# BASH Programmering - Introductie HOW-TO

door Mike G mikkey at dynamo.com.ar
Vertaald door: Ellen Bokhorst, bokkie at nl.linux.org Do jul 27 09:36:18 ART 2000

Dit artikel is bedoeld om je te helpen een begin te maken met het programmeren van basis tot middelmatige shell-scripts. Het is niet bedoeld als een document voor gevorderden (zie de titel). Ik ben GEEN expert noch een goeroe shellprogrammeur. Ik besloot dit te schrijven omdat ik er veel van zal leren en het wellicht van nut kan zijn voor andere mensen. Feedback wordt zeer gewaardeerd vooral in de vorm van een patch :).

## Inhoudsopgave

# 1 Introductie

## 1.1 Ophalen van de laatste versie

*http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html*

## 1.2 Benodigdheden

Bekendheid met GNU/Linux opdrachtregels, en bekendheid met basisprogrammeerconcepten is prettig. Alhoewel dit geen introductie is in programmeren, wordt er van veel basisconcepten een uitleg gegeven (of op z'n minst wordt dit geprobeerd).

## 1.3 Gebruik van dit document

Dit document probeert te helpen bij de volgende situtaties

- Je hebt een idee over programmeren en je wilt beginnen een aantal shellscripts te coderen.
- Je hebt een vaag idee over shellprogrammering en wilt een soort van referentie.
- Je wilt een aantal shellscripts zien met wat opmerkingen om te beginnen je eigen shellscripts te gaan schrijven.
- Je gaat over van DOS/Windows (of je deed dit al) en wil "batch"processen maken.
- Je bent een echte nerd en leest iedere beschikbare how-to

# 2 Zeer eenvoudige Scripts

In deze HOW-TO wordt getracht je een aantal hints te geven over shellscriptprogrammering welke sterk is gebaseerd op voorbeelden.

In deze sectie zijn een aantal kleine scripts te vinden die je hopelijk op weg helpen een aantal technieken te doorgronden.

### 2.1 Traditioneel hello world script

```
#!/bin/bash
echo Hello World
```

Dit script bestaat slechts uit twee regels. In de eerste regel wordt het systeem duidelijk gemaakt welk programma moet worden gebruikt om het bestand uit te voeren.

De tweede regel bestaat uit alleen die actie waarmee 'Hello World! op de terminal wordt afgedrukt.

Als je iets krijgt als *./hello.sh: Command not found.* is waarschijnlijk de eerste regel '#!/bin/bash' niet goed, roep 'whereis bash' of zie 'bash zoeken' om te bezien hoe je deze regel zou moeten schrijven.

### 2.2 Een zeer eenvoudig backupscript

```
#!/bin/bash
tar -cZf /var/my-backup.tgz /home/me/
```

In plaats dat er nu een bericht op de terminal wordt weergegeven, maken we in dit script een tar-ball van de homedirectory van een gebruiker. Het is niet de bedoeling dat dit wordt gebruikt. Later in dit document wordt een handiger backupscript gepresenteerd.

## 3 Alles over omleiding

### 3.1 Theorie en snelle naslag

Er zijn 3 file descriptors, stdin, stdout en stderr (std=standaard).

In principe kun je:

1. stdout naar een bestand omleiden
2. stderr naar een bestand omleiden
3. stdout naar stderr omleiden
4. stderr naar stdout omleiden
5. stderr en stdout naar een bestand omleiden
6. stderr en stdout naar stdout omleiden
7. stderr en stdout naar stderr omleiden

1 is een representatie van stdout en 2 van stderr.

Een opmerking hierover: met de opdracht less kun je zowel stdout (welke in de buffer blijft) en stderr, welke op het scherm wordt weergegeven, bekijken, maar is verwijderd als je door de buffer probeert te 'bladeren'.

## 3.2 Voorbeeld: stdout naar bestand

Hiermee wordt de uitvoer van een programma naar een bestand weggeschreven.

```
ls -l > ls-l.txt
```

Hier zal een bestand, genaamd 'ls-l.txt' worden aangemaakt en hierin zal worden opgevangen wat je krijgt als je de opdracht 'ls -l' typt en uitvoert.

## 3.3 Voorbeeld: stderr naar bestand

Dit zorgt dat de uitvoer van stderr van een programma naar een bestand wordt weggeschreven.

```
grep da * 2> grep-errors.txt
```

Hier zal een bestand, genaamd 'grep-errors.txt' worden aangemaakt en hier zal dan het stderr gedeelte van de uitvoer in staan van de opdracht 'grep da *'.

## 3.4 Voorbeeld: stdout naar stderr

Dit zorgt dat de sterr uitvoer van een programma naar dezelfde filedescriptor als stdout wordt weggeschreven.

```
grep da * 1>&2
```

Hier wordt het stdout gedeelte van de opdracht naar stderr gezonden, je merkt dat mogelijk op verschillende manieren.

## 3.5 Voorbeeld: stderr naar stdout

Dit zorgt dat de stderr uitvoer van een programma naar dezelfde filedescriptor wordt weggeschreven als stdout.

```
grep * 2>&1
```

Hier wordt het stderr gedeelte van de opdracht naar stdout gestuurd, als je middels een pipe-symbool de uitvoer naar less stuurt, zul je zien dat de regels die normaal gesproken 'verdwijnen' (als ze naar stderr worden geschreven) nu behouden blijven (omdat ze zich op stdout bevinden).

## 3.6 Voorbeeld: stderr en stdout naar bestand

Hiermee zal alle uitvoer van een programma in een bestand worden geplaatst. Soms is dit geschikt voor cron entries, als je een opdracht in absolute stilte wilt laten uitvoeren.

```
rm -f $(find / -name core) &> /dev/null
```

Dit (denkend aan een cron entry) zal ieder bestand genaam 'core' in iedere directory verwijderen. Merk op dat je er nogal zeker van moet zijn wat een opdracht doet als je de uitvoer ervan gaat verwijderen.

# 4 Pipes

In deze sectie wordt op een zeer eenvoudige en praktische wijze uitgelegd hoe gebruik te maken van het pipe-symbool en waarvoor je het zou kunnen gebruiken.

## 4.1 Wat het zijn en waarvoor je het zou kunnen gebruiken

Pipe-symbolen laten je gebruik maken van (zeer eenvoudig uitgelegd) de uitvoer van een programma als de invoer van een ander programma.

## 4.2 Voorbeeld: eenvoudige pipe met sed

Dit is een zeer eenvoudige manier om gebruik te maken van een pipe-symbool.

```
ls -l | sed -e "s/[aeio]/u/g"
```

Wat er hier gebeurt is het volgende: eerst wordt de opdracht 'ls -l' uitgevoerd, en in plaats dat de uitvoer ervan wordt afgedrukt, wordt het naar het programma sed gestuurd (piped), wat op zijn beurt afdrukt wat het moet afdrukken.

## 4.3 Voorbeeld: een alternatief voor ls -l *.txt

Waarschijnlijk is dit een moeilijkere manier om een ls -l *.txt uit te voeren, maar het dient hier ter illustratie, niet voor het oplossen van een opsommingsdilemma.

```
ls -l | grep "\.txt$"
```

Hier wordt de uitvoer van het programma ls -l naar het grep programma gezonden, welke op zijn beurt de regels afdrukt die overeenkomen met de reguliere expressie "\.txt$".

# 5 Variabelen

Je kunt net als in iedere andere programmeertaal gebruik maken van variabelen. Er zijn geen gegevenstypen. Onder bash kan een variabele bestaan uit een nummer, een teken, of een reeks tekens.

Je hoeft een variabele niet te declareren. Door er slechts een waarde aan toe te kennen zal het worden aangemaakt.

## 5.1 Voorbeeld: Hello World! door gebruik te maken van variabelen

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

In regel 2 wordt een variabele aangemaakt met de naam STR en hieraan wordt de string "Hello World!"toegekend. De WAARDE van deze variabele wordt vervolgens opgehaald door er een '$' voor te plaatsen. Let er alsjeblieft op (probeer het uit!) dat als je het '$' teken niet gebruikt, de uitvoer van het programma iets anders zal zijn, en het waarschijnlijk niet datgene zal zijn wat je wilt dat 't is.

### 5.2 Voorbeeld: Een zeer eenvoudig backupscript (iets beter)

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -cZf $OF /home/me/
```

In dit script wordt iets anders geïntroduceerd. Ten eerste zou je nu bekend moeten zijn met de aanmaak van de variabele en de toekenning in regel 2. Let op de expressie '$(date +%Y%m%d)'. Als je het script uit laat voeren, zal je bemerken dat het de opdracht binnen de haakjes uitvoert, en de uitvoer ervan afvangt.

Merk op dat de naam van het bestand, de uitvoer, iedere dag anders zal zijn, vanwege de opmaakoptie van de opdracht date (+%Y%m%d). Je kunt dit wijzigen door een ander formaat op te geven.

Nog wat meer voorbeelden:

echo ls

echo $(ls)

### 5.3 Lokale variabelen

Lokale variabelen kunnen worden aangemaakt door gebruik te maken van het keyword *local*.

```
#!/bin/bash
HELLO=Hello
function hello {
        local HELLO=World
        echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

Dit voorbeeld zou genoeg moeten zijn om aan te tonen hoe je een lokale variabele gebruikt.

## 6 Voorwaardelijke opdrachten

Voorwaardelijke opdrachten laten je een beslissing nemen of een actie wel of niet zal worden uitgevoerd, de beslissing wordt genomen door de waarde te bepalen van een expressie.

### 6.1 Droge Theorie

Voorwaardelijke opdrachten bestaan er in veel vormen. De basisvorm is: **if** *expressie* **then** *statement* hierbij wordt 'statement' alleen uitgevoerd wanneer de 'expressie' de waarde true oplevert.

'2<1' is een expressie die de waarde false oplevert, terwijl '2>1' de waarde true oplevert.

Er zijn nog andere vormen voorwaardelijke opdrachten, zoals: **if** *expressie* **then** *statement1* **else** *statement2*. Hier wordt 'statement1' uitgevoerd als de 'expressie' true oplevert, anders wordt 'statement2' uitgevoerd.

Nog een andere vorm van een voorwaardelijke opdracht is: **if** *expressie1* **then** *statement1* **else if** *expressie2* **then** *statement2* **else** *statement3* In deze vorm is slechts de "ELSE IF 'expressie2' THEN 'statement2'"toegevoegd wat maakt dat statement2 wordt uitgevoerd als expressie2 de waarde true oplevert. De rest is wat je er zelf van maakt. (zie vorige vormen).

Iets over syntax:

De basis van een 'if' constructie onder bash is:

if [expressie];

then

code als 'expressie' is true.

fi

### 6.2 Voorbeeld: Basis voorwaardelijke opdracht if .. then

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo waarde van expressie leverde true op
fi
```

De code die zal worden uitgevoerd als de expressie tussen de blokhaken true oplevert, is te vinden achter het 'then' woord en voor het 'fi' woord waarmee het einde van de voorwaardelijk uit te voeren code wordt aangegeven.

### 6.3 Voorbeeld: Voorbeeld basis voorwaardelijke opdracht if .. then ... else

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expressie levert de waarde true op
else
   echo expressie levert de waarde false op
fi
```

### 6.4 Voorbeeld: Voorwaardelijke opdrachten met variabelen

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo expressie levert de waarde true op
else
    echo expressie levert de waarde false op
fi
```

## 7 Loops for, while en until

In deze sectie tref je for, while en until loops aan.

De **for** loop werkt iets anders dan in andere programmeertalen. Eigenlijk laat het je een serie 'woorden' binnen een string herhalen.

De **while** voert een stuk code uit als de controlerende expressie true oplevert, en stopt alleen wanneer het false is (of als er binnen de uitgevoerde code een expliciete break werd gevonden).

De **until** loop is bijna gelijk aan de while loop behalve dat de code wordt uitgevoerd terwijl de controlerende expressie de waarde false oplevert.

Als je het vermoeden hebt dat while en until erg op elkaar lijken, heb je gelijk.

## 7.1 Voorbeeld met for

```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

In de tweede regel, declareren we i als variabele voor de verschillende waarden in $( ls ).

De derde regel zou zonodig langer kunnen zijn, of er zouden meer regels voor kunnen komen voor de done (4).

'done' (4) geeft aan dat de code die de waarde van $i gebruikte beëindigd is en $i een nieuwe waarde aan kan nemen.

Dit script heeft weinig nut, en een nuttiger wijze om gebruik te maken van de for loop zou zijn het in het voorgaande voorbeeld te gebruiken waarbij slechts bepaalde bestanden overeenkomen.

## 7.2 Met C vergelijkende for

fiesh raadde het toevoegen van deze vorm van een loop aan. Het is een for loop meer vergelijkbaar met die van C/perl...

```
#!/bin/bash
for i in `seq 1 10`;
do
        echo $i
done
```

## 7.3 Voorbeeld met while

```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
    echo De teller is $COUNTER
    let COUNTER=COUNTER+1
done
```

Dit script 'emuleert' de welbekende (C, Pascal, perl, enz) 'for' structuur

## 7.4 Until voorbeeld

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
```

```
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

# 8 Functies

Zoals in bijna iedere programmeertaal, kun je functies gebruiken om stukken code op een wat logischer wijze te groeperen of te oefenen in de goddelijke kunst van recursie.

Het declareren van een functie is slecht een kwestie van het schrijven van function mijn_func { mijn_code }.

Het aanroepen van een functie is net als het aanroepen van ieder ander programma, je tikt gewoon de naam ervan in.

## 8.1 Voorbeeld van een functie

```
#!/bin/bash
function quit {
     exit
}
function hello {
     echo Hello!
}
hello
quit
echo foo
```

In de regels 2-4 staat de 'quit' functie. In de regels 5-7 staat de 'hello' functie. Als je er niet geheel zeker van bent wat dit script doet, probeer het dan uit!

Merk op dat functies niet in een specifieke volgorde hoeven te staan.

Bij het uitvoeren van het script, zal je bemerken dat als eerste de functie 'hello' wordt aangeroepen en ten tweede de 'quit' functie en de functie komt nooit bij regel 10.

## 8.2 Voorbeeld van een functie met parameters

```
#!/bin/bash
function quit {
   exit
}
function e {
    echo $1
}
e Hello
e World
quit
echo foo
```

Dit script is bijna identiek aan het voorgaande script. Het belangrijkste verschil is de functie 'e'. Deze functie drukt het eerste argument dat het ontvangt af. Argumenten binnen functies worden op dezelfde wijze behandeld als argumenten die aan het script worden gegeven.

# 9 Gebruikersinterfaces

## 9.1 Het gebruik van select om eenvoudige menu's te maken

```
#!/bin/bash
OPTIONS="Hello Quit"
select opt in $OPTIONS; do
    if [ "$opt" = "Quit" ]; then
     echo klaar
     exit
    elif [ "$opt" = "Hello" ]; then
     echo Hello World
    else
     clear
     echo onjuiste keuze
    fi
done
```

Als je dit script uitvoert, zal je zien dat dit de droom is van een programmeur voor op tekst gebaseerde menu's. Je zal waarschijnlijk bemerken dat het erg lijkt op de 'for' instructie, in plaats van dat ieder 'woord' in $OPTIONS wordt doorlopen, geeft het de gebruiker een prompt.

## 9.2 Gebruik maken van de opdrachtregel

```
#!/bin/bash
if [ -z "$1" ]; then
    echo usage: $0 directory
    exit
fi
SRCD=$1
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

Het zou je duidelijk moeten zijn wat dit script doet. De expressie in de eerste voorwaardelijke opdracht test of het programma een argument ($1) meekreeg en stopt als het dit niet deed, waarbij een kort bericht over het gebruik aan de gebruiker wordt getoond. De rest van het script zou vanaf hier duidelijk moeten zijn.

# 10 Diversen

## 10.1 Inlezen van gebruikersinvoer met read

In veel situaties wil je de gebruiker wellicht vragen om wat invoer, en er zijn verscheidene manieren om dit te bereiken: Dit is één van die manieren:

```
#!/bin/bash
echo Vul alsjeblieft je naam in
read NAME
echo "Hi $NAME!"
```

Een variant daarop zal het volgende voorbeeld verduidelijken waarmee je meerdere waarden met read kan verkrijgen.

```
#!/bin/bash
echo Vul alsjeblieft je voornaam en achternaam in
read FN LN
echo "Hi! $LN, $FN !"
```

## 10.2 Rekenkundige waarde bepalen

Probeer het volgende achter de opdrachtregel (of in een shell):

echo 1 + 1

Als je verwachtte '2' te zien te krijgen, zal je wel teleurgesteld zijn. Wat als je wilt dat BASH van een aantal getallen de waarde bepaalt? Dit is de oplossing:

echo $((1+1))

Hiermee zal een 'logischer' uitvoer worden geproduceerd. Dit is voor het bepalen van de waarde van een rekenkundige expressie. Je kunt dit ook als volgt bereiken:

echo $[1+1]

Als je breuken of meer rekenkunde nodig hebt of dit gewoon wilt, kun je bc gebruiken om de waarde te bepalen van rekenkundige expressies.

als ik op de opdrachtregel ëcho $[3/4]öpgaf, zou het 0 retourneren omdat bash alleen integers gebruikt bij beantwoording. Als je ëcho 3/4|bc -löpgaf, zou het op juiste wijze 0.75 retourneren.

## 10.3 Bash zoeken

UIt een bericht van mike (zie Met dank aan)

je gebruikt altijd #!/bin/bash .. je zou wellicht een voorbeeld

kunnen geven van waar bash is te vinden.

de voorkeur is 'locate bash', maar niet op alle computers is

locate geënstalleerd.

'find ./ -name bash' vanaf de rootdir zal gewoonlijk wel werken.

Aanbevolen te doorzoeken locaties:

ls -l /bin/bash

ls -l /sbin/bash

ls -l /usr/local/bin/bash

ls -l /usr/bin/bash

ls -l /usr/sbin/bash

ls -l /usr/local/sbin/bash

(kan zo geen andere directory's bedenken... ik heb het voorheen

meestal wel op een ander systeem op één van deze

plaatsen kunnen vinden.

Je zou ook nog 'which bash' kunnen proberen.

### 10.4 De return waarde van een programma verkrijgen

Onder Bash, wordt de return waarde van een programma in een speciale variabele, genaamd $?, opgeslagen.

Hiermee wordt geïllustreerd hoe de return waarde van een programma kan worden afgevangen, waarbij ik ervan uit ga dat de directory *dada* niet voorkomt. (Ook dit was een suggestie van mike)

```
#!/bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

### 10.5 Afvangen van de uitvoer van een opdracht

Dit kleine script laat alle tabellen van alle databases zien (ervan uitgaande dat je MySQL hebt geïnstalleerd). Overweeg tevens het wijzigen van de opdracht 'mysql' waarbij een geldige gebruikersnaam en wachtwoord wordt gebruikt.

```
#!/bin/bash
DBS=`mysql -uroot  -e"show databases"`
for b in $DBS ;
do
        mysql -uroot -e"show tables from $b"
done
```

### 10.6 Meerdere bronbestanden

Je kunt met de opdracht source meerdere bestanden gebruiken.

__TO-DO__

## 11 Tabellen

### 11.1 Stringvergelijkings operatoren

(1) s1 = s2

(2) s1 != s2

(3) s1 < s2

(4) s1 > s2

(5) -n s1

(6) -z s1

(1) s1 komt overeen met s2

(2) s1 komt niet overeen met s2

(3) __TO-DO__

(4) __TO-DO__

(5) s1 is niet gelijk aan null (bevat één of meer tekens)

(6) s1 is null

## 11.2 Stringvergelijking voorbeelden

Vergelijken van twee strings.

```
#!/bin/bash
S1='string'
S2='String'
if [ $S1=$S2 ];
then
        echo "S1('$S1') is niet gelijk aan S2('$S2')"
fi
if [ $S1=$S1 ];
then
        echo "S1('$S1') is gelijk aan S1('$S1')"
fi
```

Ik haal hier een opmerking aan vanuit een mail ingezonden door Andreas Beck, verwijzend naar het gebruik van *if [ $1 = $2 ]*.

Dit is niet zo'n goed idee, omdat als $S1 of $S2 leeg is, je een parse error krijgt. x$1=x$2 of "$1"="$2ïs beter.

## 11.3 Rekenkundige operators

+

-

*

/

% (rest)

## 11.4 Rekenkundige relationele operators

-lt (<)

-gt (>)

-le (<=)

-ge (>=)

-eq (==)

-ne (!=)

C programmeurs zouden de operator eenvoudigweg indelen naar de overeenkomstige haakjes.

## 11.5 Handige opdrachten

Deze sectie werd door Kees herschreven (zie met dank aan...)

Een paar van deze opdrachten zijn bijna volledige programmeertalen. Van deze opdrachten wordt alleen de basis uitgelegd. Voor een meer gedetailleerde beschrijving, kun je de man pages van de opdrachten raadplegen.

**sed** (stream editor)

Sed is een niet interactieve editor. In plaats dat een bestand wordt aangepast door de cursor op het scherm te verplaatsen, maak je gebruik van een script met de sed-instructies en geef je deze als argument samen met het te wijzigen bestand op aan sed. Je kunt sed ook als een filter beschrijven. Laten we eens wat voorbeelden bekijken:

```
$sed 's/te_vervangen/vervangen_door/g' /tmp/dummy
```

Sed vervangt de string 'te_vervangen' door de string 'vervangen_door' en leest vanuit het /tmp/dummy bestand. Het resultaat zal naar stdout (normaal gesproken de console) worden gezonden, maar je kunt aan het einde van deze regel ook '> capture' toevoegen waardoor sed de uitvoer naar het bestand 'capture' zal sturen.

```
$sed 12, 18d /tmp/dummy
```

Sed laat alle regels zien behalve de regels 12 tot 18. Het oorspronkelijke bestand wordt door deze opdracht niet aangepast.

**awk** (manipulatie van gegevensbestanden, ophalen en verwerken van tekst)

Er bestaan veel implementaties van de programmeertaal AWK (de bekendste interpreters zijn GNU's gawk en 'new awk' mawk.) Het principe is eenvoudig: AWK scant op een patroon, en voor ieder overeenkomend patroon zal een actie worden uitgevoerd.

Wederom heb ik een dummy-bestand aangemaakt met de volgende regels:

*"test123*

*test*

*tteesstt"*

```
$awk '/test/ {print}' /tmp/dummy
```

test123

test

Het patroon waar AWK naar zoekt, is 'test' en de actie die het uitvoert wanneer het een regel in het bestand /tmp/dummy met de string 'test' vindt, is 'print'.

```
$awk '/test/ {i=i+1} END {print i}' /tmp/dummy
```

3

Wanneer je naar meer patronen zoekt, kun je de tekst tussen de aanhalingstekens beter vervangen door '-f file.awk' zodat je alle patronen en acties in het bestand 'file.awk' kunt plaatsen.

**grep** (druk regels af overeenkomend met een zoekpatroon)

We hebben in de vorige hoofdstuk reeds heel wat grep opdrachten gezien, die de regels laten zien die met een patroon overeenkomen. Maar grep kan meer.

```
$grep "zoek naar dit" /var/log/messages -c
```

12

De string "zoek naar ditïs 12 keer in het bestand /var/log/messages gevonden.

[ok, dit voorbeeld was nep, het bestand /var/log/messages was aangepast :-)]

**wc** (telt regels, woorden en bytes)

In het volgende voorbeeld, zien we dat de uitvoer niet hetgeen is wat we ervan verwachtte. Het dummy bestand, zoals in dit voorbeeld gebruikt, bevat de volgende tekst: *"bash introduction howto test file"*

```
$wc --words --lines --bytes /tmp/dummy
```

2 5 34 /tmp/dummy

De volgorde van de parameters doet er voor wc niet toe. Wc drukt ze altijd in de standaardvolgorde af, dus zoals je kunt zien: <lines><words><bytes><filename>.

**sort** (sorteer regels van tekstbestanden)

Ditmaal bevat het dummy bestand de volgende tekst:

*"b*

*c*

*a"*

```
$sort /tmp/dummy
```

Zo ziet de uitvoer er ongeveer uit:

*a*

*b*

*c*

Opdrachten zouden niet zo eenvoudig moeten zijn :-) **bc** (een calculator programmeertaal)

Met Bc kunnen berekeningen vanaf de opdrachtregel worden gemaakt (invoer vanuit een bestand, niet via omleiding of een pipe maar wel vanuit een gebruikersinterface.) Het volgende demonstreert een aantal opdrachten. Merk op dat ik gebruik maak van de parameter -q om een welkomstbericht te voorkomen.

```
$bc -q
```

*1 == 5*

*0*

*0.05 == 0.05*

*1*

*5 != 5*

*0*

*2 ^ 8*

*256*

*sqrt(9)*

*3*

*while (i != 9) {*

*i = i + 1;*

*print i*

*}*

*123456789*

*quit*

**tput** (initialiseer een terminal of ondervraag een terminfo database)

Een kleine demonstratie van de mogelijkheden van tput:

```
$tput cup 10 4
```

De prompt verschijnt op (y10,x4).

```
$tput reset
```

Maak het scherm schoon en de prompt verschijnt op (y1,x1). Merk op dat (y0,x0) de linkerbovenhoek is.

```
$tput cols
```

*80*

Toont het aantal mogelijke tekens in richting x.

Het is zeer aan te bevelen (op z'n minst) met deze programma's bekend te zijn. Er zijn heel veel van deze kleine programma's die je echte magie op de opdrachtregel laten doen.

[een aantal voorbeelden zijn overgenomen uit de man pages of FAQ's]

# 12 Meer Scripts

## 12.1 Een opdracht toepassen voor alle bestanden in een directory.

## 12.2 Voorbeeld: Een zeer eenvoudig backupscript (iets beter)

```
#!/bin/bash
SRCD="/home/"
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

## 12.3 Bestandshernoemer

```
#!/bin/sh
# renna: hernoem meerdere bestanden op basis van verscheidene
# regels geschreven door felix hudson jan - 2000

# controleer dit bestand eerst op de diverse 'modes'
# als de eerste ($1) voorwaarde overeenkomt dan voeren we dat
# deel van het programma uit en stoppen

# controleer op de prefix voorwaarde
if [ $1 = p ]; then

# nu ontdoen we ons van de mode ($1) variabele en prefix ($2)
  prefix=$2 ; shift ; shift

# een snelle controle of er bestanden werden opgegeven
# als dit niet zo is dan kunnen we maar beter niets doen dan een
# een aantal niet bestaande bestanden te hernoemen!!

  if [$1 = ]; then
     echo "geen bestanden opgegeven"
     exit 0
  fi

# deze for loop verwerkt alle bestanden die we aan het programma
# opgaven
# het hernoemt per opgegeven naam
  for file in $*
    do
    mv ${file} $prefix$file
  done

# we verlaten nu het programma
  exit 0
fi

# controle op het hernoemen van een suffix
# de rest van dit gedeelte is vrijwel gelijk aan de vorige sectie
# kijk alsjeblieft in die notities
```

```
if [ $1 = s ]; then
  suffix=$2 ; shift ; shift

   if [$1 = ]; then
    echo "geen bestanden opgegeven"
   exit 0
   fi

 for file in $*
  do
   mv ${file} $file$suffix
 done

 exit 0
fi

# controleer op de replacement hernoeming
if [ $1 = r ]; then

  shift

# ik nam deze op uit voorzorg dat er geen bestanden beschadigd
# raken als de gebruiker niets opgeeft
# slechts een veiligheidsmaatregel

  if [ $# -lt 3 ] ; then
    echo "usage: renna r [expression] [replacement] files... "
    exit 0
  fi

# verwijder andere informatie
  OLD=$1 ; NEW=$2 ; shift ; shift

# deze for loop verwerkt alle bestanden die we aan het programma
# opgaven, het hernoemt één bestand tegelijkertijd
# door gebruik te maken van het programma 'sed'
# dit is een eenvoudig opdrachtregelprogramma dat standaardinvoer
# verwerkt en een expressie vervangt door een opgegeven string
# hier geven we het de bestandsnaam door (als standaardinvoer) en
# vervangen de nodige tekst

  for file in $*
  do
    new=`echo ${file} | sed s/${OLD}/${NEW}/g`
    mv ${file} $new
  done
exit 0
fi

# als we hier zijn aangekomen dat wil dat zeggen dat niets
# zinnigs aan het programma werd doorgegeven, dus vertellen
# we de gebruiker hoe het te gebruiken
echo "Gebruik;"
echo " renna p [prefix] files.."
echo " renna s [suffix] files.."
```

```
        echo " renna r [expression] [replacement] files.."
        exit 0

        # done!
```

### 12.4 Bestandshernoemer (eenvoudig)

```
#!/bin/bash
# renames.sh
# basis bestandshernoemer

criteria=$1
re_match=$2
replace=$3

for i in $( ls *$criteria* );
do
    src=$i
    tgt=$(echo $i | sed -e "s/$re_match/$replace/")
    mv $src $tgt
done
```

## 13 Wanneer er iets niet goed gaat (debuggen)

### 13.1 Manieren om BASH aan te roepen

Aardig om te doen is het volgende op de eerste regel toe te voegen

```
#!/bin/bash -x
```

Hierdoor zal wat interessante uitvoer worden geproduceerd

## 14 Over het document

Geef me gerust je aanbevelingen/correcties, of wat je dan ook interessant vindt om in dit document terug te vinden. Ik zal het zo spoedig mogelijk bijwerken.

### 14.1 (geen) garantie

Dit document wordt zonder enige garantie geleverd.

### 14.2 Vertalingen

Italiaans: door William Ghelfi (wizzy at tiscalinet.it) *is hier*

Frans: door Laurent Martelli *ontbreekt*

Koreaans: Minseok Park *http://kldp.org*

Koreaans: Chun Hye Jin *unknown*

Spaans: onbekend *http://www.insflug.org*

Ik denk dat er meer vertalingen zijn, maar heb daar geen informatie over, mail het alsjeblieft naar me als je hier informatie over hebt, zodat ik deze sectie kan bijwerken.

## 14.3 Met dank aan

- Mensen die dit document naar andere talen vertaalde (vorige sectie).
- Nathan Hurst voor het opsturen van heel wat correcties.
- Jon Abbott voor het opsturen van opmerkingen over het berekenen van de waarde van rekenkundige expressies.
- Felix Hudson voor het schrijven van het *renna* script
- Kees van den Broek (voor het opsturen van de vele correcties, het herschrijven van de sectie handige opdrachten)
- Mike (pink) deed een aantal suggesties betreft het lokaliseren van bash en het testen van bestanden
- Fiesh deed een aardige suggestie over de sectie loops
- Lion raadde aan melding te maken van een gebruikelijke foutmelding (./hello.sh: Command not found).
- Andreas Beck voor verscheidene correcties en opmerkingen

## 14.4 Historie

Nieuwe vertalingen opgenomen en kleine correcties aangebracht.

De sectie handige opdrachten herschreven door Kees.

Meer correcties en suggesties opgenomen.

Voorbeelden toegevoegd over het vergelijken van strings.

v0.8 de versie gedropt, ik denk dat de datum voldoende is.

v0.7 Meer correcties en een aantal oude TO-DO secties geschreven.

v0.6 Kleine correcties.

v0.5 De sectie over omleiding toegevoegd.

v0.4 verdween van zijn lokatie vanwegen mijn ex-baas en dit document vond een nieuwe plaats op de juiste url: www.linuxdoc.org.

voorgaand: Ik weet het niet meer en maakte geen gebruik van rcs of cvs :(

## 14.5 Meer bronnen

Introductie bash (onder BE) *http://org.laol.net/lamug/beforever/bashtut.htm*

Bourne Shell Programming http://207.213.123.70/book/