# Meer command-line utilities

Mattias Holm & Kristian Rietveld


Universiteit Leiden
The Netherlands

# Vorige week

- Hoe werkt een shell?
- Pipes en redirectie.
- Verscheidene handige utilities.
- Shell scripting.

# Doel

- Meer utilities introduceren die je samen met shell kan gebruiken.
- Uitbreiding van je "gereedschapskist".

# Overzicht

- Awk
- Perl
- Ins en outs van Makefiles
- Makefiles en shell combineren

# Awk

- Awk is een kleine programmeertaal voor het verwerken van (gestructureerde) tekst.
- Elke regel wordt beschouwd als een record en opgesplitst in velden.
- Je kan acties laten uitvoeren als deze voldoen aan een conditie.
- De conditie kan ook worden weggelaten.

# Awk (vervolg)

- Een awk regel:

```
condition { action }
```

# Een eerste voorbeeld

- Print alle regels, geen conditie:

```
awk '{ print }' bestand.txt
```

- Print de regel alleen als deze begint met w:

```
awk '/^w/ { print }' bestand.txt
```

# Meerdere regels

- Uiteraard kan je meerdere regels vormen:

```
awk '/^W/ { print "W regel" } /^G/
{ print "G regel" }' bestand.txt
```

# Velden

- $0 de hele regel
- $1, $2, ..$n record nummer n in de regel
- Gebruik in bash de goede quotes!
  Gebruik anders escape character: \$

```
echo "een twee drie" | awk '{print $2}'
echo "een twee drie" | awk "{print \$2}"
```

# Speciale variabelen

- NR: record number
- NF: aantal velden in een input record
- FNR: record number in huidige file
- FS: field separator die wordt gebruikt (regular expression)
- RS: record separator die wordt gebruikt (regular expression, standaard '\n')

# Speciale condities

- Met de conditie BEGIN kan je een actie maken die aan het begin van het programma wordt uitgevoerd.
- Daarnaast is er ook een conditie END.

```
echo "een;twee;drie" | awk 'BEGIN
{FS=";"} {print $2}'
echo "een;twee;drie" | awk '{print $2}'
```

# Voorbeeld

- Tel het aantal regels dat met W of G begint.

```
/^W/ { w++; }
/^G/ { g++; }

END {
     print "G regels:", g;
     print "W regels:", w;
}
```

# Arrays in Awk

- Arrays in Awk zijn eigenlijk "dictionaries".
- Je kan zelf kiezen wat je als subscript gebruikt (strings, integers, etc.).

```
telefoon["Holm"] = "06-12345678"
telefoon["Kris"] = "06-87654321"
print telefoon["Holm"]
```

# Meer statements

- Awk kan nog veel meer.
- Er zijn if, for, while statements.
- En ook een printf.

```
BEGIN {
      print "ARGC =", ARGC
      for (k = 0; k < ARGC; k++)
            print "ARGV[" k "] = [" ARGV[k] "]"
}

{ telefoon[$1] = $2 }

END {
      for (k in telefoon)
            print k ":", telefoon[k]
}
```

# Perl

- Zeer bekende scripttaal.
- Erg krachtig, vooral voor string manipulatie.
- Wordt vaak bespot als "write-only language".
- Opvolger (Perl 6) laat al een tijdje op zich wachten. Veel mensen proberen ook Python.

# Perl (vervolg)

- Belangrijkste element van Perl is dat regular expressions met de taal zijn verwoven.
- Dit maakt de taal zeer krachtig, maar de code vaak ook moeilijk leesbaar.
- Net als awk kan het script of in een file staan, of direct op de command line.
- Generieker dan Awk.

# Filter loop

- Perl's sterke punt is tekst verwerken.
- De filter loop is een standaard geraamte voor veel Perl programma's.
- Met "-n" krijg je de filter loop kado.

```
while (<>) {
      print;
}

while ($_ = <>) {
      print $_;
}

perl -n -e "print;"
```

# Filter loop (vervolg)

```
while (<>) {
      print if /^foo/;
}

perl -n -e "print if /^foo/;"
```

# Regels tellen

- Variabelen in strings worden net als in bash geinterpreteerd.
- Ook hier gelden de verschillende quotes.

```
while (<>) {
        $w++ if /^W/;
        $g++ if $_ =~ /^G/;
}

print "G regels: $g\n";
print "W regels: $w\n";
```

# Variabelen in Perl

- Het eerste karakter van de variabelenaam geeft het type aan:
  - $ scalar
  - @ array
  - % hash (dictionary)
  - & function

```
$i = 10;
$j = 10 + $i;
@array = ("een", "twee", "drie");
%hash = ("key" => "val", "key2" => 12);
```

# Array / Hash access

- Let op de verschillende haakjes:

```
@array = ("een", "twee", "drie");
%hash = ("key" => "val", "key2" => 12);

print $array[1] . "\n";
print $hash{"key"} . "\n";
```

# Statements

- Alle statements die je verwacht zijn aanwezig: for, foreach, while, do, if.

```
@array = ("een", "twee", "drie");
foreach $i (@array) {
     print "$i\n";
}

%hash = ("key" => "val", "key2" => 12);
foreach $i (keys %hash) {
     print "$hash{$i}\n";
}
```

# Functies

- Perl bevat zeer veel handige functies.
- Voor een overzicht: man perlfunc
- Uitleg: perldoc -f <funcname>
- Voorbeelden:
  - split (string opsplitsen)
  - chomp (newline van string verwijderen)

```
while (<>) {
      chomp;
      split;
      foreach $i (@_) {
            print "$i\n";
      }
}

@_ = split / /, $_;
```

# Regular Expressions

- Je kan direct regular expressions gebruiken met de match operator (/.../).
- Voorbeeld: zelfgemaakte grep.

```
#!/usr/bin/perl -w

if (not defined $ARGV[0]) {
     exit 0;
}

while (<STDIN>) {
     print if /$ARGV[0]/;
}
```

# make

- Make wordt voornamelijk gebruikt om het compileren van software te automatiseren.
- Eigenlijk: het genereren van files gebaseerd op andere files.
- De file wordt alleen gegenereerd als dat nodig is:
  - De file bestaat nog niet, of
  - de gegenereerde file is ouder dan de files waarop het is gebaseerd.

# make rules

- Een Makefile bestaat uit "rules".
- Deze hebben de volgende vorm:

```
target:    dependencies
           one or more commands
```

- Belangrijk! Gebruik tabs en geen spaties.

```
test:      test.c test.h
           gcc -Wall -o test test.c
```

- Invocatie: make test (make <target>)

# Speciale targets

- Het belangrijkste target is "all".
- Hier kan je aangeven welke targets moeten worden gegenereerd als je make aanroept zonder argumenten.

```
all:        test

test:       test.c test.h
            gcc -Wall -o test test.c
```

# Automatic variables

- Er zijn een aantal speciale variabelen die je in make rules kan gebruiken:
  - $@ bevat naam van het target
  - $< bevat naam van de eerste dependency
  - $^ bevat naam van alle dependencies

# Generieke "rules"

- Je wilt niet voor elke C file een aparte make rule schrijven.
- Het is mogelijk om een generieke rule te schrijven die wordt gebruikt als er geen expliciete rule is gevonden.

```
%.o:        %.c
            gcc -Wall -g -c $<
```

# Variabelen

- Zelf kun je ook variabelen introduceren.
- Vaak wordt dit gebruikt om de compiler flags te specificeren.
- Merk op dat de naam tussen haakjes moet staan bij gebruik.

```
CFLAGS = -Wall -g

test:     test.c
          gcc $(CFLAGS) -o test test.c

test:     test.c
          gcc $(CFLAGS) -o $@ $<
```

# Iets completer voorbeeld

```
CFLAGS = -Wall -g
OBJECTS = main.o feature1.o feature2.o

all:        test

test:       $(OBJECTS)
            gcc $(CFLAGS) -o $@ $<

%.o:        %.c
            gcc $(CFLAGS) -c $<
```

# Wildcard operator

- In plaats van zelf alle files op te geven, kan je dit ook automatisch doen.
- Dit kan met de wildcard operator:

```
FILES = $(wildcard %.c)
```

- De meeste grote projecten gebruiken dit *niet*, om te voorkomen dat files onbedoeld in de executable terecht komen.

# Substitution references

- In make kan je een handige operator gebruiken om de extensie van een lijst van files te veranderen.
- Syntax:

```
DEST = $(VAR:A=B)
```

- In variabele VAR, vervang elke "A" met "B".

# Substitution refs (vervolg)

- Dit wordt vaak gebruikt om extensies te wijzigen.

```
OBJECTS = $(FILES:.c=.o)
LATEX_SOURCE = $(FILES:.pdf=.tex)
```

- Zoals we nodig hebben in ons voorbeeld:

```
FILES = $(wildcard %.c)
OBJECTS = $(FILES:.c=.o)
```

# If expressie

- In make kan je if expressies gebruiken.
- Vaak gaat het om string vergelijkingen of kijken of een bepaalde variabele is gedefinineerd.

```
ifeq ($(DEBUG), 1)
      CFLAGS="-Wall -g"
else
      CFLAGS="-Wall -s -O3"
endif
```

- Je kan variabelen definieren op de make command line: *make DEBUG=1*

# Shell gebruiken in make

- Vorige week hebben we de `back ticks` gezien in bash.
- Make biedt een zelfde soort functionaliteit aan met de *"shell"* functie.

```
FILES = $(shell ls *.c)
```

# Shell gebruiken in make

- Onder een target – dependency paar kunnen we al commando's opgeven.
- Deze worden uitgevoerd door de shell die in gebruik is.
- Normaal verschijnen de commando's op stdout. Kan worden onderdrukt met @.

```
FILES = $(shell ls *.awk)
all:
        @echo $(FILES)
```

# Shell gebruiken in make

- Let op als je shell variabelen wilt gebruiken in je commando's!
- Deze moeten worden voorzien van twee dollar tekens.
- Andere variabelen worden geinterpreteerd door make, niet de shell.

```
FILES = $(shell ls *.awk)
all:
	@for i in $(FILES); do \
		echo $$i; \
	done
```

# Handige make opties

- -n print commando's zonder uit te voeren
- -C <dir> draai make in opgegeven directory
- -j N draai N jobs tegelijkertijd (nuttig op multi-core machines)

# Makefiles genereren

- Schijft iedereen dan zelf Makefiles?
- Nee, er zijn weer tools om deze automatisch te genereren:
  - cmake
  - autotools (autoconf, automake)
  - qmake
  - etc.

# Laatste opmerking

- Merk op dat make allerlei programma's uitvoert om het doel te bereiken.
- In feite is make dus een soort shell, en de Makefile een soort shell script.

# Meer informatie

- man awk
- *Classic Shell Scripting*. Robbins, Beebe. O'Reilly.
- man perlfunc (functies)
- man perlre (regular expressions)
- perldoc
- *Programming Perl*. Wall, Christiansen, Orwant. O'Reilly.
- info make

# Opdracht / Practicum

- Practicum in zaal 411.