

Notizen zum Kurs Unix-Tools

WS 01/02
Universität Bremen

© 2001, 2002 Jan Brederke

Stand: 7. Februar 2002

(Zur Notation: Text in **schattierten Kästen** wird an die Tafel geschrieben, und Text in **ovalen Kästen** ist ein Dateiinhalt, der am Rechner vorgeführt wird.)

1 Überblick über die Veranstaltung **Unix-Tools**

1.1 Vorstellung des Veranstalters

- **Jan Brederke**
- Assistent in **AG Betriebssysteme/Verteilte Systeme**

1.2 Ziele

Unter Unix stehen **leistungsfähige Tools** zur Verfügung,

- um Daten zu **analysieren** und zu **manipulieren**,
- insbesondere im **Zusammenspiel** mit weiteren Programmen.

Ergänzung zur Vorlesung Betriebssysteme von Jan Peleska.

Ziel dieses Kurses ist es,

- sich mit einigen oft gebrauchten Werkzeugen **vertraut zu machen** und
- zu wissen, wie und **wann man sie nutzbringend anwenden** kann.

1.3 Geplante **Inhalte**

Auswahl von Werkzeugen:

- sed
- perl
- make
- lex
- yacc

sed	<ul style="list-style-type: none">• nützlich insbesondere in Shell-Skripten• selektieren und manipulieren von Informationen aus Dateien oder der Standardausgabe von Kommandos
perl	<ul style="list-style-type: none">• komplexere solche Aufgaben• sowohl für kurze Skripte gut geeignet als auch für große, ausgewachsene Programme wie etwa aktive Webseiten mit Datenbankbindung• im wesentlichen auf die Grundlagen von perl beschränken, die für Alltagsaufgaben ausreichen
make	<ul style="list-style-type: none">• erlaubt die Automatisierung der vielen Schritte, die zur Generierung eines Programms aus vielen Quelldateien notwendig sind• Verwaltung der Abhängigkeiten dieser Schritte, was bei Programmänderungen wichtig wird
lex & yacc	<ul style="list-style-type: none">• generieren Frontends für selbstgeschriebene, spezialisierte Analyseprogramme• zum Beispiel für Compiler, Interpreter und interaktive Benutzerschnittstellen

1.4 Verfügbarkeit der Werkzeuge

Sind **auf allen Unix-Plattformen** verfügbar. Sollten an der **Uni überall** installiert sein. Statt lex und yacc werden wir im wesentlichen die **Gnu-Versionen flex und bison** nehmen.

Die Gnu-Versionen aller Tools sollten **auch auf Microsoft-Plattformen** verfügbar sein. Auf Microsoft-Plattformen wird allerdings **oft nmake** statt make verwendet.

1.5 Vorgehensweise

Vortrag und **Online-Übungen** im Kurs.

Kleine Hausaufgaben mit wenig Zeitaufwand. **Freiwillig**, aber **empfehlenswert** zum richtigen Verstehen.

Geschwindigkeit hängt stark von den **Vorkenntnissen** ab. Bitte **Rückmeldungen**!

Gehe gerne auf **Wünsche** zum Inhalt ein.

Bücher zum Kurs stehen auf der **Web-Seite**. Dort auch einige **Manuals** zum **Runterladen**.

Literatur
<ul style="list-style-type: none">• N. N.: sed(1) Manual-Seite in: LunetIX Linuxhandbuch (Juli 1993).• Mike Haertel, James A. Woods und David Olson: grep(1) Manual-Seite in: LunetIX Linuxhandbuch (Juli 1993).• Randal L. Schwartz and Tom Phoenix: Learning Perl O'Reilly, 3. Auflage (Juli 2001). ISBN 0-596-00132-0.• Larry Wall, Tom Christiansen und Jon Orwant: Programming Perl O'Reilly, 3. Auflage (Juli 2000). ISBN 0-596-00027-8.• Richard M. Stallman und Roland McGrath: GNU Make - A Program for Directing Recompilation Version 3.79. Free Software Foundation (April 2000). ISBN-1-882114-80-9. GNU General Public License.• John R. Levine, Tony Mason und Doug Brown: lex & yacc O'Reilly, zweite korrigierte Auflage (1995). ISBN 1-56592-000-7.• Vern Paxson: Flex, version 2.5 University of California (1990).• Charles Donnelly and Richard Stallman: Bison Free Software Foundation (1995). ISBN-1-882114-45-0. GNU General Public License.
Web-Seite des Kurses: http://www.tzi.de/~agbs/lehre/ws0102/unix-tools/unix-tools.html
Kurs Unix-Tools, WS 01/02, Jan Brederke
1

Die **Perl**-Bücher gibt es auch in **deutscher Ausgabe**, allerdings das erste noch **nicht** in der **dritten Auflage**.

U.a. im **Addison-Wesley-Verlag** gibt es **ähnliche** Bücher, insbesondere vom Autoren **Helmut Herold**.

Prüfungen: Da der Kurs nur 2 SWS hat, kann man sich **nicht darüber prüfen** lassen. Er ist als freiwillige **Ergänzung zur Betriebssysteme-Vorlesung** gedacht.

2 Der batchorientierte Zeilen-Editor **sed**

2.1 **Einfache Textersetzungen**

Einige **Beispiele** für **typische Problemstellungen** aus dem Unix-Alltag.

- | |
|---|
| <ul style="list-style-type: none">• kleine Textmodifikation• Formatierung einer Liste etwas ändern |
|---|

2.1.1 Problem: Betreff in Vacation-Text einsetzen

Teilaufgabe in einem procmail-Skript. (procmail wird in dieser Vorlesung nicht behandelt.)

```
Liebe Kollegen, liebe Freunde,

mein Email-Programm hat die Nachricht betreffs
"<SUBJECT>"
empfangen. Ich werde sie lesen, sobald ich zurueck im Buero bin,
d.h. am Dienstag, den 4. September 2001, und werde sobald wie
moeglich reagieren.

Mit freundlichen Gruessen,
Jan Brederke

.....

Liebe Kollegen, liebe Freunde,

mein Email-Programm hat die Nachricht betreffs
"Re: Forschungsantrag"
empfangen. Ich werde sie lesen, sobald ich zurueck im Buero bin,
d.h. am Dienstag, den 4. September 2001, und werde sobald wie
moeglich reagieren.

Mit freundlichen Gruessen,
Jan Brederke

Kurs Unix-Tools, WS 01/02, Jan Brederke
```

2

2.1.2 Problem: Festen Präfix in einer Dateiliste entfernen

```
sed/passwd
sed/passwd_names.sh
sed/passwd_names.txt
sed/vacation-msg-filled.txt
sed/vacation-msg-tmpl.txt
sed/vacation-msg.sh

.....

passwd
passwd_names.sh
passwd_names.txt
vacation-msg-filled.txt
vacation-msg-tmpl.txt
vacation-msg.sh

Kurs Unix-Tools, WS 01/02, Jan Brederke
```

3

2.1.3 Lösung: Betreff in Vacation-Text einsetzen

```
subject='Re: Forschungsantrag'
sed "s/<SUBJECT>/${subject}/g" \
    < vacation-msg-tmpl.txt > vacation-msg-filled.txt
```

Die Variable „subject“ wird in Wirklichkeit von procmail bestimmt.

2.1.4 Lösung: Festen Präfix in einer Dateiliste entfernen

```
sed -e 's!sed/!!' \  
< prefixfiles.txt > noprefixfiles.txt
```

Anmerkung: Wir mußten „!“ statt „/“ nehmen, weil „/“ schon im Suchmuster vorkommt.

2.2 Die Grundkommandos von sed

Die Grundkommandos von sed

Syntax
`sed [-e Editorkommando] [-f Scriptdatei] [-n] [Datei...]`

Editorkommando
Adresse Funktion

Funktion
`s/Ausdruck/Ersetzung/[g][p]` (substitute) ersetzt auf *Ausdruck* passenden Text
`d` (delete) löscht die aktuelle Zeile, nichts wird gedruckt
`!Funktion` führt *Funktion* aus, falls die *Adresse* nicht paßt

Adresse
(*leer*)
`/ Ausdruck /`

Ausdruck
ein regulärer Ausdruck wie bei `grep`

Modus
`g`: ersetze alle passenden Texte, nicht nur den ersten
`p`: falls Ersetzung stattgefunden hat, drucke Zeile (trotz Option `-n`)

Kurs Unix-Tools, WS 01/02, Jan Brederke

4

Wenn die **s-Funktion** Slashes in **Ausdruck** oder **Ersetzung** enthält, kann man statt der drei Slashes drei beliebige andere Zeichen verwenden: `s!foo!bar!`

Wenn der **Ausdruck** der **Adresse** Slashes enthält, kann man stattdessen folgendes verwenden: `\c Ausdruck c`

Weitere Funktionen, Adressen und Optionen finden sich in der Manualseite von sed. Man braucht sie fast nie, weil sich die gleichen Ergebnisse leichter mit perl erreichen lassen.

2.3 Warum Reguläre Ausdrücke

- variable Textanteile erkennen
- interessante Textanteile extrahieren

2.3.1 Problem: Einen variablen Präfix in einer Dateiliste entfernen

```
./article/implicit-concepts.tex
./article/ft-example-sli.tex
./article/circumvent.tex
./article/blocked.tex
./article/maint-tel-req.tex
./article/tina-concepts.tex
./dagstuhl-talk.tex
./dagstuhl.tex
```

```
.....

implicit-concepts.tex
ft-example-sli.tex
circumvent.tex
blocked.tex
maint-tel-req.tex
tina-concepts.tex
dagstuhl-talk.tex
dagstuhl.tex
```

Kurs Unix-Tools, WS 01/02, Jan Brederke

5

2.3.2 Lösung: Einen variablen Präfix in einer Dateiliste entfernen

```
sed -e 's!^.*!/!!' \  
    < varprefixfiles.txt > novarprefixfiles.txt
```

2.3.3 Problem: Liste aller Benutzer aus /etc/passwd extrahieren

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:daemon:/sbin:/bin/bash
wwwrun:x:30:65534:Daemon user for apache:/var/lib/wwwrun:/bin/bash
named:x:44:44:Nameserver Daemon:/var/named:/bin/bash
nobody:x:65534:65534:nobody:/var/lib/nobody:/bin/bash
brederek:x:500:100:Jan Brederke:/home/brederek:/bin/bash
mueller:x:501:100:B. Mueller:/home/mueller:/bin/bash
meyer:x:502:100:L. Meyer:/home/meyer:/bin/bash
schulze:x:503:100:G. Schulze:/home/schulze:/bin/bash
```

```
.....

root
bin
daemon
Daemon user for apache
Nameserver Daemon
nobody
Jan Brederke
B. Mueller
L. Meyer
G. Schulze
```

Kurs Unix-Tools, WS 01/02, Jan Brederke

6

2.3.4 Lösung: Liste aller Benutzer aus /etc/passwd extrahieren

Schritt für Schritt an der Tafel entwickeln:

```
sed \
  -e 's/^\[^\:]*\)\{4\}\([^\:]*\):.*\/2/' \
  < passwd > passwd_names.txt
```

2.3.5 Problem: Liste aller „echten“ Benutzer aus /etc/passwd extrahieren

Erweiterung des vorigen Problems.

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
daemon:x:2:2:daemon:/sbin:/bin/bash
wwwrun:x:30:65534:Daemon user for apache:/var/lib/wwwrun:/bin/bash
named:x:44:44:Nameserver Daemon:/var/named:/bin/bash
nobody:x:65534:65534:nobody:/var/lib/nobody:/bin/bash
brederek:x:500:100:Jan Bredereke:/home/brederek:/bin/bash
mueller:x:501:100:B. Mueller:/home/mueller:/bin/bash
meyer:x:502:100:L. Meyer:/home/meyer:/bin/bash
schulze:x:503:100:G. Schulze:/home/schulze:/bin/bash
```

```
.....

Jan Bredereke
B. Mueller
L. Meyer
G. Schulze
```

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

7

2.3.6 Lösung: Liste aller „echten“ Benutzer aus /etc/passwd extrahieren

Ein weiteres sed-Kommando vorweg, der Rest bleibt gleich:

```
sed -e '/^\[^\:]*\)\{2\}5[0-9][0-9]:.*\/!d' \
  -e 's/^\[^\:]*\)\{4\}\([^\:]*\):.*\/2/' \
  < passwd > passwd_realnames.txt
```

2.4 Aufbau der Reguläre Ausdrücke von sed (und grep)

Reguläre Ausdrücke von sed (und grep)

Ausdruck
eine Folge von *Mustern*, optional mit *Zwischenraummarkierungen* dazwischen

Muster
ein Muster ist ein einzelnes *Zeichen* oder eine *Gruppierung*

Zeichen

<i>c</i>	einzelner Buchstabe: paßt auf sich selbst
<i>.</i>	Punkt: paßt auf jedes Zeichen außer dem Zeilenende
<i>\c</i>	Backslash: macht aus Sonderzeichen normale Zeichen
<i>[Buchstaben]</i>	Menge von Buchstaben: paßt auf eines der Zeichen aus dieser Menge
	<i>Buchstaben</i> können auch Bereiche <i>von-bis</i> sein
	die Buchstaben „ <i>[</i> “, „ <i>]</i> “ und „ <i>-</i> “ müssen ggf. ganz vorne oder hinten stehen
<i>[^Buchstaben]</i>	Komplementmenge von Buchstaben: paßt genau auf keines der Zeichen

Kurs Unix-Tools, WS 01/02, Jan Bredeke 8

Reguläre Ausdrücke (Fortsetzung)

Gruppierung

<i>Muster*</i>	eine Folge von 0 oder mehr Mustern
<i>Muster?</i>	0 oder einmal das Muster
<i>Muster\+</i>	eine Folge von 1 oder mehr Mustern
<i>Muster\{n\}</i>	eine Folge von <i>n</i> Mustern (fehlt in man-Pages)
<i>Muster\{n,m\}</i>	eine Folge von <i>n</i> bis <i>m</i> Mustern (fehlt in man-Pages)
<i>\(Ausdruck\)</i>	faßt Ausdruck zusammen; markiert Text für <i>\N</i>
<i>\N</i>	das <i>N</i> -te mit Klammern eingeschlossene Muster
<i>Muster\ Muster</i>	entweder das erste oder das zweite Muster

Zwischenraummarkierung

<i>^</i>	Zeilenanfang
<i>\$</i>	Zeilenende
<i>\<</i>	Wortanfang
<i>\></i>	Wortende

Kurs Unix-Tools, WS 01/02, Jan Bredeke 9

2.5 Online-Aufgaben

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

2.5.1 Aufgabe: Erkennen von Email-Adressen

Es soll ein regulärer Ausdruck geschrieben werden, der möglichst genau auf die folgenden Email-Adressen paßt:


```

brederrek@tzi.de
brederrek@informatik.uni-bremen.de
brederrek@gemini.informatik.uni-bremen.de
brederrek@saturn.informatik.uni-bremen.de
root@saturn.informatik.uni-bremen.de
jan.bredereke@web.de

```

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

10

Es sollen aus einer Datei genau diese Email-Adressen ausgegeben werden.

2.5.2 Aufgabe: Extraktion von include-Dateinamen aus einer LaTeX-Quelle

Eine LaTeX-Quelldatei enthält Zeilen der Art

```
\verbatiminput{sed/passwd_names.sh}
```

Es soll eine Liste der Dateinamen in diesen Kommandos ausgegeben werden.

Mit „%“ auskommentierte LaTeX-Befehle sollen dabei ignoriert werden.

2.5.3 Aufgabe: Extraktion aller derzeit aktiven Benutzer aus „w“-Ausgabe

Ausgabe des w-Kommandos:

```

12:24pm up 56 day(s), 2:42, 10 users, load average: 0.00, 0.02, 0.03
User  tty      login@  idle   JCPU   PCPU   what
alone  smb/0      12:24pm                /usr/local/lib/samba/smbd -D
nobody smb/1      12:24pm                /usr/local/lib/samba/smbd -D
roofer smb/2      12:14pm                /usr/local/lib/samba/smbd -D
root   pts/1      1Aug01 2days 20:31    2 /usr/local/bin/bash -login
roofer smb/4      12:18pm                /usr/local/lib/samba/smbd -D
mawe   pts/2      1Aug01 14    6:06    25 contool
root   pts/3      1Aug01 13    9:29    5 /usr/local/bin/bash -login
cx1    pts/4      Tue 5pm 1:23    35    35 xemacs
cx1    pts/5      Thu 9am 5days 10:22    -bash
brederrek pts/6      11:19am                w

```

Gewünscht:

- Liste von Benutzernamen
- keine Benutzer mit „idle“-Zeit
- keine Samba-Daemons (d.h. nur Benutzer an Pseudo-TTys)
- keine Kopfzeile des w-Kommandos

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

11

2.6 Hausaufgaben

2.6.1 Schützen von Sonderzeichen in der Vacation-Text-Ersetzung

Problem: Wenn der Betreff selbst Schrägstriche enthält, dann wird das `sed`-Kommando falsch interpretiert. Lösungsidee: Füge vor alle Schrägstriche (und vor alle Backslashes) im Betreff Backslashes ein.

Aufgabe: Schreibe ein `sed`-Kommando, das seine Standard-Eingabe entsprechend transformiert. Nimm ggf. die man-Pages zu `sed` zu Hilfe.

2.7 Kombinieren von Tools: Bearbeiten von ganzen Dateibäumen mit `find`

Die Grundkommandos von `find`

Syntax
`find Verzeichnis,... [-Option...] [-Test...] [-Aktion...]`

Test
`-name Muster` Dateiname paßt zu Muster (mit „*“, „?“, „[...]“ wie bei Shell)
(viele Tests zu Dateialter, Dateiattributen, Dateityp, ...)

Aktion
`-print` druck Dateinamen mit Pfad
`-exec Kommando ;` führt Kommando aus
(etliche Varianten dieser Kommandos)

Option
(diverse zur Art der Baumsuche, ...)

Anmerkungen

- Im *Kommando* von `-exec` wird der Ausdruck „{“ durch den Dateinamen mit Pfad ersetzt.
- Bei Aufruf von der Shell müssen „{“ und „;“ geeignet in Anführungszeichen gesetzt werden.
- Ist keine Aktion angegeben, ist `-print` Default.
- Tests und Aktionen können mit Klammern und boolschen Operatoren zu komplexen Suchausdrücken kombiniert werden.

Kurs Unix-Tools, WS 01/02, Jan Bredeke

12

2.7.1 Aufgabe: Liste aller Dateien unter CVS

Man kann Dateien mit dem **Versionsverwaltungssystem CVS** verwalten, sogar **ganze Bäume** von Dateien. Aber **nicht alle Dateien** werden dabei unter CVS **verwaltet**. Zum Beispiel Objektdaten werden ausgelassen.

CVS legt in jedem verwalteten Verzeichnis ein **Unterverzeichnis CVS** an, und darin u.a. eine **Datei CVS/Entries**, in der alle verwalteten Dateien zusammen mit Zusatzinformationen aufgeführt sind.

Beispiel einer Datei CVS/Entries

```
D/WWW////  
/kursaufbau.txt/1.5/Mon Sep 17 16:26:56 2001//  
D/sed////  
/.cvsignore/1.7/Mon Sep 24 10:26:33 2001//  
D/perl////  
/skriptnotizen.mk/1.7/Thu Sep 27 08:02:23 2001/-kb/  
/skriptnotizen.sty/1.7/Tue Oct 2 12:52:00 2001//  
/folien.tex/1.15/Thu Oct 4 19:24:36 2001//  
/skriptnotizen.tex/1.22/Thu Oct 4 19:24:40 2001//  
/vorstellung-folien.dvi/1.1/Tue Oct 9 15:03:09 2001/-kb/  
/vorstellung-folien.tex/1.1/Tue Oct 9 15:03:07 2001/-kb/
```

Es soll eine **Liste aller Dateinamen** ausgegeben werden, die mit CVS verwaltet werden. Dies soll **für alle Unterverzeichnisse** eines Verzeichnisses geschehen. Der Einfachheit halber soll nur der Dateiname **ohne den ganzen Pfad** ausgegeben werden.

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 Brederke

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;
}
```

Kurs Unix-Tools, WS 01/02, Jan Bredeke

15

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

Kurs Unix-Tools, WS 01/02, Jan Brederke 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 Benutzer**eingaben**

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 brederrek@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 Brederke Bremen /;
```

Klammeraffe @ heißt: „Ganze Liste“.

Merkhilfe:

```
$ = $scalar  
@ = @array
```

Listen verketteten:

```
@raum = qw/ MZH 8050 /;  
@person2 = (@person, @raum);  
  
ergibt  
  
@person2 = qw/ Jan Brederke 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 Brederke Bremen /;  
$vn = shift(@person); # $vn wird "Jan",  
                        # @person wird ("Brederke" "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 Brederke 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 Ausdrücke 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 ausdrückst, 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.

3.10 Reguläre Ausdrücke von Perl

3.10.1 Ersetzungen mit `s///`

Der **Ersetzungsoperator** `s///` arbeitet **ganz ähnlich wie** sein Gegenstück **bei sed**.

Der Ersetzungsoperator ist eine **normale Anweisung**:

►► view subst-statement.pl :

```
#!/usr/bin/perl -w
use diagnostics;
while(<STDIN>) {
    s/x/u/;
    print;
}
```

►► ./subst-statement.pl

(Eingabezeilen u.a. mit „Haxe“, „Taxe“.)

Der Ersetzungsoperator **arbeitet** per Default **auf der Default-Variablen** `$_`, genau wie `while(<STDIN>)` und `print`.

Für die Wahl anderer **Begrenzungszeichen** bietet **perl** mehr Möglichkeiten als **sed**:

```
s#^https://#http://#;

s{hugo}{karl};
s[hugo](karl);
```

Paarweise vorkommende Begrenzungszeichen werden anders behandelt.

3.10.2 Einschub: Kurzform für die `while-print-Schleife`

Für Programme wie das obige, die im wesentlichen aus einer **while-print-Schleife** bestehen, gibt es eine **Kurzform**:

►► view subst-statement2.pl :

```
perl -p -e 's/x/u/'
```

ist äquivalent zu

►► view subst-statement3.pl :

```
#!/usr/bin/perl
while(<>) {
    s/x/u/;
    print;
}
```

Diesen Trick hatten wir am Anfang des **perl**-Kapitels bereits verwendet, um **sed-artige**

Aufrufe von perl zu machen.

Anmerkung: Der **Diamant-Operator** <> arbeitet hier ähnlich wie der Operator <STDIN>. Näheres dazu kommt **später**.

Es gibt ebenfalls die „**-n**“-Option von sed. Sie ergibt die gleiche Schleife, aber ohne das implizite print:

▶▶ view subst-statement4.pl :

```
perl -n -e 'if(s/x/u/){print "ergibt: $-";}'
```

ist äquivalent zu

▶▶ view subst-statement5.pl :

```
#!/usr/bin/perl
while(<>) {
    if(s/x/u/) {
        print "ergibt: $-";
    }
}
```

▶▶ ./subst-statement4.pl

Anmerkung: der Ersetzungsoperator s/// gibt die Anzahl der Ersetzungen zurück (und den leeren String sonst).

3.10.3 Unterschiede zu sed und grep

Reguläre Ausdrücke von perl: etwas andere Syntax; mehr Möglichkeiten.

Die alten Folien zu sed erweitert:

Reguläre Ausdrücke von sed vs. perl	
Gemeinsamkeiten:	
Ausdruck	eine Folge von <i>Mustern</i> , optional mit <i>Zwischenraummarkierungen</i> dazwischen
Muster	ein Muster ist ein einzelnes <i>Zeichen</i> oder eine <i>Gruppierung</i>
Zeichen	
<i>c</i>	einzelner Buchstabe: paßt auf sich selbst
<i>.</i>	Punkt: paßt auf jedes Zeichen außer dem Zeilenende
<i>\c</i>	Backslash: macht aus Sonderzeichen normale Zeichen
<i>[Buchstaben]</i>	Menge von Buchstaben: paßt auf eines der Zeichen aus dieser Menge <i>Buchstaben</i> können auch Bereiche <i>von-bis</i> sein die Buchstaben „ <i>[</i> “, „ <i>]</i> “ und „ <i>-</i> “ müssen ggf. ganz vorne oder hinten stehen
<i>[^Buchstaben]</i>	Komplementmenge von Buchstaben: paßt genau auf keines der Zeichen
.....	
Unterschiede bei perl:	
Zeichen	
<i>\c</i>	Backslash: macht aus <i>allen</i> Sonderzeichen normale Zeichen
Kurs Unix-Tools, WS 01/02, Jan Brederke	
17	

Reguläre Ausdrücke von sed vs. perl (Fortsetzung)		
Gruppierung		
sed	perl	
	<i>Muster*</i>	eine Folge von 0 oder mehr Mustern
	<i>Muster?</i>	0 oder einmal das Muster
<i>Muster</i> +	<i>Muster</i> +	eine Folge von 1 oder mehr Mustern
<i>Muster</i> { <i>n</i> }	<i>Muster</i> { <i>n</i> }	eine Folge von <i>n</i> Mustern
<i>Muster</i> { <i>n,m</i> }	<i>Muster</i> { <i>n,m</i> }	eine Folge von <i>n</i> bis <i>m</i> Mustern
<i>(Ausdruck)</i>	<i>(Ausdruck)</i>	faßt Ausdruck zusammen; markiert Text für <i>\N</i>
<i>\N</i>	<i>\N</i> oder <i>\$N</i>	das <i>N</i> -te mit Klammern eingeschlossene Muster
<i>Muster</i> <i>Muster</i>	<i>Muster</i> <i>Muster</i>	entweder das erste oder das zweite Muster
Zwischenraummarkierung		
sed	perl	
	<i>^</i>	Zeilenanfang
	<i>\$</i>	Zeilenende
<i>\<</i>	<i>\b</i>	Wortanfang
<i>\></i>	<i>\b</i>	Wortende

Kurs Unix-Tools, WS 01/02, Jan Bredeke

18

Die **Backslash-Regeln** müssen bei perl anders sein, damit die **Regel** eingehalten werden kann, daß ein **Backslash** *jedes Zeichen* zu einem **normalen** Zeichen macht.

Man kann **sowohl \1 als auch \$1** benutzen, aber letzteres paßt besser zur sonstigen Variablensyntax. Man **darf** diese **Referenzen auch außerhalb** der regulären Ausdrücke als Variablen benutzen, aber dabei ist nur die zweite Form erlaubt.

Mächtigere reguläre Ausdrücke von perl		
Zeichen		
<i>\w</i>	Abkürzung für <i>[a-zA-Z0-9_]</i>	(alphanum. Zeichen)
<i>\W</i>	Abkürzung für <i>[^\w]</i>	(kein alphanum. Zeichen)
<i>\s</i>	Abkürzung für <i>[\t\n\r]</i>	(White-Space)
<i>\S</i>	Abkürzung für <i>[^\s]</i>	(kein White-Space)
<i>\d</i>	Abkürzung für <i>[0-9]</i>	(Ziffer)
<i>\D</i>	Abkürzung für <i>[^\d]</i>	(keine Ziffer)

Außerdem gibt es eine Reihe von fortgeschrittenen Erweiterungen mit *(?...)*
(z.B. Kommentar, Lookahead-Pattern, Bedingung, ...)

Kurs Unix-Tools, WS 01/02, Jan Bredeke

19

Dokumentation dazu: auf der **perlre-Manpage**.

Kommentare in regulären Ausdrücken von perl:

perl/latex-include1.pl:

Kommentare in regulären Ausdrücken von perl

sed-Stil:

```
perl -n -e 'if(s/^[~%]*\\verbatiminput{([~}]+)}.*/$1/){print;}'
```

Wenn der x-Modifikator angegeben ist, dann

- wird White-Space in regulären Ausdrücken ignoriert,
- ist **#** das normale Kommentarzeichen.

perl/latex-include2.pl:

Kommentare in regulären Ausdrücken von perl (Forts.)

Dasselbe mit perl-Kommentaren dokumentiert:

```
#!/usr/bin/perl -w
use diagnostics;
while(<STDIN>) {
    if( s/^[~%]*
        [~}%]*
        \\verbatiminput{
        (
        [~}%]+
        )
        }
        .*
        /$1/x
    ) {
        print;    # Drucke nur, wenn eine Ersetzung stattgefunden hat.
    }
}
```

Die eigentlichen **Operatoren** bleiben zwar kryptisch, aber man kann sie erheblich besser dokumentieren und daher auch **später noch lesen**.

Anmerkung: Wir haben außerdem bessere Warnmeldungen hinzugefügt und den (noch nicht eingeführten) Diamant-Operator durch **<STDIN>** ersetzt.

3.10.4 Der Bindungsoperator **=~**

Die Textersetzung arbeitet **bisher immer auf der Default-Variablen \$_**. Deswegen arbeitete sie **eben mit** dem Lesen von **STDIN** und der Ausgabe mit **print** zusammen.

Der **Bindungsoperator** kann das ändern:

➤➤ view perl/bind-subst.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$zeile = "Fahrt ins Gruene\n";
$zeile =~ s/Gruene/Blaue/;
print $zeile;
```

➤➤ ./bind-subst.pl

3.10.5 Matchen mit `m//`

`sed` konnte dem Kommando ein **Suchmuster** voranstellen, so daß das Kommando **nur auf** den **passenden Zeilen** ausgeführt wurde. Das **Muster** stand **zwischen zwei Slashes**.

Das kann `perl` auch. **Hier** ist es ein **boolscher Ausdruck**.

➤➤ view perl/match-cmd.pl :

```
#!/usr/bin/perl -w
use diagnostics;
print "Magst Du Perl? ";
$zeile = <STDIN>;
if ($zeile =~ /ja/) {
    print "Fein!\n";
} else {
    print "Schade.\n";
}
```

➤➤ ./match-cmd.pl

Anmerkung: Man kann die Variable `$zeile` auch eliminieren.

Die Form mit den **zwei Slashes** ist **nur** die **Kurzform** des Match-Operators. Langform:

➤➤ view perl/match-cmd2.pl :

```
#!/usr/bin/perl -w
use diagnostics;
print "Magst Du Perl? ";
while (<STDIN>)
{
    if ( m{ja/nein} ) {
        print "Dann entscheide Dich!\nMagst Du perl jetzt? ";
    } elsif ( m/ja/ ) {
        print "Fein!\n";
        exit;
    } else {
        print "Schade.\n";
        exit;
    }
}
```

►► ./match-cmd2.pl

Der **zweite reguläre Ausdruck** zeigt die **normale Langform**. Der **erste** zeigt, daß man bei der Langform **auch andere Begrenzungszeichen** nehmen darf, wenn Slashes ungünstig sind.

3.10.6 **Modifikatoren**

Modifikatoren für das Matchen und für das **Ersetzen**:

```
i    Case-insensitiv
s    . umfaßt auch Newline („single line“)
m    ^ und $ matchen auch an Newline („multiple lines“)
x    Kommentare erlaubt
g    (nur s///) ersetze global alle Treffer
```

►► view perl/match-mod.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = 'Erste Zeile
zweite zeile
3. Zeile
-----
';
print $text;
$text =~ s/zeil.*/Reihe/ ; # probiere Modifikatoren: i, ig, g
# $text =~ s/^Erste.*Zeile$/Erste Reihe/ ; # probiere Modifik.: m, ms, s
print $text;
```

▶▶ ./match-mod.pl
Varianten durchspielen.

3.10.7 Interpolieren in Mustern

Muster werden **wie Doppelquote-Strings interpoliert**. Man kann also **Variablen** und **Escape-Zeichen** verwenden.

▶▶ view perl/match-mod2.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = 'Erste Zeile
zweite zeile
3. Zeile
-----
';
$von = 'zeil.*';
$nach = 'Reihe';
print $text;
$text =~ s/$von/$nach/ig ;
print $text;
$text =~ s/(Erste )(.*)/$1\U$2/s ;
print $text;
```

▶▶ ./match-mod2.pl

3.10.8 Die Match-Variablen

Die Variablen \$1, \$2, \$3, ... enthalten die **mit runden Klammern markierten Teile** des **letzten erfolgreichen Matches**. Dies gilt **sowohl** für den **Match-Operator** als auch für den **Ersetzungsoperator**.

Man darf sie auch außerhalb des Operators verwenden:

▶▶ view perl/match-var.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = 'Ein Wort ist ein WORT ist ein Wort';
if( $text =~ /\b([A-Z]+\b)/ ) {
    print "Das grosse Wort ist '$1'.\n";
} else {
    print "Kein grosses Wort zu sehen.\n";
}
```

➤➤ ./match-var.pl

Sie sind **bis** zum nächsten **erfolgreichen Match gültig**.

sed kannte nur \1, perl kennt sowohl \1 als auch \$1. \1 bezieht sich auf den **gegenwärtig matchenden** Text, während \$1 sich auf den **letzten insgesamt erfolgreich matchenden** Text bezieht.

Beispiel:

➤➤ view perl/match-var2.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "Affe Affe Affe Baer Chamaeleon\n";
print $text;
$text =~ s/(\w+) \1/Zikade Zikade/; # OK
print $text;
$text =~ s/(\w+) $1/Quagga Quagga/; # Wollen wir das wirklich???
print $text;
```

➤➤ ./match-var2.pl

Bei der **zweiten Ersetzung** **matcht** das \$1 auf das **gemerkte Muster** aus der *vorigen* Ersetzung!

Anmerkung: Wenn man \1 auf der **rechten Seite einer Ersetzung** verwendet, darf man das zwar im Prinzip, aber man bekommt eine **Warnung**.

Es gibt noch drei weitere, „**automatische**“ Match-Variablen.

Variable	Bedeutung
\$&	der gesamte matchende Text
\$‘	der Text vor dem Match (Dollar-Backtick)
\$’	der Text nach dem Match (Dollar-Apostroph)

➤➤ view perl/match-var3.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "Diese Zeile ist langweilig, oder doch nicht?\n";
print $text;
if ( $text =~ /\w+,/ ) {
    print "$‘\U$&\E$’";
} else {
    print "Kein Wort mit Komma danach gefunden.\n";
}
```

➤➤ ./match-var3.pl

Anmerkung: **Merkhilfe** für \$‘ und \$’: Backtick und Apostroph sind die **öffnenden und schließenden Anführungszeichen** in LaTeX, die folglich ein Wort einrahmen.

3.10.9 Einschub: **Langformen** für kryptische Variablennamen

Langformen für kryptische Variablennamen		
Kurzform	Langform	Bedeutung
<code>\$_</code>	<code>\$ARG</code>	die Default-Variablen
<code>\$_</code>	<code>\$MATCH</code>	der gesamte matchende Text
<code>\$'</code>	<code>\$PREMATCH</code>	der Text vor dem Match (Dollar-Backtick)
<code>\$'</code>	<code>\$POSTMATCH</code>	der Text nach dem Match (Dollar-Apostroph)
<code>\$\$</code>	<code>\$PID</code> oder <code>\$PROCESS_ID</code>	Die Prozeßnummer des laufenden Perl-Programms
<code>\$0</code>	<code>\$PROGRAM_NAME</code>	Der Dateiname des Perl-Skripts (Dollar-Null)
<code>\$.</code>	<code>\$NR</code> oder <code>\$INPUT_LINE_NUMBER</code>	Die aktuelle Zeilennummer der zuletzt gelesenen Eingabedatei
...		
Benutzung möglich nach:		
<code>use English;</code>		
Dokumentation:		
<code>man perlvar</code>		
Kurs Unix-Tools, WS 01/02, Jan Brederke		
22		

Das **letzte Beispiel nochmal**, aber mit **Langformen** (und mit kommentiertem regulärem Ausdruck).

►► `view perl/match-var4.pl :`

```
#!/usr/bin/perl -w
use diagnostics;
use English;
$text = "Diese Zeile ist langweilig, oder doch nicht?\n";
print $text;
if ( $text =~ /\w+ # Suche ein Wort,
      ,      # das von einem Komma gefolgt wird.
      /x ) {
    print "$PREMATCH\U$MATCH\E$POSTMATCH";
} else {
    print "Kein Wort mit Komma danach gefunden.\n";
}
```

Ein Nachteil: *Alle* Regulären Ausdrücke werden **langsamer**, wenn irgendwo `use English;` verwendet wird.

3.10.10 Der **split**-Operator

split: Teilt einen **String** in eine Liste von Teilstücken **auf**, an den Stellen, wo ein bestimmter Separator im String steht.

Beispiel: Die bekannte **sed**-Aufgabe, die **Liste der „echten“ Benutzer** aus `/etc/passwd` zu extrahieren:

►► view perl/passwd_realnames.pl :

```
#!/usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
# In order to avoid warnings about unused variables, we declare them
# explicitly. (The latter will be explained later.)
my ($login, $passwd, $uid, $gid, $realname, $rest);
($login, $passwd, $uid, $gid, $realname, $rest) = split /:/ ;
if($uid >= 500 and $uid <= 599) {
    print "$realname\n";
}
```

►► ./passwd_realnames.pl < ../sed/passwd

Hier wird **split** implizit auf die Variable **\$_** angewandt. Man kann **split** auch auf andere Variablen anwenden, und man kann auch andere Muster verwenden. **Alle regulären Ausdrücke** sind erlaubt.

Beispiel:

►► view perl/split-ws.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "Dies      ist ein \t      Test.\n";
@felder = split /\s+/, $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
    print "'$feld' ";
}
print "\n";
```

►► ./split-ws.pl

Was passiert, wenn wir das Muster **\s*** verwenden?

►► view perl/split-ws-ast.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "Dies      ist ein \t      Test.\n";
@felder = split /\s*/, $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
    print "'$feld' ";
}
print "\n";
```

►► ./split-ws-ast.pl

Wenn das **Muster auch den leeren String matcht**, wird nach *jedem* Buchstaben aufgeteilt.

Was passiert, wenn Felder leer sind?

►► view perl/split-leer.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "::hallo::";
@felder = split /:/, $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
    print "'$feld' ";
}
print "\n";
```

►► ./split-leer.pl

Es gibt **vorne entsprechend Leerstrings** als Listenelemente.

Leere Felder hinten werden **nicht in die Liste** aufgenommen. **Vorteil:** Bei einer Zuweisung an eine Liste von skalaren Variablen ergibt das ggf. den Wert undef **für nicht belegbare Variablen**.

Wenn man nicht weniger Felder bekommen **will**, kann man als **dritten Parameter** die **Anzahl** der gewünschten Felder angeben. **Damit** bekommt man **auch nicht mehr** als die gewünschte Anzahl.

Sonderfall: Anstelle eines Musters ein **einzelnes Blank** (*ohne* die Slashes). Fast wie `\s+`, aber führende leere Felder werden unterdrückt.

Beispiel:

►► view perl/split-blank.pl :

```
#!/usr/bin/perl -w
use diagnostics;
$text = "        Dies        ist ein \t        Test.\n";
@felder = split ' ', $text;
print "Die Felder sind: ";
foreach $feld (@felder) {
    print "'$feld' ";
}
print "\n";
```

►► ./split-blank.pl

Dies ist auch das Default als Muster.

3.10.11 Die `join`-Funktion

`join` benutzt **keine regulären Ausdrücke**. Aber sie ist das **Gegenstück** zu `split`. `join` **fügt eine Liste** von Strings **zu einem einzigen String** zusammen und fügt an den **Verbindungsstellen** einen festen String ein.

Beispiel:

▶▶ `view perl/join-date.pl :`

```
#!/usr/bin/perl -w
use diagnostics;
$date = '13/11/2001';
print "The British date is $date\n";
@felder = split /\//, $date;
$datum = join '.', @felder;
print "Das deutsche Datum ist $datum\n";
```

▶▶ `./join-date.pl`

3.11 Hausaufgabe

3.11.1 Aufgabe: Cross-Reference-Programm

Schreibe ein `perl`-Programm, das ein `perl`-Programm einliest und eine **Liste aller skalaren Variablen** (`$xy`) sowie eine **Liste aller Feldvariablen** (`@xy`) ausgibt. Es sollen **nur die alphanumerischen Namen** gesucht werden, nicht die kryptischen Kurzformen. `perl`-Kommentare sollen ignoriert werden, aber die besondere Behandlung von `$` und `@` innerhalb von einfachen Anführungsstrichen vernachlässigen wir.

Hinweise: Verwende die **zusätzlichen Konstrukte in regulären Ausdrücken**, die `perl` für alphanumerische Zeichen bereitstellt. Verwende zum **Zerlegen der Eingabe** die `split`-Operation. Benutze nach Möglichkeit auch den **Bindungsoperator**.

3.12 Slices

Auf Seite 38 hatten wir eine Lösung mit perl vorgestellt, um die **Liste der „echten“ Benutzer** aus `/etc/passwd` zu extrahieren:

►► view perl/passwd_realnames.pl :

```
#!/usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
# In order to avoid warnings about unused variables, we declare them
# explicitly. (The latter will be explained later.)
my ($login, $passwd, $uid, $gid, $realname, $rest);
($login, $passwd, $uid, $gid, $realname, $rest) = split /:/ ;
if($uid >= 500 and $uid <= 599) {
    print "$realname\n";
}
```

Nachteil: Vier **Dummy-Variablen** waren notwendig.

Man kann auch ohne die Dummy-Variablen auskommen. Wenn bei einer **Listen-Zuweisung** `undef` vorkommt, wird der entsprechende **Wert** einfach **ignoriert**:

►► view perl/passwd_realnames2.pl :

```
#!/usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
(undef, undef, $uid, undef, $realname, undef) = split /:/ ;
if($uid >= 500 and $uid <= 599) {
    print "$realname\n";
}
```

►► ./passwd_realnames2.pl < ../sed/passwd

Nachteil: Man muß ggf. **viele undefs abzählen**.

Besser: ein **Listen-Slice** (im **skalaren Kontext**):

►► view perl/passwd_realnames3.pl :

```
#!/usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
$uid = (split /:/)[2];
$realname = (split /:/)[4];
if($uid >= 500 and $uid <= 599) {
    print "$realname\n";
}
```

►► ./passwd_realnames3.pl < ../sed/passwd

Die **runden Klammern** um den Listen-liefernden Ausdruck sind hier **notwendig**.

Die resultierende **Liste** wird beim Indizieren **wie ein Array behandelt**.

Nachteil dieser Lösung: **Zwei Aufrufe** von `split`.

Noch **besser**: ein **Listen-Slice** im **Listen-Kontext**:

▶▶ `view perl/passwd_realnames4.pl :`

```
#!/usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
($uid, $realname) = (split /:/)[2, 4];
if($uid >= 500 and $uid <= 599) {
    print "$realname\n";
}
```

▶▶ `./passwd_realnames4.pl < ../sed/passwd`

Man darf **bei der Indizierung alles verwenden**, was sonst auch erlaubt ist:

▶▶ `view perl/list-slice.pl :`

```
#!/usr/bin/perl -w
use diagnostics;
@namen = qw{ null eins zwei drei vier fuenf sechs sieben acht neun MZH };
@raum = ( @namen )[-1, 5, 3, 0, 0];
print "Der Kurs findet statt in @raum.\n";
```

▶▶ `./list-slice.pl`

Wenn man **Elemente** nicht aus einer allgemeinen Liste, sondern **speziell aus einem Array** holt, gibt es eine **einfachere Notation**, das **Array-Slice**:

▶▶ `view perl/array-slice.pl :`

```
#!/usr/bin/perl -w
use diagnostics;
@namen = qw{ null eins zwei drei vier fuenf sechs sieben acht neun MZH };
@raum = @namen[-1, 5, 3, 0, 0];
print "Der Kurs findet statt in @raum.\n";
```

▶▶ `./array-slice.pl`

Die **runden Klammern** können **weggelassen** werden.

Achtung: Diese Indizierungs-**Notation** ist **anders als** die Notation **Dollar-Name-Indexausdruck**! Jetzt haben wir einen **Klammeraffen** statt eines **Dollar-Zeichens**.

Regel:

- „**Dollar-irgendwas**“: Liefert **Skalar**
- „**Klammeraffe-irgendwas**“: Liefert **Liste**

Man darf auch auf **Array-Slices** **schreiben**:

▶▶ view perl/passwd_realnames5.pl :

```
#!/usr/bin/perl -w -n
# The -n flag above loops over the input (= /etc/passwd) line by line.
use diagnostics;
chomp;
@zeile = split /:/;
if ($zeile[4] eq 'B. Mueller') {
    @zeile[4, -1] = ('B. Mueller-Luedenscheid', '/bin/tcsh');
}
$zeile = join ':', @zeile;
print "$zeile\n";
```

▶▶ ./passwd_realnames5.pl < ../sed/passwd

Dieses Programm kopiert die Eingabe in die Ausgabe und **ändert** dabei den **Nachnamen** von B. Müller **und** außerdem auch die **Login-Shell**.

3.13 Grundlagen der **Ein- und Ausgabe**

3.13.1 Der **Diamant-Operator**

Bisher hatten unsere Programme **immer von STDIN** gelesen. Die normalen Unix-Tools wie **cat**, **sed**, **grep** sind da **flexibler**: Wenn man **nichts** angibt, lesen sie auch von **STDIN**, aber **wenn** man einen **Dateinamen** angibt, lesen sie **aus** dieser **Datei**. Wenn man **mehrere Dateinamen** angibt, lesen sie **alle** diese Dateien nacheinander.

Beispiele:

▶▶ ./grep split

▶▶ ./grep split split-ws.pl

▶▶ ./grep split *

▶▶ ./grep split -

Wenn man ein **Minus** angibt, **liest** das von **STDIN**.

Das gleiche Verhalten wie diese Unix-Tools erreicht man in **perl** mit dem **Diamant-Operator** anstelle von **<STDIN>**:

▶▶ view perl/grep-poor.pl :

```

#!/usr/bin/perl -w
use diagnostics;
while (<>) {
    if ( /split/ ) {
        print;
    }
}

```

```

▶▶ ./grep-poor.pl
▶▶ ./grep-poor.pl split-ws.pl
▶▶ ./grep-poor.pl *
▶▶ ./grep-poor.pl -

```

Es wird ohne Unterbrechung aus allen Dateien nacheinander gelesen. Falls man wissen will, aus welcher Datei man gerade liest, steht dies in der Variablen \$ARGV:

```
▶▶ view perl/grep-poor2.pl :
```

```

#!/usr/bin/perl -w
use diagnostics;
while (<>) {
    if ( /split/ ) {
        print "$ARGV: $_";
    }
}

```

```
▶▶ ./grep-poor2.pl *
```

3.13.2 Die **Aufruf-Parameter**

Die **Aufruf-Parameter** eines Skripts werden **beim Programmstart** im Array @ARGV abgelegt. **Danach** kann man das Array **beliebig benutzen**.

Der **Diamant-Operator** benutzt z.B. @ARGV.

Wir können unser **obiges grep-Skript erweitern**, indem wir den **Suchausdruck** aus @ARGV holen: ▶▶ view perl/grep-poor3.pl :

```

#!/usr/bin/perl -w
use diagnostics;
$muster = shift @ARGV;
if (! defined $muster) {
    print "Aufruf: grep-poor <muster> [<datei> ...]\n";
    exit 1;
}
while (<>) {
    if ( /$muster/ ) {
        print "$ARGV: $_";
    }
}

```

►► ./grep-poor3.pl split *

Nachdem der **Suchausdruck entfernt** worden ist, wird der **Rest** der Liste dem **Diamant-Operator** übergeben.

Man kann auf diese Weise auch **Kommandozeilen-Optionen** realisieren:

►► view perl/grep-poor4.pl :

```

#!/usr/bin/perl -w
use diagnostics;
$verbose = 0;
if (($#ARGV >= 0) and $ARGV[0] eq "-v") {
    $verbose = 1;
    shift;
}
$muster = shift @ARGV;
if (! defined $muster) {
    print "Aufruf: grep-poor <muster> [<datei> ...]\n";
    exit 1;
}
while (<>) {
    if ( /$muster/ ) {
        if ($verbose) {
            print "$ARGV: ";
        }
        print;
    }
}

```

►► ./grep-poor4.pl split *

►► ./grep-poor4.pl -v split *

Anmerkung: Wenn man viele Kommandozeilenoptionen hat, dann empfiehlt es sich, ein

fertiges perl-Modul aus der Standard-Distribution zu nehmen (perldoc Getopt::Long und perldoc Getopt::Std).

3.13.3 Formatierte Ausgabe mit `printf`

Die aus C bekannte Funktion `printf` gibt es auch in perl.

Erster Parameter: Formatieranweisung.

Restliche Parameter: Auszugebende Werte

Es sind im wesentlichen **genau alle Formatieranweisungen** erlaubt, die es **auch in C** gibt. Eine Auswahl:

►► view perl/printf.pl :

```
#!/usr/bin/perl -w
use diagnostics;
printf "Eine Dezimalzahl: %d\n", 42;
printf "Eine Fließkommazahl: %f\n", 3.14159;
printf "Eine Fließkommazahl im wiss. Format: %e\n", 3.14159;
printf "Eine Fließkommazahl im passenden Format: %g\n", 3.14159;
printf "Eine Fließkommazahl im passenden Format: %g\n", 7980000000000000;
printf "Eine Integer-Zahl im passenden Format: %g\n", 42;
printf "Eine String: %s\n", "Hallo";
printf "Eine Zahl in fester Spaltenbreite : '%10d'\n", 42;
printf "Ein String in fester Spaltenbreite: '%10s'\n", "Hallo";
printf "Ein String linksbündig           : '%-10s'.\n", "Hallo";
printf "Eine Fließkommazahl mit Breite und Genauigkeit: '%10.2f'\n",
      3.14159;
printf "Ein einzelnes Prozentzeichen: %%\n";
```

►► ./printf.pl

3.14 Assoziative Felder (Hashes)

3.14.1 Was ist das?

Ein **Hash** (assoziatives Feld) ist **eine Art Feld**, dessen **Indizes nicht Zahlen** sind, sondern beliebige **Strings**.

Intern werden assoziative Felder **mit einer Hash-Tabelle implementiert**, daher der **Name**. Die Implementation ist **sehr effizient**, auch große Hashes sind kein Problem. Wie bei Listen gibt es **keine Längenbeschränkung**.

Die **Eigenschaften** eines Hashes ergeben sich aus dem bereits gesagten: Die **Werte** sind beliebige **perl-Skalare**, und sie müssen **nicht vom selben Typ** sein, wie bei Listen auch.

Mehrere **Einträge** dürfen den **gleichen Wert** haben. Die **Indizes** müssen dagegen **eindeutig** sein, zu einem Index-String kann es nur einen Wert geben.

3.14.2 **Wozu ist das gut?**

Beispiele:

- Zuordnung von **Login-Namen** zu **Real-Namen**
- Zuordnung von **Host-Namen** zu **IP-Adressen**
- Zuordnung von **IP-Adressen** zu **Host-Namen**
(Hashes gehen nur in *eine* Richtung)
- Zuordnung von **Variablennamen** zur **Häufigkeit**, mit der sie in einem Programm auftauchen

3.14.3 **Syntax**

Zugriff auf ein Feldelement:

```
$hashvar{$stringvar}
```

Jetzt mit **geschweiften Klammern**, sonst wie bei **Feldern**. Kann zum **Lesen** und zum **Zuweisen** benutzt werden.

Wird von einem **Index gelesen**, auf den **noch nicht geschrieben** wurde, gibt es **undef** zurück, analog **wie bei Listen**.

Zugriff auf ein ganzes Hash:

```
%hashvar
```

Hashes können **in Listen verwandelt** werden, und **umgekehrt**. Damit kann man **Listen-Literale** auch für Hashes verwenden:

```
%hashvar = ("foo", 35, "bar", 12.4, "baz", "hallo");
```

Die Liste enthält **Paare** von Schlüsseln und Werten.

Die **Reihenfolge der Paare** ist naturgemäß **egal**.

Wegen der Möglichkeit der Konversion in Listen können wir **Hashes leicht invertieren**:

```
%ip_addr = reverse %host_name;
```

Damit das sinnvoll klappt, **sollte das Hash eindeutig** (injektiv) sein. Sonst **gewinnt das letzte Paar** in der Liste. Da die Reihenfolge nicht definiert ist, ist das Ergebnis also **nichtdeterministisch**.

Damit man bei Listen-Literalen **nicht die Kommas abzählen** muß, darf man auch den **großen Pfeil** verwenden:

```
%hashvar = ("foo" => 35, "bar" => 12.4, "baz" => "hallo");
```

Der große Pfeil ist **einfach** eine **andere Schreibweise für ein Komma**. Das gilt für ganz perl.

Damit man das **schön zeilenweise** schreiben kann, darf am **Ende einer Liste** ein **überflüssiges Komma** stehen:

```
%hashvar = (  
    "foo" => 35,  
    "bar" => 12.4,  
    "baz" => "hallo",  
);
```

Ein **Beispiel** für eine **Anwendung**: Ein **Cross-Referenz-Programm**, das die **Anzahl der Vorkommen** der Variablennamen **zählt**:

cross-reference2.pl:

```
#!/usr/bin/perl -w  
use diagnostics;  
use English;  
undef $INPUT_RECORD_SEPARATOR;  
$prog = <STDIN>;  
$prog =~ s/\#.*$//m;  
@vars = split /[^\w%\@\$]+/, $prog;  
foreach $var (@vars) {  
    if ($var =~ /\^$(\w+)$/ ) {  
        $scalars{$1}++;  
    } elsif ($var =~ /\^@(\w+)$/ ) {  
        $arrays{$1}++;  
    } elsif ($var =~ /\^%(\w+)$/ ) {  
        $hashes{$1}++;  
    }  
}  
print "The alphanumeric scalar variables are:\n";  
while ( ($name, $num) = each %scalars) {  
    print "$name: $num\n";  
}  
print "\nThe alphanumeric array variables are:\n";  
while ( ($name, $num) = each %arrays) {  
    print "$name: $num\n";  
}  
print "\nThe alphanumeric hash variables are:\n";  
while ( ($name, $num) = each %hashes) {  
    print "$name: $num\n";  
}
```


Anmerkungen:

Indem wir `$INPUT_RECORD_SEPARATOR` **un-definieren**, lesen wir die **ganze Datei** als eine Zeile ein. (Kurzform wäre `$/`).

Beim **ersten Auftreten** eines **Variablennamens** greifen wir auf einen noch nicht vorhandenen Eintrag zu. Das ergibt `undef`. Wenn wir eine `undef`-Variable **inkrementieren**, ergibt das **1**.

Wir haben die **each-Funktion** benutzt, die wir **gleich** noch **erkären** werden.

Wir haben die **Suche auf Hash-Variablen** erweitert.

```
▶▶ ./cross-reference2.pl < cross-reference2.pl
```

3.14.4 **Funktionen** auf assoziativen Feldern

Es gibt eine Reihe von **Funktionen**, die **auf** einem **gesamten Hash** arbeiten.

Die Funktionen `keys` und `values` liefern die **Liste der Index-Strings** bzw. die **Liste der Werte**:

`keys-values.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
%realnames = (
    "brederek" => "Jan Bredereke",
    "mueller"  => "B. Mueller",
    "meyer"    => "L. Meyer",
    "schulze"  => "G. Schulze",
);
@logins = keys %realnames;
@realnames = values %realnames;
$logins = join ' ', @logins;
$realnames = join ' ', @realnames;
print "Login-Namen: $logins\n";
print "Real-Namen: $realnames\n";
```

```
▶▶ ./keys-values.pl
```

Frage: Wie sind die **Ergebnisse geordnet**? **Gar nicht**, und **anders sortiert** als sie eingegeben wurden! **Grund:** **Effiziente interne Speicherung**. Aber beide **Listen** sind **trotzdem korreliert**.

Im **skalaren Kontext** liefern `keys` und `values` die **Anzahl der Schlüssel-Wert-Paare** im Hash:

`keys-values2.pl`:

```

#!/usr/bin/perl -w
use diagnostics;
%realnames = (
    "brederek" => "Jan Bredereke",
    "mueller"   => "B. Mueller",
    "meyer"     => "L. Meyer",
    "schulze"   => "G. Schulze",
);
$login_num = keys %realnames;
$realname_num = values %realnames;
print "Login-Namen: $login_num\n";
print "Real-Namen: $realname_num\n";

```

►► ./keys-values2.pl

Ein **Hash** im **skalaren Kontext** liefert die boolsche Information, **ob** es **voll** ist **oder** **leer**:

hash-scalar.pl:

```

#!/usr/bin/perl -w
use diagnostics;
%realnames = ();
if (%realnames) { print "Voll\n"; } else { print "Leer\n"; }
%realnames = (
    "brederek" => "Jan Bredereke",
    "mueller"   => "B. Mueller",
    "meyer"     => "L. Meyer",
    "schulze"   => "G. Schulze",
);
if (%realnames) { print "Voll\n"; } else { print "Leer\n"; }
%realnames = ();
if (%realnames) { print "Voll\n"; } else { print "Leer\n"; }

```

►► ./hash-scalar.pl

Die Funktion **each** erlaubt, **über ein Hash** zu **iterieren**:

hash-each.pl:

```

#!/usr/bin/perl -w
use diagnostics;
%realnames = (
    "brederek" => "Jan Bredereke",
    "mueller"  => "B. Mueller",
    "meyer"    => "L. Meyer",
    "schulze"  => "G. Schulze",
);
while ( ($login, $realname) = each %realnames ) {
    printf "%-8s: %s\n", $login, $realname;
}

```

►► ./hash-each.pl

Jeder **Aufruf** von **each** liefert ein **Schlüssel-Wert-Paar** als **zweielementige Liste**.

Diese **Liste** wird hier **als Bedingung** in der **while-Schleife** verwendet, also in einem **skalaren Kontext**. Daher liefert sie die **Anzahl der Listenelemente**. Beim ersten Mal ist dies **2**, also **True**. **Am Ende des Hashes** liefert **each** die **leere Liste**, die Anzahl der Elemente ist **0**, und das ist **False**.

Die Information, **wo wir bei der Iteration gerade stehen**, ist in einem **Iterator** gespeichert, der **zu jedem Hash dazugehört**. Wir können also über verschiedene Hashes gleichzeitig iterieren.

Der **Iterator** wird **zurückgesetzt**, wenn:

- das **Ende erreicht** wurde,
- eine **neue Liste** an das Hash **zugewiesen** wurde, oder
- **keys** oder **values** **aufgerufen** wurde.

Das ist normalerweise genau das, **was man erwarten** würde.

Wenn die **Reihenfolge** der Iteration **wichtig** ist, dann iteriert man mit Hilfe von **keys** und sortiert die Schlüssel:

hash-iter-sort.pl:

```

#!/usr/bin/perl -w
use diagnostics;
%realnames = (
    "brederek" => "Jan Brederereke",
    "mueller"   => "B. Mueller",
    "meyer"     => "L. Meyer",
    "schulze"   => "G. Schulze",
);
foreach $login (sort keys %realnames) {
    printf "%-8s: %s\n", $login, $realnames{$login};
}

```

►► ./hash-iter-sort.pl

Oft ist es auch wichtig, **nach der Reihenfolge der Werte**, nicht der Schlüssel, eines Hashes **zu sortieren**.

Dies ist ein **Vorgriff** auf das Kapitel über Unterprogramme, und es verwendet fortgeschrittene Eigenschaften der `sort`-Funktion. Trotzdem ist dieses Sortieren so nützlich, daß es hier bereits eingeführt werden soll.

Man kann der `sort`-Funktion als Parameter ein **Vergleichs-Unterprogramm** mitgeben, um eine **eigene Sortierreihenfolge** zu definieren.

Für den **Spezialfall eines Hashes** kann das folgendermaßen geschehen:

hash-iter-sort-value.pl:

```

#!/usr/bin/perl -w
use diagnostics;
%scores = (
    "Jan Brederereke" => 42,
    "B. Mueller"      => 27,
    "L. Meyer"        => 1,
    "G. Schulze"      => 234,
);
# a sorting subroutine: first param in $a, second param in $b
sub by_score { $scores{$b} <=> $scores{$a} }
foreach $name (sort by_score keys %scores) {
    printf "%-16s: %s\n", $name, $scores{$name};
}

```

►► ./hash-iter-sort-value.pl

Das **Sortier-Unterprogramm** hat sinnvollerweise einen **Namen**, der an die **Sortierreihenfolge** erinnert. Die Parameter werden anders als sonst in `$a` und `$b` übergeben. Der „**Raumschiffoperator**“ `<=>` gibt **-1, 0 oder 1** zurück, je nachdem wie die beiden Zahlen sich zueinander verhalten. (**Für Strings** heißt das Gegenstück `cmp`.) Da wir `$b` mit `$a` vergleichen, haben wir eine **umgekehrte Sortierreihenfolge**.

Mit der Funktion `exists` kann man prüfen, ob es zu einem Schlüssel überhaupt einen Eintrag im Hash gibt:

`hash-exists.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
%user_ids = (
    "root"      => 0,
    "brederek" => 501,
    "mueller"   => undef,
);
print "Login      if(\%hash{\$key}) if(defined \%hash{\$key}) ".
      "if(exists \%hash{\$key})\n";
foreach $login (keys %user_ids, "meyer") {
    printf "%-8s: %-15s %-23s %-22s\n",
        $login,
        ( $user_ids{$login}      ? "true" : "false" ),
        ( defined $user_ids{$login} ? "true" : "false" ),
        ( exists $user_ids{$login} ? "true" : "false" );
}
```

►► `./hash-exists.pl`

Die **Nicht-Existenz** eines Schlüssels ist etwas anderes als ein `undef`-Wert für einen Schlüssel. (Und ein `undef`-Wert ist bekanntermaßen etwas anderes als ein `False`-Wert.)

Anmerkung: Wir haben eben den **bedingten Ausdruck** (`? :`) verwendet, der aus C übernommen wurde. Er entspricht `if-then-else`, liefert aber einen Wert zurück.

Die Funktion `delete` entfernt einen Schlüssel aus einem Hash (und ebenso seinen Wert):

`hash-exists2.pl`:

```

#!/usr/bin/perl -w
use diagnostics;
%user_ids = (
    "mueller" => 502,
);
print "Login      if(defined \%hash{\$key}) if(exists \%hash{\$key})\n";
foreach $login ("brederek", "mueller") {
    printf "%-8s: %-23s %-22s\n",
        $login,
        ( defined $user_ids{$login} ? "true" : "false" ),
        ( exists $user_ids{$login} ? "true" : "false" );
}
$user_ids{"brederek"} = undef;
delete $user_ids{"mueller"};
delete $user_ids{"schulze"};
print "Login      if(defined \%hash{\$key}) if(exists \%hash{\$key})\n";
foreach $login ("brederek", "mueller") { # NOT using (keys %user_ids) !
    printf "%-8s: %-23s %-22s\n",
        $login,
        ( defined $user_ids{$login} ? "true" : "false" ),
        ( exists $user_ids{$login} ? "true" : "false" );
}

```

▶▶ ./hash-exists2.pl

Man beachte, daß die Verwendung von (keys %user_ids) den **anderen Login** jeweils gar **nicht** mehr mit **ausgegeben** hätte.

Es gibt **keine Warnung**, wenn der **Schlüssel** gar **nicht** im Hash **vorhanden** war.

Interpolation in Doppel-Quote-Strings:

hash-interp.pl:

```

#!/usr/bin/perl -w
use diagnostics;
%user_ids = (
    "mueller" => 502,
);
$login = "mueller";
print "$login: $user_ids{$login}\n";
print "%user_ids\n";

```

▶▶ ./hash-interp.pl

Einzelne Elemente eines Hashes werden **wie erwartet interpoliert**. Ein **ganzes Hash** wird **nicht interpoliert**, das würde auch fast nie Sinn machen.

3.15 **Online-Aufgaben**

3.15.1 Aufgabe: Einfaches Adreßbuch

Schreibe ein einfaches **Adreßbuch**. Man soll Namen und Telefonnummern **eingeben und abfragen** können, sowie Einträge auch **löschen** können. Außerdem soll man eine **sortierte Liste aller Einträge** ausgeben können.

3.16 **Hausaufgaben**

3.16.1 Aufgabe: Web-Log-Auswertung

In der Datei `httpd_access_log.txt` (auf den WWW-Seiten des Kurses) findet man ein (gekürztes) **Zugriffs-Log eines Web-Servers**. Sie enthält **nur Angriffsversuche** von Skripten und Würmern. Schreibe ein `perl`-Skript, das **für jeden zugreifenden Rechner** zählt, **wie oft** insgesamt zugegriffen wurde. Es soll eine **nach Häufigkeit sortierte Liste** ausgegeben werden.

3.17 Unterprogramme

3.17.1 Syntax

Definition gekennzeichnet durch **Schlüsselwort** sub:

sub-marine.pl:

```
#!/usr/bin/perl -w
use diagnostics;
sub marine {
    $n++; # globale Variable $n
    print "Moin, Matrose Nr. $n!\n";
}
&marine;
&marine;
&marine;
&marine;
```

▶▶ ./sub-marine.pl

Aufruf gekennzeichnet durch **Kaufmanns-Und**. (Das Kaufmanns-Und kann man **oft auch weglassen**, aber es ist besser, es zu schreiben.)

Unterprogramme **müssen nicht vor** ihrer Benutzung **definiert** werden.

Unterprogramme haben wiederum einen **eigenen Namensraum**.

3.17.2 Rückgabewerte und der return-Operator

Unterprogramme haben **immer** einen **Rückgabewert**. Der **Wert** ist der **letzte Ausdruck, der berechnet** wurde:

return-last.pl:

```
#!/usr/bin/perl -w
use diagnostics;
sub summe_von_hugo_und_karl {
    print "Berechnung der Summe von \"$hugo und \"$karl.\n";
    $hugo + $karl;
    #### print "Berechnung beendet.\n";
}
$hugo = 3;
$karl = 5;
$summe = &summe_von_hugo_und_karl;
print "Die Summe ist $summe.\n";
```

▶▶ ./return-last.pl

Vorsicht dabei mit **Debugging**-Anweisungen!

►► ./return-last.pl ohne die Auskommentierung

Auch print hat einen **Rückgabewert**. (Er zeigt an, ob die Ausgabe erfolgreich war.)

Immerhin gibt perl eine **Warnmeldung**, wenn sie eingeschaltet sind.

Wenn ein Unterprogramm im **Listen-Kontext** berechnet wird, kann es **auch** eine **Liste zurückgeben**:

return-list.pl:

```
#!/usr/bin/perl -w
use diagnostics;
sub liste_von_hugo_bis_karl {
    if ($hugo < $karl) {
        $hugo .. $karl;
    } else {
        $karl .. $hugo;
    }
}
$hugo = 17;
$karl = 9;
@liste = &liste_von_hugo_bis_karl;
print "Die Liste ist '@liste'.\n";
```

►► ./return-list.pl

Man sieht übrigens, daß der **letzte berechnete Ausdruck** zählt, **nicht** die **letzte Zeile** des Unterprogramms.

Mit der **Funktion** wantarray kann man **herausfinden, welcher Kontext** gerade aktuell ist:

return-wantarray.pl:

```

#!/usr/bin/perl -w
use diagnostics;
sub hugo_bis_karl {
    if ($hugo < $karl) {
        if(wantarray) { $hugo .. $karl; }
        else          { $karl - $hugo; }
    } else {
        if(wantarray) { $karl .. $hugo; }
        else          { $hugo - $karl; }
    }
}
$hugo = 17;
$karl = 9;
@liste = &hugo_bis_karl;
print "Die Liste ist '@liste'.\n";
$differenz = &hugo_bis_karl;
print "Die Differenz ist $differenz.\n";

```

►► ./return-wantarray.pl

Eingebaute Funktionen wie wantarray werden **immer ohne Kaufmanns-Und** aufgerufen.

Mit dem **return-Operator** kann man sofort aus einem Unterprogramm zurückkehren:
return-operator.pl:

```

#!/usr/bin/perl -w
use diagnostics;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub element_nummer {
    foreach (0..$#namen) {
        if ($namen[$_] eq $such_name) {
            return $_
        }
    }
    return;
}
$nummer = &element_nummer;
print "Die Nummer von $such_name ist $nummer.\n";

```

►► ./return-operator.pl

Wenn man dem **return-Operator** **kein Argument** mitgibt, liefert er **undef** zurück.

►► ./return-operator.pl mit modifiziertem \$such_name

Im **Listen-Kontext** liefert er dann die **leere Liste**.

3.17.3 **Parameter**

Die **Kommunikation** von Werten **über globale Variablen** ist umständlich und **fehleranfällig**. Besser sind **Parameter** für die Funktionen.

Parameter werden **in runden Klammern** übergeben, wie bei den eingebauten Funktionen:

return-operator2.pl:

```
#!/usr/bin/perl -w
use diagnostics;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub element_nummer {
    $mein_such_name = $_[0];
    shift @_;
    @meine_namen = @_;
    foreach $name (0..$#meine_namen) {
        if ($meine_namen[$name] eq $mein_such_name) {
            return $name
        }
    }
    return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name ist $nummer.\n";
```

▶▶ ./return-operator2.pl

Das Unterprogramm **bekommt die Parameter im Array @_**.

Man kann auch eine **variable Anzahl von Parametern** übergeben, wie im obigen Beispiel. @_ ist eine Liste, daher ist das kein Problem.

Beim Aufruf wird der **alte Inhalt** von @_ **wegkopiert** und **beim Rücksprung zurückkopiert**. Analog wie bei foreach:

return-operator3.pl:

```

#!/usr/bin/perl -w
use diagnostics;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub string_ist_gleich {
    return($_[0] eq $_[1]);
}
sub element_nummer {
    $mein_such_name = $_[0];
    shift; # arbeitet auf @_
    foreach $name (0..$#_) {
        if (&string_ist_gleich($_[$name], $mein_such_name)) {
            return $name
        }
    }
    return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name ist $nummer.\n";

```

►► ./return-operator3.pl

Nach **Ende** des verschachtelten Unterprogramms wird die Variable `@_` restauriert, so daß sie **in der Schleife weiter benutzt** werden kann.

3.17.4 **Private Variablen**

Per Default sind **alle Variablen global**.

Mit dem **my-Operator** kann man **lexikalische** Variablen erzeugen, die **lokal zum aktuellen Block** sind, also z.B. zum aktuellen Unterprogramm:

return-operator4.pl:

```

#!/usr/bin/perl -w
use diagnostics;
my @namen;
my $such_name;
my $nummer;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub element_nummer {
    my $such_name;
    my @namen;
    my $name;
    $such_name = $_[0];
    shift @_;
    @namen = @_;
    $name = 0;
    while(@namen) {
        if($such_name eq pop @namen) {
            return $name;
        }
        $name++;
    }
    return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name in '@namen' ist $nummer.\n";

```

►► ./return-operator4.pl

Die lexikalischen Variablen des Unterprogramms **gelten hier bis zur schließenden Klammer**.

Die **am Anfang definierten lexikalischen Variablen** gelten für den **Rest der Datei**, da vorher kein Ende des Blocks kommt.

Wie man sieht, wird das **Array @namen der obersten Ebene nicht** durch das Unterprogramm **modifiziert**.

Man kann die **Definition** und die **erste Wertzuweisung zusammenfassen**. Man kann außerdem **mehrere Variablen gleichzeitig deklarieren**:

return-operator5.pl:

```

#!/usr/bin/perl -w
use diagnostics;
my @namen = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub element_nummer {
    my ($such_name, @namen) = @_;
    foreach $name (0..$#namen) {
        if ($namen[$name] eq $such_name) {
            return $name
        }
    }
    return;
}
my $nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name ist $nummer.\n";

```

►► ./return-operator5.pl

Jetzt muß die Liste der Variablen in **runden Klammern** stehen.

Fast **alle Unterprogramme** fangen so an.

Ein paar **Details zum Scoping**:

my-scope.pl:

```

#!/usr/bin/perl -w
use diagnostics;
while(defined(my $zeile = <>)) {
    print $zeile;
    if((my $antwort = <STDIN>) =~ /^good/i) {
        print "    Great!\n";
    } elsif ($antwort =~ /^bad/i) {
        print "    Oh, well..!\n";
    } else {
        chomp $antwort;
        print "**** '$antwort' is neither 'good' nor 'bad'.\n";
        exit 1;
    }
}

```

►► ./my-scope.pl my-scope.pl

Der **Scope** von \$zeile reicht **von** der **Schleifenbedingung** über die **ganze while-Schleife**.

Der **Scope** von \$antwort reicht **von** der **if-Bedingung** über **alle Blöcke** des if-elsif-else-Konstruktes.

Das **if-elsif-else-Konstrukt** ist hier **neu**. Es tut, was man denkt, daß es tut. Man beachte das **fehlende „e“!**.

Ein paar **Details zur Aufrufsemantik**: Die Listen-Elemente von `@_` werden per Referenz übergeben. Man kann also die Werte der aufrufenden Funktion verändern:

param-ref.pl:

```
#!/usr/bin/perl -w
use diagnostics;
my ($name1, $name2) = qw/ meyer mueller /;
sub upcase_in {
    for (@_) { tr/a-z/A-Z/ }
}
upcase_in($name1, $name2);
print "Die Namen sind '$name1' und '$name2'.\n";
```

►► ./param-ref.pl

Die Funktion `upcase_in` ändert also alle **übergebenen Variablen in Großbuchstaben**.

Es wird die Funktion `tr///` benutzt, auch `y///` genannt, die eine **zeichenweise Ersetzung** durchführt. Sie ist mit `s///` verwandt, aber ersetzt das erste Zeichen des ersten Musters durch das erste Zeichen des zweiten Musters usw.

Aber Achtung, man kann sich auch unerwünschte Effekte einhandeln:

param-ref2.pl:

```

#!/usr/bin/perl -w
use diagnostics;
my @namen = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub string_ist_gleich {
    return($_[0] eq $_[1]);
}
sub element_nummer {
    my $such_name = $_[0];
    shift; # arbeitet auf @_
    foreach $name (0..$#_) {
        if (&string_ist_gleich($_[$name], $such_name)) {
            $_[0] = 'YYYYYYYYYYY';
            return $name
        }
        $_[$name] = 'XXXXXXXXX';
    }
    return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name in '@namen' ist $nummer.\n";

```

►► ./param-ref2.pl

Die **Aufrufparameter** werden hier unerwartet **zerstört**.

Daher ist es gut, die **Parameter mit my zu kopieren**. Dann hat man „call by value“, nicht „call by reference“, und es kann nichts passieren:

param-ref3.pl:


```

#!/usr/bin/perl -w
use diagnostics;
my @namen = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub string_ist_gleich {
    return($_[0] eq $_[1]);
}
sub element_nummer {
    my $such_name = $_[0];
    shift;
    my @namen;
    foreach (@_) {
        @namen = (@namen, $_);
    }
    foreach $name (0..$#namen) {
        if (&string_ist_gleich($namen[$name], $such_name)) {
            $such_name = 'YYYYYYYYYYY';
            return $name
        }
        $namen[$name] = 'XXXXXXXXX';
    }
    return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name in '@namen' ist $nummer.\n";

```

►► ./param-ref3.pl

Man beachte, daß wir `@_` **elementweise kopieren mußten**. Hätten wir die ganze Liste kopiert, wären nur die Referenzen auf die Elemente kopiert worden.

Auch wenn man `@_` einen **ganz neuen Wert zuweist**, passiert **keine Modifikation**, weil nur die Elemente der Liste Referenzen sind, nicht die Liste selbst.

perl erlaubt **allgemein** auch die Verwendung von **Referenzen auf Variablen**, was eine Art **sicherer Ersatz für Pointer** ist. Aber das ist ein **fortgeschrittenes Thema**, auf das ich nicht eingehen werde.

3.17.5 Das Pragma `use strict`

Es ist **guter Programmierstil**, jede **Variable explizit** zu definieren.

Das schützt vor Schreibfehlern. Ein einzelner Tippfehler wird durch die **-w-Warnungen** erkannt. Aber wenn man die **falsche Schreibweise mehrfach** verwendet, **hilft das nicht**. Das folgende ist falsch:

return-operator6-wrong.pl:

```
#!/usr/bin/perl -w
use diagnostics;
my @namen_liste = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub element_nummer {
    my ($such_name, @namen) = @_;
    foreach $name (0..$#namen) {
        if ($namen[$name] eq $such_name) {
            return $name
        }
    }
    return -1;
}
my $nummer = &element_nummer($such_name, @namenliste);
print "Die Nummer von $such_name in '@namenliste' ist $nummer.\n";
```

▶▶ ./return-operator6-wrong.pl

Die falsche Schreibweise wird zweimal verwendet, deshalb erkennt `-w` das nicht. Die Deklaration der **richtigen Schreibweise** mit `my` wird **auch nicht beanstandet**, weil die Variable gleichzeitig auch **initialisiert** wird.

▶▶ ./return-operator6-wrong.pl mit `use strict`; vorneweg

Erst jetzt wird die Fehlerursache gemeldet.

▶▶ ./return-operator6-wrong.pl mit eingefügten Underscores

Faustregel: Für Programme, die länger als ein Schirm voll sind, sollte man **immer** `use strict` verwenden.

3.17.6 Parameter-Prototypen

Es ist auch möglich, **Prototypen für die Parameter** von Unterprogrammen zu definieren. Man kann damit z.B. festlegen, ob ein Skalar oder eine Liste erwartet wird (**skalarer oder Listen-Kontext**).

Die Syntax und Semantik sind aber **komplex**, und das ganze lohnt sich **nur, wenn** man **eigene perl-Module** schreibt. Daher lasse ich das Thema hier aus.

3.18 Weitere **Kontrollstrukturen**

Es gibt in `perl` eine Reihe von **alternativen Schreibweisen**, die sich oft besser lesen lassen.

3.18.1 Die unless-Anweisung

Manchmal ist es natürlicher, nicht `if(!...)` zu schreiben, sondern `unless(...)`:
`unless.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
unless ($nenner == 0) {
    print "$zaehler / $nenner = ", ($zaehler / $nenner), "\n"
}
```

▶▶ `./unless.pl`

Man darf dabei auch `else` verwenden:

`unless-else.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
unless ($nenner == 0) {
    print "$zaehler / $nenner = ", ($zaehler / $nenner), "\n"
} else {
    print "Division durch Null abgefangen!\n";
}
```

▶▶ `./unless-else.pl`

Aber **Vorsicht**: `unless-else` **kann** auch **verwirrend** sein.

3.18.2 Die until-Anweisung

Manchmal ist es natürlicher, nicht `while(!...)` zu schreiben, sondern `until(...)`:
`until.pl`:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
my $verhaeltnis = 0;
until ($zaehler < $nenner) {
    $zaehler -= $nenner;
    $verhaeltnis++;
}
print "Das ganzzahlige Verhaeltnis ist $verhaeltnis\n"

```

►► ./until.pl

3.18.3 Ausdruck-Modifikatoren

Man darf an das Ende eines einzelnen Ausdrucks einen Modifikator schreiben, was die Notation kompakter macht. Beispiel:

unless-mod.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
print "$zaehler / $nenner = ", ($zaehler / $nenner), "\n"
    unless ($nenner == 0);

```

►► ./unless-mod.pl

Hier **sieht** die Modifikator-Version auch **natürlicher aus** als die längere Variante mit geschweiften Klammern.

Die **Bedeutung** ist genau die **gleiche** wie vorher.

Ausnahme: Es beginnt **kein neuer Scope**.

Und es gibt hier **kein else**.

Das ganze geht auch für andere Konstrukte. Beispiel:

if-mod.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $tracing = 1;
my $x = 5;
print "\$x ist gleich '$x'\n"           if $tracing;
```

▶▶ ./if-mod.pl

Weiterhin gibt es es als **Modifikatoren**: **until**, **while** und **foreach**:

mod-misc.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $n = 2;
$n *= 2 until $n > 10;
print " ", ($n -= 2) while $n > 0;
print "\n";
print " $_" foreach (1..5);
print "\n";
```

▶▶ ./mod-misc.pl

Der **Operator** ***** multipliziert die Variable auf der linken mit dem Wert auf der Rechten. Wir suchen also die kleinste Zweierpotenz größer zehn.

Obwohl die **Schleifenbedingungen** jetzt **hinten** stehen, werden sie **trotzdem vor der Schleife** ausgeführt.

Man beachte, daß **vor dem Modifikator kein Semikolon** steht.

Damit es nicht gar zu kompliziert wird, ist es **nicht erlaubt, an einen Modifikator** noch einen **weiteren Modifikator hinten dran** zu schreiben. Ebenso kann man den Modifikator **nur an einen einzelnen Ausdruck** dran schreiben. Sonst muß man die normale Schreibweise nehmen.

Bei dem **foreach**-Modifikator kann man **keine andere Laufvariable** angeben, man muß mit **\$_** arbeiten.

3.18.4 „Nackte“ Blöcke

Wenn man von einer normalen **while-Schleife** das Schlüsselwort und die Bedingung **wegnimmt**, bekommt man einen **nackten Block**. Er wird genau **einmal ausgeführt**.

Sinnvoll ist das z.B., um einen **beschränkten Scope** einzuführen:

naked-block.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
my $bytenum;
{
    print "Anzahl der Bits = ";
    my $bitnum = <STDIN>;
    $bytenum = $bitnum >> 3;
    if ($bitnum % 8) {
        $bytenum++;
    }
}
print "Die Anzahl der noetigen Bytes ist $bytenum.\n";

```

►► ./naked-block.pl

Der **Scope** der Variablen `$bitnum` ist hier **sofort wieder zuende**, sobald sie nicht mehr gebraucht wird.

Der **Scope** einer Variablen sollte immer **so klein wie möglich** sein, um Programmierfehler zu vermeiden.

3.18.5 Die elsif-Anweisung

Die if-elsif-else-Anweisung hatten wir **bereits oben** bei den Details zum Scoping schon **eingeführt**.

3.18.6 Autoinkrement und Autodekrement

Das Post-Autoinkrement haben wir im Zusammenhang mit **Hashes schon gesehen**.

Es gibt **vier Operatoren**, die genau **wie in C** arbeiten:

`auto-inc.pl`:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
my $var = 1;
print "\$var == $var.\n";
print "++\$var == ", ++$var, ".\n";
print "\$var == $var.\n";
print "\$var++ == ", $var++, ".\n";
print "\$var == $var.\n";
#
print "--\$var == ", --$var, ".\n";
print "\$var == $var.\n";
print "\$var-- == ", $var--, ".\n";
print "\$var == $var.\n";

```

►► ./auto-inc.pl

Der **Auto-Inkrement-Operator** hat etwas zusätzliche **Magie**: Dies gilt nur, wenn die Variable **niemals als Zahl** verwendet worden ist. Wenn sie also ein String ist, und wenn sie nicht der leere String ist, und wenn sie sie auf das Muster

```
/^[a-zA-Z]*[0-9]*$/
```

paßt, dann wird zeichenweise, mit Übertrag, als **String inkrementiert**:

auto-inc2.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
my @strings = qw/ aa Az a0 99 zz /;
foreach my $var (@strings) {
    print "\$var = $var\n";
    print "++\$var == ", ++$var, "\n";
}

```

►► ./auto-inc2.pl

Dies ist **nützlich**, wenn man **fortlaufend neue Dateierweiterungen** generieren möchte.

Die **Dekrementoperatoren** sind **nicht magisch**.

3.18.7 Schleifensteuerung

Die **foreach**-Anweisung zum **Iterieren über eine Liste** ist bereits **bekannt**. Eine andere, **äquivalente Schreibweise** ist **for**.

Mit beiden kann man wie mit der **for**-Schleife von C iterieren:

for-loop.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
for(my $idx = 1; $idx <=5; $idx++) {
    print "$idx\n";
}
#
my $idx = 1;
while($idx <= 5) {
    print "$idx\n";
    $idx++;
}

```

►► ./for-loop.pl

Die for-Schleife lässt sich wie gezeigt **in eine while-Schleife auflösen**.

Ein kleiner **Unterschied** bleibt: Bei der for-Schleife bleibt eine **my-Deklaration** in der Initialisierung **lokal**.

Man kann an eine while-Schleife auch einen **continue-Block** anhängen. Er hat die gleiche **Wirkung** wie der **dritte Teil einer for-Schleife**:

continue-loop.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
for(my $idx = 1; $idx <=5; $idx++) {
    print "$idx\n";
}
#
my $idx = 1;
while($idx <= 5) {
    print "$idx\n";
} continue {
    $idx++;
}

```

►► ./continue-loop.pl

Das ist **wichtig**, wenn man die **Schleife vorzeitig neu starten** will, was mit den folgenden Konstrukten geht.

Der **next-Operator** erlaubt, **vorzeitig zur nächsten Iteration** überzugehen:

next-loop.pl:


```

#!/usr/bin/perl -w
use diagnostics;
use strict;
for(my $idx = 1; $idx <=10; $idx++) {
    if($idx % 2) { next; }
    print "$idx\n";
}
#
my $idx = 1;
while($idx <= 10) {
    if($idx % 2) { next; }
    print "$idx\n";
} continue {
    $idx++;
}

```

►► ./next-loop.pl

Bei einer **for-Schleife** wird sofort der **dritte Teil der Klammer** ausgeführt. Bei einer **while-Schleife** wird sofort der **continue-Block** ausgeführt, sofern vorhanden. In beiden Fällen kommt **danach** die **Schleifenbedingung**. Dies **entspricht** dem **continue-Konstrukt von C**.

Ein **Schleifenkonstrukt**, und auch ein nackter Block, kann **mit einem Label benannt** werden. Damit kann man **mehrere Blöcke auf einmal verlassen**:

last-loop2.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
ZEILE:
for(my $zeile = 1; $zeile <= 9; ((print "\n"), $zeile++)) {
SPALTE:
    foreach my $spalte (1..9) {
        next ZEILE if $spalte > $zeile;
        print "$spalte ";
    }
}

```

►► ./last-loop2.pl

Ein Label wird per Konvention in **Großbuchstaben** geschrieben. Es wird von einem **Doppelpunkt** gefolgt.

Man beachte, wie wir **zwei Anweisungen in den dritten Teil** der for-Schleife gepackt haben. Eine Alternative wäre eine while-continue-Schleife gewesen.

Der last-Operator dient dazu, eine **Schleife vorzeitig zu verlassen**:

last-loop.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $idx;
for($idx = 1; $idx <=10; $idx++) {
    print "$idx\n";
    if($idx == 5) { last; }
}
print "Index nach der Schleife: $idx\n";
#
$idx = 1;
while($idx <= 10) {
    print "$idx\n";
    if($idx == 5) { last; }
} continue {
    $idx++;
}
print "Index nach der Schleife: $idx\n";
```

►► ./last-loop.pl

Der **dritte Teil** der **for-Schleife** und der **continue-Block** werden dann **nicht** mehr ausgeführt.

Auch dies geht **über mehrere Ebenen**:

last-loop3.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my @namen1 = qw/ meyer mueller schulze /;
my @namen2 = qw/ lehmann mueller feldmann /;
NAMEN1:
foreach my $name1 (@namen1) {
    NAMEN2:
    foreach my $name2 (@namen2) {
        if($name1 eq $name2) {
            print "Der Name '$name1' kommt in beiden Listen vor.\n";
            last NAMEN1;
        }
    }
}
```

►► ./last-loop3.pl

Dies kann man **in C** mit `continue` **nicht mehr** machen!

Der `redo`-Operator dient dazu, an den **Anfang der aktuellen Iteration zurückzugehen**. Der dritte Iterations- bzw. der `continue`-Block wird dabei **nicht ausgewertet**. Auch die **Schleifenbedingung** wird **nicht nochmal ausgewertet**. Man braucht den Operator, wenn man die selbstgeschriebene Schleife belügen will:

`redo-loop.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo;
    }
    print "Die aktuelle Zeile ist: '$_'\n";
}
```

`redo-loop.txt`:

```
Dies ist die erste Zeile.
Zweite ... \
dritte ... \
und dies ist die vierte Zeile.
```

▶▶ `./redo-loop.pl redo-loop.txt`

Hier wird **zeilenweise eingelesen** und wieder ausgedruckt. **Ausnahme** ist, wenn eine Zeile auf **Backslash** endet, dann wird diese Zeile **mit der nächsten vereinigt**, ggf. auch mehrfach.

3.18.8 Der `do`-Block

Der `do`-Block verhält sich zunächst **fast** genau **wie** ein **nackter Block**. Wenn er durch einen **Schleifen-Modifikator** verändert wird, wird der **Block immer erst einmal ausgeführt**, bevor die **Schleifenbedingung** geprüft wird:

`do-block.pl`:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
my $var;
do {
    print "Eingabe = ";
    chomp ($var = <>);
} while($var != 0);
print "Ende der Schleife.\n";
do {
    print "Eingabe = ";
    chomp ($var = <>);
} until($var == 0);
print "Ende der Schleife.\n";

```

►► ./do-block.pl

Der do-Block **zählt nicht als Schleife**, so daß man **next**, **last** und **redo** **nicht** darin **anwenden** kann. Man kann **in ihn** (für **next**) oder um ihn (für **last**) aber einen **nackten Block** schreiben, der als Schleife zählt.

3.18.9 Logische Operatoren

Es gibt die logischen Operatoren **&&** (und), **||** (oder) und **!** (nicht):
log-op.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
for $a (0, 1) {
    for $b (0, 1) {
        print "$a && $b == '", ($a && $b), "'\n";
    }
}
print "\n";
for $a (0, 1) {
    for $b (0, 1) {
        print "$a || $b == '", ($a || $b), "'\n";
    }
}
print "\n";
for $a (0, 1) {
    print "! $a == '", (! $a ), "'\n";
}

```

►► ./log-op.pl

Die **Werte** sind die **erwarteten**. Allerdings liefert `!` den **leeren String** zurück, was aber auch `False` ist.

Die logischen Operatoren `&&` und `||` werden **von links nach rechts ausgewertet**. Was passiert, wenn das **Gesamtergebnis schon nach dem ersten Ausdruck klar** ist?

Die Auswertung erfolgt **nur soweit notwendig**, also **teilweise**:

log-op2.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Bitte sag 'ja': ";
if( defined ($_ = <>) && /^ja/i ) {
    print "\nDanke!\n";
} else {
    print "\nDann nicht.\n";
}
```

►► ./log-op2.pl

Hier wird der **rechte Teil nur** ausgewertet, **wenn** `$_` auch **definiert** ist.

Der logische Ausdruck hat **nicht nur** einen **logischen Wert**, sondern einen ganz **konkreten**: Den des **letzten ausgewerteten Ausdrucks**. Das ist **nützlich**, um z.B. einen **Default-Wert** zu wählen:

log-op3.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Nachname: ";
my $zeile = <>;
defined $zeile && chomp $zeile;
my $nachname = $zeile || '(Kein Nachname angegeben)';
print "Eingegebener Nachname: $nachname\n";
```

►► ./log-op3.pl

Man beachte, **daß jeder Wert**, der **False ergibt**, durch den **Default-Wert ersetzt** wird, auch z.B. `0`.

Manchmal muß man **viele Klammern** verwenden, weil die **Präzedenz** der logischen Operatoren zu hoch ist:

log-op4.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $a = '';
my $result;
($result = $a) || ($result = '(nichts)');
print "Ergebnis = $result\n";
```

▶▶ ./log-op4.pl

Anmerkung: Hier hätten wir natürlich auch `$result` nach links herausziehen können.

Für dieses Problem gibt es die logischen Operatoren `and`, `or` und `not`, die genauso funktionieren, aber eine ganz niedrige Präzedenz haben. Außerdem sind sie besser zu lesen:

log-op5.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $a = '';
my $result;
$result = $a or $result = '(nichts)'; # ausserdem gibt's: and, not
print "Ergebnis = $result\n";
```

▶▶ ./log-op5.pl

3.18.10 Der Bedingungsausdruck ?:

Aus C übernommen wurde der **Bedingungsausdruck**.

cond-op.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $a = 5;
my $b = 7;
my $max = $a > $b ? $a : $b;
print "Das Maximum von $a und $b ist $max.\n";
```

▶▶ ./cond-op.pl

Auch er wird nur **teilweise ausgewertet**:

cond-op2.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
my $zaehler = <>;
print "Nenner = ";
my $nenner = <>;
my $verhaeltnis = $nenner != 0 ? $zaehler / $nenner : 0;
print "Das Verhaeltnis ist $verhaeltnis.\n";

```

▶▶ ./cond-op2.pl

3.18.11 Die case/switch-Anweisung

In perl gibt es **keine** case/switch-Anweisung. Aber es gibt verschiedene Wege, sie **elegant zu simulieren**. Man nimmt dafür z.B. einen nackten Block:

switch.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Eingabe [Ja/Nein/Weiss nicht] = ";
$_ = <>;
SWITCH: {
    /^ja/i && do { print "Fein!\n"; last SWITCH;};
    /^Nein/i && do { print "Schade!\n"; last SWITCH;};
    /^Weiss nicht/i && do { print "Merkwuerdig!\n"; last SWITCH;};
    print "Illegale Eingabe!\n"; exit 1;
}
print "Normales Programmende.\n";

```

▶▶ ./switch.pl

Die **do-Blöcke** sind **hier notwendig**, weil sie einen Wert liefern, während ein nackter Block eine separate Anweisung gewesen wäre.

Man kann **auch** den **Bedingungsausdruck** verwenden:

switch2.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
print "Groesse als Zahl = ";
my $zahl = <>;
my $groesse =
    $zahl < 10 ? 'klein' :
    $zahl < 20 ? 'mittel' :
    $zahl < 50 ? 'gross' :
                'uebergross';
print "Die Groesse ist $groesse.\n";
```

▶▶ ./switch2.pl

3.19 **Datei-Handles und Datei-Tests**

3.19.1 Öffnen, Lesen und Schließen einer Datei

Ein **Datei-Handle** ist eine **Verbindung zur Außenwelt**, z.B. zu einer Datei. Wir haben bereits eines kennengelernt: `STDIN`.

Weiterhin gibt es die Datei-Handles `STDOUT` und `STDERR`. Ihre **Bedeutung ist wie in C** und sollte bekannt sein.

Außerdem hatten wir bereits den **Diamant-Operator** `<>` kennengelernt. Er ist eine **Kurzschreibweise für** `<ARGV>`, was genau die **bekannte Magie** hat.

Die Funktion `print` kann als **erstes Argument** ein **Datei-Handle** nehmen, dann geht die Ausgabe der folgenden Liste dorthin:

`print-handle.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
print STDOUT "Normale Ausgabe.\n";
print STDERR "Ausgabe auf Fehler-Handle.\n";
```

▶▶ `./print-handle.pl`

▶▶ `./print-handle.pl >/dev/null`

▶▶ `./print-handle.pl 2>/dev/null`

Das gleiche geht **auch mit printf**.

Man beachte, daß das **Datei-Handle nicht** von einem **Komma** gefolgt ist, damit es nicht Teil der Liste wird.

`STDOUT` wäre im obigen Beispiel das **Default** und könnte auch weggelassen werden.

Diese drei Datei-Handles sind **automatisch offen**. Man kann aber **weitere öffnen**:
`open-ex.pl`:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
open TEXT, "redo-loop.txt";
close TEXT;
#
my $filename = "redo-loop.txt";
open TEXT, "< $filename";
open AUSGABE, "> ausgabe.txt";
while(<TEXT) {
    print AUSGABE $_;
}
close TEXT;
close AUSGABE;
#
open AUSGABE, ">> ausgabe.txt";
    print AUSGABE "P.S.: Diese Zeile wurde angehaengt.\n";
close AUSGABE;

```

Die Grundform von `open` hat **zwei Argumente**, ein Datei-Handle und einen Dateinamen. Per Konvention werden Datei-Handles **immer groß** geschrieben. Da sie **nicht** von einem **Sonderzeichen eingeleitet** werden, ist so garantiert, daß sie nicht mit gegenwärtigen oder zukünftigen Schlüsselwörtern **kollidieren**.

Mit `close` wird ein offenes Datei-Handle wieder **geschlossen**.

Per **Default** wird eine Datei **zum Lesen** geöffnet. Man kann das aber **auch explizit** angeben indem man ein **<-Zeichen** vor den Dateinamen stellt. Das schützt vor Problemen, wenn der **Dateiname** aus einer **Variablen** kommt, die vielleicht mit einem **>-Zeichen** beginnt.

Mit dem **>-Zeichen** wird eine **Datei zum Schreiben** geöffnet. Sie wird dabei entweder **angelegt** oder ggf. **auf Länge Null** gekürzt.

Mit dem **doppelten >-Zeichen** wird eine Datei ebenfalls zum Schreiben geöffnet. Aber wenn die Datei existiert, wird an sie **angehängt**, anstatt sie auf Länge Null zu setzen. Das ist **nützlich für Log-Dateien**.

▶▶ `./open-ex.pl`

▶▶ `cat ausgabe.txt`

Diese ganze **Syntax** ist bewußt an die **Shell-Syntax** angelehnt.

Wenn vor oder nach dem **Dateinamen Whitespace** steht, wird dieser von `perl` automatisch **entfernt**. Das ist z.B. praktisch, wenn eine **Variable** mit dem Dateinamen noch ein **Newline am Ende** enthält. **Will man das nicht**, so muß man **vor** die **Variable** ggf. `./` setzen und **ans Ende** ein `\0`-Byte anhängen.

Das **Schließen** einer Datei-Handle **kann man auch weglassen**. Bei **Programmende** werden alle **automatisch** geschlossen. **Auch** bei einem **neuen Öffnen** wird das Datei-Handle ggf. erst geschlossen. Trotzdem ist das Schließen **nützlich**, um die **Puffer sofort rauszuschreiben**.

Datei-Handles werden **benutzt, wie** es schon von <STDIN> und print STDOUT ... **bekannt** ist.

Das **Öffnen** einer Datei **kann schiefgehen**, z.B. wg. eines falschen Dateinamens oder wegen fehlender Rechte.

Wenn man vom zugehörigen Datei-Handle **hinterher** zu **lesen** versucht, bekommt man sofort ein **Datei-Ende**. Wenn man zu **schreiben** versucht, wird die **Ausgabe weggeworfen**. Falls **Warnungen** eingeschaltet sind, werden wir allerdings gewarnt.

Aber man kann auch den **Return-Wert** von open **abprüfen**:

open-unless.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $infilename = "redo-loop.txt";
my $success =
    open TEXT, "< $infilename";
unless($success) {
    print "Kann Datei '$infilename' nicht oeffnen!\n";
    exit 1;
}
my $outfilename = "ausgabe.txt";
my $success =
    open AUSGABE, "> $outfilename";
unless($success) {
    print "Kann Datei '$outfilename' nicht oeffnen!\n";
    exit 1;
}
while(<TEXT>) {
    print AUSGABE $_;
}
close TEXT;
close AUSGABE;
```

3.19.2 Programmabbruch mit die

Diese **Prüfung** geht aber noch **wesentlich eleganter**.

Perl kann ohnehin ein Programm **mit einer Fehlermeldung** auf STDERR **beenden**,

wenn notwendig. Beispiel: **Division durch Null**.

Diese Funktionalität kann man mit der **die**-Funktion auch **selbst nutzen**. Außerdem ist der **or-Operator** hier sehr nützlich:

open-die.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
open TEXT, "< xyzzy" or die "Can't open file 'xyzzy'";
while(<TEXT>) {
    print;
}
```

Die **die**-Funktion **druckt** ihr **Argument** und **beendet** das Programm dann mit einem **Nicht-Null-Exit-Code**.

Anmerkung: **Wichtig** ist hier die **niedrige Präzedenz** des **or**-Operators. Der **||**-Operator hätte hier eine **Auswahl zwischen** dem **Dateinamen** und der nachfolgenden **Funktion** gemacht, die immer zugunsten des Dateinamens ausgegangen wäre. Es wäre also effektiv **gar keine Prüfung** durchgeführt worden.

▶▶ ./open-die.pl

Es wird die **aktuelle Zeilennummer** des **perl-Skripts** mit ausgegeben, und ggf. auch die Zeilennummern aller **offenen Eingabedateien**. Das ist gut fürs **Debugging**. Wenn der Fehler ggf. aber beim **Benutzer** liegt, will man das **nicht**. Dann sollte man ein **Newline** ans Ende der Meldung schreiben:

open-die2.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use English;
use strict;
open TEXT, "< xyzzy" or die "Can't open file 'xyzzy': $ERRNO\n";
while(<TEXT>) {
    print;
}
```

▶▶ ./open-die2.pl

Wie wir sehen, wird die **Variable \$ERRNO** durch die **Fehlermeldung des Betriebssystems** ersetzt. Es wird einfach die **letzte gesetzte Fehlermeldung** genommen. Falls **kein Fehler** auftritt, wird die Variable **nicht verändert**. Wir haben hier die Langform benutzt, die **Kurzform** wäre **\$!**. Außerdem bekommen wir **durch use diagnostics** noch eine **ausführliche Erläuterung** des Fehlers.

Die Funktion **warn** ist eine **Variante** von **die**, die das Programm **nicht beendet**. Es wird **nur** eine **Warnmeldung** auf **STDERR** gedruckt.

3.19.3 Öffnen von Pipes

Man kann auch von **Pipes lesen** und auf Pipes **schreiben**:

open-pipe.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
open INTEXT, "gunzip -c numbers.txt.gz |";
open OUTTEXT, "| gzip > numbers-new.txt.gz";
while(<INTEXT>) {
    s/7/sieben/g;
    print OUTTEXT $_;
}
```

▶▶ ./open-pipe.pl

Ein **Pipe-Symbol am Ende** zeigt an, daß das vorstehende ein Kommando ist, das die **Eingabe** für uns **liefern** soll. Ein Pipe-Symbol **am Anfang** bedeutet, daß dieses ein Kommando ist, das unsere **Ausgabe aufnimmt**.

Das erste Kommando **dekomprimiert** die Eingabe, das zweite **komprimiert** sie wieder.

Wenn **Shell-Metazeichen** im Kommando vorkommen, wird eine Shell gestartet, um sie zu interpretieren. Beispiel hier: **Ausgabeumleitung**.

Dies alles gilt auch für das Öffnen des impliziten Datei-Handles **ARGV** des Diamant-Operators. Deshalb kann jedes perl-Programm, das den Diamant-Operator benutzt, automatisch auch die Eingabe aus einer Pipe holen. Auf Seite 62 hatten wir ein solches Programm vorgestellt.

▶▶ view my-scope.pl

▶▶ ./my-scope.pl "gunzip -c numbers.txt.gz |"

Lesen und Schreiben auf **Pipes** hat allerdings einen **Nachteil**: Es ist **schwer festzustellen, ob** das Programm **erfolgreich gestartet** wurde. Meist bekommt man aus internen Gründen **keine Fehlermeldung**.

Mehr Informationen zum Datei-Öffnen gibt es in man perlopentut.

3.19.4 Datei-Tests

Oft muß man **prüfen, ob** eine **Datei existiert**:

ftest-exist.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
use English;
my $outname = 'output.txt';
die "File $outname already exists!\n"
    if -e $outname;
open OUTTEXT, "> $outname" or die "Can't open $outname: $ERRNO\n";
print OUTTEXT 'bla';

```

►► ./ftest-exist.pl

►► ./ftest-exist.pl

Ein Datei-Test **beginnt** immer **mit** einem **Minus**, auf das ein Buchstabe folgt.

Man kann sie auf **Dateinamen** oder auf **Datei-Handles** anwenden.

Es gibt eine ganze Reihe **weiterer** Datei-Tests:

Datei-Tests	
Datei-Test	Bedeutung
-r	lesbar (readable)
-w	schreibbar (writable)
-x	ausführbar (eXecutable)
-o	gehört diesem User (owner)
-e	existiert (exists)
-z	existiert und hat Länge Null (zero)
-s	existiert und hat Länge größer Null, Wert ist Länge (size)
-f	normales File (file)
-d	Verzeichnis (directory)
-l	symbolischer Link (link)
-t	ist ein TTY [nur für Datei-Handles] (tty)
-T	Text-Datei, vermutlich (text)
-B	Binär-Datei, vermutlich (binary)
-M	Änderungs-Alter, in Tagen (modification)
-A	Zugriffs-Alter, in Tagen (access)
-C	Inode-Änderungs-Alter, in Tagen (change)
...	...

Die **Lese-/Schreib-Tests** prüfen **nur** die **Zugriffsbits**. Beim späteren Zugriff kann natürlich immer noch etwas **schiefgehen**.

Der **Normal-File-Test** **-f** ist **auch** wahr, wenn es sich um ein **symbolisches Link** auf ein normales File handelt.

Der **TTY-Test** kann **nur** auf **Datei-Handles** angewandt werden. Bei Files wäre er immer False. Er ist nützlich, wenn man z.B. von STDIN wissen will, ob das Programm **interaktiv** laufen kann.

Die **Text-/Binär-Tests** **öffnen** die **Datei** und schauen sich den Anfang an. **Frage: Wann** sind diese beiden Tests **nicht komplementär**? Wenn die Datei **nicht existiert** (beide False) oder wenn sie **leer** ist (beide True).

Die **Datei-Alter-Tests** geben eine **Fließkommazahl** zurück. Zwischen kurz vor **Mitternacht** und kurz nach Mitternacht liegt also ein Wert viel kleiner als 1.0.

Weiterhin messen die **Datei-Alter-Tests** **relativ** zu dem Zeitpunkt, an dem das **perl-Skript** gestartet wurde. Das ist **sinnvoll**, wenn man **relative Datei-Alter** **vergleichen** will. Dann sollte man den gleichen Bezugspunkt nehmen. Eine **Konsequenz** ist, daß man auch ein **negatives Alter** bekommen kann, wenn das Programm schon eine Weile läuft. Man kann den **Bezugszeitpunkt** aber auch **aktualisieren**, durch `$^T = time;`

Wenn der Datei-Test **keinen Parameter** hat, arbeitet er auf der **Default-Variablen** `$_`. **Vorsicht** dabei **mit Mehrdeutigkeiten** bei dem, was folgt. Will man z.B. die **Dateigröße** in Bytes **durch 1000 teilen**, könnte der **folgende Schrägstrich** auch als Dateiname gedeutet werden. Am besten setzt man **Klammern** um einen Datei-Test ohne Parameter.

Datei-Tests **kosten** relativ viel **Zeit**. Wenn man **mehrere Tests** auf der gleichen Datei oder dem gleichen Handle macht, kann man auf die **gepufferten Daten** des ersten Tests zurückgreifen. Dies geht über das magische Datei-Handle namens `_`. Beispiel:

`ftest-repeat.pl:`

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
use English;
foreach (@ARGV) {
    print "$_\n"
        if (-T) and -s _ > 300 and -M _ > 14;
}
```

►► `./ftest-repeat.pl *`

Dies **findet** alle **Text-Dateien** im aktuellen Verzeichnis, die **länger als 300 Bytes** sind und innerhalb der letzten **zwei Wochen nicht verändert** worden sind.

3.20 **Module**

3.20.1 **Benutzung** einfacher Module

Perl bietet **sehr viele vorgefertigte Module**. **Warum** sollte man sie **benutzen**?

Beispiel: basename-Program. Es soll den Dateinamen ohne den Verzeichnispfad davor liefern:

`basename-simple.pl:`

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
my $name = shift @ARGV;
defined $name or die "Aufruf: basename-simple <dateiname>\n";
$name =~ s#.#/##;
print "$name\n";

```

▶▶ ./basename-simple.pl /usr/bin/perl

Dieses Programm **prüft sogar** ab, ob ein **Parameter** angegeben ist.

Aber es hat **mehrere Probleme**. Welche?

Erstens: Ein **Verzeichnisname** könnte ein **Newline** enthalten:

▶▶ ./basename-simple.pl '/usr/strange und ▶▶ dir/bin/perl' (d.h. mit Newline)

Anmerkung: Mit dem **Modifikator /s** könnte man dieses Problem **beheben**.

Zweitens: Dies ist **Unix-spezifisch**. perl läuft z.B. aber auch unter den Microsoft-Betriebssystemen, MacOS und VMS und AmigaOs. Alle haben andere Konventionen über Dateipfade (Backslash, zwei Doppelpunkte, eckige Klammern).

Drittens: Jemand hat dieses **Problem bereits einmal gelöst**.

Es gibt eine ganze Anzahl an **Standardmodulen**, die jeder perl-Distribution beiliegen. Und es gibt noch viel mehr Module im **Comprehensive Perl Archive Network (CPAN)**.

Liste der Standardmodule:

▶▶ man perlmodlib ab „Standard Modules“

Beispiel: File::Basename

Doku dazu:

▶▶ perldoc File::Basename

Damit sieht unser Programm so aus:

basename-mod.pl:


```

#!/usr/bin/perl -w
use diagnostics;
use strict;
use File::Basename;
my $name = '/usr/bin/perl';
my $basename = basename $name;
print "Der Unix-Basename von '$name' ist '$basename'\n";
fileparse_set_fstype 'VMS';
$name = 'Doc_Root:[Help]Rhetoric.Rnh';
$basename = basename $name;
print "Der VMS-Basename von '$name' ist '$basename'\n";

```

Die **neuen** Funktionen werden **aufgerufen**, als wären sie **eingebaute** Funktionen.

►► ./basename-mod.pl

Jetzt ist das Programm **sofort nach VMS portabel**, wie man sieht.

(Anmerkung: Das File::Basename-Modul (und „Learning Perl“) hat einen Bug: Newlines in Verzeichnisnamen behandelt es falsch, es fehlt der /s-Modifikator! Trotzdem...)

Das Modul File::Basename **exportiert** noch **einige weitere Funktionen**, die wir hier nicht benutzen, z.B. `dirname`.

Was ist, wenn wir **schon** eine **lokale Funktion** `&dirname` haben, z.B. für „**Direction-Name**“? Namenskollision!

Man kann auch **nur** eine **ausgewählte Liste** von Funktionen aus einem Modul **importieren**:

basename-mod2.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
use File::Basename qw/ basename fileparse_set_fstype /; # <----- !
my $name = '/usr/bin/perl';
my $basename = basename $name;
print "Der Unix-Basename von '$name' ist '$basename'\n";
fileparse_set_fstype 'VMS';
$name = 'Doc_Root:[Help]Rhetoric.Rnh';
$basename = basename $name;
print "Der VMS-Basename von '$name' ist '$basename'\n";

```

Was ist, wenn wir **nicht einmal** die **benötigten** Funktionen importieren wollen oder können? **Leere Liste importieren** und über den **vollen Namen** der Funktionen zugreifen:

basename-mod3.pl:

```

#!/usr/bin/perl -w
use diagnostics;
use strict;
use File::Basename qw/ /;
my $name = '/usr/bin/perl';
my $basename = File::Basename::basename $name;
print "Der Unix-Basename von '$name' ist '$basename'\n";
File::Basename::fileparse_setfstype 'VMS';
$name = 'Doc_Root:[Help]Rhetoric.Rnh';
$basename = File::Basename::basename $name;
print "Der VMS-Basename von '$name' ist '$basename'\n";

```

Weitere Dokumentation zu Modulen findet man in `man perlmod`.

3.20.2 Einige wichtige Standardmodule

Einige wichtige Standardmodule	
Modul	Beschreibung
Cwd	get pathname of current working directory
Fatal	make errors in builtins or Perl functions fatal
File::Basename	split a pathname into pieces
File::Copy	copy files or filehandles
File::Find	traverse a file tree
File::Path	create or remove a series of directories
File::Spec	portably perform operations on file names
Getopt::Long	extended processing of command line options
Getopt::Std	process single-character switches with switch clustering
I18N::Collate	compare 8-bit scalar data according to the current locale
Math::BigFloat	arbitrary length float math package
Math::BigInt	arbitrary size integer math package
POSIX	asin, cosh, floor, frexp, isupper, isalpha, creat, open, asctime, clock, ...
Term::ReadLine	interface to various readline packages
Sys::Hostname	try every conceivable way to get hostname
Text::Tabs	expand and unexpand tabs per the Unix <code>expand(1)</code> and <code>unexpand(1)</code>
Text::Wrap	line wrapping to form simple paragraphs
Time::Local	efficiently compute time from local and GMT time

(Man bekommt die **vollständige Liste** der Standardmodule mit `man perlmodlib`.)

3.20.3 Das Comprehensive Perl Archive Network (CPAN)

Hier gibt es **viele weitere Module** zum **Herunterladen**.

Adresse: www.cpan.org

Man kann nach diversen Kategorien und Stichworten **suchen**.

Bei der Installation hilft man `perlmodinstall`.

Wenn man unter Unix ist, kann man das **Herunterladen** und das **Installieren** mit dem CPAN-Modul automatisieren. Siehe `perldoc CPAN`.

3.21 **Objekt-Orientierung?**

Gibt es in perl. Aber um sie einführen zu können, **braucht man** eine Reihe von **Konzepten**, die selbst **noch nicht eingeführt** worden sind. Beispiel: Referenzen.

Ein Tutorial findet man in `man perltoot`.

Ansonsten gelten die **Grundregeln**:

1. Ein **Objekt** ist eine **Referenz**, die weiß, **zu welcher Klasse** sie gehört.
2. Eine **Klasse** ist ein **Package**, das **Methoden liefert**, die mit Objekt-Referenzen umgehen können.
3. Eine **Methode** ist ein **Unterprogramm**, das eine Objekt-Referenz **als erstes Argument** erwartet (oder ein Package-Name).

Dies wird ausführlich in `man perlobj` erläutert.

3.22 **Prozeß-Management**

3.22.1 Die `system`-Funktion

Man kann aus einem perl-Programm heraus **andere Programme starten**. (Diese Programme sind dann natürlich **betriebssystem-spezifisch**.)

`system.pl`:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
system "date";
system 'ls -l $HOME/.skel';
system "xeyes &";
```

Die Funktion heißt `system`. Der erste Aufruf gibt das Datum mit der Unix-Funktion `date` aus.

Wenn man **Shell-Variablen** benutzt, sollte man in perl **einfache Anführungszeichen** nehmen, wie im zweiten Aufruf.

Wenn man **Unix-Shell-Metazeichen** verwendet, wie etwa das **Kaufmanns-Und** im dritten Aufruf, wird automatisch eine **Unix-Shell zur Ausführung** benutzt.

▶▶ ./system.pl

Aber **Vorsicht:**

system2.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $filename = shift;
chomp $filename;
system "ls -l $filename";
```

▶▶ ./system2.pl 'system2.pl ; xeyes &'

Die **Benutzereingabe** liefert **Shell-Metazeichen**. Die `system`-Funktion interpretiert sie. Bei entsprechenden Eingaben kann dies böse enden. Ggf. **auch aus Versehen**.

Man kann die Shell vermeiden, wenn man `system` mit **mehr als einem Argument** aufruft:

system3.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my $filename = shift;
chomp $filename;
system "ls", "-l", $filename;
```

▶▶ ./system3.pl 'system2.pl ; xeyes &'

3.22.2 Ausgaben einfangen mit **Backquotes**

Man kann die **Standard-Ausgabe** eines **externen Programms** in eine Variable **einfangen**:

backquote.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
chomp(my $systemdate = `date`);
print "Das System denkt, dass das Datum '$systemdate' ist.\n";
```

▶▶ ./backquote.pl

Man beachte, daß `perl`, anders als die Shell, das **letzte Newline nicht selbsttätig entfernt**.

Die Backquotes **entsprechen** der **Ein-Argument-Form** von `system`, was die Interpretation von Shell-Metazeichen betrifft. Es gibt kein Gegenstück zur Mehr-Argument-Form.

Der **Standard-Input** des externen Programms **bleibt** mit dem Standard-Input des **perl**-Programms verbunden, wie bei der Shell-Version der Backquotes auch. **Ebenso** die **Fehler-Ausgabe**.

Man kann die Backquotes auch **im Listenkontext** verwenden:

backquote2.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
my @w_list = 'w';
foreach (@w_list) {
    if( /^( [^ ]+ ) +pts.{17} {6}.* / ) {
        print "$1\n";
    }
}
```

►► ./backquote2.pl

Wir bekommen ein **Feld von Ausgabszeilen**. Wir **bearbeiten** es hier weiter, **wie** wir es **damals** mit **sed** getan haben. (Das **w**-Kommando liefert u.a. die Liste der **eingeloggten Benutzer**.)

3.23 Ausblick auf einiges **fortgeschrittenes perl**

Prozeß-Signale: Unix-Prozesse kennen sog. Signale, z.B. SIGINT (= Ctrl-C). Ein Unix-Programm kann sie z.T. **abfangen und darauf reagieren**. Ein **perl**-Programm kann das ebenfalls, indem es in dem Hash **%SIG** den Namen eines Unterprogramms als Wert für den Namen des Signals einträgt, also z.B. für **INT**.

Damit kann man z.B. auch gut **Temp-Dateien löschen** lassen.

Environment-Variablen: Unix-Prozesse haben Environment-Variablen, z.B. **\$HOME** und **\$PATH**. Mit dem Hash **%ENV** kann man darauf zugreifen.

Datenbanken: Man kann ein **Hash an eine Datenbank binden**. Damit bleibt der Inhalt über den Programmablauf hinaus erhalten. Eine **einfache Datenbank** wird mit **perl** **mitgeliefert**, aber man **kann auch andere nehmen**, wie Oracle, Sybase, Informix, **mySQL** und andere.

Editieren von Dateien am Platz von der Kommandozeile: Die Kommandozeilenoption **-i** bewirkt, daß die Dateien des Diamant-Operators **<>** am Platz editiert werden:

in-place.sh:

```
#!/bin/sh
perl -p -i.bak -w -e 's/Zeile/Textzeile/' redo-loop.txt
```

►► ./in-place.sh

Die Option `-p` erzeugt die **bekannte Schleife** um das Programm. Falls man **nur** `-i`, ohne `.bak`, verwendet, gibt es **keine Backup-Datei**.

Abfangen von Programmabbrüchen mit eval Wenn ein Programm z.B. bei einer **Division durch Null** nicht abbrechen soll, kann man die fraglichen Kommandos innerhalb eines eval-Blockes ausführen lassen. Die Variable `$@` enthält hinterher ggf. die Fehlermeldung, sonst ist sie leer.

Elemente einer Liste mit grep auswählen: Eine Kurzform zum effizienten Auswählen:

grep.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
use English;
{
    my @odd_numbers;
    foreach (1..50) {
        push @odd_numbers, $_ if $_ % 2;
    }
    print "@odd_numbers\n";
}
{
    my @odd_numbers = grep { $_ % 2 } 1..50;
    print "@odd_numbers\n";
}
{
    open FILE, 'redo-loop.txt' or die "Can't open: $ERRNO\n";
    my @matching_lines = grep { /\bZeile\b/i } <FILE>;
    print @matching_lines;
}
{
    open FILE, 'redo-loop.txt' or die "Can't open: $ERRNO\n";
    my @matching_lines = grep /\bZeile\b/i, <FILE>;
    print @matching_lines;
}
```

Die **erste Form** ist nicht falsch, aber etwas **ineffizient**. Der **grep-Operator** wählt aus einer Liste aus. Im Block wird `$_` an das **aktuelle Element gebunden**. Wenn der Block **True** ergibt, kommt das Element in die Ergebnisliste.

Der **Name** des Operators kommt vom entsprechenden Unix-Kommando. `perl's grep` ist allerdings mächtiger. Die **dritte Form** zeigt einen entsprechenden Einsatz.

Es gibt auch eine **einfachere Syntax**, die die **vierte Form** zeigt. Wenn man nur einen einfachen Ausdruck braucht, kann man die geschweiften **Klammern weglassen** und ein **Komma** schreiben.

Elemente einer Liste transformieren mit map:

Ähnlich wie **grep**, aber es wird **nicht ausgewählt**, sondern **elementweise transformiert**:

map.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
use English;
my @text = qw/ Dies ist ein Test /;
my @neu_text = map { "\u$_" } @text;
print "@neu_text\n";
#
@neu_text = map "\u$_", @text;
print "@neu_text\n";
```

Der **Backslash-Escape** `\u` macht den nächsten Buchstaben zum Großbuchstaben.

Es gibt wiederum eine **Kurzform** ohne geschweifte Klammern, wie die zweite Version zeigt.

Der Ausdruck wird **im Listen-Kontext** evaluiert, so daß ein Original-Listenelement ggf. **mehrere Ergebnis-Listenelemente** ergeben darf.

Hash-Slices: Genau wie man ein Slice von einem **Array** machen kann, kann man auch ein Slice von einem Hash machen. Das **Ergebnis** ist ebenfalls eine **Liste**. Gekennzeichnet wird es durch **geschweifte Klammern** und einen **Klammeraffen** vor dem Hash-Namen. (Ein Array-Slice hat eckige Klammern und ein Dollar-Zeichen.)

Konstanten: Es gibt auch Konstanten:

constant.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use strict;
use constant DEBUGGING => 0;
use constant ONE_YEAR  => 365.2425 * 24 * 60 * 60;

if(DEBUGGING) {
    # ...
}
# ...
```

Vorteile: **Sicherer**. Und es kann besser **optimiert** werden. Im Beispiel wird der **Debugging-Code** ggf. **vollständig wegoptimiert**.

Sicherheit: Man kann ein `perl`-Programm ziemlich sicher machen, z.B. durch **Datenflußanalyse** mit dem **taint-Mechanismus**. Dabei führt `perl` Buch, welche Daten aus der Umgebung kommen, so daß man ihnen nicht trauen darf. Wenn solche Daten einen anderen Prozeß, eine andere Datei oder ein anderes Directory beeinflussen, bricht `perl` die Verarbeitung ab. Siehe man `perlsec` .

Debugging: `perl` hat einen guten **eingebauten Debugger**. Er ist kommandozeilenorientiert und selbst in `perl` geschrieben. Siehe man `perldebug`.

Common Gateway Interface (CGI): Das CGI-Modul erlaubt, **HTML-Formulare** zu generieren und auszuwerten.

Die Verwendung setzt allerdings sehr gute `perl`- und Sicherheitskenntnisse voraus, da **Hacker-Angriffe** auf CGI-Skripten **sehr beliebt** sind.

Referenzen: Ähnlich wie **C-Pointer**, aber **sicherer**.

Siehe man `perlreftut` und man `perlref` .

Komplexe Datenstrukturen: Sind möglich, z.B. zweidimensionale Felder, oder ein Array von Hashes, oder ein Hash von Hashes, ...

Siehe man `perldsc` und man `perllo1` .

Überladen von Operatoren: Mit dem `overload`-Modul. Z.B. für die Operatoren auf den komplexen Zahlen.

Einbetten von Dokumentation in perl-Quellcode: `perls` **eigene Dokumentation** ist in den Quellcode eingebettet, im **pod-Format** (plain old documentation). So kann man auch **eigene Module dokumentieren**, und `perldoc` wird sie gleich mit anzeigen. Siehe man `perlpod` .

Graphische Benutzerschnittstellen (GUIs): Die Tk-Module bieten dies.

4 Automatisieren der Compilierung mit make

4.1 Überblick über make

make:

- erlaubt die **Automatisierung** der vielen Schritte, die zur **Generierung eines Programms** aus vielen Quelldateien notwendig sind
- **Verwaltung der Abhängigkeiten** dieser Schritte, was bei **Programmänderungen** wichtig wird

Beispiele:

- **Kompilierung eines C-Programms** aus vielen Quell-Dateien.
- **Kompilierung der L^AT_EX-Quellen** dieser Skriptnotizen:
Folien werden **separat** kompiliert, dann in einzelne Postscript-Seiten zerlegt, diese dann in die Skriptnotizen eingebunden und zusammen kompiliert. Weiterhin werden die vielen **Beispielprogramme** aus ihren Quelldateien eingebunden.

Wenn sich etwas ändert, wird nur das Notwendige neu kompiliert, was viel Zeit spart. Dazu schaut sich make das **Veränderungsdatum** der Dateien an. Die **Korrektheit** der Abhängigkeitsprüfung wird garantiert (wenn die Regeln einmal richtig aufgestellt wurden).

Die **Abhängigkeiten** werden in einem „**Makefile**“ aufgeschrieben. **Viel Wissen** um Standard-Abhängigkeiten ist in make sogar **bereits eingebaut**. Dies funktioniert über **Datei-Endungen**.

4.2 Grundlagen von Makefiles

4.2.1 Einfaches Beispiel: Editor übersetzen

Wir nehmen an, daß wir ein **ausführbares Programm namens edit** erzeugen wollen, für das wir **acht C-Quellen** und **drei Header-Files** haben:

editor-simple/Makefile:

```
# Makefile for "edit".

edit : main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
      cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

main.o : main.c defs.h
      cc -c main.c
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c
display.o : display.c defs.h buffer.h
      cc -c display.c
insert.o : insert.c defs.h buffer.h
      cc -c insert.c
search.o : search.c defs.h buffer.h
      cc -c search.c
files.o : files.c defs.h buffer.h command.h
      cc -c files.c
utils.o : utils.c defs.h
      cc -c utils.c
clean :
      rm edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Kommentare werden mit dem Musikkreuz „#“ eingeleitet.

Das **ausführbare Programm** ist `edit`, es wird aus acht Objektdateien zusammengebaut.

Eine **Regel** besteht aus einem **Ziel**, einer **Liste von Vorbedingungen** und einer **Liste von Anweisungen**.

Ein **Ziel** ist eine Datei, die **erzeugt** werden muß. Es kann auch etwas sein, was **nur getan** werden muß, ohne daß eine Datei entsteht.

Eine **Vorbedingung** ist eine Datei, die für die Erzeugung eines Ziels notwendig ist. Ein Ziel hat häufig viele Vorbedingungen.

Eine **Anweisung** ist etwas, was `make` ausführt. Eine Regel kann mehr als eine Anweisung haben, jede auf einer Zeile.

ACHTUNG: Vor jeder Anweisungs-Zeile muß ein **Tabulator-Zeichen** stehen! Acht Blanks stattdessen sind nicht erlaubt! Das ist eine **häufige Falle**. Z.B. der Editor `vim` hilft, indem er solche Syntaxfehler farblich hervorhebt. ►► Ausprobieren.

Eine **Regel erklärt**, wie eine Ziel-Datei neu aus ihren Vorbedingungs-Dateien erzeugt wer-

den kann. Eine Regel kann auch nur beschreiben, wie eine bestimmte Aktion auszuführen ist.

Ein Makefile kann noch mehr als Regeln enthalten, aber dies ist das **Grundsche-ma**.

Im Beispiel wird jede der acht **Objektdaten** aus der zugehörigen **C-Quelle** erzeugt.

Außerdem **hängen** die Objektdaten noch **von** den **Header-Dateien** ab, die in der C-Quelle mit eingeschlossen wurden.

```
▶▶ kbd.c
```

```
▶▶ defs.h
```

`defs.h` wird von allen C-Quellen eingeschlossen. `command.h` wird nur von den Editier-Kommandos eingeschlossen, und `buffer.h` wird nur von den Low-Level-Files eingeschlos-sen, die den Editor-Puffer manipulieren.

Lange Zeilen können mit **Backslash-Newline** aufgeteilt werden.

Oft wollen wir auch **alle generierten Dateien** wieder **löschen**. Hierfür können wir eine **Aktion** definieren: `clean`

Eine Aktion **erzeugt keine Datei**. Oft hat sie auch keine Vorbedingung.

Eine **Anweisung** muß mit einem **Tab** beginnen und enthält ein **beliebiges Shell-Kom-mando**. Da `make` nichts über die Semantik dieser Kommandos weiß, sollte man die Vor-bedingungen und Ziele tunlichst **richtig aufschreiben**. Es gibt dafür aber Hilfen, dazu später mehr.

4.2.2 Grundalgorithmus von make

Um die **Übersetzung** zu **starten**, tippt man

```
▶▶ make
```

`make` fängt mit dem **ersten Ziel** an, dem **Default-Ziel**. Hier ist das `edit`, das wir deswegen nach vorne gestellt haben.

`make` schaut sich an, **wie** das **Ziel erzeugt** wird, und es schaut sich **an**, wovon es **abhängt**.

`make` kann `edit` nicht sofort erzeugen, da **auch** dessen **Vorbedingungen**, die Objektda-teien, **Regeln** haben. Es geht **rekursiv** durch die Abhängigkeiten, bis es Dateien findet, die keine Regeln haben. Aus den Abhängigkeiten stellt es eine Reihenfolge von Anweisun-gen zusammen, die alle notwendigen Schritte in der **richtigen Reihenfolge** enthält.

Falls sich eine **Vorbedingungs-Datei ändert**

```
▶▶ touch search.c
```

```
▶▶ make
```

dann werden nur die notwendigen Dateien neu erzeugt. Auch bei **komplizierteren Ab-hängigkeiten**:

```
▶▶ touch buffer.h
```

```
▶▶ make
```

Wenn **nichts zu tun** ist, sagt **make** uns das auch:

```
▶▶ make
```

Falls eine Regel **nicht** zur Erzeugung des gewünschten Ziels **benötigt** wird, wird sie **nicht benutzt**.

Man kann auf der Kommandozeile auch ein **Ziel angeben**:

```
▶▶ make clean
```

Dann wird das **Default-Ziel nicht** benutzt.

4.2.3 Grundlagen von Variablen

Im Beispiel haben wir die Objektdaten in der Regel von **edit** **zweimal aufgeführt**. Das ist **fehlerträchtig**. Mit Variablen kann man das vermeiden:

editor-simple/Makefile-var:

```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Es ist üblich, in jedem Makefile eine Variable **namens** `objects`, `OBJECTS`, `objs`, `OBJS`, `obj` oder `OBJ` zu haben.

Um Variablen zu **benutzen**, schreibt man ein **Dollar-Zeichen** und **runde Klammern**.

Man beachte, daß auch die **Regel für `clean`** viel **robuster** gegenüber Änderungen wird.

Einschub: Der Name eines Makefiles ist normalerweise `Makefile` oder `makefile`. Man kann aber auch einen anderen Namen auf der Kommandozeile angeben:

```
▶▶ make -f Makefile-var
```

```
▶▶ make -f Makefile-var clean
```

4.2.4 Grundlagen impliziter Regeln

Die **Regeln** für die Erzeugung von Objekt-Dateien aus C-Quellen sind eigentlich **immer analog**.

`make` enthält daher bereits viele eingebaute **implizite Regeln**.

`make` orientiert sich dabei an den **Dateiendungen**. Um z.B. von einer `.c`-Datei zu einer `.o`-Datei zu kommen, braucht man immer ein „`cc -c`“-Kommando.

Wenn eine `.o`-Datei erzeugt werden soll, und wenn es eine zugehörige `.c`-Datei **bereits gibt**, dann wird die implizite **Regel automatisch angewendet**, und die `.c`-Datei wird automatisch in deren **Liste der Vorbedingungen aufgenommen**.

So **könnte** unser Makefile in der **Praxis aussehen**:

```
editor-simple/Makefile-impl:
```

```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Implizite Regeln werden **häufig angewendet**.

►► `make -f Makefile-impl`

►► `make -f Makefile-impl clean`

Auf das obige „.PHONY“ für `clean` kommen wir noch zurück, ebenso auf das **Minus** vor `rm`. Das erstere verhindert, daß aus Versehen eine Datei namens `clean` erzeugt wird, das letztere bewirkt, daß Fehler wegen nicht-existierender Dateien beim Löschen ignoriert werden.

4.3 Aufbau eines Makefiles

4.3.1 Struktur eines Makefiles

Ein Makefile kann enthalten:

- explizite Regel
- implizite Regel (selbstdefiniert)
- Variablendefinition
- Direktive
- Kommentarzeile mit `#`

Eine **Direktive** macht etwas **besonderes**, wenn das **Makefile** **eingelesen** wird:

- anderes Makefile **einlesen**
- **bedingtes** Benutzen/Ignorieren von Teilen des Makefiles
- definieren einer **zusammengefaßten Anweisungssequenz**

Ein **Kommentar** geht vom **#**-Zeichen bis zum **Zeilenende**. **Ausnahme**: ein **Backslash** am Ende ohne einen zweiten Backslash davor **setzt** die Kommentarzeile **fort**.

Kommentare dürfen **fast überall** stehen. Eine Zeile nur mit einem Kommentar erscheint als **leer** und wird daher **ignoriert**.

4.3.2 Name eines Makefiles

Als **Default** wird **makefile** oder **Makefile** genommen, in dieser **Reihenfolge**. **Empfohlen** wird das **zweite**, weil es sich im Verzeichnis besser heraushebt.

Gnu-make sucht **außerdem** vorher noch nach **GNUmakefile**. Das sollte man aber nur benutzen, wenn man verschiedene Makefiles für verschiedene **makes** haben will.

Wenn man **mehrere Makefiles** im selben Verzeichnis haben will, muß man mit der Kommandozeilenoption **-f** arbeiten. Dann benutzt man als **Dateierweiterung** oft **.mk** oder auch **.m** oder **.mak**.

4.3.3 Aufbau der Regeln

```
ziele : vorbedingungen
      anweisung
      ...

oder auch:

ziele : vorbedingungen ; anweisung
      anweisung
      ...
```

Anmerkung: Die **zweite Form** habe ich noch **nie** in der Praxis **gesehen**.

Die **Reihenfolge** der Regeln ist **unwichtig**, **außer** der **ersten** Regel. Die erste Regel ist die **Default-Regel**.

Ausnahmen: Ziele, die **mit** einem **Punkt** **beginnen**, zählen nicht. Und **Regel-Muster** (s.u.) auch nicht.

Deshalb ist die erste Regel meißt die, die das **gesamte Programm zusammensetzt**. Dieses Ziel wird **oft all genannt**.

Die **ziele** sind Dateinamen, von Leerzeichen getrennt. **Meist** hat man **nur ein** Ziel pro Regel.

Eine **anweisung** beginnt immer mit einem **Tab**. (Man kann es nicht oft genug sagen!)

Man kann **Zeilen** mit **Backslashes umbrechen**. Es gibt allerdings **kein Limit** für die Zeilenlänge.

Ein einzelnes **Dollar-Zeichen** erreicht man durch zwei davon, wg. der Expansion von Variablennamen.

Eine Regel sagt **make** zwei Dinge:

- Wann ist das Ziel veraltet
- Wie aktualisiert man das Ziel bei Bedarf

Die **Bedingungen** für „veraltet“ werden von den Vorbedingungen angegeben.

Die **Vorbedingungen** sind ebenfalls **Dateinamen**, von Leerzeichen getrennt. Ein Ziel ist **veraltet**, wenn es **nicht existiert**, oder wenn es **älter als irgendeine** seiner **Vorbedingungs-Dateien** ist.

Die **Anweisungen** werden mit einer **Unix-Shell**, normalerweise **sh**, ausgeführt.

4.3.4 Unechte Ziele

Manchmal will man eine Aktion ausführen, die gar keine Datei erzeugt, z.B. das Aufräumen mittels **make clean**, siehe oben.

Was passiert, wenn man `make clean` mehrfach aufruft? Es wird **jedesmal wieder** ausgeführt, weil das Ziel nicht existiert. (Und zwar jeweils **einmal**.)

Was passiert, wenn jemand eine Datei namens `clean` aus Versehen erzeugt? Die Regel wird **nie mehr ausgeführt**!

Lösung bei Gnu-make: Dieses Ziel als „unecht“ markieren:
`editor-simple/Makefile-impl:`


```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

.PHONY ist ein **besonderes, fest eingebautes Ziel**. Seine Vorbedingungen werden **nicht als Dateien interpretiert**, sondern als unechte Ziele.

Die Regeln für unechte Ziele werden, wenn verlangt, **immer ausgeführt, egal ob eine Datei** dieses Namens **existiert** oder nicht.

Weiterhin wird für unechte Ziele **nicht nach impliziten Regeln gesucht**, was die **Geschwindigkeit** von make etwas **steigert**.

Eine **andere Einsatzmöglichkeit** für unechte Ziele ist die **rekursive** Verwendung von make in **Unterverzeichnissen**. Man könnte schreiben:

Makefile-subdirs:

```
subdirs = foo bar baz

subdirs:
        for dir in $(subdirs); do \
            $(MAKE) -C $$dir; \
        done
```

Hier rufen wir make **rekursiv** für jedes Unterverzeichnis auf. **Näheres** dazu kommt **später**.

Anmerkung: Wir müssen die **Shell-Kommandos mit Backslashes verbinden**, damit sie für make **eine einzige Anweisung** werden. Sonst würden sie in drei getrennte Shells gesteckt und ergäben Syntaxfehler.

Nachteile dieser Methode? Fehler in Unter-Makefiles werden im Haupt-Makefile nicht erkannt. Mögliche **Nebenläufigkeit** kann nicht ausgenutzt werden, z.B. auf **Mehrprozessormaschinen**.

Lösung: Die Unterverzeichnisse als unechte Ziele deklarieren:

Makefile-subdirs2:

```
subdirs = foo bar baz

.PHONY: subdirs $(subdirs)

subdirs: $(subdirs)

$(subdirs):
    $(MAKE) -C $@

foo: baz
```

Hier haben wir übrigens **explizit deklariert**, daß das **Unterverzeichnis foo** erst nach dem Unterverzeichnis **baz** bearbeitet werden darf. Das ist besonders für **parallele Übersetzung** wichtig.

Der **Ausdruck** `$@` bezeichnet das aktuelle Ziel. Diese **automatischen Variablen** erläutern wir **später**.

Unechte Ziele können Vorbedingungen haben. Das ist praktisch, wenn man in einem Makefile mehrere ausführbare Programme beschreiben will. Dann definiert man ein unechtes Ziel namens `all`, das von den echten Zielen abhängt:

Makefile-all:

```
all : prog1 prog2 prog3
.PHONY : all

prog1: prog1.o utils.o
    cc -o prog1 prog1.o utils.o
prog2: prog2.o
    cc -o prog2 prog2.o
prog3: prog3.o sort.o utils.o
    cc -o prog3 prog3.o sort.o utils.o
```

Jetzt werden **mit einem Aufruf make** alle drei Programme gebaut.

Man kann **aber auch** nur `make prog1 prog3` sagen.

Wenn ein **unechtes Ziel** eine **Vorbedingung eines anderen** unechten Ziels ist, dient es als **Unterprogramm**:

Makefile-cleanall:

```

.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm programm

cleanobj :
        rm *.o

cleandiff :
        rm *.diff

```

Leider gibt es „**unechte Ziele**“ **nur bei Gnu-make**, nicht bei allen makes. Will man Makefiles auch für diese makes schreiben helfen **Regeln ohne Vorbedingungen und ohne Anweisungen**:

Makefile-cleanforce:

```

clean: FORCE
        rm $(objects)

FORCE:

```

Wenn es das Ziel dieser Regel, hier **FORCE**, **nicht als Datei gibt**, dann wird angenommen, daß **alle Ziele**, die von diesem Ziel **abhängen**, **jedesmal neu bearbeitet** werden müssen.

Der **Name** eines solchen Ziels **ist egal**.

Die **Wirkung** ist die **gleiche** wie eben, aber es ist **nicht so gut lesbar**.

Problem: Wenn eine **Datei mit diesem Namen erzeugt** wird, geht die Regel schief.

4.3.5 Mehrere Regeln für ein Ziel

Eine **Datei** kann **Ziel von mehreren Regeln** sein.

Die **Vorbedingungen** aller Regeln werden dann **vereinigt**.

Nur **höchstens eine** Regel darf **Anweisungen** haben.

Es ist oft **praktisch**, ein paar **zusätzliche Abhängigkeiten** mit **zusätzlichen Regeln** zu schreiben, die dann keine Anweisungen haben:

Makefile-multirule:

```

objects = foo.o bar.o

foo.o : defs.h
bar.o : defs.h test.h

$(objects) : config.h

```

Hier wird die **zusätzliche Abhängigkeit** aller Objekt-Dateien von **config.h** in einer

separaten Zeile ausgedrückt. Man könnte das dazutun oder löschen, ohne die echten Regeln anzufassen. Durch die Variable `$(objects)` geht das so flexibel.

Falls **gar keine Regel** angegeben ist, wird eine **implizite** Regel gesucht.

4.3.6 Mehrere Ziele in einer Regel

Man kann auch **mehrere Ziele** in einer Regel haben. Das macht in zwei Fällen Sinn:

- man gibt **nur Vorbedingungen** an und keine Anweisungen
- für alle Ziele gibt es **ähnliche Anweisungen**

Das **erstere** setzt voraus, daß man **auch mehrere Regeln** für die Ziele hat.

Das **letztere** benötigt normalerweise **automatische Variablen**, um den aktuellen Dateinamen in die Regel einzusetzen. Beispiel:

Makefile-multitarget:

```
bigoutput littleoutput : text.g
    generate text.g -$(subst output,, $@) > $@
```

ist äquivalent zu:

Makefile-multitarget2:

```
bigoutput : text.g
    generate text.g -big > bigoutput
littleoutput : text.g
    generate text.g -little > littleoutput
```

Die automatische Variable `$@` und die verwendete Textersetzungsfunktion `subst` wird später erläutert.

4.3.7 Statische-Muster-Regeln

Eine **flexiblere Variante** von mehreren Zielen in einer Regel sind Statische-Muster-Regeln. Hier müssen die **Vorbedingungen nicht** mehr **identisch** sein, **sondern** nur noch **analog**.

Die Syntax ist:

```
ziele : ziel-muster : vorbedingungen
    anweisung
    ...
```

Das Neue ist hier das **Ziel-Muster**. Es enthält ein **Prozent-Zeichen %** als **Wildcard**. Mithilfe dieses Wildcards **muß** das Ziel-Muster **auf jedes** der **Ziele** **passen**.

Der **Vorteil** ist, daß man das **Prozent-Zeichen** **auch in** den **Vorbedingungen** verwenden darf, damit kann man die **Liste der Vorbedingungen** an das jeweilige **Ziel**

anpassen. Beispiel:

Makefile-statpat:

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c defs.h
        cc -c -o $@ $<
```

Hier hängt foo.o ab von **foo.c** und **defs.h**, und es hängt bar.o ab von bar.c und defs.h.

In das Muster **%.c** wird also jeweils der „**Stamm**“ des Ziels **eingesetzt**. Der Stamm ist der **Teil des Ziels**, der auf das **Prozent-Zeichen** **matcht**.

Es ist auch erlaubt, daß das **Prozent-Zeichen** gar **nicht** in einer Vorbedingung **vorkommt**. Was passiert dann? Dann ist diese Vorbedingung **für alle Ziele gleich**.

Wir sehen wieder die Verwendung von **automatischen Variablen**, um auch die Anweisung an das Ziel und die Vorbedingungen anzupassen. **\$@** wird durch das **aktuelle Ziel** ersetzt, **\$<** durch die **aktuelle erste Vorbedingung**. Später dazu mehr.

Ein weiteres Beispiel:

Makefile-statpat2:

```
bigoutput littleoutput : %output : text.g
        generate text.g -*$ > $@
```

Hier sind **zwar** die **Vorbedingungen konstant**, **aber** wir haben eine **elegante** Möglichkeit, die **Art der Generierung** zu steuern. Die automatische Variable ***\$** enthält den **aktuellen Stamm** des Ziels.

Was ist, wenn ein **Prozent-Zeichen** im **Dateinamen** vorkommt? Mit **Backslash** **quoten**. Was ist, wenn ein **Backslash** im **Dateinamen** vorkommt? **MSDOS**-Dateinamen!

Daher etwas **ungewöhnliche Quote-Regeln**:

- ein **Backslash** vor einem **Prozent-Zeichen** quotet dieses
- **zwei Backslashes** vor einem Prozent-Zeichen ergeben einen Backslash und ein normales Wildcard
- ein **Backslash** an anderer Stelle bleibt einfach erhalten

Beispiel:

```
C:\Prog\\100%\In\\%.c
ergibt
C:\Prog\\100%\In\Stamm.c
```

Anmerkung: MSDOS selbst erlaubt kein „%“, wohl aber Windows (und Unix)!

Eine Statische-Muster-Regel hat **Ähnlichkeit** mit einer **impliziten Regel**. **Beide** haben ein **Muster** für das Ziel und für die Vorbedingungen. (Zu impliziten Regeln später mehr.)

Der **Unterschied** ist, *wann* die Regel **angewandt** wird. Eine **implizite** Regel wird angewandt, **wenn** es **keine explizite** Regel gibt. Eine **Statische-Muster-Regel** wird explizit **genau auf** die **genannten Ziele** angewandt.

Gibt es **zwei Statische-Muster-Regeln** für ein Ziel, ist das ein **Fehler**. Passen **mehrere implizite Regeln**, entscheidet die **Reihenfolge** der Definition, welche davon angewandt wird.

Wenn man **nicht** so **genau weiß, welche Dateien** es im Verzeichnis gibt, ist eine explizite **Statische-Muster-Regel** **sicherer**. Z.B. kann eine **.o-Datei** sowohl aus einer **.c-Datei** als auch aus einer **.cpp-Datei** erzeugt werden.

Und man kann mit einer Statischen-Muster-Regel **gut Sonderfälle behandeln**, für die die impliziten Regeln nicht passen.

4.4 Anweisungen in Regeln

Eine Anweisung muß **mit einem Tab beginnen**.

Leere Zeilen und Zeilen nur mit **Kommentaren** dürfen **zwischen Anweisungszeilen** stehen, sie werden **ignoriert**.

Zeilen mit nur einem Tab sind *keine* **leeren Zeilen**, sondern **leere Anweisungen**. Beachte, daß von allen Regeln für ein Ziel nur eine davon Anweisungen haben darf!

4.4.1 Anweisungs-Ausführung

Anweisungen werden **immer von /bin/sh ausgeführt**, selbst wenn die Login-Shell oder die Environment-Variable **\$SHELL** auf etwas anderes gesetzt ist.

Jede Anweisung wird in einer eigenen Shell ausgeführt:

Makefile-shell:

```
# vermeide Probleme mit symb. Links im Pfad:
cd=cd -P

test1:
    pwd
# FALSCH:
    $(cd) editor-simple
    pwd

test2:
    pwd
# RICHTIG:
    $(cd) editor-simple ; pwd
```

```
▶▶ make -f Makefile-shell test1
```

```
▶▶ make -f Makefile-shell test2
```

4.4.2 Anweisungs-Echo

Normalerweise **druckt** make **jede Anweisung** vor ihrer Ausführung.

Das kann man mit @ vor einer Anweisung verhindern:

Makefile-print:

```
all:
    # .....
    @echo "**** Kompilation erfolgreich beendet ****"
```

```
▶▶ make -f Makefile-print
```

Wenn die **Anweisung** ohnehin **nur druckt**, will man sie nicht sehen.

Anmerkung: Die **Kommentarzeile** *wurde gedruckt*, weil sie mit **Tab** beginnt und deshalb eine (**leere**) **Anweisung** ist. Die Shell ignoriert alles ab dem Kommentarzeichen, aber **Drucken** und **Shell-Aufruf** finden statt.

Wenn man make mit dem **Parameter** -n aufruft, dann werden die Kommandos **nur gedruckt**, nicht ausgeführt. In diesem Falle werden **auch @-Zeilen** gedruckt.

```
▶▶ make -f Makefile-print -n
```

4.4.3 Parallele Ausführung

Normalerweise führt make **eine Anweisung zur Zeit** aus. Mit der **Option** -j nutzt es **Nebenläufigkeit** in den Abhängigkeiten aus, um **mehrere Anweisungen gleichzeitig** auszuführen. Gut auf **Mehrprozessormaschinen**.

Makefile-par:

```
targets = prog1 prog2 prog3

all: $(targets)

$(targets) : prog% :
    @echo "Starting to generate $@..."
    @sleep $*
    @echo "...continueing to generate $@"
    @sleep 1
    @echo "Completed $@."
```

```
▶▶ make -f Makefile-par
```

```
▶▶ make -f Makefile-par -j
```

Gibt man hinter `-j` eine **Zahl** an, so ist das die **max. Anzahl** der **Jobs**.

```
▶▶ make -f Makefile-par -j 2
```

Ein **Problem** bei paralleler Bearbeitung: **Nur ein Job** kann **Stdin** haben. Die **anderen Jobs** bekommen eine **ungültige Eingabe**. Potentiell **interaktive Kommandos** kann man also **nicht parallelisieren**. **Welcher Job** das eine **Stdin** bekommt, ist ziemlich **zufällig**.

Mit der **Option** `-l` kann man die **max. Systemlast** angeben, bei der noch parallelisiert werden soll:

```
▶▶ make -f Makefile-par -j -l 2.5
```

```
▶▶ make -f Makefile-par -j -l 0.1
```

4.5 Fehlerbehandlung

4.5.1 Returncodes

Nach Ende **jeder Anweisung** prüft **make** den **Return-Code**. Falls ein **Fehler** aufgetreten ist, **bricht** **make** die **aktuelle Regel** ab, und normalerweise **auch** die **gesamte Bearbeitung**.

Makefile-error:

```
targets = prog1 prog2 prog3

all: $(targets)

$(targets) : prog% :
    @echo "Starting to generate $@..."
    @sleep $*
    @echo "...hitting some error for $@" ; exit 42
    @echo "Completed $@."
```

```
▶▶ make -f Makefile-error
```

Wenn man seine **Kompilation testen** will, dann kann man **make** sagen, daß es **so weit wie möglich weitermachen** soll, mit Hilfe der **Option** `-k` bzw. `--keep-going`:

```
▶▶ make -f Makefile-error -k
```

4.5.2 Ignorieren von Fehlern

Manchmal ist ein **Scheitern nicht schlimm**, oder Programme geben **falsche Return-Codes** zurück. Dann kann man **make** anweisen, den Return-Code zu **ignorieren**, indem man ein **Minus** `-` der Anweisung **voranstellt**:

Makefile-error2:


```

prog = foo

.PHONY: all clean

all:

clean:
    -rm $(prog)
    @echo "Wir machen trotzdem weiter..."

```

▶▶ make -f Makefile-error2 clean

4.5.3 Inkonsistente Zieldateien bei Fehlern oder Unterbrechungen

Wenn eine **Anweisung** mit einem Fehler **abbricht**, dann kann die **Zieldatei** in einem **inkonsistenten Zustand** sein:

Makefile-error3:

```

target = generated_web_page.html

all: $(target)

$(target):
    @echo "Starting to generate $@..."
    @echo "<html><body>" > $@
    @echo "<p>bla bla" >> $@
    @sleep 2
    @echo "...hitting some error for $@" ; exit 42
    @echo "</body></html>" >> $@
    @echo "Completed $@."

clean:
    -rm $(target)

```

▶▶ make -f Makefile-error3

▶▶ cat generated_web_page.html

Ein **neuer Aufruf** von make **korrigiert** den Fehler *nicht*, weil die **Datei** nun **existiert**:

▶▶ make -f Makefile-error3

Gibt man das **spezielle Ziel** `.DELETE_ON_ERROR` an, dann werden solche Dateien **gelöscht**:

Makefile-error4:

```

target = generated_web_page.html

.DELETE_ON_ERROR:

all: $(target)

$(target):
    @echo "Starting to generate $@..."
    @echo "<html><body>" > $@
    @echo "<p>bla bla" >> $@
    @sleep 2
    @echo "...hitting some error for $@" ; exit 42
    @echo "</body></html>" >> $@
    @echo "Completed $@."

clean:
    -rm $(target)

```

▶▶ make -f Makefile-error3 clean

▶▶ make -f Makefile-error4

Aus **historischen Gründen** ist das **nicht** das **Default**.

Wenn man make **abbricht**, z.B. durch **Ctrl-C**, dann kann es ebenfalls eine **inkonsistente Zielfeile** geben.

Hier ist das **Löschen** der aktuellen Zielfeile bereits **Default**.

▶▶ make -f Makefile-error3 und Drücken von Ctrl-C

In beiden Fällen wird **nur gelöscht**, falls die Datei **seit Beginn der Bearbeitung** der Regel bereits **modifiziert** wurde. **Anderenfalls** wird die konsistente, aber veraltete Datei **einfach belassen**.

Manchmal ist eine inkonsistente Datei immer noch **besser als gar keine**. Dann verhindert das **spezielle Ziel** .PRECIOUS, daß alle Dateien, von denen es abhängt, gelöscht werden:
Makefile-error5:

```

target = generated_web_page.html

.PRECIOUS: $(target)

all: $(target)

$(target):
    @echo "Starting to generate $@..."
    @echo "<html><body>" > $@
    @echo "<p>bla bla" >> $@
    @sleep 2
    @echo "...hitting some error for $@" ; exit 42
    @echo "</body></html>" >> $@
    @echo "Completed $@."

clean:
    -rm $(target)

```

```

▶▶ make -f Makefile-error5
▶▶ cat generated_web_page.html
▶▶ make -f Makefile-error5 clean

```

4.6 Rekursive Verwendung von make

Bei größeren Projekten hat man **oft mehrere Unterverzeichnisse**, in denen jeweils übersetzt werden soll. Dann soll das **make** im oberen Verzeichnis rekursiv **makes** in den Unterverzeichnissen aufrufen. Beispiel:

```

▶▶ cd subdirs
▶▶ ls

```

Makefile:

```

subdirs = doc src test
clean_subdirs = $(patsubst %, clean_%, $(subdirs))

.PHONY: all clean $(subdirs) $(clean_subdirs)

all: $(subdirs)

$(subdirs):
    cd $@ && $(MAKE)

test: src

clean: $(clean_subdirs)

$(clean_subdirs): clean_% :
    cd $* && $(MAKE) clean

```

Hier haben wir **drei Unterverzeichnisse**. Für das **Default-Ziel all** sollen **alle drei rekursiv** mit make bearbeitet werden. (Dabei soll **test erst nach src** bearbeitet werden.)

Wir machen das, indem wir per **cd in** das **Unterverzeichnis** wechseln und dort nochmal make **aufrufen**.

Die **Variable \$(MAKE)** enthält dabei den Namen des laufenden make-Programms. Das ist **besser als direkt make** zu sagen, weil man so auch ein anderes make-Programm nehmen kann.

Man **beachte** die **&&** nach dem **cd**. Das zweite Shell-Kommando wird **nur ausgeführt**, wenn das erste **keinen Fehler** hatte. (Shell-Feature)

Wieder die **automatische Variable \$@** benutzt.

Wir haben hier **drei getrennte Regeln** für die Unterverzeichnisse, um ggf. **Parallelität** ausnutzen zu können.

Für das **Aufräumen tricksen** wir hier etwas: Mittels der Funktion **patsubst** erzeugen wir drei Ziele **clean_doc**, **clean_src** und **clean_test**, für die wir dann eine **Statische-Muster-Regel** definieren. In dieser verwenden wir dann die **automatische Variable \$***, um auf den **Stamm** und damit auf den eigentlichen Verzeichnisnamen zugreifen zu können. Danach geht es wie gehabt **rekursiv weiter**.

In den **Unterverzeichnissen** wird dann entweder die **eigentliche Arbeit** getan:
doc/Makefile:

```

doc = doc.txt

.PHONY: all clean

all: $(doc)

$(doc):
    cat /dev/null > $@

clean:
    -rm $(doc)

```

test/Makefile:

```

.PHONY: all clean

all:
    @echo "Performing tests..."

clean:
    @echo "Cleaning up tests..."

```

Oder es geht **noch eine Rekursionsstufe** weiter:

src/Makefile:

```

subdirs = prog1 prog2
clean_subdirs = $(patsubst %, clean_%, $(subdirs))

.PHONY: all clean $(subdirs) $(clean_subdirs)

all: $(subdirs)

$(subdirs):
    cd $@ && $(MAKE)

clean: $(clean_subdirs)

$(clean_subdirs): clean_% :
    cd $* && $(MAKE) clean

```

src/prog1/Makefile:

```

prog = prog1

.PHONY: all clean

all: $(prog)

$(prog): defs1.h
    cat /dev/null > $@

clean:
    -rm $(prog)

```

src/prog2/Makefile:

```

prog = prog2

.PHONY: all clean

all: $(prog)

$(prog): defs2.h
    cat /dev/null > $@

clean:
    -rm $(prog)

```

▶▶ make

▶▶ make

▶▶ make clean

4.6.1 Details der Variablen MAKE

Was passiert mit der **Rekursion**, wenn man `make -n` aufruft, um einen **Probelauf** durchzuführen?

▶▶ `make -n`

Die **Rekursion** wird **trotzdem** ausgeführt! Warum?

Zeilen, die `$(MAKE)` oder `${MAKE}` **enthalten**, werden trotz `-n` ausgeführt.

Das gleiche gilt für die **Option** `-t` bzw. `--touch`. Bei dieser Option werden die Anweisungen **ebenfalls nicht ausgeführt**, aber die **Daten der Ziele** ggf. trotzdem per `touch`-Befehl **aktualisiert**:

▶▶ make

▶▶ `touch src/prog1/defs1.h`

►► make -t

Was ist, wenn man **für** die **Rekursion** unbedingt **Anweisungen** ausführen lassen muß, diese aber **nicht** \$(MAKE) oder \${MAKE} **enthalten**? **Zeilen**, die **mit + beginnen**, werden genauso behandelt:

Makefile2:

```
my_very_special_make = make

subdirs = doc src test
clean_subdirs = $(patsubst %, clean_%, $(subdirs))

.PHONY: all clean $(subdirs) $(clean_subdirs)

all: $(subdirs)

$(subdirs):
    echo "Nun geht's ab ins Unterverzeichnis..."
    +cd $@ && $(my_very_special_make)

test: src

clean: $(clean_subdirs)

$(clean_subdirs): clean_% :
    cd $* && $(MAKE) clean
```

►► make -f Makefile2 -n

Man **beachte**, daß die **echo-Anweisung** vor der +-Zeile **nicht ausgeführt** wird, weil sie kein Plus trägt.

4.6.2 Optionen und Variablen an Sub-makes übergeben

Woher weiß das **Sub-make**, daß das oberste **make** mit der **Option -n** oder **-t** aufgerufen wurde? Die **make-Variable MAKEFLAGS** wird ins **Environment** exportiert und vom **Sub-make** gelesen:

►► cd ..

Makefile-flags1:

```
.PHONY: all

all:
    +echo "Die Make-Flags auf Ebene $(MAKELEVEL) sind >$(MAKEFLAGS)<"
    $(MAKE) -f Makefile-flags2
```

Über die **Environment-Variable** MAKELEVEL wird den Sub-makes mitgeteilt, die **wievielte Rekursionsebene** gerade aktuell ist. Auf der **obersten Ebene** ist sie **Null**. Das ist **nützlich**, um zu testen, **ob** das Makefile **rekursiv** oder **direkt** aufgerufen wurde.

Makefile-flags2:

```
CC=gcc -c
LD=gcc

.PHONY: all

all:
    +echo "Die Make-Flags auf Ebene $(MAKELEVEL) sind >$(MAKEFLAGS)<"
    +echo "Weitere Variablen sind CC: >$(CC)< und LD: >$(LD)<"
```

▶▶ make -f Makefile-flags1

Die **Option** -w bzw. --print-directory sagt, daß make am Anfang und Ende ausgeben soll, **in welchem Directory** es arbeitet. Sie wird **bei der Rekursion automatisch** dazugefügt.

▶▶ make -f Makefile-flags1 -n

Mehrere Optionen werden **in Kurzform** überreicht:

▶▶ make -f Makefile-flags1 --keep-going --silent

Die **Option** -s bzw. --silent **unterdrückt die Ausgabe der** aktuell ausgeführten **Anweisung**.

Man kann auch **weitere Werte über Environment-Variablen** per **export** weiterreichen. Aber im allg. **besser** ist der **folgende**, explizite **Weg**:

▶▶ make -f Makefile-flags1 --keep-going --silent 'CC=jcc' 'LD=jld'

Diese Definitionen werden also per MAKEFLAGS weitergereicht.

Hier haben wir einen **anderen Compiler und Linker** auf der Kommandozeile spezifiziert.

Variablendefinitionen auf der Kommandozeile gehen vor Definitionen im Makefile! Dies gilt **allgemein** und ist oft sehr nützlich.

Die Variable MAKEFLAGS gibt es bei Gnu-make, aber nicht bei allen makes. Sonst muß man die traditionelle Methode verwenden:

Makefile-flags1b:

```
.PHONY: all

all:
    +echo "Die Make-Flags auf Ebene $(MAKELEVEL) sind >$(MAKEFLAGS)<"
    $(MAKE) -f Makefile-flags2 $(MFLAGS)
```

Die **Variable** \$(MFLAGS) ist die traditionelle Art, Optionen weiterzugeben und geht auch

bei Gnu-make:

```
►► make -f Makfile-flags1b --keep-going --silent
```

Nachteil: **Variablendefinitionen** auf der Kommandozeile werden bei diesen anderen makes **nicht rekursiv** übergeben.

Die **Option -j** (Grad der **Parallelität**) ist ein **Sonderfall**: Die verschiedenen makes **sprechen sich direkt ab**, wieviele Anweisungen gerade ausgeführt werden. **Sonst** wäre die Anzahl **schwer zu begrenzen**. Dabei **zählen** die makes selbst **nicht mit**. Sonst würden evtl. alle Slots nur für die makes aufgebraucht.

4.7 Muster-Regeln

Außer normalen expliziten Regeln und Statischen-Muster-Regeln gibt es auch **Muster-Regeln**. Sie sieht aus **wie** eine **Statische-Muster-Regel**, aber der **erste Teil** mit der expliziten Liste der Ziele **fehlt**:

Makefile-patrulle:

```
prog: foo.o bar.o
    cc -o prog foo.o bar.o

%.o : %.c
    cc -c $< -o $@
```

Wenn hier **gesucht** wird, **wie ein foo.o erzeugt** werden kann, dann **paßt** diese Regel, **vorausgesetzt**, es steht ein **foo.c zur Verfügung**.

Das **Prozent-Zeichen** paßt wie bei den Statischen-Muster-Regeln auf den **Stamm** des Dateinamens, beide Prozent-Zeichen werden durch den **gleichen Text** ersetzt.

Auf der **rechten Seite** dürfen auch **mehrere Vorbedingungen** stehen.

Falls eine Vorbedingung **kein Prozent-Zeichen** enthält, dann ist sie für **alle Ziele gleich**. Dies ist nur manchmal sinnvoll, da die Anwendung nicht so genau eingeschränkt ist wie bei Statischen-Muster-Regeln.

Die **Reihenfolge** der Muster-Regeln ist **wichtig**. Es wird im wesentlichen die **erste Regel** genommen, die paßt.

Wenn eine Muster-Regel **mehrere Ziele** hat, wird das **anders** behandelt **als bei einer Statischen-Muster-Regel**:

Makefile-patrulle2:

```
# Quellen sind foo.y und foo.l

foo: foo.tab.o foo.o

%.tab.c %.tab.h : %.y
    # Option --defines schreibt auch *.tab.h:
    bison --defines $<

%.c : %.l %.tab.h
    lex -t $< > $@
```

Hier erzeugt der Parser-Generator **Bison** außer der Datei `foo.tab.c` **durch** die **Option** `--defines` auch eine **Datei** `foo.tab.h` mit Definitionen, die **für** den Scanner-Generator **Lex** **benötigt** werden.

`make` **erkennt**, daß die eine Anweisung gleich **zwei Ziele** erzeugt, und berücksichtigt das bei der Ausführung: Der Parser-Generator **Bison** wird **nur einmal** aufgerufen.

4.8 Altmodische Suffix-Regeln

In vielen Makefiles sieht man sog. **Suffix-Regeln**. Man braucht sie, weil **andere** `makes` und **alte** `makes` Muster-Regeln **nicht unterstützen**. Muster-Regeln sind **flexibler**.

Makefile-suffix:

```
prog: foo.o bar.o
    cc -o prog foo.o bar.o

.c.o:
    cc -c $< -o $@
```

Dies ist **äquivalent** zur schon oben gezeigten Muster-Regel:

Makefile-patrule:

```
prog: foo.o bar.o
    cc -o prog foo.o bar.o

%.o : %.c
    cc -c $< -o $@
```

Man beachte, daß die **Reihenfolge** der **Suffixe andersherum** ist als in der Muster-Regel.

Es ist auch das **leere Suffix** als Ziel erlaubt, dann hat die Regel **nur ein Suffix**:

Makefile-suffix2:

```
foo: foo.o

.o:
# % : %.o
    cc -o $@ $<

.c.o:
# %.o : %.c
    cc -c $< -o $@
```

Suffix-Regeln dürfen **keine Vorbedingungen** haben. Versucht man **trotzdem**, eine anzugeben, wird es als **normale Regel** mit einem **merkwürdigen Ziel-Namen** interpretiert: Makefile-suffix3:

```
prog: foo.o bar.o
    cc -o prog foo.o bar.o

.c.o: defs.h    # Falsch!
    cc -c $< -o $@
```

Suffix-Regeln funktionieren mit einer **Liste bekannter Suffixe**. Die bekannten Suffixe sind die **Vorbedingungen** des **besonderen Ziels** .SUFFIXES: Makefile-suffix4:

```
.SUFFIXES: .csp

prog: foo.o bar.o
    cc -o prog foo.o bar.o

.csp.o:
    csp-compile -c $< -o $@
```

Gibt man Vorbedingungen hierfür an, werden sie **zur Standardliste dazugefügt**. **Sonderbehandlung:** Eine Regel für .SUFFIXES **ohne Vorbedingungen** löscht die aktuelle Liste. **Danach** kann man **neue** Vorbedingungen definieren.

4.9 Automatische Variablen

Ein paar der automatischen Variablen sind schon vorgestellt worden. Hier nun eine **komplette Liste**:

Automatische Variablen

\$@ Ziel der Regel.
Falls Archivmitglied, dann Name des Archivs.
Falls mehrere Ziele, dann regelauslösendes Ziel.

\$\$ Vollständiges **Ziel**, falls **Archivmitglied**.
Falls Ziel kein Archivmitglied, dann leer.

\$< Erste **Vorbedingung** dieser Regel.

\$? Alle **neueren Vorbedingungen**.
Falls eine Vorbedingung Archivmitglied ist, dann nur Mitgliedsname.

\$^ Alle **Vorbedingungen**.
Falls eine Vorbedingung Archivmitglied ist, dann nur Mitgliedsname.
Doppelnennungen werden eliminiert.

\$+ Alle **Vorbedingungen**, wie **\$^**, aber **Doppelnennungen werden nicht eliminiert**.

\$* Stamm des Namens in der Regel.

\$(@D), \$(%D), \$(<D), \$(?D), \$(^D), \$(+D), \$(*D): **Directory-Teil** des Namens

\$(@F), \$(%F), \$(<F), \$(?F), \$(^F), \$(+F), \$(*F): **Datei-im-Directory-Teil** des Namens

Kurs Unix-Tool, WS 01/02, Jan Bredereke

25

Einschub: Archivmitglieder als Ziel.

```
foolib(hack.o) : hack.o
    ar cr foolib hack.o
```

Das **fügt** hack.o in das **Archiv** foolib ein. („r“ schreibt, „c“ stellt Warnungen ab, wenn noch nicht drin.) Es **gibt schon** eine **implizite Regel**, die genau dies tut.

Man kann diese Archiv-Notation **in Zielen und in Vorbedingungen** anwenden, aber **nicht in Anweisungen**. Dort helfen aber die automatischen Variablen.

Man darf bei der Archiv-Notation ggf. auch **mehrere Mitgliedsnamen** in die runden Klammern schreiben, getrennt durch Blanks.

Bei **\$@** können **mehrere Ziele** vorkommen, wenn wir eine **Muster-Regel** haben.

\$\$ ist **Gnu-make-spezifisch**.

\$+ ist nützlich, wenn es auf die Reihenfolge von Bibliotheken beim Linken ankommt.

Bei **\$*** hängt die Definition des „Stamms“ von der Art der Regel ab:

Statische-Muster-Regel: Genau das, was auf **% matcht**.

Muster-Regel: Das, was auf **% matcht**, ggf. mit unverändertem **Directory-Pfad davor**.
Aus dir/a.foo.b und dem Muster a.%b wird dir/foo .

Explizite Regel: Sonderbehandlung: Hier gibt es keinen Stamm. Daher **Ziel minus bekannte Suffixe**. Nur aus Kompatibilitätsgründen, man sollte es **vermeiden**.

Die **D/F-Formen** sind **Gnu-make-spezifisch**.

Beispiele:

```
$@ = dir/foo.o
$(@D) = dir
$(@F) = foo.o

$@ = bar
$(@D) = .
```

Es ist **kein Slash am Ende** des Directory-Teils.

Bei **\$\$** bezieht sich der Directory-Teil auf ***innerhalb*** des Archivs.

4.10 Ein größeres Beispiel: Makefile zu genFamMem

Nach der Weihnachtspause ein **größeres Beispiel zur Auffrischung**: Das Makefile des Programms **genFamMem** von Jan Brederke.

genFamMem ist ein Generator für Familien von formalen Anforderungen in CSP-OZ.
Homepage: <http://www.tzi.de/~brederek/genFamMem/>

Struktur der Quellen von genFamMem					
INSTALL	genFamMem /				
	bin/	dist/	src/	sty/	test/
	<i>genFamMem</i>	<i>genFamMem.tgz</i>	Makefile	*.sty	*
	*.o		*.c		*good
	*.d		*.l		*log
	*.tab.c		*.y		
	*.tab.h				
	*.tab.o				
	*.tab.d				

Legende:	
normal	handgeschriebene Datei
kursiv	generierte Datei / Verzeichnis

Kurs Unix-Tools, WS 01/02, Jan Brederke

26

Makefile-genFamMem:

```
# -----
#
# project: genFamMem - maintaining families of requirements
#
# author: Jan Brederke
# copyright: Jan Brederke, 2000
# $Id: Makefile,v 1.51 2001/02/22 12:39:36 brederek Exp $
#
# file: Makefile - main Makefile of the generator program
#
# You will need GNU's gmake for this.
# -----

# definitions:

# directories relative to the current dir:
BINDIR=../bin
TESTDIR=../test
STYDIR=../sty
DISTDIR=../dist
DISTNAME=genFamMem
```

```

DISTRROOT=../..
# directory and file relative to $(DISTRROOT)
DISTDIR_ROOT=$(DISTNAME)/dist
TARBALL=$(DISTDIR)/genFamMem.tgz
TARBALL_ROOT=$(DISTDIR_ROOT)/genFamMem.tgz
DISTCONTENTS=$(DISTNAME)/INSTALL $(DISTNAME)/src $(STYLEFILES_ROOT)
# the rest is relative to the current dir, again:
EXECUTABLE=$(BINDIR)/genFamMem
CSOURCES=main.c relation.c identifier.c genidentifier.c linerange.c calc.c \
    filterspec.c debug.c printuseshier.c
YSOURCES=firstpass.y
LSOURCES= firstpasstex.l secondpasstex.l firstpasslex.l
TSOURCES= z1.tex cspoz1.tex case-study.tex case-study-uh-fm.tex \
    case-study-uh-fm-of.tex case-study-uh-f-usf.tex cspoz2.tex
TDIRSOURCES=$(TSOURCES:%=$(TESTDIR)/%)
TDIRLOGS=$(TDIRSOURCES:%=%.log)
COBJECTS=$(CSOURCES:%.c=$(BINDIR)/%.o)
YOBJECTS=$(YSOURCES:%.y=$(BINDIR)/%.tab.o)
YGENCS=$(YSOURCES:%.y=$(BINDIR)/%.tab.c)
YGENHS=$(YSOURCES:%.y=$(BINDIR)/%.tab.h)
YGENOUTPUTS=$(YSOURCES:%.y=$(BINDIR)/%.output)
LOBJECTS=$(LSOURCES:%.l=$(BINDIR)/%.o)
LGENCS=$(LSOURCES:%.l=$(BINDIR)/%.c)
OBJECTS=$(COBJECTS) $(LOBJECTS) $(YOBJECTS)
GENCS=$(LGENCS) $(YGENCS) $(YGENHS) $(YGENOUTPUTS)
CDEPENDS=$(CSOURCES:%.c=$(BINDIR)/%.d)
YDEPENDS=$(YSOURCES:%.y=$(BINDIR)/%.tab.d)
LDEPENDS1=$(LSOURCES:%.l=$(BINDIR)/%.d)
LDEPENDS=$(filter-out $(BINDIR)/firstpasslex.d,$(LDEPENDS1))
DEPENDS=$(CDEPENDS) $(YDEPENDS) $(LDEPENDS)
TESTLOGS=$(TDIRSOURCES:%=%.log)
TESTGOODS=$(TDIRSOURCES:%=%.good)
FAMSTYLEFILES=$(STYDIR)/oz-fam.sty $(STYDIR)/csp-oz-fam.sty
EXTERNALSTYLEFILES=$(STYDIR)/csp-oz.sty $(STYDIR)/oz.sty
EXTERNALSTYLEFILESSOURCES=/home/brederek/doc/sty/csp-oz.sty \
    /home/brederek/doc/sty/oz.sty
STYLEFILES=$(FAMSTYLEFILES) $(EXTERNALSTYLEFILES)
STYLEFILES_ROOT=$(STYLEFILES:%$(STYDIR)/%=$(DISTNAME)/sty/%)

CPPFLAGS=-I$(BINDIR)
CFLAGS=-g -DYYDEBUG
YFLAGS=--verbose
YACC=bison
LEX=flex
# Gnu tar:
TAR=tar

# -----

# targets that might be used manually:

```

```

all: $(EXECUTABLE)

maintainer-all: all $(EXTERNALSTYLEFILES)

depend:
    rm -f $(DEPENDS)
    $(MAKE) $(DEPENDS)

clean:
    rm -f $(GENCS) $(OBJECTS) $(EXECUTABLE) $(TESTLOGS)

dist-clean:
    $(MAKE) clean
    rm -f $(DEPENDS)
    -rmdir $(BINDIR)

maintainer-clean:
    $(MAKE) dist-clean
    rm -f $(TARBALL)
    -rmdir $(DISTDIR)
    rm -f $(EXTERNALSTYLEFILES)

tests: $(TESTLOGS)

# (Currently, the tarball distribution contains the sources only, but not
# the (voluminous) test data. Ask me if you really need to do regression
# testing. J.B.)
dist: $(DISTDIR) $(EXTERNALSTYLEFILES)
    cd $(DISTRROOT) ; \
        $(TAR) --create --verbose --gzip --owner=0 --group=0 \
            --file=$(TARBALL_ROOT) \
            --exclude=CVS --exclude='*.swp' \
            $(DISTCONTENTS)

# -----

# some general issues:

# read the source dependences (which are auto-generated)
include $(DEPENDS)

# what are not real targets:
.PHONY: all maintainer-all depend clean dist-clean maintainer-clean tests dist

# in case of an error, generally delete a half-generated target:
.DELETE_ON_ERROR:

# -----

# the actual rules for file generation:

```



```

# the directory for binaries:
$(BINDIR):
    mkdir $(BINDIR)

# the directory for the distribution tar ball:
$(DISTDIR):
    mkdir $(DISTDIR)

# the executable:
$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) -o $(EXECUTABLE) $(OBJECTS) $(LDLIBS)

# the C objects:
$(COBJECTS): $(BINDIR)/%.o: %.c
    $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $$@ $$<

# the bison and flex objects:
$(YOBJECTS) $(LOBJECTS): %.o: %.c
    $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $$@ $$<

# the depend files:
$(CDEPENDS): $(BINDIR)/%.d: $(BINDIR) %.c
$(YDEPENDS): $(BINDIR)/%.tab.d: $(BINDIR) %.y
$(LDEPENDS): $(BINDIR)/%.d: $(BINDIR) %.l

# how to make an automatically generated dependences file for a source file:
$(BINDIR)/%.d: %.c
    $(CC) $(CFLAGS) -MM $$< | sed -e '1,1s,^,'"$(BINDIR)/," > $$@
$(BINDIR)/%.d: $(BINDIR)/%.c
    $(CC) $(CFLAGS) -MM $$< | sed -e '1,1s,^,'"$(BINDIR)/," > $$@
$(BINDIR)/%.tab.d: %.y
    $(CC) $(CFLAGS) -MM $$< | \
        sed -e '1,1s,^,'"$(BINDIR)/,"';1,1s/\.o: /\.tab.c: /\> > $$@
$(BINDIR)/%.tab.d: %.l
    $(CC) $(CFLAGS) -MM $$< | \
        sed -e '1,1s,^,'"$(BINDIR)/,"';1,1s/\.o: /\.c: /\> > $$@

# drop implicit rule for *.l:
%.c: %.l

$(BINDIR)/firstpass.tab.c: firstpass.y
    $(YACC) $(YFLAGS) --defines --name-prefix=fplyy --output-file=$$@ $$<

$(BINDIR)/firstpasslex.c: firstpasslex.l global.h
    $(LEX) $(LFLAGS) -Pfplyy -o$$@ $$<

$(BINDIR)/firstpasslex.o: $(BINDIR)/firstpass.tab.c

$(BINDIR)/firstpasstex.c: firstpasstex.l
    $(LEX) $(LFLAGS) -Pfpmtty -o$$@ $$<

```

```

$(BINDIR)/firstpasstex.o: $(BINDIR)/firstpasstex.c

$(BINDIR)/secondpasstex.c: secondpasstex.1
    $(LEX) $(LFLAGS) -Psptyy -o$@ $<

$(BINDIR)/secondpasstex.o: $(BINDIR)/secondpasstex.c

$(EXTERNALSTYLEFILES): $(EXTERNALSTYLEFILESOURCES)
    cp $^ $(STYDIR)

# -----

# tests

# all tests depend on the source file, on the regression comparison log, and
# on the executable:
$(TDIRLOGS): %.log: % %.good $(EXECUTABLE)

# how to run the individual tests:

$(TESTDIR)/z1.tex.log:
    $(EXECUTABLE) -M x $< > $@
    diff -q $<.good $@

$(TESTDIR)/cspoz1.tex.log:
    $(EXECUTABLE) --debug Member1 $< > $@
    diff -q $<.good $@

$(TESTDIR)/case-study.tex.log:
    $(EXECUTABLE) --debug SpecificationC $< > $@
    diff -q $<.good $@

$(TESTDIR)/case-study-uh-fm.tex.log:
    $(EXECUTABLE) --debug --uhier-familymember SpecificationC $< > $@
    diff -q $<.good $@

$(TESTDIR)/case-study-uh-fm-of.tex.log:
    $(EXECUTABLE) --uhier-familymember SpecificationC \
        --only-features $< > $@
    diff -q $<.good $@

$(TESTDIR)/case-study-uh-f-usf.tex.log:
    $(EXECUTABLE) --debug --uhier-feature UserSpace $< > $@
    diff -q $<.good $@

$(TESTDIR)/cspoz2.tex.log:
    $(EXECUTABLE) --debug Member1 $< > $@
    diff -q $<.good $@

```

Es geht los bei **all**:

Außerdem gibt es **clean:**, **tests:** und **dist:**.

maintainer-all:, **dist-clean:** und **maintainer-clean:** sind Varianten.

depend: wird später erläutert.

Alle sind als **.PHONY:** deklariert.

Regel für **\$(EXECUTABLE)**.

\$(OBJECTS) zusammengesetzt aus drei Teilen.

\$(OBJECTS) erzeugt aus **\$(CSOURCES)** mit Hilfe von Textersetzung. (Wird später im Detail erläutert.)

\$(CSOURCES) ist Liste der C-Quellen.

Analog für **\$(YSOURCES)** und **\$(LSOURCES)**.

\$(TSOURCES) sind die Test-Eingaben. **tests:** hängt von **\$(TESTLOGS)** ab, den aktuellen Testausgaben. **\$(TESTGOODS)** sind die Vergleichsdateien dazu. Am Ende des Makefiles ist die genaue **Durchführung der Tests** spezifiziert, mit allen Parametern.

Die **\$(OBJECTS)** werden mit einer Muster-Regel erzeugt. Dabei ist zu beachten, daß die Objekt-Datei in einem **anderen Verzeichnis** abgelegt wird.

Die Übersetzung der **Lex- und Yacc-Quellen** erfolgt in **zwei Schritten**: Erst nach C, dann in eine Objekt-Datei.

Bei **Yacc** ist es **noch** etwas **komplexer**: Aus der **.y-Datei** werden **zwei Dateien** erzeugt, eine **.tab.c-Datei** und eine **.tab.h-Datei**, die für Lex benötigt wird.

Daher benötigen wir eine **explizite Abhängigkeit** von **firstpasslex.o**, der Lex-Objekt-Datei, von **firstpass.tab.h** (bzw. von **firstpass.tab.c**, die gleichzeitig erzeugt wird).

Dieses Programm hat **einen Yacc-Parser** und **drei Lex-Scanner**. Mit der **Lex-Option -P** werden sie durch ein Präfix auseinandergehalten. Daher brauchen wir jeweils separate, **explizite Regeln**.

Alle **generierten Dateien**, auch *.c und *.h, werden im **bin/-Verzeichnis** abgelegt.

Die **Abhängigkeiten durch include-Anweisungen** usw. in den Quellen werden **automatisch extrahiert** und im Makefile berücksichtigt. Deswegen zu Anfang **include \$(DEPENDS)**

\$(DEPENDS) wird wieder aus **drei Teilen** zusammengesetzt. Diese werden wiederum **aus den Quelldateien** per **Textersetzung** generiert.

Zu jeder Quelldatei gibt es eine **.d-Datei** mit den Abhängigkeiten:

calc.d:

```
../bin/calc.o: calc.c global.h relation.h linerange.h identifier.h \
genidentifier.h calc.h
```

Diese Dateien werden **mit expliziten Regeln** aus den .c- .y- und .l-Dateien **erzeugt**.

Die **Option -MM** von gcc gibt eine solche Datei aus. Mit **sed** bearbeiten wir sie nach, damit die nach bin/ verschobene Objektdaten richtig angesprochen wird.

Details zu generierten Makefiles folgen **später**.

4.11 Implizite Regeln

Implizite Regeln sind **entweder Muster-Regeln oder altmodische Suffix-Regeln**. Implizite Regeln werden **angewandt**, **wenn** zwar eine **Abhängigkeit** spezifiziert ist, **aber keine Anweisungen** dafür gegeben sind.

Wir hatten implizite Regeln **bereits** kurz **zu Anfang** kennengelernt:

editor-simple/Makefile-impl:

```
# Makefile for "edit".

objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
       cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
       -rm edit $(objects)
```

Hier wurde der **C-Compiler** implizit aufgerufen.

Es sind eine ganze **Reihe von Regeln** bereits eingebaut, **weitere** Regeln kann man **auf** die schon **bekannte Weise** definieren.

Mit einer Reihe von **Variablen** in den **eingebauten Regeln** kann man deren **Verhalten anpassen**.

Einige der Regeln:

Einige Implizite Regeln

```
# C kompilieren:
.c.o:
    $(CC) $(CPPFLAGS) $(CFLAGS) $(TARGET_ARCH) -c -o $@ $<
# C++ kompilieren:
.cc.o:
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) $(TARGET_ARCH) -c -o $@ $<
.C.o:
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) $(TARGET_ARCH) -c -o $@ $<
# Pascal kompilieren:
.p.o:
    $(PC) $(PFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c -o $@ $<
# einzelne Objekt-Datei linken:
.o:
    $(CC) $(LDFLAGS) $(TARGET_ARCH) $^ $(LOADLIBES) $(LDLIBS) -o $@
# Yacc nach C kompilieren:
.y.c:
    $(YACC) $(YFLAGS) $< && mv -f y.tab.c $@
# Lex nach C kompilieren:
.l.c:
    $(LEX) $(LFLAGS) -t $< > $@
```

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

27

Es gibt noch **viel mehr Regeln**. Die **komplette Liste** kann man sich mit `make -p -f /dev/null` ausgeben lassen.

In **Wirklichkeit** sind diese Regeln **komplizierter** aufgebaut, z.B. gibt es eine Variable `$COMPILE.c`, die die gesamte Zeile für die C-Kompilation enthält. Ebenso gibt es noch eine Variable, mit der man die Ausgabe-Option `-o` global umdefinieren kann.

Interessant sind aber die **gezeigten Variablen**.

Die **Compiler-Variablen** sind geeignet **vorbesetzt**, z.B. mit `cc`, `g++`, `pc`, `yacc` und `lex`. Man kann sie ggf. **umdefinieren**.

Die **Variablen für zusätzliche Argumente** sind normalerweise **leer**, aber man kann sie **ebenfalls umdefinieren**.

`CPPFLAGS` sind die zusätzlichen **Optionen** für den **C-Präprozessor**, `CFLAGS` die für den **eigentlichen C-Compiler**.

4.12 Verkettung impliziter Regeln

Manchmal kann eine **Datei** durch eine **Verkettung impliziter Regeln** erzeugt werden. Beispiel:

►► `cd chain`

Makefile:

```
# Vorhandene Datei: foo.l
# Eingebaute implizite Regeln: .l.c, .c.o und .o
foo:
```

Dieses Makefile ist reichlich **minimalistisch**!

►► `make`

(Anmerkung: Es ginge sogar ganz *ohne* Makefile mit „make foo“, s.u.)

Was passiert? make weiß, daß die Datei `foo` **erzeugt werden muß**, und es weiß **durch Nachschauen**, daß die Datei `foo.o` **existiert**. In der Sammlung impliziter Regeln **findet** es einen **Weg**, das Ziel aus der vorhandenen Datei **zu erzeugen**.

Die Dateien `foo.c` und `foo.o` sind **Zwischendateien**.

Zwischendateien sind Dateien, die im Makefile **nicht erwähnt** werden.

Sie **sind anders** in zwei Punkten:

- Sie werden **nur erzeugt**, wenn sie wirklich **benötigt** werden.
- Sie werden **am Ende** des make-Laufes **gelöscht**. (Siehe `rm`.)

Man kann auch eine **im Makefile erwähnte Datei zu einer Zwischendatei** machen. Dafür muß man sie als Voraussetzung zum **besonderen Ziel** `.INTERMEDIATE` angeben.

Man kann auch ein **Mittelding** haben: Nur erzeugt, **wenn nötig**, aber **nicht automatisch gelöscht**: Dafür muß man die Datei als Voraussetzung zum **besonderen Ziel** `.SECONDARY` angeben.

Makefile2:

```
# Vorhandene Datei: foo.o
# Eingebaute implizite Regeln: .l.c, .c.o und .o
.INTERMEDIATE: foo.o
.SECONDARY: foo.c
foo:
```

►► `rm foo`

►► `make -f Makefile2`

Keine implizite Regel darf in einer Kette **mehr als einmal angewendet** werden. make wird also nicht einmal erwägen, `foo` aus `foo.o.o` zu erzeugen. Dies **vermeidet auch Totschleifen** bei der Regelsuche.

4.13 Details von Variablen

4.13.1 Syntax

Erlaubte Zeichen in Variablennamen: **Fast alle**, aber man sollte für normale Variablen **nur Buchstaben, Zahlen und Underscores** verwenden. Streng **verboten** sind **nur** „:“, „#“, „=“ und **Whitespace**.

Groß-/Kleinschreibung ist relevant.

Konventionen: Oft ganz **großgeschrieben**. **Gnu empfiehlt: Kleinschreibung** für normale, lokale Variablen, **Großschreibung** nur für Parameter impliziter Regeln oder Variablen, die evtl. auf der Kommandozeile undefiniert werden.

Verwendung von Variablen: **Dollarzeichen** und Name in **Klammern**, entweder **runde** oder **geschweifte Klammern**.

Wenn **ohne Klammern**, dann muß der Name **einbuchstabig** sein. Empfohlen **nur für** die **automatischen** Variablen.

Whitespace ist ggf. relevant:

Makefile-whitespace:

```
dir = /home/brederek    # Dies ist FALSCH!
file = foo.txt
pathfile = $(dir)/$(file)

all:

    @echo "Die Datei ist $(pathfile)."
```

►► make -f Makefile-whitespace

Whitespace **vor und nach dem Gleichheitszeichen** wird **ignoriert**, aber Whitespace **vor dem Zeilenende** wird **erhalten**. Dies kann zu Überraschungen führen.

Es gibt **kein Limit für** die **Länge** eines Variablenwertes, außer dem Swapspace.

Wenn ein Wert **undefiniert** ist, wird der **leere String** eingesetzt. Das passiert z.B. oft bei Variablen aus den impliziten Regeln.

4.13.2 Variablen-Expansion

Wenn **in einer Variablen-Definition** eine andere Variable vorkommt, dann wird sie **zunächst nicht** expandiert.

Die Variablen-**Expansion** findet **erst** statt, wenn die **äußere Variable verwendet** wird. Dies passiert dann **rekursiv**:

Makefile-var-rekursion:

```
foo:

    cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz
```

Vorteil: Die **Reihenfolge** der Definition ist **egal**.

Daher ist **folgendes nicht erlaubt**:

Makefile-var-rekursion2:

```
foo:
    cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

CFLAGS = $(CFLAGS) -Wall
```

►► make -f Makefile-var-rekursion2

make **erkennt** die **Rekursion** und meldet einen **Fehler**.

4.13.3 Anhängen an Variablen

Man kann diesen Effekt aber **anders erreichen**:

Makefile-var-append:

```
foo:
    cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

CFLAGS += -Wall
CFLAGS += -pg
```

►► make -f Makefile-var-append -n

Dies wird **häufig benutzt**. Gerade für die Variablen aus **impliziten Regeln** ist es sinnvoll.

Die **Reihenfolge** des Anhängens ist die Reihenfolge in der Datei.

Ansonsten wird bei dieser Operation der Wert der Variablen **noch nicht expandiert**, dies passiert erst bei der Verwendung. Es wird an den un-expandierten Text angehängt.

4.13.4 Zwei Arten von Variablen

Als **Gnu-Erweiterung** gibt es **zwei Arten** von Variablen:

- normale, rekursiv expandierte Variablen
- einfach expandierte Variablen

In den meisten **Fällen** sind die **normalen**, rekursiv expandierten Variablen **das richtige**. Aber **Nachteile**: Expansion **bei jeder Verwendung**. Schlecht z.B., wenn ein **Shell-Aufruf** darin.

Einfach expandierte Variablen werden zum **Definitionszeitpunkt** ein einziges Mal expandiert:

Makefile-var-simple-expand:

```
x := foo
y := $(x) bar
x := later

all:
    @echo "x = >$(x)<, y = >$(y)<"
```

►► make -f Makefile-var-simple-expand

Kennzeichen ist eine Zuweisung mit „:=“ statt „=“.

Diese Variablen **funktionieren wie in Programmiersprachen**, mit allen **Problemen**, die das bringt. Manche Leute **mögen** es aber **lieber**.

Man kann jetzt auch **mit Variablen „rechnen“**:

Makefile-var-simple-expand2:

```
include_dirs := -Ibar -Ibaz
CFLAGS := $(include_dirs) -O
CFLAGS := $(CFLAGS) -Wall

foo:
    cc -o foo $(CFLAGS) foo.c
```

►► make -f Makefile-var-simple-expand2 -n

Aber zum Anhängen ist das **nicht notwendig**, s.o.

Vorsicht mit der **Reihenfolge** der Zeilen! (Ich mußte **alles umstellen**.)

Anmerkung: Der **+=-Operator** geht natürlich **auch** für einfach expandierte Variablen.

4.13.5 override

Man kann **Variablen auf der Kommandozeile** setzen, z.B. CFLAGS. **Was ist**, wenn man im Makefile **trotzdem** immer noch eine Option **anhängen** will?

Makefile-var-append:

```
foo:
    cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

CFLAGS += -Wall
CFLAGS += -pg
```

►► make -f Makefile-var-append 'CFLAGS=-Ibar -Ibaz' -n

(Das ist **ohne** das -O .)

Das **geht schief**, die angehängten Sachen gehen verloren.

Dies geht mit override:

Makefile-var-append2:

```
foo:
    cc -o foo $(CFLAGS) foo.c

#

CFLAGS = $(include_dirs) -O
include_dirs = -Ibar -Ibaz

override CFLAGS += -Wall
override CFLAGS += -pg
```

►► make -f Makefile-var-append2 'CFLAGS=-Ibar -Ibaz' -n

So **klappt** es.

Anmerkung: Man kann override **auch für einfache Zuweisungen** benutzen, aber **sinnvoll** ist es meist **nur für** das **Anhängen**.

4.13.6 Bedingte Zuweisung

Wenn man das Makefile aus verschiedenen Teilen zusammensetzt, ist manchmal die bedingte Zuweisung sinnvoll:

Makefile-var-cond:

```
myflags ?= -frob

foo:
    bar $(myflags) $@

#####

myflags = -frob2
```

►► make -f Makefile-var-cond -n

►► make -f Makefile-var-cond -n ohne die letzte Zeile

Man braucht das aber **eher selten**. Denn **Kommandozeilen-Zuweisungen** gehen ohnehin **vor**!

Sinnvoll ist es auch, wenn man erwartet, daß Make-Variablen durch **Variablen aus dem Environment** gesetzt werden. Aber das **ist selbst** ziemlich **gefährlich**.

Ansonsten funktioniert es **gleich für beide Arten** von Variablen.

4.13.7 Ersetzungen

Ersetzungen sind schon kurz im obigen **genFamMem-Beispiel** angesprochen worden.

Eine Ersetzung ist eine **Variablenreferenz**, bei der auf den Namen ein **Doppelpunkt** folgt. Danach kommt die **zu ersetzende** Endung, ein **Gleichheitszeichen**, und die **neue** Endung:

Makefile-subst:

```
LSOURCES= firstpasstex.l secondpasstex.l firstpasslex.l
OBJECTS=$(LSOURCES:.l=.o)

foo:
    @echo $(OBJECTS)
```

►► make -f Makefile-subst

Es wird **jedes Wort** in der Variablen **getrennt bearbeitet**. Worte sind durch **White-space** abgetrennt.

Es werden **nur** die **Wort-Enden** bearbeitet.

Bei **Gnu-make** gibt es eine **erweiterte** Form: Mit **Prozentzeichen**:

Makefile-subst2:

```
LSOURCES= firstpasstex.l secondpasstex.l firstpasslex.l
OBJECTS=$(LSOURCES:%.l=../bin/%.o)

foo:
    @echo $(OBJECTS)
```

►► `make -f Makefile-subst2`

Jetzt kann man z.B. **auch** einen **Präfix** davorhängen oder entfernen. (Wie bei `genFamMem` gemacht.)

4.14 Details des Aufrufs von `make`

4.14.1 Angabe von Zielen

Das **Default-Ziel**, das von `make` erzeugt wird, ist das **erste** Ziel.

Man kann aber auch **Ziele auf der Kommandozeile explizit** angeben:

Makefile-goal:

```
all: foo bar

foo:
    @echo "Working on foo..."

bar:
    @echo "Working on bar..."

foobar.tgz:
    @echo "Creating a distribution of foo and bar..."

clean:
    @echo "Cleaning away generated files of foo and bar..."

.PHONY: clean
```

►► `make -f Makefile-goal foo`

Erzeugt nur foo, aber nicht bar

Manche Ziele werden **überhaupt nur erzeugt**, wenn sie **explizit** benannt werden:

►► `make -f Makefile-goal foobar.tgz`

Und natürlich gibt es die **unechten Ziele** wie **clean**, die oft auf der Kommandozeile angegeben werden.

4.14.2 Kommandozeilenoptionen von make

Kommandozeilenoptionen von make	
<code>-f file, --file=file</code>	nimm <i>file</i> als Makefile
<code>-n, --dry-run</code>	nichts wirklich tun, nur Kommandos drucken
<code>-t, --touch</code>	nichts tun, aber Datum aktualisieren
<code>-q, --question</code>	nichts tun und nichts drucken, nur Returncode setzen
<code>var=value</code>	Variablenzuweisung, geht vor Zuweisungen im Makefile
<code>-j [jobs], --jobs[=jobs]</code>	parallele Ausführung, ggf. Obergrenze für Zahl der Jobs
<code>-l [load], --max-load[=load]</code>	Lastgrenze für parallele Ausführung
<code>-k, --keep-going</code>	auch bei einem Fehler noch so viele Regeln wie möglich versuchen
<code>-W file, --what-if=file,</code> <code>--assume-new=file</code>	} so tun, als ob <i>file</i> ganz neu sei (gut mit <code>-n</code>)
<code>-o file, --assume-old=file</code>	} so tun, als ob <i>file</i> ganz alt sei (gut mit <code>--touch</code> danach)

Kurs Unix-Tools, WS 01/02, Jan Bredeke 28

Kommandozeilenoptionen von make (2)	
<code>-h, --help</code>	drucke Hilfe
<code>-v, --version</code>	drucke make-Version
<code>-s, --silent</code>	drucke keine Kommandos
<code>-W, --print-directory</code>	drucke Verzeichnis am Anfang und Ende
<code>--no-print-directory</code>	hebt <code>--print-directory</code> auf, z.B. in Rekursion
<code>-C dir, --directory=dir</code>	wechsle erst ins Verzeichnis <i>dir</i>
<code>-d, --debug[=options]</code>	drucke Debug-Info
<code>-e, --environmen-overrides</code>	gib Variablen aus Environment Vorrang vor Makefile
<code>-i, --ignore-errors</code>	ignoriere alle Fehler
<code>-I dir, --include-dir=dir</code>	suche auch im Verzeichnis <i>dir</i> nach include-Makefiles
<code>-p, --print-data-base</code>	drucke alle Regeln und Variablen
<code>-r, --no-builtin-rules</code>	lösche alle eingebauten impliziten Regeln
<code>-R, --no-builtin-variables</code>	lösche alle eingebauten impliziten Regeln und die Variablen dazu
<code>-S, --no-keep-going</code>	hebe Wirkung von <code>--keep-going</code> auf, z.B. in Rekursion
<code>--warn-undefined-variables</code>	warn bei Verwendung undefinierter Variablen

Kurs Unix-Tools, WS 01/02, Jan Bredeke 29

Als-Ob-Ausführung, Option -n:

Haben wir hier schon **oft benutzt**.

Vermeiden unnötiger Neu-Kompilation, Option -t:

Haben wir schon im Zusammenhang mit der **Variablen \$(MAKEFLAGS)** diskutiert.

Kompilation eines Programms testen, -k:

Haben wir schon u.a. im Zusammenhang mit den **Returncodes** diskutiert.

Variablen überschreiben:

Haben wir im Zusammenhang mit **override** schon diskutiert.

4.15 Konventionen für Makefiles

Man sollte **nicht beliebige Systemkommandos** im Makefile benutzen, damit es **portabel** bleibt. Die folgenden sind unkritisch:

Erlaubte System-Hilfsprogramme in Makefiles	
cat	mkdir
cmp	mv
cp	pwd
diff	rm
echo	rmdir
egrep	sed
expr	sleep
false	sort
grep	tar
install-info	test
ln	touch
ls	true
Nicht erlaubt: mkdir -p	
Problematisch: Symbolische Links	
Kurs Unix-Tools, WS 01/02, Jan Brederke	
30	

System-Hilfsprogramme mit veränderlichen Namen
AR=ar
BISON=bison
CC=cc
FLEX=flex
INSTALL=install
LD=ld
LDCONFIG=ldconfig
LEX=lex
MAKE=make
MAKEINFO=makeinfo
RANLIB=ranlib
TEXI2DVI=texi2dvi
YACC=yacc
CHGRP=chgrp
CHMOD=chmod
CHOWN=chown
...
Benutzung als \$(AR), \$(BISON), \$(CC), ...
Kurs Unix-Tools, WS 01/02, Jan Brederke
31

ranlib ist nicht überall vorhanden und muß ggf. ignoriert werden.

Besonders für die **Compiler-Programme** sollten Variablen vorgesehen werden, um die **Aufruf-Flags zu variieren**. Ein Beispiel war **CFLAGS** oben für CC.

Standard-Variablen für Installationsverzeichnisse

```
prefix=/usr/local
exec_prefix=${prefix}
bindir=${exec_prefix}/bin
libdir=${exec_prefix}/lib
infodir=${prefix}/info
includedir=${prefix}/include
mandir=${prefix}/man
man1dir=${prefix}/man1
man2dir=${prefix}/man2
manext=.1
man1ext=.1
man1ext=.2
...
srcdir=(bei Konfiguration gesetzt)
```

Kurs Unix-Tools, WS 01/02, Jan Brederke

32

Vorteil dieser Variablen: Man kann **alle Gnu-Pakete** auf dieselbe Weise **an** die **aktuelle Installation anpassen**. Beispiel: **Installationsverzeichnis**.

Standard-Ziele bei Gnu-make

all	alles Übersetzen (Default-Ziel)
install	alles Übersetzen und installieren
uninstall	lösche alle installierten Dateien
clean	lösche alle durch all übersetzten Dateien im Build-Verzeichnis
distclean	dito, lösche zusätzlich Konfigurationsdateien
maintainerclean	lösche alles, was mit diesem Makefile wiederhergestellt werden kann
info	erzeuge .info-Dateien
dvi	erzeuge .dvi-Dateien
dist	erzeuge eine Distributions-tar-Datei
check	führe ggf. Selbst-Tests durch
...	...

Kurs Unix-Tools, WS 01/02, Jan Brederke

33

4.16 Fortgeschrittene Makefile-Strukturen

4.16.1 Bedingte Teile eines Makefiles

Man kann **Teile** des Makefiles nur **unter Bedingungen** abarbeiten lassen:
Makefile-cond:

```

libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
    $(CC) -o foo $(objects) $(libs_for_gcc)
else
    $(CC) -o foo $(objects) $(normal_libs)
endif

```

Bedingte Ausdrücke

```

ifeq (arg1,arg2)
ifneq (arg1,arg2)
ifdef variable-name
ifndef variable-name

```

Varianten für die Syntax:

```

ifeq (arg1,arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq 'arg1' "arg2"
ifeq "arg1" 'arg2'

```

Kurs Unix-Tools, WS 01/02, Jan Brederke

34

Diese Syntax ist **Gnu-spezifisch**. Andere makes kennen bedingte Teile oft auch, aber mit leicht **variierender Syntax**.

4.16.2 Andere Makefiles einschließen

Mit

```
include datei1 datei2 ...
```

kann man **andere Makefiles einlesen**.

Wenn eine Datei **nicht existiert**, ist das zunächst noch **kein Fehler**. Wenn make den Rest eingelesen hat, versucht es anschließend, dieses **Makefile nach gegebenen Regeln zu erzeugen**. Erst wenn das nicht gelingt, gibt es einen Fehler.

4.16.3 Dynamisch generierte Abhängigkeiten

Es ist **sehr mühsam**, alle **Abhängigkeiten** der Quellen **von Hand** korrekt einzutragen und aktuell zu halten.

Die meisten modernen Compiler können solche **make-Abhängigkeiten automatisch generieren**. Man kann sie dann per **include** einschließen.

Beispiel: Siehe **genFamMem-Beispiel** von oben:.

Makefile-genFamMem

Man beachte auch, wie wir **explizit angegeben** haben, **wovon die .d-Dateien jeweils abhängen**. Wenn z.B. eine .c-Datei geändert wird, wird die **.d-Datei automatisch neu erzeugt**.

Wenn die **includeten Makefiles nicht existieren** oder veraltet sind, **geht make so vor**: In einer **ersten Runde** erzeugt **make nur die includeten Makefiles neu**. Danach **fängt es noch einmal ganz von vorne an**, mit jetzt aktuellen Makefiles.

4.17 Eingebaute besondere Ziele im Detail

Eingebaute besondere Ziele	
<code>.PHONY: ziel1 ziel2 ...</code>	Liste der unechten Ziele
<code>.DELETE_ON_ERROR:</code>	bei Fehler wird das aktuelle Ziel gelöscht (gilt für alle)
<code>.PRECIOUS: ziel1 ziel2 ...</code>	Zieldateien bei Fehler und Unterbrechung niemals gelöscht
<code>.INTERMEDIATE: ziel1 ziel2 ...</code>	nur bei Bedarf erzeugt, nach Gebrauch gleich wieder gelöscht
<code>.SECONDARY: ziel1 ziel2 ...</code>	nur bei Bedarf erzeugt, aber nicht wieder gelöscht
<code>.SUFFIXES: .c .o ...</code>	Liste der Dateinamen-Suffixe für Suffix-Regeln
<code>.DEFAULT: ; anweisung ...</code>	Anweisungen für Vorbedingungen, die kein Ziel sind
<code>.NOTPARALLEL:</code>	keine Parallelausführung, selbst bei Option <code>-j</code>
<code>.IGNORE: ziel1 ziel2 ...</code>	ignoriere Fehler für diese Ziele (Wenn leer: Für alle)
<code>.SILENT: ziel1 ziel2 ...</code>	drucke keine Kommandos für diese Ziele (Wenn leer: Für alle)
<code>.EXPORT_ALL_VARIABLES:</code>	exportiere alle Make-Variablen an Kindprozesse

Kurs Unix-Tools, WS 01/02, Jan Brederke 35

4.18 Funktionen

Sind alle **Gnu-Erweiterung**.

4.18.1 Funktionen, die Text transformieren

Funktionen, die Text transformieren	
<code>\$(subst <i>from,to,text</i>)</code>	Textersetzung: <code>\$(subst ee,EE,feet on the street)</code> ergibt 'fEEt on the strEEt'
<code>\$(patsubst <i>pattern,replacement,text</i>)</code>	Textersetzung mit Muster: <code>\$(patsubst %.c,../bin/%.o,foo.c.c bar.c)</code> ergibt '../bin/foo.c.o ../bin/bar.o'
<code>\$(strip <i>string</i>)</code>	Führenden, anhängenden und verdoppelten Whitespace entfernen: <code>\$(strip a b c)</code> ergibt 'a b c'
<code>\$(findstring <i>find,in</i>)</code>	Sucht Teilstring in String.
<code>\$(filter <i>pattern ... ,text</i>)</code>	Nur die Worte im Text, auf die die Muster passen.
<code>\$(filter-out <i>pattern ... ,text</i>)</code>	Nur die Worte im Text, auf die die Muster <i>nicht</i> passen.
<code>\$(sort <i>list</i>)</code>	Sortiert Liste von Worten, entfernt Doppel.

Kurs Unix-Tools, WS 01/02, Jan Brederke 36

Statt `$(patsubst ...)` kann man auch einfacher die Ersetzungen bei Variablenreferenzen nehmen (s.o.)

`$(strip ...)` ist gut für Stringvergleiche in bedingten Anweisungen.

4.18.2 Funktionen für Dateinamen

Funktionen für Dateinamen	
<code>\$(dir <i>names ...</i>)</code>	Directory-Anteile der Namen in der Liste
<code>\$(notdir <i>names ...</i>)</code>	Namen in der Liste <i>ohne</i> ihre Directory-Anteile
<code>\$(suffix <i>names ...</i>)</code>	Liste der Suffixe der Namen
<code>\$(basename <i>names ...</i>)</code>	Namen in der Liste <i>ohne</i> ihre Suffixe
<code>\$(addsuffix <i>suffix,names ...</i>)</code>	Hängt den Suffix an jeden Namen in der Liste an
<code>\$(addprefix <i>prefix,names ...</i>)</code>	Hängt den Prefix vor jeden Namen in der Liste
<code>\$(join <i>list1,list2 ...</i>)</code>	Konkateniert die Worte der beiden Liste paarweise
<code>\$(word <i>n,text ...</i>)</code>	Liefert das n-te Wort
<code>\$(wordlist <i>s,e,text ...</i>)</code>	Liefert die Worte von Nummer s bis e
<code>\$(words <i>text ...</i>)</code>	Liefert die Anzahl der Worte
<code>\$(firstword <i>names ...</i>)</code>	Liefert das erste Wort der Liste
<code>\$(wildcard <i>pattern</i>)</code>	Expandiert das Dateinamen-Wildcard-Muster in eine Liste von existierenden Dateinamen

Kurs Unix-Tools, WS 01/02, Jan Brederke 37

4.18.3 Funktionen – Ganz hartes Zeugs

Funktionen – Ganz hartes Zeugs	
<code>\$(foreach var,list,text)</code>	Iteration
<code>\$(if condition,then-part[,else-part])</code>	Bedingte Expansion
<code>\$(call variable,param,param,...)</code>	Aufruf einer selbstdefinierten Funktion
<code>\$(origin variable)</code>	Test, ob Variable: definiert / aus Makefile / von Kommandozeile / ...
<code>\$(shell shell-anweisung)</code>	Shell-Anweisungs-Expansion (analog Backquotes in Skripten)
<code>\$(error text...)</code>	Generiere Abbruch durch Fehler mit Meldung <i>text</i>
<code>\$(warning text...)</code>	Druckt Warnmeldung, bricht aber nicht ab. Liefert leeren String.

Kurs Unix-Tools, WS 01/02, Jan Brederke 38

4.19 Gnu-make und andere makes

Gnu-make-Spezifika, die nicht mit allen makes gehen
<ul style="list-style-type: none">• include mehrerer Dateien auf einmal• Ersetzungen bei Variablenreferenzen• Muster-Regeln mit %• parallele Ausführung• Anhängen an Variablendefinitionen mit +=• einfach-expandierte Variablen• unechte Ziele mit .PHONY• Funktionen• bedingte Ausführung• Statische Muster-Regeln• generierte Makefiles <p>... also alles, was Spaß macht. :-)</p>

Kurs Unix-Tools, WS 01/02, Jan Brederke 39

4.20 Beispiel: LaTeX-Kompilierung der Skriptnotizen

Makefile-genFamMem:

```
#
# makefile fuer die Skriptnotizen und die Folien zum
# Kurs Unix-Tools im WS 01/02
#
```

```

# Jan Brederke, Universitaet Bremen
#

#####

# generierte include-Dateien

# Die folgende, generierte Datei enthaelt zum einen die Abhaengigkeiten
# von einzelfolien/folie-%.ps zu folien.dvi, und sie definiert zum
# anderen die Variable einzelfolien mit allen diesen Postscript-Dateien.
include folien-label.mk

# Die folgenden, generierten Dateien enthalten die Abhaengigkeiten der
# Dateien folien.dvi und skriptnotizen.dvi von mit \verbatiminput
# eingeschlossenen Quelldateien.
include folien.d
include skriptnotizen.d

#####

# Variablen

#####

# Ziele fuer den manuellen Aufruf

# das Default-Ziel:
all: skriptnotizen.dvi

# Loeschen aller generierten Dateien:
clean:
    -rm skriptnotizen.dvi
    -rm skriptnotizen.aux
    -rm skriptnotizen.log
    -rm skriptnotizen.d
    -rm $(einzelfolien)
    -rmdir einzelfolien
    -rm folien.dvi
    -rm folien.aux
    -rm folien.log
    -rm folien.d
    -rm folien-label.aux
    -rm folien-label.mk
    -rm folien-label.d

# Regel: alle diese Ziele sind keine echten Dateien:
.PHONY: all clean

#####

# allgemeine Regeln

```

```

# Uebersetzen von LaTeX-Dateien in DVI-Dateien:
%.dvi %.aux: %.tex
    latex $<

# Erzeugen einer Postscript-Datei einer einzelnen Overhead-Folie aus
# einer DVI-Datei mit vielen Folien:
einzelfolien/folie-%.ps: folien.dvi einzelfolien
    dvips -pp $* -o $@ folien

# Erzeugen der make-include-Dateien fuer Abhaengigkeiten durch
# \verbatiminput in *.tex-Dateien:
%.d: %.tex
    sed -n -e 's/^[^%]*\\verbatiminput{\\([~}\\+\\)}.*$$/$$.dvi: \\1/p' \
        $< > $@

#####

# konkrete Abhaengigkeiten

skriptnotizen.dvi: skriptnotizen.tex skriptnotizen.sty \
    $(einzelfolien) folien-label.aux

folien.dvi: folien.tex

einzelfolien:
    mkdir einzelfolien

folien-label.aux: folien.aux
    sed -n -e \
's/^[^\\]*(\\newlabel{folie:([a-zA-Z]\\+\\)}{\\([~}\\+\\)}{\\([0-9]\\+\\)}\\)}\\)$$/\\1\\newcommand{\\folie\\2}{\\3}/p' \
    $< > $@

folien-label.mk: folien-label.aux
    sed -e 's!.*{\\([0-9]\\+\\)}$$!skriptnotizen.dvi: einzelfolien/folie-\\1.ps!' \
        $< > $@ && \
    sed -e 's!.*{\\([0-9]\\+\\)}$$!einzelfolien+= einzelfolien/folie-\\1.ps!' \
        $< >> $@

folien.d: folien.tex
skriptnotizen.d: skriptnotizen.tex

```

4.21 autoconf/automake – ein Überblick

Autoconf/Automake – ein Überblick

Autoconf

- Erzeugt ein Shell-Skript, das ein SW-Paket automatisch konfiguriert.
- Nur der Paket-Autor braucht Autoconf, nicht der Benutzer.
- Keine Benutzer-Interaktion notwendig.
- Alle benötigten Features werden einzeln getestet: Hybride Systeme möglich.
- Die Skripten zur Feature-Erkennung werden von allen Paketen geteilt.

Automake

- Erzeugt Makefile-Template für Autoconf.
- Alle Gnu-Standard-Make-Ziele werden automatisch generiert.

Kurs Unix-Tools, WS 01/02, Jan Brederke40

5 Lexikalische Analyse mit lex

5.1 Einführung

Lexer:

Liest Eingabestrom und zerlegt ihn in Token.

Token:

- Endprodukt
- Weiter verarbeitet, z.B. von Parser (Yacc)

Lexer heißen auch „**Scanner**“ oder „**lexikalische Analysatoren**“.

lex ist ein **Generator** für **Lexer**.

Wir benutzen hier **flex**, den **Lexer** des **Gnu-Projektes**.

Das folgende **Beispiel** sucht nach dem **String** `<username>` und ersetzt ihn durch den aktuellen Benutzernamen. **Alle andere Eingabe** wird direkt in die Ausgabe **kopiert**:

`username.l`:

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
%}  
%option main  
%%  
"<username>"    printf("%s", getenv("USER"));
```

Interessant ist hier zunächst **nur** die **letzte Zeile**:

Vorne steht ein **Muster**, **hinten** eine **Aktion**.

Es gibt eine **eingebaute Default-Aktion**. Sie **druckt** den **aktuellen Buchstaben** einfach aus.

➤➤ `make username`

Anmerkung: Dies ist ein schönes Beispiel, wie man **ohne Makefile**, nur mit **makes** eingebauten Regeln, alles **übersetzen** kann.

Lex generiert ein **C-Programm**.

`username.txt`:

```
Hallo <username>, dies ist ein Beispieltext.
```

➤➤ `./username < username.txt`

5.2 Grundaufbau einer Lex-Datei

Lex-Datei: Definitionen %% Regeln %% evtl. Unterprogramme

5.2.1 Definitionen

In den Definitionen kann **einiges für Lex** definiert werden, wie oben z.B. eine **Option main**.

Außerdem kann man **Header-Files** für den generierten C-Code angeben, in „%{“ und „}%“.

5.2.2 Regeln

Regel: Muster Aktion

Jede **Regel** besteht aus einem **Muster** und einer **Aktion**, durch **Whitespace** getrennt.

5.2.3 Muster

Muster sind **reguläre Ausdrücke**, mit einigen **Erweiterungen**.

Text in **doppelten Anführungszeichen** ist **gequotet**. Ich **empfehle** das für alle **festen Token**. Im obigen username-Beispiel war es z.B. schon notwendig wg. der spitzen Klammern.

5.2.4 Aktionen

Eine Aktion ist ein **Stück C-Code**.

Entweder ist sie eine **einzelne C-Anweisung**, mit **Semikolon** abgeschlossen, wie oben gesehen.

Oder sie ist ein **C-Block** in **geschweiften Klammern**. Dann darf sie auch über **mehrere Zeilen** gehen.

5.2.5 Unterprogramm-Abschnitt

Im **optionalen** Unterprogramm-Abschnitt kann man **Hilfsfunktionen** definieren.

Im obigen **Beispiel** war er **leer** und **fehlte**.

Hierhin kommt auch die **main()-Funktion**. Im **Beispiel** wurde eine einfache main()-Funktion allerdings auch **automatisch generiert**, durch die Option **main**.

5.2.6 Beispiel: Wort-Zähl-Programm

```
wcount.l:
%{
#include <stdio.h>
int num_lines = 0, num_chars = 0;
%}
%option noyywrap
%%
"\n"          { num_lines++; num_chars++; }
.             { num_chars++; }
%%
int main() {
    yylex();
    printf("%d Zeilen, %d Buchstaben\n", num_lines, num_chars);
}
```

Der **Punkt** ist ein regulärer Ausdruck, der genau auf einen beliebigen Buchstaben (außer Newline) paßt.

Hier haben wir eine **selbstgeschriebene main()-Funktion**. Sie ruft **yylex()** auf, die generierte Lexer-Funktion.

▶▶ make wcount

▶▶ ./wcount < wcount.l

Zum Vergleich:

▶▶ wc wcount.l

5.2.7 Beispiel: Scanner für Pascal-artige Sprache, mit weiteren Lex-Konstrukten

pcount.l:

```
%{
#include <math.h>
#include <stdio.h>
%}
%option noyywrap
DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9_]*
%%
{DIGIT}+    { printf("Eine Ganz-Zahl: '%s' (%d)\n",
                  yytext, atoi(yytext));
              }
{DIGIT}+"."{DIGIT}*    {
                  printf("Eine Fließkomma-Zahl: '%s' (%g)\n",
                  yytext, atof(yytext));
              }
"if"|"then"|"begin"|"end"|"procedure"|"function"|"="|" ";"    {
                  printf("Ein Schlüsselsymbol: '%s'\n", yytext);
              }
{ID}        { printf("Ein Bezeichner: '%s'\n", yytext); }
"+"        |
"_"        |
"*"        |
"/"        { printf("Ein Operator: '%s'\n", yytext); }
{"[^]\n"}" /* ueberspringe einzeilige Kommentare */
[ \t\n]+    /* ueberspringe White-Space */
.          { printf("Unbekanntes Zeichen: '%s'\n", yytext); }
%%
int main(int argc, char **argv) {
    if(argc > 1)
        yyin = fopen(argv[1], "r");
    yylex();
}
```

Dies Programm **liest** ein **Pascal-artiges Programm** ein und **druckt**, was es an **Token** findet.

Es wird auch **etwas bearbeitet**. So werden die **Zahlen** aus den Strings **konvertiert**.

Hier gibt es **einiges neues**:

Die Variable `yytext` enthält immer den String des **aktuellen Tokens**.

Im Definitions-Teil kann man **Abkürzungen** für Teile regulärer Ausdrücke **definieren**.

Zur **Benutzung** muß man die definierten Namen **in geschweifte Klammern** einschließen.

Die **Schlüsselworte** werden mit „|“ getrennt, der **Alternative der regulären Ausdrücke**.

Die Operatoren haben „|“ **als Aktion**. Das heißt, daß die Aktion für diese Regel dieselbe ist wie für die folgende Regel. Der **Effekt** ist ungefähr wie eine reguläre-Ausdruck-Alternative auf oberster Ebene.

In der main()-Funktion wird der Datei-Pointer **yyin** verwendet. Lex liest die **Eingabe** von diesem Datei-Pointer. Per **Default** ist er gleich **stdin**. Man kann ihn aber auch wie hier umsetzen.

Eine **Eingabedatei zum Scanner**:

pcount.txt:

```
procedure foo
  function bar
  begin
    bar = 700000000.1
  end
begin
  tmp = bar
end
{ Beginn des Hauptprogramms: }
begin
  foo;
  zahl = 000042 + 8
end
```

▶▶ make pcount

▶▶ ./pcount < pcount.txt

5.3 Online-Aufgabe: Einfacher Taschenrechner

Schreibe einen einfachen Taschenrechner. Man soll **Fließkommazahlen und Operatoren eingeben** können. Operatoren sind „+“, „-“, „*“, „/“ und „C“.

Der Rechner soll einen **Akkumulator** für die aktuelle Zahl haben. Wenn eine **neue Zahl** eingegeben wurde, dann soll sie über dem **aktuellen Operator** mit dem Akkumulator **verknüpft** werden. Anschließend soll der neue Akkumulator **ausgegeben** werden.

Der **Operator „C“** überträgt die neue Zahl einfach in den Akkumulator. Bei seiner Eingabe wird außerdem der Akkumulator auf 0 gesetzt.

5.4 Match-Algorithmus

Der **generierte Scanner sucht** in seiner Eingabe nach **Strings**, auf die eines der **Muster paßt**.

Falls **mehrere passen**, nimmt er die Regel, auf die **am meisten Text** paßt.

Falls zwei Matches die **gleiche Länge** haben, nimmt er die Regel, die **zuerst** in der Lex-Datei steht.

Wenn der Match feststeht, wird der Text des Tokens in der **Variablen yytext** bereitgestellt, und die Länge in **yylen**.

Dann wird die zugehörige **Aktion ausgeführt**.

Dann geht es wieder **von vorne** los.

Falls **kein Match** gefunden wurde, gilt die **Default-Regel**: Ein einzelnes Zeichen paßt und wird gedruckt.

Das einfachste flex-Programm:

trivial.l:

```
%option main
%%
```

►► make trivial

►► ./trivial < pcount.txt

Durch die **Option main** wird eine einfache **main()-Funktion generiert**, die die Scanner-Funktion **yylex()** aufruft.

Am **Ende der Eingabe** kehrt **yylex()** mit dem **Return-Wert 0** zurück.

Man kann auch **innerhalb einer Aktion return(n)** ausführen lassen. Wird **yylex()** danach **erneut aufgerufen**, macht sie an der **nächsten Position** einfach weiter.

Den **Return-Wert** kann man z.B. **verwenden**, um die **Art des gefundenen Tokens** an den Aufrufer mitzuteilen. Auch die **globalen Variablen yytext** und **yylen** **bleiben** für den Aufrufer erhalten.

5.5 Lesen aus Strings

Wir wollen die **Kommandozeile** für ein einfaches Programm **auswerten**. (Wir wollen dabei **nicht** die **getopt()-Funktion** aus der C-Bibliothek nehmen, sondern selbst eine schreiben.)

Man kann auch **aus Strings** statt aus Dateien **lesen**.

Leider geht das bei verschiedenen **Versionen von Lex** mit **verschiedener Syntax**.

Bei **Flex** schaltet man mit der **Funktion yy_scan_string()** die Eingabe um:

cmdline.1:

```
%{
#include <stdio.h>
int verbose = 0;
char *progName = NULL;
}%
%option noyywrap
%%
^-h"          |
^-?"          |
^"--help"      { printf("usage is: %s [--help | -h | -?] ", progName);
                  printf("[--verbose | -v ...]\n");
                  exit(0);
                  }
^-v"          |
^"--verbose"   { verbose++; }
.*            { printf("unknown option '%s'!\n", yytext);
                  exit(1);
                  }

%%
int main(int argc, char *argv[]) {
    progName = *argv;
    while(++argv,--argc) {
        yy_scan_string(*argv);
        yylex();
    }
    printf("Now starting to frobnicate with:\n");
    printf("verbose = %d\n", verbose);
}
```

5.6 Startbedingungen

Nun möchten wir eine **Option mit Parameter** dazufügen. Sie soll einen **Dateinamen** angeben:

cmdline2.1:

```
%{
#include <stdio.h>
#include <string.h>
int verbose = 0;
char *progName = NULL;
char *filename = "-";
}%
%option noyywrap
%x FNAME
%%
^-h"          |
^-?"          |
^"--help"      { printf("usage is: %s [--help | -h | -?] ", progName);
                printf("[--verbose | -v ...] ");
                printf("[--file | -f) filename]\n");
                exit(0);
                }
^-v"          |
^"--verbose"   { verbose++; }
^-f"          |
^"--file"      { BEGIN FNAME;
                filename = "";
                }
.*            { printf("unknown option '%s'!\n", yytext);
                exit(1);
                }
<FNAME>.+     { filename = strdup(yytext); BEGIN INITIAL; }
%%
int main(int argc, char *argv[]) {
    progName = *argv;
    while(++argv,--argc) {
        yy_scan_string(*argv);
        yylex();
    }
    if(!filename || !*filename) {
        printf("No filename given with option --file!\n");
        exit(1);
    }
    printf("Now starting to frobnicate with:\n");
    printf("verbose = %d\n", verbose);
    printf("file = '%s'\n", filename);
}
```

Nach der Option **-f** muß **als nächstes** der **Dateiname** kommen. Es kann **keine weitere Option** dort stehen. Die **normalen Regeln** müssen **abgeschaltet** werden, und eine **besondere Regel** für den Dateinamen muß **eingeschaltet** werden.

Dies erreichen wir mit einer **Startbedingung**. Eine Startbedingung benutzt eine **zusätzliche Zustandsvariable** des Scanners.

Oben definieren wir die Startbedingung mit **%x**.

Nach Ausführung des **Makros BEGIN FNAME** sind **nur noch Regeln aktiv**, vor denen in **spitzen Klammern** der Name der Startbedingung steht.

Mit **BEGIN INITIAL** kommt man in den **initialen Zustand** zurück.

Man kann mit **%x** **beliebig viele** solche **Zustände** definieren. **Intern** handelt es sich um eine **Integer-Variable**, die gesetzt und abgeprüft wird. Z.B. ist **INITIAL** **immer als 0** definiert.

Innerhalb der Regeln kann man den **Zustand** über das **Makro YY_START** **lesen** und z.B. einer Integer-Variablen zuweisen. (Bei **Flex** gibt es auch ein Makro **YYSTATE** mit der gleichen Wirkung.)

Man beachte im Beispiel, wie in **main()** **geprüft** wird, **ob** nach **-f** wirklich noch ein **Dateiname angegeben** wurde. (**YYSTATE** dürfen wir **außerhalb** der Regeln leider **nicht** verwenden.)

Soll eine **Regel in mehr als einem Zustand aktiv** sein, kann man die Startbedingungen mit **Komma** getrennt in den spitzen Klammern aufführen.

Gibt man einen **Stern *** als Startbedingung an, gilt die Regel in **allen Zuständen**.

5.7 Online-Aufgabe: C-Quellcode-Zähler

Schreibe ein Programm, daß eine **C-Quell-Datei** einliest und **zählt**, wieviele Zeilen es jeweils hat mit:

- C-Code
- nur Kommentar
- nur Whitespace

Hinweis zum **Algorithmus**: Lese die Eingabe in kleinen Stücken ein. Benutze Flags, um dir zu merken, ob du in dieser Zeile schon C-Code bzw. einen Kommentar gesehen hast. Wenn Du ein Newline siehst, dann zähle die C-Code-, Kommentar- und Whitespace-Zeilenzähler entsprechend weiter und setze die Flags zurück.

Hinweis zum Programmaufbau: Die Verwendung von **Startbedingungen** ist **hilfreich**.

Eine **Beispiel-Eingabe-Datei**:

cexample.c:

```
/*
Dies ist eine Test-Datei fuer den C-Code-Zaehler.
*/

int main() {
    int idx; /* Index */
    int count; /* Dies ist
                ein Zaehler */

    /* Buchstabe:
    */ char letter;
}
```

Zusatzaufgabe: Es soll auch der Fall richtig gehandhabt werden, daß ein **String** einen **Kommentar-Anfang oder -Ende enthält**. Strings dürfen auch über **mehrere Zeilen** gehen, und sie dürfen **"** enthalten.

Eine **Beispiel-Eingabe-Datei**:

cexample2.c:

```
/*
Dies ist eine Test-Datei fuer den erweiterten C-Code-Zaehler.
*/

int main() {
    int idx; /* Index */
    int count; /* Dies ist
                ein Zaehler */

    /* Buchstaben:
    */ char *letters = "Hello\
\
world\n";
}
```

5.8 Mehrere Eingabequellen nacheinander

Beim Scannen der **Kommandozeile** auf Seite 157 haben wir **alle Parameter separat** durch Aufrufe von `yylex()` gelesen. **Nachteil** war, daß `yylex()` immer mit **return** **zurückkehrte**.

Es geht auch **anders**: Wenn `yylex()` **am Ende** eines Strings bzw. einer Datei angekommen ist, ruft es die **Funktion** `yywrap()` auf. Falls `yywrap()` den Wert **false** (**d.h.** 0)

zurückliefert, dann **macht** `yylex()` **einfach weiter** und nimmt an, daß die **Eingabe** nun auf **neue Daten** zeigt.

Wenn man **keine eigene Funktion** `yywrap()` schreibt, dann muß man die **Option** `%noyywrap` angeben, wie wir es bisher immer getan haben.

Beispiel:

cmdline-wrap.1:

```
%{
#include <stdio.h>
int verbose = 0;
char *progName = NULL;
char **currArg = NULL;
int currArgNum = 0;
%}
%%
~"-h"          |
~"-?"          |
~"--help"      { printf("usage is: %s [--help | -h | -?] ", progName);
                  printf("[--verbose | -v ...]\n");
                  exit(0);
                }
~"-v"          |
~"--verbose"   { verbose++; }
.*             { printf("unknown option '%s'!\n", yytext);
                  exit(1);
                }
%%

int yywrap() {
    if(--currArgNum <= 0)
        return 1;
    yy_scan_string(++currArg);
    return 0;
}

int main(int argc, char *argv[]) {
    progName = *argv;
    currArg = ++argv;
    currArgNum = --argc;
    if(currArgNum > 0) {
        yy_scan_string(*currArg);
        yylex(); /* Nur EIN Aufruf fuer alle Parameter */
    }
    printf("Now starting to frobnicate with:\n");
    printf("verbose = %d\n", verbose);
}
```

Das gleiche geht natürlich **auch** für **mehrere Dateien** nacheinander. Dann muß man

yyin **neu zuweisen**. So eine Zuweisung haben wir **oben** schon **gesehen**, als wir die **Pascal-artige** Sprache aus einer (einzigen) **Datei gelesen** haben.

5.9 Mehrere Eingabequellen abwechselnd

Wenn man eine Eingabesprache scannt, die „**include**“-**Anweisungen** enthält, muß man **mitten im Datenstrom** auf eine andere Eingabe **umschalten** und später **zurückschalten**. Hier kann man z.B. yyin **nicht** einfach **neu zuweisen**, weil der Lexer oft ein großes Stück **vorausliest und puffert**. Erst viele Zeilen **später** würde die Zuweisung **wirksam werden**.

Daher kann man **mehrere Eingabepuffer** haben und zwischen ihnen **hin- und herschalten**.

Beispiel:

include.1:

```
%{
#include <stdio.h>
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_pos = 0;
%}
%option main
%x INCL
%%
~"#include"[ \t]+ BEGIN(INCL);
.|\\n ECHO;
<INCL>.*\\n { if(include_stack_pos >= MAX_INCLUDE_DEPTH) {
                fprintf(stderr,"Includes nested too deeply!\\n");
                exit(1);
            }
            include_stack[include_stack_pos++] =
                YY_CURRENT_BUFFER;
            yytext[strlen(yytext)-1] = '\\0'; /* remove '\\n' */
            if(!(yyin = fopen(yytext, "r"))) {
                fprintf(stderr,
                    "Cannot open include file '%s'!\\n", yytext);
                exit(1);
            }
            yy_switch_to_buffer(
                yy_create_buffer(yyin, YY_BUF_SIZE));
            BEGIN(INITIAL);
        }
<<EOF>> { if(--include_stack_pos < 0)
            yyterminate();
        else {
            yy_delete_buffer(YY_CURRENT_BUFFER);
            yy_switch_to_buffer(
                include_stack[include_stack_pos]);
        }
    }
```

Der Zustand **INCL** dient dazu, den Dateinamen einzulesen.

Dabei ist wichtig, das **Newline** zu lesen, bevor das Include ausgeführt wird. Allerdings muß es vom Dateinamen abgetrennt werden.

Das Makro **ECHO** druckt das aktuelle Token aus **yytext** einfach zur Ausgabedatei **yyout**

aus.

Analog zur Variablen `yyin`, die auf die Eingabedatei zeigt, zeigt die **Variable** `yyout` auf die **Ausgabedatei** (Default `stdout`). Sie kann ebenfalls **neu zugewiesen** werden.

YY_BUFFER_STATE ist ein opaquer Datentyp, der genau einen Zeiger auf einen Eingabepuffer faßt.

Die Variable **YY_CURRENT_BUFFER** enthält den Zeiger auf den aktuellen Eingabepuffer.

Mit `yy_switch_to_buffer(...)` kann man auf einen anderen Eingabepuffer umschalten. Man kann es auch sonst anstelle eine Zuweisung auf `yyin` verwenden.

Mit `yy_create_buffer(yyin, YY_BUF_SIZE)` kann man einen neuen Eingabepuffer erzeugen.

Mit `yy_delete_buffer(...)` kann man einen Eingabepuffer wieder freigeben.

`yyterminate()` führt im wesentlichen ein `return 0;` aus.

Außerdem gibt es noch `yy_flush_buffer(YY_BUFFER_STATE buffer)`, um einen Eingabepuffer zu leeren und wie neu zu machen.

5.10 Aufruf- und Datei-Optionen von Flex

Man kann Flex eine Reihe von **Kommandozeilen-Optionen** mitgeben. Alternativ kann man fast alle auch **in der Datei** per `%o...` angeben.

Die wichtigsten Aufruf- und Datei-Optionen:

Aufruf- und Datei-Optionen von Flex		
Datei	Aufruf	Bedeutung
%option main		Generiere eine <code>main()</code> -Funktion. Impliziert <code>noyywrap</code> .
%option noyywrap		Generiere eine <code>yywrap()</code> -Funktion.
%option stdout	-t	Der generierte Scanner geht nach <code>stdout</code> statt nach <code>lex.yy.c</code> .
%option case-insensitive	-i	Beim Matchen werden Groß- und Kleinbuchstaben nicht unterschieden.
%option nodefault	-s	Generiere keine Default-Regel. Falls keine Regel paßt, gibt es eine Fehlermeldung.
	--help	Gibt die möglichen Aufrufoptionen aus.
	--version	Gibt die Versionsnummer aus.
%option yylineno		Die Variable <code>yylineno</code> enthält immer die aktuelle Zeilennummer.

Kurs Unix-Tools, WS 01/02, Jan Brederke 41

Weitere Aufruf- und Datei-Optionen von Flex		
Datei	Aufruf	Bedeutung
%option lex-compat	-l	Maximale Kompatibilität mit originalem lex von AT&T.
%option c++	++	Generiere eine C++-Scanner-Klasse.
%option debug	-d	Der generierte Scanner macht Debug-Ausgaben, wenn die Variable yy_flex_debug ungleich Null.
%option perf-report	-p	Gibt Performance-Report auf stdout aus, der „teure“ benutzte Features auflistet.
%option prefix="XYZ"	-PXYZ	Ersetzt in allen Variablen- und Funktionsnamen das Präfix yy durch das angegebene Präfix.
%option stack		Erlaubt die Benutzung von Stacks von Startbedingungen.
...

Kurs Unix-Tools, WS 01/02, Jan Bredeke

42

5.11 Weitere Features

Flex kennt eine ganze Reihe von **weiteren Features**. Einige davon werden hier noch kurz vorgestellt. Ansonsten siehe das **Manual**.

Man kann beim **Matchen** von Mustern auf die **nächsten Zeichen vorausschauen**, ohne daß sie **zum gematchten Text gehören** werden:

```
"include "/\"[^\n]*\""
```

matcht nur dann den Text „include“, wenn auch ein Dateiname folgt.

Eine bereits **bekannte** Art von **Zeichen-Vorschau** ist der **reguläre Ausdruck** \$, der auf ein folgendes Zeilenende prüft.

Eine **weitere** Art von Vorschau erreicht man mit der **Funktion** `yyless()`. Mit ihr kann man innerhalb einer Aktion den **letzten Teil der gelesenen Buchstaben wieder zurück** in die Eingabe schieben. Der **Unterschied** zur Vorschau mit / ist, daß `yytext` zwischenzeitlich den gesamten Text enthält, und daß `yylen` entsprechend die gesamte Länge enthält. Der **Parameter** von `yyless()` ist die Anzahl der *nicht* mehr zurückzuschiebenden Zeichen.

Das **Makro** `REJECT` dient ähnlichen Zwecken. Es **bricht** die augenblickliche **Aktion ab** und **springt zur am nächsten besten passenden Regel**. Ein **Beispiel** wäre ein **Wort-Zähl-Programm**, daß zusätzlich bei einem bestimmten **Schlüsselwort** eine Funktion ausführt, aber danach trotzdem noch das Schlüsselwort als Wort mitzählt.

In regulären Ausdrücken kann man wie bekannt **Buchstabenmengen** verwenden. **Außer Buchstabenbereichen** kann man **auch** die folgenden Ausdrücke verwenden:

```
[ :alnum:] [ :alpha:] [ :digit:]  
[ :blank:] [ :space:]  
[ :lower:] [ :upper:]  
...
```

Verwendungsbeispiel:

```
[[:alpha:]0-9]+
```

(ist äquivalent zu [[:alnum:]]+)

Die Mengen werden **definiert über** die Funktionen der **C-Bibliothek** `isalnum()`, ...

Diese Buchstabenklassen sind eine **Flex-Erweiterung** von Lex.

Normalerweise haben wir **Startbedingungen** mit

```
%x F00
```

definiert. Wenn wir mit `BEGIN(F00)` in diesen Startzustand gewechselt sind, dann waren *nur* noch exklusiv die Regeln **dieses Startzustandes** aktiv.

Wenn man eine Startbedingung mit

```
%s F00
```

definiert, dann haben wir eine **inklusive Startbedingung**, und es sind im entsprechenden Zustand außerdem immer noch alle Regeln aktiv, die **keinen Startbedingung** haben.

Man kann damit **gut Default-Regeln** schreiben, aber man muß gut **aufpassen**, daß nicht zuviele Regeln aktiv bleiben. **Default-Regeln** kann man **auch mit der Startbedingung** `<*>` schreiben. Leider kennt die **originale Version von Lex** nur **inklusive Startbedingungen**.

Wenn man einen ganzen Satz von **Regeln** hat, die **alle zur selben Startbedingung** gehören, dann muß man die Startbedingung **nicht vor jede** der Regeln schreiben.

```
<STRING>foo      ...  
<STRING>bar      ...
```

Stattdessen kann man auch einen **Startbedingungsbereich** definieren:

```
<STRING>{  
foo      ...  
bar      ...  
}
```

Manchmal **reicht der endliche Automat des Startzustandes nicht** aus. Dann kann man bei Flex einen **Stack von Startbedingungen** benutzen. Dafür gibt es

```
void yy_push_state(int new_state)  
void yy_pop_state()  
int yy_top_state()
```

Hiermit kann man eine **Sonderbehandlung innerhalb einer Sonderbehandlung** machen. Man hat die **Leistungsfähigkeit von Unterprogrammaufrufen mit Return**, im **Gegensatz** zu einfachen **Gotos**.

Dies ist allerdings **Flex-spezifisch**.

5.12 Flex und andere Lexer

Flex ist fast vollständig **POSIX-kompatibel**.

Einige **kleine technische Inkompatibilitäten** mit dem **originalen Lex** von AT&T.

AT&Ts Lex kennt keine **exklusiven Startbedingungen** mit **%x**, obwohl sie im POSIX-Standard stehen.

Flex schließt **Definitionen von regulären Ausdrücken** in **Klammern** ein, damit ein nachfolgender ***** oder **+** auf die ganze Definition wirkt. Das vermeidet Überraschungen. AT&Ts Lex tut das nicht. Mit der Option **-1** kann man bei Flex das **Lex-Verhalten einschalten**.

Es gibt ein paar ähnliche **Unterschiede bei regulären Ausdrücken**, wo Flex eine sinnvollere bzw. POSIX-kompatible Alternative wählt.

Flex hat eine ganze Reihe **eigener Erweiterungen**, die es nur dort gibt:

- Scanner optional in C++
- Alle **%option**
- Fast alle obigen Aufruf-Optionen.
- Buchstabenklassen
- **Lesen aus Strings** mit `yy_scan_string()` usw.
(Andere Lexer haben jeweils **andere Wege** dafür.)
- **Verschachtelte Includes** mit `yy_switch_to_buffer()`.
(Andere Lexer haben jeweils **andere Wege** dafür.)
- **<<EOF>>** in Regeln.
- Bereiche von Startbedingungen
- Stacks von Startbedingungen
- ...

6 Syntaktische Analyse mit yacc

6.1 Einführung

6.1.1 Kontextfreie Grammatiken und Backus-Naur-Form

Beispiel: Der Kadonische Leuchtturm. Die Einwohner der Insel Kadonien lieben die Abwechslung, und sie streichen daher ihren Leuchtturm alle zwei Wochen neu an. Dies ist natürlich sehr verwirrend für die Seefahrer. Aber wir können den Seefahrern helfen: Jedes Mal geht eine zufällig sich ergebende Gruppe von kadonischen Frauen und Männern los, um den Turm neu zu streichen. Jeder Helfer streicht dabei drei Meter zusammenhängende Turmhöhe, und zwar in drei Ringen. Ein Mann streicht stets rot-weiß-rot, eine Frau stets weiß-rot-weiß. Der Turm ist zwölf Meter hoch. Damit folgen die Kadonier immer diesem Schema:

```
KADONISCHERTURM ::= HELFER HELFER HELFER HELFER
HELPER ::= MANN
          | FRAU
MANN ::= ROT WEISS ROT
FRAU ::= WEISS ROT WEISS
ROT ::= 'rot'
WEISS ::= 'weiss'
```

Die Leuchttürme auf den Nachbarinseln sind zwar auch immer in ein Meter breiten Ringen bemalt, und sie sind zum Teil auch zwölf Meter hoch. Aber die Nachbarn mögen die kadonische Leuchtturmmoden nicht und haben streng darauf geachtet, niemals das Kadonische Schema zu verwenden. Wir schreiben daher ein Programm, das die Insel Kadonien sicher anhand seines Leuchtturms erkennt. Das Programm soll eine Zeile vom benutzenden Kapitän lesen und dann ausgeben, ob er vor Kadonien liegt oder nicht.

Das obige ist eine **Grammatik in Backus-Naur-Form (BNF)**.

Terminale Symbole, die wortwörtlich in der Eingabe vorkommen müssen, stehen in Anführungszeichen.

Der **senkrechte Strich** ist wie immer die Alternative.

Außer den verwendeten Möglichkeiten kann man auch Symbole oder Symbolfolgen in **eckige Klammern** einschließen, wenn sie optional sind. Teile in **geschweiften Klammern** dürfen beliebig oft wiederholt werden, einschließlich nullmal.

Die **Erkennung der Literale** „rot“ und „weiss“ überlassen wir einem **Lexer**.

Die Erkennung des **Turm-Musters** wäre **mit Lex-Startzuständen** sehr **mühsam**.

Das Programm **Yacc** dagegen **kann aus** einer BNF-artigen **Grammatik** einen **Syntax-**

checker generieren.

Das Yacc-Programm zur obigen Grammatik:

leuchtturm.y:

```
%{
#include <stdio.h>
%}
%token ROT WEISS SCHREIBFEHLER
%%
kadonischerturm:  helfer helfer helfer helfer
                  ;
helfer:           mann
                  | frau
                  ;
mann:             ROT WEISS ROT
                  ;
frau:             WEISS ROT WEISS
                  ;
%%
int yyerror(char *msg) {
    return 0;
}
int main() {
    printf("Wie sieht der Leuchtturm aus, Sir? ");
    if(yparse() == 0)
        printf("Wir liegen vor der Insel Kadonien, Sir!\n");
    else
        printf("Wir liegen *nicht* vor der Insel Kadonien, Sir!\n");
    return 0;
}
```

Die **Grundstruktur** eines Yacc-Programms ist ähnlich wie die eines **Lex-Programms**:

- **Deklarationen**
- **Regeln**
- **Unterprogramme** (optional)

Im **Regel-Teil** finden wir die **BNF-Grammatik** wieder, nur mit etwas **anderer Syntax**.

Die **Startregel** ist die **erste Regel**.

Das **Ende der Eingabe** muß immer am Ende der Abarbeitung der Startregel kommen.

Die **%token-Anweisung** sagt, daß der **Lexer** die **Token ROT und WEISS** liefern kann. Außerdem kann er auch das Token **SCHREIBFEHLER** liefern, falls ein anderes

Wort eingegeben wird.

Man kann wie bei **Lex** am **Anfang literalen C-Code** angeben, hier eine Include-Anweisung.

Am Ende können **Unterprogramme** angegeben werden. Hier müssen wir unser `main()` hinschreiben. Die Funktion `main()` ruft die **Funktion** `yyparse()` auf. `yyparse()` **gibt 0 zurück**, wenn alles **in Ordnung** war, und einen Wert **ungleich 0**, falls die **Eingabe nicht zur Grammatik paßt**.

Außerdem müssen wir eine **Fehlermeldefunktion** `yyerror()` schreiben. Diese **hier tut gar nichts**, weil wir die Auswertung später in `main()` machen.

Ein **Lex-Scanner** kann uns die Eingabe **in Token zerlegen** und die Worte „rot“ und „weiss“ erkennen:

leuchtturm.l:

```
%{
#include "leuchtturm.tab.h"
%}
%option noyywrap
%option nodefault
%%
"rot"          { return(ROT); }
"weiss"        { return(WEISS); }
\n             { return 0; /* Nur eine Zeile lesen. */ }
[ \t]          /* Ignoriere sonstigen Whitespace. */
               { return(SCHREIBFEHLER); }
```

Am **ersten Zeilenende** gibt der Scanner 0 zurück als Zeichen, daß wir am Ende sind.

Die **Option** `noyywrap` gibt an, daß es keine weitere Eingabedateien gibt.

Die **Option** `nodefault` unterdrückt die Default-Regel für Zeichen, die nirgends passen. Stattdessen haben wir eine eigene Default-Regel.

Die **Rückgabewert-Konstanten** sind in der Datei `leuchtturm.tab.h` definiert. Sie wird **von Bison** aus der `%token-Zeile` generiert.

Bison ist die **Gnu-Version** von Yacc.

Eine solche Kombination von Bison-Parser und Flex-Scanner läßt sich mit dem folgenden **allgemeinen Makefile** übersetzen:

Makefile:

```
%c: %.y # loesche alte implizite Regel
%c: %.l # loesche alte implizite Regel
%.tab.c %.tab.h: %.y
        bison --defines $<
%c: %.l %.tab.h
        flex -t $< > $@
%: %.tab.o %.o
        cc -o $@ $^
```

Zuerst müssen wir **zwei eingebaute implizite Regeln löschen**, weil sie im Weg sind.

Dann sagen wir, wie ein **Bison-Aufruf** aus einer *.y-Datei die beiden Dateien *.tab.c mit dem Parser und *.tab.h mit den Token-Definitionen generiert.

Die **Flex-Regel** ist fast die Default-Regel, nur kommt die Abhängigkeit von der Include-Datei mit den Token-Definitionen dazu.

Schließlich muß das **ausführbare Programm** aus dem Parser und dem Scanner zusammengebunden werden.

►► make leuchtturm

►► ./leuchtturm

6.2 Online-Aufgabe: Klammerausdrücke

Die folgende Grammatik beschreibt korrekte Klammerausdrücke:

```
ka ::=
    | ka ka_einfach
ka_einfach ::= buchstabe
                | '(' ka ')'
                | '[' ka ']'
                | '{' ka '}'
                | '<' ka '>'
```

Schreibe einen Parser, der korrekte Klammerausdrücke von der Standard-Eingabe liest und erkennt.

Beachte, daß die erste Alternative von **ka** die ganz **leere Eingabe** ist.

Interessant ist hier die **starke Verwendung von Rekursion**, die **typisch** für fast alle Grammatiken ist.

Die **genaue Definition von buchstabe** überlassen wir dem Scanner.

Der folgende **Flex-Lexer** kann das Scannen übernehmen:

ka.l:

```
%{
#include "ka.tab.h"
%}
%option noyywrap
%option nodefault
%%
"("      |
")"      |
"["      |
"]"      |
"{"      |
"}"      |
"<"      |
">"      { return *yytext; }
|\\n    { return BUCHSTABE; }
```

Beachte, daß **alle 255 ASCII-Zeichen** größer als 0 **bereits implizit als Token** für Yacc **definiert** sind. Daher kann Flex die **Klammern direkt als Buchstabe** zurückgeben.

6.3 Semantik zur Syntax

Bei vielen Problemen wollen wir **nicht nur** prüfen, **ob** eine **Eingabe** zur Grammatik **paßt**.

Der **Lexer** liefert nicht nur zu jedem Token den **Typ** per Return-Wert mit, sondern er kann in der globalen Variablen `yylval` **auch** zusätzlich einen **Wert** liefern.

Entsprechend kann der **Parser** zu jedem **nicht-terminalen Symbol** auf der **linken** Seite einer Regel einen **neuen Wert** aus den Werten der Symbole auf der **rechten** Seite liefern.

Hierfür kann man in die Regeln **Aktionen** einfügen, die diesen Wert berechnen.

Weiterhin darf man **auch beliebige andere Anweisungen** in diesen Aktionen ausführen lassen.

Damit wird es möglich, einen **Parser für eine Programmiersprache** zu schreiben, der als Ergebnis einen **Syntaxbaum** des Eingabeprogramms liefert, der dann **an einen Code-Generator übergeben** werden kann.

Oder man kann einen **Taschenrechner** schreiben, der **Punktrechnung-vor-Strichrechnung** beherrscht und nebenbei auch die **Ausdrücke berechnet**:

Zunächst brauchen wir einen **Lexer**, für den wir unseren alten einfachen Taschenrechner aus Abschnitt 5.3 abwandeln:

calc.l:

```
%{
#include <math.h>
#include "calc.tab.h"
%}
%option noyywrap
DIGIT  [0-9]
%%
{DIGIT}+("."{DIGIT}*)?  {
    yylval.zahl = atof(yytext);
    return NUMBER;
}
"="                |
"("                |
")"                |
"+"                |
"_"                |
"*"                |
"/"                { return yytext[0]; }
[[:space:]]        { /* ignoriere Whitespace */ }
{ return ILLEGAL_CHAR; }
```

Dieser Lexer **kehrt** jetzt wiederum mit **return zurück**, sobald er eine Token erkannt hat.

Neu sind die Klammern und „=“, dafür haben wir **CLEAR** zur Vereinfachung **weggelassen**.

Den Code, der die **Aktionen** durchgeführt hat, haben wir **gelöscht**.

Stattdessen wird der **Typ der Eingabe** als **Returncode** übergeben. Außerdem wird eine **eingeebene Zahl** in der Variablen `yylval.zahl` an den Parser übergeben.

Die **Variable** `yylval` wird **von Yacc** zur Verfügung gestellt, um Werte vom Lexer an den Parser zu übergeben. Da ein Wert je nach Token einen **verschiedenen Datentyp** haben kann, ist `yylval` eine **C-Union** aller dieser Datentypen. Daher müssen wir immer **auf die passende Komponente zugreifen**, hier `zahl`. Die Komponenten sind im **%union-Konstrukt** am Anfang der Yacc-Datei deklariert:

calc.y:

```

%{
#include <stdio.h>
%}
%union {
    double zahl;
}
%token ILLEGAL_CHAR
%token <zahl> NUMBER
%left '-' '+'
%left '*' '/'
%type <zahl> term
%%
eingabe:          /* empty */
                | eingabe berechnung
                ;
berechnung:      term '=' { printf("Ergebnis: %g\n", $1); }
                ;
term:            NUMBER
                | term '+' term { $$ = $1 + $3; }
                | term '-' term { $$ = $1 - $3; }
                | term '*' term { $$ = $1 * $3; }
                | term '/' term { $$ = $1 / $3; }
                | '(' term ')' { $$ = $2; }
                ;
%%
int yyerror(char *msg) {
    printf("\nEingabefehler: %s\n", msg);
    return 0;
}
int main() {
    return yyparse();
}

```

Hier haben wir **nur einen Datentyp** und daher nur eine Union-Komponente.

Wir haben wieder eine **main()-Funktion**. Und unsere **yyerror()-Funktion** macht nun eine **Ausgabe**, um den Benutzer über die Art des Fehlers zu informieren.

Mit den **%token**-Anweisungen definieren wir wieder die **erlaubten Token**. Die Token-Definitionen und die Deklaration von `yylval` gehen wieder in die **Datei `calc.tab.h`**.

In der **Grammatik** gibt es etwas neues: **Aktionen** in geschweiften Klammern. Am Ende einer **berechnung** wird das **Ergebnis ausgegeben**.

Die **Variable \$1** enthält den **Wert des ersten Tokens**, hier von **term**.

In der Regel für `term` sehen wir **weitere solche Variablen**, die auf das **zweite** und **dritte** Token zugreifen.

Die Variable `$$` nimmt den **Wert des gesamten Ausdrucks** auf, so daß der Wert von `term` in weiteren Regeln verwendet werden kann.

Mit der `%type`-Anweisung oben haben wir festgelegt, daß `term` eine `zahl` ist, also vom Typ `double`.

Wenn der **Lexer** eine `NUMBER` zurückliefert, und wenn `yyval.zahl` die **Zahl** enthält, dann kann man also in Yacc über `$1` auf diese Zahl **zugreifen**. Der **Typ** von `NUMBER` ist oben in der **erweiterten Token-Anweisung** deklariert worden.

Wenn keine Aktion angegeben ist, dann ist die **Default-Aktion**, `$1` auf `$$` zuzuweisen. Dies ist hier bei `NUMBER` geschehen.

6.4 Operator-Assoziativität

```
▶▶ make calc
```

```
▶▶ ./calc
```

Was passiert bei

```
▶▶ 9-2=
```

```
▶▶ 9-2-1=
```

Es gibt **zwei Möglichkeiten**, den Ausdruck **zu parsen**, entweder $(9-2)-1$ oder $9-(2-1)$. Die **Grammatik ist mehrdeutig**. Durch die `%left`-Anweisung am Anfang sagen wir Yacc, daß wir nur die erste Möglichkeit haben wollen.

Eine `%left`-Anweisung ist **wie** eine `%token-Anweisung`, nur daß der Operator immer **linksassoziativ** ist.

Entsprechend gibt es auch `%right`.

Und es gibt `%nonassoc` für Operatoren, die **nicht mehrfach verwendet** werden dürfen, wie z.B. `if(a<x<b)` in vielen Sprachen **nicht erlaubt** ist.

6.5 Operator-Präzedenz

Was passiert bei

```
▶▶ 2+3*4=
```

```
▶▶ 3*4+2=
```

Woher „weiß“ der Rechner, daß **Punktrechnung vor Strichrechnung** geht? Die **Grammatik** ist wiederum **mehrdeutig**.

Die **Token** haben eine **Präzedenz**. Um so später sie deklariert werden, um so höher

ist die Präzedenz. Hier werden * und / nach + und - deklariert. Token auf der **gleichen Zeile** haben **gleiche Präzedenz**.

Wenn die **Präzedenz gleich** ist, dann kommt wieder die **Assoziativität** zum Tragen.

6.6 Fehlerbehandlung

Was passiert bei Eingabefehlern?

```
➤➤ 2+=
```

Diese Fehlermeldung ist **nicht** sehr **aussagekräftig**.

Man kann **mehr Informationen** ausgeben lassen, wenn man das **Makro** `YYERROR_VERBOSE` im **C-Deklarationsteil** auf einen beliebigen Wert definiert.

```
➤➤ Machen.
```

```
➤➤ make calc
```

```
➤➤ ./calc
```

```
➤➤ 2+=
```

Wenn man die **Token einzeln** bearbeiten läßt, sieht man auch genau, wo es schiefgeht:

```
➤➤ ./calc
```

```
➤➤ 2
```

```
➤➤ +
```

```
➤➤ =
```

So sieht man, **welche Token erwartet** werden.

Man kann sich auch die **Zeilennummer** in der Eingabe ausgeben lassen, aber das macht bei **interaktiven Programmen wenig Sinn**.

Oft will man den **Fehler auch im Parser behandeln** und dann die **Bearbeitung fortsetzen**, zumindest vorläufig.

Hierfür gibt es das besondere Token `error`:

```
calc-err.y:
```

```

%{
#include <stdio.h>
#define YYERROR_VERBOSE
%}
%union {
    double zahl;
}
%token ILLEGAL_CHAR
%token <zahl> NUMBER
%left '-' '+'
%left '*' '/'
%type <zahl> term
%%
eingabe:          /* empty */
                | eingabe berechnung
                ;
berechnung:       term '=' { printf("Ergebnis: %g\n", $1); }
                | error '=' { yyerrok; printf("Neue Eingabe:\n"); }
                ;
term:             NUMBER
                | term '+' term { $$ = $1 + $3; }
                | term '-' term { $$ = $1 - $3; }
                | term '*' term { $$ = $1 * $3; }
                | term '/' term { $$ = $1 / $3; }
                | '(' term ')' { $$ = $2; }
                ;
%%
int yyerror(char *msg) {
    printf("\nEingabefehler: %s\n", msg);
    return 0;
}
int main() {
    return yyparse();
}

```

Wenn ein **Token gefunden** wird, auf das **keine Regel paßt**, ist das ein Fehler. Dann wird **versucht**, das Token **error zu matchen**. Wenn das in der **aktuellen Regel nicht** geht, wird diese **abgebrochen**, und das gleiche in der **nächst höheren halbfertigen Regel** versucht.

Wenn eine **error-Regel gefunden** wurde, dann **müssen drei Token passen**, damit **wieder normal** bearbeitet wird.

In unserem Falle sind wir **bei einem = sicher**, daß wir **wieder richtig** sind. Daher wird **yyerrok aufgerufen**, das den Parser **sofort** wieder in den **Normalzustand** bringt.

```
▶▶ make calc-err
```

```
▶▶ ./calc-err
```

6.7 Aufruf-Optionen von Bison

Aufruf-Optionen von Bison		
Option	Kurzform	Bedeutung
--defines	-d	Schreibe die .tab.h-Datei für Lex.
--verbose	-v	Schreibe eine .output-Datei mit allen Parser-Zuständen und allen Regel-Konflikten
--debug	-t	Definiere das Makro YYDEBUG, so daß Debugging möglich wird.
--name-prefix= <i>prefix</i>	-p <i>prefix</i>	Ersetze den Präfix yy in allen Namen durch den angegebenen Präfix. (Erlaubt mehrere Parser.)
--help	-h	Gibt alle Kommandozeilen-Optionen aus.
...

Kurs Unix-Tools, WS 01/02, Jan Brederke 43

Wird im **C-Deklarationsteil** das Makro **YYDEBUG** definiert, und wird die **Variable yydebug** auf einen Wert ungleich 0 gesetzt, dann macht der Parser sehr viele **Debugger-Ausgaben**. Man kann genau sehen, **wann welche Regel** angewandt wird. Die **Ausgabe** ist aber *sehr umfangreich*.

6.8 Weitere Features

Man kann auch **Aktionen in der Mitte von Regeln** haben.

Die **Variablen \$1** für die **semantischen Werte** usw. können auch **variable Typen** haben.

Man kann die **Präzedenz von Operatoren für einzelne Regeln** ändern.

Man kann auch explizit angeben, welches die **Start-Regel** ist.

Mit den **Makros YYABORT** und **YYACCEPT** kann man das **Parsen sofort beenden**, entweder als Fehler oder als Erfolg.

Mit dem **Makro YYERROR** kann man **aus einer Aktion** heraus die **Fehlerbehandlung** anstoßen, für kontextsensitive Fehler.

Den **Status der Fehlerbehandlung** kann man mit **YYRECOVERING** abfragen.

Wenn bei der **Fehlerbehandlung** das **Look-Ahead-Token** im Weg ist, kann man es mit **yyclearin** löschen lassen.

6.9 Bison und andere Versionen von Yacc

Bison **kann** praktisch **alles**, was AT&s Yacc auch kann.

Zusätzliche Features von Bison:

- **Bessere Zeilennummer-Ausgabe** bei Fehlern.
- Einen **anderen Dateinamen** für die Ausgabedatei. (Standard: *Immer* `y.tab.c`)
- Eine Option, das `yy` in **allen Namen umzubenennen**, um mehrere Parser in einem Programm haben zu können.
- Einige kleine **technische** Zusätze.

A Lösungen der Hausaufgaben und Online-Aufgaben

Kapitel 2:

Der batchorientierte Zeilen-Editor sed

Abschnitt 2.5.1 auf Seite 8:

Erkennen von Email-Adressen

```
sed -n -e 's/^.*\(<\(jan\.\\)\?brederek\(e\\)\?@'\
'\(web\\|tzi\\|\(saturn\.\\|gemini\.\\)\?informatik\.uni-bremen\\)\.de\>\\)' \
'.*/\1/p'
```

Anmerkung: Dieser Ausdruck paßt auf einige Adressen mehr als die angegebenen. Für die praktische Nutzung ist das aber ausreichend.

Beachte: der \|-Operator nimmt nach links und rechts jeweils allen Text so weit er kann.

Anmerkung: Wenn am Anfang kein \sim vor dem $.*$ steht, macht das die Ausführung *erheblich* langsamer, auch wenn das Ergebnis das gleiche ist.

Abschnitt 2.5.2 auf Seite 9:

Extraktion von include-Dateinamen aus einer LaTeX-Quelle

```
sed -n -e 's/^[^%]*\\verbatiminput{\([^%]+\)}.*$/\1/p' \
< ../skriptnotizen.tex
```

Abschnitt 2.5.3 auf Seite 9:

Extraktion aller derzeit aktiven Benutzer aus „w“-Ausgabe

```
sed -n -e 's/^[^ ]+\) \+pts.\{17\} \{6\}.*$/\1/p' \
< w-output.txt
```

Anmerkung: Man kann auch ohne den Zähl-Wiederholungsoperator $\{\}$ auskommen.

Abschnitt 2.6.1 auf Seite 10:

Schützen von Sonderzeichen in der Vacation-Text-Ersetzung

```
sed -e 's!\([/\\]\)!\\1!g'
```

Abschnitt 2.7.1 auf Seite 10:
Liste aller Dateien unter CVS

```
find . -name Entries -exec sed -n -e 's!~/\([^/]*\).*!\1!p' '{}' ';' 
```

Anmerkung: Wir ignorieren die Directory-Einträge in `Entries`, da `find` sich um sie kümmert.

Kapitel 3: Die Skriptsprache perl

Abschnitt 3.5.1 auf Seite 22: Interaktive Multiplikation

mult-interaktiv.pl:

```
#!/usr/bin/perl -w
use diagnostics;
print "Erste Zahl: ";
chomp($zahl1 = <STDIN>);
print "Zweite Zahl: ";
chomp($zahl2 = <STDIN>);
$produkt = $zahl1 * $zahl2;
print "Das Produkt ist $produkt.\n";
```

Abschnitt 3.6.1 auf Seite 22: Doppelte Zeilen entfernen

doppelzeilen.pl:

```
#!/usr/bin/perl -w
use diagnostics;
while (defined($zeile = <STDIN>)) {
    if (defined($alteZeile)) {
        if ($zeile ne $alteZeile) {
            print $zeile;
        }
    } else {
        print $zeile;
    }
    $alteZeile = $zeile;
}
```

Eine andere Lösung („There is more than one way to do it“):

doppelzeilen2.pl:


```
#!/usr/bin/perl -w
use diagnostics;
if (defined($alteZeile = <STDIN>)) {
    print $alteZeile;
    for (; defined($zeile = <STDIN>); $alteZeile = $zeile) {
        if ($zeile ne $alteZeile) {
            print $zeile;
        }
    }
}
```

Abschnitt 3.8.1 auf Seite 27: Liste invertieren

list-invert.pl:

```
#!/usr/bin/perl -w
use diagnostics;
print reverse <STDIN>;
```

Abschnitt 3.8.2 auf Seite 27: Liste invertieren mit push und pop

list-invert-push.pl:

```
#!/usr/bin/perl -w
use diagnostics;
@liste = ();
while(<STDIN>) {
    push @liste, $_;
}
while(@liste) {
    print pop @liste;
}
```

Abschnitt 3.8.3 auf Seite 27:
Liste invertieren nur pop

```
list-invert-pop.pl:
#!/usr/bin/perl -w
use diagnostics;
@liste = <STDIN>;
while(@liste) {
    print pop @liste;
}
```

Abschnitt 3.8.4 auf Seite 27:
Liste invertieren mit unshift

```
list-invert-shift.pl:
#!/usr/bin/perl -w
use diagnostics;
@liste = ();
while(<STDIN>) {
    unshift @liste, $_;
}
print @liste;
```

Abschnitt 3.8.5 auf Seite 28:
Liste invertieren mit Indizes

```
list-invert-indizes.pl:
#!/usr/bin/perl -w
use diagnostics;
@liste = ();
while(<STDIN>) {
    $liste[++$#liste] = $_;
}
while($#liste >= 0) {
    print $liste[$#liste];
    $#liste--;
}
```

Anmerkung: „\$#liste“ darf erst dekrementiert werden, wenn das hinterste Arrayelement fertig ausgelesen ist. Die **print**-Zeile darf nicht mit der folgenden Zeile vereint werden.

Abschnitt 3.8.6 auf Seite 28:
Liste invertieren mit Zuweisungen

list-invert-assign.pl:

```
#!/usr/bin/perl -w
use diagnostics;
@liste = ();
while(<STDIN>) {
    @liste = ($_, @liste);
}
print @liste;
```

Abschnitt 3.8.7 auf Seite 28:
Liste invertieren mit Zerlegung durch Listenzuweisung

list-invert-listassign.pl:

```
#!/usr/bin/perl -w
use diagnostics;
@liste = ();
while(<STDIN>) {
    @liste = ($_, @liste);
}
while(@liste) {
    ($tmp, @liste) = @liste;
    print $tmp;
}
```

Abschnitt 3.8.8 auf Seite 28:
Liste invertieren mit negativen Indizes

list-invert-negindizes.pl:

```
#!/usr/bin/perl -w
use diagnostics;
@liste = <STDIN>;
for ($idx = 1; $idx <= @liste; $idx++) {
    print $liste[-$idx];
}
```

Abschnitt 3.11.1 auf Seite 40: Cross-Reference-Programm

cross-reference.pl:

```
#!/usr/bin/perl -w
use diagnostics;
@prog = <STDIN>;
$prog = join ' ', @prog;
$prog =~ s/\#.*$/m;
@vars = split /[^\w\@\$]+/, $prog;
$scalars = "";
$arrays = "";
foreach $var (@vars) {
    if ($var =~ /\^$(\w+)$/ ) {
        $var = $1;
        if (!($scalars =~ /\b$var\b/)) {
            $scalars .= "\n$var";
        }
    }
    elsif ($var =~ /\^@\(\w+\)$/ ) {
        $var = $1;
        if (!($arrays =~ /\b$var\b/)) {
            $arrays .= "\n$var";
        }
    }
}
print "The alphanumeric scalar variables are:$scalars\n";
print "\nThe alphanumeric array variables are:$arrays\n";
```

Abschnitt 3.15.1 auf Seite 55: Einfaches Adreßbuch

adressbuch.pl:

```
#!/usr/bin/perl -w
use diagnostics;
while (1) {
    print "Command [(a)dd, (d)elele, (q)uery, (l)ist, (e)xit]? ";
    $cmd = <STDIN>;
    if((!defined $cmd) or $cmd =~ /\^e/i) {
        print "\nBye!\n"; exit 0;
    } elsif($cmd =~ /\^a/i) {
        print "name = ";
        chomp ($name = <STDIN>);
        if (defined $name_phone{$name}) {
            print "Name already exists!\n";
        } else {
            print "phone = "; chomp ($phone = <STDIN>);
            $name_phone{$name} = $phone;
        }
    } elsif($cmd =~ /\^d/i) {
        print "name = "; chomp ($name = <STDIN>);
        if (!exists $name_phone{$name}) {
            print "Name does not exist!\n";
        } else {
            print "Deleting $name with $name_phone{$name}\n";
            delete $name_phone{$name};
        }
    } elsif($cmd =~ /\^q/i) {
        print "name = "; chomp ($name = <STDIN>);
        if (!defined $name_phone{$name}) {
            print "Name does not exist!\n";
        } else {
            print "$name: $name_phone{$name}\n";
        }
    } elsif($cmd =~ /\^l/i) {
        foreach $name (sort keys %name_phone) {
            print "$name: $name_phone{$name}\n";
        }
    } else {
        print "Unknown command!\n";
    }
}
```

Abschnitt 3.16.1 auf Seite 55: Web-Log-Auswertung

weblog.pl:

```
#!/usr/bin/perl -w
use diagnostics;
use English;
sub by_rev_count { $client_count{$b} <=> $client_count{$a} }
while (<>) {
    $client = (split)[0];
    $client_count{$client}++;
}
foreach $client (sort by_rev_count keys %client_count) {
    print "$client: $client_count{$client}\n";
}
```

Kapitel 5:

Lexikalische Analyse mit lex

Abschnitt 5.3 auf Seite 155: Einfacher Taschenrechner

simple-calc.l:

```
%{
#include <math.h>
#include <stdio.h>
double akku = 0.0, neue_zahl = 0.0;
enum op_t {plus, minus, mal, durch, clear} op = clear;
%}
%option main
DIGIT    [0-9]
%%
{DIGIT}+("."{DIGIT}*)?  {
    neue_zahl = atof(yytext);
    switch(op) {
    case plus:    akku = akku + neue_zahl; break;
    case minus:   akku = akku - neue_zahl; break;
    case mal:     akku = akku * neue_zahl; break;
    case durch:   akku = akku / neue_zahl; break;
    case clear:   akku = neue_zahl;        break;
    default:      /* Dies wird nie erreicht. */
                  printf("Arrrrgh!"); exit(1);
    }
    printf("Ergibt: %g\n", akku);
}
"+"      { op = plus; }
"- "     { op = minus; }
"* "     { op = mal; }
"/ "     { op = durch; }
"[Cc]"   { op = clear; }
[" \t\n"] { /* ignoriere Whitespace */ }
.        { printf("Ungueltiges Zeichen '%s'!\n", yytext); }
```

►► make simple-calc

Abschnitt 5.7 auf Seite 159: C-Quellcode-Zähler

```
ccount.1:
%{
#include <stdio.h>
int codeLines = 0, commentLines = 0, blankLines = 0;
int currCodeLine = 0, currCommentLine = 0;
%}
%option noyywrap
%x COMMENT
%%
"/*"          { BEGIN COMMENT; currCommentLine = 1; }
<COMMENT>"*/"  { BEGIN INITIAL; currCommentLine = 1; }
<*>[ \t]       { /* Blank, Tab ignorieren */ }
.              { currCodeLine = 1; }
<COMMENT>.     { currCommentLine = 1; }
<*>"\n"        { if(currCodeLine)
                  codeLines++;
                  else if(currCommentLine)
                      commentLines++;
                  else
                      blankLines++;
                  currCodeLine = 0;
                  currCommentLine = 0;
                }

                /* Die letzte Zeile muss mit einem Newline abgeschlossen sein. */
%%
int main() {
    yylex();
    printf(
        "%d Code-Zeilen, %d Nur-Kommentar-Zeilen, %d ganz leere Zeilen\n",
        codeLines, commentLines, blankLines);
}
```

▶▶ make ccount

▶▶ ./ccount < cexample.c

Zusatzaufgabe:

ccount2.1:

```
%{
#include <stdio.h>
int codeLines = 0, commentLines = 0, blankLines = 0;
int currCodeLine = 0, currCommentLine = 0;
%}
%option noyywrap
%x COMMENT
%x STRING
%%
"/*"          { BEGIN COMMENT; currCommentLine = 1; }
<COMMENT>"*/" { BEGIN INITIAL; currCommentLine = 1; }
"\\""        { BEGIN STRING; currCodeLine = 1; }
<STRING>"\""   { BEGIN INITIAL; currCodeLine = 1; }
<STRING>"\\\\"  { currCodeLine = 1; }
<STRING>"\\\\"  { currCodeLine = 1; }
<STRING>"\n"   { codeLines++; /* String-Fortsetzungszeile */
                currCodeLine = 1;
                currCommentLine = 0;
            }
<INITIAL,COMMENT>[ \t] { /* Blank, Tab ignorieren */ }
<INITIAL,STRING>.      { currCodeLine = 1; }
<COMMENT>.             { currCommentLine = 1; }
<INITIAL,COMMENT>"\n"  {
                        if(currCodeLine)
                            codeLines++;
                        else if(currCommentLine)
                            commentLines++;
                        else
                            blankLines++;
                        currCodeLine = 0;
                        currCommentLine = 0;
                    }
                /* Die letzte Zeile muss mit einem Newline abgeschlossen sein. */
%%
int main() {
    yylex();
    printf(
        "%d Code-Zeilen, %d Nur-Kommentar-Zeilen, %d ganz leere Zeilen\n",
        codeLines, commentLines, blankLines);
}
```

```
▶▶ make ccount2
```

```
▶▶ ./ccount2 < cexample2.c
```

Anmerkung: Die Lösung im Lex&Yacc-Buch ist komplizierter, aber kümmert sich nicht um Strings.

Abschnitt 6.2 auf Seite 172: Klammerausdrücke

ka.y:

```
%{
#include <stdio.h>
%}
%token BUCHSTABE
%%
ka:
    /* empty */
    | ka ka_einfach
    ;
ka_einfach:
    BUCHSTABE
    | '(' ka ')'
    | '[' ka ']'
    | '{' ka '}'
    | '<' ka '>'
    ;
%%
int yyerror(char *msg) {
    return 0;
}
int main() {
    printf("Klammerausdruck: ");
    if(yparse() == 0)
        printf("\nDer Klammerausdruck ist korrekt.\n");
    else
        printf("\nDer Klammerausdruck ist *nicht* korrekt.\n");
    return 0;
}
```

➤➤ make ka

➤➤ ./ka

Inhaltsverzeichnis

1	Überblick über die Veranstaltung Unix-Tools	1
1.1	Vorstellung des Veranstalters	1
1.2	Ziele	1
1.3	Geplante Inhalte	2
1.4	Verfügbarkeit der Werkzeuge	2
1.5	Vorgehensweise	3
2	Der batchorientierte Zeilen-Editor sed	3
2.1	Einfache Textersetzungen	3
2.1.1	Problem: Betreff in Vacation-Text einsetzen	4
2.1.2	Problem: Festen Präfix in einer Dateiliste entfernen	4
2.1.3	Lösung: Betreff in Vacation-Text einsetzen	4
2.1.4	Lösung: Festen Präfix in einer Dateiliste entfernen	5
2.2	Die Grundkommandos von sed	5
2.3	Warum Reguläre Ausdrücke	5
2.3.1	Problem: Einen variablen Präfix in einer Dateiliste entfernen	6
2.3.2	Lösung: Einen variablen Präfix in einer Dateiliste entfernen	6
2.3.3	Problem: Liste aller Benutzer aus <code>/etc/passwd</code> extrahieren	6
2.3.4	Lösung: Liste aller Benutzer aus <code>/etc/passwd</code> extrahieren	7
2.3.5	Problem: Liste aller „echten“ Benutzer aus <code>/etc/passwd</code> extrahieren	7
2.3.6	Lösung: Liste aller „echten“ Benutzer aus <code>/etc/passwd</code> extrahieren	7
2.4	Aufbau der Reguläre Ausdrücke von sed (und grep)	8
2.5	Online-Aufgaben	8
2.5.1	Aufgabe: Erkennen von Email-Adressen	8
2.5.2	Aufgabe: Extraktion von include-Dateinamen aus einer LaTeX-Quelle	9
2.5.3	Aufgabe: Extraktion aller derzeit aktiven Benutzer aus „w“-Ausgabe	9
2.6	Hausaufgaben	10
2.6.1	Schützen von Sonderzeichen in der Vacation-Text-Ersetzung	10
2.7	Kombinieren von Tools: Bearbeiten von ganzen Dateibäumen mit find	10
2.7.1	Aufgabe: Liste aller Dateien unter CVS	10

3	Die Skriptsprache perl	12
3.1	Überblick über das <code>perl</code> -Kapitel	12
3.2	<code>perl</code> statt <code>sed</code> von der Kommandozeile aus	12
3.3	Ein einfaches <code>perl</code> -Programm	12
3.4	Skalare Daten	14
3.4.1	Zahlen, Strings	14
3.4.2	Einschub: Zugang zur <code>perl</code> -Dokumentation	16
3.4.3	Einschub: Eingebaute Warnungen	17
3.4.4	Skalare Variablen	17
3.4.5	Ausgaben mit <code>print</code>	18
3.4.6	Interpolation von Variablen in Doppel-Quote Strings	18
3.4.7	Einschub: Erläuterungen zu Warnungen	18
3.4.8	Vergleichsoperatoren	19
3.4.9	Die <code>if</code> -Kontrollstruktur	19
3.4.10	Boolsche Werte	19
3.4.11	Benutzereingaben	19
3.4.12	Der <code>chomp</code> -Operator	20
3.4.13	Schleifen	20
3.4.14	Der <code>undef</code> -Wert und die <code>defined</code> -Funktion	21
3.5	Online-Aufgaben	21
3.5.1	Aufgabe: Interaktive Multiplikation	22
3.6	Hausaufgaben	22
3.6.1	Aufgabe: Doppelte Zeilen entfernen	22
3.7	Listen und Felder	22
3.7.1	Zugriff auf Feldelemente	22
3.7.2	Besondere Feldindizes	22
3.7.3	Listen-Literale	23
3.7.4	Listen-Zuweisung	23
3.7.5	Felder in Strings interpolieren	24
3.7.6	Die <code>foreach</code> -Anweisung	25
3.7.7	Die Default-Variable <code>\$_</code>	25

3.7.8	Skalarer und Listen-Kontext	26
3.7.9	<STDIN> im Listen-Kontext	26
3.8	Online-Aufgaben	27
3.8.1	Aufgabe: Liste invertieren	27
3.8.2	Aufgabe: Liste invertieren mit push und pop	27
3.8.3	Aufgabe: Liste invertieren nur mit pop	27
3.8.4	Aufgabe: Liste invertieren mit unshift	27
3.8.5	Aufgabe: Liste invertieren mit Indizes	28
3.8.6	Aufgabe: Liste invertieren mit Zuweisungen	28
3.8.7	Aufgabe: Liste invertieren mit Zerlegung durch Listenzuweisung . .	28
3.8.8	Aufgabe: Liste invertieren mit negativen Indizes	28
3.9	Hausaufgaben	28
3.10	Reguläre Ausdrücke von Perl	29
3.10.1	Ersetzungen mit s///	29
3.10.2	Einschub: Kurzform für die while-print-Schleife	29
3.10.3	Unterschiede zu sed und grep	30
3.10.4	Der Bindungsoperator =~	32
3.10.5	Matchen mit m//	33
3.10.6	Modifikatoren	34
3.10.7	Interpolieren in Mustern	35
3.10.8	Die Match-Variablen	35
3.10.9	Einschub: Langformen für kryptische Variablennamen	37
3.10.10	Der split -Operator	37
3.10.11	Die join -Funktion	40
3.11	Hausaufgabe	40
3.11.1	Aufgabe: Cross-Reference-Programm	40
3.12	Slices	41
3.13	Grundlagen der Ein- und Ausgabe	43
3.13.1	Der Diamant-Operator	43
3.13.2	Die Aufruf-Parameter	44
3.13.3	Formatierte Ausgabe mit printf	46

3.14	Assoziative Felder (Hashes)	46
3.14.1	Was ist das?	46
3.14.2	Wozu ist das gut?	47
3.14.3	Syntax	47
3.14.4	Funktionen auf assoziativen Feldern	49
3.15	Online-Aufgaben	55
3.15.1	Aufgabe: Einfaches Adreßbuch	55
3.16	Hausaufgaben	55
3.16.1	Aufgabe: Web-Log-Auswertung	55
3.17	Unterprogramme	56
3.17.1	Syntax	56
3.17.2	Rückgabewerte und der <code>return</code> -Operator	56
3.17.3	Parameter	59
3.17.4	Private Variablen	60
3.17.5	Das Pragma <code>use strict</code>	65
3.17.6	Parameter-Prototypen	66
3.18	Weitere Kontrollstrukturen	66
3.18.1	Die <code>unless</code> -Anweisung	67
3.18.2	Die <code>until</code> -Anweisung	67
3.18.3	Ausdruck-Modifikatoren	68
3.18.4	„Nackte“ Blöcke	69
3.18.5	Die <code>elsif</code> -Anweisung	70
3.18.6	Autoinkrement und Autodekrement	70
3.18.7	Schleifensteuerung	71
3.18.8	Der <code>do</code> -Block	75
3.18.9	Logische Operatoren	76
3.18.10	Der Bedingungsausdruck <code>?:</code>	78
3.18.11	Die <code>case/switch</code> -Anweisung	79
3.19	Datei-Handles und Datei-Tests	81
3.19.1	Öffnen, Lesen und Schließen einer Datei	81
3.19.2	Programmabbruch mit <code>die</code>	83

3.19.3	Öffnen von Pipes	85
3.19.4	Datei-Tests	85
3.20	Module	87
3.20.1	Benutzung einfacher Module	87
3.20.2	Einige wichtige Standardmodule	90
3.20.3	Das Comprehensive Perl Archive Network (CPAN)	90
3.21	Objekt-Orientierung?	91
3.22	Prozeß-Management	91
3.22.1	Die <code>system</code> -Funktion	91
3.22.2	Ausgaben einfangen mit Backquotes	92
3.23	Ausblick auf einiges fortgeschrittenes <code>perl</code>	93
4	Automatisieren der Compilierung mit <code>make</code>	97
4.1	Überblick über <code>make</code>	97
4.2	Grundlagen von Makefiles	97
4.2.1	Einfaches Beispiel: Editor übersetzen	97
4.2.2	Grundalgorithmus von <code>make</code>	99
4.2.3	Grundlagen von Variablen	100
4.2.4	Grundlagen impliziter Regeln	101
4.3	Aufbau eines Makefiles	102
4.3.1	Struktur eines Makefiles	102
4.3.2	Name eines Makefiles	103
4.3.3	Aufbau der Regeln	103
4.3.4	Unechte Ziele	104
4.3.5	Mehrere Regeln für ein Ziel	107
4.3.6	Mehrere Ziele in einer Regel	108
4.3.7	Statische-Muster-Regeln	108
4.4	Anweisungen in Regeln	110
4.4.1	Anweisungs-Ausführung	110
4.4.2	Anweisungs-Echo	111
4.4.3	Parallele Ausführung	111
4.5	Fehlerbehandlung	112

4.5.1	Returncodes	112
4.5.2	Ignorieren von Fehlern	112
4.5.3	Inkonsistente Zielfile bei Fehlern oder Unterbrechungen	113
4.6	Rekursive Verwendung von make	115
4.6.1	Details der Variablen MAKE	118
4.6.2	Optionen und Variablen an Sub-makes übergeben	119
4.7	Muster-Regeln	121
4.8	Altmodische Suffix-Regeln	122
4.9	Automatische Variablen	123
4.10	Ein größeres Beispiel: Makefile zu genFamMem	126
4.11	Implizite Regeln	132
4.12	Verkettung impliziter Regeln	133
4.13	Details von Variablen	134
4.13.1	Syntax	134
4.13.2	Variablen-Expansion	135
4.13.3	Anhängen an Variablen	136
4.13.4	Zwei Arten von Variablen	136
4.13.5	override	137
4.13.6	Bedingte Zuweisung	138
4.13.7	Ersetzungen	139
4.14	Details des Aufrufs von make	140
4.14.1	Angabe von Zielen	140
4.14.2	Kommandozeilenoptionen von make	141
4.15	Konventionen für Makefiles	142
4.16	Fortgeschrittene Makefile-Strukturen	143
4.16.1	Bedingte Teile eines Makefiles	143
4.16.2	Andere Makefiles einschließen	144
4.16.3	Dynamisch generierte Abhängigkeiten	144
4.17	Eingebaute besondere Ziele im Detail	145
4.18	Funktionen	146
4.18.1	Funktionen, die Text transformieren	146

4.18.2	Funktionen für Dateinamen	146
4.18.3	Funktionen – Ganz hartes Zeugs	147
4.19	Gnu-make und andere makes	147
4.20	Beispiel: LaTeX-Kompilierung der Skriptnotizen	147
4.21	autoconf/automake – ein Überblick	150
5	Lexikalische Analyse mit lex	151
5.1	Einführung	151
5.2	Grundaufbau einer Lex-Datei	152
5.2.1	Definitionen	152
5.2.2	Regeln	152
5.2.3	Muster	152
5.2.4	Aktionen	152
5.2.5	Unterprogramm-Abschnitt	153
5.2.6	Beispiel: Wort-Zähl-Programm	153
5.2.7	Beispiel: Scanner für Pascal-artige Sprache, mit weiteren Lex-Konstrukten	154
5.3	Online-Aufgabe: Einfacher Taschenrechner	155
5.4	Match-Algorithmus	156
5.5	Lesen aus Strings	156
5.6	Startbedingungen	157
5.7	Online-Aufgabe: C-Quellcode-Zähler	159
5.8	Mehrere Eingabequellen nacheinander	160
5.9	Mehrere Eingabequellen abwechselnd	163
5.10	Aufruf- und Datei-Optionen von Flex	165
5.11	Weitere Features	166
5.12	Flex und andere Lexer	168
6	Syntaktische Analyse mit yacc	169
6.1	Einführung	169
6.1.1	Kontextfreie Grammatiken und Backus-Naur-Form	169
6.2	Online-Aufgabe: Klammerausdrücke	172
6.3	Semantik zur Syntax	174

6.4	Operator-Assoziativität	177
6.5	Operator-Präzedenz	177
6.6	Fehlerbehandlung	178
6.7	Aufruf-Optionen von Bison	180
6.8	Weitere Features	180
6.9	Bison und andere Versionen von Yacc	181
A	Lösungen der Hausaufgaben und Online-Aufgaben	L-1