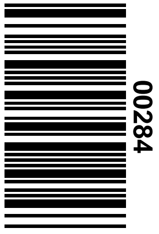


Handleiding

Unix

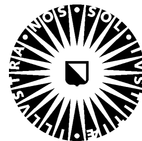
Piet van Oostrum

Handleiding Unix

Piet van Oostrum

4^{e} editie

Juli 1998



Universiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CD Utrecht
Corr. adres: Postbus 80.089
3508 TB Utrecht
Telefoon 030-2531454
Fax 030-2513791

Inhoudsopgave

1 Inleiding **4**

1.1 Wat is Unix? . . . 4

1.2 De shell . . . 6

1.3 De basisbestanddelen van Unix . . . 7

1.3.1 Files . . . 7

1.3.2 Commando's voor files . . . 8

1.3.3 Processen . . . 9

1.4 Unix documentatie . . . 11

2 Processen en de Shell **13**

2.1 Het basismodel . . . 13

2.1.1 Een uitgebreider model . . . 14

2.1.2 De shell als proces . . . 15

2.2 I/O redirection . . . 16

2.2.1 Error uitvoer . . . 17

2.2.2 Pipes . . . 19

2.3 Achtergrondprocessen . . . 20

2.3.1 Het stoppen van processen . . . 21

3 Files **24**

3.1 De inhoud van files . . . 24

3.2 Meer informatie over files . . . 27

3.3 Directories . . . 30

3.4 Devices . . . 30

3.5 Links . . . 31

3.5.1 Harde links . . . 32

3.5.2 Symbolische links . . . 32

3.6 Wildcards 34
3.7 Copiëren, verplaatsen en verwijderen 35
3.8 Overzicht 37

4 Shells **38**
4.1 De Bourne shell 39
4.1.1 Shell uitvoer 40
4.1.2 Argumenten 41
4.1.3 Variabelen 41
4.1.4 Environment variabelen 42
4.1.5 Quoting 44
4.1.6 Exit status 45
4.1.7 Samengestelde commando's 45
4.1.8 Programmeerstructuren 46
4.1.9 Commando substitutie 49
4.1.10 Test en expr 49
4.1.11 I/O redirection 52
4.1.12 Een voorbeeld 54
4.1.13 Shell functies 56
4.1.14 Overzicht 58
4.2 De Korn shell 58
4.3 De C shell 59

5 Filters **60**
5.1 Grep, egrep, fgrep 61
5.1.1 Reguliere expressies 63
5.1.2 Speciale reguliere expressies voor *grep* 63
5.1.3 Speciale reguliere expressies voor *egrep* 64
5.1.4 Fgrep 65
5.2 Tr 65
5.3 Uniq 67
5.4 Sort 67
5.5 Head en tail 71
5.6 Sed 72
5.7 Comm 73

5.8 Diff . . . 74

6 Awk **76**

6.1 Statements . . . 77
6.2 Expressies . . . 78
6.3 Patronen . . . 80
6.4 Voorbeelden . . . 80
6.5 Velden . . . 82
6.6 Printf . . . 83
6.7 Arrays . . . 85
6.7.1 Meerdimensionale arrays . . . 87
6.8 Operaties op teksten . . . 87
6.9 Wiskundige functies . . . 89
6.10 Eigen functies . . . 89
6.11 Output redirection in awk . . . 89
6.12 Argumenten van de *awk* aanroep . . . 90

7 Versiebeheer **92**

7.1 Versies . . . 92
7.2 Revisienummering . . . 94
7.3 Samenwerking . . . 95
7.4 RCS informatie . . . 96
7.5 Vertakkingen . . . 97
7.6 Merge . . . 97

8 Compilatiebeheer **99**

8.1 Macro's . . . 100
8.2 Standaard regels . . . 101
8.3 Make en RCS . . . 102

A Talstelsels **104**

Index **106**

Hoofdstuk 1

Inleiding

Deze handleiding behandelt het gebruik van hulpmiddelen op het Unix systeem die vooral voor het ontwikkelen van software handig zijn. We noemen dit ook wel software tools en je moet ze zien als een soort gereedschapskist voor de informaticus waar allerlei nuttige dingen inzitten waarmee je programma's en documenten kunt maken. De gereedschapskist van Unix heeft als bijzondere eigenschap dat er niet alleen maar kant-en-klare gereedschappen inzitten, vergelijkbaar met hamers en beitels, maar dat je ook allerlei gereedschappen kunt combineren tot nieuwe gereedschappen, zoiets als een boormachine met hulpstukken maar dan nog veel krachtiger, omdat je de hulpstukken ook weer zelf kunt maken . . .

We zullen o.a. behandelen:

- Het gebruik van Unix
- De structuur van het Unix systeem
- Welke tools er zijn
- Hoe tools te combineren tot nieuwe tools
- Tools voor software ontwikkeling

1.1 Wat is Unix?

Unix is een *Operating System* (OS), in het nederlands ook wel *besturingssysteem* of *bedrijfssysteem* genoemd. Een OS is een verzameling programma's die op je computer aanwezig is als je hem opstart, en die allerlei dingen voor je regelen zonder dat je er zelf veel aan hoeft te doen. Voorbeelden van dingen die het OS voor je doet:

- zorgen dat je schijf netjes ingedeeld is zodat er geen informatie verdwijnt
- zorgen dat een opdracht uitgevoerd wordt als je die intypt
- besturen van je beeldscherm zodat je in- en uitvoer daarop verschijnt
- je printer aansturen als je iets wilt afdrukken

Als je geen OS zou hebben zou je eerst allerlei programma's moeten maken om die taken uit te voeren, maar hoe zou je die programma's moeten maken als je geen OS had? Het OS heeft dus als voordeel dat je zelf een heleboel dingen niet hoeft te programmeren. Het heeft echter nog een belangrijk voordeel, nl. dat het *andere programma's onafhankelijk maakt van de soort apparatuur die jij toevallig hebt.* Stel dat het OS zich niet zou bezig houden met de indeling van je schijf. Dan zou elk programma dat gegevens moet opslaan zelf de behandeling van de schijf moeten uitvoeren. Als je dan een keer zou besluiten om een nieuw soort schijf te kopen zou het best kunnen zijn dat je mooie tekstverwerker niet meer zou werken omdat die nog niet voor deze schijf aangepast was. oor nu de behandeling van de schijf in het OS te stoppen zit dit probleem maar op één plaats, en kun je van de leverancier van je OS of je schijf eisen dat die het probleem oplossen. Dit is een voorbeeld van *abstractie* waarbij je een oplossing aanbiedt aan programma's (in dit geval dus voor de besturing van een apparaat), wat onafhankelijk is van de toevallige eigenschappen van het apparaat. Moderne OS's hebben meer abstracties dan oudere, en je ziet dan ook wel bij oudere systemen, dat je *wèl* je teksteverwerker moet aanpassen aan het soort beeldscherm dat je hebt, gewoon omdat het OS geen abstractie voor beeldschermen bevat.

Wat meestal ook tot het OS gerekend wordt maar strikt genomen er los van staat is een grote verzameling handige programma's, bijv. tekst-editors, compilers.

Er zijn vele OS's, sommige zijn al uitgestorven, andere staan al met een been in het graf. Heel bekende zijn MS-DOS en MS-Windows op IBM-compatibele computers, en System 7 (of MacOS) op de Macintosh. Op grotere computers en z.g. werkstations is Unix in allerlei varianten heel populair. Linux is een variant van Unix die gratis te verkrijgen is envooral op PC's gebruikt wordt.

Wat zijn de kenmerken van Unix?

- Het is in gebruik op zeer veel verschillende soorten computers (diverse merken, modellen, CPU's etc.). Vanaf PC's tot supercomputers
- Verschillende personen kunnen tegelijk de compuetr gebruiken. Dit wordt *multi-user* mogelijkheid genoemd
- Een gebruiker kan meer dan één programma tegelijkertijd draaien. Dit wordt *multi-tasking* genoemd
- Bescherming tegen kwaadwillige medegebruikers (je kunt niet de gegevens van iemand anders vernielen en tegen programmafouten (een fout programma heeft meestal geen invloed op andere programma's)
- Naar keuze gebruik alleenstaand of in grote netwerken
- Standaard grafische interface beschikbaar (X Window System)
- Veel tools beschikbaar (vaak gratis)
- Tools kunnen eenvoudig met elkaar gecombineerd worden

- Negatieve punten van Unix zijn o.a. dat er veel verschillende versies bestaan, dat de documentatie niet altijd zo begrijpelijk is, en dat de commando's vaak onbegrijpelijk zijn voor niet-ingewijden

1.2 De shell

Waneer je met een Unix systeem wilt werken moet je eerst inloggen. Je geeft je *username* (gebruikersnaam) en je *password*. Als alles goed is kun je daarna gaan werken. Wat er in feite gebeurt is dat het Unix systeem een programma voor je opgestart heeft dat opdrachten van je kan aannemen en die uitvoeren. Zo'n programma wordt een *shell* genoemd. Er zijn twee soorten shells: de grafische shells waarmee je met icoontjes op je scherm een overzicht van de commando's en gegevens gepresenteerd krijgt en de tekstuele shells waarbij je commando's moet intypen. De grafische shells zijn het gemakkelijkste voor simpele werkzaamheden omdat je met een simpele muisklik een programma kunt starten. Je hoeft ook niet na te denken over hoe de naam van het commando ook al weer was, want je ziet ze voor je neus. Je hoeft niet te weten in welke volgorde je de verschillende gegevens moet invoeren want meestal staat er netjes een formuliertje voor je klaar waar je dat allemaal kunt invullen. Aan de andere kant mis je hierdoor veel flexibiliteit. Daarom is het handig om in ieder geval ook een tekstuele shell te hebben. Een voorbeeld van iets dat gemakkelijker gaat met een tekstuele shell: Als je 100 keer dezelfde opdracht moet uitvoeren met iedere keer iets andere kenmerken dan is dat meestal handiger in een tekstuele shell bijvoorbeeld omdat je zoiets kunt zeggen als: *`repeat 100 opdracht`*, i.p.v. 100 keer op een muis te klikken. We zullen ons in dit college voornamelijk bezig houden met tekstuele shells.

De Unix shell heeft een simpele structuur: de shell nodigt je uit een opdracht in te typen door de *prompt* af te drukken. Dit is een simpel stukje tekst waaraan je kunt herkennen dat het tijd is om een opdracht te geven. Je typt je opdracht in, die misschien is afdrukt, en na afloop wordt de prompt weer getoond. De prompt kan een simpel teken zijn zoals `$`, maar je hebt ook de mogelijkheid om zelf iets ingewikkelders op te geven. Een Unix *sessie* kan er dan als volgt uitzien (zie fig. 1.1.

Figuur 1.1 Een Unix sessie

```
login: piet password: hadjegedacht $ ls
aap noot mies
$ cp aap test1
$ ls
aap noot mies test1
$ rm test1
$ exit
```

In dit voorbeeld gebruiken we verschillende lettertypes voor de `uitvoer` van de computer en voor *`wat wij intypen`*. Speciale toetsen zoals "Control/D" die je krijgt door De "control" toets vast te houden terwijl je de letter "d" intypt geven we aan met [*^D*]. Hierbij maakt het i.h.a. niet uit of je een kleine letter "d" of een hoofdletter "D" intikt.

In te typen regels moet je afsluiten met de "return" toets, maar om het overzichtelijk te

houden laten we die niet zien.

Verder zul je af en toe een constructie als ⟨naam⟩ tegenkomen. Dit betekent dat je hier zelf iets moet invullen, dus bijvoorbeeld ***piet*** i.p.v. ⟨naam⟩.

1.3 De basisbestanddelen van Unix

Het Unix systeem bestaat natuurlijk uit vele onderdelen, maar voor de gebruiker zijn twee aspecten heel belangrijk: de *files* en de *processen*. Files zijn het makkelijkst te begrijpen dus daar beginnen we mee.

1.3.1 Files

Een file is een verzameling gegevens die een naam heeft. In het nederlands zeggen we ook wel eens *bestand*. De informatie in de file kan van alles zijn: tekst (gedicht, brief, scriptie), een programmatekst, een adressenbestand, een gecompileerd programma, plaatjes, muziek, getallen, troep, etc. Afgezien van tekst is aan een file zelf lang niet altijd te zien wat de informatie voorstelt. Het is zelfs mogelijk de informatie zo te versleutelen (met een geheime code) dat het zonder sleutel niet meer te bekijken is.

Elke file heeft een naam; namen mogen in de meeste moderne Unix systemen max. 255 tekens lang zijn (op oudere Unix systemen soms 14), en de naam mag bestaan uit alle letters, cijfers en speciale tekens behalve de schuine streep ("/")[1]. Het is zelfs toegestaan om spaties en andere onzichtbare tekens in een filenaam te gebruiken maar dat wordt afgeraden omdat zulke filenamen problemen kunnen geven. De volgende tekens kun je beter ook niet gebruiken, hoewel ze strikt genomen wel toegestaan zijn:

```
~ ‘ $ ^ & * ( ) [ ] | \ " ’ < > ? ;
```

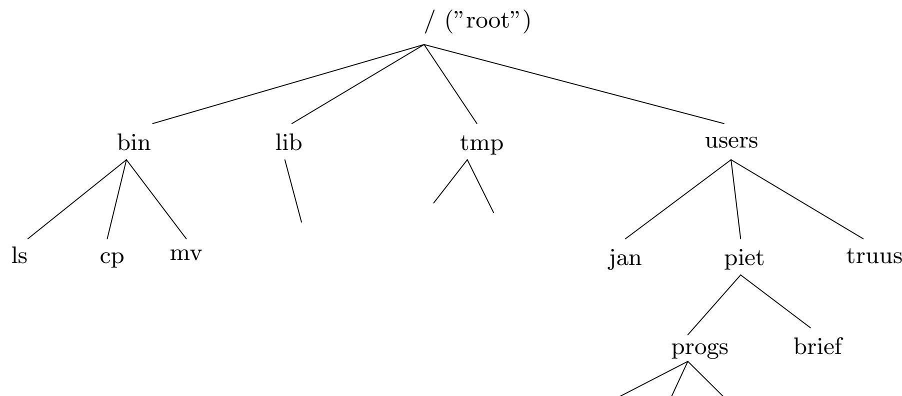
Op Unix is een hoofdletter in een filenaam iets anders dan een kleine letter!

Om te voorkomen dat je alleen maar een grote bak met files hebt waar je verder niet meer je weg in kunt vinden kun je de files organiseren in z.g. *directories*. Een directory is in feite een speciaal soort file (door het systeem als zodanig herkenbaar) waar weer informatie over andere files en evt. weer nieuwe directories staat. Op deze manier wordt een hele hiërarchie van files opgebouwd, zie bijv. fig. 1.2

Het begin van de hiërarchie (ook wel de "root" of "top" genoemd) wordt aangeduid met de naam "/", alle andere files of directories krijgen een naam die bestaat uit de componenten vanaf de root gescheiden door "/", dus bijvoorbeeld "/bin/ls", "/users/piet/brief". Deze samengestelde filenamen worden ook wel *padnamen* genoemd. Omdat je op deze manier nogal lange padnamen kunt krijgen is er een afkortingsmechanisme bedacht. Dit mechanisme werkt doordat het Unix systeem voor je een z.g. *werkdirectory* bijhoudt. Dit is de directory waarin je op dat moment aan het werk bent. Als je inlogt is dat je privé directory, maar je kunt dat tijdens het werken veranderen. Als je nu een filenaam gebruikt die *niet* met "/" begint dan betekent dit dat de file (of directory) gezocht wordt vanaf je werkdirectory. Dus als ik in de werkdirectory "/users/piet" de filenaam "brief" gebruik, betekent dit voluit:

[1]Dus niet zoals op sommige andere systemen, max. 8 letters/cijfers, een punt en dan nog 3 letters/cijfers

Figuur 1.2 Directory hiërarchie



"/users/piet/brief". Padnamen die met "/" beginnen worden *absolute* paden genoemd, de andere *relatieve*.

Elke directory heeft twee bijzondere elementen met namen "." en "..". De eerste (".") is de directory zelf, dus "/users/piet/." is hetzelfde als "/users/piet". Voor absolute padnamen is dat niet zo interessant, maar wel voor relatieve padnamen. Als je bijvoorbeeld een programma hebt wat een directorynaam nodig heeft is dit de gemakkelijkste manier om je werkdirectory op te geven. De tweede ("..") betekent de directory die in de hiërarchie een stapje hoger ligt. Dus "/users/piet/progs/.." is hetzelfde als "/users/piet". Ook hier is het niet zo interessant voor absolute padnamen, maar wel voor relatieve. Stel bijv. dat ik als werkdirectory heb "/users/piet/progs" en ik wil de brief file aanduiden. Dan kan dat met "/users/piet/brief", maar ook met "../brief", en natuurlijk ook met "/users/piet/progs/../brief" of "/users/piet/./brief" of ".././brief" maar dat is meer iets voor masochisten.

1.3.2 Commando's voor files

De eenvoudigste commando's voor het manipuleren van files zijn:

- *ls* voor het geven van een lijst van files in je werkdirectory (of in een andere op te geven directory). Als de werkdirectory "/users/piet" is dan geeft:
  $ *ls*
  de lijst van files en directories in "/users/piet"
  $ *ls progs*
  de lijst van files in "/users/piet/progs"

- *cp* voor het copieren van files:
  $ *cp brief brief2*
  maakt een copie van de file "brief" en noemt de copie "brief2". De beide files zijn hierna onafhankelijk. Je kunt met *cp* geen copie van een directory maken. Je kunt wel een heel rijtje files naar een andere directory copiëren, bijv':

$ *cp brief brief2 progs*
copieert de beide files naar de directory "progs".

- *mv* voor het verplaatsen of hernoemen van files:
  $ *mv brief2 brief3*
  verandert de naam van "brief2" in "brief3".
  $ *mv brief brief3 progs*
  verplaatst de files "brief" en "brief3" naar de directory "progs".
  $ *mv brief progs/brief*
  is een omslachtige manier om één van deze files te verplaatsen.

- *rm* verwijdert één of meer files:
  $ *rm progs/brief3*
  verwijdert een van de files die we hierboven verplaatst hebben.

Uitgebreidere informatie over deze en meer file-opdrachten vinden we in sectie 3.7.

Opgave 1.1 Wat doet het volgende commando?
$ *cp progs/brief .*

Opgave 1.2 Hoe kun je hetzelfde effect bereiken als de opdracht:
$ *mv brief brief2 progs*
zonder het *mv* commando te gebruiken?

1.3.3 Processen

Bijna alle commando's die je geeft hebben tot gevolg dat één of ander programma gestart wordt. Het *ls* commando bijvoorbeeld heeft tot gevolg dat het gelijknamige programma opgestart wordt. Dit programma is te vinden in de file "/bin/ls". (Niet dat er veel te zien is aan die file). Laten we eerst eens kijken naar de verschillende soorten programmafiles die er zijn:

Als je een programma schrijft dan maak je een tekst geschreven is één of andere programmeertaal. De meeste programma's op Unix zijn geschreven in de programmeertaal "C", maar ook andere talen worden gebruikt. Deze programmatekst is op zichzelf niet geschikt om door de computer uitgevoerd ("*geexecuteerd*") te worden. Daarvoor moet hij eerst vertaald of *gecompileerd* worden. Dit gebeurt door een compiler aan te roepen, bijvoorbeeld het commando:
$ *cc -o ls ls.c*
compileert de programmatekst in de file "ls.c" en maakt een nieuwe file "ls" die wèl geschikt is om uitgevoerd te worden. Zo'n file wordt wel een *executable* of *binary* genoemd, vandaar dat veel van dit soort files in de directory "/bin" te vinden zijn. Voor mensen is er niet veel herkenbaars in zo'n file, voor Unix des te meer.

Een executable file die op de schijf staat doet nog niets, pas wanneer deze in het geheugen geladen wordt en de computer gaat de instructies die erin staan uitvoeren, dan gebeurt er wat. We zeggen dan dat er een *proces* opgestart is. De reden dat we spreken over een *proces* en niet over een *programma* is dat hetzelfde programma best meer keren tegelijk aktief kan zijn.

Omdat Unix een multi-user en multi-tasking systeem is kunnen bijv. twee of meer gebruikers dezelfde editr aan het gebruiken zijn.

Zelfs kan één gebruiker hetzelfde programma meer dan een keer tegelijk geaktiveerd hebben bijvoorbeeld in verschillende windows op het beeldscherm. Om nu onderscheid te kunnen maken tussen deze verschillende aktivaties noemen we elke aktivatie een *proces*. Daarbij moet je in het oog houden dat deze processen op zich niets met elkaar te maken hebben behalve dat ze toevallig hetzelfde programma executeren.

Dus een proces:

- is een “draaiend” programma
- heeft een eigen stuk geheugen en wat andere attributen die we in sectie 2 zullen zien
- Als ik (of iemand anders) een programma nog eens opstart krijg ik een nieuw proces

We zullen in volgende hoofdstukken zien op welke manieren je een aantal processen kunt krijgen. Soms is het nuttig om te weten welke processen er aktief zijn. Daarvoor is het commando *ps*. Hieronder zie je een voorbeeld van de uitvoer van het ps commando[2]:

```
$ ps
  PID TTY      TIME COMD
 23347 ttyq0    0:00 ps
 23330 ttyq0    0:00 ksh
```

Voor elk proces krijg je een regel informatie die de volgende onderdelen bevat:

PID	Process Id, een nummer dat het proces onderscheid van alle andere
TTY	De “terminal” waarop het draait
TIME	Hoeveel tijd het proces echt gebruikt heeft (exclusief wachttijd)
COMD	het command waarmee het proces opgestart is

Omdat het uitvoeren van het “ps” commando ook een proces tot gevolg heeft zul je dit in ieder geval altijd aantreffen. In bovenstaand voorbeeld is er nog één ander proces, nl. de shell (het commando is in dit geval *ksh*). In dit oninteressante voorbeeld zijn er geen andere processen door de gebruiker opgestart. Deze vorm van aanroep van het *ps* commando geeft alleen de processen die op dezelfde “terminal” (meestal betekent dit een window op het scherm) draaien. Als je nog andere windows hebt waarin processen draaien dan kun je het commando *ps -u* ⟨gebruiker⟩ geven. Dit geeft dan alle processen van ⟨gebruiker⟩. Om je eigen processen te krijgen moet je dan ⟨gebruiker⟩ vervangen door je eigen loginnaam, bijvoorbeeld *ps -u piet*.

In werkelijkheid zijn er op een Unix systeem nog veel meer processen aktief, die echter voor de gemiddelde gebruiker niet interesant zijn, en daarom niet door het *ps* commando afgedrukt worden. Wil je ze allemaal zien, dan kun je het commando *ps -ef* geven. Behalve dat dit alle processen geeft, geeft het per proces ook nog veel meer informatie.

[2] Er zijn Unix systemen waar het *ps* commando een andere uitvoer geeft, en ook andere parameters accepteert. Raadpleeg zo nodig de Unix documentatie

1.4 Unix documentatie

Alle standaard Unix commando's worden beschreven met z.g. *man pages*. Een *man page* is een (meestal) korte beschrijving van een commando die met behulp van de *man* opdracht (gevolgd door de naam van het te beschrijven commando) opgevraagd kan worden. Zo kan bijv. de documentatie van het *ls* commando opgevraagd worden met:
$ *man ls*
Behalve commando's kan *man* ook andere onderdelen van het Unix systeem beschrijven. Figuur 1.3 geeft een voorbeeld van zo'n "man-page", nl van het *cd* commando waarmee

Figuur 1.3 Man page voor "cd"

```
CD(1)

NAME
     cd - change working directory

SYNOPSIS
     cd [ directory ]

DESCRIPTION
     If directory is not specified, the value of shell parameter $HOME is used
     as the new working directory.  If directory specifies a complete path
     starting with /, ., .., directory becomes the new working directory.  If
     neither case applies, cd tries to find the designated directory relative
     to one of the paths specified by the $CDPATH shell variable.  $CDPATH has
     the same syntax as, and similar semantics to, the $PATH shell variable.
     cd must have execute (search) permission in directory.

     Because a new process is created to execute each command, cd would be
     ineffective if it were written as a normal command; therefore, it is
     recognized and is internal to the shell.

SEE ALSO
     pwd(1), sh(1), chdir(2)
```

je de werkdirectory verandert. In de beschrijving van het commando staan wat dingen die verderop besproken worden, maar het voorbeeld is hier genomen om de structuur van zo'n "man-page" te laten zien.

Zo'n man-page bestaat i.h.a. uit de volgende onderdelen:

kop Als kopregel de naam van het commando met tussen haakjes het sectienummer, dus "CD(1)" in bovenstaand voorbeeld. Sectie 1 is voor commando's, 2 voor aanroepen van het Unix systeem vanuit programma's, 3 voor extra functies die je in een programma kunt gebruiken, etc.

NAME Nog eens de naam, gevolgd door een korte beschrijving. De beschrijving wordt ge-

bruikt om te kunnen zoeken als je de naam niet meer precies weet: de opdracht:
$ *man -k directory*
geeft je alle commando's waar het woord "directory" in deze beschrijving voorkomt.

SYNOPSIS geeft een beknopte beschrijving van de manier waarop het commando aangeroepen kan worden, dus welke extra argumenten het commando kan of moet hebben. Hierbij worden de volgende notaties gebruikt: Tussen vierkante haken "[ ]" dingen die je wel op mag geven maar die niet hoeven, en met "..." wordt aangegeven dat het vorige ding herhaald mag worden. In bovenstaand voorbeeld betekent [directory] dus dat wel of niet een directory opgegeven mag worden. Het "..." geval is hier niet aanwezig maar dat zie je vaak bij commando's waar één of meer filenamen meegegeven moeten worden dan staat er "file ldots". Als de filenamen dan ook weggelaten mogen worden kan er een combinatie gegeven worden, dus bijvoorbeeld "[file ...]".

DESCRIPTION beschrijft wat het commando doet, wat de eventuele extra argumenten betekenen en hoe verschillende varianten van het commando gegeven kunnen worden. Veel Unix commando's hebben z.g. *opties* die het programma beïnvloeden, en die meestal met een – teken beginnen. Bij het *ps* commando hebben we bijvoorbeeld de *-u* optie gezien. De opties worden ook in de DESCRIPTION sectie beschreven (en ze worden natuurlijk in de SYNOPSIS genoemd).

SEE ALSO geeft man-pages aan die gerelateerde opdrachten of functies beschrijven, in dit geval o.a. het *pwd* commando, dat de naam van de werkdirectory afdrukt

Opgave 1.3 Bekijk de man-page van het *pwd* commando, van het *cp* commando en van *man* zelf. Welke opties heeft *man*?

Hoofdstuk 2

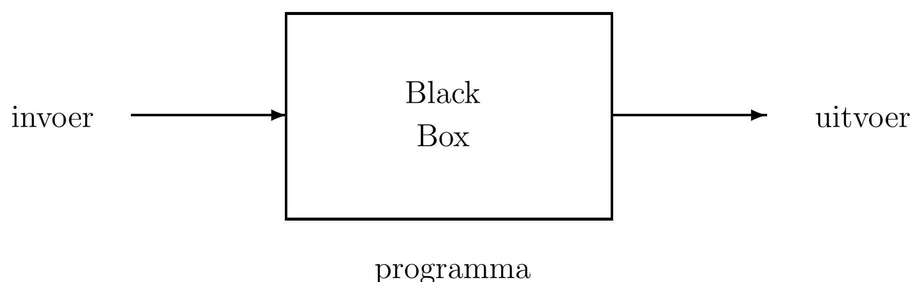
Processen en de Shell

In dit hoofdstuk gaan we dieper in op processen. We zullen leren hoe we processen kunnen opstarten, wat er bij een proces hoort, en hoe processen met elkaar kunnen communiceren. Omdat we deze dingen vaak met behulp van de shell doen, zullen we die ook eens wat beter bekijken. Er is echter nog veel meer over de shell te zeggen en dat doen we in een ander hoofdstuk (4).

2.1 Het basismodel

We beginnen met een simpele voorstelling van zaken: een proces is een "zwarte doos" waar iets in gaat en iets uitkomt (figuur 2.1). Stel bijvoorbeeld dat het programma dat we opge-

Figuur 2.1 Een proces als "black box"



start hebben tekens leest die we intypen, alle hoofdletters vervangt door kleine letters en het resultaat op het beeldscherm laat zien. De invoer is dan de tekst die we intypen en de uitvoer is de gewijzigde tekst. Heel veel Unix programma's werken op een soortgelijke manier: er gaat iets in en er komt iets uit en verder veranderen ze niets. Dit soort programma's worden *filters* genoemd, naar analogie van het koffiefilter. We hebben al een paar programma's gezien die op dezelfde manier werken: *ps* en *ls* bijvoorbeeld. Weliswaar gebruiken deze programma's geen invoer, maar dat mag natuurlijk ook. Ze produceren wel uitvoer. Een programma dat alleen maar invoer inleest en geen uitvoer produceert zou ook een filter zijn, maar is niet zo zinvol. Een heel simpel filter is het programma *cat* dat alleen maar de invoer naar de uitvoer copieert. Dat lijkt nu niet zo zinvol maar we zullen zien dat je dit ook op een zinvolle manier

kan gebruiken.

Het programma `cp` is *geen* filter. Het leest nl. niets van ons toetsenbord en het geeft geen uitvoer op het beeldscherm. Het verandert wel iets aan de files in onze computer en is daarom geen filter.

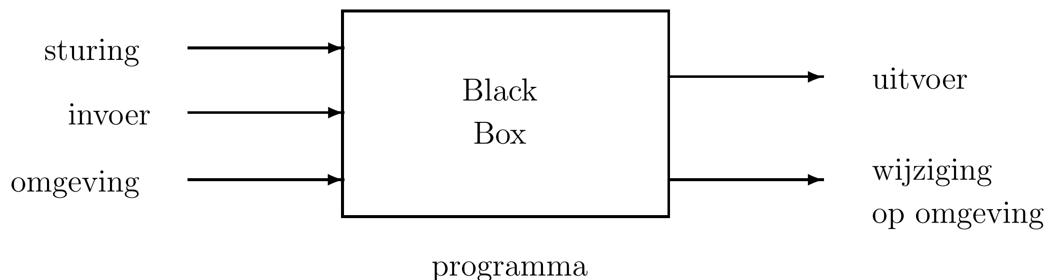
Opgave 2.1 Welke van de programma's genoemd in hoofdstuk 1 zijn filters?

2.1.1 Een uitgebreider model

We zagen in het voorgaande al dat het basismodel te simpel is. Het houdt bijvoorbeeld geen rekening met de omgeving waarin het proces draait, of die veranderd wordt, zoals bij het voorbeeld van `cp`.

Ons nieuwe model wordt (figuur 2.2): Hier zien we de volgende componenten:

Figuur 2.2 Het uitgebreide procesmodel



invoer informatie en/of data dat het programma "actief" inleest

uitvoer informatie die door het programma getoond wordt

sturing extra informatie die het gedrag van het programma beinvloedt, bijv. opties

omgeving De "passieve" omgeving die de werking van het programma beinvloed maar die niet expliciet gespecificeerd wordt, bijv. de draaiende processen die door het `ps` commando afgedrukt worden, of de files die het `ls` commando afdrukt.

wijziging op omgeving De veranderingen die het programma veroorzaakt, bijvoorbeeld de files die erbij komen of verdwijnen bij de commando's `cp`, `mv` en `rm`.

N.B. Soms is het niet helemaal duidelijk waar de grenzen liggen, maar dat is niet zo erg.

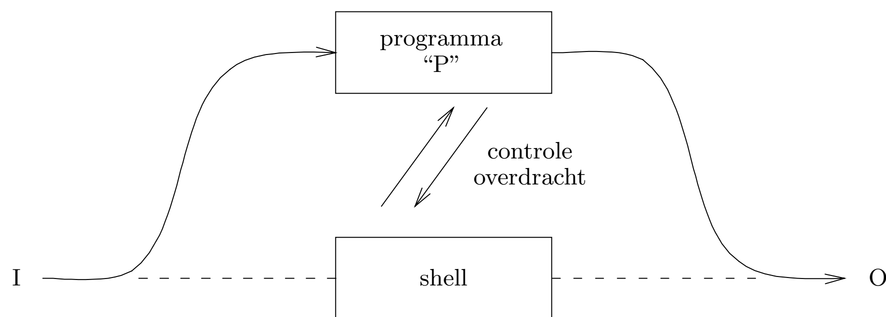
Omdat we graag de "aansluitingen" *invoer* en *uitvoer* in dit schema willen onderscheiden van andere gegevens die het programma leest en/of schrijft (b.v. rechtstreeks van/naar files), worden deze meestal *standaard invoer* en *standaard uitvoer* genoemd (in het engels *standard input* en *standard output*[1]).

[1]In C programma's zijn dit de files *stdin* resp. *stdout*, ofwel file descriptors 0 en 1.

2.1.2 De shell als proces

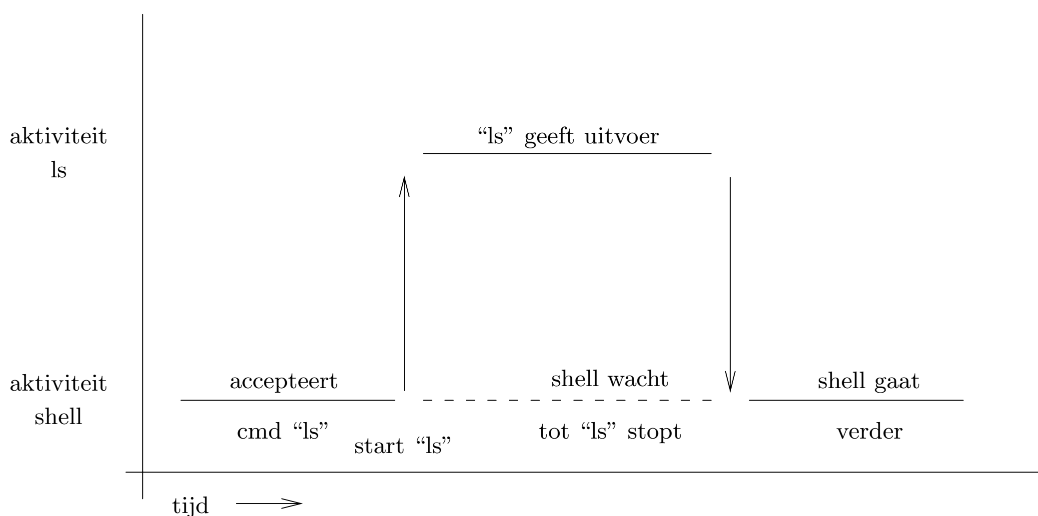
De shell is zelf ook een programma, en wanneer we commando's intypen is dat de invoer van het proces dat de shell draait. De shell kan ook uitvoer produceren, en het verandert ook de omgeving. Eén van de veranderingen is dat de shell een nieuw proces opstart als we een commando geven. Het shell proces stopt dan tijdelijk met zijn eigen aktiviteit en wacht tot het opgestarte proces beëindigd is, en gaat daarna weer verder. De invoer en uitvoer worden tijdelijk door het opgestarte programma overgenomen. We kunnen dat schematisch weergeven zoals in figuur 2.3.

Figuur 2.3 Opstarten van een proces



We kunnen dit gebeuren ook in een tijdsdiagram aangeven waarin de aktiviteiten van de verschillende processen staan (figuur 2.4). Als voorbeeld hebben we het commando *ls* genomen.

Figuur 2.4 Tijdsdiagram van het opstarten van een proces



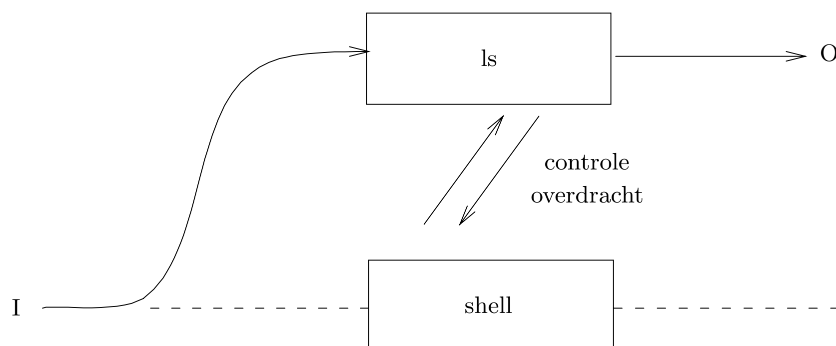
2.2 I/O redirection

I/O redirection ofwel het omleggen van in- en uitvoer is een krachtig mechanisme in Unix waarmee de in- en uitvoer van een proces (zoals aangegeven in figuur 2.1) veranderd kunnen worden zonder dat het programma aangepast hoeft te worden. Laten we eens met een voorbeeld beginnen: De opdracht `ls` geft een lijst van de filenamen in de werkdirectory. Deze lijst wordt afgedrukt op de uitvoer, bijvoorbeeld het beeldscherm. Soms is het nuttig om zo'n lijst te bewaren in een file. Het zou natuurlijk mogelijk zijn om het programma `ls` zo te schrijven dat het een optie krijgt om aan te geven dat de uitvoer naar een file moet worden gestuurd. Het is echter ook mogelijk om het volgende shell-commando te geven:

```
$ ls > lijst
```

De combinatie `> lijst` is een z.g. *output redirection*. Deze heeft tot gevolg dat de shell vóórdat het programma "ls" opgestart wordt eerst de uitvoer verlegt naar de file "lijst" en dan het proces opstart met de verlegde uitvoer. Het programma "ls" is zich er niet van "bewust" dat er iets veranderd is en schrijft vrolijk de lijst op zijn uitvoer, maar door de omlegging komt deze terecht op de file. Let er dus op dat de tekst `> lijst` niet aan het programma "ls" wordt doorgegeven maar dat dit door de shell verwerkt wordt. Het grote voordeel is dat je in één klap dit mechanisme beschikbaar hebt voor alle programma's die hun standaard uitvoer gebruiken (bijv. alle filters). Figuur 2.5 laat zien hoe de relatie zit. Na afloop van het proces "ls" is de uitvoer van de shell weer gewoon wat het daarvoor was. Strict genomen is de uitvoer van de shell zelf nooit veranderd geweest, er wordt alleen maar een lokale wijziging t.b.v. het nieuwe proces gemaakt. De filenaam mag ook zonder spaties na het `>` teken gegeven worden. Bij deze constructie wordt de vorige inhoud van de file weggegooid als die er al was. Als u die wilt laten staan, dus de uitvoer van het programma *achter* de bestaande inhoud zetten dan moet u de constructie `>> lijst` gebuiken. Als de file "lijst" niet bestaat maakt het niet uit welke van de twee u gebruikt.

Figuur 2.5 Output redirection



Op dezelfde manier is het mogelijk de invoer van een proces te verleggen. De notatie daarvoor is "`< file`". Ook hier mag de spatie na het `<` teken weggelaten worden. Als voorbeeld nemen we het programma `cat`. Dit programma copieert de standaard invoer naar de uitvoer, wat bij normaal gebruik niet zo interessant is. Zie het volgende voorbeeld:

```
$ cat
aap noot mies
aap noot mies
^D
$ cat > hulp
aap noot mies
^D
$ cat < hulp
aap noot mies
```

De procesplaatjes horend bij de drie voorbeelden zijn te zien in figuur 2.6.

2.2.1 Error uitvoer

Soms moet een programma aangeven dat er iets mis is, bijvoorbeeld een file die niet gevonden kan worden of een invoer die niet klopt. Voorbeeld:

```
$ ls xxx
Cannot access xxx: No such file or directory
```

Je wilt niet dat deze *foutmeldingen* verward kunnen worden met de normale uitvoer van het programma. De bovenstaande zin mag dan voor ons wel duidelijk zijn, maar als je de uitvoer van het *ls* programma wilt bewaren om later door een ander programma te laten bewerken, dan zou dit als een rijtje filenamen beschouwd kunnen worden, wat natuurlijk niet de bedoeling is. Daarom is er behalve standaard invoer en uitvoer ook nog een *standaard error* uitgang, waarom de meeste programma's hun foutmeldingen weergeven[2]. Als standaard uitvoer omgelegd wordt, dan blijft standaard error gewoon staan. Wel is het mogelijk standaard error apart om te leggen, bijvoorbeeld wanneer er zoveel foutmeldingen komen dat het beter is ze op een file op te slaan:

```
$ ls 2>foutlijst
```

schrijft de fouten naar de file "foutlijst" terwijl de uitvoer gewoon naar het scherm blijft gaan.

```
$ ls >lijst 2>foutlijst
```

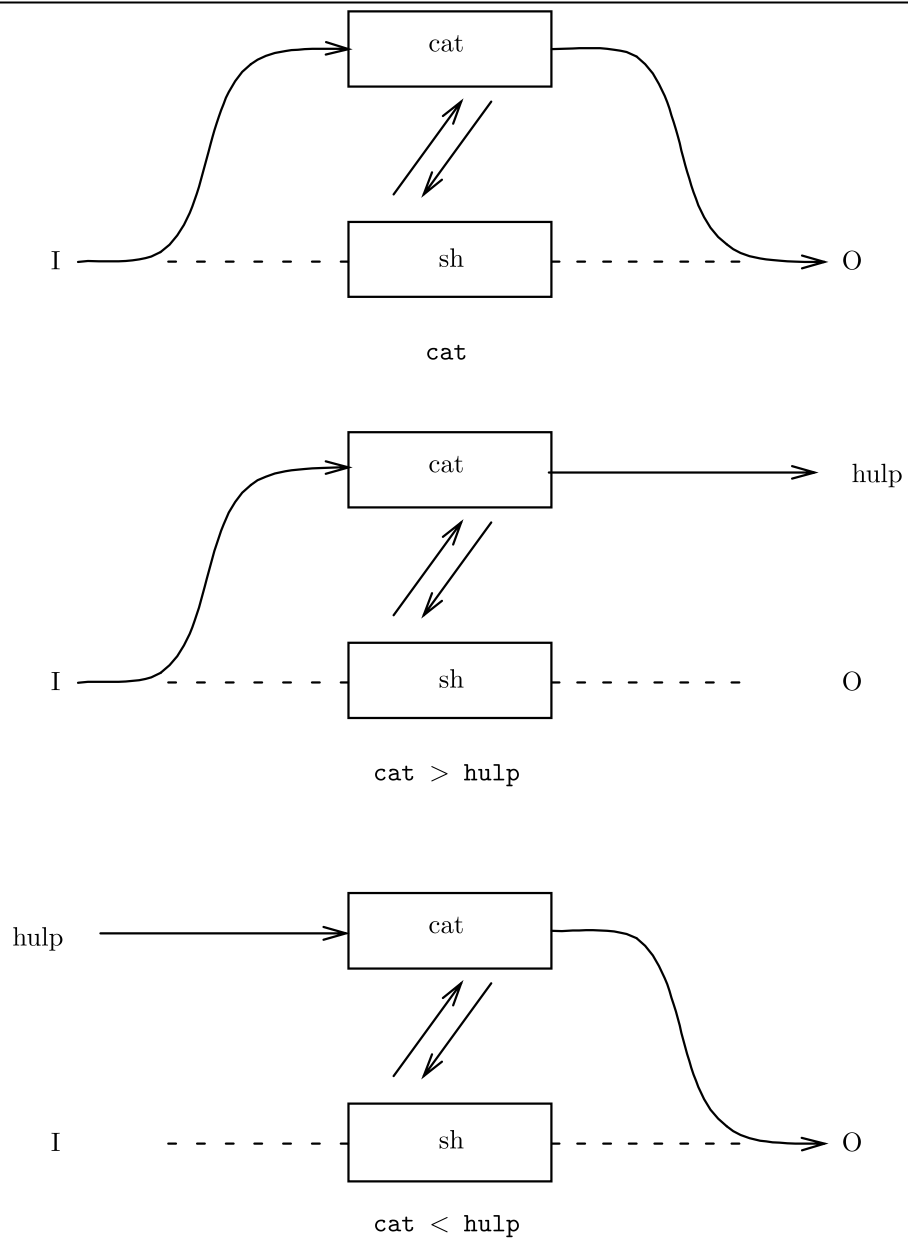
geeft de lijst van filenamen op de file "lijst" en de eventuele foutmeldingen op de file "foutlijst". De wat merkwaardige constructie

```
$ ls >lijst 2>&1
```

legt zowel de standaard uitvoer (1) als de standaard error (2) om naar dezelfde file.

[2]In C programma's *stderr* of file descriptor 2

Figuur 2.6 In- en Output redirection



2.2.2 Pipes

Pipes vormen een middel om twee programma's aan elkaar te koppelen en zijn eigenlijk een omlegging van zowel invoer als uitvoer. Het shell-commando:

```
$ programma1 | programma2
```

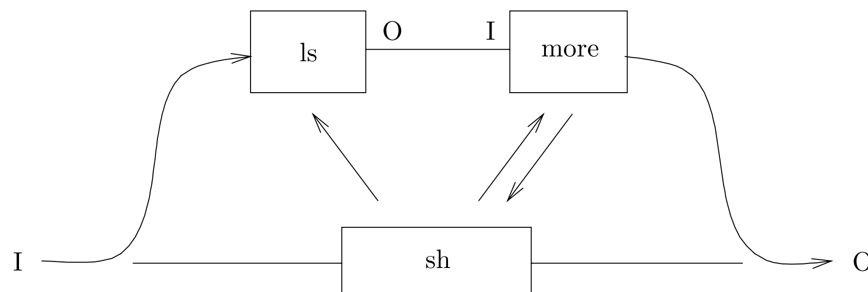
heeft tot gevolg dat twee processen opgestart worden, één met programma1 en één met programma2. De standaard uitvoer van programma1 wordt gekoppeld aan de standaard invoer van programma2. Hierdoor kan de uitvoer van het eerste programma direct verwerkt worden door het tweede, zonder dat er extra files nodig zijn[3]. Het uiteindelijke resultaat is hetzelfde als de volgende serie opdrachten:

```
$ programma1 > hulpfile
$ programma2 < hulpfile
$ rm hulpfile
```

behalve dat in het laatste geval de beide programma's ná elkaar gedraaid worden, terwijl bij het gebruik van de *pipe* ze *tegelijkertijd* gedraaid worden (dat is het voordeel van multitasking). Het Unix systeem synchroniseert de processen zo dat het eerste wacht wanneer er teveel uitvoer is dat nog niet door het tweede verwerkt is, en omgekeerd dat het tweede wacht wanneer het eerste niet genoeg geproduceerd heeft. Het is overigens op dezelfde manier ook mogelijk om meer dan twee processen in een pijpleiding aan elkaar te knopen. Een pipe kan niet samen met output redirection op het eerste programma (logisch) maar wel met input redirection. Evenzo kan er bij het laatste programma geen input redirection maar wel output redirection gebruikt worden. Standaard error redirection kan wel plaatsvinden (bij elk van de onderdelen).

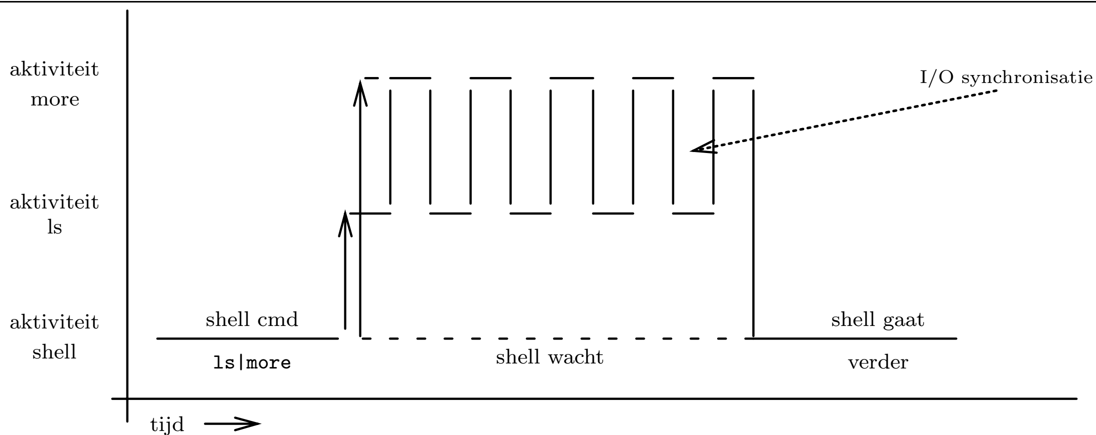
Het procesplaatje voor het commando `ls | more` staat in figuur 2.7 en het tijdsdiagram in figuur 2.8. ("More" is een programma dat zijn invoer in brokken laat zien ter grootte van een scherm. Dit wordt meestal gebruikt om de uitvoer van een ander programma te kunnen bekijken zonder dat het van het scherm afvliegt).

Figuur 2.7 Processen met pipe



[3] Op sommige andere OS's wordt deze constructie uitgevoerd doordat het OS stiekem wel een file voor je maakt.

Figuur 2.8 Processen met pipe (tijdsdiagram)



Opgave 2.2 Wat laat het commando:
$ *ps | more* zien? Bedenk eerst zelf het antwoord en probeer het daarna.

2.3 Achtergrondprocessen

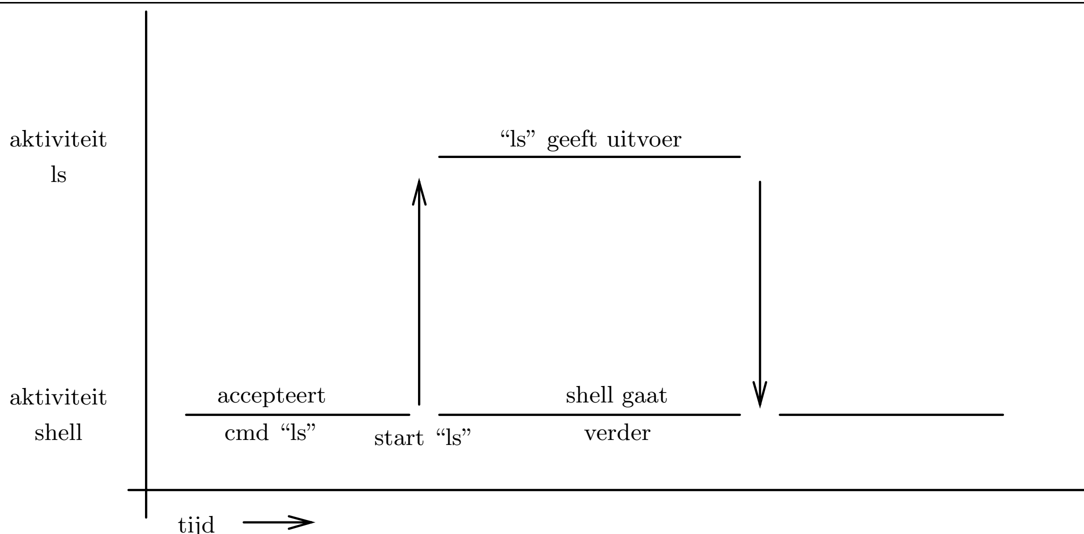
In de voorbeelden die we tot nu toe gezien hebben wacht de shell na het opstarten van een proces tot dit klaar is voor er weer een nieuwe opdracht geaccepteerd wordt. In een multitasking systeem is dit niet echt nodig en er zijn omstandigheden waar dit ongewenst is. Stel bijvoorbeeld dat een groot programma gecompileerd moet worden dan is het jammer als je tijdens het compileren niets kunt doen. Weliswaar kun je het te compileren programma nog niet gebruiken maar het zou mogelijk zijn om bijvoorbeeld je electronische post intussen af te werken. Op een werkstation met verschillende windows kan dat in een ander window, maar als je op een simpele terminal zit te werken dan kan dat niet. Er zijn ook andere situaties denkbaar waarbij je een commando wilt geven maar niet wilt wachten tot het klaar is, maar direct nieuwe wilt uitvoeren. Hiervoor heeft de shell een mogelijkheid om aan te geven dat het programma "in de achtergrond" uitgevoerd moet worden. Door achter de opdracht het teken `&` te zetten geef je aan de shell te kennen dat het commando uitgevoerd moet worden zonder wachten. In dit geval drukt de shell het proces nummer (PID) af van het nieuwe proces zodat we later nog naar dit proces kunnen refereren. Dit nummer is hetzelfde als wordt afgedrukt door het *ps* commando. Als het opgestarte commando uitvoer genereert dan kan deze door de uitvoer van andere programma's of door onze volgende opdrachten heen geprint worden. Daarom is het vaak handiger om achtergrondprocessen met output redirection te doen. Zie het volgende voorbeeld:

```
$ ls
file1 file2
$ ls &
[1] 27472
$ file1 file2
ls > lijst &
[2] 27474
[1] - Done ls &
$
```

In dit voorbeeld zien we dat de uitvoer van het eerste achtergrondprogramma op de plaats komt waar we normaal het volgende command intypen. Verder zien we dat de shell het procesnummer geeft (27472 resp. 27474). Hier geeft de shell ook nog een "intern" nummer tussen "[ ]" (een z.g. *job nummer*) maar niet alle shells doen dit. En deze shell geeft ook nog een seintje als het achtergrond proces klaar is. Wanneer een pipeline in de achtergrond gestart wordt ontstaan er natuurlijk meer processen maar de shell geeft alleen het nummer van het laatste proces.

In figuur 2.9 vind je het tijdsdiagram behorend bij het commando `ls &`.

Figuur 2.9 Tijdsdiagram voor een achtergrondproces



2.3.1 Het stoppen van processen

Soms is het nodig een opgestart proces te stoppen. Het kan bijvoorbeeld zijn dat je een vergissing hebt gemaakt (het verkeerde commando gestart) en dat je het wilt stoppen voordat het teveel schade aanricht. Of je hebt een eigen programma gestart dat weigert te stoppen. Of je hebt gewoon geen zin te wachten tot het programma beëindigd is. Enz. . .

Er zijn verschillende manieren om dit te doen:

- Een proces dat "normaal" opgestart is, d.w.z. zonder `&`, dus als voorgrond proces kan vaak afgebroken worden door `^C` in te typen. Het kan zijn dat dit teken door een ander vervangen is. Je kunt dit uitvinden door de opdracht *stty* of *stty -a* te geven en te zoeken naar iets als `intr =`, daarachter staat dan het betreffende teken. `^C` heeft tot gevolg dat het proces onherroepelijk gestopt is (sommige programma's schakelen echter het gebruik van `^C` uit).

- Een minder drastische maatregel kan bij sommige shells genomen worden door `^Z` te typen, in dat geval wordt het voorgrond proces wel gestopt, maar kan later nog doorgestart worden. In dat geval zal de shell een boodschap geven zoals:
  `[1] + Stopped`
  Het getal tussen de [ ] haken is weer een job nummer. Ook hier kan het voorkomen dat ip.v. `^Z` een ander teken genomen moet worden. Via het *stty* commando kan dat ook gevonden worden. Kijk naar de `susp =` of `swtch =` waarde.

- Een achtergrondprocess reageert niet op `^C` of `^Z`. Het kan echter gestopt worden met het *kill* commando met het procesnummer als argument (dat is één van de redenen waarom de shell dat nummer geeft). Als je het procesnummer niet meer weet dan kan je het misschien vinden via het *ps* commando. Voorbeeld:

  ```
  $ ls > lijst &
  [2] 28027
  $ kill 28027
  ```

  Als de shell jobnummers geeft kan je deze ook gebruiken door er een *%* teken voor te zetten, dus in bovenstaand voorbeeld *kill %2*. Als een job uit meer dan één proces bestaat (bijvoorbeeld bij een pipeline) dan slaat het commando op alle processen in die job.

  Sommige programma's reageren niet op het gewone *kill* commando, in dat geval kun je *kill -KILL* proberen (weer gevolgd door het nummer). Het kill commando heeft hetzelfde effect als `^C` dus het proces is daarna verdwenen.

- Een achtergrond job kan gestopt worden zonder het te laten verdwijnen met het *stop* commando met het procesnummer of *%*⟨jobnummer⟩.

- Gestopte jobs kunnen doorgestart worden als achtergrond job door het commando *bg %*⟨jobnr⟩ of als voorgrond job d.m.v het commando *fg %*⟨jobnr⟩. In het eerste geval geldt het alsof de job oorspronkelijk met `&` opgestart was, in het laatste geval alsof het zonder `&` opgestart was.

- Mocht je niet meer weten welke jobs je hebt lopen en in wat voor toestand ze staan dan geeft het commando *jobs* je een lijstje:

```
$ ls > lijst &
[1]     28095
$ cc test.c &
[2]     28096
$ ls > lijst2
^Z
[3] + Stopped                  ls -lR
$ stop %1
$ jobs
[3] + Stopped                  ls > lijst2
[2] - Running                 cc test.c &
[1]   Stopped (signal)         ls > lijst & $
```

Hoofdstuk 3

Files

In dit hoofdstuk gaan we dieper in op de structuur van files in het Unix systeem en op de commando's om files te manipuleren.

3.1 De inhoud van files

Zoals we al in hoofdstuk 1 gezien hebben kan een file allerlei soorten informatie bevatten. Het Unix systeem houdt echter niet voor ons bij wát voor soort informatie er in een file zit. De informatie zal vaak tekst zijn, die we met een editor kunnen bewerken of op een printer afdrukken. Als we de editor echter een andere file willen laten bewerken, bijvoorbeeld een executeerbaar programma, of een plaatjesfile, dan kunnen er wel eens vreemde tekens op het scherm komen. Het zou zelfs kunnen gebeuren dat de editor of printer vast komen te zitten.

Als we een tekstfile hebben (en dan kun je eigenlijk alleen goed controleren door de file in een editor te nemen en er goed naar te kijken) dan kan die tekst weer voor verschillende doeleinden te gebruiken zijn: Het kan gewoon een kladnotitie zijn, of een programma, geschreven in C of een andere programmeertaal. Er is een commando *file* dat probeert te gissen wat voor soort informatie er in een file zit, maar dat er ook nogal eens naast zit.

```
$ file test.c
test.c:       c program text
```

Tekst files kun je ook op andere manieren bekijken. We hebben al het programma *more* gezien waarmee we een file scherm voor scherm kunnen bekijken, en *cat* waarmee je een file naar het scherm kunt copiëren. Beide programma's gebruiken een interessante manier om invoer-argumenten aan te geven, die ook door veel andere Unix programma's gebruikt wordt (maar helaas niet door alle). We zullen *cat* als voorbeeld gebruiken.

We hebben al gezien dat *cat < mijnfile* de inhoud van "file" copieert naar de standaard uitvoer. In dit geval weet *cat* niet dat de invoer van "mijnfile" komt. Het is echter ook mogelijk om te zeggen: *cat mijnfile*. In dat geval ziet *cat* wèl dat om "mijnfile" gevraagd wordt. Er mag zelfs een heel rijtje files opgegeven worden en in dat geval zal *cat* alle files één voor één lezen en de inhoud van elke file naar de standaard uitvoer copieren. Op de standaard uitvoer komt dus de aan elkaar geplakte informatie van alle opgegeven files. Daar komt ook

de naam van het programma vandaan: *cat* *concateneert* de files.

Deze programma's hebben dus de volgende conventie: Als er één of meer filenamen opgegeven zijn dan worden deze files gelezen. Als er geen filenaam opgegeven is dan wordt van standaard invoer gelezen. Let erop dat in het geval van *cat < mijnfile* er geen filenaam aan *cat* gegeven wordt, het is dan de shell die de file koppelt aan de standaard invoer. Het uiteindelijke effect van *cat mijnfile* is natuurlijk wèl hetzelfde als dat van *cat < mijnfile*.

Opgave 3.1 Wat is het effect van *cat mijnfile mijnfile > jouwfile* ?

We gaan nu de inhoud van een file wat gedetailleerder bekijken. Zie figuur 3.1.

Figuur 3.1 Inhoud van een file

```
$ cat > vb
voorbeeld
$ cat vb
voorbeeld
$ od -bc vb
0000000 166 157 157 162 142 145 145 154 144 012
          v   o   o   r   b   e   e   l   d  \n
0000012
```

Eerst maken we een file "vb" met een stukje tekst. De tweede opdracht laat zien dat de tekst inderdaad in de file staat. Met de volgende opdracht laten we iets meer van de interne struktuur zien. Het commando *od* (*octal dump*) kan de inhoud van een file tonen in getalvorm. Iedere file bestaat uit een verzameling *bytes*. Een byte kan als teken geinterpreteerd worden maar ook als een (klein) getal. En byte bestaat uit 8 *bits* waarbij elke bit twee waarden kan aannemen: 0 of 1. Een byte kan daarom $2^8 = 256$ waarden aannemen, die we meestal aangeven als 0 t/m 255. Het is vaak handiger om de waardes van een byte niet in decimale notatie aan te geven maar in *octale* of *hexadecimaal* notatie (zie appendix A). De opdracht *od -bc vb* geeft de bytes van de file "vb" in zowel octale notatie (*-b*) als character notatie (*-c*). De "opties" *-b* en *-c* zijn hier gecombineerd, maar je kunt ook één van de twee opgeven.

Als we eerst naar de character notatie kijken dan zien we behalve de tekst die we ingevoerd hebben nog een extra teken dat aangegeven wordt door `\n`. Dit is het "newline" teken dat aan het eind van iedere regel staat. Er is geen speciaal teken voor het einde van de file.

De tekens van een tekst file zijn meestal weergegeven d.m.v. de z.g. ASCII[1] code. Hierin zijn 32 z.g. control codes opgenomen waarvan er maar enkele gebruikt worden (o.a. LineFeed voor "newline") en 96 teksttekens waaronder de spatie. Zie tabel 3.1 voor de octale en 3.2 voor de hexadecimale waarden van de tekens. Om de waarde van een teken te krijgen moet je de getallen links en boven of rechts en beneden optellen.

Er is ook een uitgebreidere codering met o.a. letters met accenten erin, de z.g. ISO[2] 8859-1 of ISO-Latin1 code. Deze wordt steeds meer gebruikt. Deze code bevat 256 elementen

[1]American Standard Code for Information Interchange.
[2]International Standards Organization

+	000	001	002	003	004	005	006	007
000	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
010	BS	HT	NL	VT	NP	CR	SO	SI
020	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
030	CAN	EM	SUB	ESC	FS	GS	RS	US
040	SP	!	"	#	$	%	&	’
050	(	)	*	+	,	−	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	‘	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	DEL

Tabel 3.1: ASCII code tabel (octaal)

+	00	01	02	03	04	05	06	07	
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	
	BS	HT	NL	VT	NP	CR	SO	SI	00
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	
	CAN	EM	SUB	ESC	FS	GS	RS	US	10
20	SP	!	"	#	$	%	&	’	
	(	)	*	+	,	−	.	/	20
30	0	1	2	3	4	5	6	7	
	8	9	:	;	<	=	>	?	30
40	@	A	B	C	D	E	F	G	
	H	I	J	K	L	M	N	O	40
50	P	Q	R	S	T	U	V	W	
	X	Y	Z	[	\	]	^	_	50
60	‘	a	b	c	d	e	f	g	
	h	i	j	k	l	m	n	o	60
70	p	q	r	s	t	u	v	w	
	x	y	z	{	|	}	~	DEL	70
	08	09	0A	0B	0C	0D	0E	0F	+

Tabel 3.2: ASCII code tabel (hexadecimaal)

+	00	01	02	03	04	05	06	07	
80	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	
	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3	80
90	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	
	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC	90
A0	NBSP	¡	¢	£	¤	¥	¦	§	
	¨	©	ª	«	¬	-	®	¯	A0
B0	°	±	²	³	´	µ	¶	·	
	¸	¹	º	»	1/4	1/2	3/4	¿	B0
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	
	È	É	Ê	Ë	Ì	Í	Î	Ï	C0
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	
	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	D0
E0	à	á	â	ã	ä	å	æ	ç	
	è	é	ê	ë	ì	í	î	ï	E0
F0	ð	ñ	ò	ó	ô	õ	ö	÷	
	ø	ù	ú	û	ü	ý	þ	ÿ	F0
	08	09	0A	0B	0C	0D	0E	0F	+

Tabel 3.3: ISO 8859-1 code tabel (hexadecimaal)

waarvan de eerste 128 hetzelfde zijn als ASCII. Daarna komen er dan nog 32 nieuwe control tekens, en dan nog 96 afdrukbare tekens. Zie tabel 3.3 voor het niet-ASCII gedeelte. Omdat deze verzameling zelfs voor de Europese talen niet voldoende is, is er een nieuwe codetable, Unicode of ISO-10646 genoemd waarin plaats is voor de tekens van alle in gebruik zijnde talen. Echter hiervoor geldt dat per teken 16 bits nodig zijn. Deze tabel is te groot om hier op te nemen.

3.2 Meer informatie over files

Het *ls* commando kan meer informatie over één of meer files geven. Dit commando heeft veel opties waarvan we er enkele zullen bespreken. De meest gebruikte is *ls -l* die uitgebreide informatie over de files in de werkdirectory geeft, of – wanneer één of meer files opgegeven worden – over die files.

```
$ ls -l vb
-rw-r--r--    1 piet     staff          10 Jan 16 14:10 vb
```

De uitvoer bevat een aantal “kolommen” met informatie:

1. *mode:* De eerste kolom bevat een rijtje met letters of streepjes. Het eerste teken geeft aan of dit een “gewone” of een “speciale” file is. Speciale files zijn o.a. directories die met een “d” aangegeven worden. Gewone files worden aangegeven door een “-” teken. Verder wordt “l” gebruikt voor een symbolische link (zie 3.5.2), de andere zijn erg exotisch.

De volgende tekens in de eerste kolom geven de beveiliging van de file aan. Ook wel *protecties* of *permissies* genoemd. Deze tekens zijn in groepjes van 3 opgedeeld. Elk groepje bevat "rwx" of een selectie hieruit. De eerste 3 geven aan wat de eigenaar van de file mag doen, de tweede groep van 3 geven aan wat de "groepsgenoten" mogen, en de laatste groep van 3 wat de rest van de gebruikers mogen. Voor dit gebruik (en voor andere beveiligingen) zijn de gebruikers van een Unix systeem in "groepen" ingedeeld. Elke groep heeft een naam, bijv. "student" voor studenten en "staff" voor stafleden. Verderop in bovenstaande uitvoer zien we dat deze file bij de groep "staff" hoort, en het tweede groepje "rwx" tekens slaat dus op de groep "staff". De "rwx" letters hebben de volgende betekenis voor gewone files: r=read, dat wil zeggen dat de informatie gelezen mag worden, w=write, dat wil zeggen dat de file veranderd mag worden, en x=execute, dit betekent dat de file een commando is dat uitgevoerd kan worden. Voor directories hebben de letters een iets andere betekenis, daar komen we verderop nog op terug. In bovenstaande voorbeeld geldt dus dat de eigenaar (gebruiker "piet") de file mag lezen en schrijven, en dat de groepsgenoten en de rest van de gebruikers alleen mogen lezen.

2. *aantal links:* Dit getal is op dit moment niet interessant, maar zie 3.5.1.
3. *eigenaar:* De loginnaam van de eigenaar van de file.
4. *groep:* De naam van de groep (zoals hiervoor besproken).
5. *grootte:* Het aantal bytes dat de file bevat.
6. *datum/tijd:* De datum en tijd dat de file aangemaakt is of de laatste keer gewijzigd (modificatietijd)
7. *naam:* De naam van de file.

Het `ls` commando heeft vele opties waarvan we in tabel 3.4 de belangrijkste geven: Bij dit commando mogen de opties gecombineerd worden, dus `ls -ltu` is hetzelfde als `ls -l -t -u`. (Dit kan niet bij alle commando's gedaan worden).

Een aantal van de bovengenoemde eigenschappen ("attributen") van een file kunnen met commando's gewijzigd worden:

- `chown` ⟨naam⟩ ⟨file⟩ ...
  Wijzig de eigenaar van één of meer files. De vorige eigenaar heeft daarna alle zeggenschap over de file verloren (en kan het dus niet meer terugzetten). Alleen gebruiken als je zeker bent dat je dit wilt en als de file in een geschikte directory staat.
- `chgrp` ⟨naam⟩ ⟨file⟩ ...
  Wijzig de groep van één of meer files.
- `chmod` ⟨mode⟩ ⟨file⟩ ...
  Wijzig de mode (permissies) van één of meer files. De ⟨mode⟩ kan op twee manieren opgegeven worden:
  1. Als een octaal getal van 3 cijfers: elk cijfer stelt een groepje van permissies voor, van links naar rechts voor de eigenaar, groep en de rest. Elk cijfer is opgebouwd door de getalwaarden van "rwx" op te tellen waarbij r=4, w=2, x=1. Dus 641="`rw-r----x`".

-l	Geef veel informatie, i.p.v. alleen de file naam
-a	Geef alle filenamen, ook die met een "." beginnen. Normaal worden namen die met een "." beginnen niet getoond.
-t	Sorteer op tijd (meestal de modificatietijd, tenzij -u gegeven wordt). Dit gebeurt op afnemende tijd, dus de jongste files eerst. Zonder "-t" wordt op alfabetische volgorde gesorteerd.
-u	Gebruik de tijd van het laatste *gebruik* ("use") van de file i.p.v. de modificatietijd. Dit geldt zowel voor het sorteren bij "-t" als voor het afdrukken bij "-l".
-r	Keer de sorteervolgorde om.
-d	Als de filenaam die als argument meegegeven wordt een directory is, dan wordt een regel informatie over deze directory afgedrukt. Normaal wordt in zo'n geval informatie over alle files in de directory afgedrukt.
-R	Voor elke directory die opgegeven wordt, worden ook alle files in die directory afgedrukt, en alle files in subdirectories in die directory, enz.

Tabel 3.4: De belangrijkste opties van *ls*

2. Symbolisch als ⟨wie⟩=⟨rwx⟩, waarbij ⟨wie⟩ kan zijn: "u" (user) voor de eigenaar, "g" voor de groep en "o" (other) voor de rest. Combinaties zijn ook toegestaan, en de afkorting "a" (all) kan gebruikt worden i.p.v. "ugo", maar deze mag ook weggelaten worden. ⟨rwx⟩ is een selectie uit "rwx". Dus het bovenstaande voorbeeld (`rw-r----x`) kan verkregen worden met *chmod u=rw,g=r,o=x file*. I.p.v. "=" mag ook "+" gebruikt worden om permissies toe te voegen, en "−" om permissies weg te halen. Bijv. *chmod +r,go-w file* geeft iedereen "read" permissie, maar verwijdert de "write" permissie voor "group" en "other".

Opgave 3.2 Hoe kun je de grootte van een file en de filenaam wijzigen?

Voor tekst files kun je extra informatie krijgen d.m.v. het *wc* ⟨file⟩ ... (word count) commando. Dit telt het aantal woorden en regels in de file(s). Een "woord" is voor dit gebruik gedefinieerd als een rijtje tekens wat gescheiden wordt door wit-ruimte (spaties, tabs, nieuwe regels, begin en einde van de file). Hoewel dit commando ook voor niet-tekst files gebruikt kan worden geeft het daarvoor geen zinnige informatie.

Zonder opties worden 3 getallen gegeven (aantal regels, woorden, characters), gevolgd door de filenaam. Je kunt een selectie van de getallen krijgen (of een andere volgorde) door de opties *-l* (lines), *-w* (words) of *-c* (characters) op te geven of een combinatie ervan. Ook bij dit commando mogen de opties gecombineerd worden. Het commando *wc* is ook als filter te gebruiken en geeft dan de informatie over de standaard invoer (er wordt dan geen filenaam afgedrukt). Zo kan het aantal files in de werkdirectory bepaald worden met het commando *ls | wc -l*.

3.3 Directories

Een directory is intern in het Unix systeem een lijst van files en (sub)directories. Deze informatie staat echter niet in een leesbaar formaat en kan alleen met speciaal daarvoor ontworpen commando's gemanipuleerd worden. We hebben al het *ls* commando gezien waarmee we de inhoud van een directory zichtbaar kunnen maken.

Een nieuwe directory kunnen we maken met het commando *mkdir* ⟨naam⟩ .... Een directory kun je ook verwijderen met het commando *rmdir* ⟨naam⟩ ..., maar dit mag alleen maar als de directory leeg is. Dus eventuele files erin moeten eerst met het *rm* commando verwijderd worden, en evt. subdirectories met *rmdir* (waarvoor natuurlijk weer hetzelfde geldt).

Protecties van een directory hebben een iets andere betekenis dan voor gewone files. De protecties kunnen getoond worden met het commando *ls -ld* ⟨naam⟩ en gewijzigd worden met *chmod* net als voor gewone files.

De "r" permissie voor een directory betekent dat de informatie van de directory opgevraagd mag worden, dus er mag een lijst van filenamen gegeven worden (bijv. met het *ls* commando). Als de "r" permissie niet aanstaat dan kan ls niet de lijst van filenamen geven.

De "w" permissie betekent dat de inhoud van de directory veranderd mag worden, concreet betekent dit dat er files aangemaakt, verwijderd of hernoemd mogen worden in de directory. In het bijzonder is het zo dat files die geen "w" permissie hebben maar staan in een directory die wel "w" permissie heeft, verwijderd mogen worden of een andere naam kunnen krijgen, terwijl de file zelf niet veranderd mag worden. Dat kan tot onverwachte beveiligingsproblemen leiden, want iemand die de file niet mag veranderen zou in dat geval wel de file kunnen verwijderen en vervangen door een file met dezelfde naam maar een andere inhoud!! Moderne Unix systemen hebben soms extra voorzieningen om dit soort problemen te voorkomen.

De "x" permissie op een directory betekent dat je "door de directory heen mag gaan", d.w.z. als je een de naam van een file in de directory weet dan kun je die filenaam gebruiken. Wat je met de file zelf mag doen hangt natuurlijk weer af van de permissies van die file. Het verschil tussen de "r" en de "x" permissies op een directory is dus dat "r" je toestaat om de filename uit te vinden en dat "x" je toestaat om ze te gebruiken. Meestal worden deze permissies samen gebruikt (of samen uitgezet). De permissie alleen "x" kan soms handig zijn om één of enkele personen toegang te geven tot een file, als je hiervoor niet de groepsstructuur kunt gebruiken. Je verzint gewoon een niet-raadbare filenaam (zoiets als "BoHEr@:xg.,Pd") en vertelt die aan je vrienden, je zet de file in een directory met permissie "--x". Zie figuur 3.2 voor een voorbeeld van het effect van het ontbreken van de permissies "r" en "x".

3.4 Devices

Devices (apparaten) in het Unix systeem hebben ook een filenaam, meestal in de directory `/dev`. Met het *ls -l* commando is te zien dat deze "files" geen gewone files zijn aan het eerste teken van de uitvoer. Voor een "device" staat deze op "b" of "c". Het verschil tussen deze is voor ons niet van belang.

De meeste devices zijn voor gewone gebruikers niet toegankelijk omdat je dan het systeem zou kunnen verstoren. En zijn er echter een paar die van belang zijn:

Figuur 3.2 Directory permissies "r" en "x"

```
$ mkdir hulp
$ cat > hulp/tekst
een tekst file
^D
$ ls hulp
tekst
$ chmod -r hulp
$ ls -ld hulp
d-wx--x--x    2 piet     staff        512 Jan 18 10:58 hulp
$ ls hulp
Cannot access directory hulp: Permission denied
$ ls -l hulp/tekst
-rw-r--r--    1 piet     staff         15 Jan 18 10:58 hulp/tekst
$ chmod 400 hulp
$ ls hulp
tekst
$ ls -l hulp/tekst
Cannot access hulp/tekst: Permission denied
```

/dev/tty is altijd de terminal waarop je zit te werken. Je kunt dus altijd iets naar je terminal schrijven met *echo boodschap > /dev/tty*, of van je terminal lezen met *...< /dev/tty* zelfs al zit je in een situatie waar je niet meer weet hoe de standaard invoer en/of uitvoer staat. Als je in een window werkt dan is "/dev/tty" aan je window gekoppeld, en in verschillende windows kan het dus iets anders betekenen.

/dev/tty⟨xx⟩ wordt gebruikt voor een specifieke terminal. Dus elk window heeft zijn eigen ⟨xx⟩ nummer. Soms wordt ook /dev/pty⟨xx⟩ gebruikt. Wees voorzichtig met het gebruik hiervan, want misschien werkt er iemand anders op deze terminal.

/dev/null is het "zwarte gat". Als je leest van "/dev/null" dan krijg je onmiddelijk een end-of-file (dus je leest een lege file), en alles wat geschreven wordt naar "/dev/null" verdwijnt onmiddelijk. Dit kan handig zijn als je een programma wilt draaien maar je wilt de uitvoer of de foutmeldingen niet zien. Je gebruikt dan *> /dev/null* of *2>/dev/null*.

3.5 Links

In het Unix systeem is het mogelijk dat een file onder meer dan één naam bekend is. In dat geval staat er dus in een directory, of in verschillende directories, meer dan één ingang voor de file. We spreken dan van een link. Het gaat dan dus om één file, niet om meer copieën van dezelfde file. Het commando *ln* kan gebruikt worden om links aan te maken. Er zijn twee soorten links: *harde* en *symbolische*.

3.5.1 Harde links

Een harde link wordt gemaakt door de opdracht

> *ln* ⟨file1⟩ ⟨file2⟩

Hierbij is ⟨file1⟩ een bestaande file en ⟨file2⟩ is de nieuwe naam de gecreëerd moet worden. De nieuwe filenaam mag in een andere directory zitten dan de oude, maar dat gaat niet altijd goed. Kort gezegd moeten de beide directories "bij elkaar in de buurt" zitten[3]. In het algemeen geldt dat één gebruikersdirectory structuur goed genoeg is.

Het maakt geen verschil of eerst ⟨file1⟩ gemaakt wordt en dan *ln* ⟨file1⟩ ⟨file2⟩ gegeven of omgekeerd. In beide gevallen hebben we dezelfde file met twee namen (meer mag ook). Je kunt het vergelijken met een bedrijf dat meer dan één verwijzing in het telefoonboek heeft, bijvoorbeeld onder "Boekhandel" en onder zijn naam. Elke wijziging op de inhoud van de file via de ene naam zal ogenblikkelijk zichtbaar zijn via de andere naam. Alleen een hernoeming van één van de namen (via het *mv* commando) zal geen effect hebben op de andere naam. Je kunt één van de namen verwijderen met het *rm* commando, maar zolang er nog minstens één andere link bestaat blijft de file bestaan. Pas wanneer de laatste harde link verwijderd wordt verdwijnt ook de file.

In de uitvoer van *ls -l* hebben we als tweede kolom het aantal links gezien (dit is dus het totaal aantal namen dat de file heeft). Dit kan handig zijn om te weten of er nog meer links bestaan, al kan dit commando je niet vertellen hoe de andere links heten (daar is geen commando voor).

Je kunt met het *ln* commando ook een hele rij files een nieuwe link geven in een andere directory, net als bij het *cp* of *mv* commando (zie 3.7). Dus *ln file1 file2 dir* maakt in de directory "dir" nieuwe links voor file1 en file2. De links hebben dezelfde naam als de originele files (ze verschillen alleen van directory).

Het is niet mogelijk om een bestaande directory een nieuwe naam te geven via een harde link, dus *ln dir newdir* waarbij "dir" een directory is, is niet toegestaan. Dit is gedaan om te voorkomen dat de directory structuur een onontwarbare knoop wordt. Toch zie je bij een *ls -l* uitvoer voor directories vaak wel een aantal links staan dat groter dan 1 is. Dit komt omdat het Unix systeem wèl een aantal links voor directories aanmaakt, nl:

- De naam "." in de directory zelf is een link naar de directory.
- In elke subdirectory is de naam "⟨subdir⟩/.." ook een link naar de directory waar ⟨subdir⟩ inzit. Dus als een directory n subdirectories heeft, is de link-count $n + 2$.

Als gebruiker kun je hier verder niets aan veranderen.

3.5.2 Symbolische links

Een *symbolische* link (ook wel eens *zacht* genoemd) is een indirecte verwijzing naar een andere file of directory. Het kan vergeleken worden met een verwijzing in het telefoonboek, waar

[3] Technisch gezien moeten ze op dezelfde disk partitie zitten.

bij “Stadhuis” staat: “Zie Gemeente”. Een symbolische link kan gemaakt worden met het commando

ln -s ⟨file1⟩ ⟨file2⟩

waarbij ⟨file1⟩ de bestaande file of directory is en ⟨file2⟩ de nieuwe verwijzing wordt.

Er zijn een aantal belangrijke verschillen tussen harde en symbolische links:

- De beperkingen voor harde links gelden niet voor symbolische. Dus ook verwijzingen naar directories zijn toegestaan, en de eis dat de beide namen “dicht bij elkaar” moeten zijn is er niet.
- Symbolische links worden niet meegeteld in de link count
- Het maakt wèl verschil uit in welke volgorde de link gemaakt wordt
- Als een file waarnaar alleen symbolische links bestaat (dus geen harde) verwijderd wordt, is hij echt weg, ook al zijn er nog symbolische links naar toe. De symbolische links hangen dan dus in het luchtledige.
- Als de file waarnaar een symbolische link verwijst een andere naam krijgt verwijst de symbolische link niet meer naar die file. Je moet dan een nieuwe symbolische link maken. Omgekeerd als je een andere file hernoemt naar de naam waar een symbolische link naar verwijst dan wijst de symbolische link voortaan naar die andere file
- In een *ls -l* lijst wordt door een “l” als eerste teken aangegeven (in de “mode” kolom) dat het niet een gewone file is maar een symbolische link en de symbolische link wordt weergegeven als ⟨symbolische naam⟩ `->` ⟨echte naam⟩
- Je kunt bovenstaande samenvatten door te zeggen dat symbolische links niet naar files verwijzen maar naar *filenamen.*
- Het is toegestaan dat een symbolische link verwijst naar een andere symbolische link. De keten moet uiteindelijk op een echte file uitkomen. Meestal is er een beperking op het aantal verwijzingen dat je zo mag hebben (bijv. niet meer dan 8).

Zie onderstaand voorbeeld:

```
$ ln -s hulp/tekst symb
$ ls -l symb
lrwxr-xr-x    1 piet     staff  10 Jan 18 13:33 symb -> hulp/tekst
$ ls -l hulp/tekst
-rw-r--r--    1 piet     staff  15 Jan 18 10:58 hulp/tekst
```

Zoals te zien is, is het aantal links voor “hulp/tekst” niet veranderd.

3.6 Wildcards

Dit onderwerp hoort eigenlijk bij de shell thuis, maar omdat het ook over files gaat wordt het hier kort ingeleid. Meer informatie vind je in hoofdstuk 4.

Het gebeurt nogal eens dat we eenzelfde operatie willen doen op een aantal files die allemaal dezelfde soort namen hebben. Bijvoorbeeld namen die eindigen op ".c" of allemaal met "test" beginnen.

De shell heeft voor zulke verzamelingen filenamen een afkorting d.m.v. z.g. *wildcards* (jokers). Bijvoorbeeld alle filenamen die eindigen op ".c" kan opgegeven worden als `*.c` en alle filenamen die met "test" beginnen als `test*` enz. We noemen dit soort dingen "filenaam patronen". Hieruit blijkt dus dat een `*` speciaal behandeld wordt door de shell. We hebben al meer tekens gezien die door de shell speciaal behandeld worden, zoals `< > |&`. In hoofdstuk 4 zullen we nog meer van deze tekens zien. Dit is de reden dat we afraden om zulke tekens in een filenaam te gebruiken. We noemen deze tekens "*metacharacters*". Wanneer we toch deze tekens in een filenaam moeten gebruiken dan kunnen we de filenaam tussen aanhalingstekens (`''`) zetten. De aanhalingstekens zijn dus ook metacharacters. Hoe we die weer in een filenaam kunnen krijgen dat gaat nu te ver.

De wildcard tekens hebben de volgende betekenis:

`*` Elk rijtje tekens mag i.p.v. het `*` ingevuld worden, inclusief helemaal niets. Uitzondering: als het `*` aan het begin staat of direct na een "/" dan mag géén "." ingevuld worden als eerste teken.

`?` Er mag één teken ingevuld worden i.p.v. het `?` teken. Uitzondering: aan het begin of na een "/" mag geen "." ingevuld worden.

`[abc]` Er mag één teken ingevuld worden uit het rijtje dat tussen haakjes staat. Er mag ook een interval opgegeven worden zoals `[a-z]` (kleine letters) of `[0-9]` (cijfers).

De shell werkt a.v. wanneer een patroon met wildcards gegeven wordt: Alle filenamen (normaal uit de werkdirectory) worden bekeken en er wordt gekeken of deze filenaam gemaakt kan worden door de wildcards in het patroon te vervangen zoals boven aangegeven. Alle bestaande filenamen die op deze manier gevormd kunnen worden worden daarna op alfabetische volgorde achter elkaar gezet.

Als er directory onderdelen in het filenaam patroon aanwezig zijn dan tellen die gewoon mee. Als het een absolute padnaam betreft (die dus met "/" begint) dan worden de filenamen in dat pad bekeken, als het een relatief padnaam betreft dan worden de filenamen t.o.v. de werkdirectory bekeken. In de onderdelen mogen ook weer wildcards voorkomen. Wildcards zullen echter nooit door een "/" vervangen worden. M.a.w. elke wildcard blijft in één directory component.

De voorbeelden in tabel 3.5 kunnen dit verduidelijken:

Het rijtje filenamen dat uit een wildcard patroon komt (de "expansie") wordt door de shell gezet in de plaats van het patroon en aan het betreffende commando doorgegeven. Dit kan dus alleen als het commando inderdaad meerdere filename accepteert (tenzij er toevallig precies één filenaam uitkomt). Het commando kan niet meer nagaan of het rijtje filenamen uit een patroon afgeleid is of dat deze een voor een ingetypt zijn.

patroon	betekenis
*[0-9]**	Alle filenamen in de werkdirectory die met een cijfer beginnen
/users/piet/.c*	alle files in de directory "/users/piet" die eindigen op ".c" en niet met "." beginnen
/*[A-Z]	De files in alle subdirectories van de werkdirectory (maar *niet* in diepere) waar een hoofdletter in de filenaam zit maar die niet met een "." beginnen
*hulp*vb*	alle filenamen die met "hulp" beginnen en op "vb" eindigen, inclusief "hulpvb" en "hulp.vb", maar niet "hulp/vb".

Tabel 3.5: Voorbeelden van wildcards

Opgave 3.3 Wat doet het commando `ln -s *.c mijndir` als "mijndir" een subdirectory is?

Opgave 3.4 Hoe krijg je een lijst van alle files die op een cijfer eindigen (inclusief die met een "." beginnen)?

3.7 Copiëren, verplaatsen en verwijderen

Zoals als in 1.3.2 summier besproken is hebben we de commando's `cp`, `mv` en `rm` voor resp. copiëren, verplaatsen en verwijderen van files.

Het simpelste gebruik can `cp` is

```
$ cp file1 file2
```

Er wordt een nieuwe copie gemaakt onder de naam `file2`. De oude file `file1` blijft onveranderd. Als `file2` als bestaat dan moet er write-permissie op staan, anders wordt een foutmelding gegeven. De oude inhoud van `file2` gaat verloren, ook wanneer er een link naar bestond (de link wijst naar de nieuwe inhoud). Het is mogelijk om `cp` de `file2` te laten weggooien voor de copiëer operatie (wanneer de permissie geen schrijven toelaat) met de `-f` optie. Er moet dan wel toestemming zijn om de file weg te gooien (i.h.a. betekent dat write-permissie op de directory). De permissies van de originele file kunnen in de copie overgenomen worden met de `-p` optie, als deze niet gegeven wordt dan krijgt de nieuwe file standaard-permissies.

Het is mogelijk een hele verzameling files te copieren naar eenzelfde directory met het commando

```
$ cp file1 file2 ... dir
```

Alle opgegeven files worden dan naar `dir` gecopiëerd met hun originele filenaam (zonder een eventuele directory die meegegeven wordt). Dus

```
$ cp a/b c/d x
```

waarbij `x` een directory moet zijn creert de files `x/b` en `x/d`, en niet `x/a/b` en `x/c/d`! Je kunt dit natuurlijk ook met wildcards doen.

Het is ook mogelijk hele directory bomen te copiëren:

```
$ cp -r dir1 dir2
```

copiëert (recursief) de hele boom onder *dir1* naar *dir2*. Als de directory *dir2* al bestaat dan komt de copie als *dir2/dir1* te voorschijn, waarbij alleen het laatste deel van *dir1* gebruikt wordt. Als *dir2* niet bestaat, dan wordt *dir2* de copie. Dus als *xx/yy* nog niet bestaat en de volgende commando's worden gegeven:

```
$ cp a/b xx/yy
$ cp c/d xx/yy
```

Dan worden de volgende directory bomen gemaakt: `xx/yy` met een copie van de files en directories onder *a/b*, en `xx/yy/d` (omdat xx/yy dan wel bestaat) met een copie van de files en directories onder c/d. Even goed opletten dus!

Tenslotte is er nog de *-i* optie die tot gevolg heeft dat voor elke file gevraagd wordt of je die wel of niet wilt copiëren.

Het *mv* commando wordt gebruikt om een file een andere naam te geven of te verplaatsen naar een andere directory. Het lijkt in dat opzicht op het *cp* commando waarbij het origineel verdwijnt. Het systeem zal zoveel mogelijk trachten niet een echte copie te maken, maar alleen de naam te veranderen. In feite lukt dit als een harde link van de oude naar de nieuwe plaats ook mogelijk is. Anders wordt een echte copie gemaakt en de oude file verwijderd. De opties *-i* en *-f* werken als bij het *cp* commando.

Opgave 3.5 Waarom is er geen *mv -r*?

Om files te verwijderen gebruiken we het *rm* commando, waarbij één of meer files opgegeven kunnen worden. Directories kunnen alleen opgegeven worden als ook de *-r* optie opgegeven wordt. De *-i* en *-f* opties werken als bij *cp* en *mv*. Bovendien klaagt *rm -f* niet als een file niet aanwezig is.

Omdat het command *rm ** nogal desastreus kan zijn, moet je opletten dat je dit niet per ongeluk intikt. Als je bijv. wilt intypen: *rm *.o* maar je typt per ongeluk een spatie achter het * dan ben je een hoop files kwijt. Er is een simpele truc om je tegen dit soort ongelukken te beschermen: zet in de belangrijke directories een file met naam `-i`. Als je nu *rm ** geeft dan komt er na expansie van de wildcards: *rm -i a b c ...* en wordt er dus om toestemming gevraagd!! De inhoud van deze file is niet belangrijk maar het kan handig zijn om erin te zetten waarvoor deze file dient. Om de file uiteindelijk weer weg te gooien helpt het commando *rm -i* natuurlijk niet, zelfs niet met quotes, omdat deze het *rm* commando nooit bereiken. Het commando *rm ./-i* werkt echter wel.

-f	cp, mv, ln, rm	"Force" de operatie, ongeacht de permissies
-i	cp, mv, ln, rm	Interactief – vraag toestemming
-p	cp	Neem permissies over
-r	cp, rm	Doe de operatie recursief over directory bomen
-s	ln	Maak een symbolische link

Tabel 3.6: Opties voor *cp*, *mv*, *ln* en *rm*

3.8 Overzicht

ls	geef info over files / directories
chmod	wijzig mode (permissies)
chown	wijzig eigenaar (user)
chgrp	wijzig groep
cp	copieer file(s)
mv	hernoem of verplaats file(s)
rm	verwijder file(s) of links
cat	concateneer file(s)
mkdir	maak een nieuwe directory
rmdir	verwijder directory
ln	maak een link
wc	tel aantal woorden, characters en regels

Tabel 3.7: Overzicht file manipulatie commando's

Hoofdstuk 4

Shells

U ziet het goed: er staat *shells* in het meervoud. Er bestaan in het Unix systeem meerdere shells. We hebben het al gehad over het verschil tussen grafische (visuele) shells, waar we met een muis op icoontjes kunnen klikken of iconen op andere laten vallen, en tekstuele shells waar je met het toetsenbord commando's in moet tikken.

Grafische shells hebben het voordeel dat het makkelijk werkt en dat je minder hoeft te onthouden ("hoe heette dat commando ook al weer …?") omdat je de dingen die voorhanden zijn voor je ziet. Aan de andere kant ben je dan vaak ook beperkt tot de dingen die je met muiskliks kunt aangeven, anders moet je toch weer terugvallen op het intikken van dingen.

Een belangrijk voordeel dat de tekstuele shells in Unix hebben is dat ze *programmeerbaar* zijn, d.w.z. je kunt er een soort van programmaatjes mee maken. Dat betekent dat het aantal mogelijkheden in principe oneindig groot wordt, iets wat bij grafische shells in het algemeen moeilijk te realiseren is. Je ziet dus dat er voor beide soorten een plaats is. In dit hoofdstuk houden we ons alleen bezig met de tekstuele shells en dan vooral met de programmeeraspecten ervan.

De originele shell van Unix was de Bourne shell (genoemd naar degene die het geprogrammeerd heeft). Op de Universiteit van Berkeley heeft men op een gegeven moment een nieuwe shell ontwikkeld. De programmeeraspecten van deze shell lijken heel veel op de programmeertaal C, daarom wordt deze de C shell (csh) genoemd. Behalve dat de taal op C lijkt verschilt deze ook van de Bourne shell doordat er een mechanisme inzat om vorige commando's te herhalen (of te modificeren) zonder ze helemaal opnieuw te hoeven intikken (het z.g. *history* mechanisme). De csh bezit echter een aantal tekortkomingen en bovendien kun je niet zonder meer de programmaatjes van de csh uitwisselen met de Bourne shell.

De Korn shell (geschreven door Korn) is een uitbreiding van de Bourne shell met o.a. ook een history mechanisme. Het voordeel van de Korn shell (ksh) is dat Bourne shell programmaatjes ongewijzigd in de Korn shell gebruikt kunnen worden.

In dit hoofdstuk zullen we ons voornamelijk bezig houden met de Bourne shell met een paar korte uitstapjes naar de Korn shell en de C-shell.

4.1 De Bourne shell

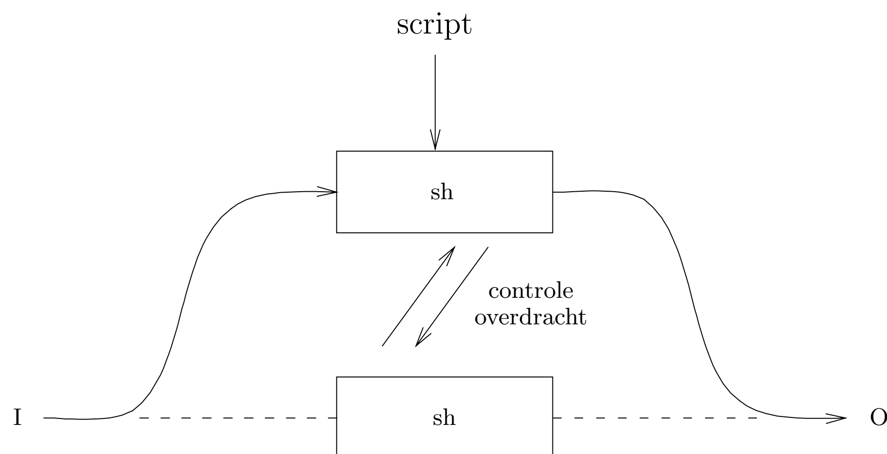
Tot nu toe hebben we de shell gebruikt om opdrachten via het toetsenbord in te geven. We kunnen echter ook de uit te voeren opdrachten in een file zetten en dan de shell verzoeken om deze opdrachten uit te voeren. Zo'n file wordt een shell script genoemd. De soort opdrachten die de shell kan uitvoeren is in beide gevallen hetzelfde.

```
$ cat > script
ls -l hulp
^D
$ ls -l hulp
-rw-r--r--    1 piet     staff          15 Jan 18 10:58 tekst
$ sh script
-rw-r--r--    1 piet     staff          15 Jan 18 10:58 tekst
```

Door het commando *sh script* aan te roepen geven we de shell opdracht om de opdrachten in de file "script" uit te voeren. Er wordt dan een *nieuw* proces opgestart met de shell (sh) als programma. We zeggen wel dat er een *subshell* opgestart wordt. Deze subshell leest de opdrachten uit "script" één voor één en voert ze uit, net als wanneer ze ingetikt zouden zijn.

In figuur 4.1 zien we het procesdiagram en in figuur 4.2 het tijdsdiagram. Let erop dat de standaard invoer van de tweede shell hetzelfde is als die van de eerste (dus *niet* het script is). Het is zelfs mogelijk om een nieuw commando van het script te maken door de "x" permissie

Figuur 4.1 Executie van een shell script

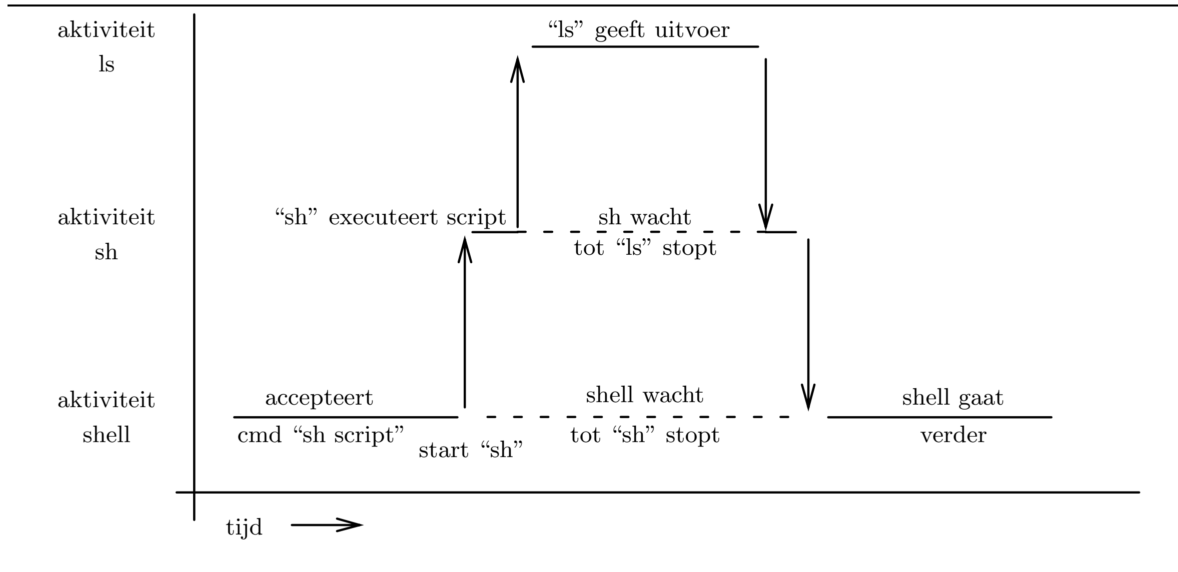


voor script aan te zetten met *chmod +x script* en dan gewoon het commando *script* te geven. In dat geval is het handig om de filenaam van de te gebruiken shell in de eerste regel te zetten a.v.:

```
#! /bin/sh
```

(/bin/sh is de file waar de *executable* van de Bourne shell staat).

Figuur 4.2 Tijdsdiagram voor figuur 4.1



In bovenstaand voorbeeld hadden we maar één commando in het shell script maar het mogen er ook meer zijn. We kunnen elk commando op een nieuwe regel beginnen, maar we kunnen commando's ook gescheiden door puntkomma's (;) op één regel zetten. Dit is weer een manier om samengestelde commando's te maken (we hebben al eerder de pipeline als een andere manier gezien). We zullen verder in dit hoofdstuk nog meer manieren zien.

4.1.1 Shell uitvoer

Soms is het nodig dat de shell iets afdrukt, vooral bij het gebruik van shell scripts kan dat nuttig zijn. Hiervoor bestaat het `echo` commando.

```
$ echo aap noot mies
```

drukt de tekst "aap noot mies" af op de standaard uitvoer van de shell, gevolgd door een "newline" teken. Wat het echo commando doet is al zijn argument afdrukken, gescheiden door een spatie en gevolgd door een newline. Wanneer de argumenten door meer dan één spatie gescheiden worden dan wordt er toch maar één afgedrukt (de andere worden door de shell opgegeten). Je moet natuurlijk uitkijken met het afdrukken van *metacharacters* zoals `* < >`, want deze worden door de shell onderschept. Als je speciale tekens wilt afdrukken of als het aantal spaties belangrijk is, dan kun je de tekst tussen aanhalingstekens (' ') zetten, dus bijvoorbeeld

```
$ echo '*** Let op:      dit is <speciaal> ***'
```

Het echo commando kan wel gebruikt worden in samenhang met de al besproken metacharacters zoals:

```
$ echo de test files zijn: test*
de test files zijn: test1 test2 test3
$ echo tekst > boodschap
```
schrijft de tekst naar de file “boodschap”.
```
$ echo Fout geconstateerd >&2
```
schrijft de foutmelding naar de standaard error van de shell.

4.1.2 Argumenten

Het is mogelijk aan een shell script argumenten mee te geven net als aan gewone commando’s, maar het shell script moet wel iets speciaals doen om deze te verwerken.

```
$ cat > lla
ls -la
^D
$ chmod +x lla
$ lla
```
lijst van files volgt ...
```
$ lla hulp
```
Dezelfde lijst van files volgt, niet die van “hulp”

Om de argumenten te kunnen gebruiken heeft de shell de volgende constructies:

- `$*` of `$@` wordt vervangen door alle meegeven argumenten. In bovenstaand voorbeeld zou het shell script dus beter `ls -la $*` kunnen bevatten
- `$0` wordt vervangen door de naam van het shell script (dit kan handig zijn o.a. voor gebruik in foutmeldingen, zodat de gebruiker weet uit welk shell script de foutmelding komt
- `$1` ... `$9` worden resp. vervangen door het eerste ... het negende argument voor zover aanwezig.
- `$#` wordt vervangen door het *aantal* meegegeven argumenten.

Als er meer dan 9 argumenten zijn en we willen die individueel behandelen dan moeten we een speciale techniek gebruiken (zie de “shift” opdracht in 4.1.12 verderop).

4.1.3 Variabelen

We kunnen in de shell ook eigen variabelen gebruiken, zoals in de meeste programmeertalen:

```
$ var=waarde
$ echo $var
waarde
```

Een variabele krijgt een waarde door het shell commando ⟨naam⟩=⟨waarde⟩. Er mogen geen spaties om het = teken staan en waarde mag geen speciale tekens bevatten (ook geen spaties).

Moeten er toch speciale tekens of spaties in de variabele gezet worden dan moet de waarde tussen aanhalingstekens gezet worden, zoals in

```
$ mijnvar='*Hello World*'
$ echo $mijnvar
*Hello World*
```

De waarde van een variabele is altijd tekst. Ook al gebruiken we een getal dan wordt dit getal in tekstvorm opgeslagen. Dus de waarde "007" is iets anders dan "7".

Wat we ook in bovenstaand voorbeeld gezien hebben is dat bij het *gebruik* van een variabele we de naam van de variabele moeten laten voorafgaan door het `$` teken. De `$` is natuurlijk ook weer een metacharacter.

Een variabele mag ook gebruikt worden midden in een stuk tekst en wordt dan a.h.w. aan de omringende tekst vastgeplakt, bijv. `+$mijnvar+` levert met bovenstaand voorbeeld op: `+*Hello World*+`. Dit geeft natuurlijk een probleem als de tekst achter de variabele begint met een letter of cijfer omdat die dan als onderdeel van de naam van de variabele genomen worden. Daarom mag de variabele ook zó gebruikt worden: `${`⟨naam⟩`}`.

Op deze laatste constructie bestaan een aantal varianten die handig zijn om te testen of een variabele wel of niet een waarde heeft.

- `${`⟨naam⟩`-`⟨woord⟩`}` Gebruik de waarde van de variabele als die bestaat, anders wordt de waarde van ⟨woord⟩ gebruikt. De waarde van de variabele verandert niet.
- `${`⟨naam⟩`=`⟨woord⟩`}` Net zo maar nu krijgt de variabele de waarde van ⟨woord⟩ als hij nog geen waarde had.
- `${`⟨naam⟩`+`⟨woord⟩`}` Dit is precies andersom: Als de variabele wèl een waarde heeft dan wordt ⟨woord⟩ gebruikt, anders wordt een lege tekst ingevoegd. De variabele verandert niet.
- `${`⟨naam⟩`?`⟨woord⟩`}` Als de variabele een waarde heeft, dan wordt deze gebruikt, anders wordt er een foutmelding gegeven met ⟨woord⟩ als tekst en de shell stopt.

In bovenstaande voorbeelden geldt dat een variabele een waarde heeft als er een ⟨naam⟩`=`⟨waarde⟩ gegeven is of als de variabele geërfd wordt (zie de volgende sectie). Het maakt niet uit of de variabele een lege waarde heeft (bijv. door ⟨naam⟩`=""`) of niet. Als je een lege waarde ook als "ongedefinieerd" wilt laten gelden dan kun je een `:` achter de naam zetten, dus bijv. `${`⟨naam⟩`:-`⟨woord⟩`}`.

In alle bovengenoemde gevallen kun je ⟨woord⟩ tussen aanhalingstekens zetten om meer dan één woord op te nemen.

In tabel 4.1 vind je nog een paar speciale variabelen die de shell gebruikt.

4.1.4 Environment variabelen

Variabelen worden niet zonder meer doorgegeven aan subshells wanneer een shell script uitgevoerd wordt. Elke shell begint met een nieuwe collectie. Soms is het toch handig om bepaalde

$$	Het proces nummer van de shell zelf
$?	De exit status code van het laatst uitgevoerde voorgrond commando (zie 4.1.6)
$!	Het proces nummer van het laatst opgestarte achtergrond commando
PS1	De "prompt" die de shell afdrukt om te laten zien dat je een commando in kunt tikken (meestal `$ `)
PS2	De prompt die de shell gebruikt om aan te geven dat er een vervolg van je commando verwacht wordt (als je commando niet op één regel past) (meestal `>`).
IFS	Input Field Separators: tekens die het eind van een ⟨woord⟩ (argument) aangeven. Moeilijk!

Tabel 4.1: Speciale shell variabelen

variabelen wel door te kunnen geven. Dit kan met *environment variabelen.* Een variabele die doorgegeven moet worden geven we aan met het *export* commando. In bovenstaand voorbeeld:

```
$ export mijnvar
```

veroorzaakt dat "mijnvar" ook in subshells beschikbaar is. Dit geldt zelfs niet alleen voor shells maar ook andere programma's die door de shell opgestart worden kunnen de variabele "mijnvar" raadplegen[1].

De variabelen die een programma op die manier "erft" vormen de z.g. *environment.* Elke shell begint meestal met een environment, die een aantal standaard variabelen bevat. Zie tabel 4.2 voor een paar voorbeelden. De environment kan afgedrukt worden met het commando *env* of *printenv*. Environment variabelen kunnen veranderd worden door de shell (met *var=waarde*), maar om de gewijzigde waarde naar op te starten programma's door te geven moet expliciet het *export* command gegeven worden. **Het is niet mogelijk om vanuit een subshell de environment van de hoofdshell te veranderen!**

HOME	De "home" directory van de gebruiker, d.w.z. de werkdirectory direct na het inloggen. Dit is normaal de hoofddirectory van de begruiker, bijvoorbeeld "/users/piet".
PATH	De lijst van directories waar de shell commando's zoekt. De directories worden gescheiden door `:`, bijv. "`/bin:/usr/bin:.`".

Tabel 4.2: Enkele standaard environment variabelen

De environment voor één opdracht veranderen kan simpeler door de waardetoekenningen vóór de opdracht te zetten:

[1]In C programma's kan dat met de functie *getenv*.

```
$ AA=125 BB=tekst commando
```

voert het commando uit met de variabelen AA en BB toegevoegd aan de environment. Het export commando moet in dit geval *niet* gegeven worden en na de uitvoering van deze opdracht is de environment van de shell niet gewijzigd.De variabelen AA en BB hebben hun vorige waarde behouden (of zijn er zelfs niet meer als ze niet bestonden).

Het is een goede gewoonte om voor environment variabelen namen met HOOFDLETTERS te gebruiken maar dit is niet verplicht.

4.1.5 Quoting

We hebben al een paar keer gezien dat sommige tekens tegen de shell beschermd moeten worden wanneer ze in een filenaam of een ander argument voorkomen, de z.g. metacharacters. We gaan nu wat gedetailleerder naar dit punt kijken. De tekens die problemen kunnen geven zijn:

```
;  &   (  )  |  ^  <  >  newline  spatie  tab
$ ‘ ’ " \ * ? [ { } #
```

Een aantal hebben we al gezien. De rest zal in dit hoofdstuk besproken worden. Een overzicht van de betekenis van alle metacharacters kunt u aan het eind van dit hoofdstuk vinden in tabel 4.5.

De tekens in de eerste regel in het overzicht hierboven hebben als extra eigenschap dat ze scheiding aanbrengen tussen commando's en argumenten, dus:

```
$ ls>lijst is hetzelfde als:
$ ls > lijst
```

Een newline beëindigt een commando, behalve als dat onmogelijk is. Bijvoorbeeld als een regel eindigt op een `|`, dan moet er nog meer volgen en dan wordt de volgende regel erbij genomen. Als u een commando hebt dat zo groot is dat het niet op één regel past dan kan dit over meer regels verdeeld worden door ieder regel behalve de laatste te laten eindigen met een `\` (backslash) teken.

ls een van bovengenoemde metacharacters voorkomt in een argument dat aan een commando meegegeven moet worden (of in de commandonaam of bij een toekenning aan een variabele) dan hebben we al gezien dat we `' '` kunnen gebruiken. Tussen de `' '` worden alle tekens letterlijk overgenomen. Het enige teken dat er niet tussen kan staan is de `'` zelf. Wanneer de `'...'` constructie tegen iets anders aanstaat (zonder één van bovengenoemde scheiders) dan vormt de ... tekst (na weghalen van de `' '`) één geheel ermee.

Een tweede manier om metacharacters te beschermen van de shell is om er een `\` voor te zetten (behalve dus voor de newline). `\` zegt dat het volgende teken als "gewoon" teken opgevat moet worden. Bijvoorbeeld `\\` kan gebruikt worden om één `\` op te nemen. Zoals uit de vorige alinea blijkt werkt dit niet tussen `'...'`. Dit is dus dé manier om een `'` op te nemen.

De derde manier om metacharacters te beschermen is om ze tussen `"..."` te zetten. Dit is bijna hetzelfde als `'...'`, alleen worden tussen `" "` de tekens `$`, `\` en `‘` (die we nog zullen

tegenkomen) wèl als speciale tekens beschouwd. Hier nog een ingewikkeld voorbeeld met combinaties van quotes:

```
$ echo "'<"'$$'\>\'
'<$$>'
```

Eerst gebruiken we `""` om de `'<` te beschermen (we kunnen niet `'` gebruiken om `'` te beschermen), dan gebruiken we `''` om de `$`'s te beschermen (dit kan niet met de `""`), tenslotte als afwisseling `\` om de `'>` te beschermen. We hadden natuurlijk ook vóór ieder teken een `\` kunnen zetten.

4.1.6 Exit status

Elk proces dat eindigt geeft een code af (exit status, ook wel return value of status code genoemd) die door het aanroepende proces geraadpleegd kan worden. De bedoeling is dat aan deze code gezien kan worden of het programma zonder fouten beëindigd is. Elk goed geschreven programma zal dit doen (maar helaas zijn er af en toe programma's die het niet goed doen).

In de shell kan dit gebruikt worden om acties te ondernemen afhankelijk van het succes van een commando. De code 0 wordt altijd gebruikt voor succes, en andere getallen voor fouten. In de "man" pagina's van een commando worden meestal de verschillende waarden besproken. Bijvoorbeeld *ls* zal een niet-0 exit status geven als niet alle gevraagde files benaderd konden worden omdat er een niet bestond of omdat er een probleem met de permissies was.

De exit status van het laatst uitgevoerde voorgrond commando kan gevonden worden in de variabele *$?*:

```
$ ls foutnaam
Cannot access foutnaam: No such file or directory
$ echo $?
2
```

Een shell kan zelf stoppen met een exit status door het *exit* ⟨code⟩ commando. Wanneer geen code meegegeven wordt of de shell stopt door aan het einde van een script te komen dan wordt exit status 0 gegeven.

4.1.7 Samengestelde commando's

We hebben al een manier gezien om commando's samen te stellen tot ingewikkeldere, nl met de pipe (`|`). We gaan nu de andere mogelijkheden bekijken.

Een *simpel commando* is gewoon een commandonaam (programmanaam) gevolgd door argumenten. Simpele commando's kunnen via het `|` symbool aan elkaar geregen worden tot een *pipeline*. (I.p.v. simpele commando's mogen in een pipeline ook de in 4.1.8 genoemde besturingsstructuren gebruikt worden als onderdelen van een pipeline.) Een pipeline wordt altijd als één *job* beschouwd. De exit status van het laatste commando geldt als exit status voor de hele pipeline.

Pipelines (of losse commando's) kunnen weer aan elkaar geregen worden tot een *lijst*. Dit gebeurt door één van de volgende symbolen tussen de onderdelen te zetten:

```
; & && ||
```

De eerste twee mogen ook nog als afsluiting gebruikt worden.

De betekenis van deze symbolen is:

`;` De onderdelen worden één voor één uitgevoerd. (De shell wacht bij een `;` tot het linker commando klaar is en gaat dan verder met het rechter.) De `;` mag ook vervangen worden door newlines.

`&` De shell start het linker onderdeel op in de achtergrond en gaat dan meteen verder. Dit is dus een uitbreiding van wat we in hoofdstuk 2 al gezien hebben.

`&&` De shell start het linker onderdeel op en wacht tot het klaar is. Na afloop wordt naar de status code gekeken, als deze 0 is (=succes) dan wordt het rechter deel ook uitgevoerd, anders wordt het rechterdeel overgeslagen. Bij `a && b && c ...` wordt na ieder commando gekeken of de rest nog wel uitgevoerd moet worden. Zodra er één commando met niet-succes terug komt stopt de hele rij. Dit is handig wanneer je een serie moet uitvoeren waar het volgende commando niet verder kan als het vorige mislukt is.

`||` Idem maar nu wordt het vervolg pas uitgevoerd als het linker commando *mislukt* is. Als het linker commando success oplevert (status code 0) dan stopt deze lijst. Dit is handig als het rechter commando dingen recht moet zetten als het linker commando fout gaat.

Wanneer `&&` en `||` gemengd worden met `;` en `&` dan gaat de werking van een `&&` of `||` niet verder dan de eerstvolgende `;` of `&`.

```
$ prog1 && echo first ; echo last
```
als "prog1" succes heeft wordt "first" afgedrukt. Altijd wordt "last" afgedrukt.

De exit status van in de achtergrond (*asynchroon*) uitgevoerde commando's wordt nooit meegerekend voor de exit status van de lijst. Het laatste in de voorgrond uitgevoerde commando bepaalt wat de exit status van de lijst is.

4.1.8 Programmeerstructuren

De structuren in deze sectie mogen ook in een pipeline gebruikt worden net als een simpel commando.

Commando's kunnen als een eenheid gebruikt worden door ze tussen haakjes te zetten. Hiervoor kunnen zowel accolades { } als ronde haakjes ( ) gebruikt worden. Tussen de haakjes mag een *lijst* staan. Het verschil tussen deze constructies is dat de (⟨lijst⟩) constructie in een *subshell* wordt uitgevoerd en de { ⟨lijst⟩ } niet. Bovendien moet er na de { altijd een spatie staan. De constructie met ( ) wordt het meest gebruikt.

Echte programmeringsstructuren zoals *if* en *while* zijn er ook. De notatie is alleen wat anders dan in de meeste programmeertalen gebruikelijk is.

Let op: in deze constructies komen keywords voor (if then else elif fi case esac for while until do done { }). Deze kunnen alleen aan het begin van een regel of na een *;* of & gegeven worden anders worden ze als argument van het vorige commando beschouwd!

If then else

```
if ⟨lijst⟩
then ⟨lijst⟩
else ⟨lijst⟩
fi
```

De ⟨lijst⟩ achter *if* wordt uitgevoerd. Als deze succes oplevert (exit status 0) dan wordt het gedeelte achter *then* uitgevoerd, anders het gedeelte achter *else*. Het *else* mag weggelaten worden en *else* mag ook gecombineerd worden met de volgende *if* tot *elif* (in dat geval wordt er geen extra *fi* toegevoegd:

```
if p1
then eerste keus
elif p2
then tweede keus
else laatste keus
fi
```

De volgende twee constructies doen hetzelfde:

```
$ p1 && p2
$ if p1; then p2 fi
```

Opgave 4.1 Waarom staat er in dit voorbeeld niet: "*if p1 then p2 fi*"?

While do

```
while ⟨lijst⟩
do ⟨lijst⟩
done
```

Voer de ⟨lijst⟩ achter de *while* uit, en als deze mislukt (exit status niet 0) dan stopt de *while*. Als de eerste ⟨lijst⟩ succes oplevert (exit status 0) dan wordt de tweede ⟨lijst⟩ uitgevoerd en de lus weer opnieuw uitgevoerd.

In plaats van *while* mag ook *until* gebruikt worden. De lus wordt dan uitgevoerd zolang het resultaat van de eerste ⟨lijst⟩ niet 0 is (en stopt dus zodra de exit status ervan 0 is).

For do

```
for ⟨naam⟩ [ in ⟨word⟩ ...]
do ⟨lijst⟩
done
```

Dit is de manier om één of meer commando's uit te laten voeren voor een lijst van parameters. ⟨naam⟩ is een variabele die achtereenvolgens de waarden achter *in* krijgt, en met deze waarde wordt dan de ⟨lijst⟩ uitgevoerd

```
$ for naam in aap noot mies
  do echo "*** $naam ***"
  done
*** aap ***
*** noot ***
*** mies ***
```

Het *in* gedeelte mag weggelaten worden en betekent dan hetzelfde als *in $**, dus alle argumenten van het shell script worden afgewerkt.

Case in

```
case ⟨word⟩ in ⟨patroon⟩) ⟨lijst⟩ ;; ...
esac
```

Dit is wel de meest ingewikkelde besturingsstructuur van de shell maar tegelijkertijd misschien ook wel de krachtigste.

De constructie *in* ⟨patroon⟩*)* ⟨lijst⟩ *;;* mag meerdere keren gegeven worden. Het ⟨patroon⟩ is net als een filenaam-patroon zoals we in 3.6 besproken hebben of een rijtje filenaam-patronen gescheiden door een | symbool. De filenaam-patronen worden hier echter niet gebruikt om filenamen te "matchen" maar om het ⟨woord⟩ achter *case* te matchen. Een verschil met de filenaam matching is verder nog dat hier de tekens `. /` gewoon meedoen net als alle andere tekens. Als er een match is dan worden de commando's van de bijbehorende ⟨lijst⟩ uitgevoerd.

We kunnen het beste met wat simpele voorbeelden beginnen.

```
case $1 in -* | +*) echo "option found";; esac
```

De shell "weet" dat er achter *in* patronen komen, deze moeten dus *niet* gequote worden. In dit voorbeeld testen we of het eerste argument met een `-` of `+` teken begint (wat vaak als een optie beschouwd wordt).

```
case $# in 1) arg=$1 ;;
           *) echo "geef een argument mee">&2 ;;
esac
```

Als het aantal argumenten (*$#*) 1 is dan bewaren we het argument in de variabele *arg*, anders geven we een foutmelding.

De `;;` achter het laatste geval mag weggelaten worden, maar het is verstandig om het er wel neer te zetten voor het geval je later een extra geval erachter gaat zetten.

4.1.9 Commando substitutie

Commando substitutie is een manier om de uitvoer van een commando in de tekst van een andere opdracht te gebruiken. Stel bijvoorbeeld dat we het aantal regels dat een file "hulp" heeft in een variabele "aantal" willen zetten. Dat kunnen we als volgt doen:

```
$ aantal=`wc -l < hulp`
```

(We gebruiken de constructie `< hulp` om de filenaam kwijt te raken.) De shell neem het commando tussen `` ` ` `` (de z.g. *backquotes*) en vervangt dit stuk door de standaard uitvoer van dit commando. Hierbij worden newlines vervangen door spaties en een eventuele afsluitende newline wordt weggegooid. Binnen de backquotes kunnen variabelen gebruikt worden. De backslash `\` kan gebruikt worden om sommige tekens onschadelijk te maken: in het bijzonder `\` zelf, de `` ` `` (zodat je ze kunt nesten) en `"` wanneer het hele commando weer binnen een `"..."` constructie zit.

Het commando tussen `` ` ` `` kan in feite een ⟨lijst⟩ zijn.

4.1.10 Test en expr

De shell heeft op zich geen taalconstructies voor expressies (bijv. rekenwerk). Hiervoor moeten externe programma's gebruikt worden. Dit is één van de voorbeelden van de Unix filosofie: laat een programma één ding goed doen, liever dan veel dingen half, en doe ingewikkelde dingen door programma's samen te laten werken. We zullen nu zien hoe we dit kunnen gebruiken voor *tests* die we met `if` en `while` kunnen gebruiken en voor rekenwerk, bijvoorbeeld voor het gebruik bij variabelen.

Test

Het "test" programma wordt gebruikt om allerlei tests uit te laten voeren voor de shell. We hebben bij de bespreking van `if` en `while` gezien dat de conditie (true/false) bepaald wordt door de exit status. Het zal daarom niet verwonderen dat het "test" programma geen uitvoer produceert, maar alleen een status code aflevert waarbij 0=true en alle andere waarden=false.

Wat "test" moet testen wordt aangegeven door de argumenten die het meekrijgt. De argumenten vormen samen een soort expressie, waarbij elke operand en elke operator als apart argument meegegeven moet worden. Dus niet:

```
$ test 2=3
```
maar
```
$ test 2 = 3
```

Verder moet je er natuurlijk op letten dat metacharacters (bijvoorbeeld haakjes) gequote moeten worden. Variabelen worden natuurlijk gewoon gesubstitueerd maar een variabele met een lege waarde verdwijnt tenzij die tussen `""` gezet wordt.

De belangrijkste tests die uitgevoerd kunnen worden zijn (de s_i zijn strings, de n_i zijn argumenten die als getallen beschouwd worden) vind u in tabel 4.3: Er zijn nog wat minder

-z s_1	true als s_1 lengte 0 heeft (leeg is)
-n s_1	true als s_1 niet lengte 0 heeft
$s_1 = s_2$	true als de strings s_1 en s_2 gelijk zijn
s_1 != s_2	true als de strings s_1 en s_2 ongelijk zijn
s_1	true als s1 niet de lege string is
n_1 -eq n_2	true als n_1 en n_2 gelijk zijn *als getal*. Er zijn ook de tests: -ne, -gt, -ge, -lt, and -le voor resp $\neq, >, \geq, <, \leq$.
!	de *not* operator
-a	de *and* operator
-o	de *or* operator
( )	haakjes, deze moeten gequote worden bijv. als `\(...\)`.
-r *file*	true als de *file* bestaat en gelezen kan worden.
-w *file*	true als de *file* bestaat en geschreven kan worden
-x *file*	true als de *file* bestaat en geexecuteerd kan worden
-f *file*	true als de *file* bestaat en een gewone file is
-d *file*	true als de *file* bestaat en een directory is
-l *file*	true als de *file* bestaat en een symbolic link is
-s *file*	true als de *file* bestaat en minstens 1 character heeft (niet leeg is)

Tabel 4.3: Argumenten van het programma `test`

belangrijke tests, zie hiervoor het "manual".

Voorbeelden:

```
if test -f hulpfile -a -r hulpfile
then ...
```

Test of hulpfile een gewone file is en gelezen kan worden (met alleen `-r` zou het ook nog een directory kunnen zijn!).

```
if test `wc -l < hulpfile` -gt 10
```

Test of er meer dan 10 regels in de hulpfile zitten.

Een alternatieve manier om het programma test aan te roepen is door `[ ...]` i.p.v. `test ...`.

```
$ if [ -f hulpfile -a -r hulpfile ]
```

Expr

Expr is een programma dat kan rekenen. In dit geval wordt het resultaat van de berekening door *expr* op de standaard uitvoer geschreven, waardoor dit meestal tussen backquotes gezet wordt. Expr kan ook enige van de tests uit de vorige sectie uitvoeren zodat je niet altijd combinaties van `expr` en `test` hoeft te gebruiken (je moet dan wel zorgen dat de output van `expr` verdwijnt). Daarom geeft `expr` ook een status code af, en wel:

2	Als er een fout optreedt
1	Als het resultaat van de expressie 0 is of de lege string
0	In alle andere gevallen

Ook bij *expr* moet je letten op metacharacters die gequote moeten worden, zoals `* | & ( )`. Voor *expr* geldt ook dat alle operanden en operators als aparte argument gegeven moeten worden, en bovendien moet je bij string operanden opletten dat die niet als operator opgevat kunnen worden. Tabel 4.4 laat zien welke constructies *expr* kent. Noodzakelijke quotes zijn al aangegeven met een `\`. Vooral bij de string operators moet erg

`+ - * / %`	rekenen met integers. `%` is de rest bij deling. `* / %` gaan voor `+ -`
`= \> \>=` `\< \<= !=`	vergelijkingen. Als beide kanten een getal zijn wordt getalsmatig vergeleken, anders als string.
`\&`	e_1 `\&` e_2 levert 0 op als beide kanten leeg of 0 zijn, anders wordt e_1 opgeleverd.
`|`	e_1 `|` e_2 levert e_1 op als deze niet leeg en niet 0 is, anders wordt e_2 opgeleverd
`:`	e `:` r matcht de waarde e met de reguliere expressie r. Reguliere expressies worden in sectie 5.1.1 behandeld.
`match` e r	idem
`substr` s m n	Pak het gedeelte van de string s vanaf het m^e teken dat n tekens lang is.
`index` s_1 s_2	geeft aan op welke opositie in s_1 de string s_2 voorkomt, of 0 als s_2 niet voorkomt.
`length` s	Geeft de lengte van de string s

Tabel 4.4: Argumenten van het programma *expr*

uitgekeken worden dat een argument niet als operator opgevat kan worden, bijvoorbeeld

```
$ par='+'
$ expr length $par
```

geeft een foutmelding omdat expr als argumenten meekrijgt `length +` en denkt dat er een optelling moet gebeuren (wat in dit geval natuurlijk nergens op slaat). Quotes helpen niet omdat die door de shell afgevangen worden. In dit geval kun je dan maar beter zoiets doen:

```
$ expr length X$par - 1
```

```
n=1
while test $n -le 9
do echo $n
   n=`expr $n + 1` done
```

Drukt de getallen 0 t/m 9 af.

4.1.11 I/O redirection

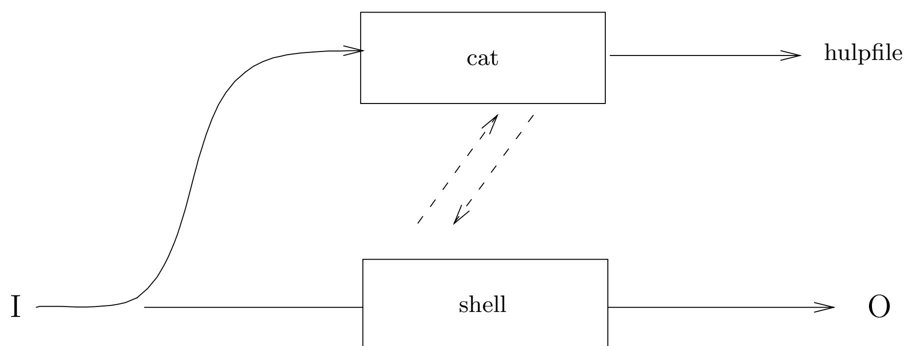
In sectie 2.2 hebben we al wat van de I/O redirection gezien. Hier volgt het complete verhaal. Output redirection kan met > ⟨file⟩ of >> ⟨file⟩. De eerste vorm begint met een schone ⟨file⟩, de tweede vorm schrijft achter de ⟨file⟩ als deze al bestond. Input redirection kan met < ⟨file⟩.

Input redirection op een shell script kan onverwachte effecten hebben. Als voorbeeld nemen we volgende interactieve sessie:

```
$ cat > hulpfile
xyz
abc
^D
```

In dit voorbeeld leest *cat* van de standaard invoer net als de shell, dus van het toetsenbord. Zie het proces schema in figuur 4.3. Het *cat* commando is een handige manier om even wat

Figuur 4.3 Lezen van standaard invoer (interactief)



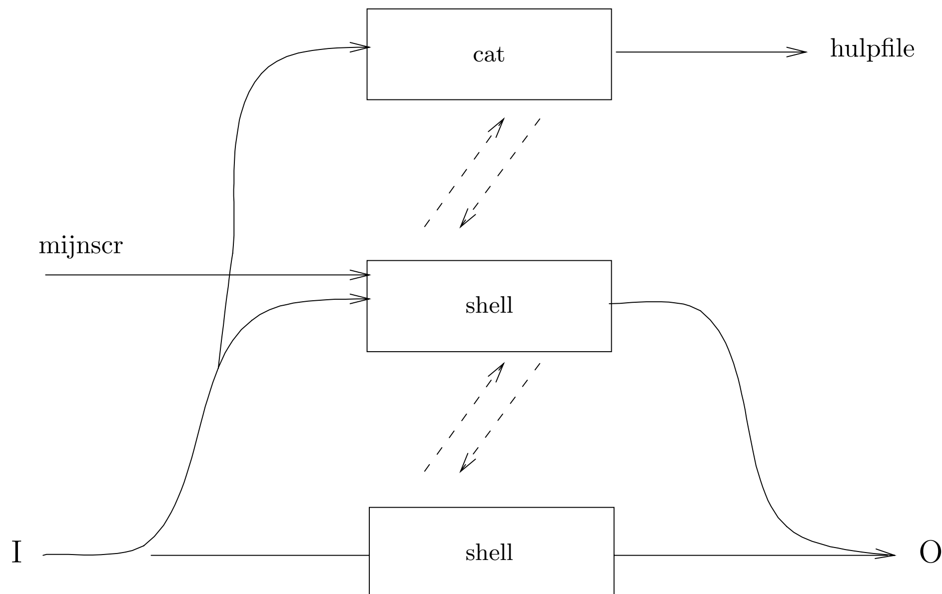
tekst in een file te krijgen.

We gaan nu dit in een shell script gebruiken: Het shell script *mijnscr* bevat (o.a.) het commando:

```
$ cat > hulpfile
xyz
abc
```

Het bijbehorende proces schema staat in figuur 4.4. Zoals je kunt zien wordt het shell script uitgevoerd door een aparte shell. De standaard invoer van deze shell is dezelfde als die van de interactieve shell, dus *niet* het script. Het *cat* commando leest dus niet van het script. Om dit toch mogelijk te maken heeft de shell een aparte constructie, het z.g. *here document*. De vorm hiervan is bijvoorbeeld:

Figuur 4.4 Lezen van standaard invoer (interactief)




```
$ cat > hulpfile <<WOORD
xyz
abc
WOORD
```

Een *here document* kan met of zonder output redirection gebruikt worden, of gevolgd door een pipe. Het is in feite gelijkwaardig aan de constructie `< file`, alleen wordt het commando nu gevolgd door de inhoud van de file. Het *WOORD* dat achter de `<<` staat geeft het einde van de invoer aan, het moet dan als enige op een regel staan, zelfs extra spaties zijn niet toegestaan. Er zijn een paar varianten op deze constructie:

- Als *WOORD* of een deel ervan gequote is (met `' " \`), dan wordt de tekst onveranderd doorgegeven. De afsluitende regel moet echter *niet* de quotes bevatten.
- Als *WOORD niet* gequote is, dan worden er wel een paar wijzigingen op de tekst uitgevoerd: Ten eerste worden variabelen gesubstitueerd, verder worden commando's tussen backquotes uitgewerkt en gesubstitueerd, en `\` kan gebruikt worden om regels aan elkaar te plakken (`\` aan het eind van de regel) en om een `` \ ` $ `` te quoten. Alle andere `\`'en worden onveranderd doorgegeven.
- Als de constructie `<<-WOORD` wordt gebruikt dan verwijdert de shell eventuele TAB tekens aan het begin van iedere regel. Dit gebeurt nadat eventuele substituties zijn uitgevoerd. WOORD zelf mag ook door TAB tekens voorafgegaan worden zowel bij de `<<-` constructie als aan het eind, en deze worden eerst verwijderd voor de vergelijking wordt gedaan.

De afsluitende regel mag weggelaten worden als dit het laatste uit een shell script is.

Tabel 4.5 bevat o.a. een overzicht van de behandelde I/O redirection constructies.

4.1.12 Een voorbeeld

We gaan nu een voorbeeld van een shell script behandelen. We bouwen dit voorbeeld langzaam op.

Het probleem dat we willen oplossen is het *bundelen* van een verzameling files. D.w.z. we willen een stel files bij elkaar pakken in één file op zo'n manier dat ze ook weer gemakkelijk uit te pakken zijn. We gaan ervanuit dat we alleen tekstfiles gebruiken (dus met gewone ASCII tekens erin). We schrijven hiervoor een shellscript *bundel* dat de files die als argumenten meegegeven worden ingepakt op de standaard uitvoer schrijft. De reden dat we standaard uitvoer gebruiken i.p.v. een uitvoer*file* is dat we dan ook het resultaat naar een *mail* programma kunnen *pipen* om het te versturen. Als we het resultaat naar een file willen sturen kan dat met output redirection.

We zouden de files natuurlijk gewoon met *cat* aan elkaar kunnen plakken maar dan is niet meer te zien waar elke file begint. Daarom moeten we iets tussen de files zetten, en dit moet iets zijn dat niet in een van de files voorkomt. De laatste eis is problematisch dus daarom nemen we daarvoor maar een stuk tekst waarvan het onwaarschijnlijk is dat dit is een file voorkomt. Bijvoorbeeld "`!+XYZ-ldsakjfhe:235!`". Om het gemakkelijk te maken dit later te wijzigen stoppen we dit in een shell variabele. Ons eerste shell script wordt dan:

```
scheider='!+XYZ-qwertyuiop:235!'
for f in $*
do
  cat $f
  echo $scheider
done
```

Maar hier zijn twee problemen: Ten eerste staan de originele filenamen niet in de uitvoer, en ten tweede moeten we een apart programma schrijven om de gebundelde uitvoer weer uit elkaar te halen. Het tweede probleem zou opgelost kunnen worden door als uitvoer een *shellscript* te genereren dat elke file als een *here document bevat*. Dan is tegelijkertijd het eerste probleem opgelost. Degene die het resultaat ontvangt hoeft dan alleen maar dit in een file te zeten, bijv. *ontvangst* en dan het commando *$ sh ontvangst* te geven.

De uitvoer van ons bundel script moet er dus zo uitzien (voor één file xyz):

```
#To unbundle, sh this file
# -------------- # xyz # -------------- #
echo 'xyz'
cat > xyz <<'End of file xyz'
⟨Inhoud van file xyz⟩
End of file xyz
```

Als dit shell script uitgevoerd wordt dan wordt van elke file die uitgepakt wordt ook nog even de naam getoond (via het *echo* commando).

Hier is een shell script wat het bovenstaande voorbeeld genereert:

```
echo '#To unbundle, sh this file'
for f in $*
do
  echo "# -------------- # $f # -------------- #"
  echo "echo '$f'"
  echo "cat > $f <<'End of file $f'"
  cat $f
  echo "End of file $f"
done
```

Als verdere uitbreiding willen we het commando *bundel* een optie *-v* geven die tot gevolg heeft dat bij het *inpakken* de filenamen ook afgedrukt worden als deze optie gegeven is. We moeten dan alleen uitkijken dat de filenamen op *standaard error* afgedrukt worden en niet op *standaard uitvoer*, want anders komen ze tussen het uitpak-script in. We beginnen nu het script met een loop waarin we eventuele *-v* argumenten verwijderen, en als er een voorkomt, zetten we een variabele *VERBOSE* op *"y"*. Van tevoren zetten we deze op *"n"* en later in het script testen we voor iedere filenaam of *VERBOSE = "y"*.

Als we een *-v* tegenkomen moet deze verwijderd worden zodat hij niet als filenaam geïnterpreteerd kan worden. Dit gebeurt met de *shift* opdracht. Deze verwijdert het eerste argument van het shellscript (*$1*) en schuift alle andere argumenten één plaats op zodat *$2* nu *$1* wordt, etc.

Verder zien we in het script de *break* opdracht die tot gevolg heeft dat de loop verlaten wordt. We doen dit als we een ander argument dan *-v* tegenkomen, want dan nemen we aan dat het een filenaam is.

```
VERBOSE="n"
for f in $*
do
  case $1 in
    -v) VERBOSE="y"; shift;;
    *)  break;;
  esac
done

echo '#To unbundle, sh this file'
for f in $*
do
  echo "# --------------- # $f # --------------- #"
  if [ "$VERBOSE" = "y" ]
  then echo "bundling $f" 1>&2
  fi
  echo "echo '$f'"
  echo "cat > $f <<'End of file $f'"
  cat $f
  echo "End of file $f"
done
```

Opgave 4.2 Verander het shell script zo dat het afdrukken van de filenamen bij het *uitpakken* óók door de *-v* optie gestuurd wordt.

Tenslotte kunnen we nog testen of de files die als argumenten opgegeven worden wel echte files zijn en gelezen kunnen worden:

```
if [ -f $f -a -r $f ]
then
⟨als hierboven⟩
else
  echo "Illegal file $f" 1>&2
fi
```

4.1.13 Shell functies

In bovenstaand voorbeeld hebben we twee keer een boodschap die naar standaard error gestuurd moet worden (via de *1>&2* constructie). Dit vraagt om het gebruik van een functie.

Shell functies worden gedefinieerd a.v.

```
⟨functienaam⟩ ()
{ ⟨lijst⟩
}
```

Let erop dat tussen de { en de ⟨lijst⟩ een spatie moet staan en dat de } op een aparte regel

moet staan. De { mag ook op een aparte regel staan.

De functie kan worden aangeroepen als een normaal commando, met argumenten. De ⟨lijst⟩ tussen {} wordt dan uitgevoerd. Binnen de functie zijn *$1* etc. de argumenten van de functie, niet die van het shell script. In de gewone (Bourne) shell zijn na een functieaanroep de originele argumenten van het shell script verdwenen. Die zullen dus eerst in gewone variabelen gestopt moeten worden.

Bij het gebruik van een functie *err_out* voor het geven van de meldingen op standaard error wordt ons voorbeeld zoals in figuur 4.5.

Figuur 4.5 Bundle script met tests en shell functie

```
#! /bin/sh
VERBOSE="n"

err_out()
{
  echo "$*" 1>&2
}

for f in $*
do
  case $1 in
    -v) VERBOSE="y"; shift;;
    *)  break;;
  esac
done

echo '#To unbundle, sh this file'
for f in $*
  do
  if [ -f $f -a -r $f ]
  then
    echo "# --------------- # $f # --------------- #"
    if [ "$VERBOSE" = "y" ]
    then err_out "bundling $f"
    fi
    echo "echo '$f'"
    echo "cat > $f <<'End of file $f'"
    cat $f
    echo "End of file $f"
  else
    err_out "Illegal file $f"
  fi
done
```

Opgave 4.3 Maak een uitbreiding zodat een directory naam als argument betekent dat alle

files uit die directory (inclusief geneste subdirectories) ook in de bundel opgenomen worden. Definieer hiervoor een shell functie die één file- of directorynaam behandeld en die recursief aangeroepen wordt.

4.1.14 Overzicht

In tabel 4.5 staat een overzicht van de metacharacters van de shell.

<table>
<tr><th colspan="2">I/O redirection</th></tr>
<tr><td>></td><td>prog > file schrijft standaard uitvoer naar file</td></tr>
<tr><td>>></td><td>prog > file schrijft standaard uitvoer achter file</td></tr>
<tr><td><</td><td>prog < file leest standaard invoer van file</td></tr>
<tr><td><<word</td><td>here document</td></tr>
<tr><td>|</td><td>prog1 |prog2 standaar invoer van prog2 is standaard uitvoer van prog1</td></tr>
<tr><th colspan="2">wildcards</th></tr>
<tr><td>*</td><td>match 0 of meer characters</td></tr>
<tr><td>?</td><td>match 1 character</td></tr>
<tr><td>[xyz]</td><td>match 1 character uit de verzameling xyz</td></tr>
<tr><th colspan="2">quotes</th></tr>
<tr><td>\</td><td>neem het volgende teken letterlijk</td></tr>
<tr><td>'...'</td><td>Neem alles van ... letterlijk over</td></tr>
<tr><td>"..."</td><td>Neem ... letterlijk over, behalve $, `...` en \</td></tr>
<tr><th colspan="2">samengestelde commando's</th></tr>
<tr><td>;</td><td>prog1 ; prog2 doe eerst prog1, dan prog2</td></tr>
<tr><td>&</td><td>prog & voer prog uit op de achtergrond</td></tr>
<tr><td>&&</td><td>prog1 && prog2 eerst prog1, bij succes gevolgd door prog2</td></tr>
<tr><td>||</td><td>prog1 || prog2 eerst prog1, bij niet-succes gevolgd door prog2</td></tr>
<tr><td>(...)</td><td>Voer ... uit in een subshell</td></tr>
<tr><td>{...}</td><td>Voer ... uit (groepering)</td></tr>
<tr><td>`...`</td><td>Vervang `...` door de uitvoer van ...</td></tr>
<tr><th colspan="2">diversen</th></tr>
<tr><td>$</td><td>$<i>var</i> of ${<i>var</i>} variabele substitutie</td></tr>
<tr><td>#</td><td>commentaar (aan het begin van een woord)</td></tr>
</table>

Tabel 4.5: Shell metacharacters

4.2 De Korn shell

De belangrijkste verschillen van de Korn shell met de Bourne shell zijn:

- Het teken ~ aan het begin van een woord is ook een metacharacter. Als het alleen staat of gevolgd wordt door een / dan betekent het hetzelfde als *$HOME*. Als het gevolgd wordt door een loginnaam van een andere gebruiker dan wordt het vervangen door de hoofddirectory van die gebruiker.

- Een functie heeft een eigen verzameling parameters *$1* etc. die niet die van het shell script of die van andere functies vernietigen.
- Het is mogelijk om in een functie locale variabelen te definieren met het *typeset* ⟨naam⟩ commando
- Het is mogelijk om eerder gebruikte commando's terug te halen en evt. gewijzigd weer uit te voeren.

Een shell script kan gemarkeerd worden als bestemd voor de Korn shell door Als eerste regel op te nemen

```
#! /bin/ksh
```

4.3 De C shell

De C shell gebruikt een totaal andere syntax voor samengestelde commando's zoals *if* en *while*. De syntax van deze commando's lijkt het meest op die van de taal C, maar is op sommige punten toch weer anders. Een paar voorbeelden:

```
if ( ⟨expressie⟩ ) ⟨commando⟩
if ( ⟨expressie⟩ ) then
  ⟨then-deel⟩
else
  ⟨else-deel⟩
endif
while ( ⟨expressie⟩ )
  ⟨loop-deel⟩
end
switch ⟨string⟩
case ⟨geval⟩:
  ⟨geval-deel⟩
endsw
```

Voor het zetten van gewone variabelen wordt de constructie *set* ⟨naam⟩=⟨waarde⟩ gebruikt, voor environment variabelen de constructie *setenv* ⟨naam⟩ ⟨waarde⟩ (dus zonder =).

De C shell heeft ook ingebouwde expressies zodat je kunt zeggen:

> *if ($n < 10)* (Het < teken is binnen een expressie niet speciaal, net als de andere metacharacters)
> *@ n = dlr m + 1* (Het *@* teken wordt gebruit i.p.v. *set* als de waarde van een expressie toegekend moet worden aan een variabele)

Hoofdstuk 5

Filters

In de Unix terminologie is een *filter* een programma dat leest van standaard invoer en schrijft op standaard uitvoer, maar verder niets aan de omgeving verandert. Zie figuur 2.1 op pagina 13. Een heel simpel filter is het programma *cat* dat alleen maar de invoer naar de uitvoer copieert.

Zoals we in hoofdstuk 3 al gezien hebben, heeft *cat* de volgende conventie: Als er één of meer filenamen opgegeven zijn dan worden deze files gelezen. Als er geen filenaam opgegeven is dan wordt van standaard invoer gelezen.

Heel veel filters hebben ook deze conventie. In het geval dat er filenamen opgegeven worden lezen ze deze files één voor één en verwerken ze alsof ze aan elkaar geplakt waren. Er zijn enkele filters die dit niet doen, maar je kunt in dat geval hetzelfde effect krijgen door te schrijven: *cat* ⟨files⟩ | *filter*.

Om nog eens het verschil tussen een filter en een niet-filter te illustreren: de volgende twee opdrachten doen hetzelfde:

```
cat < a > b
cp a b
```

maar *cat* wordt als filter gebruikt terwijl *cp* *geen* filter is (want het maakt een file aan).

Een groot voordeel van filters is dat ze via *pipes* aan elkaar gekoppeld kunnen worden. Omdat filters vaak gebruikt worden om op de uitvoer van een ander programma te werken, en dus niet alleen op files, zullen we de term *stroom* gebruiken voor de in- en uitvoer van een filter.

De belangrijkste filters op Unix zijn:

- *grep*, *egrep*, *fgrep*, voor het selecteren van regels die een bepaalde tekst bevatten.
- *tr* voor het omzetten van tekens in andere tekens
- *uniq* voor het ontdekken of verwijderen van dubbele regels in een stroom
- *sort* voor het sorteren van teksten
- *head* en *tail* voor het bekijken van het begin of eind van een stroom

- *sed* voor het aanbrengen van wijzigingen in een stroom
- *comm* voor het vergelijken van twee files (dit is niet 100% een filter)
- *awk* een filter dat voor heel veel doeleinden gebruikt kan worden en daarom een eigen hoofdstuk (6) krijgt.

Let op: als je een filter wilt gebruiken om een wijziging **in een file** aan te brengen kun je dat *niet* op deze manier doen:

`filter file > file` (*ook niet* `filter < file > file`)

De reden is dat de shell de uitvoer (`file`) opent voordat het filter aktief wordt en dus de originele inhoud ervan weggooit voordat het filter zelfs maar de kans krijgt om het te lezen. Je eindigt dan dus met een lege file. Het is trouwens nooit aan te bevelen om *originele* files te overschrijven. Alleen files die gemakkelijk te herstellen zijn kun je zo gebruiken, maar dan moet je de uitvoer van het filter eerst naar een andere file schrijven en deze dan hernoemen. Bijvoorbeeld:

```
filter file > filter.tmp$$ && mv filter.tmp$$ file
```

We gebruiken `$$` (het procesnummer van de shell) zodat er geen conflict ontstaat als we toevallig twee keer hetzelfde filter op hetzelfde moment zouden gebruiken.

5.1 Grep, egrep, fgrep

grep is een programma dat een tekststroom leest en de regels waarin een bepaald stuk tekst voorkomt eruit filtert. *egrep* en *fgrep* zijn speciale varianten ervan. Bijvoorbeeld het commando

```
grep de < a > b
```

leest de file *a* en geeft in *b* alleen die regels waarin de tekst “de” voorkomt. De opgegeven tekst hoeft niet als los woord voor te komen maar kan ook onderdeel van een ander woord zijn. We gaan in deze sectie uit van de invoer uit figuur 5.1 voor *grep* (in de file “text”):

We geven nu het volgende commando:

```
grep de text
```

en krijgen dan:

Figuur 5.1 Invoer voor *grep*

```
De koper
van de lage
delen
en de
belendende percelen
voelde zich in geen enkel
opzicht verantwoordelijk voor
het welslagen van het project
en had op de keper beschouwd daar
waarschijnlijk geen ongelijk in.
```

```
van de lage
delen
en de
belendende percelen
voelde zich in geen enkel
opzicht verantwoordelijk voor
en had op de keper beschouwd daar
```

Er vallen een paar dingen op:

1. Het maakt verschil of een tekst met hoofdletters of kleine letters geschreven wordt. Als we dit onderscheid niet willen dan moeten we de *-i* optie meegeven (=*ignore*).
2. Zoals al eerder opgemerkt worden ook regels waarin “de” als deel van een woord voorkomt gegeven, bijvoorbeeld de regel “belendende percelen”. Als we alléén de aparte woorden “de” willen selecteren dan moeten we moeilijker doen.

We zouden kunnen proberen deze problemen op te lossen door de zoeken naar het woord “de” met een spatie eromheen.

```
grep -i ' de ' text
```

Let erop dat we de spaties tussen quotes gezet hebben om ze tegen de shell te beschermen. We krijgen nu als uitvoer:

```
van de lage
en had op de keper beschouwd daar
```

Inderdaad, de “belendende” is niet meer geselecteerd, maar een paar andere ook niet, nl. de regels “De koper” en “en de”. Dit komt omdat daar maar aan één kant van het woord “de” een spatie voorkomt.

We willen dus eigenlijk zoeken naar het woord “de” dat aan het begin van een regel staat of voorafgegaan wordt door een spatie en bovendien ook nog gevolgd wordt door een spatie of aan het eind van een regel staat.

Om dit soort dingen mogelijk te maken, zoekt *grep* niet alleen maar naar teksten maar naar *patronen*. Een patroon staat voor een hele verzameling teksten. We zeggen dat de teksten het patroon *matchen*, en grep geeft elke regel van de invoer waarin een *deel* voorkomt dat het patroon (het eerste argument) matcht. De patronen noemen we *reguliere expressies*, ze zijn te vergelijken met de filenaampatronen bij de shell, maar er kan meer mee.

5.1.1 Reguliere expressies

In een reguliere expressie zijn de volgende tekens speciaal:

^	begin van de regel
$	einde van de regel
.	matcht ieder teken
[*xyz*]	match 1 teken uit de verzameling *xyz*
[^*xyz*]	match 1 teken uit de verzameling *xyz*
*	matcht nul of meer keer het voorgaande
\	matcht het volgende teken wordt (dit is dan dus niet meer speciaal)

Omdat de meeste van deze tekens ook speciaal zijn voor de shell verdient het aanbeveling om reguliere expressies tussen quotes (' ') te zetten.

Tussen de *[ ]* mogen ook reeksen als *a-z* gebruikt worden. De speciale tekens (*$. [* *) zijn binnen de *[ ] niet speciaal*. Alleen met de tekens ^ *-]* moet je natuurlijk oppassen: ^ kan als gewoon teken niet als eerste voorkomen, *]* moet juist als eerste (of na het ontkennings-^) voorkomen en *-* moet als eerste of laatste gebruikt worden. Als je dus wilt aangeven dat één van de tekens ^ *] -* gewenst is dan kan dit alleen met het patroon "*[]^-]*".

Voorbeelden:

r.e	*matcht*	*matcht niet*
a	a	A b
.	a * X	
[xyz]	x y z	X abc
[^0-9]	a . -	0 1 2
*	*	\
\[	[	\
a*	a aa aaa	b
[a-z]*	nul of meer a's	A B
.*	nul of meer tekens	
a.*b	alles dat met een a begint en op b eindigt	b a ba ac

5.1.2 Speciale reguliere expressies voor *grep*

In *grep* kunnen delen van een reguliere expressie aangegeven worden met de constructie *\(... \)*. Naar een dergelijk stuk kan dan verderop gerefereerd worden door *\1* ... *\9*. *\1* matcht dan exact dezelfde tekst (dus niet alleen maar hetzelfde patroon) als de *\(... \)* gematcht heeft. Dus "*\([a-z][a-z]*\) \1*" match als twee keer hetzelfde woord, met een

spatie ertussen, voorkomt. (Let op: *egrep* heeft deze mogelijkheid niet.) Wanneer meer keer de constructie *\(... \)* gebruikt wordt dan slaat *\1* op het deel dat door de eerste *\(... \)* gematcht werd, *\2* op de tweede etc. Daarbij worden alleen de *\(* geteld.

5.1.3 Speciale reguliere expressies voor *egrep*

Het commando *egrep* kent ingewikkelder reguliere expressies. Daar is ook nog het volgende toegestaan:

?	matcht nul of één keer het voorgaande
+	matcht één of meer keer het voorgaande
⟨abc⟩ | ⟨xyz⟩	match òf ⟨abc⟩ òf ⟨xyz⟩
(...)	groepering om ingewikkelder constructies mogelijk te maken

Voorbeelden voor *egrep*:

r.e	*matcht*	*matcht niet*
abc|xyz	abc xyz	Ayz
(abc|xyz)+	abc xyz xyzabc	abxyzc

In de meeste programmeertalen kunnen getallen gebruikt worden die bestaan uit een + of − teken (dat weggelaten mag worden) gevolgd door één of meer cijfers. De reguliere expressie "*[+-]?[0-9]+*" drukt dit precies uit. Om nog een ingewikkelder voorbeeld te nemen: floating point getallen (reële getallen) kunnen in het algemeen opgegeven worden als een rijtje cijfers, eventueel gevolgd door een "." en nog meer cijfers. De punt mag weggelaten worden, de cijfers vóór of achter de punt ook, maar niet beide. We kunnen dit aangeven met twee gevallen: cijfers voor de punt en nul of één keer een punt met evt. cijfers (dus "*[0-9]+(\.[0-9]*)?*") òf een punt met één of meer cijfers erachter (dus "*\.[0-9]+*"). Deze twee gevallen kunnen we met | combineren. Zo'n getal mag dan nog voorafgegaan worden door + of − en het mag gevolgd worden door een *exponent*, dat is een "e" of "E" met een geheel getal erachter. We krijgen dan dus:

```
[+-]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][+-]?[0-9]+)?
```

Opgave 5.1 In de taal C en in andere is het zo dat een geheel getal, als het begint met "0" als octaal getal beschouwd wordt. Dan mogen er dus aleen maar de cijfers 0 t/m 7 in voorkomen. Bovendien als een getal met "0x" begint is het hexadecimaal, dan mogen ook de "cijfers" a t/m f (of A t/m F) voorkomen. Geef hiervoor één reguliere expressie. Kan dit met de *grep* reguliere expressies?

We gaan nu verder met ons voorbeeld om te zoeken naar het woord "de". Dit moet dus aan het begin van een regel staan òf voorafgegaan worden door een spatie, en bovendien moet het gevolgd worden door een spatie òf aan het eind van de regel staan. We kunnen dit alleen met *egrep* doen, en wel met het commando:

```
egrep -i '(^| )de( |$)' text
```

Wanneer we in het algemeen naar woorden willen zoeken in een tekst dan moeten we er rekening houden dat het woord ook wel voorafgegaan of gevolgd kan worden door diverse

leestekens, haakjes, e.d. De spaties in het laatste voorbeeld zouden dus beter vervangen kunnen worden door zoiets als: *[!.,;?()]* of nog meer. Of beter zelfs door een reguliere expressie die alles matcht wat niet in een woord kan voorkomen. In het nederlands kunnen behalve letters ook *-* en *'* voorkomen, dus dan kunnen we gebruiken *[^A-Za-z'-]* en de opdracht wordt dan:

```
egrep -i "(^|[^A-Za-z'-])de([^A-Za-z'-]|$)" text
```

We moeten nu *""* gebruiken omdat er een ' in de expressie staan en die kan de shell niet tussen ' ' hebben. De *$* tussen de *" "* is in dit geval onschadelijk omdat *$)* geen geldige variabele is.

Tenslotte nog een *grep* voorbeeld. We willen zoeken naar dubbele klinkers in de file text.

```
grep '[aeiou][aeiou]' text
```

geeft alle regels waarin twee klinkers achter elkaar voorkomen. De klinkers hoeven niet hetzelfde te zijn. Willen we alleen die regels waarin dubbele klinkers (wel hetzelfde) voorkomen dan geven we:

```
grep '\([aeiou]\)\1' text
```

5.1.4 Fgrep

Voor het zoeken van letterlijke strings (dus zonder speciale reguliere expressie patronen) kan *fgrep* gebruikt worden. Dit is sneller dan *grep* of *egrep*. Elk teken in het zoekargument betekent bij *fgrep* dus zichzelf. Het is ook mogelijk om een lijst van strings in een file te zetten en *fgrep* naar deze strings te zoeken. Elke regel waar tenminste één van de strings voorkomt wordt dan gegeven. Dit gebeurt door het commando *fgrep -f* ⟨patfile⟩ waarbij ⟨patfile⟩ de strings bevat (één per regel).

Tabel 5.1 bevat de belangrijkste opties voor grep, egrep en fgrep. Deze programma's geven zonder speciale opties vóór iedere regel de filenaam als meer dan één filenaam opgegeven wordt.

5.2 Tr

tr is een filter dat tekens kan vertalen in andere tekens. Het wordt meestal gebruikt voor zoiets als het omzetten van hoofdletters in kleine letters of omgekeerd, maar het kan meer.

```
tr ABC XYZ
```

leest standaard invoer, vertaalt daarin A naar X, B naar Y en C naar Z en schrijft het resultaat naar standaard uitvoer. Alle andere tekens worden ongewijzigd doorgegeven. Je kunt bij *tr* geen filenamen opgeven, dus alleen van standaard invoer lezen. De beide parameters moeten evenveel tekens bevatten.

Opties voor grep, egrep, fgrep	
`-c`	geef alleen het *aantal* regels dat matcht
`-i`	maak geen onderscheid tussen hoofd- en kleine letters
`-l`	print alleen de namen van de files waarin het patroon voorkomt (één naam per regel, elke naam max. 1 keer)
`-n`	print ook het regelnummer van de gematchte regels
`-s`	print niets, maar geef alleen via de exit status aan of er een match was (0 als er een match is, 1 als er geen is, 2 als er een fout is)
`-v`	print de regels die *niet* matchen
`-e` ⟨pat⟩	⟨pat⟩ is het zoekpatroon (voor als het patroon met een `-` begint)
`-f` ⟨file⟩	lees het zoekargument van ⟨file⟩
Opties voor fgrep	
`-x`	Geef alleen een match als de *hele regel* matcht (alsof er een *grep* ^ en *$* omheen zou staan)

Tabel 5.1: Opties voor grep, egrep en fgrep

Om het makkelijker te maken aaneensluitende reeksen van tekens op te geven accepteert *tr* een notatie als *[0-9]*, wat hetzelfde is als *0123456789*. De *[]* zijn dus onderdeel van het argument en moeten dus gequote worden voor de shell. Om hoofdletters om te zetten in kleine letters gebruik je dus het commando

```
tr '[A-Z]' '[a-z]'
```

Er zijn 3 opties die het gedrag van *tr* beïnvloeden:

`-c`	vertaal niet de tekens die als eerste argument meegegeven zijn maar juist alle andere
`-d`	vertaal de tekens uit het eerste argument niet maar gooi ze weg (het tweede argument hoeft dan dus niet gegeven te worden)
`-s`	vervang rijtjes van hetzelfde vertaalde teken door één exemplaar

Als we bijvoorbeeld een tekst hebben waar we alleen de woorden uit willen hebben en niet de leestekens e.d. dan kunnen we alles dat niet een letter is vertalen naar een spatie. Alleen zouden we dan in het tweede argument zoveel spaties moeten geven als er niet-lettertekens zijn (256-52=204). Dit kan met de volgende notatie:

```
tr -c '[A-Z][a-z]' '[ *204]'
```

In plaats van het exacte aantal mag ook **0* of *** gegeven worden, dit betekent: neem zoveel als er nodig zijn.

In bovenstaand voorbeeld worden ook newline tekens vervangen door spaties, want *tr* werkt niet op regels maar op afzonderlijke tekens. Als we de newlines willen laten staan dan moeten we deze apart opnemen bij de letters. We kunnen newline aangeven als *\012* waar 012 de octale ASCII code is. Op die manier kan je niet-zichtbare tekens in de vertaalargumenten opnemen. In bovenstaand voorbeeld komen in de uitvoer ook nog te veel spaties voor, want

als er een rijtje leestekens is of leestekens met spaties dan wordt elk teken in een spatie omgezet terwijl één spatie genoeg zou zijn. De *-s* optie (van "squeeze") laat van zulke rijtjes tekens er maar één staan. Tekens die niet in het tweede argument voorkomen worden niet gesqueezed. We krijgen dan dus de opdracht:

```
tr -cs '\012[A-Z][a-z]' '[ *]'
```

Bij dit voorbeeld blijft de regelindeling van de invoer bewaard. Als we ook nog elk woord afzonderlijk op een regel willen hebben dan moeten we alle niet-letters vertalen naar een newline, en natuurlijk ook de squeeze optie gebruiken, dus:

```
tr -cs '[A-Z][a-z]' '[\012*]'
```

Opgave 5.2 Kan het hierbij voorkomen dat er een lege regel in de uitvoer zit?

Tenslotte kan de *-d* optie nog gebruikt worden om tekens niet te vertalen maar weg te gooien. Het volgende commando verwijdert alle haakjes uit de invoerstroom

```
tr -d '()'
```

De tweede parameter heeft hier niet zoveel zin, behalve als je ook de squeeze optie wilt gebruiken (de tweede parameter geeft dan aan welke tekens ge"squeezed" worden).

5.3 Uniq

uniq is een filter dat dubbel voorkomende regels in een stroom kan ontdekken. Zonder opties verwijdert *uniq* een regel als deze hetzelfde is als de vorige. Dus van een aantal regels achter elkaar die hetzelfde zijn blijft er in de uitvoer maar één exemplaar over. Regels die hetzelfde zijn maar niet bij elkaar staan worden *niet* ontdekt. Om dezelfde regels bij elkaar te krijgen kan het programma *sort* (5.4) gebruikt worden.

Er zijn een aantal opties om het gedrag van *uniq* te beïnvloeden, zie tabel 5.2. Het gewone gedrag van *uniq* is in feite een combinatie van *-u* en *-d*. Voor een voorbeeld van het gebruik van *uniq* zie sectie 5.4.

-u	print alleen de regels die *niet* dubbel voorkomen in de input
-d	print van iedere dubbel voorkomende regel één exemplaar
-c	print iedere unieke regel voorafgegaan door het aantal keren dat deze regel (aaneensluitend) voorkomt. In dit geval hebben de opties *-u* en *-d* geen effect.

Tabel 5.2: Opties van *uniq*

5.4 Sort

sort is een programma om files te *sorteren*, d.w.z. op een bepaalde volgorde te zetten. Het verandert niets aan de files zelf, maar het leest een of meer files (of standaard invoer) en

schrijft de regels (waarschijnlijk in een andere volgorde) op standaard uitvoer. De invoer kan gesorteerd worden op alfabetische volgorde maar ook op getalvolgorde.

We zullen in deze sectie het probleem behandelen hoe een lijst van de meest gebruikte woorden van een tekstfile te krijgen. We hebben in sectie 5.2 al gezien hoe we van een tekstfile een file met één woord per regel maken. We gaan er dus vanuit dat we een file hebben waar elk woord op een regel staat.

In de volgende voorbeelden gaan we uit van de volgende invoer in file *text* (zie figuur 5.2).

Figuur 5.2 Invoer voor *sort*

```
quasi
a
aap
AA
ROOD
b
zijn
rood
Rood
```

Het commando *sort text* levert op:

```
AA
ROOD
Rood
a
aap
b
quasi
rood
zijn
```

Wat we zien is dat woorden met een hoofdletter eerst komen, daarna die met een kleine letter. Dit komt omdat *sort* de ASCII-volgorde (zie tabel 3.1) hanteert als er niets speciaals aangegeven wordt en daarin staan hoofdletters voor kleine letters.

Wanneer we hoofd- en kleine letters bij elkaar willen sorteren, dan gebruiken we de *-f* (fold letters) optie.

Bijvoorbeeld *sort -f text* levert:

```
a
AA
aap
b
quasi
ROOD
Rood
rood
zijn
```

We kunnen nu van elk woord één exemplaar overhouden door de *-u* optie aan *sort* te geven. Deze zegt dat van de regels die gelijk sorteren er maar één genomen moet worden. Of we dan hoofd- of kleine letters overhouden hangt van het toeval af.

We zouden ook *uniq* (zie 5.3) kunnen gebruiken maar dan moeten we er voor zorgen dat eerst alle hoofdletters vervangen zijn door kleine letters of omgekeerd, want *uniq* maakt wèl onderscheid tussen hoofd- en kleine letters. Zoals we al gezien hebben kan dit met het programma *tr* (sectie 5.2). Het commando wordt dan dus:

```
sort -fu text
of
tr '[A-Z]' '[a-z]' < text | sort | uniq
```

Hier volgt de uitvoer van beide commando's:

```
a          a
AA         aa
aap        aap
b          b
quasi      quasi
Rood       rood
zijn       zijn
```

Het voordeel van het gebruik van *uniq* is dat we het ook kunnen gebruiken om het aantal van ieder woord te tellen. Omdat *sort* de woorden al bij elkaar gezet heeft kunnen we gewoon *uniq -c* gebruiken en de uitvoer van de pipeline
tr '[A-Z]' '[a-z]' < text | sort | uniq -c wordt dan

```
1 a
1 aa
1 aap
1 b
1 quasi
3 rood
1 zijn
```

Als we nu willen weten welke woorden het meest gebruikt zijn, dan moeten we deze uitvoer weer sorteren, maar nu niet op tekstvolgorde maar op getalvolgorde. Hiervoor gebruiken we *sort -n*. Omdat we de meest gebruikte woorden bovenaan willen hebben, moeten we omgekeerd (*reversed*) sorteren met de *-r* optie. Dus:

```
tr '[A-Z]' '[a-z]' < text | sort | uniq -c | sort -nr
```

Optie	Betekenis
-f	Sorteer hoofd- en kleine letters bij elkaar (de inhoud wordt niet gewijzigd)
-i	niet-zichtbare tekens (ASCII codes octaal 001 t/m 037 en 177) worden niet meegenomen in de vergelijking
-n	Sorteer op getalvolgorde. De regels moeten met een getal beginnen, maar er mag nog meer op staan. Verdere tekst is niet van invloed op de sortering. De *-b* optie wordt ook aangezet.
-d	woordenboekvolgorde: alleen letters, cijfers en spaties worden gebruikt bij het bepalen van de volgorde (dus bijvoorbeeld streepjes beïnvloeden de volgorde niet)
-r	sorteer op de omgekeerde volgorde
-b	Blanks (spaties en tabs) aan het begin van een veld worden niet meegenomen bij het tellen voor de *-k* optie
-M	Sorteer op maand volgorde (sorteerveld moet de naam van een maand bevatten)
-u	uniek: van regels die dezelfde sorteersleutel hebben (dus wat sorteren betreft op dezelfde plaats komen) wordt er maar één exemplaar doorgegeven
-c	Sorteer niet maar controleer alleen of de invoer al gesorteerd is (geen standaard uitvoer, alleen exit status)
-m	Merge: de invoer files moeten al gesorteerd zijn en worden in elkaar geschoven
-t⟨char⟩	⟨char⟩ is de veldscheider
-k⟨sleutel⟩	zie de tekst
-o⟨file⟩	schrijf de uitvoer naar ⟨file⟩ i.p.v. naar standaard uitvoer. Dit mag hetzelfde zijn als één van de invoer files.

Tabel 5.3: Sort opties

In tabel 5.3 geven we de belangrijkste opties van *sort*. In bovenstaande voorbeelden hebben we steeds aangenomen dat de regels in hun geheel bepaalden hoe gesorteerd moet worden. Het is mogelijk aan sort op te geven dat slecht een deel van de regel gebruikt wordt voor het bepalen van de volgorde (wèl wordt ook dan de complete regel doorgegeven naar de uitvoer). Dit kan nuttig zijn als bijvoorbeeld elke regel naam, adres en woonplaats van iemand bevat, maar je wilt op woonplaats sorteren. Met de *-k* optie kan opgegeven worden wat de *sorteersleutel* is. Hiervoor wordt de regel ingedeeld in *velden* die gescheiden worden door een *veldscheider*. De veldscheider kan aangegeven worden met de *-t* optie, bijvoorbeeld *-t:* geeft aan dat een *:* als scheider gebruikt wordt.

Het deel van de regel vanaf het begin tot de eerste *:* is dan veld 1, het deel vanaf het teken na de eerste *:* tot de volgende *:* is veld 2 etc, dus de *:* is geen onderdeel van het veld.

Als géén *-t* optie wordt gegeven dan worden spaties en tabs als veldscheider gebruikt, maar dan geldt van een aaneengesloten rijtje dat de eerste de veldscheider is en dat de daaropvolgende bij het volgende veld horen (tenzij de *-b* optie gegeven wordt). Bij de *-t:* optie betekent *::* een leeg veld tussen de *:*'s.

Een sorteersleutel wordt nu aangegeven als *-k* ⟨m.n⟩⟨t⟩,⟨m.n⟩⟨t⟩. ⟨m⟩ en ⟨n⟩ zijn getallen, waarbij ⟨m⟩ het veldnummer aangeeft en ⟨n⟩ het nummer van het teken in het veld. Het deel voor de komma geeft aan waar de sleutel begint, het deel na de komma waar de sleutel stopt. Als dit deel weggelaten wordt dan loopt de sleutel tot het eind van de regel door. Als extra kan nog het ⟨t⟩ veld aangegeven worden, wat een combinatie van de opties *b d f i n r M* mag zijn. Deze gelden dan alleen voor deze sleutel. Zo kan je dus bijvoorbeeld één veld als getal sorteren en een ander omgekeerd. Wanneer meer dan één sorteersleutel wordt opgegeven dan wordt eerst op de eerste gesorteerd, daarbinnen op de tweede etc. Dus bijvoorbeeld eerst op woonplaats en binnen de woonplaats op straat.

Voorbeelden:

> *-k 2.5* betekent dat de sleutel loopt van het vijfde teken van het tweede veld tot het eind van de regel
> *-k 2.1,4.10* betekent dat de sleutel begint op het eerste teken van veld 2 en loopt t/m het tiende teken van veld 4
> *-k 2,4* betekent dat de sleutel loopt van veld 2 t/m veld 4 (als het ⟨.n⟩ gedeelte weggelaten wordt betekent het het begin van het veld voor de komma en het eind van het veld na de komma).

N.B. Op sommige Unix systemen moeten de sleutels opgegeven worden in de vorm *+1.0 -3.9* *+* en *-* worden gebruikt voor begin en eind van de sleutel en de nummers beginnen op 0 i.p.v. 1.

5.5 Head en tail

Het commando *head* -⟨n⟩ geeft de eerste ⟨n⟩ regels van de standaard invoer of van elk van de opgegeven files. Als we bij het voorbeeld in sectie 5.4 dus de 20 meest gebruikte woorden uit een file willen hebben dan kunnen we dat doen met *head -20* (als we geen -⟨n⟩ opgeven dan wordt automatisch 10 genomen).

Het omgekeerde is *tail* -⟨n⟩, dat de laatste ⟨n⟩ regels geeft, of *tail* +⟨n⟩ dat vanaf regel ⟨n⟩ begint. Het nummer ⟨n⟩ mag ook gevolgd worden door het teken *c* dan worden characters geteld i.p.v. regels.

Als we de voorbeelden uit de vorige secties bij elkaar nemen en de meest gebruikte woorden uit een tekstfile willen hebben (waarbij er meer woorden, leestekens e.d op een regel mogen staan) dan krijgen we de volgende pipeline:

```
tr '[A-Z]' '[a-z]' < text | tr -cs '[a-z]' '[\012*]' |
                 sort | uniq -c | sort -nr | head -10
```

I.p.v. *sort -nr | head -10* kunnen we ook zeggen: *sort -n | tail -10*. In het eerste geval staat het meest gebruikte woord bovenaan, in het tweede geval onderaan.

Opgave 5.3 Gebruik *ls* en *head* of *tail* om de nieuwste file in je werkdirectory te vinden.

5.6 Sed

Al de tot nu toe behandelde filters lieten de afzonderlijke regels van de invoer ongemoeid. Alleen *tr* kon gebruikt worden om afzonderlijke tekens om te zetten, maar dit kan niet gebruikt worden om bijv. "de" te vervangen door "het". De andere filters laten wel regels weg of veranderen de volgorde, maar kunnen geen wijzigingen in de regels maken. Het programma *sed* (*stream editor*) is hiervoor bedoeld. Het kan gebruikt worden voor simpele substituties (een tekst vervangen door een andere tekst bijv.), om enkele regels weg te laten of toe te voegen, maar voor gecompliceerdere operaties is het niet zo geschikt. Daarvoor kan beter het programma *awk* (hoofdstuk 6) gebruikt worden.

Het programma *sed* heeft ook wat ingewikkelder mogelijkheden maar die zullen we niet behandelen omdat die beter met *awk* gedaan kunnen worden.

Het meest simpele gebruik van *sed* is om een tekst door een andere tekst te vervangen. Het commando:

```
sed s/de/het/ text
```

copieert de file "text" naar de standaard uitvoer, maar vervangt in elke regel "de" door "het" (ook als onderdeel van een ander woord). Alleen de eerste "de" per regel wordt veranderd tenzij *s/de/het/g* wordt gegeven (g=global, dan wordt het in de hele regel gedaan). In plaats van "g" kan ook nog een nummer gegeven worden bijv. 3 als de derde "de" in elke regel vervangen moet worden. In plaats van de "/" mag ook een ander teken genomen worden, als je alle 3 keer maar hetzelfde gebruikt.

Er mag ook een (simpele) reguliere expressie (zoals in grep, zie 5.1.1) vervangen worden, in de vervangingstekst kunnen dan de *\1* ... constructies uit 5.1.2 gebruikt worden. Hiermee kun je dus o.a. stukken tekst verwisselen. Stel je hebt regels waarin tekst voorkomt van de vorm "*x > y*" en je wilt die omzetten in "*y < x*" dan kan dat (als we aannemen dat er alleen letters omheen staan) met:

```
sed 's/\([A-Za-z]\) > \([A-Za-z]\)/\2 < \1/g'
```

Let op de quotes om de metacharacters te beschermen.

Het probleem uit sectie 5.2 om alle leestekens uit een stroom te verwijderen en te vervangen door één spatie per serie kan ook met het volgende commando:

```
sed 's/[^a-zA-Z][^a-zA-Z]*/ /g' text
```

Omdat *sed* op regels werkt wordt de newline niet vervangen.

Het is ook mogelijk om *sed* niet op alle regels te laten werken maar op een selectie. Hiervoor wordt het "*s///*" commando voorafgegaan door een selectie. De selectie kan zijn:

- een getal ⟨n⟩, dan wordt alleen de regel nummer ⟨n⟩ gedaan. Als meer dan één filenaam meegegeven wordt dan worden de regels doorgenummerd.
- *$*, dit betekent de laatste regel

- een reguliere expressie, dan worden alleen de regels die matchen gedaan
- twee van deze gescheiden door komma's, dan worden de regels van de eerste selectie t/m die van de tweede selectie gedaan
- tussen de selectie en de opdracht kan een *!* gezet worden; dan wordt de opdracht juist voor de *niet* geselecteerde regels gedaan

Bij de "t/m" selecties met twee reguliere expressies kunnen er meerdere gebieden zijn die geselecteerd worden. Bijvoorbeeld */BEGIN/,/END/* selecteert vanaf de eerste regel waarin "BEGIN" voorkomt t/m de eerstvolgende waarin "END" voorkomt; daarna wordt weer naar "BEGIN" gezocht. Als in de regel waarin "BEGIN" voorkomt, ook "END" voorkomt, dan wordt deze regel gedaan en daarna weer naar een "BEGIN" gezocht.

Enkele andere nuttige *sed* commando's (ook hier kun je selecties toepassen) :

`d`	(delete) gooi de regel weg
`a\`	voeg de volgende regels toe achter de geselecteerde regel. De in te voegen tekst moet elke regel behalve de laatste eindigen met een \
`i\`	voeg tekst toe vóór de geselecteerde regel (verder als *a*
`c\`	voeg tekst toe in plaats van de geselecteerde regel (verder als *a*), dus in feite een combinatie van *d* en *a* of *i* .
`r` ⟨file⟩	voeg de inhoud van ⟨file⟩ toe na de geselecteerde regel.
`q`	(quit) stop verdere verwerking.

Voorbeelden:

sed '/TROEP/d' gooit alle regels weg waarin de tekst "TROEP" voorkomt.
sed 5q doet hetzelfde als *head -5* (stopt na regel 5).
sed '6,$d' ook maar is minder efficient.

Wanneer meer dan één commando aan *sed* gegeven wordt dan moet elk commando voorafgegaan worden door *-e* òf alle commando's moeten in een file (script) gezet worden en *sed -f script* gegeven worden.

Tenslotte kan nog de *-n* optie meegegeven worden. In dat geval worden regels niet zonder speciale opdracht naar standaard uitvoer geschreven, maar moet de *p* opdracht gegeven worden om een regel naar standaard uitvoer te schrijven. Bij het *s///* commando kan een extra *p* achter de laatste */* gegeven worden, dan wordt de regel uitgevoerd als er inderdaad een wijziging heeft plaatsgevonden. In plaats van een optie *-n* kan ook het *sed* script met een regel *#n* beginnen.

5.7 Comm

comm vergelijkt twee files en geeft aan welke regels in de ene, welke in de andere, en welke in beide voorkomen. De twee files moeten gesorteerd zijn op ASCII volgorde anders raakt *comm* in de war. De uitvoer wordt geleverd in 3 kolommen: de eerste voor regels uit "file1", de tweede die uit "file2" en de derde voor regels uit beide files. Voorbeeld:

file1	file2	comm file1 file2		
aaa	aaa			aaa
abc	cde	abc		
def	def		cde	
klm	pqr			def
uvw	stu	klm		
xyz	xyz		pqr	
	zzz		stu	
		uvw		
				xyz
			zzz	

Als je niet alle drie de kolommen wilt, dan kun je met een selectie uit *-123* aangeven welke kolommen je niet wilt. Bijvoorbeeld *-13* geeft alleen de tweede kolom. Als “file1” de gesorteerde lijst van woorden uit een tekst is (zoals we in 5.4 gemaakt hebben) en “file2” is een gesorteerde lijst van goede woorden dan geeft het commando

```
comm -23 file1 file2
```

dus de foute woorden uit file1.

Het programma *comm* kan als filter gebruikt worden door voor één van de filenamen het teken `-` te gebruiken. Hiervoor wordt dan de standaard invoer genomen.

5.8 Diff

Het commando *diff* kan ook gebruikt worden om files te vergelijken. Het is geen filter, maar omdat de functionaliteit vergelijkbaar is met *comm* wordt het hier ook behandeld. *Diff* geeft alleen aan waarin de files verschillen, en niet waarin ze gelijk zijn. Dit kan op een aantal verschillende manieren weergegeven worden. Bijvoorbeeld het commando *diff file1 file2* met de files uit de vorige sectie geeft het volgende:

```
2c2
< abc
---
> cde
4,5c4,5
< klm
< uvw
---
> pqr
> stu
6a7
> zzz
```

Dit geeft aan welke operaties nodig zijn om op *file1* toe te passen zodat *file2* ontstaat. De operaties zijn vergelijkbaar met die van *sed*, maar ze worden gevolgd door regelnummers

uit *file2* zodat de operatie ook andersom uitgevoerd kan worden. Regels uit *file1* worden voorafgegaan door `<` , regels uit *file2* door `>` . Er staat hier dus o.a. dat regel 2 uit *file1* vervangen wordt door regel 2 uit *file2*, regel 4,5 uit *file1* door regel 4,5 uit *file2* en dat achter regel 6 uit *file1* en regel toegevoegd wordt (regel 7 uit *file2*). Als we de filenamen verwisseld zouden hebben zou er gestaan hebben dat regel 7 uit *file2* weggelaten zou zijn en dat dit overeenkomt met regelnummer 6 uit *file1* (`7d6`). De optie `-c` kan gebruikt worden om een z.g. *context diff* te genereren waarbij ook een aantal van de omliggende regels meegegeven worden. Dit is handig als de uitvoer van *diff* gebruikt wordt om een file aan te passen waarbij intussen wat wijzigingen zijn aangebracht. Omdat de regelnummers dan veranderd kunnen zijn is de normale uitvoer niet altijd bruikbaar (toevoegingen kunnen dan op de verkeerde plaats komen). Hieronder volgt een voorbeeld van de context diff. Het teken `!` geeft een gewijzigde regel aan, `+` een toegevoegde en `-` een weggelaten regel. Moderne *diff* programma's hebben een compactere vorm hiervan, (unified) die met de *-u* opgevraagd kan worden. Zie de rechterkolom.

```
$ diff -c file1 file2                  |   $ diff -u file1 file2
*** file1 Wed Mar  8 14:13:58 1995 |   --- file1 Wed Mar  8 14:13:58 1995
--- file2 Wed Mar  8 14:14:27 1995 |   +++ file2 Wed Mar  8 14:14:27 1995
***************                        |   @@ -1,6 +1,7 @@
*** 1,6 ****                           |    aaa
  aaa                                  |   -abc
! abc                                  |   +cde
  def                                  |    def
! klm                                  |   -klm
! uvw                                  |   -uvw
  xyz                                  |   +pqr
--- 1,7 ----                           |   +stu
  aaa                                  |    xyz
! cde                                  |   +zzz
  def
! pqr
! stu
  xyz
+ zzz
```

Het programma *patch* kan gebruikt worden om vanuit de *diff* uitvoer en één van de files de andere te reconstrueren.

Hoofdstuk 6

Awk

Het programma *awk* is genoemd naar de auteurs ervan: Aho, Weinberger en Kernighan. Het is eigenlijk een mini-programmeertaal waarmee filters gemaakt kunnen worden. De programmeertaal lijkt veel op de taal C, maar er zijn betere voorzieningen om met tekst te manipuleren. Bovendien is de taal heel geschikt voor het maken van filters. Er zijn verschillende versies van awk in omloop, de versie die hier beschreven wordt heet soms *nawk* (*new awk*). Er is ook een versie *GNU awk*, die soms *gawk* heet en die minstens kan wat in dit hoofdstuk beschreven staat. Awk kan beschouwd worden als een combinatie van *sed*, *expr*, *egrep* en nog wat meer.

Omdat *awk* programmaatjes (meestal *scripts* genoemd) vaak wat groter worden stoppen we ze meestal in een aparte file en roepen ze dan aan met

```
awk -f ⟨scriptfile⟩ ⟨invoerfiles⟩ ...
```

maar het is ook mogelijk het script zelf als eerste argument aan *awk* mee te geven. Heel vaak zal het dan gequote worden om speciale tekens tegen de shell te beschermen. Als we ' ' gebruiken is het zelfs mogelijk om scripts van meer dan één regel direct als argument mee te geven.

Een derde mogelijkheid is om het *awk*-script zelf executeerbaar te maken door het de x-permissie te geven en als eerste regel op te geven:

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

Een *awk* script bevat een aantal statements die voor elke regel of voor een selectie van regels uitgevoerd worden. De statements zien eruit als C statements met accolades eromheen. De selecties zijn vergelijkbaar met die van *sed* (5.6). De preciese regels voor de selecties en statements volgen later in dit hoofdstuk (6.1, 6.3).

Behalve de statements die per invoerregel gedaan worden, is het mogelijk om een “BEGIN” en een “END” sectie op te geven. De statements hiervan worden resp. aan het begin van het script (dus vóór de eerste regel) en aan het eind van het script (na de laatste regel) uitgevoerd. De “BEGIN” sectie is nuttig om bijv. variabelen een initiële waarde te geven; de “END” sectie om verzamelde resultaten af te drukken. Een compleet *awk* script ziet er dan dus a.v. uit:

```
BEGIN { ⟨statements⟩ }
⟨patroon⟩ { ⟨statements⟩ }
END { ⟨statements⟩ }
```

Ze hoeven niet in deze volgorde voor te komen en ieder deel mag weggelaten worden. Het ⟨patroon⟩ { ⟨statements⟩ } deel mag meer keer voorkomen. Wanneer *awk* een script uitvoert gebeurt er dus het volgende:

1. Als er een “BEGIN” sectie is wordt deze uitgevoerd.
2. De invoer wordt gelezen. Voor iedere regel worden alle ⟨patroon⟩ delen één voor één uitgevoerd. Als de regel door het patroon geselecteerd wordt, dan worden de bijbehorende ⟨statements⟩ uitgevoerd.
3. Na de laatste regel wordt de “END” sectie uitgevoerd als die aanwezig is.

6.1 Statements

Statements in *awk* zijn voor het grootste deel te vergelijken met die in C. De basis statements zijn de assignment (⟨variabele⟩ = ⟨expressie⟩) en de functie aanroep (⟨func⟩*(*⟨params⟩*)*). In feite zijn dit expressies die als statement gebruikt worden, dit kan met elke expressie. Extra zijn de *print* en *printf* statements en nog een paar die in tabel 6.1 genoemd worden. Deze statements kunnen gecombineerd worden tot samengestelde statements zoals te zien is in tabel 6.2.

⟨expressie⟩	bijvoorbeeld assignment of functieaanroep
`delete` ⟨array⟩`[`⟨index⟩`]`	zie 6.7
`break`	beëindig een `while` of `for`
`continue`	begin de volgende slag van een `while` of `for`
`print` ⟨expressie⟩ `...` `printf format` ⟨expressie⟩ `...`	uitvoer nette uitvoer
`next`	begin aan de volgende regel
`exit` ⟨expressie⟩	stop het *awk* script
`return` ⟨expressie⟩	keer terug van een functie

Tabel 6.1: Simpele awk statements

De statement *for (*⟨E1⟩*;*⟨E2⟩*;*⟨E3⟩*)* ⟨S⟩ is hetzelfde als de serie

```
⟨E1⟩;
while (⟨E2⟩) { ⟨S⟩; ⟨E3⟩ }
```

Statements worden gescheiden door *;* , maar de *;* mag weggelaten worden aan het eind van een regel. Je moet dus uitkijken dat het einde van een regel niet als einde van de statements opgevat wordt als je een statement over twee regels wilt verdelen.

`if ( ⟨expressie⟩ ) ⟨statement⟩`	`if` zonder `else` tak
`if ( ⟨expressie⟩ ) ⟨statement⟩ else ⟨statement⟩`	`if` met `else` tak
`while ( ⟨expressie⟩ ) ⟨statement⟩`	`while` loop (test vooraf)
`do ⟨statement⟩ while ( ⟨expressie⟩ )`	`while` loop (test achteraf)
`for ( ⟨expr1⟩ ; ⟨expr2⟩ ; ⟨expr3⟩ ) ⟨statement⟩`	`for` loop: ⟨expr1⟩ wordt vóór de loop gedaan, ⟨expr2⟩ is de loop test (als deze true is dan doen we de loopdoorgang), ⟨expr3⟩ wordt tussen de loopdoorgangen gedaan.
`for ( ⟨var⟩ in ⟨array⟩ ) ⟨statement⟩`	zie 6.7
`{ ⟨statement⟩ ...}`	samengestelde statement

Tabel 6.2: Samengestelde awk statements

6.2 Expressies

Awk heeft variabelen, maar ze kunnen niet gedeclareerd worden. Een variabele ontstaat de eerste keer dat hij gebruikt wordt. Een variabele kan een getal bevatten of tekst, en *awk* zet het één in het ander om als dat nodig is, dus als bijv. *x = "123abc"* dan levert *x + 1* op: 124. Bij het ontstaan van een variabele krijgt deze de waarde 0 of de lege tekst (*""*) als er geen waarde aan toegekend wordt.

Er zijn een stel standaard variabelen die *awk* zelf voor je bijhoudt, bijvoorbeeld *NR* is het regelnummer. Zie tabel 6.4.

Expressies in *awk* zijn ook te vergelijken met die in C. De gebruikelijke rekenkundige operaties, vergelijkingen, en assignments zijn aanwezig. Een paar operaties die C niet heeft zijn machtsverheffen (^ of ****) en de z.g. concatenatie operator. Dit is een operator die twee teksten aan elkaar plakt. Het bijzondere van deze operator is dat hij onzichtbaar is, je krijgt dus de concatenatie van twee dingen door ze na elkaar te schrijven. Als bijvoorbeeld *x = "fiets"* en *y = "wiel"* dan levert *x y* de tekst "*fietswiel*" op.

Verder zijn er de *match* operaties die vergelijkbaar zijn met die van *expr* (zie 4.1.10). De match operator ~ heeft als eerste argument een expressie en als tweede een reguliere expressie tussen *//*. De operator levert true op als het eerste argument (de tekst) de reguliere expressie matcht. De operator *!~* is het omgekeerde (negatie) ervan en geeft dus aan of het *niet* matcht. Voorbeelden:

> *"aap-noot-mies" ~ /n.*t/* is *true*
> *"aap-noot-mies" !~ /[A-Z]/* is *true*

Vergelijkingen in *awk* kunnen zowel voor tekst als voor getallen gebruikt worden. Welke van de twee gebruikt wordt hangt af van de inhoud van de twee dingen die vergeleken worden: Als ze allebei een getal bevatten dan wordt getal-vergelijking gebruikt, anders tekst-vergelijking. Voor teksten wordt de ASCII volgorde gebruikt waarbij net als in een woordenboek "hand" voor "handvat" komt. Je moet dus opletten:

<table>
<tr><td>++ --</td><td>V++ en ++V verhogen de variabele V met 1. Als dit binnen een andere expressie of statement staat dan wordt bij V++ de oude waarde gebruikt en bij ++V de nieuwe. V-- en --V verlagen V met 1 op dezelfde manier.</td></tr>
<tr><td>^ **</td><td>machtsverheffen</td></tr>
<tr><td>+ - !</td><td>dit zijn de unaire operaties + - en not, bijv. - x.</td></tr>
<tr><td>* / %</td><td>vermenigvuldigen, delen en modulo (rest bij deling)</td></tr>
<tr><td>+ -</td><td>optellen en aftrekken (x + y)</td></tr>
<tr><td>⟨concatenatie⟩</td><td>zie de tekst</td></tr>
<tr><td>< <= == != >= ></td><td>vergelijkingen</td></tr>
<tr><td>~ !~</td><td>reguliere expressie match</td></tr>
<tr><td>⟨index⟩ in ⟨array⟩</td><td>(zie 6.7)</td></tr>
<tr><td>&&</td><td>de logische en (and)</td></tr>
<tr><td>||</td><td>de logische of (or)</td></tr>
<tr><td>? :</td><td>b ? x : y betekent zoveel als if b then x else y</td></tr>
<tr><td>= += -= *= /= %= ^= **=</td><td>operatie met assignment: x += y is hetzelfde als x = x + y, analoog voor de andere.</td></tr>
</table>

Tabel 6.3: Awk expressies. Operaties in hogere vakje gaan vóór die in lagere vakjes

naam	soort	betekenis
ARGC	getal	aantal argumenten van *awk* (6.12)
ARGV	array	de argumenten van *awk* (6.12)
CONVFMT	tekst	Het format om getallen om te zetten naar strings ("%.6g")
ENVIRON	array	environment variabelen (6.12)
FILENAME	tekst	de naam van de invoerfile die verwerkt wordt
FNR	getal	het regelnummer binnen de huidige file
FS	teken	veldscheider (6.5)
NF	getal	aantal velden in de regel (6.5)
NR	getal	regelnummer (doorlopend)
OFMT	tekst	output format waarmee getallen geprint worden
OFS	tekst	output veld scheider (meestal spatie)
ORS	tekst	output record scheider (meestal newline)
RS	tekst	invoer record scheider (meestal newline)
SUBSEP	teken	scheider voor meerdimensionale indexen (6.7)
$0	tekst	de invoer regel
$1 $2, ...	tekst	de velden van de invoer regel (6.5)

Tabel 6.4: Standaard variabelen in awk

2 < 10 is *true*
"2" < "10" is *false*
2 < x is *true* als *x = 10* maar *false* als *x = "10"*

Zo nodig kun je een waarde dwingen om als getal te functioneren door er 0 bij op te tellen en als tekst door de lege tekst eraan te plakken (bijvoorbeeld met *x ""*)

6.3 Patronen

Patronen (voor de selectie van invoerregels) kunnen zijn:

- reguliere expressies tussen *//*. De reguliere expressies zijn dezelfde als in *egrep* (5.1.1). Omdat een */* de reguliere expressie afsluit, kun je een */* in een reguliere expressie opgeven als *\/*. Een reguliere expressie selecteert regels die matchen. Bijvoorbeeld */stop/* selecteert alle regels waar de tekst “stop” in voorkomt.
- Relaties met *== != < > <= >=* of met de match operators *~ !~*. Hiermee is het mogelijk regels te selecteren op basis van de waarde van variabelen. Bijvoorbeeld *NR <= 5* selecteert de eerste 5 regels van de invoer.
- Patronen kunnen gecombineerd worden met de logische operatoren *! && ||* en haakjes. Het patroon */aap/ && NR > 5* selecteert de regels waar de tekst “aap” in voorkomt na de vijfde regel.
- Twee patronen kunnen met een komma gecombineerd worden tot een “t/m” patroon, net als bij sed (5.6): *NR == 2 , /STOP/* selecteert de regels vanaf regel 2 t/m de eerstvolgende regel waarin het woord “STOP” voorkomt. Als de slotregel is gevonden dan wordt weer naar een regel gezocht die het eerste patroon selecteert. In het vorige voorbeeld zal dat dus nooit gevonden worden, maar als het patroon is */START/,/STOP/* dan wordt eerst gezocht naar een regel die de tekst “START” bevat, dan worden de regels geselecteerd t/m de eerstvolgende die “STOP” bevat, daarna wordt weer naar een “START” gezocht, enz. Als de “START” regel óók de tekst “STOP” bevat dan wordt alleen deze regel geselecteerd en begint het zoeken naar “START” direct op de volgende regel.

6.4 Voorbeelden

/interessant/ { *print $0* }

geeft alleen de regels waar het woord “interessant” in voorkomt. Dit doet dus hetzelfde als de opdracht *egrep interessant*. In dit geval kun je het ook korter opschrijven: de statement *print* betekent n.l. hetzelfde als *print $0* en het is zelfs zo dat de hele statement weggelaten mag worden, omdat als alleen een patroon opgegeven wordt, de statement *print $0* wordt genomen. Als het patroon weggelaten wordt dan wordt elke regel geselecteerd. Beide weglaten kan natuurlijk niet.

Alles bij elkaar genomen is de opdracht *awk '/*⟨regexp⟩*/'* dus hetzelfde als *egrep '*⟨regexp⟩*'*.

```
length > 72
```

geeft de regels die langer zijn dan 72 tekens.

```
{ print NR ":" , $0 }
```

print de invoer met regelnummers ervoor. Let op dat `print x, y` niet hetzelfde is als `print x y`. In het eerste geval worden x en y afgedrukt met een spatie ertussen, in het tweede geval worden x en y aan elkaar geplakt en het resultaat afgedrukt, dus zonder spatie (in feite wordt in het eerste geval tussen x en y de waarde van de variabele `OFS` gezet).

```
{ print }
NR%60 == 0 {
    print ""
    print "Pagina", NR/60
    print ""
}
```

Geeft na iedere 60 regels een pagina nummer.

Opgave 6.1 Om dit mooi te maken zou je voor de laatste pagina nog iets speciaals moeten doen als die niet toevallig vol is. Doe dit.

Nog wat voorbeelden met getallen. We hebben een file met getallen, één per regel. We willen het gemiddelde, het minimum en het maximum van de getallen hebben. Hiervoor moeten we de getallen optellen en variabelen `min` en `max` bijhouden. Het aantal kunnen we vinden in de variabele `NR`. De variabelen `min` en `max` moeten van te voren op geschikte waarden gezet worden. We kiezen een grote waarde voor `min` en een kleine (negatief) voor `max`. We moeten dan wel weten dat de getallen tussen deze waarden in liggen.

```
BEGIN { min =  9999999
        max = -9999999
        sum = 0}
{   sum += $0
    if ($0 < min) min = $0
    if ($0 > max) max = $0
}
END {   print "minimum =", min, "maximum =", max
        print "gemiddelde =", sum/NR}
```

Dit voorbeeld is niet erg “robuust”, zo zal bijvoorbeeld een lege regel als het getal 0 geteld worden. We kunnen dit verbeteren door een patroon op te nemen dat test of er echt wel een getal staat. De initiële waarden voor `min` en `max` zijn ook niet erg bevredigend, al kun je natturlijk nog wel grotere getallen invoeren. Een verbetering zou zijn om de eerste regel anders te behandelen:

```
NR==1 { min = max = $0}
```

(De initialisatie van de variabele *sum* is eigenlijk overbodig omdat die toch automatisch op 0 begint.) Een andere truc is om *min* en *max* op een *tekst* te initialiseren die groter resp. kleiner dan alle getallen is dus bijvoorbeeld: `min = "z"; max = " "`.

6.5 Velden

In de vorige voorbeelden hebben we steeds de invoerregel als één geheel genomen. Maar *awk* kan ook de regel opsplitsen in *velden*. Als we niets speciaals doen dan wordt de regel opgesplitst bij ieder stukje witruimte (spaties en tabs). De stukken tussen de witruimte gelden dan als velden. Eventuele witruimte aan het begin of eind van de regel wordt niet meegeteld. De velden worden aangegeven met `$1, $2, ...`. Het aantal velden wordt gezet in de variabele *NF*. De volgende regel wordt dus zo ingedeeld:

```
Als    a+b<c,    of  er  is  een  "fout"  (of  een  vergissing),
$1      $2       $3  $4  $5  $6     $7     $8   $9      $10
```

We mogen ook zeggen `$`⟨x⟩ waarbij ⟨x⟩ een variabele of expressie is die het veldnummer bevat.

We kunnen ook een eigen veldscheider definiëren door deze in de variabele *FS* te zetten of via de optie *-F* aan *awk* mee te geven. In dat geval wordt dat teken als veldscheider gebruikt maar dan geldt ieder voorkomen ervan als een scheider, dus ook aan het begin en eind van de regel en twee opvolgende (waar dan lege velden uit voorkomen). De veldscheider mag zelfs een reguliere expressie zijn en dan wordt elk stuk dat match als veldscheider genomen. De veldscheider is geen deel van de velden.

We gaan nu ons getallenvoorbeeld uitbreiden doordat er meer dan één getal op een regel mag staan gescheiden door spaties. Elk veld is dan dus een getal. We moeten nu zelf het aantal getallen bijhouden en voor elke regel moeten we alle getallen behandelen met een *for* statement:

```
BEGIN { min =  "z"
        max = " "}
{   for (i=1; i<=NF; i++)
    {   sum += $i
        if ($i < min) min = $i
        if ($i > max) max = $i
    }
    aantal += NF
}
END {   print "minimum =", min, "maximum =", max
        print "gemiddelde =", sum/aantal}
```

Soms is het niet geschikt om spaties als veldscheider te laten optreden. Bijvoorbeeld als we een adressenbestand hebben dan kunnen er spaties staan in de namen en adressen. We moeten dan een ander teken als scheider nemen. In Unix wordt dit vaak gedaan met een *:* wat

als enige nadeel heeft dat dit teken niet in een veld gebruikt kan worden. Als ons bestand bijvoorbeeld de velden naam, adres, postcode, woonplaats en telefoonnummer heeft, dan zal een willekeurige regel (record) er als volgt uitzien:

```
Jan Jansen:Ina Boudier Bakkerlaan 2037:3581 XY:Utrecht:030-253199
```

Als we nu alle namen met telefoonnummers willen afdrukken dan kan dit met het script {*print $1, $5*}. Of als we het telefoonnummer van Jan Jansen willen hebben:
/Jan Jansen/{*print $5*}.
Als we niet meer weten hoe "Jansen" gespeld moest worden, kunnen we de adresgegevens van de Jans[s]ens opvragen met:

```
/Jan[sz]+en/{print $1, $2, $3, $4}
```

Let op dat we in beide gevallen ook mensen krijgen die in een "Jansenstraat" wonen. We kunnen dat voorkomen door als selectiepatroon *$1 == "Jan Jansen"* resp. *$1 ~ /Jan[sz]+en/* te nemen.

6.6 Printf

De lijsten die we uit de opdrachten in de vorige sectie krijgen zijn niet zo bevredigend omdat de straten, telefoonnummers e.d. niet netjes onder elkaar staan. We kunnen dit verbeteren door de *print* statements te vervangen door *printf* (print formatted).

De *printf* statement heeft als eerste parameter een besturingsstring (*format)*). Dit is een tekst waar met *%* aangegeven wordt waar de andere parameters ingevoegd worden. De *%* wordt gevolgd door informatie over hoe de layout van de parameter moet zijn. Er zijn de volgende onderdelen waarvan alleen de laatste aanwezig hoeft te zijn:

1. een `-` teken als de uitvoer links aangeschoven moet worden, als dit niet gegeven wordt dan wordt rechts aangeschoven.

2. een getal (de breedte) dat aangeeft hoeveel plaatsen voor de uitvoer gereserveerd wordt. Als dit niet aangegeven wordt dan wordt de minimaal benodigde ruimte gebruikt. Als de parameter meer ruimte nodig heeft dan dit getal aangeeft dan wordt er ook meer gebruikt (er gaat dan dus geen informatie verloren maar de kolommen kunnen in de war raken). Als er meer ruimte opgegeven wordt dan nodig is dan wordt de rest opgevuld met spaties. Alleen als de breedte met een 0 begint wordt bij getallen die rechts aangeschoven worden aangevuld met nullen (aan de linkerkant natuurlijk).

3. een `.` met nog een getal: de precisie. Voor getallen betekent dit het aantal cijfers dat achter de punt afgedrukt wordt, voor teksten betekent dit het aantal tekens dat van de tekst gebruikt wordt.

4. een letter die aangeeft hoe de parameter afgedrukt moet worden, bijvoorbeeld als geheel getal of als tekst. Zie tabel 6.5.

c	ASCII teken, gebruik de parameter (getal) als code van het teken
d	geheel getal
e	getal in wetenschappelijke notatie (met e exponent)
f	getal met decimale punt
g	getal met notatie e of f die het beste past
o	octaal getal
x	hexadecimaal getal met kleine letters a t/m f
X	hexadecimaal getal met hoofdletters A t/m F
s	string (tekst)

Tabel 6.5: Printf besturingstekens

<table>
<tr><th>fmt</th><th>x</th><th>printf(fmt, x)</th></tr>
<tr><td>|%c|</td><td>65</td><td>|A|</td></tr>
<tr><td>|%d|</td><td>65.3</td><td>|65|</td></tr>
<tr><td>|%5d|</td><td>65.3</td><td>| 65|</td></tr>
<tr><td>|%05d|</td><td>65.3</td><td>|00065|</td></tr>
<tr><td>|%-5d|</td><td>65.3</td><td>|65 |</td></tr>
<tr><td>|%e|</td><td>65.3</td><td>|6.530000e+01|</td></tr>
<tr><td>|%f|</td><td>65.3</td><td>|65.300000|</td></tr>
<tr><td>|%7.2f|</td><td>65.3</td><td>| 65.30|</td></tr>
<tr><td>|%g|</td><td>65.3</td><td>|65.3|</td></tr>
<tr><td>|%o|</td><td>65</td><td>|101|</td></tr>
<tr><td>|%x|</td><td>127</td><td>|7f|</td></tr>
<tr><td>|%X|</td><td>255</td><td>|7F|</td></tr>
<tr><td>|%s|</td><td>voorbeeld</td><td>|voorbeeld|</td></tr>
<tr><td>|%10s|</td><td>voorbeeld</td><td>| voorbeeld|</td></tr>
<tr><td>|%-10s|</td><td>voorbeeld</td><td>|voorbeeld |</td></tr>
<tr><td>|%10.3s|</td><td>voorbeeld</td><td>| voo|</td></tr>
<tr><td>|%-10.3s|</td><td>voorbeeld</td><td>|voo |</td></tr>
<tr><td>|%.3s|</td><td>voorbeeld</td><td>|voo|</td></tr>
</table>

Tabel 6.6: Voorbeelden van printf

Als een echt *%* teken moet worden afgedrukt dan kan dat door in de besturingsstring er twee op te nemen. Zie tabel 6.6 voor voorbeelden van printf.

In tegenstelling tot *print* geeft *printf* geen newline aan het einde. Als je die wilt moet je hem met *\n* in de besturingsstring opnemen of een aparte *print ""* opdracht geven. We geven nu het naam/telefoonnummer overzicht uit 6.5 met een mooi uitgelijnde tabel. We geven ook nog even een kopje dat natuurlijk op dezelfde manier ingedeeld moet worden en daarom stoppen we de besturingsstring in een variabele:

```
BEGIN { fmt = "%-25s %10s\n"
        printf fmt, "Naam", "tel.nr."
        printf fmt, "----", "-------"}
{ printf fmt, $1, $5 }
```

Nog een voorbeeld: we willen een file met getallen netjes in kolommen afgedrukt hebben; dat kan met het volgende script:

```
{   for (i=1; i<=NF; i++)  printf " %12g", $i
    print ""
}
```

6.7 Arrays

Voor het bij elkaar houden van gegevens heeft *awk* arrays. Elk element van een array, aangegeven met ⟨naam⟩ *[*⟨index⟩*]* gedraagt zich als een variabele. De arraynaam mag hetzelfde zijn als een variabele naam; *awk* weet wanneer de variabele bedoeld wordt en wanneer het array bedoeld wordt. We hoeven niet van tevoren te vertellen hoe groot het array wordt; het is zelfs onmogelijk om dit te doen.

Als we bijvoorbeeld een file met getallen hebben en we willen de getallen in kolommen optellen dan kunnen we een array *sum* nemen dat de kolommen optelt; dat kan met het volgende script. We houden zelf bij hoeveel kolommen er geweest zijn in de variabele *MAXNF*.

```
{ for (i=1; i<=NF; i++) sum[i] += $i
  if (NF > MAXNF) MAXNF = NF
}
END { for (i=1; i<=MAXNF; i++) printf " %12g", sum[i]
      printf "\n"
}
```

De index hoeft niet een getal te zijn maar mag ook een tekst zijn: We kunnen dus een vertaaltabel maken van nederlands naar engels met

```
vert["auto"] = "car"
vert["fiets"] = "bike"
```

Alleen kun je dan niet meer met een gewone *for* statement alle elementen afhandelen. Daarvoor heeft *awk* een speciale *for* constructie

for (⟨var⟩ in ⟨array⟩) ⟨statement⟩

Hier zal de ⟨statement⟩ voor elke voorkomende index één keer uitgevoerd worden waarbij ⟨var⟩ dan de waarde van de index krijgt. Het bijbehorende element kun je krijgen met ⟨array⟩ *[*⟨var⟩*]* . De volgorde waarin je de elementen krijgt is ongedefinieerd. In bovenstaand voorbeeld kunnen we dus de vertaaltabel afdrukken met het volgende script:

```
END { for (woord in vert)
        printf "de vertaling van %s is %s\n", woord, vert[woord]
}
```

Het is ook heel makkelijk om zo'n tabel om te keren, dus een vertaling van engelse naar nederlandse woorden te maken. We vullen dan een ander array *omk* door daarin het engelse woord als index te nemen an het nederlandse als waarde, dus:

```
for ( w in vert ) omk[vert[w]] = w
```

Als we een file hebben met alleen woorden kunnen we ook tellen hoe vaak elk woord voorkomt door een array te nemen met het woord als index en het aantal als waarde:

```
{ for (i=1; i<=NF; i++) aantal[$i]++}
END{ for (w in aantal) printf "%10s %5d\n", w, aantal[w] }
```

De uitvoer hiervan zou nog gesorteerd moeten worden.

Het is mogelijk een bestaand element te verwijderen met de *delete* ⟨array⟩*[*⟨index⟩*]* opdracht. Het is ook mogelijk om te testen of een bepaalde index in het array aanwezig is met de constructie ⟨index⟩ *in* ⟨array⟩. We gebruiken dit in het volgende voorbeeld waarbij we woorden vervangen volgens een vervangingstabel. De tabel wordt eerst ingelezen uit de file "DICT" die als eerste filenaam meegegeven wordt. In deze file heeft elke regel het te vertalen woord in het eerste en de vertaling in het tweede veld. De andere filenamen zijn dan de teksten die "vertaald" worden. Als een woord niet in de tabel voorkomt nemen we het gewoon over. Om te weten of we de vertaaltabel aan het lezen zijn of een tekst file testen we de ingebouwde variabele *FILENAME*.

```
FILENAME == "DICT" { vert[$1]=$2 }
FILENAME != "DICT" {
  for (i=1; i<=NF; i++)
  { if ($i in vert) printf "%s ", vert[$i]
    else printf "%s ", $i
  }
  printf "\n"
}
```

De *if ... else* constructie kan ook vervangen worden door één *printf* met een conditionele expressie:

```
printf "%s ", $i in vert ? vert[$i] : $i
```

6.7.1 Meerdimensionale arrays

Meerdimensionale arrays (arrays met meer dan één index) kunnen ook gebruikt worden d.m.v. de constructie *a[i,j]*. Maar eigenlijk is dit nep, want *awk* gebruikt eigenlijk alleen enkele indexen. De constructie *a[i,j]* wordt gedaan door de indexen (*i* en *j*) aan elkaar te plakken met een speciaal teken ertussen en dit geheel als index te gebruiken. (Op dezelfde manier voor meer dan twee). Je moet dan dus zorgen dat het speciale teken niet in de indexen zelf voorkomt. Dit teken staat in de variabele *SUBSEP* en heeft standaard de waarde van het ASCII teken octaal 034 (ofwel *"\034"*). Dit is een niet zichtbaar teken dus de kans dat het in een tekst voorkomt is erg klein. Je kunt een andere waarde gebruiken door die in *SUBSEP* te zetten.

De bijbehorende test heeft de vorm *if ((i, j) in a)*. In werkelijkheid is dit precies hetzelfde als (het wordt omgezet in) *if (i SUBSEP j in a)*. Maar er is geen bijbehorende *for (i,j) in a*. Als je dit effect wilt bereiken moet je de gewone *for (x in a)* constructie gebruiken en dan de *x* opsplitsen met de *split* functie die in 6.8 besproken wordt:

```
for (x in a) {
  split(x, t, SUBSEP)
  i = t[1]
  j = t[2]
  ...
}
```

6.8 Operaties op teksten

awk heeft diverse functies voor het manipuleren van teksten. We hebben al gezien dat de concatenatie operator teksten aan elkaar kan plakken, en dat we teksten als index in een array kunnen gebruiken. Hier volgen een aantal functies waarmee we teksten kunnen analyseren of veranderen:

`length(`⟨**t**⟩`)` De lengte (het aantal tekens) van de tekst ⟨t⟩

`index(`⟨**s**⟩`,` ⟨**t**⟩`)` De plaats waar de tekst ⟨t⟩ in de tekst ⟨s⟩ voor de eerste keer voorkomt, of 0 als ⟨t⟩ niet voorkomt. Bijvoorbeeld:

```
index("abracadabra", "bra") --> 2
```

`match(`⟨**t**⟩`,` ⟨**re**⟩`)` De plaats waar de reguliere expressie ⟨re⟩ voor het eerst matcht in de tekst ⟨t⟩ of 0 als er geen match is. Als er een match is dan worden er nog twee ingebouwde variabelen gezet: RSTART wordt de positie van de match (dus hetzelfde als de functiewaarde) en RLENGTH wordt de lengte van de gematchte tekst. Als er geen match is dan worden deze 0 en -1. Voorbeeld:

```
match("hottentottententententoonstelling", /(tenten)+/) --> 10
     RSTART = 10   RLENGTH = 12
```

`substr(⟨t⟩, ⟨k⟩, ⟨l⟩)` Levert een deeltekst op uit ⟨t⟩ die op teken nr. ⟨k⟩ begint en ⟨l⟩ tekens lang is. Als ⟨l⟩ niet opgegeven wordt dan loopt de deeltekst tot het einde van ⟨t⟩. Voorbeeld:

```
substr("abracadabra", 5, 3) --> "cad"
```

`toupper(⟨t⟩)` levert de hoofdletter versie van ⟨t⟩ op, m.a.w. een copie van ⟨t⟩ waar elke kleine letter door de hoofdletter vervangen is (⟨t⟩ zelf blijft onveranderd).

`tolower(⟨t⟩)` levert de kleine letter versie van ⟨t⟩ op.

`sprintf(⟨fmt⟩, ⟨e1⟩ ...)` levert als functie waarde hetzelfde op als de *printf* opdracht uitgeprint zou hebben. Deze functie print dus niets, maar kan gebruikt worden om bijv. te testen of een bepaalde uitvoer nog op de regel past en zo nodig op een nieuwe regel te beginnen:

```
uitvoer = sprintf("(%d)...", $i)
if (length(uitvoer) > ...) printf "\n    "
...
```

`split(⟨t⟩, ⟨a⟩, ⟨c⟩)` Deze operatie splitst de tekst ⟨t⟩ op de plaatsen die aangegeven worden door ⟨c⟩. Elk onderdeel wordt in een element van het array ⟨a⟩ gezet, te beginnen op ⟨a⟩[1]. Het splitskenmerk ⟨c⟩ kan een teken zijn, of een reguliere expressie. Als ⟨c⟩ niet meegegeven wordt dan wordt FS genomen. De splits operatie is precies dezelfde operatie als het splitsen van *$0* in de velden *$1, ...*. Er zijn heel veel nuttige voorbeelden van deze functie. Zo kunnen we bijvoorbeeld een Unix filenaam in onderdelen opsplisten door als splitsteken de "/" te nemen:

```
split("/users/piet/doc/st", a, "/")
```

levert op (let op het eerste lege deel):

```
a[1]="" a[2]="users" a[3]="piet" a[4]="doc" a[5]="st"
```

`sub(⟨re⟩, ⟨verv⟩, ⟨t⟩)` Dit is de *substitutie* operatie die een bestaande tekst wijzigt (in tegenstelling tot de vorige functies). *sub* zoekt in de tekst ⟨t⟩ de eerste match van de reguliere expressie ⟨re⟩, en vervangt die door de tekst ⟨verv⟩. Als er geen match is dan blijft ⟨t⟩ ongewijzigd. ⟨t⟩ moet een variabele, een array element of een veld zijn. De functiewaarde van *sub* is het aantal vervangingen, dus 0 of 1. Deze operatie is vergelijkbaar met de *s///* operatie van *sed*. Voorbeeld:

```
x = "hottentottenten"
sub(/tent/, "huis", x) --> x = "hothuisottententen"
```

In de vervangingstekst kan het gematchte deel ingevuld worden door daar het `&` teken op te nemen. Als we in een tekst bijvoorbeeld alle lidwoorden tussen `*` willen zetten dan kan dat met:

```
gsub(/(de|het|een)/, "*&*")
```

Als je een echt `&` teken wilt opnemen in de vervangingstekst dan moet dat met *"\\&"*.

Als ⟨t⟩ niet opgegeven wordt dan wordt *$0* genomen. Let op: wijzigingen op *$0* hebben ook tot gevolg dat de velden *$1* etc. aangepast worden, en omgekeerd hebben wijzigingen aan de velden tot gevolg dat *$0* aangepast wordt.

`gsub(`⟨**re**⟩`,` ⟨**verv**⟩`,` ⟨**t**⟩`)` Dit is de *globale* versie van *sub*. Het vervangt *iedere* match van ⟨re⟩ en levert het aantal vervangingen op, in het volgende voorbeeld is de functiewaarde dus 2:

```
x = "hottentottententen"
gsub(/tent/, "huis", x) --> x = "hothuisothuisenten"
```

6.9 Wiskundige functies

Voor rekenwerk heeft *awk* de standaard wiskundige functies *sin*, *cos*, *atan2* [1]. De goniometrische functies werken met radialen. *exp* (e-macht), *log* (natuurlijke logaritme), *sqrt* (wortel) en *int* dat het gehele deel van een floating point getal oplevert. *rand()* geeft je een willekeurig getal tussen 0 en 1 (nooit 0 en nooit 1). *srand(x)* definieert een startpunt voor *rand*.

6.10 Eigen functies

Eigen functies kunnen gedefinieerd worden met:

```
⟨functienaam⟩(⟨p1⟩, ⟨p2⟩, ...)
{ ⟨body⟩ }
```

⟨p1⟩, ⟨p2⟩, . . . zijn de namen van de parameters. Als er minder argumenten meegegeven worden bij de aanroep dan er gedefinieerd zijn bij de definitie dan gelden de overige als locale variabelen. Bij de aanroep ⟨functienaam⟩*(*⟨a1⟩*, ...)* moet erop gelet worden dat er geen spatie mag staan tussen de ⟨functienaam⟩ en het openingshaakje, anders denkt *awk* dat concatenatie bedoeld is en wordt ⟨functienaam⟩ als variabele genomen!!

6.11 Output redirection in awk

print en *printf* schrijven normaal op standaard uitvoer. Ze kunnen ook naar een andere file schrijven door net als bij de shell de constructie `>` ⟨file⟩ of `>>` ⟨file⟩ te gebruiken. Hierbij moet ⟨file⟩ een variabele of expressie zijn die een filenaam oplevert. Dit mag een constante zijn bijvoorbeeld *"temp.out"* maar ook een in elkaar gezette filenaam als *mijnfile ".out"*. *awk* onthoudt welke files geopend zijn, en alleen de eerste keer dat je de constructie `>` ⟨file⟩

[1] *atan2(y, x)* levert de arctangens van y/x waarbij met beide tekens rekening gehouden wordt, en die bovendien goed werkt als x=0

gebruikt wordt de file schoongemaakt. Dit in tegenstelling tot de shell waar dit iedere keer gebeurt. Je kunt een file sluiten met de opdracht *close* ⟨file⟩. Hierna geldt weer dat de file schoongemaakt wordt bij de eerste > ⟨file⟩.

Als je deze constructies gebruikt dan werkt *awk* natuurlijk niet meer als filter. Maar dit kan handig zijn om een invoer in een paar afzonderlijke delen op te splitsen. Het aantal files dat *awk* open mag hebben is echter niet zo groot (een stuk of 10).

Het is ook mogelijk om de *print* en *printf* opdrachten naar een pipe te laten schrijven met de *print ...* | ⟨commando⟩ constructie. Hier geldt dat de eerste keer dat je een bepaald ⟨commando⟩ gebruikt het opgestart wordt. Wil je tussentijds aangeven dat alle invoer voor ⟨commando⟩ er is, dan kun je ook de opdracht *close(*⟨commando⟩*)* geven.

awk heeft geen standaard voorzieningen voor het schrijven naar standaard error. Sommige versies van *awk* kunnen dit met *print > /dev/stderr"*. Als dit niet het geval is dan kan de volgende afschuwelijke constructie gebruikt worden (het is natuurlijk het beste om dat maar in een fuctie te doen):

```
print ... |"cat 1>&2"
```

We geven tenslotte nog een uitgebreid voorbeeld in figuur 6.1 waarin we woorden tellen in

Figuur 6.1 Awk voorbeeld met pipe

```
BEGIN { FS="[^a-zA-Z]+"}
{ for (i=1; i<=NF; i++) aantal[tolower($i)]++ }
END { delete aantal[""]
      for (w in aantal) printf "%-20s %5d\n", w, aantal[w] | "sort"
}
```

een tekstfile waarin ook leestekens kunnen voorkomen. We filteren de woorden eruit door alle niet-letters in de veldscheider op te nemen. We tellen de woorden onafhankelijk van het feit of ze met hoofd- of kleine letters geschreven worden (door *tolower* te gebruiken) en we leveren de uitvoer gesorteerd af (door naar *sort* te pipen. Let erop dat er aan het begin of eind van een regel een leeg veld kan voorkomen. Deze worden geteld in het array element *aantal[""]*. Daarom gooien we dit element in de END sectie eerst weg. Dit is simpeler dan iedere keer testen of het veld leeg is.

6.12 Argumenten van de *awk* aanroep

We hebben al gezien dat *awk* het script (evt. via *-f* ⟨script⟩) en de invoer filenamen als argumenten meekrijgt. We hebben ook de optie *-F* gezien waarmee te veldscheider gezet wordt. In plaats van een filenaam mag ook de constructie ⟨var⟩=⟨waarde⟩ gegeven worden. Als *awk* zo'n argument tegenkomt bij de overgang van één file naar een volgende dan wordt deze assignment uitgevoerd voordat de eerste regel van de volgende file gelezen wordt. Dit kan o.a. gebruikt worden om onderscheid tussen de files te maken als niet van tevoren de filenamen bekend zijn. Het voorbeeld uit 6.7 om te testen of we met de vertaaltabel bezig zijn zou dus ook kunnen met de aanroep

awk -f script DICT intekst=1 tekst en het script:

```
intekst == 0 { vert[$1]=$2 }
intekst != 0 {
  for (i=1; i<=NF; i++)
  { if ($i in vert) printf "%s ", vert[$i]
    else printf "%s ", $i
  }
  printf "\n"
}
```

Er zijn een paar dingen om op te letten. De assignments die voor de eerste filenaam staan worden gedaan als de eerste file geopend wordt, dus ná het BEGIN deel. Als we variabelen willen zetten vóórdat het BEGIN deel uitgevoerd wordt dan kan dat door in de awk aanroep vóór het script te zetten *-v* ⟨var⟩=⟨waarde⟩. En verder geldt dat als er alleen assignments opgeven worden, dan niet automatisch de standaard invoer gelezen wordt maar dan moet dit met de filenaam `-` aangegeven worden.

De opgegeven argumenten van de aanroep van *awk* worden opgenomen in het array *ARGV*. Alleen de argumenten ná het script worden opgenomen, waarbij *ARGV[1]* het eerste is, etc. *ARGV[0]* bevat de naam van het *awk* programma. De variabele *ARGC* bevat het aantal elementen in *ARGV*, dus 1 meer dan de hoogste index.

Tenslotte bevat het array *ENVIRON* de waarden van de environment variabelen, warbij de naam van de variabele als index gebruikt wordt, bijv *ENVIRON["HOME"]*.

Hoofdstuk 7

Versiebeheer

Tijdens het werken aan een groot software project komen we vaak de volgende problemen tegen:

1. De software doorloopt verschillende stadia (versies) en we willen graag bijhouden wanneer door wie, en waarom verschillende wijzigingen aangebracht zijn. Soms is het nodig vorige versies te gebruiken.

2. Aan (delen van) de software wordt door verschillende personen gewerkt en we moeten ervoor zorgen dat ze elkaar niet in de weg lopen, wanneer ze in hetzelfde stuk wijzigingen aanbrengen.

Om deze twee problemen op te lossen zijn er *versie-beheer* pakketten ontwikkeld. De twee meest populaire pakketten op Unix zijn SCCS (Source Code Control System) en RCS (Revision Control System). De functionaliteit van deze pakketten verschilt niet zoveel, al zijn de commando's die gebruikt moeten worden verschillend. Het voordeel van RCS is dat het gratis verkrijgbaar is, iets simpeler is in het gebruik, en ook op andere computers (bijv. PC's) beschikbaar. Daarom zullen we in dit hoofdstuk RCS bespreken. N.B. RCS bestaat uit een aantal verschillende programma's, die we samen als RCS aanduiden.

7.1 Versies

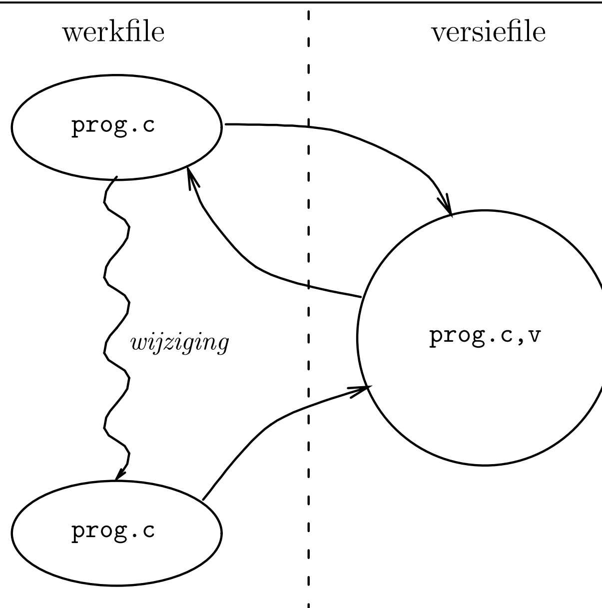
Het eerste probleem dat we op willen lossen is het volgende: We hebben een programma, geschreven in een programmeertaal[1], en op een gegeven moment willen we daar wat wijzigingen in aanbrengen. Helaas blijkt na een tijdje eraan te werken, dat de wijzigingen toch niet zo goed waren, of dat we ze eigenlijk niet eens wilden. We moeten nu onze oorspronkelijke programmatekst terug zien te krijgen. Eén manier is om telkens als we een redelijk stabiele versie hebben, hiervan maar een copie te maken en die ergens te bewaren. Sommige editors, zoals emacs doen dit al voor je. Het nadeel hiervan is, dat de gezamenlijke copiën nogal wat ruimte gaan innemen, terwijl er heel veel hetzelfde in staat. En bovendien moet je nog een

[1] RCS wordt meestal voor het beheren van programmacode gebruikt, maar het kan net zo goed gebruikt worden voor tekstbestanden e.d.

aparte administratie bijhouden waarin staat van elke copie wanneer en waarom deze gemaakt is.

RCS lost dit probleem op door van elke file die je onder RCS beheer zet een z.g. *versiefile* bij te houden met bovengenoemde administratie erin. Voor het editten, compileren e.d. gebruik je dan een *werkfile*. De werkfile is alleen nodig als je ermee bezig bent. Als je een tijdje stopt met het project kan de werkfile verwijderd worden, omdat de versiefile alle informatie bevat die nodig is om de werkfile te reconstrueren. Van tijd tot tijd – meestal wanneer een stuk werk af is – wordt de inhoud van de werkfile inde versiefile gestopt; dit heet *check-in*. Als we de werkfile uit de versiefile willen halen, bijv. omdat we de werkfile weggegooid hebben of omdat we een oude versie nodig hebben, dan heet dat *check-out*. Voor deze twee operaties heeft RCS de programma's `ci` resp. `co`. Zie figuur 7.1.

Figuur 7.1 RCS werkwijze algemeen

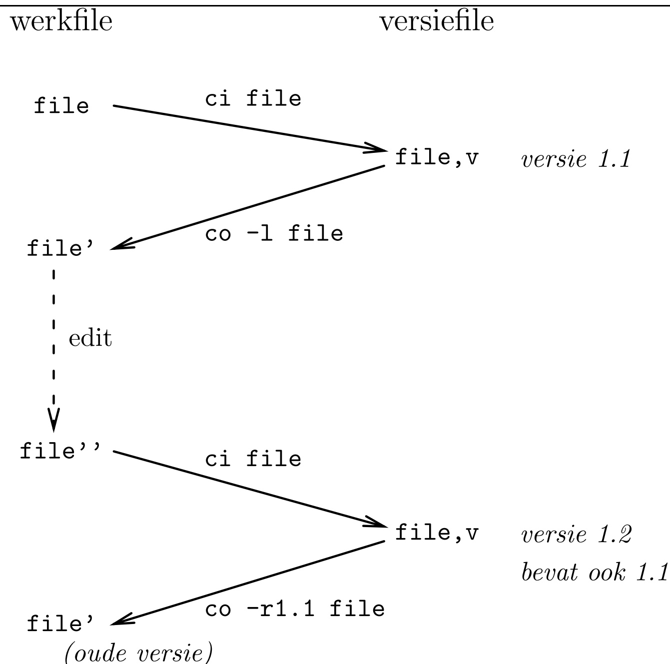


Voor RCS maakt het verschil uit of we een checkout doen met alleen het doel de werkfile te gebruiken, of met het doel er wijzigingen op aan te brengen. In het laatste geval moet nl. een bescherming aangebracht worden tegen wijzigingen die anderen evt. zouden kunnen aanbrengen. RCS (beter gezegd het programma `co`) brengt dan een *lock* aan op de file. Hiervoor gebruiken we de optie `-l` van `co`. Waneer iemand anders (of wijzelf) ook een `co -l` doen op een al gelockte file, dan krijgen we een foutmelding van RCS. Het is mogelijk om toch door te zetten, maar dan warschuwt RCS ons en de andere partij dat er mogelijkerwijs problemen kunnen ontstaan. De inhoud van de werkfile is in beide gevallen hetzelfde, maar wanneer we geen lock gebruiken maakt `co` de werkfile aan met alleen lees-permissie, zodat we niet per ongeluk gaan editten. Het is natuurlijk mogelijk om onszelf voor de gek te houden door de permissies met `chmod` te veranderen, maar als we de gewijzigde file met `ci` weer willen inchecken dan protesteert deze. We kunnen dan alsnog een lock zetten, maar het is beter om de officiële werkwijze aan te houden.

Wanneer we op een gegeven moment een oude versie terug willen hebben kunnen we deze

opgeven aan *co* met de *-r* (*revisie*) optie. De eerste versie van de file heeft revisie nummer 1.1, de tweede 1.2, etc. Bij elke check-in wordt de nieuwe versie bij de versiefile gestopt[2]. Figuur 7.2 geeft een voorbeeld van dit proces.

Figuur 7.2 RCS gebruik van revisies



ci	check in: stop de werkfile in de versiefile, maak de versiefile aan als er nog geen is. Vraagt om een beschrijving van de wijziging.
co	check out: maak een nieuwe werkfile aan vanuit de versiefile (read-only)
co -l	maak een nieuwe werkfile aan om te editten. versie wordt *gelockt*
rcs -*opties*	diverse administratieve operaties op de versiefile, bijv. het zetten en verwijderen van locks
rcsdiff	geef de verschillen tussen versies
rcsmerge	voeg parallelle wijzigingen samen (7.6
rlog	geef een overzicht van de revisies
parameter	alle commando's moeten één of meer files als parameter hebben. Naar keuze mag de werkfile of de versiefile gegeven worden.

Tabel 7.1: Basisoperaties met RCS

7.2 Revisienummering

Bij iedere check-in operatie maakt RCS een nieuwe revisie aan. De nummering begint met 1.1 – 1.2 – etc. Het eerste nummer blijft dus gelijk, het tweede wordt telkens opgehoogd. Je kunt

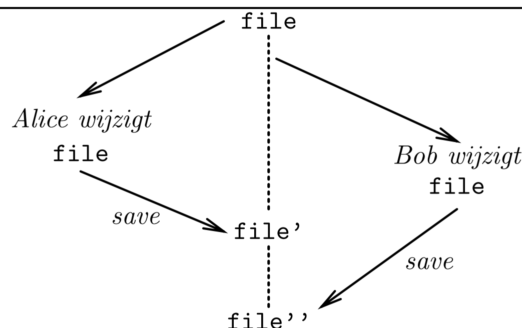
[2]In feite wordt de *diff* (zie 5.8) gebruikt i.v.m. de efficiëntie.

zelf beslissen wanneer het eerste nummer verandert. Meestal wordt dit gedaan wanneer er significante wijzigingen in de software optreden, bij software die verkocht wordt bijvoorbeeld wanneer er een nieuwe uitgave komt. Om het revisienummer te veranderen geef je aan de *ci* opdracht de *-r* optie mee, bijv. *ci -r2 file*. De nieuwe versie wordt dan 2.1, en volgende check-in's geven 2.2, etc. Het is ook mogelijk om een versie een symbolische naam te geven. Dit kan met het commando *rcs -n'Naam':'revnr'*, bijv. *rcs -nTestversie:2.1 file* geeft versie 2.1 de symbolische naam Testversie. De symbolische namen mogen in alle RCS commando's gebruikt i.p.v. de revisienummers.

7.3 Samenwerking

In het begin van dit hoofdstuk hebben we al genoemd dat RCS het samenwerken aan de ontwikkeling van een programma kan vergemakkelijken. Laten we eerst kijken naar de problemen die kunnen optreden wanneer twee (of meer) personen eenzelfde file willen wijzigen. Figuur 7.3 geeft een beeld van het foute verloop. De wijzigingen van Alice gaan verloren omdat ze worden overschreven door die van Bob.

Figuur 7.3 Edit conflict



Sommige editors, bijv. Emacs kunnen de gebruikers waarschuwen voor dergelijke situaties, maar wanneer een van beide een andere editor gebruikt of wanneer bijv. een tekenpakket gebruikt wordt om de file te wijzigen dan helpt dit niet. RCS kan hier te hulp komen doordat in eerste instantie elk van de personen een eigen werkfile heeft, en bovendien RCS locks bijhoudt bij de versiefile. Dit houdt natuurlijk in dat er één gemeenschappelijke versiefile bijgehouden wordt. Omdat Alice wil editten zal zij een *co -l file* opdracht geven, waardoor de file voor haar gelockt wordt. Als Bob daarna ook een *co -l file* doet krijgt hij een waarschuwing dat de file gelockt is. Bob heeft de mogelijkheid om toch door te gaan, en kan zelfs de lock "stelen" maar dan krijgt Alice een boodschap (per e-mail) dat Bob dit gedaan heeft. Ze kan dan eventueel gaan klagen bij hun beider baas. In een dergelijke situatie is het natuurlijk het verstandigst om even onderling overleg te plegen. Wanneer één van beide personen een *co -l* wil doen en de ander alleen maar een read-only copie wil hebben is er geen probleem.

Om een gezamenlijke verzameling versiefiles te beheren is het beter om deze in een aparte directory te zetten (ook bij gebruik door één persoon is dit vaak overzichtelijker). Wanneer er in de werkdirectory een subdirectory met naam `RCS` bestaat, dan plaats RCS hierin alle versiefiles. Dit mag ook een symbolische link naar een andere directory zijn, en dit kan voor

gezamenlijk gebruik aangewend worden. Alle personen die met die files moeten werken moeten dan wel schrijf-permissie naar die directory hebben. Het gemakkelijkst is dit als voor het project een aparte groep (in de zin van de file-permissies) aanwezig is. In een studiepraktikum situatie is dit niet een haalbare zaak, alle studenten zitten dan bijv. in de groep `student`. Als nu de versie-directory voor zowel Alice als Bob toegankelijk is via de gemeenschappelijke groep, dan is deze ook toegankelijk voor alle anderen van dezelfde groep.In dat geval kan de volgende truc toegepast worden (dit kan zelfs gebruikt worden als je niet in een gemeenschappelijke groep zit):

Een van beide (zeg Alice) maakt een aparte directory aan, zeg `/users/alice/project`. Deze krijgt toegang `rwx--x--x`. In deze directory wordt een subdirectory gemaakt met een samen afgesproken, geheime, en moeilijk te raden naam, bijv. `x@y!a+b`. Dit wordt de RCS directory en deze krijgt permissies `rwxrwxrwx`, dus toegankelijk voor iedereen! Beide personen leggen nu een symbolische link vanuit hun werkdirectory met de naam RCS en als bestemming de geheime directory. Voor Alice is dit geen probleem, omdat de directory bij haar aanwezig is; voor Bob kan dit alleen als Alice zorgt dat het hele pad naar deze directory tenminste `x` permissie voor Bob heeft. Omdat de versie-directory schrijfpermissie voor iedereen heeft kan Bob er bij. Ieder ander persoon moet eerst de geheime naam zien te vinden. Doordat de `project` directory voor buitenstaanders geen `r` permissie heeft, kunnen anderen niet via een *ls* commando de naam vinden. Alice en Bob moeten er nu voor zorgen dat hun werkdirectories ook geen `r` permissie heeft voor anderen, anders kunnen die via een*ls* commando op deze directories de waarde van de symbolische link vinden.

7.4 RCS informatie

Het kan nuttig zijn om in de werkfile informatie over de versie op te nemen. Deze informatie wordt door RCS bijgehouden in de versie-file. Om deze automatisch in de werkfile op te nemen zijn er een aantal *keywords* beschikbaar.

De belangrijkste keywords zijn: Id voor een beknopte samenvatting van filenaam, versie, datum, auteur e.d., Version voor alleen het versienummer, en Log voor een lijst van redenen van de wijzigingen (die je bij check-in hebt moeten opgeven). RCS reageert op de keywords als ze tussen $$ tekens staan, dus bijv. $Version$. Dit wordt bij check-out vervangen door bijv. $Version 1.3 $. Bij de volgende check-in wordt deze uitgebreide vorm ook herkend. Wanneer het alleen van belang is dat de informatie in de file staat, kan het keyword in commentaar geplaatst worden, dus in C bijv.

```
/* $Version$ */
```

In C programma's zien we ook vak de volgende constructie:

```
static char rcsid[ ] = $Id: ch7.tex,v 1.1 1998/07/22 12:48:41 piet Exp piet $;
```

In dat geval wordt de informatie ook in de gecompileerde code opgenomen, tenzij je een sterk optimaliserende compiler hebt, die ontdekt dat de variabele toch niet gebruikt wordt. Het Log keyword moet iets anders behandeld worden omdat RCS de bijbehorende informatie niet tussen de $$ plaatst maar op de volgende regels (omdat dit meer dan één regel is). Je moet dit dan als commentaar ook over meer regels opnemen, dus a.v.:

```
/* $Log: ch7.tex,v $
 * Revision 1.1  1998/07/22 12:48:41  piet
 * Initial revision
 *
*/
```

Voor andere programmeertalen moet je een vorm vinden die toepasselijk is. Voor TEX en LATEX documenten is er een apart package `rcs.sty` beschikbaar om op handige wijze de keywords op te nemen.

7.5 Vertakkingen

Soms is een lineaire volgorde van de revisies niet mogelijk. Bijvoorbeeld wanneer we een software product verkopen, en op een gegeven moment aan een andere versie willen gaan werken. De klanten hebben echter nog de oude versie, en als daar een probleem mee is, moeten we een wijziging op de oude versie kunnen aanbrengen. Het ophalen van de oude versie is geen probleem, dit kan met *co -r*, maar als we de gewijzigde versie weer willen opslaan moet dit niet als nieuwe revisie van de ontwikkelsoftware gebeuren. Stel dat revisie 1.3 de klantenversie is, en de ontwikkelversie is intussen tot 2.2 gekomen. Een gewone checkin zou nu revisie 2.3 opleveren, maar dit zou beschouwd worden als een nieuwe revisie van de ontwikkelversie. We willen echter een “zijtak” bouwen aan revisie 1.3. Om duidelijk te maken dat we werken aan een zijtak gebruikt RCS revisienummers met meer cijfers, zoals 1.3.1.1, 1.3.1.2, . . . Door aan het check-in commando het gemeenschappelijke deel hiervan mee te geven begint RCS een nieuwe zijtak, dus bijv.

```
ci -r 1.3.1 file
```

Zie figuur 7.4.

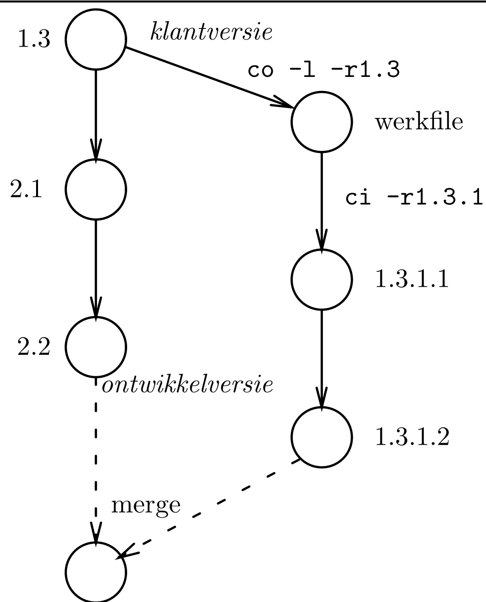
7.6 Merge

In figuur 7.4 zien we nog een mogelijkheid die RCS biedt: een “merge” tussen verschillende wijzigingen. Stel dat de wijziging die we op de klantversie hebben toegepast (dus het verschil tussen revisies 1.3 en 1.3.1.2) ook relevant is voor de ontwikkelversie. We kunnen natuurlijk handmatig alle wijzigingen die we gemaakt hebben nog eens uitvoeren op versie 2.2, maar dit is saai werk. We kunnen ook RCS het laten doen, en we spreken dan van een merge operatie. Aannemend dat *file* nog steeds revisie 1.3.1.2 bevat geven we

```
rcsmerge -r1.3 -r2.2 file
```

Na deze operatie bevat file de verschillen tussen 1.3 en 2.2 toegevoegd aan wat er al in stond. Dit is hetzelfde als de verschillen tussen 1.3 en 1.3.1.2 toegevoegd aan revisie 2.2: de merge operatie werkt als een soort vector-optelling. Na de merge kan een check-in gedaan worden zodat versie 2.3 ontstaat. Natuurlijk moet het resultaat van de merge operatie eerst gecontroleerd worden. Wanneer de merge niet lukt, bijv. omdat er een conflict was met een

Figuur 7.4 RCS zijtakken



wijziging die in revisie 2.2 zit, dan zal merge dit aangeven: In de file wordt het conflictgedeelte aangegeven met <<<<<< en >>>>>>. Het is ook mogelijk de uitvoer van de merge naar een andere file te sturen met (*-p* schrijft de merge naar de standaard uitvoer):

```
rcsmerge -p -r1.3 -r2.2 file > file.new
```

Merging kan ook gebruikt worden als twee personen tegelijkertijd wijzigingen op dezelfde file hebben aangebracht, bijv. door de lock te doorbreken.

Voor verdere details m.b.t. de opties van de RCS commando's zie de manual pagina's.

Hoofdstuk 8

Compilatiebeheer

Bij een software project waar de code verdeeld is over meerdere files dient zich vaak het probleem aan dat een kleine wijziging in één van de files tot gevolg heeft dat ook andere files gecompileerd moeten worden. Als echter het aantal files groot is, dan willen we de hoeveelheid compilaties tot zo weinig mogelijk beperken. Als een programma samengesteld is uit de files `a1.c`, `a2.c`, `b.c`, dan worden deze i.h.a. apart gecompileerd en tijdens het werken aan het project blijven de gecompileerde versies `a1.o`, `a2.o`, `b.o` beschikbaar. Wanneer b.c gewijzigd wordt dan hoeft alleen deze gecompileerd te worden en de andere `.o` files kunnen onveranderd blijven.

Ingewikkelder wordt het als we ook header files hebben, bijv. `a.h` voor de beide `a` files en `b.h` die voor alle files gebruikt wordt (bijv. omdat er prototypes van functies in `b` in staan, die vanuit de `a`'s aangeroepen worden). In dat geval zal een wijziging in `a.h` tot gevolg hebben dat we de beide `a` files moeten hercompileren, en een wijziging in `b.h` zal tot gevolg hebben dat alle files gehercompileerd moeten worden.

Het is natuurlijk mogelijk om deze informatie in het hoofd van de programmeur te laten houden, maar dat leidt tot fouten. Ook als met meer mensen aan een project gewerkt wordt leidt dit tot problemen omdat dan ook gecontroleerd moet worden welke andere files gewijzigd zijn. *Compilatiebeheer* is een hulpmiddel om deze dingen te automatiseren. Op Unix systemen wordt dit meestal met het programma *make* gedaan. Alhoewel hier gesproken wordt over compilatie, moet dit in de meest algemene zin geïnterpreteerd worden. Dus bijv. het verwerken van LaTeX documenten valt daar ook onder.

Het programma *make* heeft een beschrijving nodig waarin staat aangegeven:

- Welke afhankelijkheden er tussen files bestaan. Daarbij betekent "A is afhankelijk van B" dat om A te maken, je B nodig hebt. Of anders gezegd, dat als B verandert, A opnieuw gemaakt zal moeten worden. Dus bijv. `b.o` is afhankelijk van `b.c` en `b.h` in bovenstaand voorbeeld.

- Regels: acties om de afhankelijke file te maken, bijv. om `b.o` te maken moet het commando *`cc -c b.c`* uitgevoerd worden. Let op dat niet elke file waarvan `b.o` afhankelijk is in het commando hoeft voor te komen. In dit geval is de afhankelijkheid van `b.h` geregeld door een regel `#include "b.h"` in de file `b.c`.

Deze gegevens worden in een file gezet die traditioneel de naam *Makefile* of *makefile* heeft. Aannemend dat het uiteindelijke programma `ab` heet, ziet de *Makefile* voor dit programma er als volgt uit:

```
ab:     a1.o a2.o b.o
        cc a1.o a2.o b.o -o ab

b.o:    b.c b.h
        cc -c b.c

a1.o:   a1.c a.h b.h
        cc -c a1.c

a2.o:   a2.c a.h b.h
        cc -c a2.c
```

Een afhankelijkheid bestaat uit de naam van de afhankelijke file (de *target*), gevolgd door een `:`, gevolgd door de namen van de files waarvan hij afhankelijk is. Het is ook mogelijk om meer targets voor de `:` te nemen als ze van dezelfde files afhankelijk zijn.

Een actie-regel bestaat uit één of meer shell commando(s) en wordt altijd voorafgegaan dor een TAB symbool.

De opdracht *make -f Makefile ab* zal nu alle compilaties uitvoeren die nodig zijn om het programma `ab` te maken. Als geen *-f file* optie meegegeven wordt, zoekt *make* naar de files *Makefile* of *makefile*. Als geen *target* meegegeven wordt dan wordt de eerste target uit de makefile genomen. Een regel wordt uitgevoerd als de target niet bestaat, of ouder is dan minstens een van de files waarvan hij afhankelijk is.

8.1 Macro's

Het is mogelijk om in een makefile een soort variabelen, ook wel macro's genoemd te gebruiken. Deze kunnen een tekst als parameter hebben. Als we bijv. willen kiezen welke C compiler we willen gebruiken dan kunnen we de naam daarvan in een macro opnemen. Macro's gebruiken we met $(naam) of $X als de naam slecht uit één teken bestaat. Voor de C compiler wordt meestal de naam CC gebruikt, eventuele extra opties voor de C compiler CFLAGS, en speciale opties voor het samenstellen van het programma uit de `.o` files de naam LDFLAGS. Onze makefile wordt dan bijv.

```
CC=gcc
CFLAGS=-O

ab:     a1.o a2.o b.o
        $(CC) $(LDFLAGS) a1.o a2.o b.o -o ab

b.o:    b.c b.h
        $(CC) $(CFLAGS) -c b.c

a1.o:   a1.c a.h b.h
        $(CC) $(CFLAGS) -c a1.c

a2.o:   a2.c a.h b.h
        $(CC) $(CFLAGS) -c a2.c
```

Het is mogelijk om via de *make* opdracht de waarden van variabelen te veranderen zonder de *makefile* te hoeven aanpassen: *make CFLAGS=-g*. Environment variabelen kunnen zelfs gebruikt worden om ongedefinieerde macro's te zetten. Als in bovenstaand voorbeeld CC niet in de makefile gezet wordt, maar in een environment variabele dan kunnen we gemakkelijk kiezen welke C compiler gebruikt moet worden.

8.2 Standaard regels

In bovenstaand voorbeeld is al duidelijk dat sommige regels wel erg veel op elkaar lijken. Het is mogelijk om hiervoor standaardregels te definiëren. Om een `.o` file te maken van een `.c` file moeten we altijd de C compiler aanroepen, dus de algemene regel zou zo iets zijn:

```
*.o:    *.c
        $(CC) $(CFLAGS) -c *.c
```

Het kan *niet* op deze manier opgeschreven worden maar wel op deze:

```
.c.o:
        $(CC) $(CFLAGS) -c $*.c
```

`$*` is een speciale macro die gebruikt kan worden in deze standaardregels en die als waarde heeft het deel van de filenaam vóór de punt. Het is ook mogelijk om `$<` te gebruiken voor de te compileren file (de *.c* file in dit voorbeeld. (De macro's `$*` en `$<` werken alleen met standaard regels, niet voor gewone regels. Voor gewone regels is er een macro `$@` die de hele filenaam van de target als waarde heeft.) De standaardregels kunnen alleen gebruikt worden als de *suffixen* *.o* en *.c* als zodanig aangemeld zijn. Dit gebeurt met een regel van de vorm:

```
.SUFFIXES: .o .c .tex .dvi
```

Om met bovenstaande regel automatisch *latex* aan te laten roepen op *.tex* files kunnen we dus de volgende regel toevoegen:

```
LTX=latex
.tex.dvi:
        $(LTX) $*
```

In feite hoeven de regels betreffende de `.c` en `.o` files niet opgegeven te worden omdat deze ingebouwd zijn in *make*, net als regels over Pascal compilaties e.d. Het feit dat `b.o` afhankelijk is van `b.c` hoeft dan ook niet expliciet in de makefile opgenomen te worden, net zo min als de compilatiecommando's. Wat wel opgegeven moet worden is de afhankelijk van de `.h` files, omdat *make* dat niet kan weten. Ook de CC macro is standaard gedefiniëerd met waarde *cc*. Als we nu onze makefile strippen van overbodige regels, en een LaTeX documentatie toevoegen krijgen we:

```
CFLAGS=-O
LTX=latex

.SUFFIXES: .tex .dvi
.tex.dvi:
        $(LTX) $*

all:    ab ab.dvi

ab:     a1.o a2.o b.o
        $(CC) $(LDFLAGS) a1.o a2.o b.o -o ab

b.o:    b.h

a1.o a2.o:      a.h b.h

clean:  rm *.o
```

De compilatiecommando's voor de `.o` files zijn al bij *make* bekend. De targets `all` en `clean` zijn een beetje merkwaardig: er worden nl. geen files `all` en `clean` door de bijbehorende regels gemaakt. Dit zijn z.g. *phony* targets. De eerste wordt gebruikt om een paar echte targets bij elkaar te groeperen. Hiervoor wordt traditioneel de naam "all" gebruikt, maar kan natuurlijk zelf gekozen worden. Meestal staat deze als eerste target, zodat een simpel *make* commando al het werk doet. Het target "clean" wordt meestal gebruikt om overbodige files op te ruimen. Je kunt dan gewoon *make clean* zeggen.

8.3 Make en RCS

Moderne make programma's zoals GNU make hebben ingebouwde kennis over RCS. Het is ook mogelijk om zelf regels voor RCS files te definiëren, bijv:

```
.SUFFIXES .c,v
.c,v.c:
        co $<
```

Maar dit werkt niet als de RCS files in een aparte RCS directory opgeslagen worden. In dat geval kunnen natuurlijk ook voor elke file aparte regels opgenomen worden:

```
a1.c:   RCS/a1.c,v
        co a1.c
```

Dit soort regels zijn te ingewikkeld om als standaard regels opgenomen te worden, want ze bestaan niet alleen uit suffixen. Met een krachriger *make* programma zoals GNU make kan het echter wel.

Appendix A

Talstelsels

In het dagelijkse leven schrijven we getallen in het *tientallig* of *decimale* stelsel, d.w.z. dat we 10 cijfers gebruiken (0 t/m 9). Het is echter ook mogelijk en bij computers zelfs gebruikelijk om andere talstelsels te gebruiken. In het tientallig stelsel geldt voor een getal dat met de cijfers $a_2a_1a_0$ geschreven wordt dat het de waarde

$$a_2 \cdot 10^2 + a_1 \cdot 10^1 + a_0 \cdot 10^0$$

heeft. I.h.a. een getal dat met cijfers $a_n \dots a_0$ geschreven wordt heeft de waarde

$$a_n \cdot 10^n + \cdots + a_0 \cdot 10^0$$

Bijvoorbeeld het getal "137" heeft de waarde $1 \cdot 100 + 3 \cdot 10 + 7 \cdot 1$.

Waneer we een ander getal b als *basis* van ons talstelsel nemen dan zullen we analoog de cijfers 0 t/m $b - 1$ nemen en dan is de waarde van een getal met cijfers $a_n \dots a_0$ het getal

$$a_n \cdot b^n + \cdots + a_1 \cdot b^1 + a_0 \cdot b^0$$

Een veel gebruikt talstelsel is het *octale* of acht-tallige stelsel waarbij het grondtal 8 is en de cijfers dus 0 tm/ 7. Het getal dat geschreven wordt als "137" in het achttalige stelsel heeft dus de waarde (in het tientallig stelsel genoteerd) van $1 \cdot 64 + 3 \cdot 8 + 7 \cdot 1 = 95$. Om aan te geven in welk talstelsel een getal genoteerd wordt als we er meer door elkaar gebruiken zetten we het talstelsel (in decimale notate) er wel eens klein onder, dus: $137_8 = 95_{10}$.

Een ander talstelsel dat regelmatig bij computers gebruikt wordt is het *hexadecimale* stelsel dat een grondtal van 16 heeft. Omdat we dan niet genoeg cijfers hebben worden er letters gebruikt: a=10, b=11, c=12, d=13, e=14, f=15. Om het bovenstaande voorbeeld verder te gebruiken: $137_{16} = 311_{10}$.

Opgave A.1 Geef de octale en hexadecimale representatie van het decimale getal 137.

Een grappige eigenschap is dat $100_8 = 64_{10}$ en $100_{10} = 64_{16}$.

Computers werken intern vaak met het binaire stelsel waarvan het grondtal 2 is, en de cijfers dus 0 en 1 (ook wel bits[1]) genoemd). Octale getallen zijn gemakkelijk om te zetten naar

[1] Afkorting van *bi*nary digi*ts*

het binaire systeem door elk octaal cijfer uit te schrijven als 3 binaire cijfers en die achter elkaar te zetten, bijvoorbeeld $137_8 = (001)(011)(111) = 1011000_2$. Op dezelfde manier kan een hexadecimaal getal omgezet worden naar een binair getal door elk hexadecimaal cijfer in 4 bits om te zetten. Omgekeerd kan ook: hak het binaire getal van rechts af in stukken van 3 (voor octaal) of 4 (voor hexadecimaal) en vervang elk stuk door het betreffende cijfer.

Index

/dev/null, 31
/dev/tty, 31
$\LaTeX$, 102

aap noot mies, 40
abracadabra, 88
absolute padnaam, 8
abstractie, 5
achtergrond, 20, 46
adressenbestand, 82
afhankelijkheid, 99
alfabetische volgorde, 68
apparaten, 30
ARGC, 79, 91
ARGV, 79, 91
array, 85
  meerdimensionaal, 87
ASCII, 25, 68, 78, 84
assignment, 77
asynchroon, 46
atan2, 89
attributen, 28
awk, 76

backquotes, 49
backslash, 44
bedrijfssysteem, 4
BEGIN, 76
bescherming, 5
bestand, 7
besturingssysteem, 4
bg, 22
binair, 104
binary, 9
bits, 104
Bourne shell, 38
branch, 97
bundel, 54
byte, 25

C shell, 38, 59
case, 48
cat, 13, 24, 60
cd, 11
checkin, 93
checkout, 93
chgrp, 28
chmod, 28
chown, 28
ci, 93
co, 93, 103
comm, 73
commando substitutie, 49
compilatiebeheer, 99
compileren, 9, 99
concatenatie, 25
concateneren, 25
control code, 25
control toets, 6
control-C, 22
control-Z, 22
CONVFMT, 79
copieren, 8, 35
cos, 89
cp, 8, 14, 35, 60
csh, 38

decimaal, 104
delete, 86
DESCRIPTION, 12
devices, 30
diff, 74, 94
directory, 7, 30
  maken, 30
  verwijderen, 30

echo, 40
egrep, 61, 64
eigenaar, 28

END, 76
ENVIRON, 79, 91
environment, 43
environment variabelen, 43
executable, 9, 39
exit status, 45
exp, 89
export, 43
expr, 50

fg, 22
fgrep, 61, 65
fiets, 78
file, 7
file descriptor, 14, 17
filenaam, 7
FILENAME, 79
files, 24
- copieren, 8, 35
- lijst, 8
- vergelijken, 73, 74
- verplaatsen, 9, 35
- verwijderen, 9, 35

filter, 13, 60
floating point, 64, 89
FNR, 79
for, 48, 77, 85
foutmeldingen, 17
FS, 79
functie
- awk, 89
- shell, 56

gebruiker, 28
geheugen, 10
gemiddelde, 81
getal, 64
getalvolgorde, 68
getenv, 43
GNU, 76
grafische shell, 6
grep, 61, 63
groep, 28, 96
gsub, 89

harde link, 32
head, 71
header files, 99
here document, 52
hexadecimaal, 25, 104
HOME, 43, 91
hoofdletters, 62, 68

I/O redirection, 16, 52
if, 47, 78
ifs, 43
index, 85, 87
input redirection, 52
int, 89
invoer, 6, 14
ISO, 25

job, 45
joker, 34

keywords, RCS, 96
kill, 22
kleine letters, 62, 68
Korn shell, 38, 58
ksh, 38

Latin1 code, 25
length, 87
lijst, 46
linefeed, 25
link, 31
- harde, 32
- symbolische, 32

ln, 31–33
lock, 93
log, 89
logaritme, 89
ls, 8, 27

macro
- make, 100

make, 99
man, 11
match, 48, 63, 78, 87
maximum, 81
meerdimensionaal, 87
merge, 97
metacharacters, 34, 40, 44
minimum, 81
mkdir, 30
mode, 27

modificatietijd, 28
multi-tasking, 5, 19
multi-user, 5
mv, 9, 35

NAME, 11
nawk, 76
newline, 25, 44, 66, 85
NF, 79
NR, 78, 79

octaal, 25, 104
octal dump, 25
od, 25
OFMT, 79
OFS, 79
omgeving, 14, 60
Operating System, 4
ORS, 79
output redirection, 16, 20, 52, 89

padnaam
  absoluut, 8
  relatief, 8
padnamen, 7
patch, 75
PATH, 43
patronen, 63
patroon, 48, 77
permissies, 28
pipe, 19, 45, 90
pipeline, 45
print, 81
printf, 83, 84
proces, 9, 13
  stoppen, 21
proces nummer, 20
programma, 9
prompt, 6
protecties, 28
ps, 10, 20

quotes, 45

rand, 89
RCS, 92, 93, 102
  branch, 97
  keywords, 96
  lock, 93
  merge, 97
  revisie, 93
  takken, 97
rcsmerge, 97
redirection, 16
regelnummers, 81
reguliere expressie, 51, 63, 72, 78
relatieve padnaam, 8
return, 6
return value, 45
rlength, 87
rm, 9, 35
rmdir, 30
RS, 79
rstart, 87
rwx, 28

SCCS, 92
script, 39, 76
sed, 72, 76
SEE ALSO, 12
sessie, 6
shell, 6, 13, 15, 38
  variabelen, 41
shell functie, 56
shell script
  argumenten, 41
shift, 41, 55
sin, 89
sort, 67
sorteersleutel, 70
sorteren, 67
spatie, 82
split, 88
sprintf, 88
sqrt, 89
squeeze, 67
srand, 89
standaard error, 17, 55, 90
standaard invoer, 14
standaard uitvoer, 14, 55
standard input, 14
standard output, 14
status code, 45
stderr, 17, 90
stdin, 14

stdout, 14
stop, 22
stream editor, 72
stroom, 60
sturing, 14
sub, 88
SUBSEP, 79, 87
subshell, 39
substitutie, 72, 88
substr, 88
symbolische link, 32, 96
SYNOPSIS, 12

tab, 82
tabel, 86
tail, 71
takken, RCS, 97
talstelsel, 104
target, 100
tekst, 24
terminal, 10
test, 49
tientallig, 104
tijdsdiagram, 15
tolower, 88
toupper, 88
tr, 65, 69
troep, 73

uitvoer, 6, 14
Unicode, 27
uniq, 67, 69
Unix, 5
until, 47

variabelen
    awk, 78
    make, 100
velden, 70, 82
veldscheider, 70, 82
verplaatsen, 9, 35
versie-beheer, 92
versiefile, 93
vertaaltabel, 86
verwijderen, 9, 35
    directory, 30

wc, 29
werkdirectory, 7
werkfile, 93
while, 47, 78
wildcard, 34
window, 10
woord, 64
word count, 29
wortel, 89

zwarte doos, 13