarvoor soms een alternatief whence voor bestaat. Voorbeeld: which find

De shell zal een dollarteken gevolgd door de naam van een variabele vervangen door de waarde van de betreffende variabele. Nu kunnen we hetzelfde effekt als voorheen krijgen met $L.

Hadden we echter L de waarde 'l' gegeven, dan zou het resultaat geweest zijn: `l: command not found` Bash en ksh brengen ons in dit geval verder met eval $L Hiermee wordt de uitdrukking 'l' nog eens geëvalueerd, nadat de shell de waarde van L heeft gesubstitueerd. Een opdrachtregel als $apenkool zal niets doen en ook geen foutmelding opleveren als de betreffende variabele niet gedefinieerd is.

We krijgen de waarde van de variabele L terug met echo $L. Het dollarteken maakt eigenlijk geen deel uit van de naam. Als we nu een script of ander programma aanroepen waarin de waarde van variabele L gebruikt wordt, zal die echter ongedefinieerd zijn, tenzij we eerst de opdracht export L geven om een lokale variabele te exporteren naar de programma-omgeving. Echo en export zijn ingebouwde functies van de shell.

2.2. aanhaling

In de vorige paragraaf werd de waarde die aan L werd toegewezen omgeven door enkele aanhalingstekens. Deze zijn nodig om van de uitdrukking één woord te maken, inclusief de spaties. Dubbele aanhalingstekens hadden ook voldaan. Het verschil is, dat binnen dubbele aanhalingstekens de variabelen nog steeds worden vervangen door hun waarde, en alleen tekst tussen enkele aanhalingstekens letterlijk wordt overgenomen.

Tussen dubbele aanhalingstekens bestaat nog de mogelijkheid om aan te geven dat een enkel teken exact moet worden gekopiëerd; dit gebeurt door er een backslash voor te zetten als escape symbool.

De programmeertaal C definieert een aantal escape symbolen, die ook in de shell kunnen worden gebruikt.

`\a` (alert)

<BELL> (laat een pieptoon horen)

`\b` (backspace)

<BS> verplaatst de cursor een stap terug. ('x\bx' kan worden gebruikt om een x vet af te drukken.)

`\f` (formfeed)

<FF> geeft een nieuwe pagina

`\n` (newline)

<LF> geeft een nieuwe regel

`\r` (carriage return)

<CR> verplaatst de cursor terug naar het begin van de regel

`\t` (tabulator)

<TAB> verplaatst de cursor naar de volgende tab-stop

`\t` (vertical tab)

<VT> verplaatst de cursor een stap omlaag

Verder kunnen ASCII tekens worden aangeduid in octale notatie, bijvoorbeeld \007 (ook \a) voor het <BELL> teken.

Hierbij zij aangetekend dat de echo opdracht van de shell van zichzelf een <LF> (regeleinde) toevoegt, tenzij er echo -n is gebruikt; gebruikers van bash moeten echo de `-e` vlag meegeven om de escapes te laten werken. Het resultaat kan verschillen als er oktale codes boven 0200 worden gebruikt.

Een speciale functie van de shell is de commando substitutie, aangegeven door achterwaartse aanhalingstekens, bijv. L=‘l‘ Hier wordt een aparte subshell gestart, waarin het commando ls -alg wordt uitgevoerd; het resultaat is weer te zien met echo "$L"

Probeer nu maar eens wat er gebeurt als de dubbele aanhalingstekens worden weggelaten. Een alternatieve notatie is om de te substitueren opdracht tussen `$(` en `)` te zetten. Daarmee kunnen gesubstitueerde processen bovendien genesteld worden. Let op het verschil tussen de notaties $(lijst;van;commando's) en (lijst;van;commando's): beide voeren de opgegeven opdrachten uit in een aparte subshell, maar waar de uitvoer van de tweede variant gewoon tussen de standaard uitvoer verschijnt, wordt de uitvoer in het eerste geval een deel van de opdrachtregel.

Om een variabele af te scheiden van de rest van de tekst kan deze worden omgeven door dubbele aanhalingtekens of door accolades, zodat de volgende uitdrukkingen alle de waarde van de variabele L evalueren en er de letters 'OVE' achter plakken.

```
echo "$L"OVE
echo ${L}OVE
echo "${L}OVE"
```

De beperkingen van de shell als programmeertaal worden duidelijk als je bij de variabele i 1 wilt optellen. Rekenen is mogelijk met behulp van het hulpprogramma expr, bijv. i=$(expr $i + 1) Let erop dat de shell spaties nodig heeft om de woorden te onderscheiden.

2.3. parameters

We hebben inmiddels enkele manieren gezien waarop de shell ingevoerde tekens bewerkt, zoals de substitutie van variabelen en verwijderen van aanhalingstekens. Vervolgens wordt een commando in woorden opgeknipt, waarbij witruimte de scheiding tussen de woorden aangeeft. De variabele IFS bevat de tekens die hierbij als scheidingsteken worden gebruikt, standaard zijn dit <SP>, <TAB> en <LF>. Let erop dat een uitdrukking tussen enkele of dubbele aanhalingstekens altijd als een woord wordt gezien, ook al bevat ze spaties. Twee aanhalingstekens direkt achter elkaar tellen als een woord met lengte nul (0).

In een shell script zijn de parameters (c.q. argumenten en vlaggen) beschikbaar via de speciale variabelen $0 t/m $9 (en zonodig ook ${10} en hoger). $0 is de naam van het programma zelf. Daarmee is het mogelijk om bijvoorbeeld gunzip een link naar gzip te laten zijn en het programma verschillend te laten werken afhankelijk van de naam waarmee het is aangeroepen.

De shell kent nog een aantal speciale parameters; we zijn $? al tegengekomen, dat telkens de exit status van het laatste uitgevoerde commando krijgt. $$ bevat het Unix proces nummer van de shell die uw script draait. Dit kan ook handig zijn om unieke filenamen te maken voor tijdelijke bestanden.

De parameters $* en $@ leveren de complete parameterlijst op, met uitzondering van $0. Tussen dubbele aanhalings tekens gezet zal "$*" een woord opleveren dat overeenkomt met `"$1 $2 $3..."`, terwijl "$@" een reeks woorden `"$1" "$2" "$3"...` geeft. De variabele $# geeft het aantal argumenten.

Hieronder volgt een scriptje om met het gebruik van parameters door de shell te experimenteren. De shift opdracht gooit de eerste parameter weg en schuift de rest een positie op, met uitzondering van $0. De regel met while wordt nader toegelicht in Paragraaf 2.6.

```
#!/bin/sh

echo "Script $0 aangeroepen met $# parameters:"

while [ $# -gt 0 ]
do
  echo -n "\"$1\" "
  shift
done
echo
exit
```

Het verwerken van de parameterlijsten kan nog wat ingewikkelder worden. Zo kan met de notatie ${parameter:-woord} een standaardwaarde (default) worden aangegeven voor een weggelaten parameter of met {$parameter:?woord} een foutmelding worden gegeven als de parameter verplicht is, bijvoorbeeld

```
cp ${1:?"De eerste parameter is verplicht"} \
   ${2:-"."}
```

Zoals gezegd wordt een opdracht beëindigd door een puntkomma of regeleinde. Hier is een backslash gebruikt om aan te geven dat de opdracht nog niet afgelopen is.

2.4. als-dan

De meest elementaire programma-constructie laat één of meer opdrachten al of niet uitvoeren, afhankelijk van de uitkomst van een test, bijvoorbeeld:

```
if /usr/bin/test "A" = "a"
then
  echo "Gelijk"
else
  echo "Verschillend"
fi
```

Eerst wordt de opdracht achter if uitgevoerd. Er wordt dan gekeken welke exit status die opdracht oplevert. De meeste programma's geven standaard 0 terug, en 1 of een andere foutcode als er problemen zijn, /bin/true of : is altijd 0, en /bin/false altijd 1. echo 1 zal weliswaar 1 als output geven, maar het resultaat is 0.

De opdracht(en) achter then worden enkel uitgevoerd als de if-opdracht 0 oplevert, anders wordt het deel achter else uitgevoerd, maar het opnemen van een else-tak is niet verplicht. De if-constructie wordt pas uitgevoerd als het sleutelwoord fi is gelezen. In een interactieve shell krijg je een prompt string te zien om aan te geven dat de shell op de volgende opdracht wacht. Deze prompt wordt bepaald door de variabele PS1. Als een opdracht niet afgesloten is, krijgt u de prompt string PS2 te zien. De voorwaardelijke opdrachtregels worden hier ingesprongen om de leesbaarheid te verbeteren.

De Bourne shell is een vrij beperkt programma. Voor eenvoudige berekeningen kan het hulpprogramma expr worden gebruikt en test voor testen en vergelijken. Moderne shells als ksh, bash of tcsh hebben deze faciliteiten vaak ingebouwd. In plaats van de notatie test `expressie` gebruiken ze [`expressie`]. De shell wil spaties zien tussen de verschillende onderdelen van een expressie, terwijl in de opdracht variabele=waarde juist geen spaties voor of na het =-teken mogen komen. Voor uitgebreidere informatie zie Paragraaf 3.1 en Paragraaf 3.2.

Bovenstaand voorbeeld zal aangeven dat de strings `"A"` en `"a"` verschillend zijn. Hetzelfde geldt voor test "1" = "01" terwijl test "1" -eq "01" moet opleveren dat beide uitdrukkingen hetzelfde getal voorstellen. De opdracht test -e "$file" komt op hetzelfde neer als ls "$file" >/dev/null 2>&1. Zet variabelen tussen dubbele aanhalingstekens, anders zal het fout gaan als de variabele niet bestaat of leeg is.

Er bestaan nog twee beknopte voorwaardelijke opdrachten: `commando1` && `commando2` en `commando1` || `commando2` . In beide gevallen wordt eerst `commando1` uitgevoerd. In het eerste geval wordt `commando2` alleen uitgevoerd als het eerste exit status `0` (nul) oplevert, in het tweede geval als het resultaat van `commando1` ongelijk is aan nul. Het resultaat is dus `true` als `commando1` en c.q. of `commando2` `true` opleveren.

2.5. meerkeuze tests

Om de waarde van een uitdrukking in de standaard uitvoer te krijgen wordt de echo opdracht gebruikt. Om een waarde van de standaard invoer te lezen en toe te kennen aan een of meer variabelen wordt read `naam1 naam2`... gebruikt. De variabelen krijgen dan elk een woord van de invoer als waarde. Als er geen variabelen worden opgegeven, wordt er een regel gelezen en in z'n geheel toegekend aan REPLY.

De if constructie kan met meerdere tests worden uitgebreid:

```
if test1
then
  commando1
else
  if test2
  then
     commando2
  else
  ...
  fi
fi
```

Dit kan ook korter worden genoteerd.

```
if test1
then
  commando1
elif test2
then
  commando2
else
  ...
fi
```

Om de inhoud van een variabele met meerdere waarden te vergelijken gebruiken we de case constructie, zoals in het volgende voorbeeld.

```
echo -n $vraag  # variabele moet gedefinieerd zijn
read antwoord
```

```
antwoord=$(expr substr "$antwoord" 1 1 | tr \
  "[a-z]" "[A-Z]")
case $antwoord in
[YJDOS] )
  exit 0 ;;
N )
  exit 1 ;;
* )
  echo "Ongeldig antwoord!" 1>&2
  exit 2 ;;
esac
```

Hier wordt een vraag gesteld en een antwoord van de gebruiker ingelezen, waarvan de eerste letter wordt genomen en kleine letters in hoofdletters vertaald. De case constructie vergelijkt antwoord met een aantal patronen en voert de reeks bijbehorende opdrachten uit, die wordt afgesloten met een dubbele puntkomma.

`Y` (yes), `J` (ja), `D` (da), `O` (oui) of `S` (si) retourneren exit status `0` (true), `N` (nein of njet) retourneren `1` (false) en alle andere invoer komt overeen met de asterisk en resulteert in 2 met bijbehorende foutmelding op het standard error kanaal.

Een variant hierop is de select constructie, die niet in de originele Bourne shell voorkomt, waarmee eenvoudige menu's kunnen worden gemaakt.

```
echo -n $vraag
select woord in ja nee
do
  if /usr/bin/test "$woord" != ""
  then
    break
  fi
done
```

Hier wordt de keuze van de gebruiker beperkt tot `1` (ja) of `2` (nee). Let op dat "$woord" hier tussen aanhalingstekens moet staan omdat de vergelijking anders niet goed gaat als de waarde van de variabele een lege string is (gebruiker heeft ongeldige invoer gepleegd).

2.6. herhalingen

In het voorbeeld hierboven wordt de invoer zo lang herhaald, totdat de gebruiker een geldige waarde heeft ingevoerd. Een meer algemene herhaling wordt geschreven als

```
while test
do
  opdracht1
  ...
  opdrachtn
done
```

Hierin wordt telkens de opdracht achter while herhaald en zo lang als het resultaat `0` of true oplevert wordt de reeks opdrachten tussen do en done herhaald, wat dus ook nul keer het geval kan zijn en de test (inderdaad is dat vaak het test commando) wordt altijd een keer meer uitgevoerd dan de opdrachten. Een voorbeeld van het gebruik van while zagen we in Paragraaf 2.3.

De until constructie lijkt hier sterk op. Hierin wordt de lijst van opdrachten herhaald zolang als het resultaat ongelijk is aan `0`. De uitvoering van het script gaat dan verder met de opdracht na 'done'.

```
until test
do
  opdracht1
  ...
  opdrachtn
done
```

De for constructie wordt ook vaak gebruikt.

```
for variabele in lijst
do
  opdracht1
  ...
  opdrachtn
done
```

Achter for staat de naam van een variabele zonder dollarteken. Bij elke doorgang krijgt de variabele de waarde van het volgende woord in de lijst. We geven enkele voorbeelden.

```
for file in *
do
  if [ -f "$file" ]
  then
    wc "$file"
  fi
done
```

Hier wordt voor elke file in de huidige directory het commando wc uitgevoerd, dat het aantal regels, woorden en tekens in het bestand telt. Gebruik van wc * is niet zo netjes omdat dan ook directory's en speciale files worden meegenomen.

```
for i in 1 2 3 4 5 6 7 8 9 10
do
  for j in 1 2 3 4 5 6 7 8 9 10
  do
    echo -n $( expr $i '*' $j) " "
  done
  echo
done
```

Herhalingsopdrachten kunnen prima genest worden zoals in bovenstaand voorbeeld, dat de tafels van vermenigvuldiging afdrukt. Let op de enkele aanhalingstekens rond het sterretje. Er staat `-n` achter echo om de tafels op één regel te houden en er wordt een spatie tussenruimte ingevoegd.

Als het woord in en de volgende lijst afwezig zijn dan krijgt de variabele achtereenvolgens de parameters toegewezen waarmee het script werd aangeroepen, zodat we het programma van Paragraaf 2.3 kunnen herschrijven als

```
echo "Programma $0 is aangeroepen met $# parameters"
for i
do
  echo $i
```

```
done
```

Een while lus kan oneindig doorlopen als voor de test : of /bin/true wordt ingevuld. Een lus kan voortijdig worden verlaten met de break opdracht; met break 2 wordt in geval van twee geneste lussen de buitenste verlaten, enzovoorts. De opdracht continue zorgt ervoor dat de lijst opdrachten niet verder wordt afgewerkt en het script verder gaat met de test.

2.7. samenstellingen

Tot de samengestelde commando's behoren if, case, select, while, for, until en de functies, die verderop aan bod komen. Het simpelste samengestelde commando is

```
{
  opdracht 1
  ...
  opdracht n
}
```

Het groep commando zal de commando's in de groep na elkaar uitvoeren. De exit status van de laatste opdracht is tevens het resultaat van de groep. Zo kan bijvoorbeeld een pijplijn als een opdracht worden behandeld met { `opdracht1` | `opdracht2`; }.

Als een lijst commando's in plaats van accolades tussen ronde haken () wordt gezet, worden die binnen een aparte subshell uitgevoerd. Dit geldt ook voor $(`lijst`). In het laatste geval wordt de uitvoer van de lijst als deel van het script gebruikt.

2.8. functies

De definitie van een functie bestaat uit diens naam met een paar haken erachter, doorgaans gevolgd door een lijst commando's.

```
functie-naam()
{
  opdracht 1
  ...
  opdracht n
}
```

Een functieaanroep bestaat uit de naam van de functie gevolgd door de eventuele argumenten gescheiden door spaties. De betreffende commando's worden dan één voor één in de huidige shell uitgevoerd, waarna het script verder gaat waar het gebleven was. Binnen de functie zijn de argumenten beschikbaar als `$1 ...`

De uitvoering van de functie stopt bij de afsluitende accolade of na een return opdracht, die bij voorkeur de laatste opdracht binnen de functie vormt. Achter return kan eventueel een exit status volgen, die aan het aanroepende script beschikbaar komt in $?. De exit status is altijd een geheel getal. Als een string als resultaat gewenst is, kan dat resultaat bijvoorbeeld met echo worden uitgevoerd en de functie via commando substitutie worden aangeroepen. In plaats van for i in 1 2 3 4 5 6 7 8 9 10 kan for i in ‘range 1 10‘ worden geschreven met de volgende functie:

```
range()
```

```
{ typeset i i

  i=$1
  while [ "$i" -le "$2" ]
  do
    echo $i
    i=`expr "$i" + 1`
  done
  unset i
}
```

Het kan handig zijn om functiedefinities in een apart bestand `functies` te zetten en die met ./functies of source functies in te lezen. Bash zal een naam eerst proberen te koppelen aan een alias, dan een functie en als laatste een bestand met dezelfde naam gaan zoeken in zijn pad.

Binnen de functie zijn dezelfde variabelen als in de rest van het script te gebruiken en gewijzigde waarden blijven ook na afloop bestaan. In de Korn en Bash shell kunnen ook lokale variabelen worden gedeclareerd met de typeset opdracht, die alleen binnen de functiecontekst bestaat. In Bash kan hetzelfde ook met local worden bereikt. Een lokale variabele maskeert een eventuele globale variabele met dezelfde naam.

2.9. enkele voorbeelden

Het volgende voorbeeldprogramma geeft de namen van de ingelogde gebruikers. Na een tweetal functiedefinities volgt het hoofdprogramma, dat bestaat uit een pijp van vijf segmenten.

```
word()
{ if [ "$#" -gt 1 ]
  then
    i=0; n=$1; shift
    for j in $@
    do
      if [ $i = $n ]
      then
        echo "$j"
        break
      else
        i=$(expr $i + 1)
      fi
    done
  fi
}

item()
{ OFS=$IFS; IFS=":"
  NR=$1; shift

  word $NR "$@"
  IFS=$OFS
}

# het hoofdprogramma
who \
| while read LINE1
  do
```

```
    while read LINE2
    do
      if [ "$(word 0 "$LINE1")" =
           "$(item 0 "$LINE2")" ]
      then
        echo -e "$(item 0 "$LINE2")\011$(item 4
          "$LINE2")"
      fi
    done </etc/passwd
  done \
| sort \
| uniq -c
```

De functie `word` retourneert het 1e argument, dat wordt gebruikt om een regel in woorden te splitsen; handiger dan daar awk voor te gebruiken. De functie `item` maakt handig gebruik van de ingebouwde shell variabele IFS om de dubbele punt in plaats van de spatie als scheidingsteken te gebruiken. Het hoofdprogramma doet niets anders dan uit de uitvoer van who de username te knippen en die te vergelijken met het corresponderende veld uit `/etc/passwd`. De uitvoer wordt dan gesorteerd en dubbele regels verwijderd.

Deze functies worden ook weer gebruikt in het volgende script voor systemen die niet over een which commando beschikken (zie Paragraaf 2.1). Dit script houdt echter geen rekening met aliassen en ingebouwde functies van de shell.

```
which_p()
{ f=0
  while :
  do
    f=`expr $f + 1`
    dir=`item "$f" "$PATH"`
    if [ "$dir" = "" ]
    then
      break
    fi
    path=$dir/$1
    if [ -f "$path" ]
    then
      echo "$path"
      break
   fi
done
}

# het hoofdprogramma 'which'
if [ `expr match "$1" '.*/.*'` -gt 0 ]
then
  if [ -f "$1" ]
  then
    echo "$1"
  fi
else
  echo `which_p "$1"`
fi
```

Het volgende programma toont verschillen tussen twee bestanden, maar het is een stuk beperkter dan het diff commando. Hier wordt zoals uitgelegd in Paragraaf 1.5 het standaard invoerkanaal omgeleid naar het bestand dat als eerste argument op de commandoregel is meegegeven en er het tweede bestand wordt gekoppeld aan het derde invoerkanaal.

```
doit()
{ LINE=0
  while read EEN<&0
  do
    LINE=`expr $LINE + 1`
    read TWEE<&3
    if [ "$EEN" != "$TWEE" ]
    then
      echo "$LINE""c""$LINE"
      echo "< $EEN"
      echo "---"
      echo "> $TWEE"
    fi
  done
}

# hoofdprogramma
if [ "$#" -ne 2 ]
  then
  echo "Usage: $0 file1 file2"
  exit 1
fi
if [ -r "$1" -a -r "$2" ]
then
  doit 0<$1 3<$2
  exit 0
else
  echo "Invoerbestand(en) onleesbaar"
  exit 1
fi
```

2.10. een voorbeeld van een recursieve functie

Een functie mag ook zichzelf aanroepen (recursie), waarbij je moet voorkomen dat dat proces oneindig doorgaat. In onderstaand voorbeeld voor bash en ksh vindt de recursie alleen plaats als $\$1 > 0$ en wordt het argument verlaagd om te zorgen dat na een aantal stappen $1 gelijk wordt aan 0.

```
Hanoi()
{ typeset i aantal

  if [ "$1" -gt "0" ]
  then
    aantal=$(expr $1 - 1)
    Hanoi "$aantal" "$2" "$4" "$3"
    echo "Verplaats een schijf van $2 naar $3"
    Hanoi "$aantal" "$4" "$3" "$2"
  fi
}
```

```
# Aanroepen met:
Hanoi 4 "A" "B" "C"
```

Voor wie het spel de Torens van Hanoi nog niet kent: je hebt een toren van 4 (of 64, maar dan duurt het spel erg lang) gouden schijven van verschillende diameter, die je één voor één van stapel A naar B moet verplaatsen, waarbij nooit een grotere schijf bovenop een kleinere mag komen te liggen. De optimale oplossing is om de bovenste drie schijven eerst recursief van A naar C te verplaatsen, waarna de onderste schijf van A naar B kan worden verplaatst en de rest weer recursief van C naar B.

Om te bewijzen dat een recursief algorithme korrekt is, is inductie nodig. Het is triviaal in te zien dat het algorithme een stapel van nul schijven van A naar B verplaatst door niets te doen. Voor n > 0 verplaats je eerst de bovenliggende (n - 1) schijven van A naar C en na het verschuiven van de n^e, weer van C naar B. Als we mogen aannemen dat het verplaatsen van (n - 1) schijven korrekt wordt uitgevoerd, dan gaat werkt het op deze manier ook voor een toren van n schijven.

De inductieregel stelt, dat aangezien het algorithme het gewenste effekt heeft voor n = 0 en dat voor elke n ≥ 0 geldt, dat als het goed gaat voor n, ook een toren van (n + 1) schijven korrekt wordt verwerkt. Op dezelfde manier is te bewijzen dat als elke schijf groter is dan de schijf erboven, het algorithme nooit een grotere schijf bovenop een kleinere zal leggen: de toren van (n - 1) schijven is altijd kleiner dan de onderste en de onderste schijf komt nooit ergens bovenop te liggen.

Uit het algorithme volgt een recurrente betrekking voor het aantal verplaatsingen:

$$V(0) = 0$$
$$V(n) = 2 \times V(n\text{-}1) + 1$$

Door een tabel te maken van de waarden voor verschillende n zie je al snel dat dit overeenkomt met $V(n) = 2^n\text{-}1$, wat vervolgens geverifiëerd kan worden.

$$V(0) = 2_0\text{-}1 = 0$$
$$V(n) = 2 \times (2_{n\text{-}1}\text{-}1) + 1 = 2_n - 1$$

Bewijzen dat er geen snellere weg bestaat, is moeilijker. Het kritieke pad is het verplaatsen van de onderste schijf van A naar B. Voor dat het zover is, moeten de schijven erboven eerst op één of andere manier naar C worden verplaatst en daarna weer naar B. Een van de weinige alternatieven is om eerst van A naar C te verplaatsen, en daarna van C naar B, maar dat maakt het enkel langer. Welke keuze je ook maakt, het heeft geen invloed op het verplaatsen van de resterende schijven.

2.11. signalen

We hebben al gezien hoe we processen in de achtergrond kunnen starten en kennis gemaakt met een manier om de manier om twee processen te coördineren door middel van een pijp: het eerste proces mag zolang in de pijp schrijven totdat de kleine buffer vol is; dan zal Unix het stoppen totdat er weer voldoende ruimte vrij is. Het lezende proces wordt telkens als de pijp leeg is in de wacht gezet.

De meeste vormen van interprocescommunicatie vergen een hogere programmeertaal. Voor de shell programmeur zijn de signalen het belangrijkst. Unix heeft twintig tot dertig voorgedefinieerde signalen plus twee vrij definieerbare. Met de opdracht kill `signaal proces` stuur je een signaal naar het opgegeven proces. De nummers van de processen zijn met ps te achterhalen. Het nummer van het proces zelf is te vinden in de variabele $$ en in bash zit het ouderproces in $PPID. De nummers van de signalen verschillen enigszins per Unix versie; de signalen kunnen ook met symbolische namen worden aangeduid; voor bash beginnen die met de letters `SIG`, maar in de korn shell moet dit voorvoegsel worden weggelaten.

Het effekt van een signaal is in de meeste gevallen dat het opgegeven programma onmiddellijk wordt beeindigd. Voor `SIGKILL` (9) is dit altijd het geval. Het ligt voor de hand dat alleen root het recht heeft om andermans processen te doden. Het besturingssysteem kan zelf besluiten een proces een signaal te geven. Als een programma stopt met `segmentation violation`, `bus error`, `floating point exception` of `illegal instruction` duidt dat meestal op een programmeerfout; shell scripts veroorzaken deze fouten zelden.

De gebruiker heeft een snellere manier om signalen naar het proces op de voorgrond te sturen. Deze kunnen vrij worden veranderd met het commando stty. Met stty -a krijg je een lijst van de ingestelde waarden, bijvoorbeeld Ctrl-D voor `SIGHUP`, Ctrl-C of Del voor `SIGINT`, Ctrl-\ `SIGQUIT`. Met Ctrl-Z wordt een lopend programma tijdelijk onderbroken. Onder bash kan een script zichzelf onderbreken met de suspend opdracht. De ingebouwde opdrachten met fg en bg zetten een onderbroken job voort in de voor- resp. achtergrond.

* (fixme: uitleggen wat een job is)

Met de opdracht sleep `seconden` wordt een programma tijdelijk onderbroken en zal het na het opgegeven aantal seconden met `SIGALRM` worden gewekt. De wait opdracht onderbreekt een proces zolang totdat het van Unix het signaal `SIGCHLD` krijgt dat het opgegeven kind proces of job in de achtergrond beëindigd is. Wait zonder argumenten wacht totdat alle kinderen afgestorven zijn. Als een proces in de voorgrond gestart is, wacht de ouder zonder expliciet wait commando.

Een proces kan ervoor kiezen bepaalde signalen gewoon te negeren, met uitzondering van `SIGKILL` en `SIGSTOP`. Het `SIGHUP` signaal dat aangeeft dat een terminalgebruiker de (telefoon)verbinding heeft opgehangen wordt genegeerd met nohup commando&. De standaard uitvoer en error kanalen worden dan omgeleid naar het bestand `nohup.out`.

Signalen worden asynchroon verwerkt, onafhankelijk van waar een proces mee bezig was en na afloop gaat het programma verder waar het gebleven was, tenzij het beëindigd is.

Met het commando trap `commando signa(a)l(en)` wordt aangegeven dat een bepaald commando moet worden uitgevoerd als het volgende signaal c.q. één van de signalen wordt ontvangen. Dit zal vaak de naam van de functie zijn die het betreffende signaal moet afhandelen. Als een script voortijdig moet worden afgebroken zal zo'n functie bijvoorbeeld tijdelijke bestanden verwijderen.

Trap heeft geen effekt op kind processen. Als het commando - luidt of ontbreekt, dan wordt de oorspronkelijke handelwijze hersteld. Als het commando een lege string is, dan zal de lijst van signalen door het script worden genegeerd.

3. enkele Unix tools

De Unix shell is van oudsher de enige programmeertaal zonder ingebouwde optelling. Optellen en een eindeloze reeks andere functies kunnen echter gerealiseerd worden door externe programma's. De verzameling van tools die standaard met Unix worden geleverd vormen een k rachtig en flexibel geheel. Dit hoofdstuk behandelt er een paar die vooral geschikt zijn voor gebruik in scripts; het pretendeert niet volledig te zijn.

Een modale Unix commandoregel begint met de naam van het programma, gevolgd door de opties, die met een minteken beginnen, met daarna de overige parameters, die vaak bestanden zijn. Veelal mogen er ook meerdere bestanden voorkomen. De opties mogen in willekeurige volgorde worden gebruikt en opties van één letter gecombineerd van prog -a -b `file` tot bijv. prog -ab `file`. Verder kunnen argumenten aan opties worden gekoppeld met iets als prog -a `-variabele=waarde` -b … of prog -a `-variabele waarde` -b …; let op wanneer je wel of geen spaties moet gebruiken.

3.1. expr

Wie een rekenmachine zoekt gebruikt meestal bc of xcalc. Expr is speciaal voor shell scripts. Elk argument is een getal of een operator, en moet door spaties gescheiden worden; speciale tekens moeten door backslashes worden voorafgegaan. Expr kent geen variabelen; daarvoor zijn shell variabelen. Om een getal met drie te vermenigvuldigen gebruiken we iets als i=‘expr "3" * "$i"‘

De rekenkundige operatoren + voor optellen, - voor aftrekken, * voor vermenigvuldigen, / voor delen en % voor rest werken met gehele getallen. De vergelijkingsoperatoren >, <, >=, <=, = (in GNU expr ook ==) en != kunnen ook voor strings worden gebruikt. In tegenstelling tot de shell retourneren ze een `1` als de vergelijking waar is en anders `0`, evenals de operatoren | (logische ‘of’) en & (logische ‘en’). In plaats van een nul mag de lege string worden gebruikt en expressies kunnen tussen haakjes ( ) gezet worden. Let op dat in if $(expr "$i" = "") de aanhalingstekens om de variabelenaam niet gemist kunnen worden.

De Korn shell en bash hebben ook een ingebouwde rekenfunctie: de te berekenen uitdrukking wordt omgeven door $((en)). Hier worden geen aanhalingstekens gebruikt en de spaties zijn niet vereist. Er zijn extra operatoren: = staat voor toewijzing, dus voor vergelijking wordt == gebruikt. Behalve in de decimale notatie kunnen getallen ook octaal worden genoteerd door ze met `0` te beginnen of hexadecimaal door er `0x` voor te zetten. Het gebruik van de ingebouwde functie bespaart Unix de tijd voor het maken van het ‘expr’ proces.

Expr kent ook een patroonherkenning operator : die een string vergelijkt met een reguliere expressie (zie Paragraaf 3.3). Deze operator levert het aantal tekens dat overeenkomt op.

Modernere versies van expr bezitten een aantal string functies.

match `string regexp`

: een alternatieve notatie voor patroonherkenning

substr `string begin lengte`

: retourneert een deel van de string van lengte tekens vanaf positie *`begin`*; posities worden vanaf `1` geteld

index `string tekens`

: retourneert de eerste positie in de *`string`* waar één van de reeks tekens voorkomt

length `string`

: retourneert het aantal tekens in de string

3.2. test

Het test commando, dat wordt gebruikt in voorwaardelijke opdrachten, zijn we al tegengekomen. Let op dat test een ingebouwde functie van moderne shells als bash of ksh is, die erg lijkt op `/usr/bin/test`. Die ingebouwde test functie kan ook worden genoteerd als [`expressie`]. De eerste toepassing is om te testen of een bestand bestaat.

-d `file`

: waar als de file een directory is

-f `file`

: waar als de file een normaal bestand is

-r `file`

: waar als je het recht hebt om de file te lezen

-w `file`

waar als je het recht hebt om naar de file te schrijven

-s `file`

waar als het bestand niet leeg is

-t `kanaal`

waar als het kanaal (file descriptor) aan een terminal gekoppeld is. Unix koppelt elk bestand dat geopend wordt aan een nummer. Om te lezen of te schrijven verwijs je naar dat nummer. een bestand kan evt. dubbel geopend zijn. De shell zal bij het inloggen kanaal 0 openen voor invoer van de terminal en 1 en 2 openen voor uitvoer naar de terminal; voor commando `<file` zal de shell het bestand openen met nummer 3 of hoger, het commando uitvoeren en de file weer sluiten.

Hedendaagse Unices hebben wat meer opties. De tweede groep tests vergelijkt strings.

`string1` = `string2`

waar als beide strings gelijk zijn

`string1` != `string2`

waar als beide strings verschillen

`string1`

waar als de string niet de lege string "" is

-n `string1`

waar als de lengte van de string gen nul bedraagt

Om in plaats van strings gehele getallen te vergelijken worden de operaties `-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le` gebruikt. Getallen worden als strings weergegeven. Het verschil is dat test "02" -eq "2" `0` (true) zal opleveren en /usr/bin/test "02" = "2" `1` (false). Dit resultaat krijgt u te zien met echo $?. /usr/bin/expr "02" = "2" zal op de standaard uitvoer `1` (true) schrijven en de exit status `0` (true) retourneren.

De vierde groep zijn de verbindingen. ! `expressie` staat voor logische negatie, `expressie1` -a `expressie2` vormt een logische en en `expressie1` -o `expressie2` een logische of. Verder kunnen subexpressies tussen ( ) haakjes worden gezet.

3.3. grep en reguliere expressies

Grep is een tool om in tekstbestanden te zoeken; met binaire bestanden kunnen veel van de hier besproken tools problemen geven. U kunt het strings commando gebruiken om leesbare tekstfragmenten te vissen uit alle soorten files.

Sinds het ontstaan van Unix zijn er veel versies met verschillende kenmerken in omloop en bovendien zijn er egrep, fgrep en agrep bij gekomen. Ten behoeve van de portabiliteit zijn egrep of fgrep aan te bevelen.

Hieronder volgt de syntaxis van de fgrep opdracht. De rechte haken geven aan dat een onderdeel optioneel is en worden in een echte opdrachtregel niet opgeschreven, evenmin als cursieve termen letterlijk worden genomen.

fgrep [*`optie`* ...] *`string`* ... [*`file`* ...]

Fgrep is het beperktste en tevens snelste lid van de familie. Als er een enkele string als argument wordt meegegeven functioneert het als een filter dat van de regels in de invoer degene doorlaat die de opgegeven string bevatten.

Er kan op meerdere strings worden gezocht, die dan worden gescheiden door een regeleinde. De strings moeten dan tussen aanhalingstekens worden gezet om een argument te vormen en alle regels die één van de strings bevatten worden afgedrukt. Fgrep kent de volgende opties.

`-v`
: alle regels afdrukken, behalve degene die de gezochte strings bevatten

`-c`
: alleen het aantal regels afdrukken

`-l`
: de namen van de files afdrukken, die de gezochte strings bevatten

`-i`
: geen onderscheid maken tussen hoofd- en kleine letters

`-w`
: regels waarin de zoekstring een deel van een woord vormt tellen niet mee

`-x`
: alleen regels die helemaal gelijk zijn aan de zoekstring tellen mee

Er bestaan meer opties, maar die verschillen per systeem. Als je wilt testen of een bepaalde string in een bepaalde file voorkomt kun je de `-s` (silent) optie gebruiken (BSD), of `-q` (quiet, GNU) om de standaard uitvoer te onderdrukken.

Het volgende voorbeeld geeft een lijst van al je processen. (de juiste opties van ps hangen af of je OS tot de System V of BSD familie behoort)

```
ps -al | fgrep `fgrep $USER /etc/passwd | cut -f 3 -d ':'`
```

De binnenste fgrep selekteert de regel van de ingelogde gebruiker uit het wachtwoordbestand. Cut knipt daaruit het derde veld, waarbij de dubbele punt als scheidingsteken wordt gebruikt. De buitenste fgrep selecteert de regels met dit User ID uit de uitvoer van ps. Vergelijk met het voorbeeld uit Paragraaf 2.9.

De functionaliteit van grep zelf verschilt nogal per systeem, en daarom wordt hier het uitgebreidere egrep besproken. De opties zijn hetzelfde, maar het zoekt naar reguliere expressies in plaats van alleen letterlijke strings. Reguliere expressies lijken op de wildcards (jokertekens, zie Paragraaf 1.6) * en ? die de shell gebruikt om naar bestandsnamen te zoeken, maar de notatie verschilt sterk. Denk eraan de zoekstring tussen enkele aanhalingstekens te zetten om verhaspelen door de shell te voorkomen. Het kan geen kwaad om zelf eens wat te proberen op een grote woordenlijst.

```
egrep -i 'abc
def' /usr/share/dict/words
```

Hiermee zoeken we naar alle woorden waarin de letters `abc` en/of `def` direkt achter elkaar staan. De woordenlijst zou ook in bijv. `/usr/dict/words` kunnen staan.

Reguliere expressies moeten voldoen aan de volgende regels.

- Een enkel teken dat geen speciale functie heeft, komt overeen met dat teken zelf
- Om te zoeken naar een teken dat een speciale functie heeft, moet je er een \ voorzetten, bijv. egrep '\\'
- Een reeks tekens tussen vierkante haken wordt gebruikt om te zoeken naar een enkel teken uit deze reeks, zo correspondeert `[0123456789]` met een enkel cijfer.
- Tussen vierkante haken kan een reeks tekens worden aangegeven met een koppelteken tussen het eerste en laatste teken, bijv. `[0-9]`.
- Een punt correspondeert met een willekeurig teken, bijv. `.a.b.c..de` met 'barbecuede'.
- De reguliere expressie *exp* 1*exp* 2 correspondeert met *string* 1*string* 2 als *exp* 1 correspondeert met *string* 1 en *exp* 2 correspondeert met *string* 2, waarbij *string* 1 de langst mogelijke string is waarvoor nog een correspondentie met *exp* 2 mogelijk is.
- De reguliere expressie (*exp*) correspondeert met alle strings die corresponderen met *exp* .
- De reguliere expressie *exp* 1|*exp* 2 correspondeert met alle strings die corresponderen met *exp* 1 of *exp* 2. Iets handiger dan een regeleinde.
- De reguliere expressie *exp* 1* correspondeert met alle strings die nul of meer keer corresponderen met *exp* 1. Zo komt `a.*b.*c.*d.*e` overeen met o.a. 'Abcoude' en blijkt bijv. egrep '(bla)*(bla)' dezelfde output op te leveren als fgrep bla.
- De reguliere expressie *exp* 1+ is equivalent met *exp* 1(*exp1*)*.
- De reguliere expressie *exp* 1? is equivalent met *exp* 1|().
- De reguliere expressie *exp* 1\{*aantal* 1\} is equivalent met *exp* 1 *aantal* 1 keer achter elkaar.
- De reguliere expressie *exp* 1\{*aantal* 1,\} is equivalent met *exp* 1 *aantal* 1 of meer keer achter elkaar.
- De reguliere expressie *exp* 1\{*aantal* 1,*aantal* 2\} is equivalent met *exp* 1 minstens *aantal* 1 en hoogstens *aantal* 2 keer achter elkaar.
- De reguliere expressie ^*exp* 1 correspondeert met alle regels waarin *exp* 1 aan het begin van de regel voorkomt.
- De reguliere expressie *exp* 1$ correspondeert met alle regels waarin *exp* 1 aan het eind van de regel voorkomt.

Een uitbreiding op de Reguliere Expressies, die oudere Unices missen, zijn de karakter klassen die een uitbreiding vormen van de lijsten karakters.

[[:alpha:]]
: de letters van het alfabet

[[:upper:]]
: de hoofdletters

[[:lower:]]
: de kleine letters

[[:digit:]]
: de cijfers

[[:xdigit:]]
: de hexadecimale cijfers

[[:punct:]]

de leestekens

[[:graph:]]

de zichtbare tekens

[[:print:]]

de afdrukbare tekens (zichtbare tekens plus de spatie

[[:blank:]]

de spatie en het tabulatie-teken

[[:space:]]

alle witruimte ([[:blank:]] plus vertical tab en form feed

[[:cntrl:]]

de stuurtekens

Welke tekens tot deze klassen behoren is afhankelijk van uw locale. Als Unix uw moedertaal is, dan typt u export LANG=C of export LANG=POSIX. De klasse [:upper:] bevat dan alleen de ASCII tekens A - Z, maar met export LANG=nl_NL worden ook de diakritische tekens als ÄÛÉÒÑÅßÇijŘŌŒİı met ASCII nummers boven 127 toe gerekend. Ook de volgorde van de output van sort -d -f is afhankelijk met de ingestelde locale.

De equivalentie klassen zijn eveneens afhankelijk van de locale en worden door GNU nog niet goed ondersteund. De notatie [[=*letter*=]] duidt de verzameling van tekens aan die in het woordenboek op de zelfde plaats komen, zoals hoofdletters en letters met een accent.

De GNU versie heeft weer een uitgebreidere set van regels, waaronder de mogelijkheid om terug te refereren naar een eerdere RE tussen haakjes met een backslash gevolgd door het nummer van die expressie, bijv. `([a-z]) is een \1.*"` correspondeert met regels als ‘a is een aapje, dat eet uit zijn poot’, ‘b is een bakker, die bakt voor ons brood’.

Nog uitgebreider dan egrep is agrep, (http://www.tgries.de/agrep/) een algorithme van Sun Wu en Udi Manber. Hiermee kun je zoeken op strings die ongeveer gelijk zijn aan het zoekpatroon.

De belangrijkste extra optie is het maximum toegestane aantal fouten (letters die je moet wijzigen, toevoegen, vervangen om ze met het zoekpatroon te laten overeenkomen). Als eerste optie wordt dan `-1, -2,` ... aangegeven.

Een andere extra optie is mogelijkheid te zoeken naar alle strings die een uitbreiding van het zoekpatroon bevatten. In plaats van egrep 'a.*b.*c.*d.*e' schrijf je dan agrep -p 'abcde'.

3.4. find en xargs

Een van de meest voorkomende taken van shell scripts is het zoeken naar bepaalde bestanden en het bewerken ervan. Script programmeurs gebruiken hiervoor liever find dan ls. Find zoekt een directory en diens subdirectory's recursief af en levert als uitvoer een reeks padnamen (namen van bestanden plus directory's) die aan bepaalde criteria voldoen. De uitvoer wordt niet gesorteerd.

Om een commando uit te voeren op elk van de gevonden bestanden zou je iets kunnen doen als

```
for bestand in ‘find ....‘
do
  commando $bestand
done
```

Dit heeft het nadeel dat de output van find potentieel enorm lang kan zijn, terwijl shells maar een beperkte lengte voor een commandoregel kunnen accepteren. Deze maximale lengte kun je vinden met het commando find /usr/include -name limits.h -follow -exec grep ARG_MAX \{\} \; 2>>/dev/null

De oplossing is de xargs opdracht, die het commando naar wens herhaalt, bijvoorbeeld tien keer met telkens honderd argumenten, als er duizend bestanden zijn gevonden. De syntaxis van deze opdrachten luidt:

find [*file* ...] [*expressie*] | \
xargs [*optie* ...] [*commando*] [*argument* ...]

Het volgende voorbeeld zoekt alle core dumps in uw home directory en subdirectory's. find ~ -name "core" -print | xargs file

De lijst van argumenten begint met een reeks directory's die worden doorzocht, gevolgd door een optionele reeks van tests en een of meer akties. De akties worden alleen uitgevoerd als alle opties `true` opleveren; er mag ook `-a` tussen worden gezet. Als meerdere opties worden verbonden door `-o` dan zal de combinatie waar (0) opleveren als één ervan waar is. Het resultaat van een optie kan worden omgekeerd door er een `!` voor te zetten en opties mogen tussen `( )` haakjes worden gegroepeerd. De haakjes moeten door een backslash worden beschermd tegen substitutie door de shell.

Find kent de volgende tests:

`-name` *"string"*

waar als de naam van de file overeenkomt; de string mag wildcards bevatten mits er haakjes omheen of backslashes voor staan

`-perm` *getal*

waar als de permissie bits overeenkomen met het oktale getal. man chmod geeft meer informatie over deze numerieke modes.

`-type` *t*

waar als het bestand van het type *t* is; `f` staat voor een gewone file, `d` voor een directory, `b` voor een block special file en `c` voor character special file. Nieuwere systemen kennen ook `p` voor named pipes, `s` voor sockets en `l` voor symbolic links (snelkoppelingen).

`-links` *n*

waar als het bestand *n* links bezit.

`-user` *naam*

waar als het bestand eigendom is van gebruiker *naam*; dat mag een login naam of nummer zijn.

`-group` *naam*

waar als het bestand eigendom is van groep *naam*; dat mag een groep naam of nummer zijn.

`-size` *n*

waar als de grootte van het bestand *n* disk blocks bedraagt. Een disk block is 512 bytes groot, niet te verwarren met de blokken van 1024 bytes van df.

`-inum` *n*

waar als het bestand inode nummer *n* heeft.

`-atime` *n*

waar als het bestand gedurende de laatste *n* dagen is gebruikt.

`-mtime` *n*

waar als het bestand gedurende de laatste *n* dagen is gewijzigd.

`-newer` *file*

waar als het bestand recenter is gewijzigd dan de opgegeven *file*

In de voorafgaande opties kan gehalve *n* om een aantal aan te duiden ook de notatie +*n* worden gebruikt voor aantallen groter dan *n* of -*n* om aantallen kleiner dan *n* aan te duiden.

Het volgende voorbeeld zoekt naar andermans files in uw home directory. find ~ \! -user $USER -type f -print | xargs ls -l

Find kent de volgende akties:

`-print`

drukt de naam van het huidige bestand af op de uitvoer. In de GNU versie kan deze aktie worden weggelaten.

`-exec` *commando*

waar als de exit status van het *commando* gelijk is aan nul. Het commando moet worden beëindigd met een puntkomma. Als het commando `{}` bevat, dan zal dat worden vervangen door de huidige filenaam.

`-ok` *commando*

waar als de exit status van het *commando* gelijk is aan nul. Find vraagt dan eerst om bevestiging en voert het commando alleen uit als er `y` wordt ingevoerd.

De GNU versie heeft nog veel meer opties, onder andere:

`-follow`

volg symbolische links

`-mount`

sla gemounte filesystemen over

`-maxdepth` *nivo*

daal de directoryboom slechts tot het opgegeven aantal *nivo's* af.

Hieronder een iets andere manier om core files te zoeken. Let op de backslashes. find ~ -name core -exec file \{\}\; -print | grep core Een handige manier om de inhoud van directory *origineel* met alle subdirectory's te kopiëren naar directory *bestemming* gebruikt de pass-through optie van cpio: find `origineel` -print | cpio -pdm `bestemming`. SCO Unix heeft hiervoor een copy commando.

3.5. directory's doorlopen met du

De du utility geeft een overzicht van de hoeveelheid schijfruimte die een bepaalde directory samen met de onderliggende subdirectory's in beslag neemt. Behalve als een uitgebreide versie van df kun je het ook beschouwen als een snelle variant op find, die alleen directory's weergeeft. De syntaxis luidt:

du [-a] [-k] [-s] [-x] [*directory* ...]

Als er geen directory's opgegeven zijn, bekijkt du de huidige directory. De opties zijn als volgt:

`-a` (all)
: geef de grootte van alle files, niet alleen directory's

`-k` (kilobyte)
: rapporteer de schijfruimte in kilobytes; de standaardeenheid kan 512 bytes of 1024 bytes zijn.

`-s` (summary)
: geef slechts één regel per directory

`-x` (exclude)
: sla directory's op gemounte schijven over

3.6. sed, editen vanuit een script

Met behulp van egrep kun je informatie zoeken in een tekstbestand. Om regels te wijzigen is er de stream editor sed, een broertje van de ed editor, die weer een voorloper van vi is, die speciaal gemaakt is voor gebruik in scripts. Sed kopiëert telkens een volgende regel van de invoer naar de patroonbuffer, voert de opdrachten in zijn edit script één voor één uit en kopiëert de buffer naar de uitvoer. De syntaxis luidt:

sed [-n] [-e *script*] [-f *sfile*] [*file* ...]

Als de filenaam of -namen ontbreken dan zal sed de standaard invoer verwerken. Het edit script kan op de commandoregel achter de `-e` vlag tussen aanhalingstekens worden meegegeven of worden gelezen uit de file die met de `-f` vlag wordt aangeduid. Als de `-n` vlag wordt gebruikt, dan wordt er alleen uitvoer gegeneerd door de print opdracht.

Onderstaand scriptje print de regels uit het wachtwoordbestand die de letters `rot` of `root` bevatten, net als egrep. Zonder de `-n` vlag zouden deze dubbel worden geprint. sed -n -e '/ro\+t/p' /etc/passwd

Sed scripts zien er nog cryptischer uit dan shell scripts; ze bestaan in het algemeen uit regels van de vorm

[*adres*1 [, *adres*2]] [!] [argument ...]

Een scriptregel zonder adres wordt toegepast op elke regel in de invoer, met één adres wordt de functie toegepast op elke regel die met *adres*1 overeenkomt, en met twee adressen op de regels vanaf de eerste die overeenkomt met *adres*1 tot de volgende die overeenkomt met *adres*2. Als dat niet geworden wordt, tot aan het eind van de file. Als achter het adres een uitroepteken staat wordt de functie toegepast op alle regels die niet geselecteerd zijn.

Een adres kan een decimaal regelnummer zijn, cat vanaf 1 geteld wordt, `$` voor de laatste regel of een reguliere expressie tussen schuine strepen. De GNU versie staat toe de reguliere expressie te begrenzen met een ander teken dan de slash. Het volgende voorbeeld retourneert de regels tussen 'root' en 'bin' uit hter wachtwoordbestand. sed -e '\#root#,\#bin#!d' /etc/passwd

Sed kent onder andere de volgende functies:

`p` (print)

: kopiëer de inhoud van de patroonbuffer naar de uitvoer; zonder de `-n` optie kan dat resulteren in dubbel afgedrukte regels.

`n` (next)

: drukt de inhoud van de patroonbuffer af op de uitvoer en leest de volgende regel van de invoer in de patroonbuffer.

`d` (delete)

: maak de patroonbuffer leeg en ga verder met de volgende regel.

a \
`tekst`

: plaats een regel tekst in de uitvoer voordat de volgende regel wordt gelezen.

c \
`tekst`

: Maak de patroonbuffer leeg, schrijf de tekst naar de uitvoer en lees de volgende regel in.

i \
`tekst`

: plaats een regel tekst in de uitvoer. Append en insert hebben hooguit één adres.

`q` (quit)

: beëindig het programma zonder de rest van de invoer te lezen. Quit gebruikt hooguit één adres.

`=` (regelnummer)

: schrijf het regelnummer naar de uitvoer op een aparte regel.

`w` *`file`*

: voeg de inhoud van de patroonbuffer toe aan het opgegeven bestand.

`r` *`file`*

: lees de inhoud van het opgegeven bestand en schrijf die naar de uitvoer alvorens de volgende invoerregel te lezen.

`h` (hold)

vervang de inhoud van de houdbuffer door die van de patroonbuffer.

`g` (get)

vervang de patroonbuffer door de inhoud van de houdbuffer.

`x` (exchange)

verwissel de inhoud van de patroonbuffer en de houdbuffer.

`y/`*string*`1/`*string*`2/` (yield)

vervang tekens uit *string*1 door overeenkomstige tekens van *string*2; beide strings moeten even lang zijn.

`N` (next)

voeg de volgende regel toe aan de patroonbuffer met een regeleinde (`\n`) ertussen. Normaliter bevat de buffer geen `\n`.

`D` (delete)

Verwijder de eerste regel van de patroonbuffer en ga verder met de volgende invoerregel.

`G` (get)

plaats een regel text in de uitvoer voordat de volgende regel wordt gelezen.

`P` (print)

kopiëer de eerste regel van de patroonbuffer naar de standaard uitvoer.

`H` (hold)

voeg de inhoug van de patroonbuffer toe aan de houdbuffer.

`#` *commentaar*

Na een hekje wordt de rest van de regel genegeerd, net als een lege regel

{ *functie*1
*functie*2 ...
}

In plaats van een functie wordt de hele groep uitgevoerd. Functies op één regel mogen ook door puntkomma's worden afgesloten.

`s/`*regul. expressie*`/`*vervanging*`/[`*flags*`]` (substitute)

zoek tekst en vervang die. Als voor het adres reeds een zoekexpressie is gebruikt, kan achter de `s` een lege expressie staan. De flags kunnen een `g` omvatten als er meerdere substituties in een regel kunnen voorkomen, een `p` om de patroonbuffer te printen als er een vervanging is gemaakt en `w` *file* om de inhoud van de patroonbuffer naar de file te schrijven als er iets is vervangen.

`:`*naam* (label)

doet niets, maar markeert een positie om naar toe te springen.

`b [naam]` (branch)

ga verder met de volgende functie achter het label *naam*. Als de naam is weggelaten, spring naar het eind van het script en ga verder met de volgende invoerregel.

`t [naam]` (test)

ga verder met de volgende functie achter het label *naam* als er sinds het inlezen van de invoerregel of sinds de vorige test opdracht een succesvolle substitutie heeft plaatsgevonden. Als de naam ontbreekt wordt er naar het eind van het script gesprongen.

Het volgende voorbeeld zet regelnummers voor de invoer; BSD en gnu systemen bereiken hetzelfde met cat -n. Omdat de regelnummers op aparte regels komen, wordt een tweede sed commando gebruikt om ze samen te voegen en de regeleinden te vervangen door spaties.

```
sed -n -e '=; p' | sed -n -e 'N; s/\n/ /p'
```

De volgende variant illustreert hoe met tweetal adressen meerdere regelbereiken worden gevonden.

```
ls /bin | sed -n -e '/name/,/t/{=;p;}' | \
  sed -n -e 'N; s/\n/ /p'
```

Het volgende voorbeeld zet een spatie achter ieder teken en een lege regel achter iedere regel en vervangt alle kleine letters door hoofdletters.

```
sed -e 's/\(.\)/\1 /g
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
G'
```

Voor meer informatie en voorbeelden, zie the seders' grab-bag. (http://seders.icheme.org/) Hierna volgt een wat uitgebreider shell script dat gebruik maakt van sed om commentaar in C^{++} stijl te veranderen in standaard C code. Het maakt gebruik van ledige substituties om te testen of een bepaald patroon in de invoerregel voorkomt. De functie extension behoeft wellicht verbetering voor namen als `app-1.0.0.tar.gz`.

```
subst()
{ sed -e '
         //\*/s/$//
         t loop
         b cont

:loop
         /\*//s/$//
         t cont
         n
         b loop
:cont
         /\/\//s/$//
         t found
         b
:found
```

```
         /\".*\/\/.*\"/s/$//
         t
         /\*\/\/\*/s/$//
         t
         /\/\*.*\/\/.*\*\//s/$//
         t
         /\/\//s/$/ \*\//
         /\/\/.*\/\*.*\*\//s/\*\///
         /\/\//s//\/\*/
         t
         ' $1
}

stringlen()
{ if [ "$1" = "" ]
  then
    echo "0"
  else
    expr length "$1"
  fi
}

substring()
{ if [ "$1" = "" ]
  then
    echo ""
  else
    expr substr "$1" "$2" "$3"
  fi
}

lastindex()
{ name="$1"
  DotPos1=0
  DotPos2=`expr index "$name" "$2"`
  while [ "$DotPos2" -gt "0" ]
  do
    DotPos1=`expr $DotPos1 + $DotPos2`
    DotPos2=`expr $DotPos2 + 1`
    name=`substring "$name" $DotPos2 999`
    DotPos2=`expr index "$name" "$2"`
  done
  echo $DotPos1
}

extension()
{ DotPos=`lastindex "$1" "."`
  if [ "$DotPos" -gt "0" ]
  then
   substring "$1" `expr $DotPos + 1` `stringlen "$1"`
  else
    echo ""
  fi
}

# main loop - find files to be modified
find . -print \
```

```
| while read i
  do
    if [ -f "$i" ] && [ -r "$i" ]
    then
      if [ -w "$i" ]
      then
        if [ "‘extension $i‘" = "c" ] || \
           [ "‘extension $i‘" = "h" ]
        then
          if grep ’//’ $i >>/dev/null
          then
            echo "File $i is modified."
            mv "$i" "$i"~
            subst "$i"~ >$i
          fi
        fi
      else
        echo "File $i is not writable!"
      fi
    fi
  done
```

3.7. tabellen verwerken met awk

De scripttaal awk (klinkt als `‘look awk‘`) is genoemd naar haar makers, Aho, Weinberger en Kernighan van A, T & T. Awk is uitgebreider dan sed en vooral geschikt voor het verwerken van tabellen. Veel Unix commando's als ls, ps, en who produceren tabellen, platte tekstfiles bestaaande uit kolommen met een vaste breedte. De syntaxis luidt:

awk [*optie* ...] [-f *sfile*] *'script'* [*file* ...]

De belangrijkste optie is `-F`*t*, waarmee het teken *t* als field separator wordt ingesteld. Het script zelf kan weer als een groot argument op de commandoregel worden meegegeven of in een aparte file met `-f`, en als er geen verdere filenamen zijn meegegeven, dan zal awk de standaard invoer verwerken. Awk scripts hebben over het algemeen de vorm

```
[BEGIN {actie}]
[patroon] [{actie}]
...
[END    {actie}]
```

Net als bij sed zal elke invoerregel één voor één met de patronen worden vergeleken en voor elk patroon dat overeenkomt zal de bijbehorende actie worden uitgevoerd. Als er geen patroon is opgegeven wordt de actie altijd uitgevoerd en als er geen actie staat, dan wordt de invoerregel afgedrukt. De eventuele actie achter begin wordt voor het inlezen van de eerste regel uitgevoerd en de actie achter end na het eind van de laatste regel. Het eerste voorbeeld verwijdert lege regels uit de invoer: awk '$0'

Hierin staat `$0` voor de invoerregel. De invoerregel wordt door awk gesplitst in velden, gescheiden door witruimte of een ander scheidingsteken, genaamd `$1`, `$2`, ... `NF` is het aantal velden en `NR` het aantal records c.q regelnummer. Het toevoegen van regelnummers aan de input gaat met awk eenvoudiger dan met sed: awk '{print NR " " $0}'

De patronen kunnen boolese combinaties van reguliere expressies en / of relationele expressies zijn. De boolese combinatoren zijn ! (niet), && (en), || (of) en de ( ) haakjes; de reguliere expressies zijn eerder besproken; de relationele combinatoren zijn < (kleiner), > (groter), <= (kleiner of gelijk), >= (groter of gelijk), == (gelijk), != (ongelijk), ~ (komt overeen met reguliere expressie) en !~ (komt niet overeen met RE).

Expressies kunnen verder bestaan uit variabelen, numerieke constanten (gehele getallen of drijvende komma) en string constanten (tussen dubbele aanhalingstekens) en de rekenkundige operatoren + (optellen), - (aftrekken), * (vermenigvuldigen), / (delen), % (rest), ^ of ** (machtsverheffen), ++ (1 optellen) en -- (1 aftrekken) als in de programmeertaal C en de string operator ' ' (samenvoeging).

Behalve enkelvoudige patronen kunnen net als bij sed reeksen regels worden opgegeven met een tweetal patronen. De volgende regel selecteert de regels 10 t / m 20 van een bestand. awk 'FNR==10, FNR==20' /etc/hosts

Een awk script kan ook variabelen aanmaken; declareren is niet nodig. Een variabele kan zowel als string als getal worden gebruikt en wordt automatisch geïnitialiseerd met waarden `0` en `""`. De operatoren bepalen het type van het resultaat: zo levert `("3x" + 1)` 4 op maar `("x3" + 1)` 1. Behalve een constante tussen schuine strepen kan ook een variabele als reguliere expressie worden gebruikt. De eenvoudigste acties zijn `variabele = expressie` en `print expressie`. Het is ook mogelijk om de veldvariabelen $1, $2,... een andere waarde toe te kennen.

Moderne awk varianten kennen associatieve arrays. Arrays hoeven niet gedeclareerd te worden: het is voldoende een element van een array een waarde te geven met `naam[ index] = waarde`. Voor de index kunnen gehele getallen of willekeurige strings worden gebruikt, zelfs meerdere getallen zijn toegestaan om pseudo-multidimensionale arrays te maken. De waarden van de index hoeven niet opeenvolgend te zijn.

Awk kent een aantal rekenkundige functies.

- sin(*x*) berekent de sinus van x, met x in radialen
- cos(*x*) berekent de cosinus van x
- exp(*x*) berekent de e-macht van x
- int(*x*) berekent de entier van x
- log(*x*) berekent de natuurlijke logarithme van x
- sqrt(*x*) berekent de wortel van x
- rand berekent een toevalsgetal tussen 0 en 1
- srand(*x*) maakt x tot generator van een reeks toevalsgetallen

Awk kent verschillende string functies.

- gsub(*r,s,t*) vervangt in de string *t* overal de substring *r* door *s* en retourneert het aantal malen dat de substitutie is uitgevoerd
- index(*s,t*) retourneert de positie van substring *t* in *s* of 0 is die niet is gevonden
- length(*s*) retourneert de lengte van de string
- substr(*s,p,n*) retourneert de substring van *s*, vanaf positie *p* van *n* tekens lang
- tolower(*s*) retourneert *s* met alle hoofdletters vervangen door kleine
- toupper(*s*) retourneert *s* met alle kleine letters vervangen door hoofdletters

Awk lijkt een beetje op C. Acties c.q. statements worden gegroepeerd door accolades en afgesloten door een puntkomma of nieuwe regel. Een lang statement kan worden voortgezet door de regel met een backslash te beëindigen. Na een # hekje wordt de rest van de regel als commentaar beschouwd. Het if-statement heeft de volgende syntaxis.

```
if( expressie ) statement1 [else statement2]
```

Als de expressie ongelijk is aan 0 c.q. "", dan wordt het eerste statement uitgevoerd en anders het optionele tweede. Voor herhaling bestaan er twee varianten van while en de for-lus.

```
while( expressie) statement
do statement while (expressie)
for( expressie1; expressie2; expressie3) statement
```

De eerste expressie achter for initialiseert de lus, de tweede test of de lus beëindigd moet worden en de derde wordt na elke doorgang uitgevoerd. Om tot tien te tellen gebruik je awk 'BEGIN {} {} END {for(i = 1;i <= 10;i++) print i}'

De volgende vorm van de for-lus loopt alle elementen van een array af:

```
for (variabele in array) statement
```

Met break kan een lus voortijdig worden verlaten; met continue wordt de rest van de inhoud van de lus overgeslagen en met exit wordt de rest van de invoer overgeslagen en gaat awk verder met de eventuele actie achter end; in de end-sectie kan weer een exit opdracht volgen met een exit status.

In moderne versies van awk kunnen in een script eigen functies gedefinieerd worden met

```
function naam( parameter1, ... ) { statements }
```

Functies moeten worden gedeclareerd voordat ze kunnen worden gebruikt. Met de return opdracht kunnen ze een waarde teruggeven. Als in een functie de waarde van een parameter gewijzigd wordt, dan zal dit geen effekt hebben op de waarde van een variabele die als argument is meegeven, behalve voor arrays.

Het is mogelijk om met print naar een willekeurige file te schrijven met de notatie

```
print waarde >"file"
```

Zonder de aanhalingstekens is file een variabele die de naam van het bestand bevat. Om de uitvoer achter de bestaande inhoud van het bestand te plakken wordt het volgende gebruikt:

```
print waarde >>"file"
```

Het is ook mogelijk om binnen een awk script een extern commando te starten dat de uitvoer van print leest met de notatie

```
print waarde | "commando"
```

Om de output van een extern commando in te lezen kun je iets als system("commando" | getline) gebruiken.

Om nette tabellen te maken wordt het printf statement gebruikt met de volgende syntaxis:

```
printf "formaat", expressie1, ...
```

De formaat string wordt gewoon afgedrukt, afgezien van formaat specificaties, die worden vervangen door één van de volgende expressies. Anders dan voor print moet de formaat string worden afgesloten met een regeleinde (\n).

Een formaat specificatie geeft aan in welk formaat de bijbehorende expressie moet worden afgedrukt en heeft de vorm

%[-][0][*breedte*[.*precisie*]]*type*

De expressie wordt afgedrukt in *breedte* kolommen, mits het daarin past, en eventueel aangevuld met spaties, maar als de breedte met 0 begint, worden er voorloopnullen gebruikt. Een eventueel min-teken geeft aan dat de waarde links moet worden aangelijnd. Voor getallen geeft de *precisie* het aantal cijfers achter de komma c.q. decimale punt weer; voor strings is dat de lengte waarop de string wordt afgekapt, waarna het resultaat wordt aangevuld tot de opgegeven breedte. Het afkappen van relevante informatie kan een bug in een script vormen. De volgende formaattypen zijn mogelijk.

- %% een enkel procentteken
- %c een enkel letterteken
- %c een enkel letterteken
- %s een string
- %d een geheel getal; breuken worden niet afgerond
- %o een positief geheel getal in octale stelsel
- %x een positief geheel getal in hexadecimale stelsel
- %f een decimale breuk
- %e een getal in drijvende-komma notatie
- %g de best leesbare notatie voor het getal

Zo zullen de volgende opdrachten

```
{ printf "%10.3s\n", 3.1415692}
{ printf "%10.3f\n", 3.1415692}
```

als output geven:

```
       3.1
     3.142
```

3.8. expect en andere scripttalen

De behandeling van awk is hier vrij kort gehouden ondanks de vrij complexe materie. De clou is dat een van de belangrijkste redenen om awk binnen een shell script te gebruiken, de printf functie, ook in de korn en bourne again shells is ingebouwd. Ook de andere scripttalen als Perl, TCL, PHP3 en Python, zijn te uitgebreid om hier te behandelen. Perl lijkt enigszins op awk, met nog meer mogelijkheden. Tk maakt in combinatie met TCL of Perl GUI scripts mogelijk.

C programmeurs gebruiken make files, die aangeven welke source files met welke opties moeten worden gecompileerd om binaries en libraries te maken. Make files bevatten weer shell commando's. De m4 macroprocessor is een geval apart, dat te vergelijken is met de C preprocessor, en handig is voor assembly programmeurs. Een m4 file kan willekeurige tekst bevatten plus macrodefinities en macro's. M4 vervangt dan de macro's door hun definitie.

Postscript™ van Adobe wordt een paginabeschrijvingstaal genoemd. Postscript programma's worden geschreven door opmaakprogramma's en uitgevoerd door printers, resulterend in een of meer pagina's gedrukt tekst. Elementen die op elke pagina terugkomen worden eenmalig als een functie gedefinieerd. Verder heeft het wat van Forth weg. Het voordeel is dat de taal onafhankelijk van het type printer is.

Om een script te schrijven dat interactie, zowel in- als uitvoer, pleegt met een ander programma, kun je gebruik maken van Expect, (http://expect.nist.gov) dat weer gebruik maakt van John Ousterhout's Tool Command Language. De syntaxis luidt

expect [-di] [-c *script*] [-f *sfile*] ...

De `-d` optie geeft een hoop diagnostische uitvoer. Met de `-i` optie gedraagt expect zich als een interactieve shell. Achter `-c` kan in een heel script (tussen aanhalingstekens) worden opgegeven en met `-f` kan de naam van de file met het script worden meegegeven. Hieronder volgt een voorbeeld waarin expect wordt gebruikt om een bestand te downloaden via ftp, wat met de shell niet zou lukken.

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

# zoals aangegeven in paragraaf 1.3 wordt hierboven
# expect gestart en de rest zijn expect commando's

spawn ftp localhost
        # Met spawn wordt een extern programma gestart

expect -re "Connected.*\n"
expect -re "220.*\n"
expect -re "Name.*:.?"
        # Het expect commando laat expect wachten totdat het
# het externe programma de opgegeven strings uitprint
# of er een time-out of end-of-file optreedt.
# Na de -re optie volgt een reguliere expressie.
# \n staat voor een nieuwe regel en \r voor return.

send "anonymous\r"
expect "anonymous\r\n"

        # Het send commando stuurt de user naam naar het ftp
# proces. Dat zal de invoer terug echoen.

expect  {
          -re 331.*\nPassword: {
                              expect_user -re.*\n
                            }
         }

        # We wachten op de password-prompt van de ftp server
# en dan vraagt het script de gebruiker om zelf
# het wachtwoord in te vullen.

send "$expect_out(0,string)\r"
expect "\r\n"

        # We sturen de invoer van de gebruiker naar de ftp
# server. Die zal het wachtwoord niet laten zien.

expect  {
```

```
        -re (230.*\n)+.*\n.*\n  {}
        -re (530.*.n)+.*\n      exit
        }
}

        # Nu zijn er twee mogelijk heden: 230. User anonymous
# logged in, of 530. Login incorrect; in het eerste
# geval doen we niets en in het tweede geval stoppen
# we de expect sessie.

expect -re "ftp>.?"
send "cd pub\r"
expect "cd pub\r\n"
expect -re "(250.*\n)*250.*successful.*\n"

        # We wachten op de ftp prompt en sturen "cd pub"

expect -re "ftp>.?"
send "ls\r"
expect "ls\r\n"
expect -re "200.*\n150.*\ntotal(.*\n)+226.*\n"

        # We sturen "ls" en krijgen een aantal regels terug,
# en na de prompt sturen we "get README"

expect -re "ftp>.?"
send "get README\r"
expect "get README\r\n"
expect {
 -re local.*\n200.*\n150.*\n226.*\n.*received.*\n {}
 -re local.*\n200.*\n550.*\n                       exit

}
```

De rest van dit hoofdstuk wordt besteed aan een korte beschrijving van enkele handige Unix commando's.

3.9. at en batch

De at faciliteit zorgt ervoor dat opdrachten op een later tijdstip worden uitgevoerd. De syntaxis luidt:

at [-m] [-l] [-f *file*] [-q *letter*] [-r *job*] *tijdstip*

De `-f` optie wordt gevolgd door de naam van het uit te voeren script; als deze optie ontbreekt, dan worden commando's van de standaard invoer gelezen. De uitvoer van het script wordt de gebruiker toegemaild, dus die hoeft op het moment van uitvoering niet ingelogd te zijn. Op de meeste systemen hebben ook gewone gebruikers toestemming om at te gebruiken.

De `-m` optie zorgt ervoor dat er ook mail wordt verzonden als de opdracht geen uitvoer produceerde. Er wordt voor gezord dat de commando's dezelfde omgevingsvariabelen meekrijgen als op het moment dat at werd aangeroepen.

Het batch commando doet hetzelfde als at, behalve dat commando's pas worden uitgevoerd wanneer de belasting van het systeem voldoende laag is. Die belasting krijg je te zien met uptime. De `-q` vlag geeft aan in welke queue (wachtrij) de job terecht komt. De queue bepaalt de prioriteit waarmee een taak wordt uitgevoerd: `a` is de default at queue, `b` de default batch queue.

De tijd kan worden opgegeven als *uumm* of *uu:mm*, desgewenst met `am` of `pm` erachter. De liefhebber mag ook `now`, `midnight`, `noon` of `teatime` gebruiken. Daarachter komt de eventuele datum, die bijvoorbeeld `today` of `tomorrow` mag luiden. Als het tijdstip al voorbij is, wordt aangenomen dat het morgen moet zijn en als de dag al geweest is, dan zal de gebruiker wel volgend jaar bedoelen. Enkele voorbeelden:

```
at 0815am Jan 24, 2002
at 8:15am Jan 24
at now + 1 day
at 5 pm Friday next week
```

Met de `-l` optie krijg je een lijst van de geplande taken te zien. Met de `-r` optie kun je een taak verwijderen. Alleen `root` mag andermans taken annuleren.

3.10. cron

Cron is een Unix daemon, een systeemprogramma dat op de achtergrond loopt, die op gezette tijden opgegeven programma's start, terwijl at voor eenmalige opdrachten is; op sommige systemen start cron elke minuut een programma dat kijkt of er nog at jobs te doen zijn. Cron wordt meestal gebruikt door de systeembeheerder om bijvoorbeeld 's nachts backups te draaien, maar is vaak ook voor de andere gebruikers beschikbaar.

Net als at zal cron eventuele uitvoer van een opdracht naar de opdrachtgever mailen, tenzij je de standaard uitvoer omleidt; er wordt doorgaans een regel in een log file geschreven voor uitgevoerde jobs. Een nadeel van cron is dat taken die niet op het aangegeven tijdstip uitgevoerd kunnen worden, bijvoorbeeld omdat de computer down is, zullen worden overgeslagen. Een ander verschil met at is dat je de omgevingsvariabelen mist die worden ingesteld als je inlogt.

De gebruiker maakt gebruik van de crontab opdracht om zijn/haar lijst met cron taken te wijzigen. De syntaxis luidt:

cron [-u *user*] [-e | -l | -r]

De verticale balk `|` geeft in onze syntaxis-notatie aan dat of de `-e` vlag, of `-l` of `-u` moeten worden gebruikt: u dient het evenmin als de rechterhaken over te typen. Met de `-u` optie kan root andermans cron tabel wijzigen; deze optie kan ook handig zijn in combinatie met su. Met de `-r` optie wordt een cron tabel verwijderd. Met de `-e` optie wordt de cron tabel gewijzigd; voor dit doel wordt vi opgestart tenzij de inhoud van de omgevingsvariabele $EDITOR iets anders vermeldt.

Een cron tabel bestaat uit een aantal regels, die afgezien van blanco of commentaarregels een zestal velden bezitten.

- minuut (0 - 59)
- uur (0 - 23)
- datum (1 - 31)
- maand (1 - 12)
- dag (maandag = 0, zondag = 0 of 7)
- opdracht (gewone `/bin/sh` commandoregel)

De dag van de week of de dag van de maand moeten overeenkomen; het volgende voorbeeld zal op elke vrijdag de 13^{e} tussen elf uur en middernacht elke minuut de inhoud van de `/tmp` directory wissen en op 13 november ongeacht de dag van de week. In de plaats van een getal mogen er meerdere voorkomen met komma's, maar zonder spaties ertussen.

```
# een voorbeeld cron tabel
* 23 13 11 5 find /tmp -exec rm -f {} 2>/dev/null
```

3.11. basename en dirname

Het basename commando verwijdert het directory pad uit een filenaam en eventueel de extensie. De syntaxis luidt:

basename *padnaam* [*extensie*]

Als de filenaam niet eindigt op de extensie dan wordt de hele naam weergegeven en ook als de filenaam gelijk is aan de extensie. Die extensie kan een willekeurige string zijn. Er wordt niet gecontroleerd of de file bestaat. Het volgende voorbeeld geeft de voornaam van alle html-bestanden.

```
for file in ‘find . print | fgrep .html‘
do
  basename $file .html
done
```

Het dirname commando geeft het directory pad van een filenaam. De syntaxis luidt:

dirname *padnaam*

Als er geen slash (/) in de padnaam voorkomt, wordt de huidige directory (.) geretourneerd. Het volgende voorbeeld geeft alle directory's die html-bestanden bevatten, met dien verstande dat alleen de naam en niet de inhoud van de file wordt getest.

```
find . -print | fgrep .html | xargs -l1 dirname | uniq -c
```

3.12. clear

Het commando clear wist de inhoud van het scherm en zet de cursor in de linker bovenhoek. De omgevingsvariabele TERM bevat het type van uw terminal; als clear niet werkt, kijk of uw terminal wordt ondersteund.

3.13. sort en uniq

Met sort kunnen bestanden worden gesorteerd en / of samengevoegd. De sorteervolgorde wordt bepaald door de ingestelde taal en de standaardinstelling is dat de hele regel als sorteersleutel wordt gebruikt. De syntaxis luidt:

sort [-c] [-m] [-u] [-t*teken*] [+*pos*1 [-*pos*2]]
[-o *file*] [-T *directory*] [file ...]

De volgende opties zijn mogelijk:

`-c` (check)

alleen controleren of de invoer reeds gesorteerd is

`-m` (merge)

samenvoegen van bestanden die reeds gesorteerd zijn

`-u` (unique)

neem identieke regels slechts één keer op in de uitvoer

`-t` (tab)

het volgende teken wordt gebruikt om velden te scheiden; gebruik een teken dat niet in de invoer voorkomt als u puur op positie wildt sorteren. Standaard worden tabs en spaties gebruikt om velden te scheiden en na een spatie worden volgende spaties als deel van een veld beschouwd.

`-o` (output)

hiermee wordt aangegeven naar welk bestand de uitvoer wordt geschreven; de uitvoer mag een invoerbestand overschrijven

`-T` (temporary)

hiermee kan de directory voor tijdelijke bestanden worden opgegeven

Verder kan worden opgegeven dat de velden vanaf *pos*1 tot en zonder *pos*2 de sorteersleutel vormen. Als *pos*2 ontbreekt dan vormt de hele rest van de regel de zoeksleutel. Er wordt geteld vanaf 0. Regels waarvan de eerste *i* velden gelijk zijn worden gesorteerd op basis van het $i+1^e$ veld. De posities kunnen van de vorm *m.n* zijn, waarin *m* het veldnummer is en *n* de positie binnen het veld. De volgende opties zijn globaal geldig als ze voor *pos*1 staan, maar als ze achter een positie staan hebben ze betrekking op dat veld.

`-b` (blanks)

negeer voorloopspaties

`-d` (check)

alleen letters, cijfers en spaties zijn bepalend voor de volgorde

`-i` (ignore)

negeer tekens die niet tot het 7-bits ASCII alfabet behoren

`-f` (fold)

verander hoofdletters in kleine letters; hoofdletters verschijnen wel gewoon in de uitvoer

`-n` (numeric)

de invoer sorteren op getalswaarde i.p.v. alfabetische volgorde; dit maakt de `b` optie automatisch aktief. De invoer moet rechts aangelijnd zijn, wat met printf kan, zie paragraaf 3.7.

`-r` (reverse)

sorteer in omgekeerde volgorde

Het volgende voorbeeld sorteert het wachtwoordbestand op Group ID en binnen een groep worden de User IDs in omgekeerde volgorde gezet en per UID op het GCOS alias comment veld, dat vaak de volledige naam van de gebruiker en evt. kamernummer en telefoontoestel bevat.

```
sort -bdft: +3n +2nr -5 /etc/passwd
```

Het uniq commando kan in veel gevallen worden weggelaten en 'sort -u' worden gebruikt om dubbele regels te verwijderen. De syntaxis luidt:

uniq [-u | -c | -d] [-*kolommen*] [+*tekens*] [*in* [*uit*]]

De `-u` optie zorgt ervoor dat enkel de regels die niet dubbel voorkomen worden afgedrukt, terwijl de `-d` optie elke vaker voorkomende regel een keer in de uitvoer verschijnt en met `-c` worden alle regels voorafgegaan door het aantal keren dat ze voorkomen.

Achter het `-` teken komt een eventueel aantal kolommen dat voor de vergelijking wordt overgeslagen; achter het `+` teken komt een aantal tekenposities dat wordt overgeslagen, nadat de voorafgaande kolommen zijn overgeslagen.

Veel commando's geven eerst een titelregel, die we niet mee willen sorteren. In plaats van head (zie Paragraaf 3.17) en tail te gebruiken kan het ook met het scriptje 'knip'.

```
read LINE
echo "$LINE" 1>&2
while read LINE
do
  echo "$LINE"
done
```

Dit leidt de eerste regel om naar het standaard error kanaal; na het sorteren kunnen we beide kanalen weer samenvoegen. Het w commando geeft een tabel van ingelogde gebruikers met twee titelregels.

```
w | knip | knip | sort -f 2>&1
```

3.14. cmp, comm en diff

Met het cmp commando kunnen twee files worden vergeleken. De syntaxis luidt:

cmp [-l | -s] *file*1 *file*2

Standaard wordt het nummer en de inhoud van het eerste byte getoond dat verschilt als octaal getal. Met de `-s` (silent) optie is er geen uitvoer en kan slechts de exit status worden getest, terwijl `-l` alle bytes die verschillenden weergeeft.

Cmp is geschikt voor binaire files; comm vergelijkt twee bestanden regel voor regel. De bestanden dienen gesorteerd te zijn. De syntaxis luidt:

comm [-1] [-2] [-3] *file*1 *file*2

Zonder vlaggen wordt de uitvoer weergegeven in drie kolommen: in de eerste regels die alleen in file1 voorkomen, in de tweede regels die in file2 voorkomen, en in de derde kolom de gemeenschappelijke regels. De kolommen worden gemarkeerd door de regel met nul, een of twee tab-tekens te beginnen; de uitvoer is niet bijster overzichtelijk. De vlaggen geven de optie een of meerdere kolommen weg te laten, bijv. comm -12 ... om alleen gemeenschappelijke regels te zien.

Het diff commando zoekt naar de verschillen tussen twee bestanden. De syntaxis luidt:

diff [-b] [-r] [-e] *file*1 *file*2

Met de optie `-b` worden spaties aan het eind van een regel genegeerd. Met de optie `-r` worden—als *file*1 en *file*2 directory's zijn—alle files in die directory's vergeleken. Met de optie `-e` wordt de uitvoer in de vorm van commando's voor de ed editor (grootvader van vi) geschreven.

De uitvoer van diff kan worden gebruikt om met het patch tool *file*1 te veranderen in *file*2 of andersom. Als een programmeur aan een groot programma een paar regels heeft veranderd, hoeft hij / zij slechts de patch te verspreiden. De uitvoer van diff heeft de volgende vorm:

`n1a n3,n4` (regels toevoegen)
`n1,n2d n3` (regels verwijderen)
`n1,n2c n3,n4` (regels wijzigen)

Achter een edit commando volgt een lijst van de betroffen regels in *file*1 met telkens een `<` ervoor en een lijst regels uit *file*2 voorafgegaan door een `>`; bijv. als in de ene file op regel 13 een fiets staat waar de andere een rijwiel heeft:

```
13c13
< fiets

---

> rijwiel
```

3.15. column

Het column commando (BSD, Linux) kan tekst in kolommen verdelen om een nette uitvoer te geven. De syntaxis luidt:

column [-t] [-x] [-c *kolommen*] [-s *teken*] [*file*...]

De opties zijn:

`-c` (count)

hiermee geef je de breedte van het scherm op. In de meeste gevallen is dat niet nodig omdat de omgevingsvariabele $COLUMNS de juiste breedte bevat.

`-s` (separator)

geeft het scheidingsteken tussen de kolommen (bijv `:` voor `/etc/passwd`)

`-x` (cross)

vult de uitvoer regel voor regel op in plaats van kolom voor kolom

`-t` (table)

bepaalt het aantal kolommen in de invoertekst in maakt deze netjes op

Op de volgende manier kan de uitvoer van find op een ls-achtige manier in kolommen worden gezet.

```
find . -mindepth 1 -print | xargs -l1 basename | \
  sort -u | column
```

In het volgende voorbeeld wordt de uitvoer van ls van een titelregel voorzien.

```
( printf \
  "PERM LINKS OWNER GROUP SIZE MONTH DAY HH:MM/YEAR NAME\n"
  /bin/ls -l | sed 1d
) | column -t
```

3.16. colrm, cut, paste, join

Het colrm commando van BSD Unix verwijdert kolommen uit elke regel van de standaard invoer. Met kolommen worden hier tekenposities bedoeld. De syntaxis luidt:

colrm [*startpositie* [*eindpositie*]]

De posities worden geteld vanaf 1. Als alleen de startpositie wordt opgegeven wordt de rest van de regel verwijderd, anders van start tot en met eind.

System V Unix kent een cut commando dat ongeveer hetzelfde doet, met de volgende syntaxis:

cut [-b | -c | -f] *lijst* [-d *teken*] [*file* ...]

De opties zijn als volgt:

`-b` (bytes)

de kolommen worden opgegeven als byte-posities

`-c` (character)

de kolommen worden aangegeven als tekenposities; in Oostaziatische en Unicode alfabetten kan een teken meer dan een byte lang zijn

`-f` (field)

de kolommen zijn velden gescheiden door tabulatie-tekens

`-d` (delimiter)

geeft een alternatief scheidingsteken aan; alleen toegestaan in combinatie met de `-f` optie

Achter de `-b`, `-c` of `-f` optie volgt een lijst met veldnummers die in de uitvoer moeten worden meegenomen. Dit kan een reeks nummers zijn, gescheiden door komma's zonder spaties ertussen. Voor een enkel getal kan ook een bereik *start-eind* staan, waarbij de start of het eind ook mogen worden weggelaten om de rest van de regel aan te duiden. Zo doen de volgende twee regels hetzelfde

```
ls -al | colrm 42 54
ls -al | cut -b -41,55-
```

De volgende opdrachten doen niet hetzelfde, omdat de uitvoer van ls niet door tabulatie-tekens gescheiden is, maar door rijen spaties.

```
ls -al | awk '{ print $9}'
ls -al | cut -f 9 -d ' '
```

Om het effekt van de awk regel te bereiken doe je

```
ls -al | sed -e 's/[ ][ ]*/ /g' | cut -f 9 -d ' '
```

Op de volgende manier kun je een directory listing sorteren op extensie, waarbij functies uit een voorbeeld in Paragraaf 3.6 worden gebruikt. Voor bash is de optie `-e` vereist.

```
# optioneel argument: directory naam

  ls $1 | while read
     do
     echo -e "`extension "$REPLY"`""\t""$REPLY"
   done | sort -df | cut -f 2
```

Het omgekeerde van cut is paste. Hiermee worden regels uit meerdere files samengevoegd tot een regel, gescheiden door tabs. Paste is wat beperkter dan het join commando. De syntaxis luidt:

paste [[-s] -d *lijst*] *file* ...

De opties zijn als volgt:

`-d` (delimiter)

gebruik een element uit de lijst als veld-scheidingstekens; er wordt telkens een volgend teken uit de lijst gebruikt, om aan het eind weer het eerste te nemen

`-s` (serial)

geeft een alternatief scheidingsteken aan; alleen toegestaan in combinatie met de optie `-f`

Het volgende voorbeeld geeft een lijst bestanden in drie kolommen.

```
ls | paste - - - | column -t
```

Met join worden twee tekstbestanden samengevoegd, die vooraf gesorteerd moeten zijn. Join voegt regels uit beide bestanden samen met dezelfde inhoud van het sleutelveld. Het sleutelveld is standaard het eerste veld; velden worden gescheiden door spaties of tabulatie-tekens. Join is een standaard operatie voor databases.

De uitvoer bevat alle velden van beide bestanden, en het sleutelveld wordt maar één keer gekopieerd. Als een sleutelveld in de eerste file *n* keer voorkomt en *m* keer in de tweede, dan zal de uitvoer *n* × *m* regels met deze waarde bevatten. De syntaxis luidt:

join [-a *filenr* | -v *filenr*] [-e *string*] [-o *lijst*] [-t *teken*]
[-j [*filenr*] *veldnr* | -1 *veldnr* -2 *veldnr*] *file*1 *file*2

Join bezit een aantal opties:

`-a` (all)

produceer ook een regel in de uitvoer voor iedere regel in *filenr* die geen corresponderende regel in het andere bestand heeft; *filenr* is een `1` of `2`

`-e` (empty)

vul lege velden met de opgegeven string

`-j` (join)

geef het nummer op van het veld, dat als sleutelveld moet worden gebruikt; *filenr* is een `1` of `2`

`-1, -2`

`-1` is een synomiem voor `-j1` en `-2` voor `-j2`

`-o` (output)

geeft een lijst van velden die in de uitvoer moeten komen, wat een cut bespaart; een veld wordt aangeduid met *filenr.veldnr* en velden worden gescheiden door komma's

`-t` (tab)

geeft aan welk teken wordt gebruikt om de velden te scheiden (helaas is Unix niet erg consequent in de naamgeving van de opties)

`-v`

produceer uitsluitend een regel in de uitvoer voor iedere regel in *filenr* die geen corresponderende regel in het andere bestand heeft; *filenr* is een `1` of `2`

Met behulp van join maken we een korte variant op het voorbeeld van Paragraaf 2.9.

```
# sorteer de wachtwoord file op login naam
sort </etc/passwd >een
# knip de titelregels uit lijst van gebruikers,
 # scheid de velden door dubbele punt en sorteer
w | ( read; read; cat) | sed -e 's/[ ][ ]*/:/g' \
| sort >twee
# voeg de bestanden samen en maak er nette tabel van
(echo "USER:FULL NAME:TTY"; join een twee -t ':' \
  -o 1.1,1.5,2.2 ) | column -t -s ':'; rm -f een twee
```

3.17. head en tail

Na een aantal opdrachten die velden uit bestanden manipuleren, zijn er nog een paar die regels uit tekst filteren. Naast grep hebben we head en tail die respectievelijk de eerste en de laatste paar regels van een bestand laten zien; standaard zijn dat tien (10) regels. De syntaxis luidt:

head [-*aantal* | -n *aantal*] [*file* ...]
tail [[+ | -] *aantal* [l | b | c]] [-f] [-r] [*file*]
tail [-c *aantal* | -n *aantal*] [*file*]

De opties zijn als volgt:

`-n` (number)

het aantal regels dat wordt weergegeven; dit is hetzelfde als de optie `-aantal`

`+`*aantal* `[l|b|c]`

alleen voor tail; er wordt nu vanaf het begin van de file geteld in plaats vanaf het eind: dit is hetzelfde als `-n +`*aantal*, terwijl `-n -`*aantal* equivalent is met `-n` *aantal* en head geen teken voor het aantal krijgt

de optionele `l` achter het aantal betekent dat er regels geteld worden; als er `c` staat worden bytes geteld en een `b` staat voor disk blokken van 512 bytes

`-c` (character)

Een alternatieve notatie voor `+`/`-`*aantal*`c`

`-r` (reverse)

de regels worden in omgekeerde volgorde afgedrukt; alleen voor BSD Unix

`-f` (follow)

: laat het programma niet beëindigen als het eind van het bestand is bereikt, maar kijkt om de zoveel seconden of de file is gegroeid en drukt nieuwe regels af; handig voor log files.

De GNU versie van head kent een aantal van de opties van tail. Het eerste voorbeeld geeft alleen de namen van de bestanden; het tweede is equivalent met (read; read; cat).

```
head -0 *
tail -n +31
```

3.18. split en dd

Een heel andere manier om een bestand in regels te splitsen is split, dat meerdere output files aanmaakt van gelijke lengte. De syntaxis luidt:

split [- [l]*aantal* | [-b]*aantal* [k | m]] [*file* [*prefix*]]

De namen van de output files beginnen met de opgegeven prefix. Zonder prefix heten ze `xaa, xab, ...` De opties zijn:

`-l` (lines)

: geeft het aantal regels per uitvoer file; de letter `l` mag worden weggelaten; standaard worden files van 1000 regels gemaakt

`-b` (bytes)

: maakt files met het opgegeven aantal bytes; als er `k` achter staat, worden het kilobytes en met `m` megabytes: handig als de uitvoer een vaste grootte moet hebben voor bijvoorbeeld floppy disks.

Een gesplitste file kan worden samengevoegd met cat x* >`file`.

De functie van dd wordt omschreven als ‘convert and copy’, maar omdat cc al bestond, is voor ‘disk dumper’ gekozen. Het wordt vooral gebruikt voor direkte in- en uitvoer naar een fysiek randapparaat, omdat het niet regels maar records met vaste lengte kopieert. De syntaxis luidt:

dd [*optie*=*waarde*...]

Zonder verdere argumenten lijkt het op cat, maar dd kent een respektabele lijst opties:

`if=`*file*

: lees van de opgegeven file c.q. randapparaat

`of=`*file*

: schrijf naar de opgegeven file c.q. randapparaat `ibs=`*`bytes`*`[w|b|k]`

geeft het aantal bytes dat in één keer gelezen wordt (record grootte) op, standaard 512 bytes; als achtervoegsel kan de eenheid worden opgegeven: een woord is twee bytes, een disk-blok 512 bytes, een kilobyte 1024 bytes

`obs=`*`bytes`*`[w|b|k]`

geeft het aantal bytes dat in één keer weggeschreven wordt (record grootte) op

`bs=`*`bytes`*`[w|b|k]`

geeft het aantal bytes dat in één keer gelezen en geschreven wordt; `ibs` en `obs` worden genegeerd als `bs` is opgegeven

`cbs=`*`bytes`*

geeft de grootte van de conversie-buffer aan voor conversie naar of van IBM of EBCDIC formaat

`skip=`*`records`*

geeft het aantal records dat van de invoer wordt gelezen maar niet naar de uitvoer wordt geschreven

`seek=`*`records`*

geeft het aantal records in de uitvoerfile dat wordt overgeslagen voordat dd begint met kopiëren

`iseek=`*`records`*

net als `skip`, maar dd springt over de records heen, wat tijd bespaart

`oseek=`*`records`*

net als seek

`count=`*`aantal`*

kopieert slechts een *`aantal`* records

`files=`*`aantal`*

voegt een aantal bestanden met EOF ertussen samen; alleen handig voor tape

`conv=`*`code`*`,...`

- ascii — converteer van EBCDIC naar ASCII
- ebcdic — converteer van ASCII naar EBCDIC
- ibm — converteer van IBM EBCDIC naar ASCII
- block — converteer tekstbestand met regels van variabele lengte naar records van `cbs` bytes door regels aan te vullen met spaties
- unblock — haal spaties aan het eind weg en sluit regel af met newline
- lcase — vervang hoofdletters door kleine letters
- ucase — vervang kleine letters door hoofdletters
- swab — verwissel even en oneven bytes
- sync — als het invoer record kleiner is dan `ibs`, dan wordt het aangevuld met NULL bytes of—als de optie `block` gekozen is—bytes
- noerror — ga door als er een fout optreedt
- notrunc — maak het uitvoer bestand niet leeg alvorens ernaar te schrijven

De GNU versie kent de opties `files=`, `iseek=` en `oseek=` niet. Het volgende voorbeeld misbruikt dd voor string manipulatie.

```
echo "Hello, world" | dd bs=1 skip=7 count=6 2>>/dev/null
```

3.19. env

Het commando env is een aanvulling op set, waarmee een omgevingsvariabele kan worden ingesteld voor de duur van één commando. De syntaxis luidt:

env [- | -i] [*naam=waarde*...] [*programma* [*argumenten*]]

Env start het programma, dat de opgegeven namen met de bijbehorende waarden als omgevingsvariabelen meekrijgt. Met de optie `-` alias `-i` worden de bestaande omgevingsvariabelen ontzichtbaar voor het programma.

Env zonder argumenten geeft de waarde van alle omgevingsvariabelen, net als het commando printenv (zie Paragraaf 3.28).

3.20. expand, unexpand en tabs

Expand is een hulpmiddel om tabulatie-tekens in een tekstbestand te vervangen door spaties met behoud van paginaindeling. Programmeurs gebruiken het bijvoorbeeld als een printer tabs niet goed verwerkt. Unexpand vervangt reeksen spaties door tab-tekens, wat ruimte bespaart. De syntaxis luidt

expand [- [t]*tabstops*] [*file* ...]
unexpand [-a | -t *tabstops*] [*file* ...]

Posities worden geteld vanaf 1. Standaard wordt een tabstop om de 8 posities aangenomen. Met de optie `-t` kan een andere waarde worden opgegeven. Het is ook mogelijk om de lijst van tabstops expliciet op te geven, gescheiden door komma's, als ze op ongelijke afstanden voorkomen. Unexpand vervangt alleen de spaties totaan het eerste afdrukbare teken, of alles als de optie `-a` is gespecificeerd.

In vi kun je de tabstops zetten met :set ts=n; gerelateerde instellingen zijn :set ai (automatisch inspringen) en :set sw=`m` (inspringdiepte). Om de tab-instellingen te wijzigen op terminals die dit ondersteunen wordt het commando tabs gebruikt, met de syntaxis:

tabs [-*code* | -*n* | *n*1,...] [-T*term*] [+m*n*]
tabs [-T*term*] [+m [*n*]] *n*1 [,*n*2...]

De opties zijn als volgt:

`-T` (terminal)

geef het terminaltype op, als Unix dat niet in de omgevingsvariabele TERM vindt

`+m` (margin)

geef de linker marge op; in de tweede notatie mag `m` zonder getal gebruikt worden voor de standaardwaarde van 10

`-n`

geef een repeterende tabstop op. In de tweede vorm wordt de `-` weggelaten

`n1,...`

geef de lijst van tabstops expliciet op

`code`

een aantal sets tabstops zijn voorgedefinieerd, bijv. `-a` voor S/370 assembler, `-c` voor Cobol, `-f` voor Fortran, etc.

3.21. file

Het commando file onderzoekt het begin van een file en probeert daaruit af te leiden wat voor type bestand het is. Dit is iets betrouwbaarder dan op de extensie van de filenaam af te gaan, maar het blijft gissen. Als uw systeem een bepaald soort document niet herkent, kunt u zelf patronen toevoegen aan `/etc/magic`. De syntaxis luidt:

file [-c] [-f *ffile*] [-m *mfile*] [*file* ...]

De opties zijn als volgt:

`-c` (check)

controleer het formaat van de magic file

`-f` (files)

onderzoek de files waarvan de namen in *ffile* staan

`-m` (magic)

lees de zoekpatronen uit *mfile* in plaats van de standaard locatie

3.22. fold en fmt

Fold breekt lange tekstregels af, zodat de rest op de volgende regel terechtkomt. De syntaxis luidt:

fold [-bs] [-w *lengte*] [*file* ...]

De volgende opties worden herkend:

`-b` (backspace)

de speciale behandeling van backspace, tab en carriage return wordt opgeheven, zodat ze als gewone tekens met breedte 1 tellen

`-s` (spaces)

breek de regel af na een spatie in plaats van midden in een woord

`-w` (width)

hanteer de opgegeven regellengte in plaats van de standaardwaarde 80

Fmt is een simpel programma uit BSD Unix om een tekstbestand wat netter op te maken met een standaard regellengte van 72 (75 voor GNU). Blanco en ingesprongen regels blijven behouden. Het programma pr is gemaakt om tekst in kolommen te zetten met kop- en voetregels voor het afdrukken. Voor C code is er indent, dat de lay-out van een programma helemaal kan omgooien. De syntaxis luidt:

fmt [-*opties*] [*file* ...]

De oude BSD versie kent geen opties; de GNU versie heeft o.a.:

`-c` (crown-margin)

de inspringing van de eerste twee regels van elke alinea blijft bewaard en de rest springt evenveel in als de tweede regel; bedoeld voor wie alinea's begint met in te springen

`-t` (tagged)

als de optie `-c`, maar de eerste twee regels moeten verschillend inspringen

`-u` (uniform)

vervangt meervoudige spaties door een enkele, behalve aan het eind van een regel, waar een enkele spatie blijft staan, maar meerdere spaties door twee worden vervangen

`-w=`*lengte* (width)

geeft de regellengte op

3.23. install

Het commando install commando wordt het meest gebruikt in make files om bestanden naar hun bestemming te kopiëren. De vergelijking van de man pages laat zien dat de System V en GNU uitvoeringen sterk verschillen. De syntaxis luidt ongeveer:

install [-f] [-m *mode*] [-u *user*] [-g *groep*] *file directory*

Enkele van de opties:

`-f` (force)

overschrijf een eventueel bestaand bestand in de doeldirectory

`-m` (mode)

de toegangsrechten van de kopie worden ingesteld op de opgegeven mode, die zowel als octaal getal als in symbolische vorm kan worden opgegeven (voor meer info: man chmod)

`-u` (user)
`-o` (owner)

stelt de eigenaar van de kopie in: dit mag een naam of nummer uit `/etc/passwd` zijn; install wordt doorgaans door root uitgevoerd

`-g` (groep)

geeft de groep op die de kopie zal bezitten

3.24. look

Look print alle regels in een gesorteerd tekstbestand die met de opgegeven string beginnen. Het maakt gebruik van een binair zoekalgorithme, waardoor het veel sneller is dan fgrep, maar ook beperkter in zijn functionaliteit. De syntaxis luidt:

look [-d] [-f] *string* [*file*]

Tenzij een bestand is opgegeven, kijkt look in het standaardwoordenboek, bijv. `/usr/dict/words` of `/usr/share/dict/words`. Enkele opties zijn:

`-d` (dictionary)

houdt lexigrafische volgorde aan en vergelijkt enkel letters, cijfers en witruimte

`-f` (fold)

maakt geen onderscheid tussen hoofd- en kleine letter

3.25. mail

Mail is het standaard interactieve e-mail programma voor Unix gebruikers. Er zijn inmiddels e-mail clients met heel wat meer features, maar mail kan gebruikt worden om berichten te versturen vanuit een programma. De sterk vereenvoudigde syntaxis luidt:

mail [-s *onderwerp*] [*adres* ...]

Als het adres ontbreekt, dan wordt mail gebruikt om interactief te binnengekomen post te lezen en eventueel te wissen; met een adres zal mail een bericht van de standaard invoer lezen en versturen met de eventuele onderwerp-regel (graag tussen aanhalingstekens als er spaties in voorkomen). Als er meerdere ontvangers zijn, moeten de adressen met komma's worden gescheiden.

3.26. mesg

Mesg wordt meestal in `~/.profile` (zie Paragraaf 1.3) gebruikt om het ontvangen van berichten op de terminal met talk of write toe te staan of te blokkeren. De syntaxis luidt:

mesg [n | y]

Met `y` worden berichten aan u doorgegeven, met `n` niet. Zonder argument geeft mesg de momentane instelling. De volgende listing laat zien dat er van twee ingelogde gebruikers een mesg aan heeft staan en de ander uit.

```
crw-rw-rw- 1 root   root  3, 0 Jun  5 19:29 /dev/ttyp0
crw--w---- 1 daniel tty   3, 1 Jun  5 19:26 /dev/ttyp1
crw------- 1 radical tty 3, 2 Jun  5 19:31 /dev/ttyp2
crw-rw-rw- 1 root    root 3, 3 May 18 00:44 /dev/ttyp3
```

3.27. od en strings

Od wordt gebruikt om de inhoud van binaire files in octaal of andere formaten weer te geven. De vereenvoudigde syntaxis luidt:

od [- [b] [c] [d] [h] [o] [x]] [-A *radix*] [-j *bytes*] [-N *bytes*] \
[-t *formaat*] [*file*]

Er zijn onder andere de volgende opties:

`-b` (bytes)

bytes worden weergegeven als octaal getal

`-c` (character)

bytes worden weergegeven in ASCII, met onzichtbare tekens als escape (zie Paragraaf 2.2)

`-d` (decimal)

woorden van twee bytes worden weergegeven als positief decimaal getal

`-x` (hexadecimal)

woorden worden weergeven in hexadecimale notatie

`-v` (verbose)

als een regel meerdere keren herhaald wordt, dan wordt-ie ook net zo vaak afgedrukt

`-A` (address)

geef het getallenstelsel op dat voor de adressen moet worden gebruikt

`-j` (jump)

geeft aan dat een aantal bytes moet worden overgeslagen alvorens de rest af te drukken; het aantal wordt in het decimale stelsel genoteerd, tenzij het met een `0` (octaal) of `0x` (hexadecimaal) begint;

achter het aantal kan nog de eenheid `b` (blokken van 512 bytes), `k` (1024 bytes) of `m` (1048576 bytes) volgen; oudere versies van od zetten dit aantal achter de filenaam met een `+` ervoor in octale notatie of decimaal met een punt achter het getal

`-N` (number)

geef het aantal bytes aan dat van iedere file moet worden afgedrukt; achter het aantal kan nog de eenheid `b` (blokken van 512 bytes), `k` (1024 bytes) of `m` (1048576 bytes) volgen

`-t` (type)

een alternatieve manier om het formaat aan te geven: het type kan met `a`, `c`, `d`, `f`, `o`, `u` of `x` worden aangeduid, waarin `a` staat voor ASCII, `f` voor floating point, `d` voor decimaal en `u` voor een positief decimaal getal; achter de letter kan nog de lengte worden opgegeven met `1`, `2`, `4` of `8` bytes

Strings is een ruw tooltje om tekstfragmenten uit binaire bestanden, met name programma's, te filteren. De syntaxis luidt:

strings [- [a]] [- [n]*lengte*] [-t d | o | x] [file ...]

De opties luiden:

`-a` (all)

doorzoek hele bestand in plaats van alleen de data sectie van een programma; deze optie kan worden afgekort tot een enkel min-teken

`-n` (number)

geeft aan dat alle strings met minimum *lengte* worden geretourneerd in plaats van de standaard waarde 4; deze optie mag worden afgekort tot *-lengte*

`-t` (type)

zorgt ervoor dat voor elke gevonden string de positie in de file wordt afgedrukt in decimale, octale, of hexadecimale notatie

3.28. printenv

Printenv drukt de waarde van omgevingsvariabelen af. De syntaxis luidt:

printenv [variabele ...]

Zonder argumenten worden alle variabelen afgedrukt, net als met set; als de namen van variabelen worden meegegeven, dan worden die afgedrukt. printenv TERM doet vrijwel hetzelfde als echo $TERM.

De exit status is `0` (true) als alle variabelen bestaan en `1` (false) als er één ongedefinieerd is.

3.29. sleep

Sleep onderbreekt de uitvoering van een programma tijdelijk. De syntaxis luidt:

sleep *seconden*

Sleep wordt vooral gebruikt om in een oneindige lus de processor niet nodeloos te belasten in de trant van

```
while true
do
  if [ er_is_post ]
  then
    lees_post
    break
  sleep 10
done
```

3.30. su

Met het substitute user kan een nieuwe shell worden opgestart onder een andere naam; standaard is dat root, ook wel eens als superuser aangeduid. Na een exit commando komt u weer terug in de oude shell. Su vraagt uiteraard om het wachtwoord, tenzij het door root wordt uitgevoerd. Het wordt ook vaak gebruikt om een programma juist minder privileges te geven voor de veiligheid. De syntaxis luidt:

su [-] [*naam*] [-c *commando*]

Standaard erft de nieuwe shell de oude omgeving (directory en variabelen), maar de `-` optie maakt een login shell en voert `~/.profile` uit e.d. Met de `-c` optie wordt in plaats van een interactieve shell een enkel commando uitgevoerd. Vergelijk eens de volgende twee opdrachten.

```
su -c "whoami "
su -c "who am i"
```

3.31. tee

Tee vormt een t-stuk in een pijpleiding. Het is een eenvoudig filter dat de standaard invoer behalve naar de uitvoer ook naar de opgegeven files kopieert. Zie ook Paragraaf 1.7. De syntaxis luidt:

tee [-a] [-i] [*file* ...]

De opties luiden:

`-a` (append)

plakt de uitvoer achter de file(s) in plaats van ze te overschrijven

`-i` (ignore)

Negeer een eventueel `SIGINT` signaal

3.32. time

Time kan zowel een intern shell commando als extern programma zijn. Het voert een programma uit en geeft weer hoeveel tijd het gekost heeft als verstreken tijd en het aantal seconden dat de computer eraan heeft besteed in de user en system modus. De syntaxis luidt:

time *commando* [*argument* ...]

3.33. tr

Tr is een eenvoudig filter waarmee tekens in een tekstbestand door andere vervangen kunnen worden zonder de kracht of complexiteit van bijvoorbeeld sed; de bewerkte standaard invoer verschijnt op de standaard uitvoer. De Sytem V, BSD en GNU versies lopen nogal uiteen. De syntaxis luidt ongeveer:

tr [-c] [-s] [-d] *string*1 [*string*2]

De opties zijn:

(geen)

Vervang alle tekens in de verzameling *string*1 door het corresponderende teken in *string*2, die even lang moeten zijn, bijvoorbeeld tr '[:upper:]' '[:lower:]' om alle hoofdletters door kleine te vervangen

`-c` (complement)

Vervang alle tekens in *string*1 door tekens die niet in *string*1 voorkomen; het resultaat kan per besturingssysteem verschillen

`-d` (delete)

Verwijder alle tekens die voorkomen in *string*1

`-s` (squeeze)

als een teken uit *string*1 meerdere keren achter elkaar voorkomt, dan blijft er maar één over in de output, bijvoorbeeld tr -s '\000-\040' verwijdert dubbele spaties, lege regels e.d.

De opties kunnen ook gecombineerd worden. In de strings mogen de backslash escapes van Paragraaf 2.2 en de karakter klassen van Paragraaf 3.3 worden gebruikt. Tevens kunnen reeksen tekens worden aangegeven met *begin-eind* en [*teken***aantal*].

3.34. wc

Wc is een eenvoudig tooltje om de lengte van een tekst te tellen. De syntaxis luidt:

wc [-l] [-w] [-c] [*file* ...]

De opties zijn:

`-l` (lines)
: tel het aantal regels

`-w` (words)
: tel het aantal woorden

`-c` (characters)
: telt het aantal bytes; sommige versies hebben een optie `-m` die karakters telt

Standaard worden regels, woorden en bytes of tekens geteld. Als er meerdere files worden opgegeven, dan volgt nog een regel met het totaal over alle files (verwarrend als u een bestand `total` hebt :-)

3.35. yes

Yes is een simpel programmaatje, dat zolang `y` of een andere string produceert totdat het wordt afgebroken. De syntaxis luidt:

yes [*string*]

Het wordt vooral gebruikt in pijpleidingen, zoals yes n | fsck / Fsck zal vragen of u wel zeker weet dat u het root file systeem wilt controleren; omdat dat riskant is op een aktief systeem wordt die vraag automatisch met `n`(ee) beantwoord.