Standard- und Programmiertools für UNIX

Eine Einführung in einige wichtige Tools

Stephan Thesing
Version 1.18

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Über dieses Scriptum

Dieses Scriptum wurde anlässlich der Neugestaltung der bis dato ”Unix-Kurs” genannten Veranstaltung im Sommersemester 1999 neu erstellt. Der alte Unix-Kurs wurde zu dieser Zeit in zwei Veranstaltungen aufgeteilt; im Wintersemester werden die Einsteigerthemen, wie Rechnerbenutzung, XWindow, etc. behandelt, im Sommersemester einige speziellere Tools aus der Unix-Umgebung, die im weiteren Verlauf des Studiums von gewisser Bedeutung, z.B. zur Erstellung von Dokumentation bei den FoPras, sind.

In diesem Script, das den Teil des Kurses, welcher im Sommersemester stattfindet, abdeckt, werden zuerst einige allgemeine Tips&Tricks für die Arbeit in der Unix-Umgebung gegeben.

Danach werden in Kapiteln 2 und 3 die Textverarbeitungstools `sed` und `awk` vorgestellt, die eine schnelle Bearbeitung von Daten erlauben, die in Textform vorliegen. Im folgenden Kapitel 4 wird die weit verbreitete Scriptsprache `perl` vorgestellt. Diese bietet auch über `sed` und `awk` hinausgehende Möglichkeiten der Datenmanipulation und hat insbesondere durch den Einsatz im WWW (als Sprache für CGI Scripte) eine große Bedeutung erlangt.

Die Verwaltung verschiedener Files (Programmsourcen, Objektcode, Dokumentationsquellen, etc.) in größeren Projekten schafft durch die untereinander gegebenen Abhängigkeiten der Files oft Probleme; so kann z.B. eine vergessene Neuübersetzung eines geänderten Moduls zu schwer nachzuvollziehenden Fehlern im kompletten Programm führen. In Kapitel 5 wird das GNU `make` Tool vorgestellt, mit dem es möglich ist, diese Abhängigkeiten zu verwalten und nach Änderungen in Teilmodulen alle nötigen Neugenerierungen automatisch auszuführen.

Die Erstellung von Dokumentationen, etwa Hausarbeiten, Seminarausarbeitun-

gen, Diplomarbeiten, FoPra-Dokumentationen, ist eine oft wiederkehrende Aufgabe im Studium. In Kapitel 6 wird daher das LaTeX Textsatzsystem vorgestellt, mit dem Dokumente in professioneller Qualität erzeugt werden können. Zusätzlich behandelt dieses Kapitel auch noch eine Reihe zusätzlicher Tools, mit denen etwa Abbildungen oder Literaturangaben erzeugt werden können.
Bei größeren Programmierprojekten ist eine gründliche Dokumentation unerläßlich. Zusätzlich soll diese auch als Handbuch zu den erstellten Programmen dienen. In Kapitel 7 wird das `texinfo` System vorgestellt, mit dem man aus einer Quelle sowohl ausdruckbare Handbücher, als auch Online-Versionen, in welchen interaktiv herumgewandert werden kann, erstellt werden können.
In Kapitel 8 schließlich wird auf einige Quellen mit zusätzlichen Informationen zu den vorgestellten Tools verwiesen.
Die meisten der Kapitel dieses Scriptes enthalten einen Abschnitt mit einer Reihe von Beispielen, die die dargestellten Konzepte illustrieren sollen.
Als letztes ergeht nun noch die Aufforderung an alle Benutzer dieses Scriptes, Feedback zu geben; sowohl was Fehler im Script betrifft, als auch Themen, die im Rahmen dieses Kurses behandelt werden sollten, oder Beispiele, die in die zugehörigen Abschnitte aufgenommen zu werden verdienen.

1.1.1 Konventionen

In diesem Script werden verschiedene Konventionen benutzt:

- Um auf die Manual-Page (siehe Abschnitt 1.2.3) einer Programmes, eines Files oder einer Funktion zu verweisen, geben wir den Namen der Manual-Page, zusammen mit dem Manual-Abschnitt an, z.B.: *ls(1)*. Wir geben die Manual-Page ebenfalls an, wenn wir eigentlich über das Programm (die Funktion,...) reden.

  Man beachte, daß einige Manual-Pages mehrmals (in verschiedenen Abschnitten) vorkommen, etwa *intro(1)*, *intro(2)* (die Einführungen zu den jeweiligen Manual-Abschnitten).

- Ausgaben auf dem Bildschirm werden in Typewriter dargestellt, etwa `uid=10016(thesing)` als Ausgabe des *id(1)* Kommandos.

- Eingaben des Benutzers werden fett gedruckt dargestellt, etwa **id thesing**.

- Teile, die variabel sind, etwa ein Benutzername, werden kursiv dargestellt, etwa wie in `ls ~`*`username`*`/.netscape`.

1.2 Einige Tips & Tricks

In diesem Kapitel werden einige nützliche Tips & Tricks vorgestellt, mit denen man einige Probleme beheben kann, die bei der Arbeit unter UNIX immer wieder auftauchen. Zuerst wird das wichtige Thema der *Libraries* besprochen, die Unterschiede zwischen statischen und dynamischen Libraries erklärt.
Im folgenden Abschnitt werden einige nützliche Umgebungsvariablen vorgestellt und gezeigt, wie man das Verhalten des Systems mit ihnen beinflußen kann.
Der letzte Abschnitt zeigt auf, wo man Hilfe bei auftretenden Fragen bekommen kann.

1.2.1 Libraries

Libraries sind eine Sammlung von Objectcode-Files, die eine Reihe häufiger benutzter Funktionen enthalten. So enthält etwa die Standard-C Library (`/usr/lib/libc.a`) u.a. die Funktion *printf(3)*, mittels derer man Text ausgeben kann. Abbildung 1.1 illustriert den Weg von einem Sourcecode-File zu einer Library, wobei *gcc(1)* zum Kompileren benutzt wird und *ar(1)* zum Zusammenfügen der Objectcode-Files zu einer Library[1].
Wenn man eigene Libraries generieren will, dann muß man die Objectcode-Files mittels *ar(1)* zusammenstellen und danach einen Index dieses Archivs (Library) erzeugen, indem man *ranlib(1)* aufruft, z.B.:

ar cr libfoo.a file1.o ... fileN.o
ranlib libfoo.a

Die Optionen `rc` besagen, daß die Files hinzugefügt werden sollen und die Library ggf. neu kreiert wird, wenn sie noch nicht existiert.
NB: Wenn man eine Library neu übersetzt, sollte man die alte Version mittels **rm libfoo.a** vorher löschen, da sonst das Ersetzen (`r` = replace) der alten Objectcode-Files lange dauern kann.
Diese Library kann man dann mittels **-lfoo** beim Linken angeben[2].
Probleme können allerdings auftreten, wenn die Objectcode-Files voneinander abhängen, etwa in `file1.o` Funktionen aus `file2.o` aufgerufen werden. Denn beim Linken wird eine Library nur von vorne nach hinten durchsucht. Taucht in einem Objectcode-File in einer Library eine Referenz auf, muß diese in einem weiter hinten liegenden File aufgelöst werden.

[1] Gibt man *gcc* die Option **-v**, kann man die Aufrufe der einzelnen Stufen der Compilierung verfolgen

[2] Zusätzlich kann es nötig sein, mit **-L*dir*** noch das Directory anzugeben, in welchem die Library liegt.

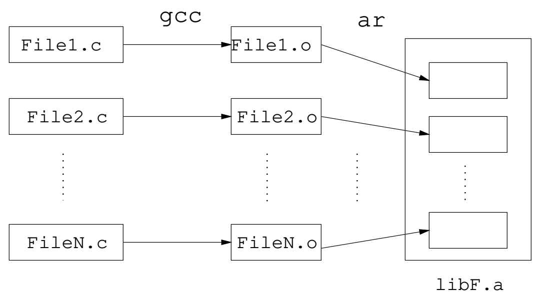

Abbildung 1.1: Vom Sourcecode-File zur Library.

Nun fügt *ar(1)* aber die Objectcode-Files in der Reihenfolge der Library hinzu, in welcher sie auf der Kommandozeile angegeben wurden. Um eine optimale Reihenfolge zu finden, benutzt man am besten die beiden Tools *lorder(1)* und *tsort(1)*. Ersteres erzeugt eine Abhängigkeitsliste, die angibt, welches File Referenzen auf ein anderes File enthält. *tsort(1)* sortiert diese Liste dann so, daß Objectcode-Files, welche referenziert werden, nach den Referenzen stehen. Man benutzt also etwa folgende Befehlsfolge:

ar rc libfoo.a ‘lorder file1.o . . . fileN.o | tsort‘
ranlib libfoo.a

In obigem Beispiel werden `lorder` und `tsort` in einer Subshell aufgerufen (wegen der einfachen Rückstriche ’ ‘’) und deren Ausgabe dann als Fileliste für `ar` eingesetzt.
Bei der letzten Stufe des Übersetzens eines Programmes, dem *Linken*, werden die einzelnen Objectcode-Files des Programmes mit denen aus den Libraries zusammengestellt, welche Funktionen enthalten, die das Programm benutzt. Alle diese Objectcode-Files werden aneinandergehängt und ergeben letztendlich das ausführbare Programm, siehe Abbildung 1.2.
Neben dieser Methode des Linkens (siehe übrigens *ld(1)* und *link(5)* dazu), welche man *statisches Linken* nennt, gibt es noch eine andere Methode, das *dynamische Linken*: Beim statischen Linken wird jedesmal eine Kopie der aus einer Library benötigten Objctcode-Files in das Programm übernommen, verbraucht also Platz auf der Platte und (bei der Ausführung des Programmes) auch im Speicher.
Vor allem bei sehr häufig genutzten Libraries, etwa `libc.a`, `libX11.a`, hat man also u.U. eine große Anzahl von Kopien. Außerdem muß man bei einer Änderung der Library (etwa einem Update des XWindow-Systems) alle Programme neu übersetzen, damit sie die neue Library verwenden können. Beim dynamischen Linken hat man hingegen nur (höchstens) eine Kopie jeder Library im Speicher. Alle Programme, die einen Teil dieser Library benutzen, verwenden

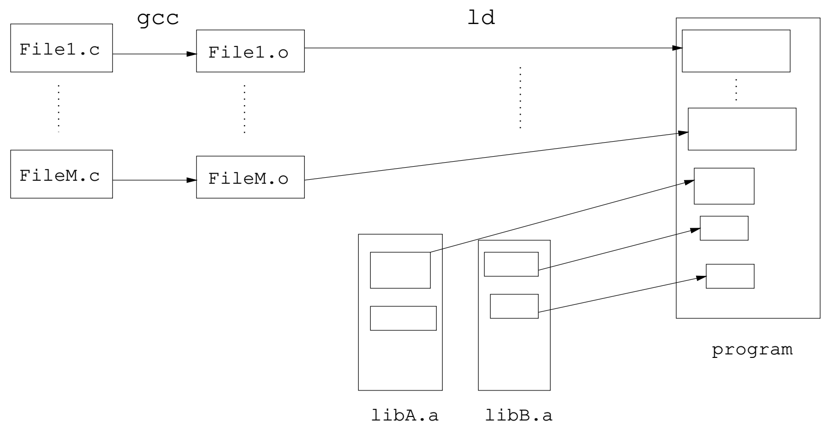

Abbildung 1.2: Vom Sourcecode zum Programm (statisch)

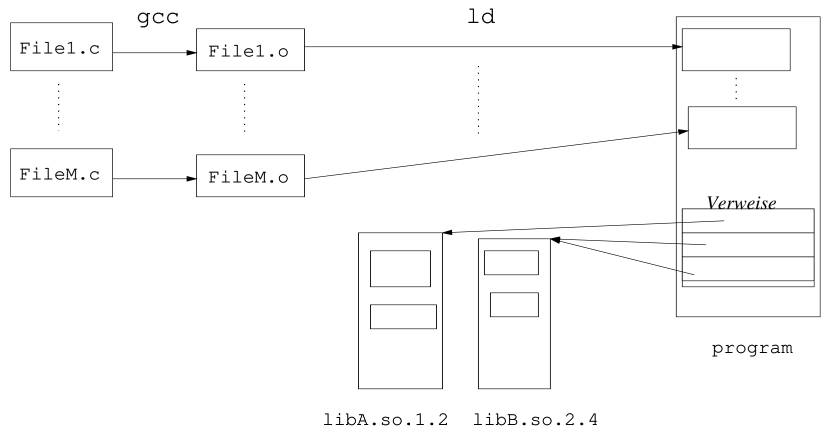

Abbildung 1.3: Dynamisches Linken

dieselbe Kopie im Speicher. Beim Linken werden nun nicht mehr die einzelnen Objectcode-Files aus der Library in das Programm übernommen, sondern nur Verweise auf die Library im Programm-File eingetragen. In Abbildung 1.3 ist dieser Sachverhalt noch einmal dargestellt.
Beim Laden des Programmes löst das Betriebssystem, mittels des *ld.so(1)* dynamischen Linkers[3], diese Verweise auf:

- Wenn die referenzierte Library noch nicht im Speicher ist (also dieses Programm das erste ist, welches einen Teil dieser Library braucht), dann wird die Library gesucht und in den Speicher geladen.

[3]Dieser wird vom Startupcode des Programmes geladen und ausgeführt.

- Ist die Library im Speicher, werden alle Referenzen im Programm auf die Library an die tatsächlichen Speicheradressen der Library angepaßt.

Dabei muß allerdings die Library auf eine spezielle Weise erzeugt worden sein, damit nämlich alle Referenzen in ihr unabhängig zu ihrer Startadresse im Speicher (die für verschiedene Programme variieren kann) auftreten[4]. Bei der Kompilierung von C-Programmen muß man z.B. die Option `-fPIC` (oder `-fpic`) angeben, siehe *gcc(1)*. Auch das Zusammenstellen der einzelnen Objectcode-Files zur Library muß speziell erfolgen, so daß man nicht die selben Library-Files für statisches und dynamisches Linken verwenden kann.
An dieser Stelle sei auf das GNU `libtool` Programm verwiesen[5], das sich um all diese speziellen Kommandos kümmert und vor allem auf verschiedenen Plattformen verwendet werden kann.
Allgemein haben statische Libraries immer die Endung `.a`, dynamische Libraries immer etwas in der Form von `.so.`*`Major`*`.`*`Minor`*. Hierbei gibt *Major* die Hauptversion und *Minor* die Unterversion der Library an. Damit werden mehrer Versionen derselben Library unterstützbar.
Die dynamischen Libraries werden beim Laden eines Programmes gesucht, wenn sie noch nicht im Speicher vorhanden sind. Damit *ld.so(1)* sie finden kann gibt es mehrere Mechanismen. Zum einen wird ein Cache mit Library-Namen und ihren Positionen auf der Platte angelegt, der das Suchen dynamischer Libraries beschleunigt. Dieser wird (meist beim Booten des Systems) mittels *ldconfig(8)* angelegt und kann i.d.R. nur vom Superuser geändert werden. Zum anderen sucht *ld.so(1)* in Verzeichnissen, welche in der Umgebungsvariablen `LD_LIBRARY_PATH` (durch Doppelpunkte getrennt) angegeben werden; etwa wird nach **setenv LD_LIBRARY_PATH .:$HOME/lib** im aktuellen Verzeichnis und im Unterverzeichnis `lib` im Home-Verzeichniss nach Libraries gesucht. Als letztes wird auch noch in den Directories gesucht, welche mit `-L`*`dir`* beim Linken des Programmes angegeben wurden.
Findet `ld.so` eine dynamische Library `libfoo.so.`*`Major`*`.`*`Minor`*, dann

- Falls die *Major* Nummer der gefundenen Library sich von der Nummer der gesuchten Library unterscheidet, wird sie ignoriert.

- Es wird immer die Library mit der höchsten Minor Nummer (und gleichen Major Nummern) genommen. Dadurch kann immer die aktuellste Version einer Library verwendet werden. Ist die höchste gefundene Minor Nummer *kleiner* als die Minor Nummer, die beim Linken des Programmes gefunden

[4]Es darf also nur relative Adressierung benutzt werden.
[5]Siehe etwa unter `http://www.gnu.org`

wurde (kann z.B. passieren, wenn man ein Programm auf einer SPARC-10 linkt und dann auf einer SPARC-1 ausführt), dann wird eine Warnung ausgegeben, die Library aber trotzdem akzeptiert.

Wenn man wissen will, welche dynamischen Libraries von einem Programm benutzt werden, gibt es zwei Wege:

- Man setzt die Umgebungsvariable `LD_TRACE_LOADED_OBJECTS` in der Shell auf irgendeinen Wert (etwa mittels des Kommandos **setenv LD_TRACE_LOADED_OBJECTS**) und ruft dann das zu untersuchende Programm einfach auf. Statt das Programm auszuführen, werden nun alle dynamischen Libraries, die das Programm benötigt, ausgegeben (und ihre Filenamen, sollten sie gefunden werden). Natürlich muß man, wenn man danach wieder Programme ausführen will, die Umgebungsvariable wieder löschen (**unsetenv LD_TRACE_LOADED_OBJECTS**).

- Alternativ kann man das *ldd(1)* Programm aufrufen: **ldd *program***, was zur selben Ausgabe führt. (Bem.: Unter NetBSD oder Linux ist *ldd(1)* nichts anderes, als ein Wrapper, der die obige Umgebungsvariable setzt und dann das Programm ausführt! Ob dies auch für SunOS zutrifft kann ich mangels Sourcecode nicht sagen, vermute es aber.)

Beim Kompilieren und Linken eigener Programme ist das dynamische Linken immer als Default eingestellt. Gibt man also eine Library mittels **-lfoo** zum Linken an, dann wird zuerst nach derjenigen dynamischen Library `libfoo.so.`x`.`y gesucht, die die höchste Major Nummer x hat. Bei mehreren dynamischen Libraries mit derselben Major Nummer wird diejenige mit der höchsten Minor Nummer y genommen.
Nur wenn keine dynamische Library gefunden wird, wird (ebenfalls unter Verwendung von `LD_LIBRARY_PATH`) nach einer statischen Library `libfoo.a` gesucht. Manchmal ist es aber wünschenswert, das statische Linken zu erzwingen. Dies kann man *gcc(1)* mittels der **-static** Option mitteilen. Mit dieser Option werden dann keine dynamischen Libraries gesucht.
Man kann auch gegen einige Libraries dynamisch, gegen andere statisch Linken. Siehe hierzu `ld(1)`.

1.2.2 Environment Variablen

Environment Variablen beeinflussen das Verhalten vieler Programme. So verwenden z.B. fast alle Programme den mittels der Variablen `EDITOR` angegeben Editor, wenn der Benutzer aus ihnen heraus Text editieren muss (z.B. *mail(1)*).
Eine Liste der gängisten Variablen ist in *environ(5)* zu finden.
Noch zu erwähnen sind:

- `MANPATH`: Diese Variable enthält, durch Doppelpunkte getrennte, Directories, in denen nach Manual-Pages gesucht wird. In jedem so angegebenem Directory sollten Subdirectories mit Namen `cat`*`section`* oder `man`*`section`* existieren, in welchen die formatierten (`cat`) oder rohen Manual-Pages (`man`) stehen. Der Default für den Suchpfad nach Manual-Pages ist `/usr/share/man`. Nützlich ist dies, wenn man Software selbst installiert hat und auch deren Manual-Pages angezeigt bekommen möchte. Legt man die Seiten etwa unterhalb des `$HOME/man` Directories an, so werden die lokalen Seiten nach einem `setenv MANUALPATH $HOME/man:/usr/share/man` ebenfalls von *man(1)* angezeigt.

- `INFOPATH`: Das `info` Kommando zeigt ebenfalls Informationen zu Programmen an (siehe Abschnitt 1.2.3). `INFOPATH` enthält eine Liste (durch Doppelpunkte getrennt) von Directories, in denen nach Info-Seiten gesucht wird, entspricht in seiner Funktion also `MANPATH`. Der Default für die Suche ist i.A. `/usr/local/info`.

Im folgenden werden noch einige Environment Variablen vorgestellt, die von der *tcsh(1)* verwendet werden, um deren Verhalten zu beeinflußen.

tcsh Variablen

Die *tcsh(1)* ist eine oft benutzte interaktive Shell, die wohl die meisten Benutzer als ihre Login-Shell benutzen. Ihr Verhalten kann durch das Setzen von lokalen Variablen vielfältig beeinflußt werden.
Lokale Variablen werden in dieser Shell mittels **set *var=value*** gesetzt, im Gegensatz zu globalen Variablen, die mittels **setenv *var value*** gesetzt werden. Der Unterschied zwischen beiden besteht darin, daß nur globale Variablen an Programme weitergegeben werden.

- `autolist`: Nach einer versuchten Completion (mittels TAB oder CTRL-D), werden alle Möglichkeiten der Vervollständigung (alle in Frage kommenden Filenamen, etc.) automatisch angezeigt.

- `cdpath`: Diese Variable enthält eine Liste von Directories und wird mittels **set cdpath=(*path1 path2* ...)** gesetzt. Will man mittels **cd** in ein Directory wechseln, das nicht im akuellen Directory existiert, dann wird dieses Directory in den Directories von `cdpath` gesucht und dahin gegangen.

- `fignore`: Diese Variable enthält eine Liste von Endungen, die bei der Completion ignoriert werden sollen. Will man z.B. keine `.dvi` Files oder Objectcode Files angezeigt bekommen, kann man dies mittels **set fignore=(.dvi .o)** erreichen.

- `histdup`: In der `tcsh` sind alle Kommandos in der History-Liste gesammelt, so daß man durch einfaches Blättern mittels der Cursortasten vorherige Kommandos wieder erreichen kann. Die Variable `histdup` steuert, was mit doppelten Einträgen in der Histroy-Liste passiert (etwa, wenn man dasselbe Kommando mehrmals hintereinander eingibt). Setzt man sie auf `prev` (**set histdup=prev**), so wird ein Kommando nur dann in die History-Liste aufgenommen, wenn es vom vorherigen Kommando verschieden ist. Setzt man sie auf den Wert `all`, dann wird das Kommando nur in die Liste aufgenommen, wenn es noch gar nicht in ihr vorkommt. Ein Setzen auf den Wert `erase` bewirkt, daß das Kommando am Ende der Liste aufgenommen wird und alle anderen Vorkommen desselben Kommandos aus der Liste entfernt werden.

- `implicitcd`: Wird diese Variable gesetzt, dann kann man in ein Unterverzeichnis wechseln, indem man einfach dessen Namen eingibt (und das sonst übliche `cd` davor wegläßt).

- `notify`: Ist diese Variable gesetzt, wird die Beendigung einer Hintergrundjobs (mittels **&** gestartete Kommandos) sofort bei der Beendigung des Jobs auf dem Bildschirm gemeldet. Ansonsten wird bis zum nächsten Drücken der Return-Taste gewartet.

- `rmstar`: Ist diese Variable gesetzt, dann wird bei der Eingabe von **rm *** nachgefragt, bevor alle Files gelöscht werden.

Als weiteres nützliches Schmankerl kann man in der tcsh unter XWindow den Rechnernamen und das aktuelle Directory[6] in der Titelleiste anzeigen lassen. Dazu muß man einen besonderen Alias namens `cwdcmd` besetzen, etwa folgendermassen[7]:

```
alias cwdcmd
 'echo -n "^[]2;${HOST}:$cwd^G"'
```

Hierbei steht `^[` für den ASCII Code 27 (Escape) und `^G` für Control-G. Eingeben kann man dies z.B. im `vi` mittels '**CTRL-VESC**' bzw. **CTRL-VCTRL-G**. Auch auf der Kommandozeile der Shell kann man mit CTRL-V das folgende Zeichen quoten, es also direkt übernehmen.
Die Shell verfügt über eine ganze Reihe zusätzlicher Variablen, ein Studium von *tcsh(1)* ist daher oft nützlich, wenn auch langwierig.

[6] Das nebenbei bemerkt immer in der Variablen `$cwd` steht.

[7] Die folgenden Zeilen sind nur der Lebarkeit halber im Script umgebrochen, eingeben sollte man sie als eine Zeile.

1.2.3 Hilfe!

Auf UNIX Systemen gibt es mehrere Wege, um an Information zu Programmen zu gelangen:

- Meistens gibt das Programm selbst, wenn z.B. ohne Optionen oder mit **--help** [8] oder **-h** aufgerufen eine kurze Meldung mit den verfügbaren Optionen aus.
- Mit ziemlicher Sicherheit existiert für das Programm eine Manual-Page. Diese kann mit dem `man` Programm aufgerufen werden. Z.B. liefert **man man** Hilfe über das `man` Programm selbst.

  Die Manual-Pages sind nach Abschnitten geordnet, etwa Programme, Systemaufrufe, Funktionen, etc.pp, dabei sind die Abschnitte durchnumeriert (1,...,8,l,n):

  - In Abschnitt 1 werden Benutzerkommandos beschrieben.
  - Abschnitt 2 widmet sich den Systemaufrufen.
  - Die Library-Funktionen werden in Abschnitt 3 behandelt.
  - Abschnitt 4 beschreibt Devices (z.B. Tastatur oder `/dev/null` (*null(4)*), das "Schwarzes-Loch-Gerät".)
  - In Abschnitt 5 findet man Informationen zu Fileformaten, etwa von `/etc/passwd`: *passwd(5)*.
  - Die verfügbaren Spiele werden in Abschnitt 6 beschrieben.
  - Einige nützliche Sachen, die sonst nirgendwohin passen, sind in Abschnitt 7 zusammengestellt.
  - Abschnitt 8 gibt Informationen zu Kommandos, die i.A. nur für den Systemverwalter interessant sind.
  - In Abschnitt l werden lokale Kommandos beschrieben.

  Ein Aufruf von **man 1 printf** gibt z.B. die Erläuterungen zu dem `printf` Programm, während **man 3 printf** die Erläuterungen zu der Funktion `printf` liefert. Eine Einführung in die verschiedenen Abschnitte der Manual-Pages erhält man mittels **man *section* intro**.

  `man` hilft nur weiter, wenn man den Namen des Programmes (der Funktion,...) bereits kennt. Sucht man nach Informationen zu einem bestimmten Schlagwort, hilft *apropos(1)* weiter, das alle Manual-Pages, in denen das

[8] Fast alle Programme aus dem GNU Projekt unterstützen diese Option. SunOS Programme dagegen nicht.

Schlagwort vorkommt, auflistet. Ebenso sollte man sich *whatis(1)* einmal zu Gemüte führen.

- Viele GNU Programme, etwa `make`, `texinfo` kommen neben rudimentären Manual-Pages auch noch mit einer anderen Form der Dokumentation, den *info*-Einträgen. Diese stellen eine verzeigerte Sammlung von Seiten dar (ähnlich einem HTML Dokument), in denen man aus Menues Punkte auswählen und auch Querverweisen folgen kann. Das `info` Programm dient dabei zur Navigation innerhalb dieser Einträge. In Kapitel 7 werden wir das `texinfo` System kennenlernen, mit dem man genau solche Info-Seiten erzeugen kann (und auch noch normale, druckbare Dokumentation!). Anfangs ist die Navigation innerhalb der Info-Seiten etwas gewöhnungsbedürftig (vor allem für die GUI, ein-Knopf-für-alles verwöhnten Benutzer von heute), allerdings leicht zu erlernen. Eine Einführung in `info` erhält man (wen verwundert's) mittels **info info**.

- Für alle die, die ohne das Web nicht mehr leben können gibt es unter der URL `http://www.delorie.com/gnu/docs` Dokumentationen zu den meisten GNU Programmen.

Kapitel 2

sed

2.1 Einführung

In diesem Kapitel werden wir sed, den *Stream EDitor*, kennenlernen. sed ist ein Editor, aber anders als z.B. vi oder emacs, ist er nicht interaktiv, sondern läuft im Batch-Betrieb. D.h. man gibt sed ein Script mit, das er dann auf die Eingabedaten anwendet und das Ergebnis wieder ausgibt, siehe Abbildung 2.1.
sed arbeitet zeilenorientiert: Er lädt immer eine Zeile in seinen Puffer, wendet die Kommandos aus dem Script darauf an und gibt das Resultat dann am Schluß wieder aus[1].
Man kann sed das Script auf zwei Arten mitgeben:

- Man gibt nach der Option **-e** das Script auf der Kommandozeile an: **sed -e *script***.
- Man schreibt das Script in ein File und gibt den Namen des Files nach der **-f** Option an: **sed -f *file***.

Z.B. hat **sed -e 's/Uni Saarbruecken/Uni des Saarlandes/g' Antrag** zur Folge das alle Vorkommen von `Uni Saarbruecken` in dem File `Antrag` durch `Uni des Saarlandes` ersetzt werden. Das Kommando ist in diesem Fall das *Substitute* Kommando (erkenntlich an dem s vor dem ersten Schrägstrich)[2]. Dieses ersetzt Vorkommen des Pattern zwischen den ersten beiden Schrägstrichen durch den Text zwischen dem zweiten und dritten Schrägstrich. Das g Flag am Ende signalisiert dem Kommando, daß es alle Vorkommen in einer Zeile ersetzen soll, nicht nur das erste, was den Default darstellt.

[1] Nun ja, durch das Konzept des "Hold-Spaces" und Multi-Line Pattern ist es nicht ganz so einfach, siehe Abschnitt 2.3.3.

[2] Die Hochkommata "'" dienen nur zum Quoten der Scriptes in der Shell, da dieses Leerzeichen enthält.

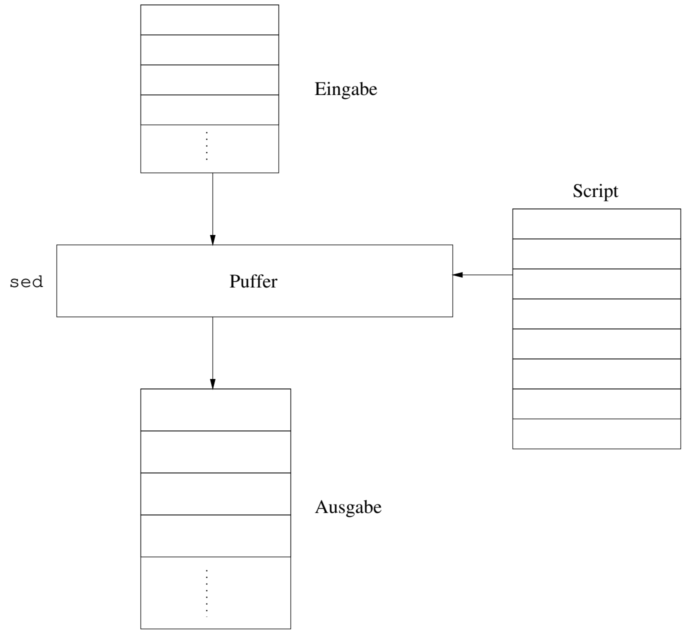

Abbildung 2.1: `sed`'s Arbeitsweise

Schreibt man das folgende Script in ein File `scr.sed`:

```
s/Uni Saarbruecken/Uni des Saarlandes/g
/XXX/d
```

und ruft dann z.B. **sed -f scr.sed Antrag** auf, dann werden wie oben alle Vorkommen von `Uni Saarbruecken` ersezt; danach wird jetzt aber auch noch jede Zeile gelöscht, in der der Text `XXX` vorkommt[3]. Das Löschen wird durch das zweite Kommando in dem Script ausgeführt, dem *Delete* Kommando, erkenntlich an dem `d`. Dem Kommando ist eine *Adresse* vorgestellt, die angibt, bei welchen Zeilen es ausgeführt werden soll. Adressen werden wir weiter unten noch besprechen.
Normalerweise nimmt `sed` seine Eingabedaten von der Standardeingabe, man kann aber auch die Namen von Files auf der Kommandozeile angeben, so daß deren Inhalt nacheinander bearbeitet wird: **sed -e** ***script file***$_1$ ***file***$_2$ **...**.
Wie oben gesagt, gibt `sed` nach der Abarbeitung des Scriptes die Zeile in seinem Puffer wieder aus. Man kann dieses Verhalten abschalten, indem man die **-n** Option auf der Kommandozeile angibt. Dann muß man allerdings den Kommandos

[3]Eine beliebte Methode, Stellen in Files zu kennzeichnen, an denen noch etwas bearbeitet werden muß, ist es, sie mit einem Kommentar der Form `XXX` *`Kommentar`* zu versehen.

im Script explizit sagen, daß sie etwas ausgeben sollen. Wir werden weiter unten sehen, welche Optionen man diesen Kommandos dazu geben muß.
`sed` gibt seinen Buffer immer auf der Standardausgabe aus[4]. `sed` verändert niemals Files, die man als Eingabe auf der Kommandozeil angegeben hat[5].
`sed` bietet die Möglichkeit, Kommandos im Script nur dann anzuwenden, wenn der Puffer auf ein bestimmtes *Pattern* paßt, respektive die zugehörige Zeile in einem bestimmten *Adreßbereich* liegt (etwa bestimmte Zeilennummern). Sowohl Pattern als auch Adressen werden wir weiter unten genauer besprechen.
Es gibt eine ganze Reihe von Versionen von `sed`. Wir werden im folgenden ausschließlich die GNU Version von `sed` betrachten.

2.2 Pattern Matching

Pattern begegnen einem in der UNIX Welt sehr häufig. `sed` benutzt die meisten der Standard-Pattern die auch die Shells anbieten, und noch einige weitere.
Wie üblich werden dazu bestimmte Zeichen zu *Metazeichen*, die dann für Wiederholungen, etc. stehen.
In `sed` werden Pattern an zwei Stellen verwandt:

- In Adressen. Adressen dienen dazu, einige Kommandos im Script nur auf bestimmte Zeilen der Eingabe anzuwenden.
- Im *Substitute* Kommando von `sed`. Dieses dient dazu, bestimmte Teile einer Eingabezeile durch andere Teile zu ersetzen. Siehe Abschnitt 2.3.2

Das Matchen des Patterns erfolgt dabei anders, als bei der Shell. Es wird geprüft, ob das Pattern *irgendwo* im Puffer passt, nicht vom Anfang an (außer, man legt das Pattern mit den Metazeichen `^` oder `$` explizit an den Anfang/ans Ende des Puffers, s.u.). So passt etwa das Pattern `Pattern` auf die Zeile `Das Matchen des Patterns erfolgt...`. Diese Semantik führt oft zu ungewünschten Matches. Man sollte also, wo nötig, das Pattern explizit mit `^` und `$` auf den Anfang/das Ende zwingen.
Besonders zu beachten ist, das `sed` immer versucht von links her den längsten Teil einer Zeile zu matchen. D.h. das Pattern **a*a*** matcht in der Eingabe `aaaa` die ersten vier '`a`' auf den ersten Teil und das leere Wort auf den zweiten Teil des Patterns. Dies ist für das Pattern im Substitute Kommando wichtig, da man dort die Teile, die von einem bestimmten Teil des Patterns gematcht wurden, wieder im Ersetzungstext einsetzen kann.

[4] D.h. man muß ihn mit >***file*** umlenken, wenn man das Ergebnis in einem File haben will.

[5] Es sei denn, man tut etwas törichtes in der Art von **sed -e *script* foo >foo**. Dies führt zu lustigen Effekten, allerdings nie den gewünschten.

Im folgenden Abschnitt werden nun zuerst die einzelnen Metazeichen vorgestellt und erläutert. In Abschnitt 2.2.2 werden dann die Adressen vorgestellt.

2.2.1 Pattern

`sed` kennt die folgenden Meta-Zeichen:

`.` Der Punkt steht für ein beliebiges Zeichen, ausgenommen ein Newline (\n).

`[...]` Dies definiert eine Klasse von Zeichen. Meta-Zeichen zwischen diesen Klammern verlieren ihren Meta Status, werden also wieder zu dem Zeichen selbst (`[.]` matcht also nur den Punkt selbst.). Ein `^` als erstes Zeichen nach der öffnenden Klammer dreht den Sinn des Matchings um: die Klasse matcht dann nur die Zeichen, die *nicht* zwischen den Klammern auftreten. Ein `-` gibt *Bereiche* von Zeichen an, so matcht `[0-9]` die Ziffern 0 bis 9. Taucht das Zeichen `]` als erstes Zeichen nach der öffnenden Klammer (und ggf. einem `^`) auf, steht es für die eckige rechte Klammer selbst, also matcht `[^]0-9]` alles ausser den Ziffern 0 bis 9 oder der rechten eckigen Klammer `]`.

`^` Steht dieses Zeichen am Anfang des Patterns, paßt es auf den Anfang des Puffers. Steht es später im Pattern, hat es keine spezielle Bedeutung mehr und passt also nur auf das Zeichen `^` selbst.

`$` Steht dieses Zeichen am Ende des Patterns, passt es auf das Ende ders Puffers es woanders, verliert es seine spezielle Bedeutung und passt also nur noch auf das Zeichen `$` selbst.

`*` Der Stern markiert beliebig oftes Auftreten des Zeichens, das durch das vor im stehenden Pattern beschrieben wird. So passt `0*` auf `0`, `00`, etc. aber auch das leere Wort. Ebenso passt `[0-9][0-9]*` auf eine nicht leere Folge von Ziffern.

`\{`n,m`\}` Matcht n bis m maliges Auftreten des Zeichens, das im vorstehenden Pattern beschrieben wird. Als Spezialform passt `\{`n`\}` auf *genau* n Auftreten und `\{`n`,\}` auf *wenigstens* n Auftreten.

c Jedes andere Zeichen c matcht genau dieses Zeichen.

Besonders zu beachten ist, daß der `*` und der `\{`n`,`m`\}` Operator immer nur ein vorhergehendes *Zeichen* (welches auch durch eine Klasse angegeben werden kann) wiederholen, aber keine anderen vorhergehenden Pattern. So ist `a\{3\}*` ein illegales Muster und passt *nicht* auf beliebig ofte Wiederholungen von drei `a`'s. Dahingegen ist `[a-zA-Z]*` legal und passt auf beliebig viele Buchstaben.

2.2.2 Adressen

Adressen werden in `sed` benutzt, um bestimmte Script-Kommandos nur auf bestimmte Zeilenbereiche anzuwenden. Eine Adresse kann dabei sein:

- Ein Bereich, der durch eine Anfangszeile und eine Endzeile gegeben ist. Dieser hat die Form *`Anfang`*`,`*`Ende`*. Dabei können *Anfang* und *Ende* sein:

  - Eine Zeilennummer[6], wobei das Symbol `$` für die letzte Zeile steht (deren Nummer i.A. nicht im voraus bekannt ist).
  - Ein Pattern, eingeschlossen in '`/`': `/`*`pattern`*`/`.

  Der Bereich einer solchen Adressangabe wird durch folgende Schritte bestimmt.

  1. Es wird verglichen, ob die aktuelle Zeile auf den Anfang passt. Wenn ja, wird das Aktiv-Flag gesetzt.
  2. Ist das Aktiv-Flag gesetzt, wird die Zeile in den Puffer übernommen und von den folgenden Kommandos im Script bearbeitet.
  3. Es wird geprüft, ob die Zeile auf das Ende passt. Wenn ja, wird das Aktiv-Flag gelöscht.

- Eine einzelne Spezifikation, die wieder eine Zeilennummer oder eine Spezifikation der Form `/`*`pattern`*`/` sein kann. Hier wird jede Zeile mit dieser Adresse verglichen und dann durchgelassen, wenn die Zeilennummer stimmt, oder das Pattern in der Zeile passt (je nachdem, welche Adressart man angegeben hat.).

Wenn einer Adresse ein Ausrufungszeichen folgt, wird die Reichweite der Adresse negiert, die folgenden Kommandos also nur auf Zeilen angewandt, die nicht in den Adressbereich fallen.
Fehlt eine Adresse vor einem Kommando, wird dieses Kommando auf alle Zeilen angewandt, die bis zu ihm durchgelassen wurden.
Einige Beispiele für Adressen und ihre Reichweiten:

- `1,$` Reicht von der ersten bis zur letzten Zeile.
- `1,/^$/` Reicht von der ersten Zeile bis zur *ersten* Leerzeile[7].
- `/^%%%SOE/,/^%%%EOE/` Umfasst alle Blöcke im File, die durch ein `%%%SOE`, `%%%EOE` Paar geklammert sind.
- `1,10!` Umfasst alle Zeilen, *ausser* den ersten zehn.

[6] `sed` numeriert die Eingabeeilen von 1 an durch.

[7] Nützlich, um mittels `1,/^$/d` den Header einer Mail zu löschen.

2.3 Kommandos

Ein Kommando kann eine der folgenden Formen haben:

Kommando[*Argumente*]
adrKommando[*Argumente*]

Wobei *adr* eine Adresse in den oben dargelegten Formen ist, *Kommando* ein Buchstabe, welcher das Kommando beschreibt und *Argumente* die Argumente zu dem Kommando. Wir werden bei der Beschreibung der einzelnen Kommandos angeben ob und welche Argumente es benutzt. Es sei angemerkt, daß sich nicht alle Kommandos mit allen Adressangaben vertragen. So gibt es einige, denen höchstens eine einfache Adresse (Zeilennummer oder Pattern), aber kein Bereich vorangestellt werden kann. Dies wird ebenfalls bei der Beschreibung des Kommandos erwähnt werden.
Enthält ein Script mehrere Kommandos, müssen diese durch ein Semikolon oder einen Zeilenumbruch getrennt werden. Z.B.:

```
s/FB14/Fachbereich 14/g ; s/s\./siehe/g
```

oder, wenn das Script in einem File steht:

```
s/FB14/Fachbereich 14/g
s/s\./siehe/g
```

2.3.1 Gruppierung

In `sed` werden die geschweiften Klammern benutzt, um Kommandos einzugruppieren. In solchen Gruppen, kann man wiederum Kommandos mit den geschweiften Klammern gruppieren.
Beispiel:

```
1,/^$/ { s/gargoyle.cs.uni-sb.de/cs.uni-sb.de/
         /[Ww]arning/d
       }
```

Die Möglichkeit der Gruppierung erlaubt es, auch die Kommandos, die nur eine einfache Adressangabe vertragen (s.o.) auf Adressbereiche anzuwenden, man schreibt einfach statt ***adr1,adr2Kommando*** die Gruppierung ***adr1,adr2*{ *Kommando*}**.

2.3.2 Basis-Kommandos

Wir listen die Kommandos in alphabetischer Reihenfolge auf.

Append

Das Append Kommando, repräsentiert durch den Kommando-Buchstaben `a` fügt Text nach der aktuellen Zeile ein. Dieser Text wird am Schluß des Scriptes ausgegeben, ist jedoch bis dahin nicht weiter verfügbar. D.h. der Text wird nicht an den Puffer angehängt und verändert diesen also auch nicht. Er wird nur nach dem Pufferinhalt ausgegeben, wenn dieser schließlich ausgegeben wird. Dieses Kommando kann nur mit einer einfachen Adresse benutzt werden und nimmt eine spezielle Form von Argumenten:

[*adr*]`a\`
`Text`
Das Kommando muß sofort von einem Rückstrich und dem Zeilenende gefolgt werden. Der auszugebende Text folgt dann mit Beginn der nächsten Zeile. Wenn der Text Newlines enthält, also über mehrere Zeilen geht, muß vor dem Ende jeder Zeile (ausser natürlich der letzten) ein Rückstrich stehen.

Change

Das Change-Kommando (`c`) ersetzt die Zeilen, die durch eine ggf. vorstehende Adresse (auch einen Adressbereich) durch den Text, der als Argument folgt. Dabei ist die Form dieses Textes so, wie beim Append Kommando:

[*adr*][`,` *adr*]`c\`
`Text`

Delete

Das Delete Kommando (`d`) löscht einfach die Zeilen, die zu im durchgelassen werden. Das heisst, der Puffer wird verworfen, die nächste Zeile wird gelesen und das Script wieder vom Anfang an abgearbeitet. Allerdings wird Text, der durch ein Append oder Insert Kommando eingefügt wurde, noch ausgegeben:

[*adr*][`,` *adr*]`d`

Insert

Das Insert Kommando (`i`) ist ein Analogon zum Append Kommando, nur wird der Text am Schluß nicht nach, sondern vor dem Puffer ausgegeben:

[*adr*]`i\`
`Text`

Line-Number

Das Line-Number Kommando (=) gibt einfach die Zeilennummer der aktuellen Zeile aus. Es kann nur eine einfache Adresse vorgestellt haben:

[*adr*]=

List

Das List Kommando (`l`) gibt den Puffer aus (ähnlich wie das Print Kommando), nur werden nicht-druckbare Zeichen (etwa ESC) als ASCII Codes ausgegeben, etwa erzeugt **sed -n -e 'l'** bei der Eingabe von **ESCa** die Ausgabe `\033a\n$`. Die Form des Kommandos ist:

[*adr*][`,`*adr*]`l`

Next

Das Next Kommando gibt den Puffer aus, liest die nächste Zeile ein und fährt mit der Bearbeitung des Scriptes beim nächsten Kommando, nicht am Scriptanfang, fort:

[*adr*][`,`*adr*]`n`

Print

Das Print Kommando (`p`) gibt den Puffer aus. Wenn `sed` nicht mit der **-n** Option aufgerufen wurde, kann der Pufferinhalt mehrmals ausgegeben werden (etwa am Ende des Scriptes):

[*adr*][`,`*adr*]`p`

Quit

Das Quit Kommando (`q`) macht genau das, was sein Name vermuten lässt: Der Puffer wird ausgegeben (wenn `sed` keine **-n** Option hatte) und `sed` wird beendet. Dieses Kommando kann nur eine einfache Adresse vorgestellt haben:

[*adr*]`q`

Read

Das Read Kommando (`r`) ähnelt dem Append Kommando, nur wird der Text, der angehängt werden soll aus einem File gelesen, dessen Namen (durch genau einen

Zwischenraum vom Kommando getrennt) als Argument folgt. Diesem Kommando kann nur eine einfach Adresse vorgestellt werden:

[*adr*]`r` *`file`*

Substitute

Das Substitute Kommando (`s`) ist das wohl am häufigsten benutzte Kommando in `sed`. Es dient dazu, bestimmte Teile des Puffers zu ersetzen. Welche Teile ersetzt werden sollen, wird im ersten Argument als Pattern spezifiziert, die Ersetzung im zweiten Argument:

[*adr*][`,`*adr*]`s/`*`pattern`*`/`*`replacement`*`/`[*`flags`*]

Im Pattern ist neben den allgemeinen Metazeichen noch Gruppierung erlaubt. D.h. man kann bestimmte Teile in `\(`, `\)` Paare einschliessen. Wenn das Pattern passt, wird der Teil zwischen diesen Paaren in speziellen Registern gespeichert. Im Ersetzungstext kann man dann mittels `\`n, wobei *n* eine Ziffer von 1 bis 9 ist, den Inhalt des zugehörigen Registers wieder einsetzen. Z.B. würde

s/^#define \([^][^]*\) \(.*\)$/\1 \2/

in einem C-Header File die Definiton eines Macros durch den Namen und den Wert des Macros ersetzen.
Als weiteres spezielles Zeichen wird '`&`' im Ersetzungstext durch den gesamten Match des Patterns ersetzt.
Das Kommando kennt die folgenden Flags:

n Ersetzt das n-te Auftreten des Patterns in der Zeile, nicht das erste.

g Ersetzt alle Auftreten des Patterns, nicht nur das erste.

p Gibt die Zeile aus, falls etwas in ihr ersetzt wurde. Wenn es mehrere Ersetzungen gab (etwa, weil das `g` Flag angegeben wurde), dann wird die Zeile mehrmals ausgegeben.

w $file$ Wenn eine Ersetzung gemacht wurdem wird die Zeile in das File $file$ geschrieben.

Man kann die beiden Argumente auch durch ein anderes Zeichen, als den Schrägstrich '`/`' trennen, z.B. **s+Uni Saarbruecken+Uni des Saarlandes+**. Dies ist besonders nützlich, wenn das Pattern oder der Ersetzungstext selbst einen Schrägstrich enthalten.
Einige weitere Beispiele dazu finden sich in Abschnitt 2.4.

Translate

Das Translate Kommando (`y`) dient dazu, Buchstaben umzusetzen, etwa Kleinbuchstaben in Großbuchstaben. Dazu wird das erste Zeichen aus dem ersten Argument durch das erste Zeichen aus dem zweiten Argument ersetzt:

[*adr*][`,`*adr*]`y/`*`Arg1`*`/`*`Arg2`*

Z.B. wird nach **sed -e 'y/abcdef/ABCDEF/'** jeder Kleinbuchstabe von 'a' bis 'f' durch den entsprechenden Großbuchstaben ersetzt[8].
Ebenso wie beim Substitute Kommando kann man statt des Schrägstriches ein anders Zeichen zur Trennung der beiden Argumente benutzen.

Write

Das Write Kommando (`w`) schreibt den Puffer (Aber *nicht* die mittels eines Append, Insert, etc. angefügten Texte) in das File, dessen Name (durch genau einen Zwischenraum getrennt) als Argument folgt:

[*adr*][`,`*adr*]`w` *`file`*

2.3.3 "Advanced Features"

In diesem Abschnitt wollen wir kurz drei etwas komplexere Bereiche von `sed` beleuchten:

- Sprünge im Script
- Den Hold-Space
- Multi-Line Pattern

Sprünge

Bisher wurde davon ausgegangen, daß `sed` die Kommandos des Scriptes sequentiell von oben nach unten abarbeitet. Manchmal ist es aber wünschenswert, zu anderen Teilen des Scriptes zu verzweigen und sei es nur, um nicht Teile duplizieren zu müssen. Hierfür gibt es in `sed` drei Kommandos:

- Das Label Kommando setzt Sprunkmarken in das Script.
- Das Branch Kommando verzweigt zu der durch ein Label gekennzeichneten Stelle im Script.

[8] Nützlich, um Hexadezimalzahlen zu normalisieren.

- Das Test Kommando prüft, ob im Puffer mittels des Substitute Kommandos eine Ersetzung stattgefunden hat und verzweigt in diesem Fall zu einer durch ein Label gekennzeichneten Stelle.

Label

Das Label Kommando (`:`) ist eigentlich kein Kommando, da es nichts tut. Es setzt nur eine Marke in das Script, die dann angesprungen werden kann:

`:`*`label`*

Der Name des Labels (*label*) darf bis zu sieben Buchstaben enthalten.

Branch

Das Branch Kommando (`b`) springt zu der Stelle des Scriptes an der das angegebene Label steht. Wenn man das Label fortlässt, wird zum Ende des Scriptes gesprungen:

[*adr*][`,` *adr*]`b` [*label*]

Test

Das Test Kommando (`t`) stellt einen bedingten Sprung dar. Als Bedingung wird geprüft, ob im Puffer mittels eines Substitute Kommandos bereits eine Ersetzung stattgefunden hat. Wenn ja, wird zu dem angegebenen Label, bzw. dem Ende des Scriptes gesprungen, wenn das Label fehlt:

[*adr*][`,` *adr*]`t` [*label*]

Der Hold-Space

Der Hold-Space stellt einen Zwischenspeicher dar, in welchen der Puffer mittels des Hold Kommandos kopiert werden kann. Das Get Kommando kopiert dann den Hold-Space wieder in den Puffer, während das Exchange Kommando die beiden vertauscht. Dies kann nützlich sein, um z.B. Zeilen in der Eingabe zu vertauschen. Angenommen, man hat eine Reihe von Files der Form

```
Titel: ...
Autor: ...
```

Und möchte in diesen nun den Autor vor dem Titel stehen haben. Folgendes `sed` Script leistet gerade dies:

```
/^Titel:/{
h;d
}
/^Autor:/{
p; g
}
```

Der erste Teil kopiert jede Zeile, die mit 'Titel:' beginnt in den Hold-Space und löscht sie dann, so daß sie nicht ausgegeben wird. Sobald die Zeile mit 'Autor:' am Anfang erreicht ist, wird diese ausgegeben und dann der Inhalt des Hold-Spaces (die Titel-Zeile) in den Puffer kopiert, der dann am Schluß des Scriptes ausgegeben wird.

Get

Das Get Kommando (`g`) kopiert den Inhalt des Hold-Spaces in den Puffer dessen voheriger Inhalt verlorengeht:

[*adr*][`,` *adr*]`g`

Hold

Das Hold Kommando (`h`) kopiert den Inhalt des Puffers, aber keine mittels der Append oder Insert Kommandos angefügten Texte, in den Hold-Space, dessen vorheriger Inhalt verlorengeht:

[*adr*][`,` *adr*]`h`

Exchange

Das Exchange Kommando (`x`) vertauscht den Inhalt des Hold-Spaces mit dem Puffer:

[*adr*][`,` *adr*]`x`

Multi-Line Pattern

Wir hatten anfangs gesagt, daß `sed` nur jeweils eine Zeile bearbeiten kann. Manchmal ist es aber nötig, Aktionen über mehrer Zeilen auszuführen, etwa wenn man eine Phrase aus mehreren Wörtern ersetzen will, da zwischen den einzelnen Wörtern ja ein Zeilenumbruch stehen kann[9]. Man kann `sed` mittels spezi-

[9] Wie in der Ersetzung `Uni Saarbruecken` → `Uni des Saarlandes`

eller Kommandos anweisen, die nächste Eingabezeile (oder den Inhalt des Hold-Spaces) an den Puffer anzuhängen. Die Zeilen werden dann durch ein Newline getrennt, welches in einem Substitute Kommando dann durch `\n` gematcht werden kann. Auch kann man den Puffer an den Inhalt des Hold-Spaces anhängen. Alle bisher vorgestellten Kommandos arbeiten immer auf dem gesamten Puffer[10], etwa das Delete Kommando. `sed` stellt nun einige Kommando Varianten bereit, die nur auf der ersten Zeile im Puffer arbeiten.
Will man also z.B. 'Uni Saarbruecken' durch 'Uni des Saarlandes' auch über Zeilenumbrüche hinweg ersetzen, kann man folgendes Script benutzen:

```
s/Uni Saarbruecken/Uni des Saarlandes/g
/Uni$/ {
 N
 s/Uni\nSaarbruecken/Uni\
des Saarlandes/
P;D
}
```

Dieses Script ersetzt zunächst alle Vorkommen von 'Uni Saarbruecken', die auf einer Zeile sind. In dem Fall, daß ein 'Uni' am Ende der Zeile steht, könnte eine mehrzeilige Ersetzung nötig sein, also wird die nächste Zeile mittels des Next-ML Kommandos (`N`) an den Puffer angehängt. Das folgende Substitute Kommando ersetzt nun alle durch ein Newline getrennten Vorkommen von 'Uni Saarbruecken'. Als Besonderheit beachte man, daß die Ersetzung geschieht, indem das 'Uni' auf der ersten Zeile verbleibt und der neue Text 'des Saarlandes' an den Anfang der zweiten Zeile gesetzt wird. Da man in Ersetzungstexten keine Angaben der Form `\n` machen kann, muß man im Script einen Zeilenumbruch an dieser Stelle machen, der wiederum durch einen Rückstrich eingeleitet werden muß. Das Print-ML Kommando (`P`) gibt nun die erste Zeile aus. Das folgende Delete-ML Kommando (`D`) löscht dann die erste Zeile im Puffer und macht mit der zweiten Zeile am Anfang des Scriptes weiter, damit auch Vorkommen in dieser Zeile ersetzt werden können.

Delete-ML

Das Delete-ML (für Multi-Line) Kommando (`D`) löscht den Teil des Puffers bis einschließlich des ersten Newlines. Wird dadurch alles gelöscht (d.h. es war nur eine Zeile im Puffer), dann wird die nächste Eingabezeile gelesen. Das Script wird in jedem Fall wieder von der ersten Position an abgearbeitet:

[*adr*][`,` *adr*]`D`

[10] Der jetzt mehrere Zeilen enthalten kann:-)

Get-ML

Das Get-ML Kommando (`G`) hängt ein Newline, gefolgt vom Inhalt des Hold-Spaces an den Puffer an. Ist der Hold-Space leer, wird trotzdem ein Newline angehängt:

[*adr*][`,` *adr*]`G`

Hold-ML

Das Hold-ML (`H`) Kommando fügt ein Newline und den Puffer an den Inhalt des Hold-Spaces an. Das Newline wird auch angefügt, wenn der Hold-Space komplett leer ist:

[*adr*][`,` *adr*]`H`

Next-ML

Das Next-ML Kommando (`N`) liest die nächste Eingabezeile und hängt sie, durch ein Newline getrennt, an den Puffer an:

[*adr*][`,` *adr*]`N`

Print-ML

Das Print-ML Kommando (`P`) gibt den Teil des Puffers bis zum ersten Newline aus. Ansonsten identisch zum Print (`p`) Kommando:

[*adr*][`,` *adr*]`P`

2.4 Beispiele

2.4.1 Konfigurationsfiles

Oft kann man Files generisch anlegen, d.h. große Teile der Files sind konstant, nur einige Teile müssen je nach der Umgebung oder den Eingabedaten angepaßt werden.
Makefiles sind ein Beispiel hierfür; einige Teile, z.B. die Pfade zu Installations-directories müssen je nach Installation angepaßt werden. `sed` kann nun benutzt werden, um diese Einfügungen zu machen, indem man für solche Parameter Platzhalter vorsieht, die dann mittels des Substitute Kommandos durch die wirklichen

Plätze ersetzt[11]. Man erstellt also eine Schablone (z.B. `Makefile.in`), die die Platzhalter, z.B. in der Form `@INSTDIR@`, enthält und lässt `sed` dann ein Script mit dem folgenden Kommando ausführen: **s/@INSTDIR@/*real-inst-dir*/g**.
Dies kann man einfach in ein Bourne-Shell Script integrieren:

```
#! /bin/sh

....

#$instdir contains the dir  the stuff should be put to
#$programs contains the list of programs to build
...

# generate sed script
cat >/tmp/sed.scr <<EOM
s/@INSTDIR@/${instdir}/g
s/@PROGS@/${programs}/g
s/@VERSION@/${progversion}/g
EOM

for f in Makefile.in lib/Makefile.in doc/Makefile.in;
 do
 outfile=`echo $f | sed -e 's/\.in$//'`
 sed -f /tmp/sed.scr $f >$outfile
 echo created $outfile
done

rm /tmp/sed.scr
```

Dieses Shell Script generiert ein temporäres `sed` Script `/tmp/sed.scr`, wobei die Shell Variablen (`$instdir`, ...) durch deren Werte ersetzt sind. Danach wird für jede Schablone (`Makefile.in`, ...) die Ersetzung durchgeführt und ein File erzeugt, das denselben Namen wie die Schablone hat, ausser der Endung `.in`. Hier wurde `sed` benutzt, um die Endung abzuschneiden. Eine Alternative wäre ``outfile=`dirname $f`/`basename $f .in` `` gewesen. Allerdings kann man `sed` auch bei beliebigen Endungen benutzen (`s/\.[^.]*$//`).

[11]Dies ist der Weg, den das GNU `configure` Tool geht, welches automatisch Software-Packete für die vorhandene Architektur konfiguriert.

2.4.2 Extraktion von Daten

Angenommen, man hat Datenfiles (`dat.1`, `dat.2`, ...), in denen z.B. der Name des Datensatzes in folgender Weise vermerkt ist:

```
....
%%%Datensatz: Experiment_mit_Optimierung_1
....
```

Bei der Auswertung möchte man nun z.B. wieder in einem Shell Script über alle Datenfiles laufen, den Namen des Datensatzes ausgeben und danach einige Ergebnisse. Ein Fragment des Shell Scriptes könnte dann so aussehen:

```
....
# Zsfsg. generieren
for f in dat.*; do
 dname=`sed -n -e 's/^%%%Datensatz: \(.*\)$/\1/p' $f`
 echo Datensatz $dname >>Zsfsg

 # gen. Datenzsfsg.
 ....
done
```

Hier wird die `-n` Option benutzt, die verhindert, daß Zeilen automatisch ausgegeben werden. Das Substitute Kommando ersetzt dann den Namen des Datensatzes und gibt diesen aus (durch die `p` Option).

2.4.3 Umformungen

Angenommen, man hat in einer Shell Variablen eine Liste von Directories, die durch Doppelpunkte getrennt sind, etwa `$MANPATH` oder `$PATH`. Wenn man jetzt in jeder dieser Directories z.B. nach einem bestimmten File suchen will kann man (für die tcsh) z.B. folgendes eingeben:

```
foreach d (`echo $MANPATH|sed -e 's/:/ /g'`)
 if ( -f $d/whatis ) echo $d hat apropos Datenbank
end
```

Das `sed` Script ersetzt einfach die Doppelpunkte durch Leerzeichen, das Ergebnis wird dann von der Schleife als eine Liste von Directories benutzt.
Angenommen, man will einen Namen daraufhin prüfen, ob er ein legaler Shell Variablenname ist. Folgender Test in einem Bourne Shell Script kann dies leisten (der Name steht in der Variablen `$aname`):

```
if [ -n "`echo $aname|sed -e 's/[-a-zA-Z0-9_]//g'`" ];
 then
 echo illegaler name
else
 ...
fi
```

Das Script ersetzt jeden Buchstaben, jede Ziffer, jeden Trennstrich und Unterstrich durch nichts, löscht sie also. Ist das Ergebnis nicht leer (der `-n` Test in der Shell), existiert ein illegales Zeichen, der Name ist also ungültig.
Manchmal will man bei einem Namen alle Schrägstriche am Ende entfernen. Dies kann einfach durch die folgende Anweisung erledigt werden:

```
newname=`echo $name | sed -e 's+/*$++'`
```

Da `sed` immer den längsten Match vornimmt werden durch diese Substitution alle Schrägstriche am Ende entfernt. Man beachte auch die Verwendung von + statt / als Trenner des Patterns von der Ersetzung.
Angenommen, man möchte einen Namen, der u.a. Trennstriche ('-'), Punkte, Schrägstriche oder Pluszeichen enthält in einen Namen verwandeln, der dem C-Präprozessor als Macroname angeboten werden kann. Hier die Lösung:

```
cppname=`echo $aname| sed -e 'y%./-+%___p%'`
```

Hierbei werden Pluszeichen durch ein 'p' ersetzt, alles andere durch Unterstriche.

2.5 Falltüren

2.5.1 Pattern

Die häufigste Tretangel in `sed` sind dessen Pattern und das Matching. `sed` kennt keine Gruppierungen in Pattern mittels Klammerung, Meta-Operatoren, z.B. '`*`', wirken immer nur auf ein Zeichen. Auch kann man in `sed` keine Sonderzeichen, etwa TAB mittels der gewohnten Escape-Sequenzen darstellen (etwa `\t`). Diese Zeichen müssen roh eingegeben werden, wozu man also den Quote-Mechanismus des Editors oder der Shell bemühen muss: CTRL-V beim vi und der Shell, CTRL-Q beim emacs. Die einzige Ausnahme sind Multi-Line Pattern, in denen ein Newline mittels `\n` gematcht werden kann. Allerdings kann man in dem Ersetzungstext ein Newline *nicht* mittels `\n` einfügen, sondern muß dies tun, indem man einen Rückstrich ('`\`') eingibt, sofort gefolgt von dem Newline.
Verwirrung schafft häufig auch die Tatsache, daß `sed` immer den längsten Match von links her vornimmt, somit spätere Pattern nicht auf den Teil der Eingabe passen, den man im Sinn hatte, etwa: `s/\(.*\) \(.*\)/\1-\2/` Dies führt

auf der Eingabe '`abcd efg hij`' *nicht* zu der vielleicht beabsichtigten Ausgabe '`abcd-efg hij`', sondern zu '`abcd efg-hij`'. Richtig wärte gewesen '`s/\([^ ]*\) \(.*\)/\1-\2/`'.

2.5.2 Zusatztext

Oft tritt man auch auf die Tretmine, daß die Texte, welche mittels eines Append, Insert oder Read Kommandos angehängt wurden, nicht von den folgenden Kommandos des Scriptes bearbeitet werden. Sie werden auch nicht von dem Hold Kommando in den Hold-Space übernommen.

2.5.3 Reihenfolgen

Man sollte auch berücksichtigen, daß Substitutionen von späteren Substitutionen wieder verändert werden können, z.B. wenn man in einem File alle Variablen namens '`$A`' nach '`$B`' und alle Variablen namen '`$B`' nach '`$C`' umbenennen will:

```
s/\$A/$B/g
s/\$B/$C/g
```

Dieses Script führt leider dazu, daß alle '`$A`' nach '`$C`', über den Umweg '`$B`', umbenannt werden. Richtig wäre gewesen:

```
s/\$B/$C/g
s/\$A/$B/g
```

2.6 Literatur

Als erstes sind natürlich die Manual-Pages und die Info Seiten von `sed` zu nennen. Wer es etwas ausführlicher mag, dem sei das gute Buch [Dou90] empfohlen, an welches sich auch dieses Kapitel angelehnt hat.
Beispiele, an denen man lernen kann, finden sich z.B. in den `configure` Shell Scripten, die fast jedem GNU Programm im Source-Archiv beiliegen. Unter NetBSD kann man unter `/usr/src/usr.bin/sed/TEST` ein `sed` Script bestaunen, welches die Lösung zum 'Turm-von-Hanoi' berechnet. Außerdem findet man dort ein Script, das mathematische Ausdrücke ausrechenen kann (in `sed`). Der GNU Source-Distribution von `sed` liegen unter dem Directory `testsuite` eine Reihe von Scripten bei, die man sich durchaus ansehen sollte.

Kapitel 3

awk

3.1 Einführung

awk ist ein weiteres Tool, mit dem man Textfiles bearbeiten kann. Es bietet dabei weit über sed hinausgehende Möglichkeiten.
awk bietet, statt der simplen Manipulationskommandos von sed, eine eigene Programmiersprache, die stark an die Sprache **C** angelehnt ist.
awk spaltet die Eingabe in Records auf, diese wiederum in Felder. Dabei sind die Zeichen, die Records und Felder in der Eingabe trennen, einstellbar. Als Default werden die Zeilen der Eingabe als Records genommen und Felder in jeder Zeile durch Leerzeichen getrennt.
awk kann nun für jeden Record bestimmte Programmstücke ausführen, wobei der Inhalt und die Anzahl der Felder in dem aktuellen Record in speziellen Variablen zur Verfügung steht. Welches Programmstück für einen Record ausgeführt wird, kann man (ähnlich sed) durch vorgestellte Adressen auswählen.
Der Aufruf von awk geschieht ähnlich, wie bei sed: Entweder gibt man das komplette Programm auf der Kommandozeile an, oder man schreibt es in ein File und gibt den Namen des Files nach der **-f** Option mit.
Als Beispiel betrachten wird dieses File:

```
1 34.5
Versuch Nr 1
2 89.54
Versuch Nr 2
3 773.0
Versuch Nr 3: ...
4 333.11
5 993.11
```

Dies kann etwa die Nummern von Messungen, gefolgt von den Messwerten eines Experimentes darstellen. Unter diesen Zahlenpaaren folgt ggf. ein Kommentar. Wenn wir jetzt den Durchschnitt dieser Messwerte haben wollen, könnten wir folgendes `awk` Programm schreiben:

```
/^[0-9]/{ summe=summe+$2; anzahl++; }
END { durch=summe/anzahl; printf("%4.2f\n",durch); }
```

Dieses Programm zeigt schon den typischen Aufbau eines `awk` Programmes: Es besteht aus mehreren Adresse/Programmstück Paaren. Dabei muß das Programmstück immer in geschweiften Klammern eingeschlossen werden. In unserem Beispiel selektiert die Adresse des ersten Stückes alle Records, welche mit einer Ziffer beginnen. Im Programmstück wird dann einfach die Summe um das zweite Feld dieses Records erhöht. Die Felder eines Record sind in den zugehörigen Programmstücken mittels der speziellen Variablen '`$`i' ansprechbar, wobei i die Nummer des Feldes ist. Als Besonderheit steht in `$0` der gesamte Record. Woher weiss `awk` aber nun, daß das zweite Feld eine Zahl darstellt? In `awk` gibt es eine Unterscheidung zwischen Strings und Zahlen eigentlich nicht. In Wirklichkeit steht in `$2` genau der Text des Feldes. Dieser wird bei Bedarf umgewandelt. Da wir hier eine Addition ausführen, wird versucht, `$2` in eine Zahl umzuwandeln, was in unserem Beispiel immer gelingt. Schlägt dieser Versuch aber fehl, weil z.B. in `$2` etwa '`foo`' steht, wird nicht etwa ein Fehler ausgelöst, sondern einfach '`0`' als Ergebnis genommen.
In unserem Programm steht noch die merkwürdige Adresse '`END`'. Diese, zusammen mit ihrem Gegenstück '`BEGIN`', hat eine spezielle Rolle: Das zugehörige Programmstück wird *nach* der Verarbeitung des letzten Records, bevor `awk` beendet wird, ausgeführt. Dies ist besonders nützlich, um am Schluß Zusammenfassungen auszugeben. Im Unterschied dazu wird das Programmstück hinter einem '`BEGIN`' *vor* dem Lesen des ersten Records ausgeführt und kann zu Initialisierungen genutzt werden. In unserem Fall berechnen wir am Schluß den Durchschnitt und geben ihn dann aus.
Unsere Eingabe enthält noch Zeilen, die nicht von einer Adresse selektiert werden, nämlich die drei Kommentare. `awk` gibt diese Zeilen nicht etwa einfach aus, sondern ignoriert sie schlichtweg.
Wichtig ist auch noch die Tatsache, daß man eine Adresse auch komplett weglassen kann. Dann wird das folgende Programmstück für jeden Record ausgeführt.
Gibt es mehrere passende Adress/Programmstück Paare, welche auf eine Zeile passen, dann werden sie in der Reihenfolge ihres Auftretens im Programm abgearbeitet.

3.1.1 Versionitis

Leider existieren eine ganze Reihe von Versionen von `awk`, welche sich doch stark unterscheiden. Wir werden hier nur GNU `awk` betrachten, das die meisten Features bietet und auch frei verfügbar ist.

3.2 Adressen & Pattern

In `awk` werden ebenfalls Adressen der Form, wie sie in `sed` auftauchen (siehe Abschnitt 2.2.2), verwandt. Zusätzlich kennt `awk` aber noch weitere Adressangaben, so daß wir nochmal alle zusammenfassen:

/pattern/ Ein `sed`-ähnliches Pattern, allerdings sind mehr Operatoren erlaubt (s.u.).

rel-expr Ein relationaler Ausdruck (s. Abschnitt 3.3), der zu einem Wahrheitswert auswertet. Das Programmstück wird ausgeführt, wenn der Warheitswert 'wahr' ($\neq$`0`) ergibt.

adr && adr Der `&&` Operator steht für die Und-Verknüpfung zweier Adressen. Nur wenn beide auf den aktuellen Record passen, wird dieser bearbeitet.

adr* || *adr Ähnlich wie `&&`, nur für die Oder-Verknüpfung.

(*adr*) Klammerung von Adressen.

!*adr* Negation der Adresse. Der aktuelle Record wird nur bearbeitet, wenn die Adresse nicht passt.

adr?adr:adr Diese Adresse ist ähnlich dem `?:` Operator von **C**. Wenn die erste Adresse auf den aktuellen Record passt, dann wird noch die zweite Adresse getestet, sonst die dritte.

adr,adr Ein Bereich von Adressen, mit der gleichen Semantik wie bei `sed`.

BEGIN Diese spezielle Adresse steht vor Programmstücken, dir vor dem Einlesen des ersten Records ausgeführt werden sollen.

END Diese Adresse steht vor Programmstücken, die nach der Bearbeitung des letzen Records ausgeführt werden sollen.

Die Pattern von `sed` werden nun von `awk` erweitert, es gibt die unten aufgeführten zusätzlichen Metazeichen.

+ Der + Operator steht für ein- oder mehrmaliges Auftreten des vor im stehenden Patterns. Z.B. passt `[0-9]+` auf mindestens eine Ziffer.

? Der `?` Operator steht für null- oder genau einmaliges Auftreten des vor ihm stehenden Patterns. Also passt `lines?` auf '`line`' oder '`lines`' und sonst nichts.

| Dieser Operator steht für die Alternative zwischen zwei Pattern. So passt `a|b` entweder auf '`a`' oder auf '`b`'.

() `awk` bietet, anders als `sed`, Klammerung von Pattern. So passt `a(bc)*` auf '`a`', '`abc`', '`abcbc`', ...

3.3 Variablen und Ausdrücke

3.3.1 Konstanten

Konstanten treten in `awk` in zwei Formen auf:

- Als Strings, eingeschlossen in '`"`', z.B. `"Ein String"`.
- Als numerische Konstanten, egal ob als Ganzzahl ('`65535`') oder Fließkommazahl ('`3.14159265`').

Es sei aber angemerkt, daß `awk` bei Bedarf konvertiert. So weist das Kommando `s="9"-1` der Variablen `s` die Zahl `8` zu!
In String Konstanten stehen die von **C** her gewohnten Escape-Sequenzen zur Verfügung, um nicht-druckbare Zeichen darzustellen, etwa '`\n`' oder '`\f`'.

3.3.2 Variablen

`awk` kennt drei Typen von Variablen:

- Vom Benutzer definierte Variablen. Diese werden einfach durch ihre Verwendung im Programm definiert und müssen nicht deklariert werden (z.B. `summe` im obigen Beispiel).
- Eingebaute Variablen. Diese werden von `awk` vordefiniert. Ihre Namen bestehen immer komplett aus Großbuchstaben, so daß eine Verwechslung mit anderen Variablen schwierig ist. Weiter unten sind alle vordefinierten Variablen aufgelistet.
- Felder. Diese enthalten die Felder des aktuellen Records. Sie werden mittels `$`*expr* angesprochen, wobei *expr* zu einer Zahl auswertet, die die Nummer des Feldes angibt (von `1` an). Das 'Feld' `$0` enthält den gesamten Record.

Vordefinierte Variablen

Folgende Variablen werden von `awk` vordefiniert:

ARGC Die Anzahl von Kommandozeilenparametern.

ARGV Ein Array mit den Kommandozeilenparametern.

ENVIRON Ein assoziatives Array, das die Umgebungsvariablen enthält.

FILENAME Der Filename des aktuellen Eingabefiles.

FNR Die Nummer des aktuellen Records, relativ zum aktuellen File.

FS Der Feldseperator, also das Zeichen, das Felder trennt. Default: Leerzeichen

IGNORECASE Bestimmt, ob das Matching ohne Unterscheidung von Groß-/Kleinschreibung stattfindet. Dies ist der Fall, wenn diese Variable einen Wert ungleich 0 enthält. Default: 0

NF Die Anzahl der Felder im aktuellen Record.

NR Die Nummer des aktuellen Records (Zählung beginnt mit 1).

OFMT Das Ausgabeformat für Zahlen. Default: `%.6g`

OFS Der Feldsperator für Ausgaben mittels des `print` Kommandos (siehe dort). Default: Leerzeichen

ORS Der Recordsperator für die Ausgabe mittels `print`. Default: Newline

RLENGTH Hier wird die Länge des Substrings abgelegt, der von der `match` Funktion gematcht wurde (siehe dort.)

RSTART Hier wird die Position des Beginns des Matches bei Aufruf der `match` Funktion abgelegt.

RS Der Recordseperator, also das Zeichen, das Records in der Eingabe trennt. Default: Newline

SUBSEP Das Zeichen, das Indices in Arrayindizierung trennt. Default: ASCII-28

3.3.3 Arrays

`awk` kennt neben skalaren Variablen auch Arrays. Im Unterschied zu **C** sind Arrays in `awk` allerdings *assoziativ*. Das heißt, der Index des Arrays ist nicht eine Zahl, sondern ein String, z.B. `foo["bar"]=99;`
Dies bedeutet, daß die Arrays dadurch realisiert sind, das eine Liste von Paaren (Schlüssel,Wert) abgelegt wird. Dabei ist der Schlüssel der Index. Man kann aber weiterhin Zahlen als Indices verwenden, da `awk` sie auch als Strings auffaßt.
`awk` kennt auch mehrdimensionale Arrays. Allerdings sind diese intern wiederum als eindimensionale Arrays realisiert. Dabei werden die Indices zu einem einzigen Index aneinandergehängt, getrennt durch das Zeichen, welches in der Variablen `SUBSEP` abgelegt ist. Z.B. weisen die beiden Ausdrücke `foo[1,2]=9` und `foo["1" SUBSEP "2"]=9` dem gleichen Element den Wert `9` zu[1].
Bei der Iteration über die Elemente eines Arrays benötigt man nun einen Weg, um alle Schlüssel aufzählen zu können, bzw. einen Test, ob ein bestimmter Schlüssel im Array bereits vorhanden ist. Dazu gibt es eine Sonderform der `for` Schleife (s.u.) und den `in` Operator.
`for (`*`var`*` in `*`array`*`)` iteriert über alle Paare in dem Array *array*, wobei der Wert des Schlüssels im Rumpf der Schleife in der Variablen *var* verfügbar ist. So gibt `for (j in foo) { print foo[j]; }` alle Werte des Arrays `foo` aus.
Der Ausdruck *`index`*` in `*`array`* prüft, ob der Schlüssel *index* in dem Array *array* vorhanden ist. So würde mit dem obigen Beispiel etwa `"bar" in foo` zu 'wahr' (`1`) auswerten.

3.3.4 Operatoren

Operatoren verknüpfen Teilausdrücke zu einem neuen Ausdruck. `awk` kennt die folgenden Operatoren (geordnet nach steigender Priorität):

[1]Man beachte, daß in `awk` String Konkatenation durch das Leerzeichen angegeben wird.

Operator	Beschreibung
=,+=, −=, *=, /=, %=, ^=	Zuweisung
?:	**C** Conditional
||	Logisches Oder
&&	Logisches Und
~, !~	Matchen eines Patterns und Negation davon
<, <=, >, >=, !=, ==	Relationale Operatoren
(Leerzeichen)	String Konkatenation
+, −	Addition, Subtraktion
*, /, %	Multiplikation, Division und Modulo
+, −, !	Unäres Plus, Minus und logische Negation
^	Exponentation
++, −−	Inkrement und Dekrement (Postfix oder Prefix)
$	Feldzugriff

Darüber hinaus kann man Ausdrücke natürlich auch noch klammern, um die Auswertungsreihenfolge zu beeinflussen.
Da die meisten dieser Operatoren denen von **C** entsprechen, werden wir jetzt nur noch auf die `awk`-spezifischen Operatoren genauer eingehen.

Feldzugriff

Der Feldzugriff wurde schon im Abschnitt über Variablen beschrieben.

Exponentation

`awk` kennt, im Gegensatz zu **C**, einen primitiven Operator zum Exponentieren von Zahlen: *`expr`*`^`*`expr`*.
Z.B.: `s=2^4` weist `s` den Wert `16` zu.
Die Zuweisungvariante *`var`*`^=`*`expr`* steht als Abkürzung für *`var`*`=`*`var`*`^`*`expr`* bereit.

Konkatenation

In `awk` werden Strings aneinandergehängt, indem man ein Leerzeichen zwischen ihnen läßt. So ergibt z.B. `"foo" "bar"` den String `"foobar"`.
Da `awk` aber auch Zahlen intern als Strings auffasst, ergibt `s=9 99` das selbe Ergebnis, wie `s="999"` oder `s="9" "99"`

Match

Der Match Operator (`~`) dient zum Test, ob ein String ein Pattern matcht. Er kann in zwei Formen aufreten:

expr* ~ */pattern/ Es wird geprüft, ob der Wert des Ausdruckes auf der linken Seite von dem Pattern der rechten Seite gematcht wird. Falls dies zutrifft, wird der Wert `1` (wahr) zurückgegeben, sonst `0` (falsch).

expr* ~ *expr In dieser Variante wird der rechte Ausdruck ausgewertet und das Ergebnis als Pattern aufgefasst, mit dem dann auf den Match getestet wird (s.o.). Dies erlaubt variable Pattern in Ausdrücken.

Der Operator `!~` testet dann auf das Nicht-Matchen, ist ansonsten aber analog.
Z.B.:

```
if ($1 ~ /^[0-9]+$/) {print "Zahl";}
p=NR*2 ". Satz"; if ($1 ~ p) {print "OK";}
```

3.3.5 Funktionsaufrufe

`awk` kennt eine ganze Reihe vordefinierter Funktionen. Darüber hinaus kann man sich eigene Funktionen definieren (siehe Abschnitt 3.5.1).
Ein Funktionsaufruf sieht in `awk` (erwartungsgemäß) so aus:

$$ret{=}function(arg_1,\ arg_2\ldots)$$

Gibt die Funktion keinen Wert zurück, entfällt natürlich der *ret*= Teil.
In Abschnitt 3.5.2 sind die vordefinierten Funktionen aufgeführt.

3.4 Kommandos

Kommandos in `awk` ähneln stark denen in **C**. Im Unterschied zu **C** muß man allerdings nicht jedes Kommando durch ein Semikolon beenden, kann dies aber tun. Zum Trennen von Kommandos benutzt man entweder ein Semikolon oder einen Zeilenumbruch. Wie in **C** kann man mehrere Kommandos zu einem Block fassen, der dann wie ein einzelnes Kommando behandelt wird, indem man sie in geschweifte Klammern einschließt.
Zuerst ist anzumerken, daß ein Ausdruck auch ein Kommando ist. So ist etwa `9+9; 34*67;` ein vollkommen legales Programmstück. Natürlich wird der Wert eines Ausdruckes, der in dieser Form als Kommando benutzt wird nur berechnet und dann weggeworfen. Der einzige Zweck eines Ausdruckes besteht in den Seiteneffekten, die in ihm auftreten. Dies sind in `awk` nur die Zuweisungen, s.o.

3.4.1 Einfache Kommandos

`awk` kennt die folgenden einfachen Kommandos:

break Dies verläßt eine `for` oder `while` Schleife (s.u.)

continue Beginnt die nächste Iteration einer `for` oder `while` Schleife.

exit Beendet das komplette `awk`-Programm. Wenn vorhanden wird eine `END` Aktion noch ausgeführt.

next Liest den nächsten Record ein und beginnt die Abarbeitung des `awk`-Programmes wieder am Anfang. Die spezielle Variante `next file` springt zum Beginn des nächsten Eingabefiles und liest den ersten Record daraus.

print *expr*[,*expr*...] [*out-expr*] Dient zur Ausgabe von Werten. Die durch Kommata getrennten Ausdrücke werden ausgewertet und nacheinander, durch den String, der in der Variablen `OFS` steht, getrennt ausgegeben. Am Ende wird der String, der in der Variablen `ORS` steht, ausgegeben. Durch den *out-expr* Ausdruck kann man die Ausgabe umleiten. Ist dieser von der Form `>expr`, dann erfolgt die Ausgabe in das File, dessen Namen durch den Ausdruck *expr* gebeben ist[2]. Steht statt dessen etwas der Form `>>expr` wird an das entsprechende File angehängt. Als letztes kann man noch die Form `|expr` verwenden. Hier wird die Ausgabe in eine Pipe geleitet, deren anderes Ende an das Programm geht, welches durch den Ausdruck spezifiziert ist, etwa `| sort -u >nix`.

printf(*f-expr*[, *expr*, ...]) [*out-expr*] Die von **C** her bekannte *printf(3)* Funktion existiert auch in `awk`. Hierbei wird *f-expr* ausgewertet und der resultierende String als eine Formatschablone genommen. Für die Argumente in der Schablone werden dann die ausgewerteten Ausdrücke eingesetzt. Als Formatangaben sind dieselben verfügbar, wie in *printf(3)* beschrieben. Auch dieses Kommando kann einen Umleitung enthalten (*out-expr*), die der des `print` Kommandos entspricht.

return [*expr*] Kehrt aus einen Funktion zurück und gibt ggf. den Wert des angegebenen Ausdruckes zurück.

[2]Dieser Ausdruck muß ein String sein. Man kann also nicht `>foobar` schreiben, sondern muß `>"foobar"` verwenden.

3.4.2 Konditional & Schleifen

Neben den einfachen Kommandos des vorherigen Abschnittes kennt `awk` auch noch ein `if-then-else` Konditional und vier Schleifenkonstrukte: zwei Varianten einer `for`-Schleife und zwei Varianten der `while`-Schleife:

if (*expr*) *stat* Der Ausdruck *expr* wird ausgewertet, wenn sein Wert `1` ergibt, wird das Kommando *stat* ausgeführt.

if (*expr*) $stat_1$ else $stat_2$ Der Ausdruck *expr* wird ausgewertet. Ergibt dies `1`, wird das Kommando $stat_1$ ausgeführt, sonst das Kommando $stat_2$.

do *stat* while (*expr*) Die Anweisung *stat* wird ausgeführt. Danach wird der Ausdruck *expr* ausgewertet. Ist das Ergebnis `1`, dann beginnt die Schleife von vorne. Sonst wird die Schleife verlassen.

while (*expr*) *stat* Der Ausdruck *expr* wird ausgewertet. Ist das Ergbnis `1`, dann wird die Anweisung *stat* ausgeführt und die Schleife beginnt danach von vorne. Andernfalls wird die Schleife sofort verlassen.

for (*i-expr*;*c-expr*;*l-expr*) *stat* Zuerst wird der Ausdruck *i-expr* ausgewertet. Danach wird der Ausdruck *c-expr* ausgewertet. Ist dessen Wert `0`, wird die Schleife verlassen. Anderenfalls wird das Kommando *stat* ausgeführt und danach der Ausdruck *l-expr* ausgewertet und der nächste Durchlauf der Schleife beginnt wieder mit der Auswertung des Ausdruckes *c-expr*.

Der Ausdruck *i-expr* dient zur Initialisierung der Schleife, etwa eines Schleifenzählers. Der Ausdruck *c-expr* ist die Schleifenbedingung. Der Ausdruck *l-expr* dient zum Weiterschalten von Schleifenzählern. Z.B.:

```
for (i=1;i<=NF;i++)
  printf("Feld %d hat den Wert %s\n",i,$i);
```

Diese Schleife gibt alle Felder des aktuellen Records aus.

for (*var* in *array*) *stat* Diese Variante dient zur Aufzählung aller Schlüssel des assoziativen Arrays *array*. Im Schleifenrumpf (*stat*) steht der jeweilige Schlüssel in der Variablen *var* zur Verfügung. Siehe Abschnitt 3.3.3.

In Schleifen können die Anweisungen `break` und `continue` auftreten. `break` bewirkt dabei, daß die innerste Schleife (in welcher sie steht), sofort verlassen wird. Die Programmausführung wird mit dem Kommando nach dem Schleifenrumpf fortgesetzt.
Das `continue` Kommando beginnt sofort die nächste Schleifeniteration, springt also in einer `while`-Schleife zur Auswertung der Schleifenbedingung. In einer

for-Schleife wird zuerst noch der *l-expr* Ausdruck ausgewertet, dann zur Auswertung der Schleifenbedingung gesprungen.

3.5 Funktionen

3.5.1 Benutzerdefinierte Funktionen

In awk kann man eigene Funktionen definieren. Diese werden im Programm an derselben Stelle definiert, wo sonst die Adressen/Programmstück Paare stehen (also nicht innerhalb eines Programmstückes). Die Form der Definition:

function *name*(*param*$_1$,...,*param*$_N$) { *stats* }

Als Beispiel folgendes kleines Programm:

```
function foo(bar, baz) {
  printf("Der %d. Parameter ist %s\n",bar,baz);
}
BEGIN { for (i=0;i<ARGC;i++) foo(i,ARGV[i]);}
```

Dieses Programm definiert eine Funktion foo, die einfach etwas Text mittels printf ausgibt und ihre Argumente an zwei Stellen einsetzt. Als Resultat dieses Programmes werden alle Kommandozeilenparameter von awk ausgegeben.

Innerhalb einer Funktion kann man mittels des return Kommandos Werte zurückgeben.

3.5.2 Eingebaute Funktionen

awk verfügt über eine ganze Reihe von eingebauten Funktionen, die wir hier alphabetisch geordnet auflisten:

atan2(x,y) Gibt den Wert des arcus tangens von y/x (in Radianten) zurück.

close(*f-expr*) Wir hatten bei dem print Kommando gesehen, daß man Ausgaben in Files umlenken kann. awk erlaubt aber nur (etwa) zehn offene Files. Um Files explizit wieder zu schliessen, ruft man die close Funktion mit dem Filenamen des geöffneten Files auf. Hatte man die Ausgabe in eine Pipe zu einem Programm geleitet, muß der Ausdruck zu "|*Programm*" auswerten, um diese Pipe zu schliessen.

cos(x) Gibt den Wert des Cosinus von x (in Radianten) zurück.

delete *array*[*element*] Diese Funktion löscht das angegebene Element aus dem Array. Nützlich, um Kommandozeilenparameter zu elimieren, siehe Abschnitt 3.6. Beispiele: delete foo[1],delete ARGV[i]

exp(x**)** Gibt den Wert von e^x zurück.

gsub(r**,**s**,**t**)** Diese Funktion ersetzt alle Vorkommen des Patterns r durch den String s. Die Ersetzung findet in dem String t statt. Das Pattern kann entweder als `/pattern/` oder als Variablennamen gegeben sein. In letzterm Fall wird der Inhalt der Variablen als Pattern benutzt. Man kann den String t weglassen, in welchem Fall `$0`, also der aktuelle Record, benutzt wird. Das Ergebnis dieser Funktion ist *nicht* das Resultat der Ersetzung, sondern die Anzahl Ersetzungen, die gemacht wurden. Das Resultat der Ersetzung steht in t selbst.

index(s**,** t**)** Liefert die Position des Substrings t in dem String s. Die Zählung beginnt dabei mit der `1`: `index("laber","aber")`=2.

int(x**)** Wandelt die Zahl x zu einer Ganzzahl um, indem die Nachkommastellen abgeschnitten werden.

length(s**)** Gibt die Länge des Strings s zurück.

log(x**)** Errechnet ln x.

match(s**,**r**)** Diese Funktion versucht, das Pattern r (in der gleichen Form, wie bei der `gsub` Funktion) in dem String s zu matchen. Der Rückgabewert ist `0`, falls kein Match existiert, oder die Position des Beginns des Matches. Diese Position wird auch in der Variablen `RSTART` abgelegt. Die Länge des Matches wird in der Variablen `RLENGTH` abgelegt. So liefert also z.B. `match("aber 100 Tiere",/[0-9]+/)` den Wert `6`. In `RLENGTH` wird der Wert `3` abgelegt.

rand() Diese Funktion liefert eine Pseudo-Zufallszahl im Bereich $[0, 1]$ zurück. Man erhält bei jedem Programmaufruf immer die gleiche Folge von Zahlen, deshalb sollte man die `srand` Funktion zum Setzen eines anderen Startwertes benutzen (s.u.).

sin(x**)** Liefert den Sinus von x (in Radianten) zurück.

split(s**,*array*,*sep*)** Diese Funktion spaltet einen String in Felder auf, die durch die im String *sep* stehende Zeichenfolge getrennt sind. Die einzelnen Felder werden der Reihe nach in dem Array *array* abgelegt. Fehlt der String *sep*, wird der Inhalt der Variablen `FS` benutzt. Die Funktion gibt die Anzahl Felder zurück, die in *array* abgelegt wurden. Z.B. liefert `split("a::b::c::d",foo,"::")` den Wert `4` zurück. `foo` enthält nachher die Elemente `foo[1]="a"`, `foo[2]="b"`, etc.

sprintf(*f-expr*[,*expr*...] Diese Funktion ähnelt dem `printf` Kommando, nur gibt sie ihr Ergebnis nicht aus, sondern liefert es als String zurück.

sqrt(x) Diese Funktion gibt die Quadratwurzel von x zurück.

srand(*expr*) Diese Funktion setzt den Startwert des Zufallszahlengenerators auf den Wert von *expr*.

sub(r,s,t) Diese Funktion ähnelt der Funktion `gsub` mit dem einzigen Unterschied, daß nur das erste Auftreten des Patterns ersetzt wird.

substr(s,b,l) Diese Funktion liefert den Teilstring des Strings s zurück, der an Position b startet und l Zeichen lang ist. Die Position zählt dabei von `1` an. Wird l weggelassen, wird der gesamte Rest des Strings von Position b an zurückgegeben.

system(*expr*) Führt das in dem String *expr* angegebene Programm in einer Shell aus. Die Funktion gibt den Exit-Code des Programmes zurück, oder `-1`, falls bei der Ausführung Fehler auftraten. Die Ausgaben des Programmes landen in `/dev/null` und können nicht im `awk`-Programm benutzt werden.

tolower(s) Diese Funktion wandelt alle Großbuchstaben in dem String s in Kleinbuchstaben und gibt dann den resultierenden String zurück.

toupper(s) Diese Funktion wandelt alle Kleinbuchstaben in s in Großbuchstaben und gibt das Ergebnis zurück.

3.6 Sonstiges

3.6.1 Kommandozeilenparameter

`awk` kennt, ähnlich `sed`, die **-f** Option, mit der man ein Programm zur Ausführung angeben kann. Darüberhinaus gibt es aber noch einige Optionen:

-Fc Setzt das Trennzeichen für Felder (`FS`) auf das Zeichen c.

-f *Prog* Nimmt das Programm im File *Prog* zur Ausführung.

var=value Weist der Variablen *var* den Wert *value* zu. Die Zuweisung erfolgt vor der Abarbeitung des ersten Kommandos im Programm (also auch vor einem `BEGIN` Stück).

Der erste Kommandozeilenparameter, der keine Option ist, wird als Programm aufgefasst (nur wenn keine **-f** Option gegeben wurde). Alle weiteren Parameter werden als die Namen von Files aufgefasst, welche bearbeitet werden sollen.
Man hat hier aber ein Problem: `awk` bietet das assoziative Array `ARGV`, in welchem die Kommandozeilenparameter (ausser den oben genannten), gespeichert sind. Manchmal will man noch zusätzliche Optionen für das `awk`-Programm mitgeben, die nur von diesem interpretiert werden sollen. Dabei dürfen diese aber nicht als die Namen von Files aufgefasst werden, welche zur Bearbeitung geöffnet werden sollen. `awk` fasst daher den Inhalt des `ARGV` Arrays *nach* Abarbeitung des `BEGIN` Programmstücks als Namen der Eingabefiles auf. Will man eigene Parameter benutzen, muß man diese in dem `BEGIN` Stück aus `ARGV` entfernen, z.B.:

```
BEGIN{ for (i=1;i<ARGC;i++) {
          if (ARGV[i] ~ /--ignore/) {
             iflag=1;
             delete ARGV[i];
          }
       }
}
```

Dieses Programmstück sucht nach der Option `--ignore` und setzt, falls sie gegeben wurde, die Variable `iflag` auf `1` und löscht dann diese Option aus den Kommandozeilenparametern.

3.6.2 Spezielles

Wie oben erwähnt, kann man die Variable `RS` auf einen String setzen, der Records in der Eingabe trennt. Setzt man diese Variable auf den leeren String, `RS=""`, dann steht dies für eine Leerzeile als Trennung zwischen Records. So kann man z.B. unter Zuhilfenahme von `FS="\n"`, einen ganzen Absatz in einen Record lesen, wobei die einzelnen Felder jetzt aus den Zeilen des Absatzes bestehen.
`awk` kennt eine Funktion, `getline`, mit der man den nächsten Record einlesen und im Programm weiter fortfahren kann. Diese kommt in zwei Varianten:

getline [***var***] [<***file***] Dies nächste Zeile wird aus dem File *file* gelesen (oder dem aktuellen Eingabefile) und in Felder aufgespalten. Das Ergebnis wird in `$`i und `$0` abgelegt, `NR`, `NF` und `FNR` werden gesetzt. Wird eine Variable *var* angegeben, wird der neue Record nicht in `$0` abgelegt, sondern in dieser Variablen. Diese Funktion liefert `0` als Ergebnis, wenn kein neuer Record mehr in dem File steht, `1`, wenn ein Record gelesen werden konnte und sonst `-1`.

command | **getline** [*var*] Diese Variante liest nicht aus einem File, sondern führt das angegebene Programm *command* aus und liest dessen Ausgabe ein. Ansonsten identisch zur ersten Version.

Es sei angemerkt, daß das angegebene File nur bei der ersten Ausführung der Funktion geöffnet wird. Weitere Ausführungen der Funktion (mit demselben File natürlich), benutzen das bereits geöffnete File und lesen die nächsten Records. Ebenso wird das Programm bei der zweiten Variante nur bei der ersten Ausführung gestartet und die Ausgabe dann bei den folgenden Aufrufen Recordweise gelesen.

3.7 Beispiele

3.7.1 Filelängen aufsummieren

Angenommen, man möchte in einem Directory die Längen bestimmter Files aufsummieren, etwa die aller Java-Programme. Dazu kann man folgendes in die Shell tippen[3]:

```
ls -l `find . -name '*.java'`|
  awk '{summe+=$5}
       END{ printf("Summe: %d bytes\n",summe);}'
```

Das `ls` Kommando listet alle Files auf, deren Namen mit `.java` enden (die Namen werden vom `find` Kommando geliefert). Im fünften Feld steht dabei in der Ausgabe die Filelänge. Das `awk` Programm summiert nun in jeder Zeile diese Länge auf und gibt am Schluß die Summe aus.

3.7.2 Sortieren

Ein Weg, Files (auch über Unterdirectorygrenzen hinweg) nach ihrer Länge sortiert auszugeben bietet das folgende Konstrukt:

```
ls -R -l .| awk '{printf("%08d %s\n",$5,$0)}'|
 sort|
 awk '{for (i=2;i<=NF;i++)
        printf("%s%s",(i==2)?"":" ",$i);
       printf("\n");}'
```

[3]Hier nur der Lesbarkeit wegen auf mehrer Zeilen verteilt.

Diese Konstruktion listet erst alle Files mittels `ls`, danach wird über ein `awk`-Programm die Filelänge vor jeden Eintrag geschrieben, und zwar mit führenden Nullen aufgefüllt. Danach wird diese Ausgabe mittels `sort` sortiert, das jetzt nach dem ersten Feld sortiert, der normalisierten Filelänge. Danach wird in einem zweiten `awk`-Programm die führende Filelänge wieder entfernt, indem einfach nur die Felder ab dem zweiten Feld ausgegeben werden.
Etwas unschön ist, das die Namen der Files in der Ausgabe nicht mit vollem Pfad stehen, sondern nur mit einfachem Namen. Das Auffinden gesuchter Files wird dadurch schwerer. Abhilfe schafft, statt des ersten `ls` Kommandos das folgende einzugeben: `ls -l `find . -type f``.

3.7.3 Statistik

In der Organisation eines Workshops reichten die Gutachter ihre Bewertungen über die Paper elektronisch ein. Dabei wurden, neben den textuellen Reviews, u.a. Benotungen über die allgemeine Qualität (`Overall`) oder die technische Qualität des Papers (`Technical`) abgegeben (Noten von 0-9). Für jedes Paper wurden diese Gutachten nun in einem File gesammelt. Zum Schluß sollten zur Auswertung und Einstufung der Paper diese Benotungen ausgewertet werden: Sowohl der Durchschnitt, als auch die Standardabweichung sollten für jedes Paper ausgegeben werden. Dabei waren die Datenfiles so ausgelegt, daß die Bewertungen in Zeilen der Form `%%Overall:` n_o bzw. `%%Technical:` n_t standen.
Dazu wurden ein Shell-Script und ein `awk`-Programm verwandt:

```
#! /bin/sh

echo Auswertung der Paper
for p in Papers/*; do
 echo Paper  `basename $p`
 awk -f auswert.awk $p
 echo
done

# auswert.awk
/^%%Overall:/{overall+=$2; anzahl++; ov[anzahl]=$2;}
/^%%Technical:/{technical+=$2; te[anzahl]=$2;}
END{ durcho=overall/anzahl; durcht=technical/anzahl;
     for (i=1;i<=anzahl;i++) {
        abweicho+=(ov[i]-durcho)^2;
        abweicht+=(te[i]-durcht)^2;
     }
```

```
    abweicho=sqrt(abweicho);
    abweicht=sqrt(abweicht);

    printf("Durchschnitte:\n");
    printf("\tOverall: %2.2f\n\tTechnical: %2.2f\n",
           durcho,durcht);
    printf("StdAbweichungen:\n");
    printf("\tOverall: %2.2f\n\tTechnical: %2.2f\n",
           abweicho, abweicht);
}
```

Das Shell-Script iteriert einfach über die einzelnen Datenfiles und ruft für jedes das `awk`-Programm auf. Das `awk`-Programm summiert in jeder Zeile, die eine Bewertung enthält diese auf (die Anzahl der Bewertungen wird nur bei der `Overall` Bewertung erhöht, die immer zuerst kommt). Am Ende werden dann der Durchschnitt und die Standardabweichung berechnet und ausgegeben.

3.8 Falltüren

In `awk` gibt es meiner Erfahrung nach weniger Falltüren, als in `sed`. Natürlich kann auch hier das Matching von Pattern verblüffen (siehe Beschreibung bei `sed`).
Eine immer mal wieder auftauchende Tretmine ist die Tatsache, daß ein adressloses Programmstück immer ausgeführt wird, auch wenn eine Zeile vorher schon von anderen Adressen gematcht und in deren Programmstücken bearbeitet wurde. Abhilfe schafft hier die Verwendung des `next` Befehls am Ende der spezielleren Programmstücke.
Die automatische Konvertierung von Strings zu Zahlen und zurück kann gewöhnungsbedürftig sein. Auch die assoziativen Arrays sorgen am Anfang für einige Überraschungen, insbesondere in Verbindung mit der automatischen String/Zahl-Konvertierung.
Man sollte aufpassen, sich bei Variablennamen nicht zu vertippen. Dies hat nämlich keine Fehlermeldung zur Folge, sondern es wird eine neue Variable angelegt. Natürlich ist der Effekt dann nicht ganz der gewünschte.
Eine extrem lästige Falltür ist die sich um `awk` rankende Versionitis. Es gibt mindestens drei verschiedene Versionen von `awk`, die sich alle in den unterstützen Features unterscheiden (`awk`, `nawk` und GNU `awk`). Allgemein kann man sagen, daß GNU `awk` die meisten Features bietet und dazu noch frei verfügbar ist. Deshalb haben wir uns hier nur auf diese Version beschränkt. Sollte man auf andere

Versionen von awk treffen, sollte man immer daran denken, daß sie vielleicht nur einen Teil der hier erwähnten Features bieten.

3.9 Literatur

Auf für awk empfiehlt sich das schon im sed Kapitel erwähnte Buch [Dou90]. Ebenso sollte man sich die Manual-Pages von awk zu Gemüte führen.

Kapitel 4

`perl`: Practical Extraction and Report Language

4.1 Einführung

`perl` stellt eine weitere Programmiersprache dar. Sie ist aus der Erkenntnis entstanden, daß man in `awk` einige Dinge nur schwer tun kann, da `awk` zu stark an das Konzept der Bearbeitung von Records gebunden ist. Andererseits sind z.B. in **C** Pattern-Matching Operationen schwierig. In der Shell hingegen ist richtiges Rechnen wiederum nicht effizient möglich.
Deshalb wurde `perl` als ein Gemisch von Konzepten aus allen diesen Programmen entworfen. Darin liegt aber, jedenfalls in meinen Augen, auch eine seiner größten Schwächen: Die Konzepte sind im Gemisch teilweise sehr unübersichtlich. Man sollte daher darauf verzichten, `perl` als Sprache für größere Projekte zu verwenden[1]. Auch die in der `perl`-Version 5 aufgestülpten Objektorientierten Features ändern an der Ungeeignetheit für größere Projekte nichts.
In diesem Kapitel werden zunächst die grundlegenden Entitäten, Variablen, Ausdrücke und Kommandos erläutert. Danach werden die von `perl` verwendeten Pattern, die sich stark von den `awk`-Pattern unterscheiden, vorgestellt. Die in `perl` vorhandene Möglichkeit, Unterprogramme zu definieren wird in Abschnitt 4.5 diskutiert. Im darauf folgenden Abschnitt werden einige spezielle Variablen aufgelistet. Abschnitt 4.7 stellt die Verwendung von `perl` zur Realisierung von CGI-Scripten vor. Zum Abschluß folgen noch einige Beispiele und Warnungen zur Verwendung von `perl`.

[1]"Write once - read never"

4.2 Variablen & Ausdrücke

Für Variablen `perl` kennt drei verschiedene Datentypen:

1. Skalare. Ein Skalar kann entweder ein String sein (`"Ein String"`) oder eine Zahl. Ähnlich wie `awk` konvertiert `perl` dabei automatisch Strings zu Zahlen und umgekehrt, je nachdem, was es an der jeweiligen Stelle braucht. Eine Zahl wird in `perl` immer als eine Fließkommazahl mit doppelter Genauigkeit gespeichert.

2. Vektoren von Skalaren. Vektoren sind Listen von Skalaren, also Arrays, die mit Zahlen indiziert werden.

3. Assoziative Vektoren von Skalaren. Dies sind auch Arrays, nur das sie (wie in `awk`) assoziativ sind, also durch Strings indiziert werden.

Als Typ eines Ausdruckes kann allerdings immer nur 'Skalar' oder 'Liste von Skalaren' auftreten. Vektoren und assoziative Vektoren werden dabei zu Listen konvertiert, indem ihre Werte aufgezählt werden, bzw. die Aufzählung in der Form $\mathit{Schlüssel}_1$, Wert_1, $\mathit{Schlüssel}_2$, Wert_2, ... vorgenommen wird. Z.B. wertet ein assoziativer Vektor, der den Schlüssel `"a"` mit dem Wert `19` und den Schlüssel `"b"` mit dem Wert `42` verknüpft zu der Liste `"a",19,"b",42` aus. Ein normaler Vektor, welcher dem Index `1` den Wert `19` und dem Index `2` den Wert `42` zuordnet, wertet zu der Liste `19,42` aus.
Die Namen von Variablen geben Auskunft über ihren Typ:

- Skalare Variablen sind von der Form `$`*Name*

- Variablen, die Vektoren von Skalaren sind, haben die Form `@`*Name*

- Variablen, die assoziative Vektoren sind, schreibt man in der Form `%`*Name*

Ein *Name* ist dabei definiert als ein Buchstabe oder Unterstrich, gefolgt von Buchstaben, Ziffern oder Unterstrichen.
Gewöhnungsbedürftig ist, daß in `perl` z.B. Zugriffe auf ein Element eines Vektors als `$foo[1]` geschrieben werden (was das Element `1` des Vektors `@foo` referenziert)[2]. Zugriffe auf Elemente eines assoziativen Vektors werden als `$bar{"a"}` geschrieben (referenziert das Element des assoziativen Vektors `%bar%`, welches den Schlüssel `"a"` hat).
Eine weitere Kurzschreibweise in `perl` ist, daß man mehrere Indices in eine Referenz schreiben kann, z.B.:

[2] Im Unterschied zum vielleicht erwartetem `@foo[1]`

@foo[1,2,3] Dies steht als eine nützliche Abkürzung für die folgende Liste `($foo[1], $foo[2], $foo[3])`.

@bar{"a","b"} Steht für `($bar{"a"}, $bar{"b"})`. Man beachte eine weitere Inkonsistenz: `%bar{"a","b"}` ist verkehrt!

@foo[1..3] Steht für `($foo[1], $foo[2], $foo[3])`.

Den letzten gültigen Index in einem nicht-assoziativen Vektor `@foo` schreibt man als `$#foo`. Es sei bemerkt, daß `perl` per Default den Wert `0` als ersten Index in solch einem Array ansieht[3].

4.2.1 Skalare

Ein Skalar ist entweder eine Zahl, oder ein String. Zahlen können dezimal '`3.1415`', octal '`0755`' ooder hexadezimal '`0x80`' angegeben werden.
Strings kommen in `perl` in insgesamt sieben (nein, kein Tippfehler) Varianten:

- Man kann den String als Zeichenkette angeben, welche durch einfache Anführungszeichen (`'`) eingeschlossen ist: `'Ein String'`
- Dann kann man eine Zeichenkette angeben, die durch doppelte Anführungszeichen (`"`) umschlossen wird: `"Ein $wert String"`. In dieser Zeichenkette findet Variablenersetzung statt[4]. Hat z.B. die Variable `$foo` den Wert `9999`, so wertet der String `"Dies sind $foo Zeichen"` zu dem String `'Dies sind 9999 Zeichen'` aus. Dies funktioniert auch mit komplizierteren Zugriffen, so wertet etwa `"6*9 = $bar{'2'}"` zu `'6*9 = 42'` aus.
- Ähnlich wie in der Shell kann man `perl` anweisen, ein Kommando in einer Subshell auszuführen und die Ausgabe dieses Kommandos als String zu interpretieren. Dazu muß man das auszuführende Kommando in Rückanführungszeichen (`` ` ``) einschliessen: `` `ls -l` `` könnte dann z.B. zu dem String `'ccRcxIrM.i   ssh-thesing'` auswerten.

Will man viel Text in einen String packen, ggf. mit einer Menge Zeilenumbrüche, dann bietet es sich an, die[5] '<<' Syntax zu benutzen. Dabei wird der Text beginnend mit der nächsten Zeile, bis zum Auftreten eines bestimmten Begrenzers als String genommen. Z.B.

---

[3] Dies kann aber durch eine Zuweisung and eine spezielle Variable geändert werden: `$[=1` setzt den ersten Index eines Vektors auf 1.

[4] Ähnlich wie in der Shell.

[5] Ebenfalls von der Shell her bekannte.

```
$foo=<<EOM;
Dies ist
ein Text
EOM
```

Dies liefert dasselbe, wie `$foo='Dies ist\nein Text\n';` Man beachte, daß der Begrenzer am Anfang einer Zeile auftreten muß, damit er als Begrenzer erkannt wird.
`perl` kennt nun vier Möglichkeiten, solche Sammlungen vorzunehmen:

- Als Begrenzer (wie `EOM` in unserem Beispiel) wird ein einzelnes Wort verwandt. Dies entspricht der unten erläuterten Angabe von `"EOM"` als Begrenzer.
- Als Begrenzer nimmt man eine Zeichenfolge[6], welche man in einfachen Anführungszeichen (`'`) einschliesst, z.B. `<<'EOM'`. Dies entspricht dann den einfachen Strings, d.h. der Text wird genauso gesammelt und zu einem String gemacht, wie er folgt.
- Als Begrenzer wird eine Zeichenfolge eingesetzt, die man in doppelten Anführungszeichen einschliesst, z.B. `<<"EOM"`. Dies entspricht den Strings, welche in doppelten Anführungszeichen eingeschlossen sind, d.h. in dem gesammelten Text wird ebenfalls Variablenersetzung durchgeführt.
- Endlich kann man auch die Rückanführungszeichen (`` ` ``) verwenden, z.B. ``<<`EOM` ``. Dann wird jede Zeile des folgenden Textes als ein Kommando aufgefasst, welches dann ausgeführt wird. Die Ausgaben aller dieser Kommandos (bis zum Begrenzer) werden dann aneinandergehängt und als Strings genommen.

Man beachte, daß diese Sammlungen immer an der Stelle (und für) eines Strings stehen. Z.B. muß man im obigen Beispiel erst die Zuweisung mit einem Semikolon beenden, bevor der zu sammelnde Text beginnt. Man kann auch mehrere Sammlungen in einem Kommando haben, z.B. `print <<A <<B;`. Diese müssen dann in der Reihenfolge ihres Auftretens angegeben werden.
In `perl` kann man Steuerzeichen in Zeichenketten einsetzen, indem man die in Tabelle 4.1 aufgelisteten Escape-Sequenzen verwendet.
Eine verwirrende Besonderheit von `perl` ist, daß ein Ausdruck auch den Wert 'undefiniert' haben kann. Dieser wertet immer zu `0` (bzw. `""`) aus, und repräsentiert einen Fehler. Man kann mittels der `defined` Funktion prüfen, ob ein Ausdruck definiert ist, oder nicht:

```
if (defined(open(FH,"/tmp/xx")))....
```

[6] Welche auch Leerzeichen enthalten kann.

Sequenz	Bedeutung
`\n`	Newline
`\r`	Carriage-Return
`\t`	TAB
`\f`	Form-Feed
`\b`	Backspace
`\o`*oo*	ASCII Code, der durch die Oktalzahl *oo* gegeben ist.
`\x`xx	ASCII Code, der durch die Hexzahl xx gegeben ist.
`\u`	Das folgende Zeichen wird als Großbuchstabe ausgegeben.
`\l`	Das folgende Zeichen wird als Kleinbuchstabe ausgegeben.
`\U`	Die Zeichen bis zum `\E` werden in Großbuchstaben ausgegeben.
`\L`	Die Zeichen bis zum `\E` werden in Kleinbuchstaben ausgegeben.
`\E`	Beendet `\U` oder `\L` Bereich.

Tabelle 4.1: Escape-Sequenzen in `perl`.

4.2.2 Listen

Listen sind von der Form i_1, i_2, ... Wie oben gesagt, werten Vektoren zu Listen aus.

4.2.3 Wahrheit

`perl` kennt, ähnlich wie **C**, keinen Datentyp `bool`. An Stellen, an denen ein Wahrheitswert nötig ist (etwa in einer `if` Anweisung), wird das Argument ausgewertet und dann folgendermassen interpretiert:

- Eine Zahl, die nicht `0` ist, wird als 'wahr' interpretiert. Die `0` als 'falsch'.
- Der leere String (`""`) wird als 'falsch' interpretiert, genauso wie die Zeichenkette `"0"`. Jede andere nicht-leere Zeichenkette wird als 'wahr' interpretiert[7].
- Der undefinierte Wert wird als 'falsch' interpretiert.

[7] Die Zeichenkette `"00"` wird als 'wahr' interpretiert....`perl` halt:-)

- Ein Vektor wird als 'wahr' interpretiert, wenn er nicht leer ist (auch assoziative Vektoren), ansonsten als 'falsch'.

4.2.4 Kontexte

In perl gibt es Operatoren, die sowohl einen Skalar, als auch eine Liste zurückliefern, bzw. als Argument nehmen können. Was jetzt wirklich geliefert wird, wird durch den *Kontext* eines Ausdruckes entschieden. Verlangt dieser Kontext eine Liste, wird eine Liste geliefert, sonst ein Skalar. Etwa kann man mittels des Ausdruckes <STDIN> aus der Standardeingabe lesen; es gibt aber Unterschiede im Bezug auf den vorhandenen Kontext:

@foo=<STDIN>; Listenkontext. Liest alle Zeilen aus der Eingabe und legt sie als Elemente in dem Vektor @foo ab.

$baz=<STDIN>; Skalarkontext. Liest die nächste Zeile aus der Eingabe und legt sie in der skalaren Variablen $baz ab.

Es sei bemerkt, das Kontexte des öfteren zu Verwirrung führen können.

4.2.5 Filehandle

Eigentlich gibt es außer Variablen und Ausdrücken noch eine dritte Entität in perl: Filehandles. Diese stehen für geöffnete Files und werden etwa in print oder printf Operatoren gebraucht, um Ausgaben auf Files umzuleiten.
In perl werden Filehandles durch Namen dargestellt (also ohne $, @ oder %, wie bei Variablen). Nach der Tradition enthalten Filehandlenamen nur Großbuchstaben. In perl sind drei Filehandle vordefiniert: STDIN, STDOUT und STDERR, welche der Standardeingabe, Standardausgabe und der Fehlerausgabe eines Unix Prozesses entsprechen.
Um z.B. eine Zeile von der Standardeingabe in eine Variable zu lesen schreibt man: $foo=<STDIN>;
Die umschließenden '<','>' bedeuten dabei das Lesen aus dem eingeschlossenen Filehandle. Wie oben gesagt, ist diese Operation kontext-abhängig: In einem skalaren Kontext (wie im Beispiel) wird eine Zeile eingelesen, in einem Vektor-Kontext, wie etwa @foo=<STDIN>; werden alle Zeilen eingelesen und als Elemente des Vektors abgelegt.
perl bietet drei Möglichkeiten, um Filehandles zu manipulieren:

open(*File*, *Expr*) Öffnet ein File, dessen Name durch *Expr* gegeben ist und weist es dem Filehandle *File* zu. Wenn der Filename mit '<' beginnt, dann wird das File zur Eingabe geöffnet. Beginnt er mit '>', dann wird das File zur Ausgabe geöffnet. Bei einem vorstehenden '>>' wird das File zum Anhängen geöffnet. Man kann vor < und > noch ein '+' setzen, wenn man das File zum Lesen und Schreiben öffnen will. Der spezielle Filename '-' steht für `STDIN`, '>-' für `STDOUT`. Eine weitere nützliche Möglichkeit ist es, Filenamen der Form |*`Pipe`* zu verwenden. Dann wird das File zur Ausgabe geöffnet und die Ausgabe wird als Eingabe zu dem Programm *Pipe* gegeben, das in einer Subshell ausgeführt wird. Andererseits kann man auch *`Pipe`*| verwenden, dann wird das File zum Lesen geöffnet und die Eingaben sind die Ausgaben des Programmes *Pipe*, welches in einer Subshell gestartet wird.

Einige Beispiele:

```
open(PASSWD,"</etc/passwd");
open(OUTH,">/tmp/output");
open(INOUT,"+>/tmp/data");
open(MAILH,"|/usr/bin/mail");
```

`open` gibt `1` zurück, wenn das File geöffnet werden konnte, ansonsten den undefinierten Wert (s.o.).

close(*File*) Schließt das File, das durch das Handle *File* beschrieben wird.

<*File*> Dient zur Eingabe aus einem File. s.o.

4.2.6 Operatoren

`perl` kennt die in Tabelle 4.2 aufgelisteten Operatoren, von denen eine Menge der Sprache **C** entliehen sind, um aus Teilausdrücken neue Ausdrücke zu formen. Die Tabelle ist nach steigender Priorität geordnet und enthält auch die Assoziativität der Operatoren.
Wir werden die von **C** her bekannten Operatoren nicht oder nur kurz besprechen; wir beschränken uns haupsächlich auf die spezielleren Operatoren von `perl`.
Die komplexeren List-Operatoren, unären Operatoren und die Operatoren zur Dateiprüfung werden in eigenen Abschnitten besprochen. Hier beginnen wir als erstes mit dem Kommaoperatoren, z.B. `$foo[1]=99, $summe=0;` Der Kommaoperator wertet sein linkes Argument aus, verwirft den Wert (so daß nur die darin auftretenden Nebeneffekte interessant sind). Danach wird der rechte Operand ausgewertet und als Wert des gesamten Ausdruckes genommen. Diese Art von Operator ist auch in **C** vorhanden. In `perl` besteht die Gefahr der Verwechslung mit einer Liste: Je nach Kontext kann der Ausdruck `1,2` eine Liste mit zwei Einträgen darstellen, oder eine Anwendung des Kommaoperators sein[8].

[8]Ich hatte schon erwähnt, daß `perl` aus vermengten Konzepten besteht? Gut.

Operator	Assoziativität	Bemerkung
Listen-Operatoren	keine	Siehe Abschnitt 4.2.7
,	links	Kommaoperator
=, +=, -=, *=, /=, %=, **=, .=, x=, <<=, >>=, &=, |=, ^=	rechts	Zuweisungsoperatoren.
?:	recht	Conditional
..	keine	Bereichsoperator
||	links	Logisches Oder
&&	links	Logisches Und
|, ^	links	Bitweises Oder, Exklusiv-Oder
&	links	Bitweises Und
==, !=, <=>, eq, ne, cmp	keine	Gleichheit auf Zahlen und Strings
<, >, <=, >=, lt, gt, le, ge	keine	Vergleiche auf Zahlen und Strings
unäre Operatoren	keine	Siehe Abschnitt 4.2.8
Dateiprüfung	keine	Dateiprüfung, siehe in Abschnitt 4.2.9
<<, >>	links	Bit-Shift links, rechts
+, -, .	links	Addition, Subtraktion und String-konkatenation
*, /, %, x	links	Multiplikation, Division, Modulo und Stringreplizierung
=~, !~	links	Matching-Operatoren
!, ~, *unäres Minus*	rechts	Logische sowie bitweise Negation, Vorzeichen
**	rechts	Potenzierung
++, --	keine	Inkrement, Dekrement
(	links	Die öffnende Klammer eines Funktionsaufrufes

Tabelle 4.2: Operatoren in perl.

Den Bereichsoperator `..` hatten wir schon kennengelernt. Dieser Operator ist ein Beispiel für einen Operator, der vom Kontext abhängend entweder eine Liste liefert, oder eine Zahl. Wenn der Operator (wie im obigen Beispiel) in einem Listen-Kontext verwandt wird, liefert er die Liste, die aus den aufgezählten Werten von seinem linken bis zu seinem rechten Operand reicht, z.B.: `(1..4)` wertet aus zu `(1,2,3,4)`. Dies funktioniert auch, wenn die Operanden Zeichen sind, etwa `("a".."c")=("a","b","c")`.
Wird der Operator aber in einem skalaren Kontext verwandt (etwa wie in `if (1..10)`), dann verhält er sich äußerst merkwürdig. Er simuliert nämlich die von `sed` und `awk` her bekannten Adressbereiche. Dies geschieht dadurch, daß für jedes Auftreten dieses Operators ein internes Flag gespeichert wird. Bei jeder Auswertung dieses Operators geschieht nun folgendes: Ist das Flag 'falsch', dann bleibt es falsch, wenn der linke Operand zu 'falsch' auswertet. Wertet er zu 'wahr' aus, dann wird das Flag auf wahr gesetzt. Der Wert des Flags wird zurückgeliefert. Ist das Flag 'wahr', dann wird bei jeder Auswertung des Operators geprüft, ob der rechte Operand zu 'wahr' auswertet. Ist dies der Fall, wird das Flag auf 'falsch' gesetzt. Man vergleiche dies mit dem Adressbereichsalgorithmus im Kapitel über `sed`[9]. NB: Besteht einer der Operanden lediglich aus einer Zahl, so wird in der Auswertung der Bedingung dieser Seite diese Zahl gegen die interne Variable `$.` verglichen, in welcher die aktuelle Zeilennummer abgelegt ist.
Der Vergleichsoperator <=> vergleicht zwei Zahlen und liefert als Ergebnis `-1`, falls der linke Operand kleiner ist, als der rechte. Sind beide Operanden gleich, ist das Resultat `0`; ist der linke Operand größer, als der rechte, ist das Ergebnis `+1`. Z.B.: `1 <=> 2`
In `perl` muß man Vergleiche zwischen Strings und solche zwischen Zahlen unterscheiden. Um Strings zu vergleichen, benutzt man die Operatoren `eq` für Gleichheit (oder `ne` für Ungleichheit), z.B. `"aber" eq "nicht"`, `"nix" ne "nada"`. Der `cmp` Operator entspricht dem <=> Operator, arbeitet aber auf Strings. Um auf lexikographisch kleiner, größer, kleiner-gleich, größer-gleich zu testen, benutzt man bei Strings die `lt`, `gt`, `le` und `ge` Operatoren: `"nix" le "aber"`, `"Linux" gt "SunOS"`.
Der Stringreplizierungsoperator wiederholt einen String: Der String, der durch Auswertung des linken Operanden entsteht wird sooft wiederholt, wie die Zahl angibt, zu der der rechte Operand auswertet. Z.B. wertet `"a" x 4` zu `"aaaa"` aus.
Die Matching-Operatoren `=~` und `!~` entsprechen denen von `awk` (siehe Abschnitt 3.3). Wir werden auf sie etwas näher in Abschnitt 4.4 eingehen.
Als letztes bleibt noch der Potenzierungsoperator `**`; dieser entspricht dem `awk`-Operator `^` und muß somit nicht näher erläutert werden.

[9] Ich hatte doch schon erwähnt, daß `perl` eine merkwürdige Semantik hat?

4.2.7 Listen-Operatoren

Die Bezeichnung Listen-Operator ist für die hier vorgestellten Konstrukte zunächst verwirrend, sehen sie doch auf den ersten Blick aus, wie Funktionsaufrufe. In der Tat sind sie dies jedoch nicht, sondern stellen fest verdrahtete Bestandteile von `perl` dar.
Der Grund dafür liegt darin, daß sie in einigen verschiedenen Formen auftreten, die eine gesonderte Behandlung nötig machen.
Die Listen-Operatoren treten i.A. in einer (oder mehrerer) der folgenden Formen auf:

op Als einfacher Operator.

op Liste Als Operator, gefolgt von einer Liste.

op NameVal Liste Als Operator, gefolgt von einem Namen oder einem Ausdruck, gefolgt von einer Liste.

Nicht alle Operatoren unterstützen alle Formen, wir geben die unterstützten Formen bei den einzelnen Operatoren an.
Es sei noch angemerkt, daß man die *Liste* in den obigen Formen nicht in Klammern einschliessen *muß*, es jedoch darf. Z.B. ist in dem Ausdruck `chmod 0755, "/etc/passwd", "/etc/ping";` die Liste ungeklammert.
Hier nun die Operatoren:

chmod *Liste* Bietet die Funktionalität des *chmod(1)* Kommandos direkt als Operator. Das erste Element der Liste ist der neue Modus[10] der den Dateien, welches die zweiten, dritten, etc. Elemente der Liste sind, versucht wird zuzuweisen. Ein Beispiel hatten wir schon in der Einleitung zu diesem Abschnitt gesehen. Dieser Operator gibt die Anzahl der Files zurück, für die die Operation erfolgreich war.

chown *Liste* Wechselt den Eigentümer und die Gruppe von Dateien (äquivalent zum *chown(1)* Kommando). Hierbei stellt das erste Element der angegebenen Liste die neue numerische User-Id, das zweite die neue numerische Gruppen-Id dar. Die restlichen Elemente sind Filenamen. Z.B.: `chown (0,0,"/etc/sendmail.cf","/etc/passwd")`[11]

Auch dieser Operator gibt die Anzahl der Files zurück, für die die Operation erfolgreich war.

[10] Als Zahl, nicht symbolisch (wie bei *chmod(1)* möglich).
[11] Hier sieht man, daß die *Liste* auch geklammert werden darf.

die [*Liste*] Dieser Operator gibt die aneinandergehängten Werte der *Liste* (oder den String `"Died"`, falls die Liste fortgelassen wurde) auf dem Filehandle `STDERR` aus und beendet das Programm. In einem `eval` Ausdruck allerdings wird nur die Auswertung des `eval` abgebrochen (und in der Variablen `$@` der Fehlertext gespeichert), siehe dort. Beispiel: `if ($a!=1) { die("Fehler: a nicht 1");}` Angemerkt sei noch, daß die Zeilennummer und das `perl`-File, in welchem das `die` aufgerufen wurde, an die Fehlermeldung angehängt werden, wenn diese nicht mit `\n` endete.

exec ***Liste*** Dieser Operator entspricht in etwa dem *exec(2)* Systemaufruf und führt ein anderes Programm an Stelle des `perl`-Programmes aus. Beispiel: `exec "cat", "/tmp/xx" , "-"` Dieser Operator kehrt nur zurück, wenn die Ausführung des Programmes fehlschlug.

kill ***Liste*** Sendet ein Signal an eine Liste von Prozessen. Das erste Element der Argument-Liste ist das Signal, welches gesendet werden soll. Es kann entweder numerisch angegeben werden, oder als String, der den Signalnamen darstellt. Die Nummern bzw. Namen der Signale findet man u.a. in *signal(3)* (wobei die Namen ohne das in der Manual-Page auftretende `SIG` zu nehmen sind). Beispiel: `kill "HUP", 1`

Dieser Operator gibt ebenfalls die Anzahl der Prozesse zurück, denen das Signal erfolgreich zugestellt werden konnte.

print ***File Liste***

print ***Liste***

print Der `print` Operator ist für die Ausgabe von Werten zuständig. Per Default wird die Ausgabe auf `STDOUT` geleitet, man kann die aber mittels `select` (siehe Abschnitt 4.2.8) umleiten. In der ersten Form kann man aber auch explizit ein Filehandle angeben, durch welches die Ausgabe erfolgen soll. Der Operator gibt die zusammengehängten Werte der *Liste* aus. Fehlt die Liste, wird der Inhalt der Variablen `$_` ausgegeben. Ein oft gemachter Fehler ist es, zwischen dem Filehandle und der Liste ein Komma zu setzen. Der Operator gibt `0` zurück, wenn bei der Ausgabe ein Fehler auftrat, sonst `1`.

Beispiel: `print FOO "Schoen ist das Leben", "\n"`

printf ***File Format*, *Liste***

printf ***Format*, *Liste*** Dieser Operator ähnelt der *printf(3)* Funktion und dem Operator gleichen Namens in `awk`. Wie beim `print` Operator kann man ein

Filehandle angeben, in welches die Ausgabe erfolgen soll. Default ist wiederum `STDOUT`. In jedem Fall muß man diesem Operator als *Format* einen Format-String übergeben, der den gleichen Aufbau hat, wie der in *printf(3)* beschriebene. Die Elemente der *Liste* werden dann an den Argumentpositionen im Format-String eingesetzt. Man beachte, daß wie beim `print` Operator *kein* Komma zwischen *File* und *Format* gesetzt werden darf. Sehr wohl aber *muß* man *Format* und *Liste* durch ein Komma trennen. Diese Funktion gibt ebenfalls `0` zurück, falls bei der Ausgabe Fehler auftraten, sonst `1`.

Beispiel: `printf "%d: %d\n", $line, $numchar;`

return ***Liste*** Kehrt aus einer Subroutine zurück (siehe Abschnitt 4.5) und gibt dabei die ausgewertete *Liste* zurück.

reverse ***Liste*** Dieser Operator liefert in einem Vektor-Kontext eine Liste zurück, die aus der umgedrehten übergebenen *Liste* besteht. In einem skalaren Kontext wird nur das erste Element der Liste genommen, als String interpretiert, dieser umgedreht und zurückgegeben.

Beispiel: `reverse (1,2,3)=(3,2,1),reverse @foo`

sort ***SubName Liste***

sort ***Liste*** Sortiert eine Liste. Wenn *SubName* angegeben ist, stellt es den Namen einer Subroutine dar, die beim Vergleich von Elementen aufgerufen wird. Ansonsten wird die *Liste* in der kanonischen Reihenfolge sortiert. Die Subroutine muß die Werte `-1`, `0` oder `1` zurückgeben, je nachdem ob die zwei zu vergleichenden Elemente in der Relation 'kleiner', 'gleich' oder 'größer' stehen (vgl. `cmp` und <=> Operatoren). Die beiden Elemente werden der Subroutine dabei in den Variablen `$a` und `$b` übergeben.

Beispiel:

```
sub canonically { $a <=> $b; }
@sorted=sort canonically @unsorted;
```

system ***Liste*** Führt das in der *Liste* angegebene Programm in einer Subshell aus und gibt den Exit-Code des Programmes zurück.

Z.B.: `system "ls", "-l", "/";`

unlink ***Liste*** Löscht die angegebenen Files und gibt die Anzahl erfolgreich gelöschter Files zurück. Man kann Directories nur löschen, wenn `perl` das `-U` Flag mitgegeben wurde.

Z.B.: `unlink "/tmp/xx.$$", "/tmp/yy.$$";`

utime ***Liste*** Setzt die Zugriffs- und Modifikationszeiten für Files. Die ersten beiden Elemente der *Liste* sind die Zugriffs- und Modifikationszeiten, als Sekunden seit dem 1. Januar 1970 (GMT), die restlichen Filenamen. Gibt die Anzahl erfolgreicher Änderungen zurück.

Z.B.: `$jetzt=time; utime $jetzt, $jetzt, "xx.c";`

warn ***Liste*** Hängt die Werte der gegebenen Liste aneinander und gibt sie auf `STDERR` aus. Anders, als der `die` Operator wird aber die Ausführung des Programmes fortgesetzt.

Z.B.: `if ($a!=$b) {warn "Parameter stimmen nicht"}`

4.2.8 Unäre Operatoren

Auch die unären Operatoren sehen aus, wie Funktionasufrufe. Sie werden nur deshalb als Operatoren betrachtet, weil man ihr Argument nicht einklammern muß. So sind z.B. `cos(3.0)` und `cos 3.0` identisch.
Hier die Operatoren:

alarm ***Expr*** Sendet nach *Expr* Sekunden ein `SIGALARM` an den Prozess (siehe *signal(3)*), etwa `alarm 4`.

chdir ***Expr*** Wechselt in das Verzeichnis, das durch den String *Expr* gegeben wird. Bei Erfolg gibt der Operator `1` zurück, sonst `0`. Etwa: `chdir ("/tmp");`

chop ***Liste***

chop ***Variable***

chop Der `chop` Operator trennt das letzte Zeichen eines Strings ab. Wird er auf eine Liste angewandt, trennt er das letzte Zeichen jedes Elementes ab. Wird kein Argument gegeben, trennt er das letzte Zeichen von `$_` ab. Der Operator kann auf auch L-Werte[12] angewandt werden, so führt `chop($a=<STDIN>);` dazu, daß in `$a` eine Zeile eingelesen und danach das letzte Zeichen entfernt wird. Der Operator gibt das Zeichen zurück, das er abgetrennt hat.

Beispiele:

[12] Ein L-Wert ist etwas, was auf der linken Seite einer Zuweisung stehen kann.

```
chop @a;
chop $a;
chop $a[3];
```

close ***File*** Schliesst das Filehandle *File*.

cos ***Expr*** Berechnet den Cosinus des Argumentes.

defined ***Expr*** Gibt `0` zurück, falls das Argument den undefinierten Wert besitzt, `1` sonst.

each ***AVektor*** Gibt eine Liste aus (Schüssel, Wert) zurück. Bei jeder Auswertung wird ein weiteres Paar aus dem assoziativen Vektor *AVektor* geliefert, so daß die Elemente aufgezählt werden.

Das folgende Beispiel gibt alle (Schlüssel, Wert) Paare des assoziativen Vektors `%bar` aus:

```
while ( ($s,$w)= each %bar) {
  printf("Schluessel %s, Wert %s\n",$s,$w);
}
```

Sind alle Werte aufgezählt, wird die leere Liste zurückgegeben, die als Bedingung zu 'falsch' auswertet (siehe Abschnitt 4.2.3). Man kann eine Aufzählung nicht in der Mitte zurücksetzen. Will man wieder von vorne beginnen, muß man erst alle Paare aus dem Vektor gelesen haben.

eval ***Expr***

eval Der String *Expr* wird als `perl`-Programm interpretiert und ausgeführt. Dies hat den gleichen Effekt, als stünde dieser Programmtext an der Stelle des `eval`. Tritt in dem Programm ein Fehler auf, wird die Fehlermeldung in `$@` geschrieben. Wird *Expr* weggelassen, dann wird der Wert von `$_` benutzt.

exit ***Expr***

exit Berechnet den (numerischen) Wert von *Expr* und bricht dann das Programm mit diesem Exit-Code ab. Wenn man das Argument forlässt, wird `0` als Exit-Code genommen.

exp ***Expr***

exp Berechnet e^{Expr}. Wird das Argument weggelassen, wird `$_` genommen.

fileno *File* Gibt die Filenummer des Files zurück, das durch *File* referenziert wird.

getc *File*

getc Liest ein Zeichen aus dem durch *File* referenzierten File und gibt es zurück. Lässt man das Argument weg, wird aus `STDIN` gelesen. Ist kein Zeichen mehr verfügbar, wird `""` zurückgegeben.

gmtime *Expr*

gmtime Wandelt eine Zeit, die als Sekunden seit dem 1. Januar 1970 übergeben wird, in eine Liste mit neun Elementen um. Z.B.:

```
($sek,$min,$std,$mtag,$mon,$jahr,$wtag,$jtag,$sz)=
    gmtime (time);
```

Dir Liste besteht aus Sekunden, Minuten, Stunden, Tag (des Monats), Monat (0..11), Jahr (zweistellig), Wochentag (0..6, Sonntag=0), Tag im Jahr und ob Sommerzeit herrscht (`$sz=1` im obigen Beispiel). Lässt man *Expr* weg, dann wird `gmtime (time)` ausgeführt. Die zurückgegebene Zeit ist die, welche in der UTC (GMT) Zeitzone herrscht.

hex *Expr*

hex Wandelt das Argument (oder `$_`, falls kein Argument gegeben wurde), welches als hexadezimale Zeichenkette angesehen wird in eine Zahl um. Z.B.: `hex "0x80"=128`.

int *Expr*

int Wandelt das Argument (oder `$_`) in eine Ganzzahl um, indem die Nachkommastellen abgeschnitten werden.

keys *AVektor* Gibt eine Liste mit den Schlüsseln des assoziativen Vektors *AVektor* zurück. Z.B. ergibt `keys %bar` die Liste `"a","b"`, falls vorher `%bar` mit `$bar{"a"}=9; $bar{"b"}=42;` initialisiert wurde.

length *Expr*

length Gibt die Länge des Argumentes (bzw. `$_`) zurück: Dieses wird als String aufgefasst und dann dessen Länge genommen.

localtime ***Expr***

localtime Wandelt das Argument, welches als Sekunden seit dem 1. Januar 1970 gegeben ist, in eine Liste derselben Form wie bei `gmtime` um. Die Zeit wird allerdings relativ zur lokalen Zeitzone genommen. Ohne Argument wird `localtime(time)` ausgeführt.

log ***Expr***

log Berechnet den natürlichen Logarithmus des Argumentes (oder `$_`).

lstat ***File***

lstat ***Expr*** Führt die zu `stat` äquivalente Operation aus. Der einzige Unterschied ist, daß Daten über einen symbolischen Link selbst und nicht über die Datei, auf die er zeigt, zurückgegeben werden.

oct ***Expr***

oct Wandelt den Argumentstring (oder `$_`), welcher als Octalzahl angesehen wird, in eine Zahl um. Z.B. `oct "010"=8`.

ord ***Expr***

ord Gibt den ASCII-Code des ersten Zeichens des Argument-Strings (oder `$_`) zurück. Das Ergebnis ist immer positiv.

pop ***Vektor*** Kürzt den Vektor *Vektor* um dessen letztes Element und gibt dieses zurück. So enthält nach

```
@foo=(1,2,3);
$el=pop @foo;
```

der Vektor `@foo` die Elemente `1,2` und `$el` den Wert `3`.

rand ***Expr***

rand Gibt einen 'Zufalls'-Wert zwischen 0 und dem Wert von *Expr* zurück (bzw. zwischen 0 und 1). Das Ergebnis ist eine Fließkommazahl.

readlink ***Expr***

readlink Gibt den Filenamen, auf welchen der symbolische Link, dessen Name als Argument übergeben wird, (bzw. `$_`) zeigt. Ist der übergebene Name kein Link, oder tritt sonst ein Fehler auf, wird der undefinierte Wert zurückgegeben und die Variable `$!` auf den Fehlercode gesetzt.

Beispiel: `$realname=readlink "/etc/localtime";`

require ***Expr***

require Dieser Operator lädt ein externes `perl`-Programm (Package, etc.) zum aktuellen Programm dazu. Das Argument (bzw. `$_`) ist der Name des Programms. Dieser wird in allen Directories gesucht, die in dem Vektor `@INC` abgelegt sind. Wird das Programm nicht gefunden, wird das komplette Programm abgebrochen.

reset ***Expr***

reset Wird benutzt, um Variablen bzw. Suchmuster zu löschen. Falls ein Argument (String) gegeben wird, dann werden alle Variablen, deren Namen mit den dort spezifizierten Buchstaben anfangen, gelöscht. Wird kein Argument gegegeben, werden nur alle `?`*`Pattern`*`?` Suchmuster (siehe Abschnitt 4.4) gelöscht. Z.B. löscht `reset "a"` alle Variablen, die mit dem Buchstaben '`a`' beginnen. `reset "a-z"` löscht alle Variablen, die mit einem Kleinbuchstaben beginnen.

rmdir ***Expr***

rmdir Entfernt das angegebene Directory (oder das in `$_` abgelegte). Liefert `1` bei erfolgreicher Durchführung, sonst `0`. Z.B.: `rmdir "/tmp/data"`

shift ***Vektor***

shift Dieser Operator kürzt den angebenen Vektor (oder `@ARGV`) um dessen ersten Wert und liefert diesen Wert zurück. Falls der Vektor kein Element enthält, wird der undefinierte Wert zurückgegeben. Wird `shift` ohne Argument innerhalb einer Subroutine benutzt, wird statt `@ARGV` der Vektor `@_` gekürzt. Ansonsten vgl. mit `pop`, etc.

sin ***Expr***

sin Liefert den Sinus von *Expr* (bzw. `$_`).

sleep ***Expr***

sleep Legt den Prozess für die angegebene Anzahl Sekunden schlafen. Ist kein Argument gegeben, wird der Prozess für immer schlafen geschickt. Der Schlaf kann durch ein `SIGALARM` (siehe `alarm`) unterbrochen werden. Der Operator gibt die Anzahl Sekunden, die der Prozess geschlafen hat, zurück.

sqrt ***Expr***

sqrt Gibt die Quadratwurzel des Argumentes (oder von `$_`) zurück.

srand ***Expr***

srand Setzt den Startwert für die Generierung von Zufallszahlen (siehe `rand`). Ohne Argument wird `srand(time)` ausgeführt.

stat ***File***

stat ***Expr*** Gibt einen aus dreizehn Elementen bestehenden Vektor zurück, welcher Informationen über das File enthält, welches entweder über ein Filehandle oder seinen Namen übergeben wird. Die Elemente des Vektors sind: Device-Nummer, I-Node-Nummer, Modus-Bits, Anzahl Links auf das File, User-ID des Besitzers, Gruppen-ID des Besitzers, File-Typ, Größe, Zugriffszeit, Modifizierungszeit, Erstellungszeit, Blockgrösse und Anzahl Blocks in dem File. Die Einträge entsprechen den Feldern des Records, der bei dem *stat(2)* Systemaufruf benutzt wird.

study ***Skalar***

study Dieser Operator untersucht den angegebenen String (oder `$_`) und merkt sich Optimierungen, so daß nachfolgende Mustervergleiche in diesem String schneller gehen (können).

Z.B.: `study $foo; print "ja" if $foo ˜ /^nix/;`

tell ***File***

tell Gibt die Position des Filepointers in dem angegeben *File* zurück. Wird das Argument weggelassen, wird das zuletzt referenzierte File verwandt.

Z.B.: `$pos=tell STDIN;`

umask ***Expr***

umask Setzt die umask für den Prozess, siehe *umask(2)*. Das Argument ist eine Zahl, die die Maske angibt. Die alte Maske wird als Resultat zurückgegeben. Ohne Argument wird keine neue Maske gesetzt, sondern nur die alte returniert.

Beispiel: `$old=umask 077;`

undef ***Expr***

undef Dieser Operator löscht die Definition seines Argumentes, das ein L-Wert sein muß und gibt selbst den undefinierten Wert zurück. Ohne Argument wird nur der undefinierte Wert geliefert. Z.B.: `undef $foo[2];`

values ***AVektor*** Ähnlich zu `keys`, nur das hier alle Werte des assoziativen Vektors *AVektor* als Liste geliefert werden.

4.2.9 Dateiprüfung

`perl` erlaubt es, mit primitiven Operatoren vielfältige Tests auf Eigenschaften von Files durchzuführen. Das Ergebnis ist entweder `1`, falls der Test positiv ausfällt, oder `0`. Ausnahmen: Die Operatoren `-A`, `-C` und `-M` liefern eine Zeit zurück (es wird die Anzahl Tage seit dem Ereignis zurückgegeben). Alle Operatoren zur Dateiprüfung nehmen ein Argument, welches den Namen der Datei (des Verzeichnisses) angibt, oder ein Filehandle. Auf dem so refernziertem File wird dann der Test durchgeführt. Man kann das Argument auch fortlassen. Dann wird als Ersatz der Inhalt von `$_` als Filename benutzt. Ausnahme: Der `-t` Operator benutzt als Default das Filehandle `STDIN`.
Alle Operatoren haben die Form *−Buchstabe File*. Dabei muß der Buchstabe direkt im Klartext folgen; folgendes wird also nicht funktionieren:
`$a="f"; if (-$a "/tmp/xx")....`
Hier nun also die Operatoren, in alphabetischer Reihenfolge:

-A Gibt die Zeit zurück, die seit dem letzten Zugriff auf die Datei vergangen ist, etwa `if (-A "/tmp/xx" > 1)....`

-B Die Datei ist eine Binärdatei (Gegenstück zu `-T`). Der Test wird durchgeführt, indem ein Block der Datei gelesen wird und die Anzahl von Steuerzeichen in ihm überprüft wird. Sind mehr als 10% davon enthalten, ist die Datei eine Binärdatei.

-b Testet, ob die Datei ein Block-Device (z.B. `/dev/sd0a`) ist.

-C Zeit, die seit Erstellung der Datei vergangen ist.

-c Testet, ob die Datei ein Character-Special Device ist (etwa `/dev/rsd0a`).

-d Testet, ob das Argument ein Directory ist.

-e Testet, ob die Datei existiert.

-f Testet, ob das Argument eine einfache Datei ist.

-g Testet, ob die Datei das Set-Group-ID Bit gesetzt hat (siehe *chmod(2)*).

-k Testet, ob die Datei das Sticky Bit gesetzt hat (siehe *chmod(2)*).

-l Prüft, ob das Argument ein symbolischer Link ist.

-M Gibt die Zeit seit der letzten Änderung der Datei zurück.

-O Prüft, ob die User-Id der Datei dieselbe wie die reale User-Id des Prozesses ist (also die des Benutzers, der `perl` gestartet hatte).

-o Prüft, ob die User-Id der Datei dieselbe ist, wie die effektive User-Id des Prozesses ist (ist verschieden von der realen User-Id für Programme, die das Set-User-Id Bit gesetzt hatten).

-p Prüft, ob die Datei eine Pipe (FIFO) ist.

-R Prüft, ob die Datei mit der realen User-Id des Prozesses lesbar ist.

-r Prüft, ob die Datei mit der effektiven User-Id des Prozesses lesbar ist.

-S Testet, ob die Datei ein Socket ist (siehe *socket(2)*).

-s Testet, ob die Datei nicht leer ist.

-T Testet, ob die Datei eine Textdatei ist (siehe `-B`).

-t Prüft, ob die Datei ein Terminal ist.

-u Liefert `1`, wenn das Set-User-Id der Datei gesetzt ist.

-W Prüft, ob die Datei mit der realen User-Id schreibbar wäre.

-w Prüft, ob die Datei mit der effektiven User-Id schreibbar wäre.

-X Prüft, ob die Datei mit der realen User-Id ausführbar wäre.

-x Testet, ob die Datei mit der effektiven User-Id ausführbar wäre.

-z Prüft, ob die Datei die Größe 0 hat.

4.2.10 Sonstige Funktionen

Dieser Abschnitt präsentiert einige weitere Funktionen, die nicht zu den Listen Operatoren oder unären Operatoren gehören.

atan2(x,y) Gibt den arcus tangens von x/y zurück.

delete *$Array{Key }* Löscht den angegebenen Schlüssel aus dem assoziativen Vektor *Array*. Z.B.: `delete $foo{"nix"};`

eof *File*

eof Gibt `1` zurück, falls das angegebene Filehandle am Ende der Datei angekommen ist, sonst `0`. Ohne Argument wird das zuletzt benutzte Filehandle verwandt. Z.B.: `if (eof(STDIN))...`

fork Stellt den *fork(2)* Systemaufruf in `perl` zur "Verfügung. Es wird eine Kopie des Prozesses angelegt. Im Vaterprozess liefert diese Funktion die Prozess-Id des Kindprozesses zurück. Im Kindprozess eine 0. Tritt bei dem Anlegen des neuen Prozesses ein Fehler auf, ist das Ergebnis der Funktion der undefinierte Wert.

grep(*Expr*,*Liste*) Diese Funktion wertet den Ausdruck *Expr* nacheinander für die Elemente der Liste aus. Der Rückgabewert ist dann die Liste der Elemente, für die der Ausdruck 'wahr' ergab. Innerhalb des Ausdruckes steht das jeweilige Element der Liste als `$_` bereit.

Z.B. stehen in diesem Beispiel am Schluß in `@foo` alle Zeilen aus `@bar`, die keine Leerzeilen sind:

```
@foo=grep(! (length($_)==0),@bar);
```

index(s,t,p)

index(s,t) Gibt die Position des ersten Auftretens des Substrings t im String s zurück. Die Suche beginnt dabei an der Position p (oder am ersten Zeichen, wenn p fehlt). Die Positionen zählen dabei ab 0[13]. Wird der Substring nicht gefunden, wird `-1` zurückgegeben (bzw. `$[-1`).

Z.B. liefert `index("laberaber","aber")` den Wert 1 zurück, wohingegen der Ausdruck `index("laberaber","aber",2)` den Wert 5 liefert.

join(*Expr*,*Liste*) Diese Funktion verbindet die Strings, welche in der *Liste* angegeben sind, mit dem String, welcher durch *Expr* dargestellt wird und gibt die resultierende Zeichenkette zurück.

Z.B:

```
$s=join(":",$name,$nachname,$telefon,$login);
$t=join(" ",@foo);
```

link(o,n) Erstellt einen Link von der Datei mit Namen o zu dem neuen Namen n. Z.B. `link("./lib/libfoo.a", "/usr/lib/libfoo.a");` Das Ergebnis der FUnktion ist `1`, im Erfolgsfall, sonst `0`.

[13]Genauer, ab dem Wert der in der Variablen `$[` abgelegt ist.

mkdir(f,m) Erstellt ein Directory mit Namen n und Zugriffsmodus m. Gibt `1` zurück, falls das Directory erstellt werden konntze, sonst `0`. Z.B. `mkdir("/tmp/data", 0700)`.

push(*Array*,*Liste*) Fügt die Elemente der *Liste* am Ende des Vektors *Array* an. Z.B. `push(@foo, 1, 2, 3)`.

rename(o,n) Bennent das File mit dem Namen n in o um.

Z.B. `rename("/tmp/xx", "/tmp/yy")` Rückgabewerte sind wieder `1` für Erfolg und `0` sonst.

rindex(s,t,p)

rindex(s,t) Sucht den Substring t im String s. Einziger Unterschied zu `index` ist, daß das letzte Auftreten des Substrings gesucht wird.

scalar(*Expr*) Erzwingt einen skalaren Kontext zur Auswertung des Ausdruckes *Expr*. Wird verwendet, um auch in Vektor Kontexten die skalare Version einer Funktion zu erzwingen.

select(*File*)

select Setzt das angebene Filhandle als Standardausgabe und liefert das alte Filehandle zurück. Wird kein Filehandle angegeben, wird nur das alte zurückgeliefert. Z.B. `$oldfh=select(OFH);`

splice(*Array*, o, l, *Liste*)

splice(*Array*, o, l)

splice(*Array*, o) Diese Funktion entfernt aus dem Vektor *Array* l Elemente, beginnend an der Position o. D.h. es werden die Elemente `$`*`Array`*`[`o`]` ...`$`*`Array`*`[`o+l−`1]` entfernt. Dann werden an der Stelle der entfernten Elemente die Elemente der *Liste* eingefügt. Wird die Liste weggelassen, werden keine Elemente eingefügt. Wird l weggelassen, werden alle Element ab o bis zum Ende des Vektors entfernt. Die Funktion gibt die entfernten Elemente als Liste zurück.

Z.B.:

```
@foo=(1,2,3,4);
@removed=splice(@foo,1,2,"a","b");
```

`@removed` enthält `(2,3)` und `@foo` enthält am Ende `(1,"a","b",4)`

split(/*Pattern*/,*Expr*,l)

split(/*Pattern*/,*Expr*)

split(/*Pattern*/)

split Spaltet den String, zu welchem *Expr* auswertet in Subfelder auf und gibt diese in einem Vektor zurück. Das *Pattern* beschreibt die Begrenzer der Felder. l gibt die maximale Anzahl Felder an, die in dem Vektor abgelegt werden sollen. Wird l weggelassen, werden so viele Felder wie möglich abgelegt. Wird *Expr* weggelassen, wird statt dessen `$_` verwandt. Wird auch noch das *Pattern* fortgelassen, werden als Begrenzer Leerzeichen/TABs verwandt.

In einem sklaren Kontext gibt diese Funktion die Anzahl Felder zurück, die abgespalten wurden. Das Array `@_` enthält in diesem Fall die Felder.

Z.B.

```
@bar=split(/:/,
    "thesing:x:1001:200:S. Thesing:/home/thesing",
    5);
($login,$passwd,$uid,$gid,$rest)=split(/:/,
    "thesing:x:1001:200:S. Thesing:/home/thesing",
    5);
```

Nachher stehen in `@bar` fünf Elemente. Das letzte enthält den Substring des Argumentes, beginnend mit '`S. Thesing`'.

In der zweiten Zeile enthält `$login` nachher den String `"thesing"`, `$uid` enthält `"1001"`, etc.

sprintf(f,*Liste*) Diese Funktion ähnelt `printf`, nur daß hier das Ergebnis nicht ausgegeben wird, sondern als String zurückgeliefert. Ansonsten identisch zu `printf`.

substr(s,o,l)

substr(s,o) Diese Funktion gibt den Substring des Strings s zurück, der an Position o startet und l Zeichen lang ist (oder bis zum Ende des Strings s reicht). Dabei zählt als erste Position der Wert von `$[` (Default: `0`). Ist o negativ, beginnt die Zeichenkette soviele Zeichen vom Ende von s entfernt, wie der Betrag von o angibt. Wird l weggelassen, wird immer der Substring bis zum Ende von s genommen.

Eine Besonderheit von `substr` ist, daß es auch als L-Wert, also auf der linken Seite einer Zuweisung auftreten kann. Dies ersetzt dann den durch es beschriebenen Substring durch die linke Seite der Zuweisung (s kann hier also nur eine Variable oder ein Vektorelement sein).

Zum Beispiel:

```
$t=substr("laber",1);
substr($t,0,1)="ue";
```

Nach der ersten Zuweisung enthält `$t` den String `"aber"`. Nach der zweiten den String `"ueber"`.

symlink(*o*,*n*) Legt einen symbolichen Link von dem File mit dem Namen o auf das File n an. Gibt `1` bei Erfolg, sonst `0` zurück.

time Gibt die Anzahl Sekunden seit dem 1. Januar 1970 GMT zurück.

times Gibt einen Vektor aus vier Elementen zurück, die Zeiten (in Sekunden) angeben: die Zeit, die der Prozess im User-Modus gelaufen ist, die Zeit, die er im System-Modus lief, die Zeit, die seine Kinder[14] im User-Modus liefen und die Zeit, die seine Kinder im System-Modus liefen. Z.B.: `($u,$s,$cu,$cs)=times;`

unshift(*Array*,*Liste*) Stellt das Gegenstück zu `shift` dar: Diese Funktion fügt die Elemente der übergebenen *Liste* am Anfang des Vektors *Array* ein und gibt die Anzahl der Elemente des resultierenden Vektors zurück. Z.B.: `unshift(@bar,1,2)`

wait Wartet, ähnlich *wait(2)*, auf die Beendigung eines Kindprozesses und liefert dann die Prozess-Id dieses Prozesses. Das Resultat ist `-1`, falls kein Kindprozess existiert.

wantarray Diese Funktion liefert `1`, falls der Kontext, in dem sie ausgewertet wird, ein Vektor-Kontext ist. Andernfalls wird `0` zurückgegeben. Nützlich, um in eigenen Funktionen abhängig vom Kontext eine Liste oder eine Liste zurückzugeben.

[14]Also mittels `fork` oder Subshells erzeugte Prozesse.

4.3 Kommandos

In diesem Abschnitt werden nun (endlich) die Kommandos von `perl` vorgestellt. Wir beginnen mit den einfachen Kommandos und werden danach zu den Conditionalen und Schleifen von `perl` kommen.
Zu beachten ist, daß in `perl` Kommandos immer mit einem Semikolon beendet werden müssen (ausser Blöcken). Dies steht im Gegensatz zu Pascal oder `awk`, aber in der Tradition von **C**.

4.3.1 Einfache Kommandos

Einfache Kommandos sind einmal Ausdrücke, denen optional noch ein *Guard* folgen kann. Ein Guard ist dabei eine Bedingung, die vor Auswertung des Ausdruckes geprüft wird[15]. Trifft diese Bedingung nicht zu, wird der Ausdruck nicht ausgewertet. `perl` kennt vier Typen von Guards:

if *Expr* Prüft, ob der mitgegebene Ausdruck[16] zu 'wahr' auswertet. Wenn dies der Fall ist, wird der Rumpfausdruck ausgewertet.

Z.B. `$foo=99 if $bar!=1;`

unless *Expr* Ist die Negation des `if` Guards. Es wird also geprüft, ob der Ausdruck zu 'falsch' auswertet. Wenn dies der Fall ist, wird der Rumpfausdruck ausgewertet. Z.B. `$foo=99 unles $bar=1;`

Diese beiden Guards wurden nur einmal ausgewertet; auch der Rumpfausdruck wird höchstens einmal ausgewertet. Die folgenden beiden Guards werten den Rumpfausdruck wiederholt aus, solange die Bedingung erfüllt ist. Wiederum wird die Bedingung vor dem Rumpfausdruck ausgewertet:

while *Expr* Die Bedingung wird ausgewertet; ist das Ergebnis dabei 'wahr', wird der Rumpfausdruck ausgewertet. Dann beginnt das Spiel wieder von vorne mit der Auswertung der Bedingung.

Z.B.: `$foo++ while 2*$foo<$bar;`

until *Expr* Ist die Negation des `while` Guards. Die Bedingung wird ausgewertet, ergibt dies 'falsch', wird der Rumpfausdruck ausgewertet und danach wiederum die Bedingung, etc.

Z.B. `$foo++ until 2*$foo>=$bar;`

[15] Mit einer Ausnahme: `do`-Blöcke werden manchmal vor Prüfung der Bedingung ausgeführt.
[16] Nicht mit dem Rumpfausdruck verwechseln! Dieser Ausdruck ist die Bedingung.

Wie in **C** kann man Kommandos zu einem *Block* zusammenfassen, indem man sie in geschweifte Klammern einschließt. Ein Block kann in `perl` an Stelle eines Kommandos eingesetzt werden. Allerdings sollte man beachten, daß vor allem die zusammengesetzten Kommandos *nur* Blöcke, keine einzelnen Kommandos, als Bestandteile nehmen. Dies reduziert allerdings eine Quelle von Mehrdeutigkeiten und Fehlern[17].
Als letztes einfaches Kommando muß nun noch der `do`-Block besprochen werden. Er stellt eine weitere Eigenart von `perl` dar und führt auch zu einigen semantischen Verrenkungen. Das `do`-Konstrukt tritt in drei Formen auf:

do *Block* Dieses Konstrukt fÜhrt nacheinander alle Kommandos in dem *Block* aus und gibt den Wert des letzten Kommandos zurück.

do *Subroutine*(*Liste*) Die Subroutine mit dem Namen *Subroutine* wird aufgerufen. In den Klammern werden die Argumente zur Routine mitgegeben. Auch wenn die Subroutine keine Argumente benutzt, müssen die Klammern angegeben werden, um diese Variante von der letzten zu unterscheiden. Das Ergebnis dieses Aufrufes ist der Wert, der von der Subroutine zurückgegeben wird. Siehe Abschnitt 4.5.

do *Expr* Nimmt den String, der durch den Ausdruck gegeben ist und interpretiert ihn als einen Filenamen eines `perl`-Programmes, das dazu geladen und ausgeführt wird. Ähnelt also dem `require`.

Für die erste und zweite Variante des `do`-Blockes ändert sich die Semantik der `while` und `until` Guards. Folgen sie nämlich einem `do`-Block, dann wird *zuerst* der Block ausgeführt, bevor die Bedingung der Guards getestet wird. Dies gilt aber nicht für die `if` und `unless` Guards, die immer noch vor dem Block ausgewertet werden.
Beispiele:

```
do {$foo+=$bar; $bar--;} while $bar!=0;
do canonical(1,2);
do "cgi.pl";
```

4.3.2 Zusammengesetzte Kommandos

Die zusammengesetzten Kommandos in `perl` bestehen aus sechs Varianten des Conditionals, sechs Schleifen und einem Block-Konstrukt.
Die Conditionale:

if (*Expr*) *Block*$_1$

[17]Ganz im Gegensatz zur sonstigen `perl`-Tradition, wie ich bemerken möchte:-)

if (*Expr*) *Block*$_1$ else *Block*$_2$

if (*Expr*$_1$) *Block*$_1$ elsif (*Expr*$_2$) *Block*$_2$... else *Block*$_N$ Die Bedingung *Expr* wird ausgewertet. Ergibt dies 'wahr', wird *Block*$_1$ ausgeführt. In der zweiten Variante wird *Block*$_2$ ausgeführt, wenn die Bedingung 'falsch' ergibt. Die dritte Variante prüft in diesem Fall die zweite Bedingung *Expr*$_2$ und führt, wenn dies 'wahr' ergibt den zweiten Block aus, etc.

Die anderen drei Conditionale erhält man, indem man `if` durch `unless` ersetzt, was einfach den Sinn der Bedingung umkehrt (siehe voriger Abschnitt).
Die sechs Schleifenkonstrukte von `perl` haben die Gemeinsamkeit, daß man die Schleifen mit Labeln versehen kann. Diese können dann benutzt werden, um gezielt aus bestimmten Schleifen wieder auszusteigen (mittels `last`). Ein Label ist dabei ein Name, gefolgt von einem Doppelpunkt. Das Label muß direkt vor dem Schleifenkopf stehen (siehe unten stehende Beispiele).

Label*: while (*Expr*) *Block

Label*: while (*Expr*) *Block* continue *Block Die Bedingung *Expr* wird ausgewertet. Ergibt sie 'falsch', wird die Schleife verlassen. Sonst wird der *Block* ausgeführt. Ist ein mittels `continue` eingeleiteter Block gegeben, wird dieser nach Ausführung des Schleifenrumpfes und vor der nächsten Prüfung des Schleifenbedingung ausgewertet. Dies geschieht auch, wenn im Schleifenrumpf ein `next` Kommando den nächsten Schleifendurchlauf erzwingt.

Wenn man `while` durch `until` ersetzt erhält man die nächsten beiden Schleifenvarianten. Hierbei wird dann einfach der Sinn der Bedingung umgekehrt, analog zu den `while` und `until` Guards des vorigen Abschnittes.

Label*: for (*i-Expr*; *c-Expr*; *r-Expr*) *Block Dies entspricht der von **C** oder `awk` bekannten Schleife gleichen Namens. In der Tat ist sie in `perl` äquivalent zu folgendem Konstrukt:

```
i-Expr;
Label:
while (c-Expr) Block continue {r-Expr;}
```

Label*: foreach *Variable* (*Array*) *Block Diese Schleife dient, ähnlich wie ihr Gegenstück in `awk`, dazu, Schlüssel/Wert Paare eines Vektors aufzuzählen. Dabei werden die Schlüssel des Vektors *Array* nacheinander in der variablen *Variable* abgelegt und stehen dann im Rumpf der Schleife zur Verfügung.

Z.B.

```
foreach $v (%nam) { print "n[$v]=$nam{$v}\n"; }
```

Als letztes bleibt noch das folgende Konstrukt:

Block* continue *Block Bei dieser Konstruktion wird der Block der auf das Schlüsselwort `continue` folgt dann ausgeführt, wenn in dem ersten Block eine `next` Anweisung erfolgt.

In jedem Block, auch wenn dieser nicht Rumpf einer Schleife ist, kann man mit den folgenden Kommandos den Durchlauf steuern:

next *Label* Erzwingt den nächsten Durchlauf der Schleife mit dem angegebenen *Label*. Wird es weggelassen, ist die innerste Schleife gemeint. In einem normalen Block (keine Schleife), wird ein folgender `continue` Block durchlaufen und der Block nochmal durchgeführt. In einer Schleife wird ebenfalls der `continue` Block durchlaufen, bzw. die *r-expr* einer `for` Schleife ausgewertet. Danach wird wieder zur Auswertung der Schleifenbedingung gesprungen.

last *Label* Verlässt den Rumpf der durch das *Label* bezeichneten Schleife (bzw. die innerste), ohne `continue` Blöcke auszuführen. In einem normalen Block wird einfach nur der Block sofort verlassen.

redo *Label* Ähnelt der `next` Anweisung. Einziger Unterschied ist, daß keine `continue` Blöcke ausgewertet werden und auch in einer Schleife wird die Schleifenbedingung nicht erneut ausgewertet, sondern sofort der Rumpf neu ausgeführt.

Beispiele:

```
while ($n>0) {
 $s+=$n;
 $n--;
}

for ($s=0; $n>0; $n--) {
 $s+=$n;
}

A:
while ($n>1) {
 B:
  while ($m>9) {
   ...
   last A, if $n+$m>21;
```

```
  }
}

FOO: {
 if ($a == 1) { $x=2; last FOO;}
 if ($a == 2) { $x=4; last FOO;}
}
```

Man beachte, daß das letzte Beispiel das Äquivalent einer **C** `switch` Anweisung ist:

```
 switch (a) { case 1: .... case 2: ....}
```

4.4 Pattern

`perl` kennt eine Vielzahl von Pattern, die wir im folgenden vorstellen. Dabei wird das Matching immer gegen eine Zeichenkette betrachtet.

^ Passt auf den Anfang der Zeichenkette.

$ Passt auf das Ende der Zeichenkette.

***** Passt auf beliebig (auch keine) Wiederholungen des vorstehenden Patterns.

+ Passt auf ein- oder mehrmalige Wiederholung des vorstehenden Patterns.

? Passt auf null- oder einmaliges Auftreten des vorstehenden Patterns.

{n,m} Passt auf n- bis m-malige Wiederholung des vorstehenden Patterns. Spezialformen: `{`n`,}` für mindestens n-malige Wiederholung, `{`n`}` für genau n-malig.

| Alternative z.B. `a|b`

[...] Eine Klasse, wie von `sed` und `awk` her bekannt.

(...) In `perl` können Pattern geklammert werden. `perl` merkt sich bei einem Matching, welche Teile des Strings gegen geklammerte Patternteile matchten. Diese sind dann mittels \n wieder in referenzierbar.

. Der Punkt matcht jedes Zeichen, ausser `\n`.

\b Matcht Wortübergänge, also Stellen zwischen `\w` und `\W`, siehe unten.

\B Matcht Stellen, die keine Worübergänge sind.

Die folgenden speziellen Sequencen können in Pattern eingesetzt werden, um bestimmte Zeichen zu matchen:

- Die Sequenzen `\n`, `\r`, `\t`, `\f` stehen für die gewohnten Steuerzeichen Newline, Carriage Return, TAB, Form-Feed.
- `\d` steht für eine Ziffer, ist also äquivalent zu `[0-9]`.
- `\D` steht für etwas, das keine Ziffer ist, also `[^0-9]`.
- `\w` steht für ein Wortzeichen, äquivalent zu `[0-9a-zA-Z_]`.
- `\W` steht für etwas, das kein Wortzeichen ist.
- `\s` steht für eine Leerzeichen, also `[\t\n\r\f ]`
- `\S` steht für die Negation von `\s`.
- `\0`*oo* matcht das Zeichen, dessen ASCII-Code durch die Okatalzahl *oo* gegeben ist.
- `\x`xx matcht das Zeichen, das den Hexadezimalcode xx hat.
- `\`n matcht den Teilstring, der von dem n-ten Klammerausdruck (s.o.) schon gematcht wurde. Zählung beginnt mit 1.

Z.B.:

```
/^\d+/
/(\d+) and \1/
```

4.4.1 Operatoren

Im Zusammenhang mit Pattern sind folgende Operatoren nützlich.
Wir hatten schon in Abschnitt 4.2.6 die beiden Matching-Operatoren `=~` und `!~` erwähnt. Diese dienen dazu, Zeichenketten an ein Pattern zu *binden*. Z.B.: `$foo =~ /^\d/+`. Dabei negiert `!~` noch das Ergebnis des Matches. Der eigentliche Operator, der das Matching vornimmt ist jedoch das `/`*`Pattern`*`/`: Er liefert in einem skalaren Kontext entweder 'wahr' oder 'falsch' zurück, je nachdem, ob der Match gelang oder nicht. Dabei wird die durch die Matching-Operatoren angegebene Zeichenkette gematcht, bzw. `$_`, wenn diese nicht verwandt wurden. Gelingt ein Match, so werden in den Variablen `$1`, `$2`, etc. die Teile des Strings abgelegt dir jeweils durch den ersten, zweiten, etc. Klammerausdruck im *Pattern* gematcht wurden. So steht nach

`"123 456" =~ /(\d+) (\d+)/` in `$1` der String `"123"` und in `$2` der String `"456"`.
In einem Vektor-Kontext liefert dieser Operator einen Vektor zurück, der aus den gematchten Teilstrings besteht (also dem, was sonst in `$1`, etc. steht). Misslang der Match, wird der Nullvektor zurückgegeben. In diesem Kontext werden die Variablen `$`*n* *nicht* gesetzt.
Beispiel: `($first,$last) = ("123 456" =~ /(\d+) (\d+)/)`
Der `?`*Pattern*`?` Operator ist nahezu identisch zu `/`*Pattern*`/`, nur daß er nur einmal den Match ausführt und diesen dann immer wider zurückgibt, solange bis die `reset` Funktion ausgeführt wird.
Ähnlich zum Substitute-Kommando von `sed` kennt auch `perl` einen Operator, mit dem man Ersetzungen surchführen kann. Genau wie die beiden vorigen Operatoren, benutzt er dazu einen Ausdruck, der an ihn (via z.B. `=~`) gebunden ist (bzw. `$_`). Allerdings muß dieser Ausdruck in diesem Fall ein L-Wert sein, da ja die Zeichenkette verändert wird. Die Form des Operators ist:

`s/`*Pattern*`/`*Text*`/`*Flags*

Dabei wird in der gebundenen Zeichenkette das erste Vorkommen des *Pattern*s durch den *Text* ersetzt. Im Text können, wie bei `sed`, gemathcte Subpattern via `\`*n* eingesetzt werden (s.o.). Will man alle Vorkommen ersetzen, muß man das Flag `g` angeben. Soll der Match Groß-/Kleinschreibung ignorieren, gibt man `i` an. Will man statt eines einfachen Textes den Wert eines Ausdruckes einsetzen, gibt man diesen an der Stele von *Text* an und hängt das `e` Flag an.
Dieser Operator gibt die Anzahl der gemachten Ersetzungen zurück. Als Seiteneffekt werden in dem gebundenen L-Wert die Ersetzungen gemacht.
Beispiele:

```
$foo =~ s/^(\d+) (\d+).*$/\1-\2/;
if (s/#//) { print "# enthalten";}
```

Als letzter wird nun noch der Translate-Operator vorgestellt. Dieser hat die analoge Funktion zum Translate-Kommando von `sed`: Er ersetzt Vorkommen bestimmter Zeichen durch bestimmte andere Zeichen. Seine Form:

`tr/`s`/`r`/`

Hierbei sind s und r Zeichenfolgen. Das erste Zeichen aus s wird durch das erste Zeichen aus r ersetzt, etc. Für die Bindung gilt das gleiche, wie bei den anderen Operatoren dieses Abschnittes. Siehe *perl(1)* für eine ausführlichere Diskussion dieses Operators.

4.5 Subroutinen

In perl kann man Subroutinen definieren, indem man das Schlüsselwort sub, den Namen der Subroutine und dann einen Block, der den Rumpf der Routine darstellt, angibt:

```
sub unsinn {
 print "hallo\n";
 return "42=6*9";
}
```

Subroutinen in perl halten einige Überraschungen bereit. So werden die Argumente, die man an sie übergeben kann, nicht etwa im Kopf der Routine nach dem Namen aufgelistet, sondern in dem Vektor @_ übergeben. So enthält dieser nach dem Aufruf do unsinn(1,2) etwa die Werte 1,2.
Auch sind Variablen in perl erstmal global. In Subroutinen muss man mittels dem local oder dem my Konstrukt explizit lokale Variablen definieren, z.B.

```
 sub unsinn {
 local ($foo,$bar)=@_;
 my ($baz,$foobar)=($foo,$bar);

 print $foo+$bar . "\n";
}
```

Dies deklariert die lokalen Variablen $foo und $bar und weisst ihnen auch gleich den Wert des ersten bzw. zweiten Parameters der Routine zu. Die Variablen $baz und $foobar sind ebenfalls lokal. Der Unterschied zwischen beiden Konstrukten besteht darin, daß mit my definierte Variablen lexikalisch lokal zu dem Block sind, in welchem sie eingeführt werden. D.h. sie sind ausserhalb des Blockes komplett unsichtbar. Mittels local eingeführte Variablen überdecken ebenfalls gleichnamige Variablen in dem Block, allerdings sind sie auch in allen Subroutinen, die aus diesem Block aufgerufen werden, sichtbar[18]. Z.B. gibt das folgende Beispiel 20 aus:

```
 $bar=1;

 sub foobar {
  print $bar;
 }
```

[18]Dynamisches Scoping: Krank? Sicher.

```
sub nix {
 local $bar=20;

 &foobar;
}

&nix;
```

Es sei noch bemerkt, daß man in `perl` lokale Variablen in jedem Block einführen kann, also nicht nur in Subroutinen.
Um aus einer Subroutine Resultate zurückzugeben, gibt es in `perl` zwei Möglichkeiten: Man kann den `return` Operator benutzen (wie im ersten Beispiel dieses Abschnittes). Per Default wird als Ergebnis einer Subroutine der Wert des letzten in ihr ausgewerteten Ausdruckes zurückgegeben, so daß obiges Beispiel äquivalent ist zu:

```
sub unsinn {
 print "hallo\n";
 "42=6*9";
}
```

`perl` übergibt Werte an Subroutinen immer per Werte-Übergabe ('Call-by-value'). Will man z.B. eine Array in einer Subroutine verändern, muß man es explizit per Referenz ('call-by-name') übergeben. Dies geschieht, indem man bestimmte Namen per Referenz übergibt.
Eine der am meisten kranken Konstrukte in `perl` ist nun, daß diese Übergabe dadurch realisiert ist, daß beim Aufruf *alle* Objekte, mit einem bestimmten Namen, übergeben werden[19]. In der Subroutine kann man dann auf alle diese Objekte wieder per Referenz Bezug nehmen, z.B.:

```
sub unsinn {
 local (*anarray)=@_;

 $anarray{"Nix"}="bar";
}

 do anarray(*foo);
 do anarray(*baz);
```

Dieses Beispiel übergibt in den beiden Aufrufen einmal alle Objekte, deren Namen `foo` ist, bzw. alle, deren Namen `baz` ist. In der Routine wird dann mittels

[19] Also z.B. die skalare Variable `$foo`, der Vektor `@foo` und der assoziative Vektor `%foo`.

des `*anarray` der Name `anarray` per Referenz an diese Objekte angehängt. Die Zuweisung im Rumpf verändert also einmal den Vektor `%foo`, einmal den Vektor `%baz`.
Zum Schluß noch die Bemerkung, daß man eine Subroutine neben einem Konstrukt wie `do unsinn();` noch als `&unsinn;`, bzw. `&unsinn(1,2);` aufrufen kann.

4.6 Spezielle Variablen

Dieser Abschnitt listet einige der vielen speziellen Variablen von `perl` auf:

$| Wird diese Variable auf einen Wert !=0 gesetzt, so wird bei Ausgaben via `print` der Puffer der Datei sofort nach jeder Ausgabe auch weggeschrieben. Sonst sind die Ausgaben gepuffert und werden erst später bei Füllung des Puffers geschrieben.

$& Enthält, ähnlich wie bei `sed`, nach einem Matching den String, der auf das gesamte Pattern passte.

$_ Der Inhalt dieser Variablen wird oft als Default benutzt, wenn bei Operatoren ein Argument weggelassen wird.

$$ Die Prozessnummer des `perl`-Programmes.

$? Der letzte Exit-Code eines in einer Subshell ausgefÜhrten Kommandos.

$[Der Index des ersten Elements eines Vektors. Default: 0

$! Der Fehlercode nach einem Fehler bei z.B. einem `print` Enthält den Wert, der auch in der **C** Variablen `errno` verfügbar wäre, siehe *errno(2)*.

$@ Die Fehlermeldung im Klartext.

@ARGV Dieser Vektor enthält die Kommandozeilenparamter, mit denen das Programm aufgerufen wurde.

@INC Dieser Vektor enthält Namen von Directories, in denen nach `perl`-Programmen gesucht wird. Siehe `require`.

%ENV Dieser assoziative Vektor enthält die Umgebungsvariablen. Der Name der Variablen ist dabei der Schlüssel.

4.7 CGI-Scripte

4.8 Beispiele

Dieses Beispiel[20] bennent die Bezeichner in einem GTK-Sourcefile um[21] um. Nützlich, um eine neue Klasse auf dem Code einer alten zu basieren. Der Aufruf geschieht mittels `gtkrn` *Old-Name New-Name Files*

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

my $old= shift;
my $new= shift;

sub capitalize {
   my $str= shift;
   $str=~ s/^([a-z])/uc($1)/ei;
   $str=~ s/_([a-z])/uc($1)/gei;
   return $str;
}

sub replacestr {
  my ($head,$tail)= @_;
  if ($tail =~ /^(.*)$old(\w*)/i) {
   my $head2= $1;
   my $tail2= $2;
   if ($head eq "gtk_")
     { return "gtk_".$head2.$new.$tail2; }
   if ($head eq "GTK_")
     { return "GTK_".$head2.uc ($new).$tail2; }
   if ($head eq "Gtk")
     { return "Gtk".$head2.capitalize ($new).$tail2; }
  }
  return $head.$tail;
}

for $file (@ARGV) {
   open (F, $file) || die $!;
   open (G, ">$file.tmp") || die $!;
```

[20] Beigestuert von Henrik Theiling

[21] GTK ist ein Toolkit, mit welchem man graphische Komponenten, wie Fenster, etc., konstruieren kann.

```
    while (<F>) {
        s/(gtk_?)(\w+)/&replacestr($1,$2)/eig;
        print G;
    }
    close G;
    close F;
    rename ("$file.tmp", "$file");
}
```

4.9 Falltüren

perl stellt eine einzige Sammlung von Falltüren dar. Die ziemlich verwinkelte Semantik und der merkwürdige Aufbau einiger Konstrukte, zusammen mit vielen Default-Annahmen und Ersetzungen macht das Programmieren darin nicht gerade einfach.
Besonders dumm ist die Tatsache, daß Variablen durch ihre Benutzung eingeführt werden und somit Tippfehler fatale Folgen haben können. Durch das globale Scoping sind größere Projekte nur mit heftigsten Kopfschmerzen zu entwickeln.
Im Rest dieses Kapitels wurden schon einige Falltüren erwähnt, so daß dieser Abschnitt hiermit bereits enden kann.

4.10 Literatur

Es gibt vielfältige Literatur über perl. Besonders im O'Reilley Verlag sind eine ganze Reihe von Büchern zu perl erschienen. Diesem Kapitel lag [WS93] zu Grunde. Angesichts der grausigen Übersetzung dieses Werkes würde ich aber eher das Original empfehlen.

Kapitel 5

GNU `make`

5.1 Einführung

Bei der Erstellung von Programmen durchläuft man (auf der praktischen Seite) immer wieder denselben Vorgang:

1. Programm-Sourcecode erstellen/ändern

2. Einzelne Sourcefiles zu Objectcode-Files kompilieren

3. Die Objektcode-Files zum Programm zusammenlinken

Bei größeren Projekten kann man eine ganze Reihe von Sourcefiles haben. Bei jeder Änderung nun alle Sourcefiles neu zu kompilieren und die entstandenen Objectcode-Files zu linken ist damit eine ziemlich teure Angelegenheit[1]. Deshalb möchte man nach einer Änderung am Sourcecode nur die Files neu kompilieren, die von dieser Änderung betroffen sind. Als Beispiel sind in Abbildung 5.1 die Abhängigkeiten eines winzigen Projektes dargestellt. Dabei bedeutet z.B. der Pfeil von `foo.c` zu `foo.o`, daß das Objectcode-File `foo.o` vom Sourcefile `foo.c` abhängt, also bei einer Änderung von `foo.c` das File `foo.o` neu erzeugt werden muß.
Man sieht hier auch, daß eine Änderung in einem File, etwa `bar.h`, die Neugenerierung von mehreren Files, `foo.o` und `baz.o`, nötig macht.
Obwohl Abhängigkeiten in Form von Sourcecode–Objectcode Beziehungen die am häufigsten auftretenden sind, gibt es auch andere Formen von Abhängigkeiten und Generierung von Files aus anderen (Source-)Files. Z.B. steht dieses Script

[1] Auf einem 133MHz Pentium Rechner dauert z.B. die komplette Neugenerierung eines Betriebssystem-Kernels für NetBSD eine gute halbe Stunde. Dabei ist dieses Projekt mit 459 resultierendes Objectcode-Files noch relativ klein.

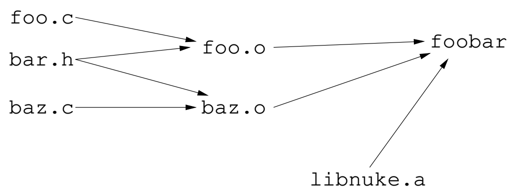

Abbildung 5.1: Abhängigkeiten in einem Projekt

```
foobar:  libnuke.a foo.o baz.o
       gcc -o foobar foo.o baz.o -lnuke

foo.o:  foo.c bar.h
       gcc -o foo.o foo.c -I../include -c

baz.o:  baz.c bar.h
       gcc -o baz.o baz.c -I../include -c
```

Abbildung 5.2: `Makefile` Beispiel

als Postscript-Datei im Netz zur Verfügung. Diese Postscript-Datei wurde mittels `dvips` aus einem `DVI`[2]-File erzeugt. Dieses wiederum ist die Ausgabe des `latex` Tools, das eine Datei mit Text und Formatierungsbefehlen, `Script.tex` als Eingabe nimmt.

GNU `make` ist ein Tool, mit welchem man die Abhängigkeiten zwischen Files und Kommandos zur Generierung von Files aus anderen Files angeben kann. Darüberhinaus bietet es auch noch die Möglichkeit, eigene *Variablen* zu definieren, die in Kommandos und Abhängigkeiten ausgewertet werden, sowie eine Reihe eingebauter *Funktionen*, mit deren Hilfe man etwa Filenamen umformen kann. Als Eingabe nimmt `make` ein einzelnes File mit dem Namen `Makefile`, in welchem alle Abhängigkeiten und Kommandos abgelegt sind.

In Abbildung 5.2 sieht man das zu den Abhängigkeiten in Abbildung 5.1 korrespondierende `Makefile`.

Neben dem GNU `make` gibt es eine ganze Reihe weiterer `make`-Varianten. Diese unterscheiden sich deutlich im Umfang ihrer features und sind nicht kompatibel zu GNU `make`. Da dieses aber den (m.E.) größten Funktionsumfang bietet, wird es hier ausschließlich vorgestellt. Man sei gewarnt, daß alles außer einfachen Regeln und Kommandos in anderen Versionen von `make` nicht so funktioniert, wie in GNU `make`!

[2]Das Ausgabeformat des LaTeX-Systems, siehe Kapitel 6

Im weiteren wird nun zunächst der Aufbau und Inhalt eines `Makefiles` dargestellt, bevor in Abschnitt 5.3 die *Regeln* genauer vorgestellt werden, mit denen man die Abhängigkeiten zwischen Files beschreibt. Im darauf folgenden Abschnitt werden die *Variablen* so wie sie von `make` bereitgestellt werden, behandelt. *Kommandos*, die für die Neugenerierung von Files verntwortlich sind, werden in Abschnitt 5.3.1 behandelt. Danach folgen noch Abschnitte über die Direktiven in einem `Makefile` und die Aufrufparameter des `make`-Tools. Danach werden die in `make` vordefinierten impliziten Regeln behandelt. Den Abschluß bilden ein Abschnitt über die `autoconf` und `automake` Tools, welche bei der Erstellung von protierbaren `makefiles` helfen, sowie Abschnitte mit Beispielen, Warnungen und etwas Literatur.

5.2 Makefiles

Die Abhängigkeiten zwischen Files und die Kommandos, um Files aus seinen Abhängigkeiten neu zu bauen, werden `make` in einem speziellen Eingabefile mitgeteilt. Gibt man `make` nicht mittels der `-f` *`File`* Option explizit den Namen dieses Files an, so wird nach einem File gesucht, welches den Namen `Makefile` oder `makefile` oder aber `GNUMakefile` trägt. Ein Makefile kann folgende Bestandteile enthalten:

- *Regeln* geben Abhängigkeiten und Kommandos an
- *Variablenzuweisungen* weisen `make`-Variablen Werte zu
- *Direktiven* steuern z.B. die konditionale Ausführung von Makefiles
- *Kommentare* werden eingeleitet durch `#` und reichen bis zum Ende der Zeilen, in welchen sie stehen.

Ein etwas ausführlicheres Beispiel eines `Makefiles`:

```
# Makefile f"ur Unix-Kurs Script
# Spezielles Ziel
.PHONY: all ps clean

# variablenzuweisungen
FIGSS=Abb.Libs.fig Abb.Link.fig Abb.Dyn.fig \
        Abb.Sed.Work.fig Abb.Make.fig1.fig
FIGSO=$(patsubst %.fig,%.eps,$(FIGSS))

MAG_DEFAULT=0.5
```

```
SRCS=Script.tex $(FIGSO)

# Regeln
all: Script.dvi
ps: Script.ps
# Regeln mit Kommandos
clean:
	rm -f $(OBJS) Script.ps *.dvi *.aux *.bbl \
	              *.blg *.log *.toc

Script.ps: Script.dvi
	dvips -o $@ $<

Script.dvi: $(SRCS)
	latex $<
	bibtex Script
	checklatex $<

# Implizite Regel
%.eps: %.fig
	fig2dev -L ps -p dummy -m $(word 1,$(MAG_$*) \
		$(MAG_DEFAULT)) $< >$@
```

Die Komponenten können in einem `Makefile` in beliebiger Reihenfolge auftreten. In den nun folgenden Abschnitten werden wir uns der Reihen nach den einzelnen Komponenten ausführlicher widmen.

5.3 Regeln

Regeln bestehen aus zwei Teilen

- Abhängigkeiten, welche die Abhängigkeit zwischen bestimmten Files angeben
- Kommandos, die zur Regenerierung abhängender Teile bei Änderungen dienen

Dabei dürfen die Kommandos auch weggelassen werden, wenn man nur die bloße Tatsache einer Abhängigkeit zwischen Files ausdrücken will[3]
Abhängigkeiten werden dabei durch zwei Komponenten beschrieben. Das *Ziel* ist der abhängige Teil. Die *Quellen* sind die Teile, von denen das Ziel abhängt. Aus

[3]I.A. wird es natürlich genau eine Regel geben, die Kommandos zum Bauen der Abhängigkeit enthält.

denen es also bei einer Änderung neu generiert wird. Z.B. ist in Abbildung 5.1 `baz.o` ein Ziel und `baz.c` und `bar.h` seine Quellen. Dies wird im zugehörigen `Makefile` in Abbildung 5.2 durch die Abhängigkeit

```
baz.o:  baz.c bar.h
```

ausgedrückt.

Ein Ziel kann also durchaus mehrere Quellen haben. D.h. wenn sich eine dieser Quellen ändert, muß auch das Ziel neu gebaut werden. Ebenso kann man aber in einer Abhängigkeit auch mehrere Ziele angeben. Dies dient der Abkürzung und so steht etwa die Abhängigkeit

```
baz.o foobar.o:  bar.h
```

für zwei Abhängigkeiten `baz.o: bar.h` und `foobar.o: bar.h`

Man kann in Anhängigkeiten auch Wildcards benutzen, wie z.B. in der tcsh. Diese werden dann beim Lesen des `Makefiles` gegen die Files im akt. Directory expandiert, etwa sagt `foo.o: *.h`, daß `foo.o` von allen Header-Files abhängt. Man beachte aber, daß die Expansion beim Lesen der Regeln passiert, eine Abhängigkeit der Form `foo: *.o` also nicht zum gewünschten Ergebnis (`foo` hängt von allen Objectcode-Files ab) führt, wenn die Objectcode-Files selbst noch generiert werden müssen und also beim Lesen des `Makefiles` noch nicht existieren.

Ziele und Quellen entsprechen meist realen Files, die und aus denen neu gebaut wird. Es ist aber auch möglich, Ziele zu definieren (und dann natürlich auch diese als Quellen zu benutzen), welche keinem File entsprechen. Solche Pseudo-Ziele stehen also einfach als eine Art Abkürzung für irgendwelche Tätigkeiten, die ausgeführt werden sollen. `make` ist es egal, ob die Kommandos einer Regel wirklich ein File erzeugen, das den Namen des Ziels trägt. Solange die Kommandos ohne Fehler ablaufen, geht `make` davon aus, daß die Regel das Ziel korrekt neu gebaut hat. Pseudo-Ziele benutzt man z.B. um eine Distribution „aufzuräumen", also etwa Objectcode-Files wieder zu entfernen. Dazu könnte man eine Regel der Form

```
clean:
        rm -f *.o
```

benutzen. Ein Pseudo-Ziel, das oft benutzt wird ist das Ziel `all`, welches die Compilierung des gesamten Projektes anstösst.

Probleme kann man bekommen, wenn aus Versehen wirklich mal ein File mit dem Namen `all` erzeugt wurde. Dann führt ein `make all` nicht zur Neugenerierung des Projektes, da ein File mit dem Namen ja bereits existiert[4]

Deshalb kann man `make` mitteilen, daß bestimmte Ziele Pseudo-Ziele sind. Dazu gibt man eine Abhängigkeit für das spezielle `.PHONY` an:

[4]Wenn man voraussetzt, daß alle Abhängigkeiten jÜnger als dieses File sind.

```
.PHONY: all
```

Dies teilt `make` mit, daß es jedes File `all` beim Bauen des Zieles `all` ignorieren und die Kommandos der Regel für `all` auf jeden Fall ausführen soll.
Ein anderes spezielles Ziel, `.SUFFIXES` ist in Verbindung mit impliziten Regeln wichtig, die in einem späteren Abschnitt besprochen werden.

5.3.1 Kommandos

Kommandos folgen auf die Angabe der Abhängigkeiten in einer neuen Zeile. Genauer gesagt fasst `make` jede Zeile, die auf eine Abhängigkeitsangabe folgt und *mit einem TAB beginnt* als eine Kommandozeile auf. Ein oft gemachter Fehler ist es, Kommandozeilen nicht mit einem TAB zu beginnen, was `make` dann zu manchmal nicht offensichtlichen Fehlermeldungen veranlasst (da es die Zeile dann als Abhängigkeit auffasst). Es kann mehrere Kommandozeilen in einer Regel geben. Die Kommandos werden dann in der Reihenfolgen, in der sie im File auftauchen, ausgeführt. Jede Kommandozeile wird[5] an eine eigene Subshell übergeben, wobei als Vorgabe `/bin/sh` eingestellt ist. Da jede Zeile an eine eigene Shell zur Ausführung übergeben wird, sollte man sich bewußt sein, daß Kommandos, wie etwa `cd`, die den lokalen Status einer Shell ändern, in nachfolgenden Kommandozeilen keine Auswirkungen hinterlassen. Natürlich kann man in einer Kommandozeile einen beliebigen, von der Shell interpretierbaren Ausdruck hinschreiben, etwa

cd src; make; cd ../doc; make

Dieser wird dann von *einer* Shell abgearbeitet und zeigt auch die gewünschten Effekte.
Manchmal ist es nötig, längere Kommandos anzugeben. Der Lesbarkeit wegen kann man solche Kommandos über mehrere Zeilen verteilen, wenn man am Ende jeder Zeile einen Rückstrich '\' angibt (natürlich, bis auf die letzte Zeile). Trotzdem wird dies dann von `make` intern wieder zu einer Zeile zusammengesetzt. In Kommandos sind neben den im `Makefile` definierten Variablen noch einige spezielle Variablen verfügbar, die die Namen der Abhängigkeiten u.ä. angeben. Diese werden dann vor jeder Ausführung der Kommandos automatisch belegt und expandiert, siehe den nächsten Abschnitt.
Tritt bei der Ausführung eines Kommandos ein Fehler auf, angezeigt durch einen Rückgabewert des Kommandos $\neq 0$, so geht `make` davon aus, daß der Neubau des zugehörigen Zieles fehlgeschlagen ist[6]. Als Folge davon wird der komplette `make`-Prozess abgebrochen.

[5] Nach Variablenexpansion, etc.
[6] Etwa, weil die Compilierung eines Files wegen Syntaxfehlern in der Eingabe misslang.

Es kann manchmal sein, daß das Fehlschlagen eines Kommandos nicht unbedingt bedeutet, daß der Generierungsprozess fehlschlug, z.B. wenn in einem Kommando ein Directory (etwa /usr/local/man) kreiert werden soll, welches bereits existiert. Man kann daher solchen Kommandos ein '-' voranstellen, was zur Folge hat, daß ein Fehlschlag dieses Kommandos ignoriert wird, etwa '-mkdir /usr/local/man'-

make gibt in der Standardeinstellung jedes Kommando (nach der Expansion von Variablen) auf stdout aus, bevor es an die Shell zur Ausführung übergeben wird. Dies kann manchmal unerwünscht sein, weshalb man durch Vorstellen eines '@' vor das Kommando dieses Verhalten unterdrücken kann, etwa '@echo "Installing..."'

5.4 Variablen

In make gibt es fünf Arten von Variablen:

- Auf der Kommandozeile können Variablen gesetzt werden, etwa durch make CC=gcc. Dies setzt die Variableb CC auf den Wert gcc. Soclhermassen gesetzte Variablen können nicht mehr durch normale Zuweisungen im Makefile verändert werden. Will man die verändern, muß man die override Direktive benutzen, s.u.

- Für jede (globale) Umgebungsvariable, die beim Aufruf von make im Environment der Shell definiert war, wird eine make-Variable gleichen Namens und Inhaltes angelegt. Davon ausgenommen sind nur die folgenden Variablen:

  SHELL Wird immer auf /bin/sh gesetzt

  MAKE Wird auf die Aufrufsequenz von make gesetzt. Nach einem Aufruf der Form gnumake CC=gcc all wird MAKE also auf den Wert gmake gesetzt.

  MAKEFLAGS Enthält die Optionen und Variablen, die auf der Kommandozeile angegeben wurden. Im vorigen Beispiel würde MAKEFLAGS also den Wert CC=gcc enthalten.

- Variablen, die im Makefile *lazy* gesetzt werden, z.B. durch eine Zeile der Form CFLAGS=$(FLAGS) -Wall. Dies setzt die Variable CFLAGS auf den Inhalt der Variablen FLAGS, gefolgt von -Wall. Die Variablen heissen lazy, weil der Inhalt der Variablen FLAGS erst dann eingesetzt wird, wenn diese Variable schließlich expandiert wird. Es wird also der bei jeder Expansion gerade aktuelle Wert in FLAGS eingesetzt. Dies bedeutet aber auch, daß Konstrukte der Form `CFLAGS=$(CFLAGS) -Wall` verboten sind[7], da sie bei der Expansion zu unendlicher Rekursion beim Einsetzen von Variablenwerten führen.

- Variablen, die in einem `makefile` *eager* gesetzt werden, etwa wie in `CFLAGS:=$(CFLAGS) -Wall`. Hier wird der Variablen `CFLAGS` der gerade aktuelle Wert der Variablen `CFLAGS` zugewiesen, gefolgt von `-Wall`. Dadurch unterscheiden sich Variablen diesen Typs von *lazy* Variablen: Lazy Variablen expandieren ihren Wert bei jeder Expansion der Variablen, während eager Variablen ihre rechte Seite nur einmal, bei der Definiton der Variablen expandieren und dann bei jeder Expansion der Variablen diesen expandierten Wert einsetzen.

- Automatische Variablen treten nur in Kommandos auf. Sie werden bei der Anwendung einer Regel, wenn also ein Ziel aus seinen Quellen neu gebaut werden muss, automatisch auf bestimmte Werte gesetzt, etwa den Namen des Ziels (`$@`), etc. Eine komplette Liste aller automatischen Variablen folgt weiter unten.

Variablenreferenzierung wird geschrieben, indem man ein Dollarzeichen, gefolgt von dem Namen der Variablen, der im übrigen nur Buchstaben, gefolgt von Buchstaben, Unterstrichen oder Ziffern, entahlten darf, in runden Klammern, etwa `$(CFLAGS)`.
Variablen können in einem `Makefile` in rechten Seiten von Variablendefinitonen, Zielen oder Quellen in Abhängigkeiten, Kommandos und Direktiven auftreten.
`make` expandiert Variablen rekursiv, d.h. nach jeder Expansion einer Variablen wird die komplette Zeile wieder neu expandiert. Dies hat zur Folge, daß man als Namen einer Variablen wieder eine Variablenreferenz schreiben kann, etwa ist folgendes vollkommen legal und expandiert bei der Expansion der Variablen `C` schliesslich zu dem Text '`y`':

```
A=B
B=y
C=$($(A))
```

Wird `C` irgendwo benutzt, also `$(C)` expandiert, expandiert dies zunächst zu `$(B)`, da die Referenz zur Variablen `A` zuerst expandiert wird. Dies expandiert dann im zweiten Schritt zu `y`.
Es gibt einige Variablen, die eine spezielle Bedeutung haben:

MAKE

[7]Und auch in `make` zu einer Fehlermeldung führen.

MAKEFLAGS Wurden schon weiter oben erwähnt

VPATH Enthält eine Liste von Directories, durch Doppelpunkte getrennt, in denen nach Files gesucht werden soll. Hat man z.B. alle Header-Dateien in einem eigenen Verzeichnis `include` untergebracht müsste man eigentlich bei jeder Regel etwas der Form `foo: foo.o include/bar.h` schreiben. `make` sucht nun aber auch in allen Verzeichnissen, welche in `VPATH` angegeben sind, nach Quellen und Zielen, so daß nach der Zuweisung `VPATH=include` die obige Anhängigkeit einfach als `foo: foo.o bar.h` geschrieben werden kann. Neben dieser Möglichkeit bietet `make` auch noch die Option, nur nach Files mit bestimmter Endung in bestimmten Directories zu suchen. Siehe dazu die Beschreibung der `vpath` Direktive in Abschnitt 5.5.

Da man an eine lazy Variable nicht durch eine einfache Zuweisung einen Wert anhängen kann (s.o.), stellt `make` noch einen Zuweisungsoperator bereit, der Werte an die Definiton einer Variablen anhängt: '+='. So liefert etwa folgende Zuweisung den gewünschten Effekt, `-Wall` an die akt. Definition von `CFLAGS` anzuhängen:

```
CFLAGS+=-Wall
```

Möchte man einer Variablen nur dann einen Wert zuweisen, wenn sie nicht schon vorher (etwa über die Kommandozeile) definiert wurde, dann benutzt man den '`?=`' Zuweisungoperator:

```
CFLAGS ?= -Wall -O2 -g
```

Dies weißt `CFLAGS` nur dann einem Wert zu, wenn die Variable bis dahin unbekannt (nicht definiert) war.

5.4.1 Automatische Variablen

Automatische Variablen können nur in Kommandos auftauchen und erhalten ihren Wert durch die Abhängigkeiten der Regel, zu der sie gehören. So gibt es etwa eine automatische Variable `$@`, der der Wert des Zieles der Regel zugewiesen wird. `make` kennt die folgenden automatischen Variablen:

`$@` Der Name des Ziels der Regel.

`$<` Der Name der *ersten* Quelle in der Abhängigkeit.

`$?` Alle Quellen, die jünger als das Ziel sind.

`$^` Alle Quellen, getrennt durch ein Space. Aus dieser Liste werden doppelte Einträge entfernt.

`$+` Wie `$^`, nur das doppelte Einträge nicht entfernt werden.

`$*` Der Stamm einer impliziten Regel, siehe Abschnitt 5.7.

Beispiel:

```
foo: foo.o bar.o
      gcc -o $@ $<
# Resultiert in 'gcc -o foo foo.o'

bar: foo.o bar.o bar.o
      gcc -o $@ $^
# Resultiert in 'gcc -o bar foo.o bar.o'

baz: baz.o foo.o baz.o
      gcc -o $@ $+
# Resultiert in 'gcc -o baz baz.o foo.o baz.o'
```

5.4.2 Funktionen

`make` bietet eine ganze Reihe von Funktionen, die bei der Expansion von Text ähnlich wie Variablen ausgewertet werden.
Die Syntax eines Funktionsaufrufes hat dann die Form

`$(`*`Funktions-Name Argumente...`*`)`

Funktionen dienen dazu,

- Text umzuformen
- Patternmatching zu machen
- Schleifen zu realisieren
- Kommandos in einer externen Shell aufzurufen

Textmanipulation

Text wird in `make` manchmal als eine Folge von *Worten* aufgefasst (z.B. Filenamen). Worte werden dabei durch Leerzeichen oder TABs voneinander getrennt.
Zur Manipulation von Text stehen folgende Funktionen zur Verfügung:

- `$(subst` *`from`*`,`*`to`*`,`*`text`*`)` Ersetzt jedes Vorkommen von *from* in dem Text *text* durch *to*.

- `$(patsubst pattern,repl,text)` Ersetzt in jedem Wort in *text*, welches auf das Pattern *pattern* passt, dieses Pattern durch *to*. In dem Pattern kann ein '`%`' auftauchen, das eine beliebige Folge von Zeichen beschreibt. Taucht `%` in *to* auf, so wird es durch den Match ersetzt, der durch das `%` im *Pattern* vorgenommen wurde.

Als Beispiel dieses Fragment eines `Makefiles`:

```
FILES=a.c b.c c.c
OFILES=$(patsubst %.c,%.o,$(FILES))

AFILES=yacc.tab.c yacc.tab.h
ACFILES=$(subst .tab,,$(AFILES))
```

Hier expandiert `$(OFILES)` später zu `a.o b.o c.o`.
Die Variable `$(ACFILES)` expandiert zu `yacc.c yacc.h`.

- `$(strip text)` Diese Funktion entfernt allen Whitespace[8] vom Anfang und Ende des Textes.

- `$(findstring find,text)` Liefert *find*, wenn *text find* enthält, den leeren Text, sonst.

Als erläuterndes Beispiel, folgendes `Makefile`:

```
SOPTS=$(strip $(CFLAGS))
ifneq ($(SOPTS),)
 ISWALL=$(findstring -Wall,$(SOPTS))
ifneq ($(ISWALL),)
 # -Wall enabled
endif
endif
```

Hier wird zuerst die Variable `$(SOPTS)` als der Wert von `$(CFLAGS)` definiert, mit allem Whitespace am Anfang und Ende entfernt. In der zweiten Zeile wird dann geprüft, ob der Wert dieser Variablen der leere Text ist. Der Grund für die Entfernung des Whitespaces ist der, daß Text, welcher z.B. nur aus Lerrzeichen besteht verschieden ist von dem leeren Text (der einfach keine Zeichen enthält). Da es in Variablendefinitionen durchaus passieren kann, daß sich Whitespace an den Anfang oder das Ende einer Definition verirrt, muß man also vor der Prüfung auf den leeren Text erst den Whitespace entfernen.

[8] Spaces, TABs

In dem Beispiel wird dann in der dritten Zeile der Variablen `$(ISWALL)` der Wert `-Wall` zugewiesen, wenn dieser Text in `$(SOPTS)` autaucht, der leere Text sonst. Mann kann dann einfach testen, ob diese Option in den Flags gegeben wurde (4. Zeile).

- `$(filter` *`pattern...`*`,`*`text`*`)` Filtert Worte aus *text*, die auf keins der Pattern *pattern...* passen.
- `$(filter-out` *`pattern...`*`,`*`text`*`)` Filtert Worte aus *text*, die auf eins der *pattern...* passen.
- `$(sort` *`text`*`)` Sortiert alle Worte in *text*.

Beispiel:

```
FILES=a.c foo.h b.c baz.y
CFILES=$(filter %.c,$(FILES))
NONCHFILES=$(filter-out %.c %.h,$(FILES))
SFILES=$(sort $(FILES))
```

Hier wird `$(CFILES)` der Wert `a.c b.c` zugewiesen, da diese Worte auf das Pattern `%.c` passen. Die Variable `$(NONCHFILES)` erhält den Wert `baz.y`, da nur dieses Wort auf keins der Pattern `%.c` oder `%.h` passt. Die Variable `$(SFILES)` expandiert zu `a.c b.c baz.y foo.h`, der sortierten Wortliste.

Filenamen-Manipulation

- `$(dir` *`Namen...`*`)` Liefert den Directory-Teil für jeden Namen
- `$(notdir` *`Namen...`*`)` Liefert den Filenamen-Teil für jeden Namen

Ein Beispiel:

```
FILES=../src/foo.c bar.c nix/
DFILES=$(dir $(FILES))

FFILES=$(notdir $(FILES))
```

Hier expandiert `$(DFILES)` zu `../src ./ nix/`. Da das zweite Wort von `$(FILES)` keinen Directory-Teil hat, wird statt des leeren Textes ein `./` eingefügt.
Der Variablen `$(FFILES)` wird der Wert `foo.c bar.c` zugewiesen. Da das letzte Wort von `$(FILES)` keinen Filenamen-Teil hat, wird es durch das leere Wort ersetzt.

- `$(suffix Namen...)` Liefert den Suffix jeden Namens
- `$(basename Namen...)` Liefert den Namen ohne Suffix

Z.B.:

```
FILES=../src/foo.c bar.c nix/

SFILES=$(suffix $(FILES))
BFILES=$(basename $(FILES))
```

Hier wird `$(SFILES)` zu `.c .c`, da das letzte Wort von `$(FILES)` keinen Filenamen-Teil hat, wird seine Endung auch zum leeren Wort. Der Variablen `$(BFILES)` wird der Wert `../src/foo bar nix/` zugewiesen. Anders, als bei dem Shell-Kommando `basename` wird also der Directory-Teil nicht abgeschnitten.

- `$(addsuffix` *Suffix*`,`*Namen*`...)` Fügt *Suffix* an jeden Namen in *Namen* an.
- `$(addprefix` *Prefix*`,`*Namen*`...)` Fügt *Prefix* vor jeden Namen an.
- `$(join` $Liste_1$`,`$Liste_2$`)` Fügt Worte aus den Listen aneinander. Dabei wird dem ersten Wort der ersten Liste das erste Wort der zweiten Liste angehängt (ohne Whitespace). Dem zweiten Wort der ersten Liste das zweite Wort der zweiten Liste, etc. Diese neu gebildeten Wort bilden das Ergebnis der Funktion.

Beispiel:

```
FILES=a b c

CFILES=$(addsuffix .c,$(FILES))

XFILES=$(addprefix foo_,$(CFILES))
# Liefert foo_a.c foo_b.c foo_c.c

YFILES=$(join $(FILES),.c .h .c)
# Liefert a.c b.h c.c
```

Der Variablen `$(CFILES)` wird der Wert `a.c b.c c.c` zugewiesen, die Variable expandiert zu `foo_a.c foo_b.c foo_c.c`. Schließlich wird der Variablen `$(YFILES)` der Wert `a.c b.h c.c` zugewiesen.

- `$(word` *`N`*`,`*`Text`*`)` Liefert das N-te Wort aus *Text*.
- `$(wordlist` *`S`*`,`*`E`*`,`*`Text`*`)` Liefert das S-te bis E-te Wort.
- `$(words` *`Text`*`)` Anzahl Worte in *Text*.

Beispiel:

```
FILES=a b c d

NFILES=$(words $(FILES))
# Liefert 4

SECOND=$(word 2,$(FILES))
# Liefert b

SECOND_TO_LAST=$(wordlist 2,4,$(FILES))
# Liefert b c d
```

- `$(wildcard` *`Pattern`*`)` Expandiert das *Pattern* im aktuellen Directory, z.B. `$(wildcard *.c)`

Schleifen

- `$(foreach` *`Var`*`,`*`Liste`*`,`*`Body`*`)` Bindet die Variable *Var* nacheinander an die Worte aus *Liste* und fügt die Auswertung von *Body* an das Ergebnis an.

Beispiel:

```
FILES=a b c d
XFILES=$(foreach x, $(FILES), $(x).c)
# Liefert a.c b.c c.c d.c
```

Shell & Sonstiges

- `$(origin` *`var`*`)` Liefert eine Zeichenkette, die den Typ der Variablen *var* angibt. Folgende Typen können auftreten:

  undefined Die Variable wurde nie definiert.

  default Die Variable hat einen voreingestellten Wert.

  environment Die Variable kommt aus dem Environment.

  file Die Variable wurde in einem `Makefile` definiert.

command line Die Variable wurde auf der Kommandozeile definiert.

override Der Variablen wurde mittels der `override` Direktive ein Wert gegeben, siehe Abschnitt 5.5.

automatic Die Variable ist eine automatische Variable.

- `$(shell` *`Args`*`)` Führt das Kommando *Args* in einer Subshell aus und liefert dessen Ausgabe als Ergebnis.

Beispiel:

```
ifneq ($(origin PATH),environment)
PATH=$(shell echo $$PATH)
endif
```

5.5 Direktiven

Direktiven dienen dazu

- Nur Teile eines `Makefiles` einzulesen, abhängig von bestimmten Bedingungen.
- Andere `Makefiles` einzubinden
- Variablenwerte zu überschreiben oder deren Gültigkeitsbereich zu ändern
- Suchpfade für Quellen zu setzen

Direktiven stehen an den Stellen, an denen auch Variablenzuweisungen oder Regeln stehen dürfen.

5.5.1 Conditionale

Conditionale haben die folgende generelle Form:
`if`*Bedingung Argumente*
True-Teil
`else`
False-Teil
`endif`
Falls die *Bedingung* zutrifft, wird der *True-Teil* verwendet (gelesen), ansonsten der *False-Teil*. Hierbei kann das `else` und der *False-Teil* auch fehlen. Natürlich kann man Conditionale auch schachteln.
Es sei angemerkt, daß `make` die Auswertung beim Einlesen des `Makefiles` vornimmt, nicht zur Auswertungszeit der Regeln.
Es gibt folgende Bedingungen für die Conditionale:

ifdef ***Variable*** Diese Bedingung ergibt 'wahr', falls die Variable *Variable* definiert ist.

ifndef ***Variable*** Diese Bedingung ergibt 'wahr', falls keine Variable *Variable* definiert ist.

ifeq (*A*,*B*) Ergibt 'wahr', falls die Texte *A* und *B* gleich sind.

ifneq (*A*,*B*) Ergibt 'wahr', falls die Texte *A* und *B* ungleich sind.

Bei den letzten beiden Conditionalen kann man statt `(A,B)` auch noch die folgenden Formen benutzen, die äquivalent sind:

- `"A" "B"`
- `'A' 'B'`

Beispiel:

```
ifdef CC
 # Variable CC definiert
 ifeq ($(CFLAGS),)
    # CFLAGS ist leer
  CFLAGS=-Wall -O3 -g
 else
    # CFLAGS nicht leer
  CFLAGS+=-Wall
 endif
endif
```

5.5.2 Direktiven für Variablen

Variablen, die auf der Kommandozeile von `make` gesetzt wurden, können normalerweise nicht mehr durch Variablenzuweisungen überschrieben werden, d.h. Zuweisungen an sie haben keinen Effekt. Möchte man jetzt aber sichergehen, daß eine Variable einen bestimmten Wert zugewiesen bekommt, kann man die `override` Direktive benutzen. Sie kann in drei Formen auftreten:

override ***Var=Value***

override ***Var:=Value***

override ***Var+=Value***

Diese Varianten entsprechen den gleichnamigen Zuweisungsoperatoren auf Abschnitt 5.4. Der Effekt ist der, daß die Variable auf jeden Fall den Wert auf der rechten Seite der Zuweisung erhält, auch wenn sie vorher auf der Kommandozeile definiert wurde. Da dieses Feature die Möglichkeiten des Benutzers, Variablen auf fixe Werte zu zwingen (etwa mittels `CC=gcc` den Compiler auf den GCC festzulegen) torpediert, sollte es nur vorsichtig eingesetzt werden.
Ruft man in einem `Makefile` `make` wieder via eines Kommandos auf, dann werden die Werte der Variablen des aktuellen `make`-Prozesses nicht an das Sub-`make` weitergegeben. Manchmal ist dies aber wünschenswert, wenn nämlich per Variablen wichtige Konfigurationsparameter festgelegt werden. Traditionell behilft man sich, indem man dem Aufruf von `make` (im `Makefile`) die Variablen explizit per Zuweisung als Argumente zum Aufruf mitgibt. `make` bietet aber auch die Möglichkeit, Variablen zu exportieren. Dies entspricht dem Export von Variablen in der Bourne-Shell (und benutzt auch die gleiche Syntax):

export *Var*

export *Var=Value*

export *Var:=Value*

export *Var+=Value*

Dies hat zur Folge, daß die Variable *Var* auch in Sub-`make`s mit ihren aktuellen Werten definiert sind. Wie in der Bourne-Shell kann man das `export` auch vor eine Zuweisung setzen, um eine Zeile zu sparen.
Will man eine Variable wieder lokal machen, also ihren Export verhindern, benutzt man die folgende Direktive:

unexport *Var*

5.5.3 Sonstige Direktiven

Um den Text eines anderen Files in ein `Makefile` einzubinden, benutzt man die folgende Direktive:

include *File*

Diese Direktive wird ebenfalls nur beim Einlesen des `Makefiles` ausgewertet. Der Inhalt des angegebenen Files wird dann eingelesen und so behandelt, als hätte er an der Stelle dieser Direktive gestanden.
Diese Direktive ist besonders bei großen Projekten extrem nützlich, wenn Standardeinstellungen in einem File gehalten werden können, das dann von verschiedenen anderen `Makefiles` eingebunden werden kann.

In Abschnitt 5.4 wurde schon die spezielle Variable `VPATH` vorgestellt, in welcher Pfade angegeben werden können, in denen nach Quellen gesucht wird. Allerdings hat diese Variable den Nachteil, daß sie für alle Quellen benutzt wird, egal ob es sich bei den Quellen um **C**-Files oder Header-Files handelt.
`make` stellt die `vpath` Direktive bereit, mit der für spezifische Quellen Suchpfade gesetzt werden können:

vpath *Pattern Path* Setzt den Suchpfad für alle Quellen, die auf das Pattern *Pattern* passen. In dem Pattern steht `%` für eine beliebige Folge von Zeichen. Z.B.: `vpath %.h ../include:/usr/include` setzt den Suchpfad für Header-Dateien

vpath *Pattern* Löscht den Suchpfad für alle Quellen, die auf *Pattern* passen.

vpath Löscht alle Suchpfade

5.6 Aufrufparameter

`make` kennt eine ganze Reihe von Kommandozeilen-Parametern:

`make` *Optionen Zuweisungen Ziele*

Neben der Definition von Variablen auf der Kommandozeile (*Zuweisungen*), die schon in Abschnitt 5.4 besprochen wurden und der Angabe von Zielen (*Ziele*), die neugebaut werden sollen, gibt es auch noch eine Reihe von Optionen:

-f *File* Liest das angegebene *File* als Makefile (statt der Defaults)

-n Führt keine Kommandos aus, sondern gibt nur auf dem Bildschirm aus, welche Kommandos ausgeführt werden würden um die Ziele neu zu bauen.

-k Bricht nach einem Fehler in einem Kommando nicht ab, sondern versucht noch andere Sub-Ziele neu zu bauen die nicht von dem fehlgeschlagenen Ziel abhängen.

-C *Verzeichnis* Wechselt vor dem Lesen des `Makefiles` in das angegebene *Verzeichnis*. Nützlich für `make`-Aufrufe in Unterverzeichnissen.

-i Ignoriert alle Fehlermeldungen von Kommandos und nimmt an, daß alle Kommandos immer ohne Fehler ausgeführt wurden.

-q Versucht nicht, Ziele neu zu bauen, sondern gibt den Exit-Status 0 zurück, wenn alle Ziele jünger als ihre Quellen sind. Müssten Kommandos ausgeführt werden, um Ziele zu aktualisieren, wird ein Wert $\neq 0$ zurückgegeben.

-s Verhindet, daß die Kommandos vor der Ausführung noch auf dem Bildschirm ausgegeben werden.

5.7 Implizite Regeln

Bisher wurde immer angenommen, daß man alle Abhängigkeiten einer Regel namentlich aufführt. Es kann aber auch nützlich sein, eine Regel für alle Ziele/Quellen zu haben, die auf ein bestimmtes Pattern passen, etwa eine Regel um aus C-Files (*.c) Objectcode-Files (*.o) zu machen. `make` bietet zu diesem Zweck implizite Regeln. Hierbei wird das Ziel durch ein Pattern beschrieben, wobei das '`%`' für eine beliebige Folge von Zeichen steht. In den Quellen kann dann auch ein '`%`' auftauchen, das dann durch den Match des Patterns im Ziel ersetzt wird. Zum Beispiel würde folgende Regel zur Compilierung von C-Files zu Objectcode-Files verwendet werden können:

```
%.o: %.c
	$(CC) -o $@ -c $<
```

Wenn nun z.B. ein Ziel namens `foo.o` neu gebaut werden muß, dann wird `%` auf `foo` gematcht und nach einer Quelle `foo.c` gesucht.
Da die Namen der Ziele und Quellen in impliziten Regeln nicht mehr direkt angesprochen werden können, muß man in ihnen automatische Variablen verwenden, die an bestimmte Teile von Ziel und/oder Quellen gebunden werden, siehe Abschnitt 5.4. Es gibt eine automatische Variable, die nur in impliziten Regeln definiert ist: `$*` Sie wird gerade an den Stamm, also den Teil, welcher das `%` matcht, gebunden. In obigem Beispiel würde `$*` an `foo` gebunden werden.
Natürlich kann man bei impliziten Regeln auch Quellen angeben, die kein Pattern enthalten. Dies ist ein einfacher Weg, alle Ziele eines bestimmten Typs von einem File (etwa einem generellen Header-File) abhängig zu machen.
Zu erwähnen ist noch, daß `make` implizite Regeln immer nur anwendet, wenn es keine expliziten Regeln[9] für das Ziel gibt. So kann man in Spezialfällen (für bestimmte Files) die implizite Regel durch Angabe einer konkreten überstimmen. Es ist aber möglich, explizite Regeln zu geben, die keine Kommandos enthalten, also nur Abhängigkeiten hinzufügen. Diese ergänzen die implizite Regel, überschreiben sie aber nicht.
`make` hat bereits eine ganze Reihe von impliziten Regeln vordefiniert, etwa zur Compilierung von C-Files zu Objectcode. Eine komplette Liste findet man in [SM98].

[9] Mit Kommandos

5.8 makedepend/autoconf

5.8.1 Abhängigkeiten generieren

Bei großen Projekten steigt die Anzahl der Abhängigkeiten zwischen Files meist rapide an. Vergißt man, eine Abhängigkeit in das Makefile einzutragen, kann dies zur Folge haben, daß ein Ziel nach einer Ändrung an einer seiner Quellen nicht neu gebaut wird, was zu extrem schwer zu findenden Fehlern führt[10]. Deshalb wäre es wünschenswert, wenn man Abhängigkeiten automatisch generieren lassen könnte.
Für **C**-Files gibt es ein Programm, makedepend, welches für eine Sammlung von Source-Files die Header-Files in Abhängigkeiten der Objectcode-Files ausgibt.
Da Header-Files wiederum anderer Header-Files einbinden können, ist diese Aufgabe nicht vollkommen trivial. Nehmen wir die Abhängigkeiten aus Abbildung 5.1; nehmen wir weiter an, daß das File bar.h noch ein File ../boo.h inkludiert.
Ein Aufruf von makedepend foo.c baz.c würde dann folgende Abhängigkeiten generieren:

```
foo.o: foo.c bar.h ../boo.h
baz.o: baz.c bar.h ../boo.h
```

Per Default hängt makedepend diese Ausgaben an ein bestehendes Makefile im akt. Verzeichnis an. Genauer gesagt wird ein vorhandenes Makefile gelesen und nach einer Zeile der Form

```
# DO NOT DELETE THIS LINE -- make  depend  depends  on it.
```

gesucht. Alles nach dieser Zeile wird gelöscht und durch die neuen Abhängigkeiten ersetzt. So muß man nicht vor jedem Aufruf von makedepend das Makefile neu generieren.
Man kann aber mittels der Option -f*File* auch ein alternatives *File* angeben, in welches die Abhängigkeiten geschrieben werden. Dieses kÖnnte man dann etwa mittels der include Direktive in das eigentliche Makefile einbinden.

5.8.2 Configurationsfiles erstellen

Wenn man portable Software schreiben will (muß), dann hat man zwei Probleme

- Feststellung der Fähigkeiten eines Zielsystems.

[10]Was der Author bei der Entwicklung eines Device-Drivers schmerzhaft erfahren musste.

- Bereitstellung fehlender Funktionalität für ein bestimmtes System.

Der erste Punkt ist oft schwieriger, als man denkt. So gibt es derzeit so viele Varianten verschiedenster UNIX und andersartiger Systeme, daß es kaum möglich ist, statisch alle Fähigkeiten eines Systems zu kennen. Auch soll die Software ja auch auf Systemen laufen, die zwar bestimmte Fähigkeiten[11] besitzen, die aber zum Zeitpunkt der Programmierung des Systems dem Entwickler nicht bekannt waren.
Während man bei dem zweiten Punkt auf sich selbst gestellt bleibt[12], gibt es für den ersten Punkt ein Packet, das einen unterstützt.
Das `autoconf` Packet unterstützt nun diese Art der Portabilität. Als Voraussetzung dafür muß die Software schon portabel angelegt sein. Etwa muß für bestimmte Features des Betriebssystems eine Abfrage im Code eingebaut sein, die im Falle des Fehlens dieses Features einen Workaround anwendet.
Dies geschieht (wenn man in **C** programmiert) meistens, indem man bestimmte Präprozessorsymbole einführt, die im Falle des Vorhandenseins eines Features definiert sind, sonst undefiniert[13]. Für bestimmte Werte im `Makefile` selbst, etwa das Verzeichnis, in welches man die Software installieren will, oder bestimmte zu benutzende Programme, kann man dort `make`-Variablen einführen, etwa `$(CC)` für den zu verwendenden C-Compiler.
Um diese Informationen zu erhalten, läßt man auf der Zielmaschine zur Installation ein Configurations-Script laufen:

- Dieses führt Tests auf die in Frage kommenden Eigenschaften aus
- erzeugt `Makefiles` aus Schablonen `Makefile.in`
- erzeugt ein Header-File `config.h` aus `config.h.in`

In dem erzeugten `Makefile` werden bestimmte Platzhalter durch gefundene Werte, etwa den Pfad zu dem zu benutzenden C-Compiler, ersetzt. Das Header-File enthält dann Definitionen für C-Präprozessor Symbole, abhängig von den angetroffenen Features.
Ein Beispiel für den Lauf eines solchen `configure` Scriptes:

```
> ./configure --prefix=$HOME --enable-shared
...
creating Makfile
creating src/Makefile
creating config.h
```

[11] Etwa einen bestimmten Systemaufruf oder Libraryroutine.

[12] D.h. wenn das Zielsystem keine `vsprintf` Routine kennt, muß man halt selbst eine (abgespeckte) Version davon bereitstellen.

[13] Im Sinne der `#ifdef` Präprozessor-Direktiven.

Hier wurden zwei `Makefiles` erzeugt: Das eine im aktuellen Verzeichnis, das andere im Verzeichnis `src`.
Beispiel für ein `Makefile.in`

```
srcdir = @srcdir@
VPATH = @srcdir@

CC = @CC@

INSTALL = @INSTALL@
INSTALL_PROGRAM = @INSTALL_PROGRAM@
INSTALL_DATA = @INSTALL_DATA@

install: all
        ${INSTALL_PROGRAM} indent ${bindir}/gindent
```

In dieser Schablone werden alle Platzhalter der Form `@`*Name*`@` durch Werte ersetzt, die durch das Configurations-Script bestimmt werden. So wird etwa `@CC@` durch den Namen des C-Compilers ersetzt, der benutzt werden soll; `@srcdir@` durch den Namen des Verzeichnisses, in welchem der Source-Code des Projektes steht. `@INSTALL@` wiederum wird durch den Namen eines BSD-`install` kompatiblen Installationsprogrammes ersetzt, etc.
Beispiel für ein `config.h.in`:

```
/* config.h.in. Generated automatically from
   configure.in by autoheader.*/

/* Define if on AIX 3.
   System headers sometimes define this.
   We just want to avoid a redefinition error message.  */
#ifndef _ALL_SOURCE
#undef _ALL_SOURCE
#endif

/* Define if using alloca.c.  */
#undef C_ALLOCA

/* Define to empty if the keyword does not work.  */
#undef const
```

In dieser Schablone für das Header-File kann man neben normalen Direktiven (hier dem `_ALL_SOURCE` Teil) Zeilen der Form

`#undef` *`Symbol`*

einfügen. Diese werden dann von dem Configurations-Script durch Zeilen der Form `#define` *Symbol* ersetzt, wenn das entsprechende Feature vorhanden ist. Man braucht dieses File meistens nicht selbst zu erzeugen, da das Programm `autoheader` es automatisch aus den Test in dem `configure.in` File ableiten kann, s.u.
Das Configurations-Script selbst ist ein normales Bourne-Shell Script, welches man allerdings nicht selbst schreiben muß, sondern das man aus einer Beschreibung der durchzuführenden Test mittels des `autoconf` Programms erzeugt. Diese Beschreibung steht i.A. in einem File namens `configure.in`. Dieses File enthält neben normalen Shell-Befehlen sogenannte *Tests*. `autoconf` ersetzt diese Tests durch Shell-Befehle, welche die von dem Test gewünschte Prüfung durchführen.
Beispiel für ein `configure.in`:

```
dnl Process this file with autoconf to produce a
    configure script.
AC_INIT(indent.c)
AC_PROG_CC
AC_PROG_CPP
AC_PROG_INSTALL
AC_AIX
AC_ISC_POSIX
AC_REPLACE_FUNCS(memcpy)
AC_UNISTD_H
AC_DIR_HEADER
if test -z "$dirheader"; then
  AC_DEFINE(NODIR)
fi
AC_XENIX_DIR
AC_OUTPUT(Makefile, src/Makefile)
```

Hier dient die Zeile `AC_INIT(indent.c)` zur Initialisierung, Kommentare werden durch `dnl` eingeleitet. Beispielhaft prÜft der Test `AC_PROG_INSTALL` auf ein BSD-kompatibles `install` Programm (s.o.) und ersetzt den Platzhalter `@INSTALL@` durch dieses Programm.
Der Test `AC_UNISTD_H` prüft, ob das Header-File `unistd.h` vorhanden ist und setzt dann das Symbol `HAVE_UNISTD_H` in `config.h`.
Die letzte Zeile startet die Ersetzung von den Schablonen in die Files `Makefile` und `src/Makefile`.
`autoconf` bietet unzählige Tests:

- Test auf Vorhandensein & Ort verschiedenster Utilities
- Test auf Libraries

- Test auf bestimmte Funktionen in Libs
- Systemaufruftest
- Größe von Datentypen
- Header-Files
- typedefs, Definitionen in Headern
- ...

Für eine ausführliche Beschreibung der Tests sei auf die Dokumentation [ME98] des `autoconf` Packetes verwiesen.

5.9 Beispiele

Bei der Generierung von Dokumenten mit LaTeX (siehe Kapitel 6) bindet man oft Abbildungen ein, die mittels `xfig` erzeugt wurden. Diese Abbildungen werden dann mit dem `fig2dev` Programm vom xfig-Format zu Encapsulated-Postscript konvertiert. Dabei benutzt man einen Aufruf der Form

`fig2dev -L ps -p dummy -m` *`Mag XFIG-File >EPS-File`*

Hierbei steht *Mag* für den Skalierungsfaktor der Abbildung. Dieser kann für verschiedene Abbildungen variieren, da alle Abbildungen in `xfig` meistens in der gleichen Größe entworfen werden, im Text aber verschieden groß erscheinen sollen.
Folgende Regeln und Variablendefinitionen erlauben es nun, innerhalb eines `Makefiles` eine Standardskalierung anzugeben, die aber für bestimmte Abbildungen überschrieben werden kann.

```
FIGSRCS=abb1.fig abb2.fig
FIGEPS=$(patsubst %.fig,%.eps,$(FIGSRCS))

# Default Skalierung
MAG_DEFAULT=0.5

# Spezielle Sklalierung fuer  abb1
MAG_abb1=0.8

# fig -> eps
%.eps: %.fig
	fig2dev -L ps -p dummy \
	  -m $(word 1,$(MAG_$*) $(MAG_DEFAULT)) $< >$@
```

Alle XFIG-Files werden in der Variablen `$(FIGSRCS)` angegeben, daraus wird mittels einfacher Ersetzung die Liste aller EPS-Files gemacht (die dann in den Quellen des Dokumentzieles als Abhängigkeiten angegeben werden können). Fur das File `abb1.eps` wird eine abweichende Vergröserung angegeben. In der impliziten Regel für die Konvertierung von XFIG-Files zu EPS-Files wird nun ein Trick angewandt: Mittels der `word` Funktion wird das erste Wort der Liste `$(MAG_$*) $(MAG_DEFAULT)` genommen. Der erste Teil enthält dabei den Namen des Stammes (z.B. `abb1`) der impliziten Regel. D.h. das erste Wort expandiert zum Inhalt der Variablen `$(MAG_abb1)`, wenn man `abb1.eps` neu bauen soll. Diese Variable ist aber nur dann definiert, wenn man eine spezielle Skalierung für die Abbildung definiert hat, expandiert im Standardfall also zum leeren Text. Damit ist das erste Wort der Liste die Standardskalierung, wenn man keine spezielle Skalierung für die fragliche Abbildung definiert hat. Ist eine solche definiert, ist sie das erste Wort der Liste und wird von der `word` Funktion ausgewählt.

5.10 Falltüren

Eine beliebte Falltür im Zusammenhang mit Variablen ist, daß Variablen lazy ausgewertet werden, wenn sie mittels des = Operators zugewiesen wurden. Das Anhängen von Werten an Variablen funktioniert also nicht durch `CFLAGS=$(CFLAGS) -Wall`, sondern man muß den speziellen += Operator verwenden: `CFLAGS+=-Wall`.
Ebenso vergißt man leicht, daß Variablen, die auf der Kommandozeile gesetzt wurden, nicht mehr überschrieben werden können (außer, man benutzt die `override` Direktive).
Auch werden Variablen nicht an Sub-`make` Prozesse weitergereicht, es sei denn man exportiert sie explizit.

5.11 Literatur

Die ultimative Referenz für GNU `make` ist natürlich die Dokumentation [SM98], welche mit dem Programm als Info-Seiten installiert sein sollte (`info make`). Die meisten Bücher über `make`-ähnliche Programme gehen nicht auf GNU `make` ein, können aber lesenswert sein, um die allgemeinen Prinzipien zu verstehen. Ich habe leider noch keins gefunden, das empfehlenswert wäre.
Auskunft über das `makedepend` Programm gibt dessen Manual-Seite. Auf jedem System, auf dem X-Windows installiert ist, sollte es auch eine Version des `makedepend` Programmes geben, da es mit der X-Windows Distribution instal-

liert wird. Die Manual-Seite sollte sich also zusammen mit denen des X-Windows System finden (meistens `/usr/X11R6/share/man`).
Eine Referenz über die Prinzipien und verfügbaren Tests des `autoconf` Packetes bietet dessen Dokumentation [ME98], die auch als Info-Seiten installiert sein sollten.

Kapitel 6

LaTeX

6.1 Einführung

6.2 Dokumentgliederungen

6.3 Texthervorhebungen

6.4 Seitenumbruch

6.5 Umgebungen

6.6 Listen, Tabellen

6.7 Fußnoten, Verweise

6.8 Mathematische Formeln

6.9 Abbildungen

6.10 Eigene Macros, Umgebungen

6.11 Fehlermeldungen

6.12 Literaturangaben: `bibtex`

6.13 Indices: `makeindex`

6.14 Abbildungen: `xfig`

6.15 Abbildungen: `gnuplot`

6.16 Beispiele

6.17 Falltüren

6.18 Literatur

[Kop96, Kop97a, Kop97b]

Kapitel 7

Texinfo

7.1 Einführung

7.2 Knoten

7.3 Menüs

7.4 Kapitel, Abschnitte, etc.

7.5 Querverweise

7.6 Texthervorhebungen

7.7 Listen, Tabellen, etc.

7.8 Indices

7.9 Seitenumbruch

7.10 Fußnoten

7.11 Conditionale

7.12 Macros

7.13 Ausdruck

7.14 Beispiele

7.15 Literatur

[Cha98]

Kapitel 8

Informationsquellen

Literaturverzeichnis

[Cha98] Robert J. Chassel. *texinfo manual*. Free Software Foundation, 3.12 Edition, Februar 1998.

[Dou90] Dale Dougherty. *sed & awk*. O'Reilly & Associates, 1990.

[Kop96] Helmut Kopka. *LaTeX: Einführung*. Addison-Wesley, 1996. ISBN: 3-8273-1025-3.

[Kop97a] Helmut Kopka. *LaTeX: Ergänzungen*. Addison-Wesley, 1997. ISBN: 3-8273-1229-9.

[Kop97b] Helmut Kopka. *LaTeX: Erweiterungen*. Addison-Wesley, 1997. ISBN: 3-89319-666-8.

[ME98] David MacKenzie und Ben Elliston. *Autoconf. Creating Automatic Configuration Scripts*. FSF: Free Software Foundation, Inc., 2.13 Edition, Dezember 1998.

[SM98] Richard M. Stallman und Roland McGrath. *GNU make manual*. Free Software Foundation, 3.77 Edition, Mai 1998.

[WS93] Larry Wall und Randal L. Schwartz. *perl*. O'Reilly & Associates, 1993.