Een inleiding tot Perl

Staf Wagemakers

26th April 2003

Dit document is afgeleid van PERL TUTORIAL: `http://agora.leeds.ac.uk/nik/Perl/start.html` *door Nik Silver*

Contents

1 "Je eerste Perl programma."

Perl is een scriptingtaal geschreven om vele taken eenvoudiger te maken, de oorspronkelijke bedoeling was om taken welke met de standaard Unix utilities (sed, sh, awk) te moeilijk te realiseren waren te vereenvoudigen, de huidige toepassing van perl gaan veel verder dan de oorspronkelijke doelstelling. Perl een scripting taal noemen klopt misschien niet helemaal, elk perl programma wordt immers door perl eerst gecompileerd voordat het uitgevoerd wordt.

1.1 Een eerste voorbeeld.

Beneden vind je een eenvoudig voorbeeld van een perl programma.

```
#!/usr/bin/perl
#
# Een eenvoudig perl voorbeeld.
#
print "Hallo wereld"; # toon "Hallo wereld
```

1.1.1 Waar is perl?

We zullen het bovenstaande perl programma stap voor stap bekijken. De eerst regel vertelt waar de perl interpreter zich bevindt, het kan zijn dat je dit moet aanpassen naar het path op je systeem (bv. /usr/local/bin/perl). Voor diegene welke al met de standaard Unix scripts gewerkt hebben weten dat dit gebruikelijk is bij de meeste scripting talen onder Un*x.

1.1.2 Commentaar.

Net zoals bij de standaard Un*x shell scripts betekent het # symbool commentaar, alles wat achter # komt wordt door perl genegeerd.

1.1.3 Printen.

Met de instructie `print` kunnen we tekst printen, natuurlijk kunnen we met `print` meer dan gewoon tekst printen zoals variabelen etc. , maar hierover later meer. Instructies worden in perl net zoals bij C gescheiden door een punt-komma ";".

1.2 Perl programma's uitvoeren.

Typ het programma over met een tekst editor naar keuze (vi, emacs, ...), indien je Emacs gebruikt kan je beste Emacs instellen in perl-mode met M-X perl-mode dit vergemakkelijk het schrijven met perl-programma's.

Nadat je het programma hebt ingetypt geeft het executable rechten met "chmod +x programma_naam". Nu kan je het programma uitvoeren met:

./programma of **perl programma**.

Als er iets mis gaat kan je met **perl -w programma** meer foutmeldingen (warnings) krijgen welke hopelijk de oorzaak van het probleem bloot leggen.

Natuurlijk kan je de eerste regel ook vervangen door `#!/usr/bin/perl -w` zodat bij het uitvoeren van je programma altijd warnings krijgt.

Met **perl -d programma** starten we perl in debugger mode, welke je de mogelijk geeft om je programma te debuggen.

Met perl -e '.....' kunnen we vanop de commandolijn een lijn van instructies uitvoeren.

1.3 Programmeer opdrachten!

1. Typ het voorbeeld over en voer het uit.

2 Scalaire variabelen

Scalaire variabelen kortweg ook scalairen genoemd zijn niets meer dan normale variabelen zoals tekst (strings) en cijfers.

2.1 Variabelen namen.

Een scalaire variabele begint in perl altijd met een dollar teken, je moet in perl zoals in andere programmeertalen zoals C of pascal de variabelen niet eerst definieren. Aan een geldige variabele kan je dus meteen een cijfer of een string toekenen

```
$var =9;          #geeft aan $var de waarde 9
$var='9';         #geeft aan $var de string "9"
$var="0009";   #geeft aan $var de string "0009"
```

Bepaalde variabel namen mag je niet gebruiken omdat deze in perl al iets betekennen zo mag je een variabelnaam niet met een cijfer laten beginnen. `$_` is ook een speciale variabele en mag je bijgevolg ook niet gebruiken.
Net zoals C houdt perl rekening met hoofd en kleine letter `$A` is dus niet hetzelfde als `$a`.

2.2 bewerkingen.

Perl gebruikt doorgaans dezelfde syntax als C voor bewerkingen en vergelijkingen.

```
$a = 1 + 2;       # tel 1 en 2 op en bewaar het resultaat in $a
$a = 3 - 4;       # trek 3 van 4 af bewaar het resultaat in $a
$a = 5 * 6;       # vermenigvuldig 5 met 6
$a = 7 / 8;       # deel 7 door 8
$a = 9 ** 10;     # 9 tot de macht 10
$a = 5 % 2;       # $a is de reset van 5 / 2
++$a;             # vermeerdert $a met 1 en geef $a terug
$a++;             # geef $a terug en vermeerder $a met 1
--$a;             # verminder $a met 1 en geef $a terug
$a--;             # geef $a terug en verminder $a met 1
```

Net zoals in C is er wel degelijk een verschil tussen `++$a` en `$a++` zo zal `perl -e '$a=1; print $a++;'` "1" uitprinten en `perl -e '$a=1; print ++$a;'` "2".

2.3 Interpolatie

Als je in perl iets tussen "..." afprint met daar tussen een variabel dan wordt de inhoud van de variabele afgeprint, dit noemen we *interpolatie* . Een eenvoudig voorbeeldje.

```
$a="appelen";
$b="peren";
print $a .  'en' .  $b;
```

De laatste lijn van ons programma kunnen we ook eenvoudigere schrijven door interpolatie te gebruiken:

```
print “$a en $b”;
```

Om van interpolatie gebruikt te maken, moeten de dubbele quotes (“) gebruiken, hadden we in het voorgaande voorbeeld enkele quotes (’) dan zou perl letterlijk “$a en $b’ afgeprint hebben. Wat hier niet de bedoeling was, dit geldt ook voor andere karakters zoals (\n,\t ...) enkel met dubbele quotes worden deze omgezet.

Hopelijk maakt het bovenstaande een het verschil tussen enkele een dubbele quotes duidelijk.

2.3.1 Enkel problemen met interpolatie

Interpolatie is een praktisch iets, het laat toe om gegevens (variabelen) met tekst te mengen, Soms kan dat dit voor perl problemen geven om variabelen van tekst te onderscheiden. Een eenvoudig voorbeeldje:

```
#!/usr/bin/perl
$a="Perl programma";
print "$as \n";
```

Het is de bedoeling om dat de uitvoer van het programma “`Perl programmas`” is, maar in plaatst hiervan drukt het juist een nieuwe lijn karakter (\n). Dit komt omdat voor perl `$as` een variabele is. Welke geen data bevat. Door de variabele te omringen met `{}` kunnen we onderscheid maken tussen de variabelen en de tekst.

2.4 Programmeer Opdrachten.

Schrijf een perl programma dat “Hallo mooie wereld!” afprint elk woord is een perl variabele.

3 Array's

Array's zijn een verzameling van scalaire variabelen.

3.1 Data in array's plaatsen.

Array's worden in perl vooraf gegaan dat een @, op deze manier kunnen we het onderscheid maken tussen scalairen en array's.

```
@fruit = (“appelen”,”peren”,”citroenen”);
@muziek = (“fluit”,”gitaar”);
```

Het bovenstaande voorbeeld kent drie elementen toe aan de array `@fruit` en twee aan de array `@muziek`. We kunnen elementen van een array's oproepen door middel van een index welke van nul begint zoals bij C gebeurt dit door middel van rechten haakjes `[]`.

```
print $fruit[2];
```

Zal citroenen afprinten, merk op dat we `$fruit` schrijven omdat een element van een array een scalaire variabele is.

Indien meerdere woorden aan een array willen toekennen kunnen we ook gebruik maken van `qw( ... ... ...)`. qw staat voor quote words dit bespaart ons wat schrijf werk. Het voorgaande voorbeeld kunnen we dus ook schrijven als:

```
@fruit = qw(appel peer citroen);
@muziek = qw(fluit gitaar);
```

In het voorgaande voorbeeld hebben we gezien hoe we strings aan een array kunnen toekennen op dezelfde manier kunnen hier ook andere array's bij betrekken.

```
@meermuziek = (“harp”, @muziek, “piano”);
@meermuziek = (“harp”,”fluit”,”gitaar”,”piano”);
```

Beide regels geven hetzelfde resultaat.

3.2 Push & pop.

Een array kunnen we ook als een stapel (stack) gebruiken met de push instructie kunnen we elementen aan een array toevoegen met pop halen we de laatst bijgevoegde element er terug uit. Enkele voorbeelden:

```
push(@fruit, “kiwi”);
```

Voegt “kiwi” toe aan het einde van de array `@fruit`. Om meer data in een array te duwen kunnen we de volgende uitdrukkingen gebruiken:

```
push(@fruit,”kiwi”,”banaan”);
push(@fruit,qw(kiwi banaan));
push(@fruit,@meerfruit);
```

De push instructie geeft een de lengte terug van de array.

Om het laatste element uit een array te halen kunnen we de pop instructie gebruiken. Indien de pop instructie op onze originele array @food toepassen:

```
$f = pop(@fruit); # $f is nu "citroen"
```

De pop instructie verwijdert het laatste element uit de array en kent deze aan `$f` toe.

3.3 Array's aan scalairen toekennen.

We kunnen een array ook als een scalaire variabele gebruiken.

```
#!/usr/bin/perl
@fruit = qw (appel peer citroen);
$n=@fruit;
$l="@fruit";
print "$n $#fruit $l\n";
```

In het bovenstaande voorbeeld kennen we de array `@fruit` toe aan de variabele `$n`, `$n` bevat nu het aantal elementen in de array `@fruit` (3). `$#fruit` geeft de index terug van het laatste element in een array in ons geval is dit 2.

Bij `$l="@fruit"` wordt de array fruit omgezet in scalaire `$l` bevat nu de elementen uit de array `@fruit` gescheiden door de "list separator", standaard is dit een spatie we kunnen dit vervangen door een andere karakter door `$"` variabele te wijzigen. `$"` is een van de vele speciale variabelen die in perl gebruikt worden. Als een array in combinatie met scalairen gebruikt wordt gebeurt er dus omzetting. Dit wordt ook wel eens context translatie genoemd.

We kunnen ook meerdere elementen uit een array halen, enkele voorbeelden:

```
($a, $b) = ($c, 4d)          # $a=$c; $b=$d;
($a, $b) = @fruit            # $a=$fruit[0]; $b=$fruit[1];
($a, @watfruit) = @fruit     # $a=$fruit[0]; @watfruit is de rest van @fruit
(@watfruit, $a) = @fruit     # @watfruit is @fruit, $a is niet gedefinieerd.
```

De laatste uitdrukking heeft in feite weinig nut en kan dus het beste vermeden worden.

3.4 Programma argumenten

Argumenten welke aan een programma gegeven worden, worden bewaard in de array `@ARGV`. `$ARGV[0]` bevat het eerste argument `$ARGV[1]` het tweede, enz.

```
#!/usr/bin/perl
print "de argumenten van dit prg zijn @ARGV\n";
print "dit is het eerste argument $ARGV[0]\n";
```

3.5 Array's weergeven.

```
#!/usr/bin/perl
@fruit = qw (appel peer citroen);
print @fruit;
print @fruit .  "\n";
print "@fruit";
```

Probeer het bovenstaande voorbeeld eens uit als oefening en probeer het resultaat te verklaren.

4 Bestanden lezen en schrijven

In dit hoofdstuk bekijken we hoe we bestanden kunnen en schrijven, zoals in C heeft ook standaard perl geen speciale functies om het toetsenbord te lezen ook hier kunnen we gebruik maken van de standaard bestandsfuncties.

4.1 Een voorbeeld.

Het volgende programma leest het password bestand /etc/passwd in een array en druk het af op het scherm.

```
#!/usr/bin/perl
#
# programma opent /etc/passwd, lees het
# print het en sluit het

$file = '/etc/passwd';              # bestandsnaam
open(INFO,$file);                   # opent het bestand
@lines = <INFO>;                    # leest het bestand in een array
close(INFO);                        # sluit het bestand
print @lines;                       # print de array
```

De open functie opent een bestand om te lezen, het eerste argument is de "file handle" (`INFO`) het tweede is de naam van het te openen bestand. Als de bestandsnaam tussen quotes plaatsen (enkele of dubbele) wordt de naam letterlijk genomen. Er wordt dus geen vertaling naar shell string, '~/docs/nog_te_lezen' zal niet juist geïnterpreteerd worden. Als je wel een shell vertaling wilt gebruiken moet je de bestandsnaam tussen <> plaatsen, <~/docs/nog_te_lezen> zal wel juist geïnterpreteerd worden.

Nadat het bestand geopend is kunnen we de "filehandle " gebruiken, in ons voorbeeld lezen we het bestand in een array, het bestand wordt dus volledig in het geheugen gelezen. Hadden we geschreven `$line = <INFO>` hadden we een lijn uit het bestand gelezen.

De "close" functie sluit tenslotte het bestand.

Een bestand kunnen we voor verscheidene doeleinden openen, bij het openen van het bestand moeten aangeven wat we met het bestand willen doen.

```
open(INFO, $file)            # open het bestand om te lezen
open(INFO, ">$file")        # open het bestand om te schrijven, als
                             # het bestand al bestaat wordt het overschreven.
open(INFO,">>$file")        # open het bestand om er data aan toe te voegen
open(INFO,"<$file")         # ook om te lezen
```

4.2 STDIN, STDOUT & STDERR

Perl heeft enkele standaard filehandles stdin (standard input), stdout (standard output) en stderr (standard error).

```
#!/usr/bin/perl
print STDERR "Hoe heet je???  >";
$naam=<STDIN>;
print STDOUT "Dag $naam";
```

Het bovenstaande voorbeeld vraagt je naam nadat je deze ingetypt hebt drukt het deze af. De vraag wordt geschreven naar STDERR de begroeting naar STDOUT. Als het programma uitvoert met

./4_2.pl > outfile

wordt STDOUT geredirect naar outfile, alles wat naar STDOUT geschreven wordt wordt dan naar "outfile" geschreven. Als we `2>&1` aan het opstart commando toevoegen wordt ook STDERR naar "outfile" geschreven.

Merk op dat `$naam` ook de return bevat.

We kunnen ook een array van het toetsenbord lezen, maar dan moeten we de array beeindingen met ^D.

4.3 Programmeer opdrachten

Wijzig het voorbeeld zodat voor elke lijn het # karakter geprint wordt, hiervoor hoeft je enkel een lijn aan het programma toe te voegen en een andere te wijzigen. Tip: gebruik de `$''` variabele.

5 Lussen en sprongen.

In dit hoofdstuk bekijken we een belangrijk onderdeel namelijk "programma controle". Hiermee kunnen we bepaalde zaken een aantal keer uitvoeren, het verloop van programma beïnvloeden enz.

5.1 foreach

Met `foreach` kunnen we de elementen in een array - of een soortgelijke structuur - doorlopen.

```
#!/usr/bin/perl
@fruit=qw ( appel peer citroen );
foreach $stukfruit (@fruit)
  {
     print "$stukfruit\n";
  }
```

In het bovenstaande wordt elk element uit de array `@fruit` een voor een toegekend aan `$stukfruit`. Als je in perl geen variabele op geeft wordt er vaak een standaard variabele gebruikt. Bij scalairen is dit `$_` het bovenstaande programma kunnen we dus ook schrijven als:

```
#!/usr/bin/perl
@fruit=qw ( appel peer citroen );
foreach (@fruit)
  {
     print; print "\n";
  }
```

5.2 Waar of niet-waar?

Voor perl is als wat niet 0 is en niet leeg is (een lege string bv) waar. Het getal 0, de string 0 en een lege string zijn niet-waar.

Perl gebruikt doorgaans dezelfde syntax als C voor vergelijkingen, enkele voorbeelden:

```
$a == $b # is $a gelijk aan $b?  Let op = is geen vergelijking!
$a != $b # is $a numeriek niet gelijk aan $b?
$a eq $b # is de string $a gelijk aan $b?
$a ne $b # is de string $a niet gelijk aan $b?
```

Je kan de logische and, or en not vergelijkingen gebruiken:

```
($a && $b)  # is $a en $b waar?
($a||$b)    # Is $a of $b waar?
!($a)       # Is $a niet-waar?
```

5.3 for

Het volgende voorbeeld telt van 0 tot 9 door een for lus te gebruiken.

```
for ($i=0; $i < 10; ++$i)  # start met $i=0
                           # doe het totdat $i < 10
                           # tel bij $i een bij
  {
     print "$i\n";
  }
```

In feite zijn `for` en `foreach` hetzelfde commando als je in het bovenstaande voorbeeld `for` door `foreach` verandert dan werkt het nog altijd. Het is echter een goede gewoonte om `foreach` te gebruiken om array's te doorlopen en `for` te gebruiken bij de rest dit maakt de code beter leesbaar.

5.4 while & until

Beneden vind je een programma dat data van het toetsenbord leest totdat het juiste passwoord ingetypt is.

```
#!/usr/bin/perl
print "Passwoord?  > ";       # Vraag het voor invoer
$a=<STDIN>;                  # lees invoer
chop $a;                     # verwijder \n
while ($a ne "geheim")       # while als invoer niet-juist is
 {
    print "Passwoord > ";    # Vraag opnieuw
    $a=<STDIN>;              # lees opnieuw
    chop $a;                 # verwijder \n
 }
```

De `while` blok tussen `{}` wordt uitgevoerd totdat `$a` gelijk is aan "geheim", de werking van de `while` lus is vrij duidelijk. In het bovenstaande programma kommen we een nieuwe instructie tegen nl "chop". Bij de invoer wordt de return mee in `$a` geplaatst om onze vergelijking succesvol te laten verlopen dienen we dus het laatste karakter te verwijderen, dit is precies wat het chop commando doet.

Het tegenovergestelde van `while` is de `until` instructie, de `while` instructie voert iets uit zolang de vergelijking waar is. De `until` instructie voert de lus uit totdat iets waar is.

In het bovenstaande programma wordt er twee keer lezen een keer boven de `while` en een keer binnen de lus. Dit kunnen we oplossen door gebruik te maken van `do ... while`.

```
#!/usr/bin/perl
do
{
    print "Passwoord > ";      # Vraag naar invoer
    $a=<STDIN>;               # lees invoer
    chop $a;                  # verwijder \n
}
while ($a ne "geheim")        # while als invoer niet-juist is
```

5.5 if & else

Natuurlijk heeft perl ook een `if` en `else`.

```
if ($a)
{
   print "De string is niet leeg\n";
}
else
{
  print  "De string is leeg\n";
}
```

Zoals we gezien hebben is voor perl een lege string niet-waar. Het bovenstaande voorbeeld illustreert dit. Merk op dat als we aan $a de string "0" zouden toekennen dit voor perl ook niet-waar is.

5.5.1 elsif

Een handige uitbreiding van `if ... else` is `elsif`, een voorbeeld.

```
if (!$a)                                  # de !  operator
{
   print "De string is leeg\n";
}
elsif (length($a) == 1)                   # als 't bovenstaande niet-waar
                                          # is, probeer dit
{
   print "De string bevat 1 karakter\n";
}
elsif (length($a) == 2)                   # als dat ook niet-waar is,
                                          # probeer dit
{
   print "De string bevat 2 karakters\n";
}
else                                      # alles niet waar?  Doe dit
{
   print "De string bevat veel karakters\n";
}
```

De werking van het bovenstaande programma is vrij simpel, mocht de werking toch niet helemaal duidelijk dan zal de commentaar je helpen. Het voorbeeld bevat ook een nieuwe instructie nl. `length` deze geeft de lengte van een string terug.

5.6 next & last

Het volgende voorbeeld demonstreert de next instructie.

```
for ($g=0; $g<10; $g++)
{
    next if ($g % 2);
    print "$g\n";
}
```

Met `$g % 2` berekenen we de rest van het getal bij een twee deling, deze is nul - dus nietwaar - bij even getallen. Bij oneven getallen is de vergelijking waar en wordt de next instructie uitgevoerd. Merk op dat de instructie welke if uitvoert voor de vergelijking staat, dit is bij perl ook toegestaan en maakt het code vaak leesbaarder. Met de next instructie gaan we terug naar het begin van de for lus.

Enkel de even getallen worden dus uitgeprint.

De last instructie beëindigd een lus een voorbeeldje:

```
for ($g=0; $g<10; $g++)
{
    last if ($g > 5 );
    print "$g\n";
}
```

Het bovenstaande voorbeeld beëindigt het programma als $g > 5$.

5.7 Fout afhandeling

De meeste perl functies geven niet-waar terug indien er zich een fout voordoet. Als we bv. een een bestand proberen te open dat niet bestaat als de functie open niet-waar terug geven.

```
#!/usr/bin/perl
print "Geef een bestandsnaam > ";
$bestand=<STDIN>;
chop $bestand;
if (!open(FILE,$bestand)) {
   print "Kan $bestand niet openen:  $!\n";
   exit 1;
   }
print "$bestand geopend\n";
close FILE;
```

Als de functie open niet-waar is wordt er een foutmelding afgedrukt en via de functie exit wordt het programma gestopt. `$!` is en speciale perl variabele welke de foutmelding bevat. Perl zou perl niet zijn als we dit niet korter zoude kunnen schrijven.

```
#!/usr/bin/perl
print "Geef een bestandsnaam > ";
$bestand=<STDIN>;
chop $bestand;
open(FILE,$bestand) || die "Kan $bestand niet openen:  $!\n";
print "$bestand geopend\n";
close FILE;
```

De functie “die” stopt een programma en print een foutmelding af, `||` is een or vergelijking. We hadden evengoed `or` kunnen schrijven.

5.8 Programmeer opdrachten.

1. Pas de oefening uit het voorgaande hoofdstuk aan zodat er regel nummer voor elke lijn worden afgedrukt.

```
1 root:x:0:0:root:/root:/bin/bash
2 daemon:x:1:1:daemon:/usr/sbin:/bin/sh
3 bin:x:2:2:bin:/bin:/bin/sh
```

Gebruik hiervoor de volgende structuur.

```
while ($line = <INFO>
{
...
}
```

Als dit gelukt is pas je het programma aan zodat de lijnnummers afgeprint worden als 001, 002, ..., 009, 010, 011, 012 etc.

2. Pas de vorige oefening aan zodat er een bestandsnaam gevraagd wordt, als het bestand niet bestaat wordt er een foutmelding afgedrukt.

3. Pas 2 aan zodat lege lijnen niet afgedrukt worden, de teller telt lege lijnen ook niet.

6 Patroon vergelijkingen

Een van de meest gebruikte mogelijkheden van perl zijn de krachtige string manipulaties. Het hart hiervan zijn de regular expressions welke in vele andere Unix tools (vi, sed, ...) tegenkomen.

6.1 Een eerste voorbeeld

```
#!/usr/bin/perl
print "Typ een zin > "; $zin=<STDIN>;
chop $zin;
if ($zin =~ /Het/)
 {
   print "Je zin bevat \"Het\"\n";
   print "Dit is het deel voor \"Het\" $`\n";
   print "Dit hebben we gevonden $&";
   print "Dit is het deel achter \"Het\" $'\n";
 }
```

In het bovenstaande voorbeeld komt een eerste reguliere expressie voor met (`$zin =~ /Het/`) wordt er gekeken of er in de zin die je typte het woord "Het" voorkomt, de vergelijking houdt rekening met hoofd en kleine letters. In dit voorbeeld worden ook 2 nieuwe speciale variabelen gebruikt de `$`` bevat alles voor het gevonden patroon `$'` wat achter het gevonden patroon voorkomt. `$&` bevat datgene dat met de vergelijking overeenkomt.

Als we geen variabele op geven wordt er vergeleken met de string `$_` De vergelijking ziet er dan als volgt uit:

```
if (/Het/)
 {
   print "$_ bevat \"Het\"";
  }
```

Merk op dat we "/" mogen vervangen door een ander karakter.

6.2 Reguliere expressies

In regulieren expressies worden vele speciale karakters gebruikt, deze speciale karakters zijn de bouwstenen en de kracht van reguliere expressies. Deze kunnen er vaak erg ingewikkeld uitzien, het vraagt ook heel training om ermee vertrouwd te geraken. In de volgende tabel vind je enkele speciale karakters en hun betekenis:

.	Elk enkel karakter buiten \n
^	Het begin van een lijn of string
$	Het einde van een string
*	Nul of meer van het laatste karakter
+	Een of meer van het laatste karakter
?	Nul of 1 van het laatste karakter

Enkele voorbeelden:

/t.e/	komt overeen met “thee”, “teen”, “toen”, “tien”. Maar niet met “te”, “talen”
/^f/	f aan het begin van een regel
/^ftp/	ftp aan het begin een regel
/n$/	n aan het einde van een regel
/en$/	en aan het einde van een regel
/ond*/	on gevolgd door nul of meer d’s komt overeen met “on”, “ond”, “hond”, “hondd”, enz.
/.*/	elke string zonder een nieuwe lijn.
/^$/	een lege regel

Het volgende programma demonstreert het verschil tussen * en ?

```
#!/usr/bin/perl
$_=<STDIN>;
chop;
if (/^c?ola/) {
   print "/^c?ola/ is True\n"
   }
if (/^c*ola/) {
   print "/^c*ola/ is True\n"
   }
```

Probeer het bovenstaande programma eens uit, en probeer de resultaten te verklaren.

De rechte haakjes betekenen in een reguliere expressie “of”, als een string 1 van karakters binnen de rechte haakje bevat is de uitdrukking waar. Een min teken binnen de rechte haakjes betekent “tussen” en een ^ niet. Enkele voorbeelden:

[qjk]	q of j of k
[^qjk]	niet q, j, k mogen niet voorkomen
[a-z]	elke kleine letter
[^a-z]	geen kleine letters
[a-zA-z]	elke letter
[a-z]+	elke niet lege sequentie van kleine letters

Een rechte streep | komt overeen met een logische “or”, de ronde haakjes gebruiken we om zaken te groeperen.

bus|tram	bus of tram
(pa|da)den	padeb of daden
(da)+	da of dada, dadada ...

En om het lijstje compleet te maken:

\n	nieuwe regel
\t	een tab
\w	Elk alfanumeriek karakter. Is hetzelfde als [a-zA-Z0-9_]
\W	Elk niet alfanumeriek karakter. Is hetzelfde als [^a-zA-Z0-9_]
\d	Elk cijfer. Is hetzelfde als [0-9]
\D	Elk niet-cijfer Is hetzelfde als [^0-9]
\s	Elke spatie karakter: spatie, tab, nieuwe regel, ...
\S	Elk niet spatie karakter
\b	Een woord (boundary) scheiding (alles tussen \w en \W)
\B	Geen woordscheiding

Om speciale karakters in een reguliere expressie te kunnen gebruiken, dienen we hiervoor een "\" gebruiken om bv een letterlijk sterretje te gebruiken dienen we dienen"*" te schrijven.

Reguliere expressies komen in het begin verwarrend over, zoals eerder gezegd worden ze is meerdere programmeertalen gebruiken eenmaal dat je geleerd hebt ze te gebruiken zijn dus universeel inzetbaar. Het belangrijk om je tijd te nemen om je patronen op te bouwen.

Nog enkele voorbeelden:

[01]	0 of 1	
\/0	gedeeld door nul "/0"	
\/\s0	"/ 0" gedeeld door, elk spatie teken, nul	
\/ *0	gedeeld door, mogelijk enkele spaties, nul	("/0", "/ 0", "/ 0", ...)
\/\s*0	gedeeld door, mogelijk enkele spatie tekens, nul	
\/\s*0\.0*	als het vorige, maar met een decimaal punt met mogelijke enkele nullen er achter	("/0.", "/0.00

6.3 Programmeer Opdrachten.

In de vorige opdrachten heb je een programma gemaakt dat niet lege lijnen telde. Pas het aan zodat het alleen lijn telt waarin

* de letter "k"

* het woord "het" of "Het", tip gebruik \b om woord "scheidingen" te detecteren.

voorkomt. Probeer de $_ te gebruiken om de match operator ~ te vermijden.

7 Vervanging

Reguliere expressies kunnen we niet alleen gebruiken in vergelijking, maar om delen in een string te vervangen.

7.1 Een voorbeeld

Het volgende voor vervangt "antwerpen" in "Antwerpen".

```
$zin = ~ s/antwerpen/Antwerpen/
```

De syntax komt overeen met die van patroon vergelijkingen, "s" staat voor het Engelse substitute. Net als bij patroonvergelijkingen kunnen ook $_ gebruiken als we de "match operator" ~ weglaten.

```
s/antwerpen/Antwerpen/
```

Het resultaat van een substituut is gelijk aan het aantal vervangingen dus 0 of 1 in ons voorbeeld, we kunnen dit resultaat gebruiken bij bv. een if. Merk op dat het bovenstaande voorbeeld enkel het eerste patroon vervangt, als er in een zin twee keer "antwerpen" voorkomt zal enkel de eerste vervangen worden.

7.2 Opties

Als we alle patronen in een vervangen willen vervangen kunnen we de "g" (global) optie gebruiken hiervoor moeten we enkel "g" aan het einde van een substituut toevoegen. Het volgende voorbeeld vervangt alle "antwerpen" door "Antwerpen" in $_.

```
s/antwerpen/Antwerpen/g
```

Ook hier geeft de substitute het aantal vervangingen terug, 0 (niet-waar) als "antwerpen" niet in $_ voorkomt of een waarde groter dan 0 (waar) bij een vervanging.

Als we antwerpen willen vervangen met hoofdletters op een willekeurige plaats aNtwerpen, ANTwerpen, AnTwErPeN, ... kunnen we dit schrijven als

```
s/[Aa][Nn][Tt][Ww][Ee][Rr][Pp][Ee][Nn]/Antwerpen/g
```

Wat veel schrijfwerk is, we kunnen dit korter schrijven door de optie "i" (ignore) te gebruiken. Met de opties "i" wordt er geen rekening gehouden met hoofd en kleine letters in de vergelijking.

```
s/antwerpen/Antwerpen/gi
```

Natuurlijk kunnen we "i" optie ook alleen gebruiken zonder de "g" of een andere optie. De "i" optie is ook bruikbaar bij een normale patroon vergelijking /../.

7.3 Herinneringen

Als we ronde haakjes in een vergelijking worden waardes van het resultaat bewaard in $1,$2,$3 ...

```
#!/usr/bin/perl
$_="03/457.20.61";
if (/(\d+)\/(\d+)\.(\d+)\.(\d+)/)
   {
      print "Het zonenummer is $1\n";
   }
```

In het bovenstaande voorbeeld wordt "herinnering" gebruikt om het zone nummer uit een geldig telefoonnummer te halen.

Herinneringen zijn ook handig binnen vervangingen hiervoor kunnen we de speciale karakters \1, ...\9 gebruiken.

```
#!/usr/bin/perl
$_="Dit Is een Test"
s/([A-Z])/:\1:/g;
print "$_\n";
```

Het bovenstaande voorbeeld omringt alle hoofdletters met : in een zin.

We kunnen herinneringen ook in een normale patroonvergelijking gebruiken.

```
#!/usr/bin/perl
$_="twee maal twee is vier";
if(/(\b\w+\b).*\1/) {
  print "$1 komt meerdere keren voor in $_\n";
  }
```

\b komt overeen met een woordscheiding (boundary), \b\w+\b komt overeen met elke group van alfanumerieke karakters tussen woordscheidingen of simpelere gezegd een woord. Deze uitdrukking wordt dan toegekend aan de eerste herhindering variabele door ze tussen ronde haakjes te plaatsen, met .*\1 testen we of de herinnerings variabele nog een keer in de zin voorkomt.

De volgende uitdrukking verwisselt het eerste karakter met het laatste in een zin.

```
s/^(.)(.*)(.)$/\3\2\1/;
```

^ en $ komen overeen met het begin en het einde van een string, ^(.) komt dus overeen met het eerste karakter in een string (.)$ met het laatste het resultaat van beiden uitdrukkingen wordt herinnerd doordat ze tussen ronde haken staan. In het bovenstaande voorbeeld komt \1 dus overeen met het eerste karakter, \3 met het laatste en \2 met alles tussen het eerste en laatste karakter. De hele lijn wordt dan vervangen door \3\2\1.

7.4 Interpolatie binnen reguliere expressies

Als we een scalaire binnen een reguliere uitdrukking gebruiken dan zal er interpolatie gebruikt worden.

```
$zoek="het";
$_="het is het";
s/$zoek/xxx/g;
print;
```

Het bovenstaande voorbeeld vervangt $zoek in $_.

Zoals we al gezien hebben kunnen er zich bij interpolatie problemen voordoen als we bv. opzoek willen gaan naar $zoek gevolgd door "gene" ("hetgene" in ons voorbeeld) dan kunnen we niet schrijven $zoekgene vermits dit voor perl de variabele $zoekgene is. Ook hier kunnen we dit oplossen door ${..}

```
s/${zoek}gene/xxx/g;
```

7.5 Translatie

De `tr` functie dient voor karakter per karakter translatie, de syntax van tr in perl is hetzelfde als die van het Unix commando tr. Het volgende voorbeeld vervangt elke "a" door "e", elke "b" door "d" en elke "c" door een "f" in variabele zin. Ook de tr functie retourneert het aantal gebeurde vervangingen.

```
$zin=~ tr/abc/edf/;
```

De speciale karakters van regural expressions zijn niet van toepassing op het `tr` commando, zo heeft het sterretje geen speciale betekenis in tr. Het volgende voorbeeld telt het aantal sterretjes in de variabele $zin en bewaart aantal in $teller.

```
$teller=($zin=~ tr/*/*/);
```

Het min teken "-" betekent nog steeds "tussen", a-z komt dus nog altijd overeen met alle kleine letters. Het volgende voorbeeld is een typische toepassing van het tr commando het vervangt alle grote letters door kleine in de variabele $_.

```
tr/A-Z/a-z/;
```

7.6 Programmeer opdrachten.

1. Je huidige programma dat je in de vorige oefeningen gemaakt hebt telt het aantal lijnen in een bestand welke een bepaald woord bevat. Pas het aan zodat het lijnen telt met dubbele letters. Als dit gelukt is pas het dan aan zodat het de dubbele letters omringd door ronde haakjes. Het is dus de bedoeling dat je programma een lijn als deze produceert.

`023 Niet a(ll)e doelen heiligen a(ll)e mi(dd)elen.`

2. Pas het programma aan zodat het een bepaald woord omringt, het woord wordt aan het programma doorgeven als een argument.

8 Split, Join & substr

Split is een veel gebruikte perl functie, het laat je toe om een string op te splitsen en ze in een array te plaatsen. Met join kan je een array samenvoegen tot een enkele string. Substr laat je toe delen uit een string in een andere string plaatsen.

8.1 Split

De split functie gebruikt reguliere expressies en werkt standaard - zoals gewoonlijk - met de $_ variabele tenzij anders gedefinieerd. De split functie wordt als volgt gebruikt.

```
$info="Jansen:Natascha:Actrice:Nieuwsstraat, 14";
@personal=split(/:/,$info);
```

Wat hetzelfde resultaat geeft als

```
@personal= (“Jansen”,”Natascha”,”Actrice”,”Nieuwsstraat, 14”);
```

Als we $_ willen gebruiken dan kunnen we het laatste argument weglaten.

```
@personal=split(/:/);
```

Als de velden gescheiden worden door een willekeurig aantal : dan kunnen we de volgende reguliere uitdrukking gebruiken.

```
$_="Jacobs:Kristien::assistente:::Berglaan 17";
@personal=split(/:+/);
```

Wat overeen komt met

```
@personal=(“Jacobs”,”Kristien”,”assistente”,”Berglaan 17”);
```

Maar dit

```
$_="Jacobs:Kristien::assistente:::Berglaan 17";
@personal=split(/:/);
```

geeft het volgende resultaat

```
@personal=(“Jacobs”,”Kristien”,””,”assistente”,””,””,”Berglaan 17”);
```

De volgende voorbeelden splitsen een woord in karakters, een zin in woorden en een paragraaf in zinnen.

```
@karakters=split(//,$woord);
@woorden=split(/ /,$zin);
@zinnen=split(/\./,$paragraaf);
```

8.2 Join

"join" doet juist het tegenovergestelde van "split" het voegt de elementen van een array samen tot een scalaire.

```
#!/usr/bin/perl -w
@personal=("Jacobs","Kristien","assistente","Berglaan 17");
$personal=join(':',@personal);
print "$personal\n";
```

8.3 Substr

Met substr kunnen we delen uit een string in een andere string plaatsen.

```
substr("Er was eens",3,5);          # geeft "was e" terug
substr("Er was eens",3);            # geeft "was eens" terug
substr("Er was eens",-4,2);         # geeft "ee" terug
```

Het eerste argument van substr is de string welke we willen gebruiken, het tweede argument is positie vanaf waar we in de string beginnen als dit negatief is beginnen we van het einde te tellen. Het derde en laatste argument is het aantal karakters dat we willen.

8.4 Programmeer opdrachten

Schrijf een programma dat /etc/passwd lijn per lijn inleest, en per gebruiker de loginnaam en homedirectory afdrukt.

9 Hashes

Net als array's zijn hashes niets meer dan een verzameling van scalairen, het verschil is dat we bij array's we de gegevens altijd met een nummer benaderen. Bij hashes kunnen we de gegevens kunnen benaderen met string als index.

9.1 Een hash aanmaken.

Een hash in perl herkennen we aan %, als we data uit een hash willen halen gebruiken we {}. Een voorbeeld: stel we willen een lijst maken met mensen hun naam en hun leeftijd.

```
%leeftijd=("kristien",29,
           "vannessa",23,
           "lut",32,
           "annelies","ongeveer 27",
           "tammara",35);
```

Zoals je kan zien komt het aanmaken van een hash grotendeels overeen met het aanmaken van een array, alleen schrijven % in plaats van @.

9.2 Data uit een hash halen

Data uit een hash halen is ook analoog als bij een array alleen schrijven { }i.p.v. [ ].

```
$leeftijd{"kristien"};      # geeft 29 terug
$leeftijd{"vannessa"};      # geeft 23 terug
$leeftijd{"lut"};           # geeft 32 terug
$leeftijd{"annelies"};      # geeft "ongeveer 27" terug
$leeftijd{"tammara"};       # geeft 35 terug
```

Een hash kunnen we ook converteren naar een array's

```
@info=%leeftijd             # @info bevat nu 10 elementen
$info[5]                    # geeft 29 terug
%nog=@info                  # %nog bevat dezelfde data als %leeftijd
```

Een hash wordt door perl intern alfabetisch opgeslagen als we een hash omzetten naar is deze ook alfabetisch gesorteerd. Als we een array omzetten naar een hash dan worden de even elementen als "key" gebruikt.

9.3 Een hash doorlopen

Als we een hash doorlopen met foreach dan wordt elk element uit de hash gehaald ongeacht of het nu een key of value is.

```
print "$_\n" foreach (%leeftijd)
```

geeft dus de volgende uitvoer:

annelies
ongeveer 27
kristien
29
lut
32
tammara
35
vannessa
23

Merk op dat in het voorbeeld de print instructie voor de voorwaarde staat, in perl kunnen we dit ook zo schrijven om de code leesbaarder te maken. Dit werkt trouwens ook bij bv. een if vergelijking.

Meestal willen echter alleen de keys doorlopen of alleen de values, met keys %hash krijgen we een array terug welke enkel de keys bevat welke we eventueel met foreach kunnen doorlopen. values doet juist hetzelfde maar dan voor de values.

```
foreach $persoon (keys %leeftijd)
{
  print "$persoon is $leeftijd{$persoon}\n";
}
foreach $jaar (values %leeftijd)
{
  print "iemand is $jaar\n";
}
```

9.4 Programmeer opdrachten

Als je een perl programma of een ander script/programma start onder Unix dan zijn er een aantal omgevings variabelen gezet welke we kunnen gebruiken in ons programma. Zo bevat $USER je loginnaam $DISPLAY je X display naar waar je programma data moet sturen enz. Bij cgi scripts bevatten deze omgevings variabelen veel bruikbare informatie.

Bij perl worden deze in de hash %ENV bijgehouden waarvan de keys de variabelnamen bevatten. Probeer het volgende in een perl programma te gebruiken:

```
print “Je bent $ENV{’USER’} en je gebruikt $ENV{’DISPLAY’}\n”;
```

10 Functies

Natuurlijk kunnen we in perl ook zelf functies maken, in perl noemen we dit subroutines. Subroutines mogen eender waar in het programma geplaatst worden, maar het is een goede gewoonte ze allemaal aan het begin of het einde van je programma te plaatsen.

10.1 Een subroutine aanmaken

Een subroutine ziet er als volgt uit:

```
#!/usr/bin/perl
sub mysubroutine
{
  print "Dit is niet echt een intersante subroutine\n";
  print "Hij print juist wat tekst\n";
}
mysubroutine();
```

Een subroutine begin altijd met "sub" gevolgt door de subroutine naam de instructie welke van de subroutine deel uitmaken worden gegroepeerd binnen { }. Een subroutine kunnen we op twee manieren aanroepen:

```
&mysubroutine();
```

of

```
mysubroutine();
```

De eerste methode werkt overal, de twee kunnen we enkel gebruiken als de subroutine al in de code gedeclareerd is.

10.2 Parameters

In het vorige voorbeeld kunnen we parameter meegeven aan onze subroutine maar deze worden verder in onze subroutine niet gebruikt. Parameters plaatsen tussen ronde haakjes.

```
&mysubroutine;          # subroutine aanroepen zonder parameters
&mysubroutine($_);      # subroutine aanroepen met 1 parameter
&mysubroutine(1+2,$_)   #subroutine aanroepen met 2 parameters
```

Binnen de subroutine zijn de parameter beschikbaar in @_ array, deze heeft niets te maken met $_ scalaire, de volgende subroutine print alle argumenten af.

```
sub printargs
{
    print "@_\n";
}
&printargs("Hallo","wereld");           # print "Hallo wereld"
&printargs("Het","is","mooi","weer."); # print "Het is mooi weer"
```

Zoals bij elke array hebben we toegang tot de individuele elementen via $_[...]

```
sub telop
{
if ($#_ != 1) {
   print "Het aantal parameters klopt niet\n";
}
else {
   print $_[0]+$_[1];
   }
}
telop(1,2);
```

Bovenstaande subroutine tel twee getallen op, eerst wordt er gecontroleerd of het aantal parameter klopt. Zoals we in het hoofdstuk array's gezien hebben komt $#_ overeen met de index van het laatste element van array @_.

Toch nog even benadrukken dat $_[0], $_[1] niets gemeen hebben met scalaire $_.

10.3 Waardes teruggeven

De laatst geëvalueerde waarde wordt door de subroutine teruggeven. Eventueel kunnen we ook de return instructie gebruiken om de een waarde terug te geven.

```
sub telop
{
   $_[0]+$_[1];
}
print telop(1,2);
```

10.4 Locale variabelen

De @_ variabele is binnen een subroutine een locale variabele de $_[0], $_[1], enz zijn dit natuurlijk ook vermits het elementen zijn van @_. Andere variabelen kunnen we ook lokaal maken, dit doen we door het local statement te gebruiken. De volgende subroutine test of de ene string in een andere string voorkomt.

```
sub inside
{
    local($a, $b);                    # Maak locale variabelen aan
    ($a,$b)=($_[0],$_[1]);            # Ken waardes toe
    $a=~s/ //g;                       # Verwijder spaties
    $b=~s/ //g;                       # van de locale variabelen
    ($a=~ /$b/ || $b=~ /$a/)          # komt $b in $a voor
                                      # of komt $a in $b voor
}
print &inside("Hallo","Hallo daar");
```

De twee eerste lijnen van de subroutine hadden we ook kunnen vervangen door:

```
local($a,$b)=($_[0],$_[1]);
```

11 Dank aan...

- Rene van Leeuwen <leeuwen.rene.van at planet.nl> : voor hel melden van het wegvalen van "^" bij de html versie van dit document.

12 Gebruikte software

- Debian GNU/Linux
- LyX
- Latex
- Ghostscript
- perl
- vim
- emacs