3 Die Skriptsprache perl

Grundlage: Das **Buch** „Learning Perl“, 3. Auflage, von O’Reilly.

3.1 Überblick über das perl-Kapitel

Überblick über das perl-Kapitel

- perl statt sed von der Kommandozeile aus
- Ein einfaches perl-Programm
- Skalare Daten
- Listen und Felder
- Reguläre Ausdrücke von Perl
- Grundlagen der Ein- und Ausgabe
- Assoziative Felder
- Unterprogramme
- Weitere Kontrollstrukturen
- Datei-Handles und Datei-Tests
- Prozeß-Management

Kurs Unix-Tools, WS 01/02, Jan Bredereke 14

3.2 perl statt sed von der Kommandozeile aus

„There is more than one way to do it.“ (Motto von perl)

Beispiel eines der vorigen sed-Skripts jetzt mit perl:

```
perl -p -e 's!^.*/!!' \
    < ../sed/varprefixfiles.txt
```

▶▶ Kommando **vorführen**.

Das s///-Kommando und die -e-Option sind **gleich**.

Die -p-Option läßt perl sich **ähnlich zeilenorientiert** verhalten wie sed.

Achtung: Im allgemeinen ist die **Syntax der regulären Ausdrücke** von sed und perl leider **leicht verschieden!**

3.3 Ein einfaches perl-Programm

(Die „Wirbelwind-Tour“ aus „Learning Perl“, S. 17:)

Ein einfaches `perl`-Programm

```
#!/usr/bin/perl
@lines = `perldoc -u -f atan2`;
foreach (@lines) {
  s/\w<([^>]+)>/\U$1/g;
  print;
}
```


erste Zeile mit `#!`

sagt dem **Unix-Kernel**, das folgende Skript mit `perl` auszuführen

Backquotes

führt **externes Kommando** aus (analog zu Shell)

Kommando perldoc ...

zeigt Dokumentation zu `perl` an

▶▶ Kommando von der Shell aus ausführen

`@lines`

Array-Variable, die die Ausgabe aufnimmt

`foreach` **...**

Schleife über alle Zeilen

`s///`**-Kommando**

ähnlich wie bei `sed`

Syntax für reguläre Ausdrücke leicht anders, z.B.

- `+` ohne Backslash
- `$1` statt `\1`

mächtigere Operatoren: z.B. `\U` für „uppercase“

`print`**-Kommando**

(klar)

▶▶ Programm und seine Wirkung **vorführen**.

3.4 Skalare Daten

„Skalar“: Variable enhält **einen einzigen Wert**.

3.4.1 Zahlen, Strings

Es gibt **intern keine Integer-Zahlen**, nur Fließkommazahlen.

Fließkommaliterale:
1.25
255.000
-6.5e24
-12E-24

Man darf aber **auch Integer-Zahlen schreiben**.

Integer-Literale:
0
2001
-40
61298040283768
61_298_040_283_768

Unterstriche für bessere **Lesbarkeit**, ansonsten gleicher Wert.

Nicht-Dezimale Literale:
0377
0xff
0b11111111

Oktal – **hexadezimal** – **binär.** Immer **gleich 255** dezimal.

Anmerkung: **binäre** Zahlen sind erst **ab Perl 5.6** verfügbar.

Unterstriche auch in nicht-dezimalen Literalen erlaubt, ab Perl 5.6:

0x1377_0b77

Numerische Operatoren:
+
-
*
/
%
**

Der **Modulo-Operator** % wandelt seine Argumente **erst in Integer-Zahlen** um.
10.5 % 3.2 = 10 % 3 = 1

Vorsicht: Modulo von **negativen Zahlen** ist implementationsabhängig.

: **Exponentiation

Strings:

- **Folgen** von **Zeichen**
- **Länge** nur von Rechner-Speicher begrenzt
- anders als in C **kein** besonderes **Terminationszeichen**
- können ggf. **auch ganze Binärdateien** enthalten

Einfach-Quote String-Literale:

```
'Hugo'
'Zwei
Zeilen'
'Wie geht\'s'
'\'\\'
```

Doppel-Quote String-Literale:

```
"Hugo"
"Hallo Welt\n"
"Er heisst \"Hugo\""
"Hugo\tErna"
```

Backslash-Escapes in Doppel-Quote Strings:

Backslash-Escapes in Doppel-Quote Strings

Konstrukt	Bedeutung
`\n`	Newline
`\r`	Return
`\t`	Tab
`\f`	Formfeed
`\b`	Backspace
`\a`	Bell
`\e`	Escape-Zeichen
`\007`	oktale Darstellung (hier: 007 = Bell)
`\x7f`	hexadezimale Darstellung (hier: 7f = Delete)
`\cC`	Control-Zeichen (hier: Control-C)
`\\`	Backslash
`\"`	Doppel-Quote (Gänsefüßchen)
`\l`	nächster Buchstabe klein geschrieben
`\L`	alle folgenden Buchstaben klein geschrieben, bis `\E`
`\u`	nächster Buchstabe groß geschrieben
`\U`	alle folgenden Buchstaben groß geschrieben, bis `\E`
`\Q`	alle nicht-Wort-Zeichen mit Backslash versehen, bis `\E`
`\E`	Ende von `\L`, `\U` oder `\Q`

Kurs Unix-Tools, WS 01/02, Jan Bredereke 16

Außerdem werden **Variablen durch ihren Wert ersetzt**, ähnlich wie in Shell-Skripten auch.

String-Operatoren:

```
"Hallo" . "Welt"
"Hallo" . ' ' . "Welt"
'Hallo Welt' . "\n"
"Hugo" x 3                 "HugoHugoHugo"
"Hugo" x (4+1)
```

Automatische Konvertierung Zahlen ↔ Strings:

`5 x 4`	`"5" x 4`	`"5555"`
`"12" * "3"`	`12 * 3`	`36`
`"  12Hugo34" * "3"`	`12 * 3`	`36`
`"Hugo" * "3"`	`0 * 3`	`0`
`"Z" . 5 * 7`	`"Z" . 35`	`"Z35"`

Die Konversion erfolgt **automatisch**, **wenn** der **Operator es erfordert**

3.4.2 Einschub: Zugang zur `perl`-Dokumentation

Beispiel: Nachschlagen der **skalaren Literale**

▶▶ `man perl`

Liefert Liste der man-Pages zu `perl`. Wir brauchen `perldata`.

▶▶ `man perldata`

Wir finden bald den Abschnitt „Scalar value constructors“

Dort finden wir die Beschreibung der Zahlen-Literale und einen Teil der Beschreibung der String-Literale.

Außerdem finden wir einen Hinweis auf weitere Backslash-Regeln im Abschnitt „Quote and Quotelike Operators“ in `man perlop`.

▶▶ `man perlop`

Suchen nach „Quote and Quotelike Operators“.

Eine Seite weiter unten kommt die gesuchte Tabelle.

Eingebaute Funktionen finden sich in der `perlfunc` man-Page.
▶▶ `man perlfunc`
Beispiel `atan2`

Schneller geht die Suche hier mit **perldoc**:
▶▶ `perldoc perldoc`

▶▶ `perldoc -f atan2`

Man kann auch in einer FAQ-Liste suchen:

▶▶ `perldoc -q books`

3.4.3 Einschub: Eingebaute Warnungen

Wenn ein **String als Zahl** interpretiert wird, werden **Buchstaben** im String **still ignoriert**. **Gefährlich!**

`perl` kann Warnungen über gefährliche Konstrukte ausgeben:

```
$ perl -w progname

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

Sollte man **immer einschalten!** – **Keine Laufzeitkosten**.

Erklärungen zu den Warnungen in der `perldiag` **man-Page**.

▶▶ `man perldiag`

3.4.4 Skalare Variablen

```
$ergebnis
$var1
$Var1
$x_to_y
```

Kennzeichen: **Dollar**-Zeichen.

Ziffern erlaubt, außer an erster Stelle.

Großschreibung ist signifikant.

Underscores erlaubt.

Beliebige Länge.

Zuweisung:

```
$a = 42;
$b = $a;
```

Das **Dollar**-Zeichen wird **immer geschrieben**, auch auf der linken Seite einer Zuweisung.

Binäre Zuweisungsoperatoren:

```
$a = $a + 5;
$a += 5;
```

Kurzform geht für **fast alle binären** Operatoren.

Auch z.B. für **String-Verkettung:**

```
$str .= " ";
```

3.4.5 Ausgaben mit `print`

```
print "Hallo Welt!\n";
```

Auch Liste von Ausdrücken erlaubt. (Listen werden später erklärt.)

```
print "Hallo ", "Welt!\n";
```

3.4.6 Interpolation von Variablen in Doppel-Quote Strings

```
$w = "Welt";
print "Hallo $w!\n";
```

```
print "Hallo ${w}all!\n";
```

3.4.7 Einschub: Erläuterungen zu Warnungen

Man kann die **Erläuterungen** aus der perldiag-man-Page **gleich mit ausgeben** lassen durch:

```
#!/usr/bin/perl -w
use diagnostics;
```

►► `perldoc diagnostics`

Für den Anfang **auf jeden Fall zu empfehlen!** Später immer noch!

Beispiel:

►► `vi diagdemo.pl` :

```
#!/usr/bin/perl
$xyz = 42;
print "\nDer Wert ist '$xy'.\n";
```

►► `./diagdemo.pl`

►► dasselbe mit #/usr/bin/perl -w!

►► dasselbe mit `use diagnostics;` am Anfang

3.4.8 Vergleichsoperatoren

Zahlen	Strings
`==`	`eq`
`!=`	`ne`
`<`	`lt`
`>`	`gt`
`<=`	`le`
`>=`	`ge`

Nicht „`=>`“, das ist ein anderer Operator!

3.4.9 Die `if`-Kontrollstruktur

```
if ($name lt "hugo") {
  print "'$name' kommt vor 'hugo'.\n";
}
```

```
if ($name lt "hugo") {
  print "'$name' kommt vor 'hugo'.\n";
} else {
  print "'$name' kommt nach 'hugo'.\n";
}
```

Die **geschweiften Klammern** sind immer notwendig.

3.4.10 Boolsche Werte

```
$ist_kleiner = $name lt "hugo";
```

Kein separater Datentyp, ist eine Zahl.

false ist:
`0`, `''`, `'0'`, `undef`
Alles andere: true

Negation:
`! $ist_kleiner`

3.4.11 Benutzereingaben

Lesen von der Standard-Eingabe:

▶▶ `view stdin-simple.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$zeile = <STDIN>;
if ($zeile eq "\n") {
  print "Das war nur eine leere Zeile!\n";
} else {
  print "Die Eingabezeile war: $zeile";
}
```

▶▶ `./stdin-simple.pl` mit Text

▶▶ `./stdin-simple.pl` nur mit Return

3.4.12 Der `chomp`-Operator

Zum **Entfernen des Newlines**.

▶▶ `view stdin-chomp.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
chomp($zeile = <STDIN>);
if ($zeile eq "") {
  print "Das war nur eine leere Zeile!\n";
} else {
  print "Die Eingabezeile war: '$zeile' (ohne Newline).\n";
}
```

▶▶ `./stdin-chomp.pl`

Man darf eine **Zuweisung** verwenden, **wo** eine **Variable erwartet** wird.

`chomp` ist eine **Funktion** und gibt die Anzahl der entfernten Zeichen zurück.

Man darf die **Klammern** um die Funktionsargumente **auch weglassen**, wenn es eindeutig ist. **Allgemeine `perl`-Regel**.

3.4.13 Schleifen

Einige von mehreren Schleifenkonstrukten.

▶▶ `view loops-simple.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
$count = 1;
while ($count < 10) {
  print "count ist jetzt $count\n";
  $count += 1;
}
print "-----------------\n";
for ($count = 0; $count < 10; $count++) {
  print "count ist jetzt $count\n";
}
```

▶▶ `./loops-simple.pl`

Mehr Schleifenkonstrukte sind in der `perlsyn`-**man-Page** beschrieben.

3.4.14 Der `undef`-Wert und die `defined`-Funktion

`undef` ist der Wert **nichtinitialisierter Variablen**, wenn nicht mit `-w` verhindert.

Auch viele **Operatoren** geben `undef` zurück, wenn sie keinen Sinn machen.

Beispiel ist das **Lesen** von der Standardeingabe am **Dateiende**:

▶▶ `view stdin-while.pl` :

```
#!/usr/bin/perl -w
use diagnostics;
while (defined($zeile = <STDIN>)) {
  chomp $zeile;
  print "Die Eingabezeile war: '$zeile'.\n";
}
```

▶▶ `./stdin-while.pl`

Die **Funktion** `defined` **prüft** auf `undef`.

Anmerkung: Hier darf `chomp` **nicht direkt auf die Zuweisung angewandt** werden, weil es dann auch auf einen `undef`-Wert angewandt würde.

3.5 Online-Aufgaben

Finden einer Lösung **interaktiv** mit den Teilnehmern, am besten mit **Laptop/Beamer**.

3.5.1 Aufgabe: Interaktive Multiplikation

Schreibe ein `perl`-Programm, das nacheinander nach zwei Zahlen fragt (mit Prompt) und dann deren Produkt ausgibt.

3.6 Hausaufgaben

3.6.1 Aufgabe: Doppelte Zeilen entfernen

Schreibe ein `perl`-Programm, das Zeilen von der Standardeingabe liest und wieder ausdruckt. Ausnahme: Wenn eine Zeile der vorigen gleicht, soll sie nicht gedruckt werden.

Benutze ggf. die **Dokumentation** zu `perl`, um Befehle oder Operatoren nachzuschlagen.

Hinweis: Der `defined`-Operator ist nützlich, um den Sonderfall der ersten Zeile handzuhaben.

Abgabe: Bitte per **Email** an brederek@tzi.de (freiwillig, aber empfehlenswert).

3.7 Listen und Felder

Liste: geordnete Menge von Skalaren.

Array/Feld: Eine **Variable**, die eine Liste enthält.

Die **Typen** der Skalare einer Liste **können verschieden** sein.

Indexmenge beginnt bei **Null**.

Die **Länge** einer Liste ist **dynamisch**. Die Grenze ist der Hauptspeicherplatz.

3.7.1 Zugriff auf Feldelemente

```
$hugo[0] = "Hurtig";
$hugo[1] = 42;
$hugo[2] = 17.5;
```

Der **Namensraum** von Feldern und Skalaren ist **getrennt**.

Elemente **oberhalb des Listenendes** haben einfach den Wert `undef`.

3.7.2 Besondere Feldindizes

Der **letzte benutzte Index** eines Feldes `hugo` ist `$#hugo`.

Eine **Zuweisung** auf `$#hugo` macht das Feld größer oder kleiner.

Negative Indizes zählen vom **Feldende** an, `-1` ist das letzte Element.

3.7.3 Listen-Literale

```
(1, 2, 3)
("karl", 4.5)
()
(1..100, 102, 104)
```

Bereichsoperator „..“.

Die `qw`-**Abkürzung**:

```
("Adam" "Bert" "Clara" "Dieter")
qw/ Adam Bert Clara Dieter /
```

Beliebiger Whitespace als **Trenner** erlaubt, auch Newline.

Auch **andere Begrenzungszeichen** erlaubt:

```
qw/ /
qw! !
qw( )
qw{ }
qw[ ]
qw< >
```

Gut z.B. für Liste von **Dateinamen**.

3.7.4 Listen-Zuweisung

Zuweisung an **mehrere Variablen gleichzeitig**:

```
($vn, $nn, $st) = ("Jan", "Bredereke", "Bremen");
```

Vertauschen von Variableninhalten:

```
($a, $b) = ($b, $a);
```

Die **Liste** wird **vor** der **Zuweisung** aufgebaut.

Was ist, wenn die **Zahl** der Variablen und Werte **nicht gleich** ist?

Weniger Variablen:

```
($vn, $nn) = ("Jan", "Bredereke", "Bremen");
```

Überzähliger Wert wird ignoriert.

Weniger Werte:

```
($vn, $nn, $st) = ("Jan", "Bredereke");
```

Überzählige Variable wird `undef`.

Zuweisung an ganze Liste:

```
@person = qw/ Jan Bredereke Bremen /;
```

Klammeraffe `@` heißt: „Ganze Liste“.

Merkhilfe:

`$`	=	$calar
`@`	=	@rray

Listen verketten:

```
@raum = qw/ MZH 8050 /;
@person2 = (@person, @raum);
```

ergibt

```
@person2 = qw/ Jan Bredereke Bremen MZH 8050 /;
```

Anwendung: Liste als **Stack**.

Wie vermeide ich es, mit **Indizes** zu jonglieren?

```
@array = ();
push(@array, 5);     # @array wird (5)
push @array, 6;     # @array wird (5, 6)
push @array, 7..9;   # @array wird (5..9)
$hugo = pop(@array); # $hugo wird 9, @array wird (5..8)
$karl = pop @array;  # $karl wird 8, @array wird (5..7)
pop @array;          # @array wird (5..6)
```

Die **Klammern** sind **optional,** wenn es eindeutig ist.

`pop` **auf** einem **leeren Array** tut nichts und liefert `undef`.

Zugriff auf das **linke Ende** einer Liste:

```
@person = qw/ Jan Bredereke Bremen /;
$vn = shift(@person); # $vn wird "Jan",
                      # @person wird ("Bredereke" "Bremen")
unshift(@person, $vn); # @person wieder wie vorher
```

Ganz **analog zu** `push` und `pop`.

3.7.5 Felder in Strings interpolieren

```
print "Den Kurs haelt @person.\n";
```

ergibt

```
Den Kurs haelt Jan Bredereke Bremen.
```

Die Listenelemente werden, **durch Blanks getrennt**, eingefügt.

Vorsicht mit Email-Adressen in Strings!

```
print "brederek@tzi.de";
```

Das ist ein Compile-Zeit-Fehler.

Lösungen:

```
print "brederek\@tzi.de";
print 'brederek@tzi.de';
```

3.7.6 Die `foreach`-Anweisung

Schleifen über Listen:

```
@array = (1..100);
foreach $elem @array {
  print "$elem\n";
}
```

Man kann die **Liste** auf diese Weise **auch verändern**:

```
@array = (1..100);
foreach $elem @array {
  $elem++;
}
```

setzt `@array` auf `(2..101)`.

Falls die **Schleifenvariable vorher einen Wert** hatte, wird er hinterher wieder **restauriert.**

3.7.7 Die Default-Variable `$_`

Index-Variable bei `foreach` **weglassen**:

```
foreach (1..100) {
  print "$_\n";
}
```

Auch sehr viele andere Befehle benutzen `$_` als Default:

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

3.7.8 Skalarer und Listen-Kontext

Der `sort`-**Operator** als Beispiel für einen **Operator auf Listen**:

```
@s = sort( qw/ solaris linux windows /);
```

sortiert die **Worte alphabetisch**.

```
5 + ?   # ? muss skalar sein
sort ?  # ? muss Liste sein
```

Ein Ausdruck steht immer entweder in einem skalaren oder einem Listen-Kontext.

Der **Wert des Ausdrucks** hängt davon ab. Beispiel:

```
@a = qw/ b c a /;
@s = sort @a;    # Liste, liefert qw/ a b c /
$n = 5 + @a;     # skalar, liefert 5 + 3 gleich 8
```

Eine **Listenvariable in skalarem Kontext** ergibt die **Anzahl** ihrer **Elemente**.

Noch ein Beispiel: Der `reverse`-Operator:

```
@r = reverse qw/ yabba dabba doo /;  # liefert qw/ doo dabba yabba /
$r = reverse qw/ yabba dabba doo /;  # liefert "oodabbadabbay"
```

Übungsfragen: Welcher Kontext ist es?

```
$hugo = ?;
@karl = ?;
($hugo, $helmut) = ?;
($hugo) = ?;
if( ? ) {...}
$hugo[ ? ] = ?;
push $hugo, ?;
print ?;
```

(Antworten: Skalar, Liste, Liste, Liste, skalar, skalar+skalar, Liste, Liste)

Skalar-liefernde Ausdücke im Listen-Kontext:
Immer Konversion zu **ein-elementiger Liste**:

```
@hugo = 6 * 7;  # liefert (42)
```

Der `sort`-**Operator im skalaren Kontext**:
Liefert `undef`. In diesem Falle keine sinnvolle Rückgabe möglich. Bei Variablennamen aber schon, siehe oben.

3.7.9 `<STDIN>` im Listen-Kontext

```
@zeilen = <STDIN>;
```

Alle **Zeilen** bis zum Dateiende werden **eingelesen**, jede **Zeile** wird **ein Listenelement**.

Abschneiden aller Newlines auf einmal (`chomp` im **Listen-Kontext**):

```
@zeilen = <STDIN>;
chomp(@zeilen);
```

oder **kürzer**:

```
chomp(@zeilen = <STDIN>);
```

3.8 Online-Aufgaben

Finden einer Lösung **interaktiv** mit den Teilnehmern, am besten mit **Laptop/Beamer**.

Die folgenden **Aufgaben** müssen **nicht unbedingt in dieser Reihenfolge** gestellt werden, vielleicht ergeben sich die Varianten ganz natürlich durch eine Diskussion.

Wenn nicht alle Varianten im Kurs bearbeitet werden, können einige **übrige als Hausaufgaben** gestellt werden.

3.8.1 Aufgabe: Liste invertieren

Schreibe ein Programm, das eine **Liste von Strings** einliest, einen pro Zeile, bis zum Dateiende, und die Liste **in umgekehrter Reihenfolge wieder ausgibt**.

(Evtl. Hinweis: Der `reverse`-Operator ist hier nützlich.)

3.8.2 Aufgabe: Liste invertieren mit `push` und `pop`

Schreibe ein Programm, das dasselbe tut, aber den `reverse`-Operator nicht benutzt, sondern die `push`- und `pop`-Operatoren.

3.8.3 Aufgabe: Liste invertieren nur mit `pop`

Variiere die Lösung, indem Du den `push`-Operator nicht benutzt, sondern die Liste in einem Stück einliest.

3.8.4 Aufgabe: Liste invertieren mit `unshift`

Schreibe ein Programm, das dasselbe tut, aber die `push`- und `pop`-Operatoren nicht benutzt, sondern den `unshift`-Operator.

3.8.5 Aufgabe: Liste invertieren mit Indizes

Schreibe ein Programm, das dasselbe tut, aber die obigen Operatoren gar nicht benutzt, sondern Feldindizes.

3.8.6 Aufgabe: Liste invertieren mit Zuweisungen

Schreibe ein Programm, das dasselbe tut, aber Zuweisungen zum Konstruieren einer invertierten Liste benutzt, d.h. Zuweisungen mit einer Liste auf der rechten Seite.

3.8.7 Aufgabe: Liste invertieren mit Zerlegung durch Listenzuweisung

Variiere die vorige Lösung, indem Du die invertierte Liste nicht auf einmal ausdruckst, sondern Zeile für Zeile, und dabei jeweils die erste Zeile mit einer Listenzuweisung extrahierst, d.h. mit einer *Liste* von Variablen auf der linken Seite der Zuweisung.

3.8.8 Aufgabe: Liste invertieren mit negativen Indizes

Variiere die Lösung mit Indizes, indem Du die Liste in einem Stück einliest und dann mit Hilfe eines Schleifenindexes und von negativen Indizes von hinten nach vorn die Liste durchgehst und Zeile für Zeile ausgibst.

3.9 Hausaufgaben

Eine oder mehrere der **obigen Aufgaben**, die im Kurs nicht mehr behandelt wurden.