Standard– und Programmiertools für Unix

Stephan Thesing

3. `perl`: Practical Extraction and Report Language

Verbindet Features: `sed`, `awk`, shell, **C**

Vielfältiges Pattern-Matching eingebaut

Interpretiert: Rapid-Prototyping

Weit verbreitet

Ursprung & Verwendungen:

- Systemverwalter-Aufgaben (Log-Files, etc.)
- Ergänzt in `awk`, `sed`, Shell fehlende Features
- WWW: CGI-Skripte
- Viele Libraries verfügbar: Viele Utilities sind entstanden

write-once read-never!!

Aufbau an **C** angelehnt

Integrierte Shell-, `awk`-, `sed`- Features: Pattern, Expansion

Datentypen: skalar, Vektor, assoziativer Vektor, Liste

```
$foo=1;
$bar="Ein String"; $bar=42;
$foobar='Noch ein String';
$arr[$foo]="Ein Vektor";
# Der Vektor selbst
@foo=@arr;
print "Dies", " ist", " eine", " Liste\n";
#Das folgende auch
print 1,2,3,4,5;
```



```
$barr{"schluessel"}="Ein assoziativer Vektor";
# Der ass. Vektor selbst
print %barr;
%xyz=%barr;
```

Files werden über Filehandle angesprochen:

```
open(FH,">/tmp/xx");
print FH "Hello world\n";
$zeile=<FH>;
```

`STDIN`, `STDOUT` und `STDERR` sind vordefinierte Filehandle

Skalare:

Zahlen: `42`, `3.14159265` In `perl` werden Zahlen immer als Fließkommmazahlen gespeichert.

Strings: Ohne/mit Ersetzung, Sammlungen....

```
$foo=42;
# perl konvertiert:
$bar=4.56*"4.22";
# ergibt 19.2432 fuer $bar
$as='Ohne Ersetzung';
$as="Mit Ersetzung: $bar";
$as=<<EOM
Wird
gesammelt
EOM
```


Programmausgaben als Strings:

```
$as=`ls -l /etc`
$bs=<<`EOM`
ls -l /etc
ls /tmp
EOM
```

Vektoren und Listen:

```
@foo=(1,2,3,4);
%bar=("S1","W1","S2","W2");
print $foo[2];
print $bar{"S1"};
```

Files werden über Filehandle manipuliert.

Öffnen:

```
open(OUT,">/tmp/xx");
open(IN,"</tmp/xx");
open(APP,">>/tmp/xx");
open(INOUT","+</tmp/xx");
open(PIPE,"ls -l |");
open(PIPE,"|mail $user");
```

Schliessen: `close(INOUT);`

Ausgabe:

```
print INOUT "Ins File schreiben";
select(INOUT); print "Ins File schreiben";
```


Aus File lesen:

```
$foobar=<INOUT>;
$foobar=<STDIN>;
$foobar=<>;
@foo=<STDIN>;
```

Achtung: skalar und Vektor lesen unterschiedlich!!

Viele, viele Operatoren:

```
$foo=1+3;
$sinx=sin($x);
if ($anumber < 99.34) ...
# Stringvergleiche:
if ($astring lt "Ein String") ...
$b=$a++;
$astring="String1"."String2";
$c=($a==2)?4:100;
```



```
printf "Dies sind %d Bytes\n", $bytes;
print "Dies sind $bytes Bytes\n";
open (FH,"</etc/passwd") || die "Kann passwd nicht oeffnen";
$foo=<FH>;
chop($foo);
$len=length($foo);
$arg1=shift @ARGV;
if ( -f $filename ) ....
$res=join(":",@ARGV);
@felder=split(/\s+/,"Ein String mit fuenf Feldern",4);
```

Viele Funktionen/Operatoren nehmen `$_` oder `@_` als implizite Argumente

Z.B.: `<STDIN>` steht für `$_=<STDIN>`

`chop` steht für `chop($_)`

Warnungen:

- Implizite Argumente machen das Lesen nicht einfacher
- Man übersieht Änderungen impliziter Variablen
- Man muß Klammerung beachten, da Ergebnis sonst manchmal verwirrend

Pattern: `awk` + mehr

Bekannt: `*`, `+`, `.`, `[]`

Neu:

? 0/1-malig: `items?`

{n,m} n- bis m-mal: `[0-9]{1,10}`

| Alternative: `for|foreach`

() Klammerung: `(a|b)*`

Spezielle Escape-Sequenzen:

```
#Ziffern (== [0-9])
\d
# Nicht Ziffer (== [^0-9])
\D
# Wortzeichen (Buchstabe, _)
\w \W
# Leerraum
\s \S
# Wortuebergaenge
\b \B
```


Beispiele:

```
# Bruchzahl
\d*\.\d+
# Erstes Wort einer Zeile
^(\w*)\W
# Letztes Wort einer Zeile
\W(\w*)$
# Filename: Basis + Extension
^(.*)\.([^.]*)$
```

Teilpattern (geklammert) stehen wieder als \\n zur Verfügung:

```
(\w*)\s+\1
(\w*)\W+(\w*)\W+\2\W+\1
```

Matching, Substitution und Translation in `perl`

```
$foo="Die UdS ist schoen";

if ($foo =~ /UdS\s*ist (\S*)\s/) { print $1; }

$foo =~ s/UdS/Uni des Saarlandes/;

$foo =~ s/UdS/Uni des Saarlandes/g;

$hexzahl =~ tr/abcdef/ABCDEF/;

while (<STDIN>) { s/UdS/Uni des Saarlandes/g; print $_; }
```


Kommandos

Ausdrücke sind Kommandos

`perl` kennt *Guards*

```
$foo=99;

$bar="nix" if $foo==99;
$bar="oh" unless $foo==0;
$bar.="x" while $foo-->0;
$bar++ until $foo--<100;
```

Achtung: Bedingung *vor* Ausdrucksauswertung geprüft.

`perl` kennt einen `do` Block. Nützlich in Verbindung mit Guards:

```
do { $foo++; $bar--; } while $bar>0;
do { $foo++; $bar--; } if $foobar eq "nix";
do { $bar++; print $bar."\n";} until $bar==0;
```

Achtung: Semantik der Guards verändert sich bei `until` und `while`

Man kann auch so einen Block einführen:

```
{ $foo="Dies ist ein Block";
  $bar=$foo."Schoen, gell?";
}
```


Schleifen, Conditional:

```
if ($foo==1) { print "\$foo ist 1\n"; }
if ($foo <100) { $bar++; } else { print "Wertefehler\n";}
if ($bar eq "END") { $theend++; } elsif ($bar eq "START") {
  $thestart++;} else {$dontknow++;}

while ($bar<100) { $foo++; $bar--;}
until ($bar==100) {$foo--; $bar++;}
for ($i=1; $i<=$#ARGV; $i++) { print $ARGV[i]."\n"; }
foreach $v (keys %names) { print $names{$v};}
```

Schleifenabbruch, etc.

```
while ($i!=19) { $foo++; } continue {$i++;}
ALABEL:
 while ($i<100) {
   if ($foo==10) { last ALABEL;}
 }
for ($i=0; $i<10; $i++) { next if $bar<9;}
until ($i==100) { redo unless ($j++!=$i); }
{ print "Ein Block"; last if $foo==1; } continue { print "Weg\n"; }
```


Beispiele

```
while ($_=<STDIN>) {
 chop $_;
 print $_."\n";
}

while (<STDIN>) {
 chop;
 print $_."\n";
}

while (<>) {
 chop;
 print $_."\n";
}
```

```
while (<>) {
 s/UdS/Uni des Saarlandes/g;
 print;
}

while (<STDIN>) {
 if (/^\d/) { $ziffer++;}
}
printf "%d Ziffernzeilen\n", $ziffer;
```


Zusammenfassung:

- `perl` bietet Features von `sed`, `awk` und der Shell
- Mächtiges Pattern-Matching eingebaut
- Viele Konstrukte
- Manchmal unübersichtliche Semantik
- Gut geeignet für *kleine* Utilities