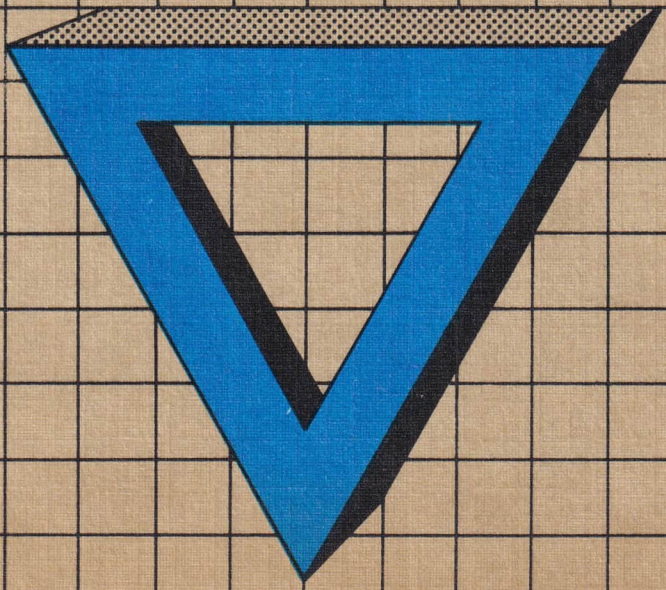
Technische Informatik

Polze

# UNIX-Werkzeuge zur Programmentwicklung



VEB Verlag Technik Berlin

Polze

# UNIX-Werkzeuge zur Programmentwicklung

# TECHNISCHE INFORMATIK

Herausgegeben von
Dr.-Ing. Ludwig Claßen
Dr.-Ing. Dieter Nedo
Dr. rer. nat. Gerhard Paulin

# UNIX-Werkzeuge zur Programmentwicklung

Prof. Dr. sc. nat. Christoph Polze

VEB VERLAG TECHNIK BERLIN

Polze, Christoph
UNIX-Werkzeuge zur Programmentwicklung /
Christoph Polze. - 1. Aufl. - Berlin : Verl.
Technik, 1989. - 239 S.: 16 Bilder, 2 Taf.
ISBN 3-341-00865-9

ISBN 3-341-00865-9
ISSN 0863-0860 Technische Informatik

1. Auflage

VT 201 · 3/6008-1 (250)
Printed in the German Democratic Republic
Druck und buchbinderische Verarbeitung:
(140) Druckerei Neues Deutschland, Berlin
Lektor: Jürgen Reichenbach
Einbandgestaltung: Gabriele Schwesinger
LSV 3054
Bestellnummer: 554 278 6
02400

*Meinen Eltern gewidmet.*

**Vorwort**

UNIX fasziniert als Betriebssystem und technologische Umgebung für eine hocheffektive Programmentwicklung. Es bietet die gute Chance, gestiegene Maßstäbe an Portabilität und Kompatibilität zu befriedigen. UNIX ist ein Mehrfachzugriffsystem, das mit einheitlicher Nutzeroberfläche auf allen derzeit verbreiteten Rechnerarchitekturen betrieben werden kann. Das ist Angebot und Herausforderung zugleich. Seit seiner Entstehung vor etwa 15 Jahren, Erstveröffentlichung Ritchie und Thompson (1974), steht UNIX rechnergeprägten oder firmentypischen »Primärbetriebssystemen« gegenüber. Was wiegt jedoch längerfristig schwerer, die einheitliche und systemunabhängige Programmentwicklung oder eine womöglich effektivere architekturspezifische Verarbeitung? Diese Frage kann auch durch ein weiteres Buch zu UNIX nicht beantwortet werden. Aber ein vergrößerter Kreis von UNIX-Vertrauten kann sich zu dieser Frage mit mehr Sachkenntnis positionieren. Es ist das Anliegen auch dieses UNIX-Buches, Informationen über UNIX und Erfahrungen im Umgang mit UNIX-Werkzeugen zu vermitteln.

Postiert man die Dokumentation kommerzieller UNIX-Systeme Schrift für Schrift nebeneinander, so mißt das Bücherbord etliche Dezimeter. Eine Auseinandersetzung mit UNIX in der Reihe »Technische Informatik« zwingt daher zu einer Auswahl, ohne sich in eine oberflächliche Darstellung drängen zu lassen. Ich habe drei Schwerpunkte gesetzt. Am Anfang steht die Beschäftigung mit der Dateiverwaltung. Die Diskussion der beiden Kommandointerpreter Bourne-Shell und C-Shell schließt sich an. Der dritte Akzent wurde der Werkzeugfamilie make, yacc und lex zugedacht. Jedesmal wurde die Absicht verfolgt, eine einführende informative Darlegung des Faktenmaterials in einige detailliertere und vielleicht auch gewisse Ansprüche erhebende Anwendungen münden zu lassen, soweit dies die Begrenzungen des Umfangs zuließen. Außerhalb der Betrachtungen mußten die gewiß sehr wichtigen Programmierungswerkzeuge lint, adb und sdb sowie m4 bleiben. Ebensowenig konnte auf das Quelltextverwaltungssystem SCCS, das Textverarbeitungsprogramm awk, die Terminalansteuerungen curses und auf die Koppelpakete micnet und uucp eingegangen werden.

Die dominierende Programmiersprache ist C. C-Texte werden als Sprachkonstrukte nicht erklärt, sondern einfach verwendet. Der große Kreis von Lesern, der Sprachen der Familie ALGOL, PASCAL oder Modula-2 kennt, wird jedoch kaum Schwierigkeiten haben, die C-Programme zu lesen. Wer alle Details der C-Programmierung nachschlagen will, sei auf das Buch von Clauß und Fischer (1988) verwiesen, das ebenfalls in der Reihe »Technische Informatik« erschienen ist. Gewiß sind diejenigen Leser im Vorteil, die parallel zur Lektüre dieses Buches Zugang zu einem UNIX-System haben und die darin gezeigten Abläufe ausprobieren, die Programmfragmente modifizieren oder analoge Beispiele konstruieren können.

Das Buchmanuskript wurde mit UNIX hergestellt. Größtenteils sind die abgedruckten Programme unmittelbar aus der Testumgebung heraus in das Manuskript kopiert worden. Dadurch sind Fehlerfallen gewiß entschärft worden, aber nicht ausgeschlossen. Ich bedanke mich daher im voraus für alle Hinweise und Verbesserungsvorschläge, die Sie, verehrter Leser, freundlicherweise an mich richten werden. Die meisten Programme wurden auf einem SCO-XENIX System V oder auf WEGA getestet. Dabei konnte bereits beobachtet werden, daß nicht alle Systemprogramme in beiden Systemen verfügbar waren oder gleiches Verhalten zeigten. Es bedarf also genauerer Betrachtungen, inwieweit sich angebotene Programmiervarianten auf speziellen UNIX-Systemen verwirklichen lassen. Die Standardisierung von UNIX ist im Gange. Das stabile Ende dieses Prozesses ist jedoch noch nicht erreicht.

Das Buch will und kann die Dokumentation der verwendeten UNIX-Kommandos nicht ersetzen. Erst das Nachsuchen in den Manualtexten des benutzten UNIX-Systems wird hoffentlich letzte Klarheit bringen, sei es in der angebotenen On-line-Form (WEGA) oder als gedruckte Dokumentation. Eine Zusammenstellung von Beschreibungen wichtiger Kommandos befindet sich in dem verbreiteten und auch in der Reihe »Technische Informatik« erschienenen Buch von Claßen und Oefler (1987).

Ein herzliches Dankeschön will ich dem Herausgeber der Reihe »Technische Informatik« und meinem verehrten Kollegen, Herrn Dr. G. Paulin, entrichten, der die Niederschrift dieses Buches initiiert und durch viele wertvolle Hinweise bereichert hat. Ebenso gilt mein Dank dem verantwortlichen Lektor des VEB Verlag Technik, Herrn J. Reichenbach, für die gute Zusammenarbeit sowie allen Fachkollegen und vielen Studenten des Bereiches Informationsverarbeitung der Sektion Mathematik der Humboldt-Universität zu Berlin, die durch ständige Diskussionen die Arbeit an diesem Manuskript begleitet haben.

Berlin Christoph Polze

Anschrift:
Prof.Dr. Christoph Polze
Humboldt-Universität zu Berlin
Sektion Mathematik
Postfach 1297
Berlin
1086

**Inhaltsverzeichnis**

# 1. Begegnung mit UNIX

UNIX ist Betriebssystem und technologische Umgebung für die Programmentwicklung zugleich. Als Betriebssystem stellt es jedem Anwender

- eine Kommandosprache,

- ein Dateiverwaltungssystem und

- einen oder mehrere Prozesse

zur Verfügung.

Die Kommandosprache dient als Verständigungsmittel zwischen dem Nutzer und dem Rechnersystem. Sie bildet die Schnittstelle zum Benutzer des UNIX. Die Basiselemente für die Arbeit mit UNIX sind die Dateien. Jeder Nutzer kann eine Hierarchie von Dateien aufbauen und verwalten. Dateien können Daten oder Programme sowie Dateiverzeichnisse beinhalten oder als sogenannte »special files« periphere Geräte steuern. Ein Prozeß ist ein Abbild des UNIX-Rechners für den Nutzer. Er umfaßt die für die Programmabarbeitung erforderlichen Rechnerressourcen, wie Speicher und CPU-Zeit. Ein Prozeß, der mit der Eröffnung einer Terminalsitzung anläuft, wird zur Wurzel einer zeitlich sich ändernden Hierarchie von parallelen Prozessen, die sämtlich für die nutzerbezogene Programmabarbeitung benötigt werden. Für seine eigene interne Arbeit als Mehrfachzugriffsystem und Timesharing-Betriebssystem verfügt UNIX über ein mächtiges System von Dateien und Prozessen.

Als technologische Entwicklungsumgebung enthält UNIX eine Vielzahl von Werkzeugen, die Nutzer für ihre eigene Programmentwicklung einsetzen können. Dazu gehören Editoren, Compiler, Programmgeneratoren und Dienstprogramme. Abhängig vom Stand seiner eigenen Einarbeitung kann der Nutzer auch von den Dateien des Betriebssystems für die Entwicklung seiner Programme profitablen Gebrauch machen.

UNIX ist ein offenes System. Das bedeutet, der Nutzer kann in einer sehr flexiblen Weise seine Programmentwicklungsumgebung subjektiv an seine Vorstellungen und Wünsche anpassen. Dazu kann er globale Datenstrukturen schaffen, die bei der Abarbeitung seiner Kommandos wirksam werden. Er kann eigene Programme in seinem Dateisystem als Werkzeuge vorhalten. Er kann Kommandofolgen, die in seiner Arbeit mit dem System häufig in ähnlicher Form wiederkehren, zu einer parametrisierten Datei zusammenfassen. Die Arbeit mit UNIX gewinnt daher sehr schnell individuelle Züge, so daß das Studium von UNIX-Texten, von Beispielen für die UNIX-Arbeit immer von erheblichen Kontextinformationen beeinflußt wird.

UNIX und die Programmiersprache C sind auf das engste miteinander verbunden. UNIX wurde in dieser Sprache implementiert, und die Sprache C wurde vor allem zu dem Zweck definiert, daß dieses Betriebssystem mit allen seinen Komponenten in C programmiert werden konnte. Daher nimmt es nicht wunder, daß die Programmierung von C-Programmen in UNIX maximal unterstützt wird. UNIX bietet seine Dienste als Entwicklungsumgebung jedoch auch für andere Programmiersprachen, Programmiersysteme oder Fachsprachen an.

Die Bindung von UNIX an die Programmiersprache C begründet die hervorragende Chance, UNIX als gleichartige Benutzerschnittstelle auf unterschiedlichen Rechnerarchitekturen, sowohl auf Groß- oder Universalrechnern als auch auf Arbeitsplatzrechnern oder Mikrorechnern

ausschöpfen zu können. Weltweit ist das Bestreben zu beobachten, die inzwischen verschieden ausgeprägten UNIX-Systeme zu standardisieren. Ein fortgeschrittenes Standardisierungsstadium ist mit dem UNIX-System V erreicht worden. Dessenungeachtet muß man damit fertig werden, daß derzeit noch durchaus unterschiedliche UNIX-Varianten anzutreffen sind. Bei der Auswahl der Beispiele wurde versucht, möglichst starke Bindungen an spezielle Systeme zu meiden. Ganz gewiß bleiben aber neben der subjektiven Prägung der UNIX-Entwicklungsumgebung auch Diskrepanzen, die durch die objektiv gegebenen Unterschiede zwischen den jeweiligen Betiebssystemversionen verursacht werden.

Als einleitendes Beispiel für eine Programmentwicklung wählen wir folgendes Programm. Gegeben ist eine Gesamtheit von Objekten. Jedes Objekt besitzt eine ganze Zahl als Attribut. Die Gesamtheit ist in Gruppen untergliedert. Die Attributwerte sollen gruppenweise akkumuliert werden. Schließlich wird die Gesamtsumme über alle Gruppen benötigt.

Diese Gesamtheit von Objekten wird durch die Datei accu.inp beschrieben. Sie enthält Schlüssel mit den Werten 0, 1 oder 2 und die ganzzahligen Attribute. Ein Schlüssel 1 eröffnet eine neue Gruppe. Jedem Schlüssel 0 folgt eine ganze Zahl. Schlüssel 2 beendet die Gruppenfolge.

Hier ist eine Datei accu.inp mit Testeingabedaten:

```
1
0  10
0  20
0  30
0  40
1
0  11
0  41
1
0   9
0  39
2
```

Diese schlichte Aufgabenstellung wird durch das PASCAL-Programm accu gelöst.

```
program accu(input,output);
var key,val:integer;
    sum,totsum:integer;
begin
  writeln('Bericht');
  totsum:=0;
  read(key);
  while key=1 do
  begin
    sum:=0;
    read(key);
    while key=0 do
    begin
      read(val);
      writeln(val);
      sum:=sum+val;
      read(key);
    end;
```

```
        totsum:=totsum+sum;
        writeln('Gruppensumme',sum);
      end;
      writeln('Gesamtsumme ',totsum);
    end.
```

Das Programm accu sei in der Datei accu.pas festgehalten.

Nun soll das Programm accu übersetzt und danach mit den Testeingabedaten abgearbeitet werden. Dazu brauchen wir erstmals die Dienste unseres UNIX-Systems.

```
1% make accu
2% accu <accu.inp
```

Dies sind zwei Kommandos. Das erste Kommando bewirkt die Umformung des PASCAL-Textes in das abarbeitbare Programm accu, und das zweite Kommando startet dieses Programm mit den Eingabedaten aus der Datei accu.inp. Die Ergebnisse werden auf dem Terminal angezeigt. Mit dem Kommando

```
3% accu <accu.inp >accu.res
```

werden die Resultate der Arbeit von accu in die Datei accu.res geschrieben. Sie hat dann den folgenden Inhalt:

```
Bericht
        10
        20
        30
        40
Gruppensumme       100
        11
        41
Gruppensumme        52
         9
        39
Gruppensumme        48
Gesamtsumme        200
```

UNIX bietet dem Nutzer Interpreter für mehrere Kommandosprachen zur Auswahl. Verbreitete Kommandointerpreter sind die Bourne-Shell die C-Shell sowie die in ihrem Leistungsumfang stark reduzierte visual shell. Die letztere ist für Benutzer gedacht, die nur einfache Arbeiten mit ihren Dateien ausführen wollen. In diesem Buch wird vor allem die C-Shell benutzt.

Kommandos bestehen aus dem Namen einer abarbeitbaren Datei und einer möglicherweise auch leeren Liste von Operanden. make ist ein wichtiges Dienstprogramm aus dem Angebot der UNIX-Werkzeuge. Das Programm accu in den Kommandos 2% und 3% wurde durch die Arbeit des ersten Kommandos erzeugt. Operanden sind Dateien mit dem PASCAL-Quelltext, mit den Eingabedaten und mit den Resultaten.

Die drei Kommandos für das kleine Beispiel machen schon ausgiebig vom Kontext der Terminalarbeit Gebrauch. So ist der Name des für diese Kommandoausführung erforderlichen Dateiverzeichnisses als Wert einer globalen Variablen vorhanden, auf den die C-Shell zugreifen muß, um die angesprochenen Dateien zu finden. Auch die Kommandonumerierung und das Prozentzeichen als Aufforderungsprompt werden durch globale

Wertsetzungen geregelt. Weitere globale Variable gehören zur Umgebung der Arbeit des C-Shell-Kommandointerpreters.

Ganz wesentlich ist die Arbeit des Dienstprogramms make von Kontextinformationen abhängig. Soll make eine PASCAL-Übersetzung auslösen, so muß eine Datei mit dem Namen makefile verfügbar sein. Um sich einen Überblick über die unmittelbar vorhandenen Dateien zu verschaffen, sollte man sich das aktuelle Dateiverzeichnis betrachten. Mit dem Kommando

```
4% ls -al
```

kann man sich den Inhalt des Dateiverzeichnisses ansehen, das die Terminalarbeit als Kontext umgibt. Sein Ergebnis lautet:

```
total 50
drwxr-xr-x   2 polze     HUB       112 Jan 26 11:01 .
drwxr-xr-x   3 polze     HUB       384 Jan 26 11:07 ..
-rwxr-xr-x   1 polze     HUB     17588 Jan 25 09:31 accu
-rw-r--r--   1 polze     HUB        56 Jan 15 08:26 accu.inp
-rw-r--r--   1 polze     HUB       395 Jan 13 08:53 accu.pas
-rw-r--r--   1 polze     HUB       188 Jan 26 11:03 accu.res
-rw-r--r--   1 polze     HUB        48 Jan 11 10:40 makefile
```

Eine Datei mit dem Namen makefile ist also tatsächlich vorhanden. Wir schauen sie uns später an.

Das Kommando 4% benutzt eine Option als Operand. Der Buchstabe l veranlaßt die lange Form der Ausgabeliste, und der Buchstabe a bewirkt, daß alle Dateien des Verzeichnisses ausgelistet werden, auch solche, deren Namen mit einem Punkt beginnt.

Manche UNIX-Systeme, beispielsweise das System WEGA, halten in ihrem Speicher Kurzbeschreibungen aller Systemkommandos ständig verfügbar. Wenn Kommandos viele Optionen haben, ist es eine gute Hilfe, während der Arbeit am Terminal die entsprechende Kurzbeschreibung eines Kommandos besichtigen zu können.

Mit dem Kommando

```
5% man ls
```

werden die Manualseiten für das Kommando ls ausgegeben. In den Programmierhandbüchern der UNIX-Systeme sind für alle verfügbaren Kommandos derartige Kurzbeschreibungen gedruckt; s. WEGA(1987). In diesem Buch werden nur die benutzten oder gezielt ausgewählte Optionen von Kommandos erklärt. Der Leser ist gut beraten, wenn er bei jedem verwendeten Kommando das Handbuch seines Systems studiert, um einen Überblick über alle Möglichkeiten des Kommandos in seinem System zu bekommen.

Anhang 1 enthält als Mustertext eine Manualseite für das Kommando script. script ermöglicht es, Protokolle über alle am Terminal ein- oder ausgegebenen Informationen als Datei zu sammeln. Die meisten der in diesem Buch abgedruckten Kommandofolgen sind aus Terminalprotokollen hervorgegangen, die mit script hergestellt wurden.

Hier ist der Anfang des Manualtextes von ls:

---

LS(1) WEGA LS(1)

**NAME**

ls - Ausgabe des Inhalts eines Directory

**SYNTAX**

**ls** [ **-aAcCdDfFgilmnqrRstux1** ] *file*

**BESCHREIBUNG**

Für jedes Argument *file*, das eine Datei darstellt, listet **ls** den Dateinamen und alle weiteren angeforderten Informationen aus. Ist das Argument *file* ein Directory, gibt **ls** den Inhalt dieses Directory zusammen mit den angeforderten Zusatzinformationen aus. Die Ausgabe wird standardmäßig alphabetisch sortiert. Wird im Aufruf kein Argument *file* angegeben, wird der Inhalt des aktuellen Arbeitsdirectory ausgegeben. Werden mehrere Argumente *file* angegeben, so werden diese Argumente zunächst sortiert. Dabei werden erst Datei- und danach Directoryargumente behandelt.

Die durch die Option **-l** erzeugte Mode-Information enthält zehn Zeichen. Das erste hat dabei folgende Bedeutung:

d Directory
b Block special file
c Character special file
p benannte Pipe
- einfache Datei

Die nächsten neun Zeichen werden in drei Gruppen zu jeweils drei Bit interpretiert - sie geben die Zugriffserlaubnisse an (Permission-Bits). Die ersten 3 Bit beziehen sich auf die Zugriffserlaubnis des Eigentümers (owner) der Datei oder des Directory; die folgenden 3 Bit beziehen sich auf die Zugriffserlaubnis der Nutzer in der gleichen Nutzergruppe, und die letzten 3 Bit beziehen sich auf die Zugriffserlaubnis aller anderen Nutzer (others). Innerhalb jeder Gruppe geben die drei Zeichen die Erlaubnis zum Lesen, Schreiben bzw. zum Ausführen der jeweiligen Datei an. Bei einem Directory wird die Ausführungserlaubnis als Erlaubnis zum Durchmustern des Directory nach einer angegebenen Datei interpretiert. Dabei wird folgende Notation verwendet:

r lesbar
w schreibbar
x ausführbar
- keine Erlaubnis

---

Wenden wir uns nun der Datei makefile zu. Wir können sie mit dem Kommando

```
6% cat makefile
```

an das Terminal senden. Das Kommando 6% bringt folgenden Text auf den Bildschirm:

```
.SUFFIXES: .pas
.pas:
        pascal $@
        -rm $*.c $*.o
```

In der UNIX-Philosophie werden Suffixe benutzt, um Dateien nach ihrem Inhalt zu klassifizieren. Das Suffix .c kennzeichnet eine Datei als Text in der Programmiersprache C, das Suffix .o besagt, die Datei ist ein Objektmodul, der durch einen Lader des UNIX-Betriebssystems in ein abarbeitbares Programm umgeformt oder einbezogen werden kann. Die beiden Suffixe .c und .o sind im UNIX-Standard fixiert. Das Suffix .pas gehört nicht dazu. Durch die Zeile

```
.SUFFIXES: .pas
```

wird es zu den standardmäßig gültigen Suffixen hinzugenommen. Nun kann das Programm make, das im Kommando 1% aufgerufen wurde, danach suchen, ob es im aktuellen Dateiverzeichnis zu dem Operanden accu eine Datei mit einem gültigen Suffix gibt. Es gibt die Datei accu.pas. Das Suffix .pas ist gültig. Die drei restlichen Zeilen von makefile beschreiben, wie aus einer Datei name.pas eine Datei name hergestellt werden kann. In unserem Fall ist name = accu. Die Zeile

```
pascal $@
```

ruft das Übersetzungssystem für PASCAL-Texte auf. Mit $@ wird abkürzend das Ziel des Kommandoarbeit bezeichnet. Das ist in diesem Fall das Argument des Aufrufs von make, also die Datei accu.

Das hier verwendete Übersetzungssystem, Bothe u.a. (1987), arbeitet in mehreren Schritten. Zuerst wird aus dem Text name.pas ein C-Text hergestellt. Er wird anschließend übersetzt. Dabei enstehen neben dem gewünschten Resultat accu die Zwischentexte accu.c und acco.o. Diese sind beide überflüssig, wenn das abarbeitbare Programm accu fertig ist. Die Zeile

```
-rm-$*.c $*.o
```

beseitigt diese Zwischentexte. Mit $* wird von dem auslösenden Namen accu.pas der suffixfreie Teil genommen. Dieser ist in unserem Fall accu. Das ergibt: $*.c ist accu.c und $*.o ist accu.o. Das Minuszeichen an dieser Zeile ignoriert etwaige Fehlermeldungen des remove-Kommandos rm.

Insgesamt wird nebenbei deutlich, daß mit dieser Interpretation von makefile PASCAL-Texte mit beliebigen Dateinamen verarbeitet werden können. Das ist die zentrale Leistung des technologischen Werkzeugs make, das gleich in dieser ersten Begegnung mit UNIX vorgestellt werden sollte. Natürlich macht die knappe Formulierung im Kommando 1% sehr intensiv vom Kontext der Benutzerarbeit Gebrauch und unterstreicht dessen Bedeutung für das Verständis von UNIX-Abläufen.

UNIX erledigt alle seine Aktivitäten in der Form von Prozessen. Will man während einer Sitzung einen Einblick in die aktuelle Prozeßstruktur des Systems erhalten, so kann man das Kommando

```
7% ps -ef
```

aufrufen. Es liefert eine Schnappschußaufnahme der vorhandenen Prozesse zum Zeitpunkt seiner Ausführung. Das Ausgabelisting stammt aus einem XENIX286-System.

```
    UID   PID  PPID  C    STIME TTY  TIME COMMAND

   root     0     0  0        ?   ? 326:38 swapper
   root     1     0  0        ?   ?  0:06 /etc/init
  polze   672     1  1 13:55:55  01  0:15 -csh
   root    22     1  0 08:55:28   ?  0:00
                      logger /dev/error /usr/adm/messages
   root    21     1  0 08:55:28   ?  0:27 update
   root   983     1  0 14:59:26  02  0:02 - tty02 m
     lp    26     1  0 08:55:30   ?  0:01 /usr/lib/lpsched
   root    30     1  0 08:55:32   ?  0:00 cron
  guest   985     1  0 14:59:37  03  0:04 -sh
  polze  1133     1  0 15:13:05  04  0:08 -csh
 mblume   767     1  0 14:34:40  1a  0:11 -csh
 apolze  1134     1  0 15:13:11  2a  0:07 -csh
 mblume  1132   767 41 15:12:46  1a  0:42 yaxlisp
 apolze  1139  1134  0 15:14:16  2a  0:02 vi skz.c
  polze  1145   672 35 15:15:16  01  0:01 ps -ef
```

Die Abkürzungen in der Kopfzeile des Listings haben folgende Bedeutungen:

| | |
|---|---|
| UID | User-Identifikation |
| PID | Prozeß-Identifikation |
| PPID | Identifikation des Parent-Prozesses |
| C | Prozessorbenutzung für Scheduling-Abläufe |
| STIME | Startzeit des Prozesses |
| TTY | Terminalkennzeichnung |
| TIME | CPU-Zeit [min:s] |
| COMMAND | Kommando, das während des ps-Schnappschusses ausgeführt wurde. |

Die Prozesse 672, 767, 983, 985, 1133 und 1134 bilden jeweils Wurzelknoten von hierarchischen Systemen von Nutzerprozessen. Davon haben nur die Prozesse 672, 767 und 1134 Nachfolgeprozesse, jeweils einen Nachfolger. Prozeß 767 und Nachfolger 1132 gehören zum Terminal 1a, und Prozeß 1134 mit seinem Nachfolger 1139 gehört zum Terminal 2a. Der Buchstabe a kennzeichnet die Terminals als nah aufgestellte.

Bei dieser Betriebsweise von XENIX286 kann die Konsole für vier Nutzerprozesse eingesetzt werden (Auswahl mit der Tastenkombination Alt-F1...Alt-F4 oder Ctrl-Alt-Fi). Diese Einrichtung wird als multi-screening bezeichnet.

Bild 1.1 zeigt die mit dem Kommando 7% erzeugte Situation als Graph mit den Prozeß-Identifikatoren als Knoten und der Relation (PPID,PID) als gerichtete Kanten.

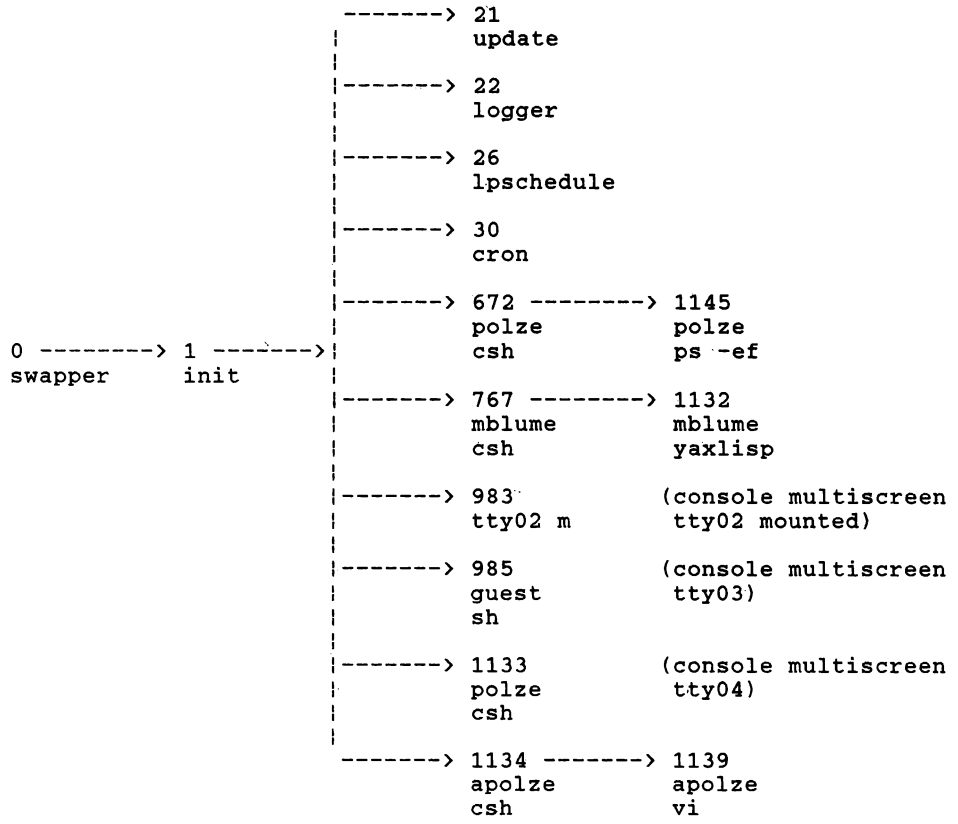



Bild 1.1. ps-Graph

Es ist für das Verständnis der UNIX-Arbeit sehr vorteilhaft, von Anfang an sich die folgenden drei Aspekte immer wieder zu verdeutlichen. Einige davon wurden bei den kurzen Kommandos 1% bis 7% schon angesprochen.

- Jede Aktivität des UNIX verläuft mit einem vielgestaltigen Dateisystem im Hintergrund.

- Die Aktionen des UNIX werden von einer Reihe von voreingestellten Variablen beeinflußt.

- Die UNIX-Arbeit ist unterbrechbar.

Diese Vorstellung soll durch das Bild 1.2 veranschaulicht werden.

Die Rolle des zugrunde liegenden Dateisystems wurde an den einführenden Kommandobeispielen bereits sichtbar. Programme und Daten traten als Dateien auf. Während der Arbeit mit UNIX wurden vorhandene Dateien benutzt. Es wurden neue Dateien erzeugt und soeben erzeugte Dateien anschließend als Programme aufgerufen. Andere Dateien, hier in der Gestalt der Datei makefile, hatten einen erheblichen Einfluß auf die UNIX-Arbeit. Für die Interpretation des Kommandos make waren Informationen aus der Datei makefile notwendig.

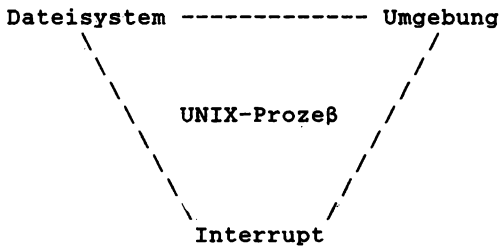



Bild 1.2. Die Umwelt von UNIX-Prozessen

Jede UNIX-Arbeit ist an eine Reihe von voreingestellten Variablen gebunden. Einerseits handelt es sich um die Variablen der sogenannten Umgebung. Andererseits verwaltet jeder Kommandointerpreter einen Satz von eigenen Variablen, die bei seinem Arbeitsbeginn automatisch gesetzt werden oder die der Benutzer im Laufe seiner Terminalarbeit mit Werten belegen kann. Die Umgebungsvariablen werden den zu startenden Kommandoprogrammen verfügbar gemacht. Sie stellen die Arbeitsumgebung der Programme dar. Die anderen Variablen dienen ausschließlich der Steuerung der Kommandointerpreter selbst. Eine Reihe von Möglichkeiten gibt es, die Umgebung zu modifizieren und sie den gebotenen Bedingungen anzupassen. Bei der genauen Beschreibung der Kommandointerpreter wird davon noch zu berichten sein.

Alle diese Variablen haben Zeichenketten als Werte. Damit wird ein erhebliches Maß an Maschinenunabhängigkeit gewährleistet. Die Standardisierungsvorschläge von X/OPEN (1987) behandeln auch die Gestaltung der Umgebung. Danach kann ein interessierter UNIX-Nutzer davon ausgehen, daß er in jedem beliebigen UNIX-System, das dem X/OPEN-Standard genügt, folgende Umgebungsvariablen vorfindet:

| | |
|---|---|
| HOME | Stellt den Pfad durch das Dateisystem des UNIX zur Verfügung, der in das nutzereigene Dateisystem führt. Diese Variable wird gesetzt, wenn sich der Nutzer am Terminal anmeldet. |
| LOGNAME | Gibt den Namen des Nutzers als Zeichenkette. |
| PATH | Ist eine durch Zeichen : getrennte Liste von Pfadnamen. Sie bestimmt die Suchordnung, die bei der Lokalisierung von Dateien eingehalten wird. |
| TERM | Bezeichnet das Terminal, für das die Datenausgabe aufbereitet wird. |
| TZ | Charakterisiert die Zeitzone, in der der Nutzer arbeitet. |

Oftmals gibt es in der Umgebung eines UNIX-Systems weitere Variable. Die Formulierung des X/OPEN-Standards läßt dies durchaus zu. Nur sollte man diese Variablen in solchen Projekten nicht verwenden, die von vornherein auf eine Verbreitung in andere Systeme hinzielen.

| | |
|---|---|
| SHELL | Kennzeichnet den angebotenen Kommandointerpreter. /bin/csh ist der C-Shell-Kommandointerpreter. /bin/sh ist die Bourne-Shell. Sie allein wird als Kommandointerpreter des X/OPEN-Standards angesehen. |

| | |
|---|---|
| MAIL | Ist die Datei, in der Nachrichten an den Benutzer abgelegt werden. |
| TERMCAP | Beschreibt das Terminal des Benutzers. Die Steuerzeichen sind als Liste zusammengefaßt. |

Einen weiteren sehr wichtigen Gesichtspunkt der UNIX-Arbeit bilden die Vorkehrungen für Interrupts. Äußere Einflüsse, Systemänderungen, Fehler in Programmen und manuelle Eingriffe über die Tastatur (<delete>-Taste) führen oder können zu Unterbrechungen führen. Dabei bleibt in der Regel die Auswirkung auf die Arbeitsumgebung, insbesondere auf das Dateisystem, undefiniert. Temporäre Dateien sind entstanden und müßten beseitigt werden, damit sie bei der Wiederaufnahme der Arbeit nicht stören.

Die Kommandointerpreter bieten für Unterbrechungen eigene Kommandos an. Die Bourne-Shell hat das Kommando trap, die C-Shell hat für den gleichen Zweck das Kommando onintr. In beiden Fällen werden Maßnahmen für bestimmte Interruptsignale präzisiert. Maßnahmen sind wieder als abzuarbeitende Kommandos zu verstehen, die beispielsweise ungültige Dateien beseitigen.

Die UNIX-Arbeit ordnet sich so dem J.-v.-Neumann-Modell der Rechnerarchitektur unter. Dessen wesentliche Abläufe bestehen in der zyklischen Wiederholung der Folge

- Operanden bereitstellen (aus dem Speicher, aus der Umgebung oder aus dem Dateisystem),

- Kommando abarbeiten (in einem Prozeß) und

- Unterbrechungsereignisse behandeln.

Für das Verständnis der UNIX-Arbeit ist es durchaus sinnvoll, sich diese Analogiebetrachtung immer wieder einmal zu vergegenwärtigen.

Die Unterbrechungsbehandlung hat eine besondere Bedeutung bei der Formulierung von Kommandodateien, sogenannten Shell-Skriptdateien, die eine Folge von Einzelkommandos zusammenfassen und damit komplexere Aufgabenstellungen beschreiben können als beispielsweise die obigen sieben kleinen Kommandos in diesem Einführungsabschnitt. Sobald ein Nutzer die Anfänge des Lernstadiums hinter sich gelassen hat, sollte er sich unbedingt mit diesem dritten Aspekt der UNIX-Arbeit vertraut machen.

# 2. Einiges über Dateien

## 2.1. Elemente der Dateiarbeit

Dateien in UNIX sind Folgen von Bytes; Folgen von Bytes - und nichts weiter - könnte man ergänzen wollen. Dazu gehören zuerst gewöhnliche Texte. Jedes Zeichen wird in seiner ASCII-Kodierung als ein Byte dargestellt. Auch abarbeitbare Programme sind Folgen von Bytes. Hier bedeutet ein Byte oder mehrere aufeinanderfolgende Bytes eine Instruktion, die durch die CPU interpretiert werden kann. Ebenso sind Dateiverzeichnisse Folgen von Bytes, also Dateien. So sei an das mit dem Kommando ls -al sichtbar gewordene Verzeichnis zum Projekt accu erinnert.

UNIX versteht auch periphere Geräte als Dateien. Befinden sich in den vorigen Interpretationen die Bytefolgen längerfristig in irgendeinem Speicher, so werden Dateien, die Geräte repräsentieren, bei ihrer Verwendung erst erzeugt oder sofort verarbeitet. Solche Dateien heißen »special files«. Sie sind in dem zentralen Dateiverzeichnis /dev (devices) zusammengefaßt.

Soll beispielsweise eine Datei von einer Diskette im Floppy-Laufwerk 1 gelesen werden, so kann die Datei fd1 verwendet werden. Durch die Arbeit des Gerätetreibers wird auf der Diskette eine Bytefolge ermittelt und unter dem Namen fd1 einmal byteweise bereitgestellt. Vielleicht sollte sie als Eingabedatei eines Verarbeitungsprogramms sofort verwendet werden oder als Operand des Kommandos cp in eine einfache Datei kopiert werden. Gleichermaßen bedeutet die Angabe der Gerätedatei lp1 etwa auf der Ausgabeposition eines Verarbeitungsprogramms die sofortige und einmalige Verwertung dieser Bytefolge, indem sie Zeichen für Zeichen gedruckt wird.

Weitere Vertreter von Dateien sind die sogenannten Pipes, die ebenfalls in dem im Abschnitt 1. abgedruckten Ausschnitt aus dem ls-Manualblatt des Programmierhandbuches erwähnt wurden. Ein Pipe verbindet die Standardausgabe eines Programms mit der Standardeingabe eines anderen. Oft werden mehrere Pipes aneinandergereiht, so daß Ergebnisse von Programmen mehrfach gefiltert oder weitergereicht werden können. Wenn Pipes einen Namen erhalten, werden sie in den Kreis der Dateien aufgenommen.

UNIX stellt jedem Programm drei Textdateien zur Verfügung:

- eine Eingabedatei mit der Deskriptorbezeichnung stdin,
- eine Ausgabedatei mit der Deskriptorbezeichnung stdout und
- eine Datei mit der Deskriptorbezeichnung stderr, empfohlen für die getrennte Ausgabe von Fehlernachrichten, oft auch als Diagnoseausgabe bezeichnet.

In der Liste der Dateideskriptoren haben sie der Reihe nach die Nummern 0, 1 und 2. Für jedes Kommando oder Verarbeitungsprogramm sind standardmäßig alle drei Dateien mit dem Terminal verbunden, an dem der Benutzer arbeitet.

### 2.1.1. Verfassen einer Textdatei

Wir geben eine Datei zeilenweise ein und nutzen dazu den Editor ed, der zum zentralen Bestand der UNIX-Kommandos gehört. Anknüpfend an das Akkumulationsbeispiel aus Abschnitt 1. soll die gleiche Aufgabe noch einmal in der Programmiersprache C beschrieben werden. Für die weiteren Betrachtungen wird sich das als nützlich erweisen.

Das folgende kommentierte Terminalskript zeigt die Eingabe des Programms accu.c und einige sich unmittelbar anschließende Schritte. Dieses und weitere Demonstrationsbeispiele in diesem Buch werden mit dem Kommando script hergestellt. Es sammelt alle Terminalausgaben in einer Datei, so daß die Dialogarbeit am Sitzungsprotokoll noch einmal nacherlebt werden kann. Anhang 1 enthält die Manualseiten für dieses Kommandoprogramm als Mustertext.

```
Script started on Wed Jan 27 09:59:50 1988
Neue C-Shell
1% set prompt='%next csh-cmd: '
%next csh-cmd: ed -p '#next ed-cmd:   accu.c
?accu.c
```

Das Kommando 1% ändert das Promptzeichen des Kommandointerpreters um. Neues Promptzeichen für die Aufforderung zur Eingabe von Kommandos ist der Text '%next csh-cmd: '. Das Kommando ed ruft den Editor auf. Als Promptzeichen des Editors wird der Text '#next ed-cmd ' verabredet. Die Datei accu.c soll ediert werden. Da es diese Datei noch nicht gibt, erscheint die Ausschrift ?accu.c. Der Editor fordert zur Eingabe des ersten Kommandos auf.

```
#next ed-cmd: a
main() {
int key,val;
int sum,totsum;
printf("Bericht\n");
}
.
```

Mit dem Editorkommando a kann an die aktuelle Edierposition eine Folge von Zeilen angefügt werden. Sie endet mit einer Zeile, die aus den Zeichen .<newline> besteht. Das <newline>-Zeichen entsteht durch Betätigen der <return>-Taste. Die Zeilen des zu edierenden Textes werden numeriert. Das Kommando p druckt die Zeilen 1 bis $. Das Zeichen $ steht für die Nummer der letzten Zeile.

```
#next ed-cmd: 1,$p
main() {
int key,val;
int sum,totsum;
printf("Bericht\n");
}
```

Es folgt ein append-Kommando, das den Eingabetext an die Zeile mit dem Textmuster /printf/ anhängt.

```
#next ed-cmd: /printf/a
totsum = 0;
scanf("%d",&key);
while (key == 1) {
  sum = 0;
  scanf("%d",&key);
```

```
    while (key == 0) {
      scnaf("%d",&val);
      printf("%10d\n",val);
      sum += val;
      scanf("%d",&key);
    }
    totsum += sum;
    printf("Gruppensumme%10d\n",sum);
  }
  printf("Gesamtsumme %10d\n",totsum);
  .
  #next ed-cmd:
  }
```

Die Zeile .<newline> beendet den Text. Danach drücken wir erneut auf die <return>-Taste (ohne Zeicheneingabe). Am Bildschirm wird dadurch die Zeile sichtbar, die auf den eingefügten Text folgt.

Offensichtlich haben wir uns bei der Texteingabe verschrieben. Daher positionieren wir mit dem nächsten Editorkommando auf die Zeile mit dem fehlerhaften Text als Muster. Mit dem Substitutionskommando s wird das bisherige Suchmuster // durch den richtigen Text scanf ersetzt und die korrigierte Zeile ausgegeben.

```
  #next ed-cmd: /scnaf/
      scnaf("%d",&val);
  #next ed-cmd: s//scanf
      scanf("%d",&val);
  #next ed-cmd: w
  344
  #next ed-cmd: q
```

Der Editor ed arbeitet mit einer Kopie der Datei. Mit dem Kommando w wird diese Kopie in das Dateisystem geschrieben. Wäre die Datei accu.c bereits vorhanden gewesen, wäre sie damit überschrieben worden. Das quit-Kommando q beendet die Arbeit des Editors. Danach meldet sich der Kommandointerpreter mit seinem Promptzeichen wieder und fordert zur Eingabe des nächsten Kommandos auf. Mit dem Kommando cat wird das Ergebnis unserer Editorarbeit sichtbar gemacht.

```
  %next csh-cmd: cat accu.c
  main() {
  int key,val;
  int sum,totsum;
  printf("Bericht\n");
  totsum = 0;
  scanf("%d",&key);
  while (key == 1) {
    sum = 0;
    scanf("%d",&key);
    while (key == 0) {
      scanf("%d",&val);
      printf("%10d\n",val);
      sum += val;
      scanf("%d",&key);
    }
    totsum += sum,
    printf("Gruppensumme%10d\n",sum);
  }
  printf("Gesamtsumme %10d\n".totsum);
  }
```

Die restlichen Kommandos spielen ein wenig mit dem erstellten C-Text. Einige darin benutzte Möglichkeiten werden teilweise erst in den folgenden Abschnitten erklärt.

```
%next csh-cmd: make accu
    cc -O  accu.c -o accu
accu.c
    rm -f accu.o
%next csh-cmd: pwd
/u/polze/priv/book/cont
%next csh-cmd: cp `pwd`/T*/accu.inp
%next csh-cmd: accu <accu.inp
Bericht
        10
            20

     ... usw. ...
    Ergebnisse wie im Kap. 1

Gruppensumme         48
Gesamtsumme         200
%next csh-cmd: !! >f
accu < accu.inp > f
f: File exists.
%next csh-cmd: !!f
accu < accu.inp > ff
%next csh-cmd: cmp ff `pwd`/T*/accu.res
%next csh-cmd: echo $status
0
%next csh-cmd: rm f ff
%next csh-cmd:
script done on Wed Jan 27 10:19:18 1988
```

Das Technologieprogramm make stellt aus der Quelldatei accu.c die abarbeitbare Datei accu her. Bemerkenswert ist es, daß make im Fall von C-Texten noch nicht einmal die Datei makefile benötigt. make protokolliert die abgearbeiteten Kommandos. Das Resultat accu wird mit den weither besorgten, aber noch vorliegenden Eingabedaten accu.inp, die bereits im vorigen Abschnitt benutzt wurden, testhalber abgearbeitet. Das Testergebnis wird mit den ebenfalls noch vorhandenen Resultaten in der Datei accu.res verglichen. Das leistet das Kommando cmp. Danach hat die Shell-Variable status die Anzahl der abweichenden Bytes als Wert. Dieser ist 0. Es gibt keine Abweichungen. Am Ende werden die Hilfsdateien f und ff beseitigt.

Bei dieser Arbeit sind mehrere Dateien entstanden. Eine Übersicht gibt unser Kommando ls.

```
% ls -l
total 34
-rwxr-xr-x   1 polze    HUB     10483 Jan 27 10:15 accu
-rw-r--r--   1 polze    HUB       344 Jan 27 10:14 accu.c
-rw-r--r--   1 polze    HUB        56 Jan 27 10:16 accu.inp
-rw-rw-r--   1 polze    HUB      1428 Jan 27 10:30 ed.script
```

Mit dem Kommando file erhält man eine Klassifizierung.

```
% file *
accu:    executable not stripped
accu.c: c program text
accu.inp:        ascii text
ed.script:       c program text
```

Das Zeichen * als Operand im Kommandoaufruf file steht für alle Dateien im aktuellen Verzeichnis. Die Datei ed.script beinhaltet das oben abgedruckte Terminalskript. Die dabei sichtbare Unschärfe bei der Klassifizierung durch das Kommando file läßt sich gewiß nicht vermeiden. Ausführbare Programme haben nach ihrer Erzeugung durch einen Compiler und Lader noch ein Verzeichnis externer Symbole in ihrem Bestand. Darauf können beispielsweise die Debugger adb oder sdb zugreifen. Mit dem Kommando strip werden ausführbare Dateien von dieser zusätzlichen Information befreit. Die Formulierung executable not stripped ist entsprechend zu verstehen.

### 2.1.2. Einige wichtige Kommandos zur Dateiverarbeitung

Der alltägliche Umgang mit UNIX ist wesentlich durch die Verarbeitung von Dateien bestimmt. Einige Kommandos dominieren in der Häufigkeit ihrer Anwendung. Sie sollen kurz vorgestellt werden. Als Orientierung liegt der Standardisierungsvorschlag von X/OPEN (1987) zugrunde. Nicht alle zur Zeit verfügbaren UNIX-Systeme besitzen alle diese Kommandos sowie alle Optionen.

Das Kommando ls druckt den Inhalt von Dateiverzeichnissen oder beschreibt die als Parameter angeführten Dateien.

```
ls [options] file1 file2
ls [options] dir1 dir2
```

Ohne Parameter druckt ls den Inhalt des aktuellen Dateiverzeichnisses.

```
ls -a  Ausgabe aller Dateinamen, auch der .-Dateien.
ls -l  Ausgabe im langen Format.
ls -i  Ausgabe auch der Inode-Nummern.
ls -s  Ausgabe der Dateigrößen in Bytes.
ls -d  Beschreibung der Verzeichnisdatei.
ls -u  Druckreihenfolge nach der letzten Benutzung.
ls -R  Rekursiv nachfolgende Verzeichnisse werden erfaßt.
```

Das Kommando cat verkettet Dateien und gibt das Resultat auf die Standardausgabedatei stdout aus.

```
cat [-s] file1 file2
```

Sind keine Dateinamen angegeben, arbeitet cat mit der Standardeingabedatei stdin. Ebenso bedeutet der Dateiname - die Standardeingabe. Die Option -s unterdrückt Ausschriften von cat, wenn angegebene Dateien nicht gefunden werden.

```
cat file          Protokolliert die Datei file.
cat f1 f2 >res    Verkettet f1 mit f2 und erzeugt als
                  Ergebnis die Datei res.
cat f1 - f2 >res  Holt die Datei f1, nimmt Eingaben von der
                  Standardeingabe entgegen (Ende Ctrl-d),
                  holt f2 und bringt das Resultat nach res.
```

| | |
|---|---|
| `cat >file` | Nimmt Eingaben von der Standardeingabe und erzeugt die Datei file. |
| `cat f1 f2 >f1` | Überschreibt den Originaldatenbestand der ersten Datei. |

Das Kommando cp kopiert Dateien. Es kann in zwei Formen angewendet werden.

```
cp file1 file2
cp f1 f2 ... fn dir
```

Von der Datei file1 wird eine Kopie file2 hergestellt. Der Datenbestand wird dupliziert. Die beiden Dateien müssen voneinander verschieden sein. Eine bestehende Datei file2 wird überschrieben. Ihre Zugriffsrechte, Eigentümer- und Gruppennummer werden an die neue Datei file2 vererbt.

Die Dateien f1 ... fn werden kopiert und in das Dateiverzeichnis dir aufgenommen. Hierbei wird für jede Datei f1 ... fn ein Duplikat hergestellt.

Das Kommando mv benennt Dateien eines Dateisystems ohne Duplizierung des Datenbestandes um (s. Abschnitte 2.2. und 2.3.2.).

```
mv [-f] file1 file2
mv [-f] f1 f2 ... fn dir
mv [-f] dir1 dir2
```

Die erste Form gibt der Datei file1 den neuen Namen file2, ohne den Datenbestand zu verdoppeln. Die zweite Form ordnet die Dateien f1 ... fn dem Dateiverzeichnis dir zu. Die dritte Form ist eine Umbenennung des Dateiverzeichnisses dir1. Der neue Name ist dir2. Bestehende Dateien oder Verzeichnisse werden überschrieben. Falls dies den Zugriffsrechten widerspricht, wird der Benutzer am Terminal gefragt, ob er seine Angaben wirklich ernst gemeint hat. Mit der Eingabe von y bestätigt er sein Kommando, und es wird ausgeführt. Im Falle der Option -f wird nicht nachgefragt.

Das Kommando rm beseitigt Dateien oder Verzeichnisse. So streicht das Kommando

```
rm [-fri] file1 file2
```

die als Parameter angegebenen Dateien. Die Option -f schaltet eventuelle Nachfragen aus. Bei der Option -i arbeitet das Kommando interaktiv. Bei jeder Datei fragt es den Nutzer, ob die Datei wirklich beseitigt werden soll. Die Option -r steuert den rekursiven Aufruf über Dateiverzeichnisse hinweg.

Das Kommando pr dient der Druckaufbereitung von Dateien.

```
pr [options] file1 file2
```

Die Dateien file1, file2, ... werden aufbereitet. Ohne Dateinamen als Parameter wirkt dieses Kommando auf die Standardeingabe. Ebenso bedeutet der Dateiname - die Standardeingabedatei. Das Resultat gelangt in die Standardausgabedatei. Viele Optionen stehen zur Verfügung. Die Zahl der Zeichen je Zeile oder die Zahl der Zeilen je Seite können vorgegeben werden. Mehrspaltendruck, Einrückungen, Tabulatormodifikationen oder Zeilennumerierung sind möglich. Seiten werden mit Kopf- oder Endtexten eingefaßt.

```
pr -l72         Druckt 72 Zeilen je Seite.
pr '-n 3'       Druckt dreistellige Zeilennummern,
                mit einem Leerzeichen vom Text getrennt.
pr -t           Druckt ohne den 5zeiligen Kopftext und ohne
                den 5zeiligen Endtext.
pr -h name      Ersetzt im Kopftext den Dateinamen durch name.
pr -2           Druckt zweispaltig.
pr -2m f1 f2 Druckt die Datei f1 in Spalte 1 und die
                Datei f2 in Spalte 2.
```

Das Kommando wc zählt Zeilen, Wörter und Zeichen in den als Parameter angegebenen Dateien. Fehlen Dateinamen, so wird die Standardeingabedatei behandelt. Das Resultat kommt in die Standardausgabedatei.

```
wc [-lwc] file1 file2
```

Die Optionen -l (Zeilen), -w (Wörter) und -c (Zeichen) sind standardmäßig alle eingeschaltet. Hier ein Druckbild, das von wc erzeugt wurde:

```
% wc *
     53     123    8408 accu
     20      40     344 accu.c
     12      20      56 accu.inp
     91     225    1637 ed.script
      5      17     117 file.prot
    181     425   10562 total
```

Für Dateien, die keine Zeilen- oder Textstruktur haben, gibt es andere Kommandos zur quantitativen Bestimmung, so das Kommando size für die Feststellung der Größe von Objektkodedateien.

```
size [-o][-x] file1 file2
```

gibt die Größe der einzelnen Abschnitte an. Die Zahlen bedeuten Länge des Textsegments, Länge des Bereichs der initialisierten Daten sowie des nicht initialisierten Datenbereichs (BSS). Dynamisch angeforderte Speicherplätze werden dabei nicht mitgerechnet. Alle Längenangaben zählen in Bytes. Mit der Option -o werden die Zahlen oktal dargestellt, mit der Option -x hexadezimal, sonst werden sie dezimal notiert. Es folgt die Summe der drei Längen, danach noch in hexadezimaler Form. Anwendung des Kommandos size auf Archive liefert die Daten für alle Objekte, die darin versammelt sind. Es folgen einige Druckbilder.

```
# size /bin/ed /bin/sed /bin/vi
/bin/ed: 22857 + 3626 + 9046 = 35529 = 0x8ac9
/bin/sed: 15500 + 1972 + 24860 = 42332 = 0xa55c
/bin/vi: 107911 + 4144 + 31840 = 143895 = 0x23217

% size [PC]accu/accu
Caccu/accu: 7564 + 718 + 1254 = 9536 = 0x2540
Paccu/accu: 16292 + 1168 + 2636 = 20096 = 0x4e80
```

Zwei Kommandos ermöglichen den Vergleich zweier Dateien Byte für Byte: das Kommando cmp und das Kommando diff.

```
cmp  [-s]  file1 file2
diff [-eb] file1 file2
```

Das Kommando cmp teilt einen entdeckten Unterschied oder das erreichte Ende einer Datei mit. Falls beide Dateien einander gleich sind, kommt keine Nachricht. Die Option -s läßt cmp überhaupt still und schweigsam arbeiten. Das Ergebnis wird immer in einer Statusvariablen des Kommandointerpreters festgehalten. Dies ist bei beiden Kommandos der Fall. Statuswert 0 bedeutet kein Unterschied, Wert 1 weist auf bestehende Unterschiede hin, Statuswert 2 kommt zustande, falls Fehler, etwa Zugriffsverstöße, auftraten. Auf dem Platz von file1 bedeutet - die Standardeingabedatei. Beim Kommandop diff kann auch file2 als - mit der gleichen Bedeutung gewählt werden.

Das Kommando diff berichtet alle Unterschiede der beiden Dateien file1 und file2. Es liefert Ausgaben der Form:

```
n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4
```

Dies entspricht den Kommandos der Editoren ed oder sed, die noch ausführlich erklärt werden. Die Zeilennummern links gehören zur Datei file1, rechts zur Datei file2. Gleiche Nummern werden, wie bei den Editoren zulässig, nur einmal notiert. Jeder solchen Angabe folgen Zeilen

```
<text
```

aus der Datei file1 oder Zeilen

```
>text
```

aus der Datei file2. Die Texte sind gegebenenfalls durch eine Zeile --- getrennt. Diese Ausgaben lassen sich für eine manuelle Verarbeitung mit dem Editor ed verwenden. Sie können aber auch in eine Skriptdatei für den Stream-Editor sed umgeformt werden, so daß mit einem Aufruf von sed die Datei file2 aus der Datei file1 hergestellt werden kann. Dies wird im Abschnitt über Editoren vorgeführt. Bei der Option -e werden die Ausgaben so aufbereitet, daß eine Transformation von der Datei file2 in die Datei file1 ermöglicht wird. Die Option -b bewirkt, daß beim Dateivergleich führende Leerzeichen oder <tab>-Zeichen unterdrückt werden.

Das Kommando echo reproduziert seine als Parameter angegebenen Argumente auf der Standardausgabedatei.

```
echo arg1 arg2     argn
```

Die Argumente werden durch Leerzeichen getrennt. Ein <newline>-Zeichen bildet das Ende der Argumenteliste. In den Argumenten dürfen Sonderzeichen auftreten.

```
\b    <backspace>-Zeichen.
\c    Druckt Argumente bis zu diesem Zeichen ohne <newline>,
      der Rest der Kommandozeile wird ignoriert.
\f    <formfeed>-Zeichen.
\n    <newline>-Zeichen.
```

```
\r    <carriage-return>-Zeichen.
\t    <tab>-Zeichen.
\v    <vertical-tab>-Zeichen.
\\    <backslash>-Zeichen.
\0n   n ein-, zwei- oder dreistellige Oktalzahl als Zeichenkode.
```

Das Kommando echo beweist seine Nützlichkeit vor allem dann, wenn die Argumente irgendwelchen Substitutionen unterzogen werden. Beispielsweise können Variable der Umgebung oder Variable der Kommandointerpreter als Argumente auftreten. Der Aufruf von echo zeigt ihre Werte. Einschlüsse in Quotezeichen haben unterschiedliche Wirkungen. In Skriptdateien bewährt sich das Kommando echo als Hilfsmittel für die Ausgabe von Protokolltexten an das Terminal oder in Diagnosedateien.

```
% echo Umgebungsvariable SHELL
Umgebungsvariable SHELL
% echo Umgebungsvariable $SHELL
Umgebungsvariable /bin/csh
% echo Umgebungsvariable '$SHELL'
Umgebungsvariable $SHELL
% echo Umgebungsvariable "$SHELL"
Umgebungsvariable /bin/csh
% echo Beispiel: '\n' "SH = $SHELL"
Beispiel:
 SH = /bin/csh
%
```

Das Sonderzeichen \c bricht die Textwiedergabe ohne <newline> ab.

```
% echo 'Bitte das naechste Kommando:\c'
Bitte das naechste Kommando:%
% echo 'Bitte\c das naechste Kommando:'
Bitte%
```

Das folgende Terminalprotokoll zeigt nochmals den Umgang mit dem Kommando echo, wobei hier fleißig mit Sonderzeichen gearbeitet wird. Die Implementationen des Kommandos echo unterscheiden sich stark voneinander. Innerhalb der Kommandointerpreter gehört echo zu den Shell-Kommandos, die dort als Unterprogramme realisiert sind. Die obige Beschreibung des Kommandos echo bezieht sich aber auf das ebenfalls vorhandene Kommandoprogramm. Daher wird durch die Umbenennung im Skriptkommando 2% vom C-Shell-Kommando echo auf das Kommandoprogramm echo umgeschaltet. Dabei wird gleich ein Unterschied in der Wirkungsweise der beiden echo sichtbar. Die verwendeten Oktalkodes gehören dem erweiterten ASCII-Zeichensatz an und sind nicht an jedem Terminal verfügbar.

```
Script started on Fri Mar  4 13:15:54 1988
Neue C-Shell
1% echo '\0204'
\0204
2% alias echo '/bin/echo'
3% echo '\0204'
ä
4% echo '\0216, der deutsche Umlaut l\0204\0341t sich schwer \
ausdr\0201cken.'
Ä, der deutsche Umlaut läßt sich schwer
ausdrücken.
```

```
5% echo 'Hier sind sie alle: \
\0257\0216\0204 \0231\0224 \0232\0201 und \0341\0256'
Hier sind sie alle:
»Ää Öö Üü und ß«
6%  !! >odfile1
echo 'Hier sind sie alle: \
\0257\0216\0204 \0231\0224 \0232\0201 und \0341\0256' > odfile1
7%
script done on Fri Mar  4 13:26:36 1988
```

Oftmals werden Umlaute durch Überdrucke künstlich hergestellt. Das bietet gleich noch ein Beispiel für die Verwendung von Sonderzeichen im Kommando echo, dargestellt mit dem folgenden Terminalskript. Beide Skripts, das vorangehende und dieses, liefern mit ihren letzten Kommandos bereits Beispieldateien für das anschließend zu erörternde Kommando od.

```
Script started on Fri Mar 11 09:05:08 1988
Neue C-Shell
1% alias echo /bin/echo
2% echo 'Ein Umlaut l"\bap\b3t sich k"\bunstlich f"\bugen.'
Ein Umlaut läßt sich künstlich fügen.
3% !! >odfile2
echo 'Ein Umlaut l"\bap\b3t sich k"\bunstlich f"\bugen.' >odfile2
4%
script done on Fri Mar 11 09:11:54 1988
```

Das Kommando od erzeugt verschieden formatierte Dumpdrucke einer Datei.

```
od [-bcdosx][file][offset]
```

Die Datei file wird oktal ausgegeben. Fehlt die Dateiangabe, so wird die Standardeingabedatei verarbeitet. Die Angabe offset legt einen Anfangspunkt fest. Optionen steuern die Ausgabeformate.

```
-b      Bytes werden oktal interpretiert.
-c      Bytes werden als ASCII-Zeichen dargestellt.
-d      Wörter werden dezimal dargestellt.
-o      Wörter werden oktal dargestellt.
-s      Wörter werden dezimal mit Vorzeichen notiert.
-x      Wörter werden hexadezimal ohne Vorzeichen notiert.
```

Hier einige Beispiele für die Arbeit des Kommandos od. Die Dateien odfile1 und odfile2 wurden bei der Demonstration des Kommandos echo erzeugt.

```
% od -bc odfile1
00000   H   i   e   r       s   i   n   d       s   i   e       a   l
      110 151 145 162 040 163 151 156 144 040 163 151 145 040 141 154
00020   l   e   :  \n 257 216 204     231 224     232 201       u   n
      154 145 072 012 257 216 204 040 231 224 040 232 201 040 165 156
00040   d     341 256  \n  \0
      144 040 341 256 012 000
00045
```

```
% od -bc odfile2
00000   E   i   n       U   m   l   a   u   t       l   "  \b   a   p
      105 151 156 040 125 155 154 141 165 164 040 154 042 010 141 160
00020  \b   3   t       s   i   c   h       k   "  \b   u   n   s   t
      010 063 164 040 163 151 143 150 040 153 042 010 165 156 163 164
00040   l   i   c   h       f   "  \b   u   g   e   n   .  \n
      154 151 143 150 040 146 042 010 165 147 145 156 056 012
00056
```

Die Kommandos grep, egrep und fgrep bilden eine Familie von Programmen. Sie dienen dazu, Zeilen in Dateien zu finden, die durch Suchmuster beschrieben werden.

```
grep  [Options]  expr  [files]
egrep [Options] [expr] [files]
fgrep [Options] [strings] [files]
```

Sind keine Dateien als Parameter vorhanden, so wird die Standardeingabedatei behandelt. Ansonsten werden die Dateien eine nach der anderen bearbeitet. Als Suchmuster dienen reguläre Ausdrücke expr (bei grep) oder erweiterte reguläre Ausdrücke (bei egrep). Im Falle von fgrep werden Zeichenketten oder Folgen von Zeichenketten akzeptiert. Wenn Zeilen Zeichenketten enthalten, die das Suchmuster erfüllen oder mit den Vergleichszeichenketten bei fgrep übereinstimmen, werden diese auf die Standardausgabe geleitet. Einige Optionen seien genannt:

```
-v      Ausgesucht werden Zeilen, die das Suchmuster
        nicht erfüllen.
-x      Nur wenn die ganze Zeile einer Zeichenkette
        gleich ist, wird sie ausgegeben (fgrep).
-c      Es wird nur die Anzahl der gefundenen Zeilen
        ausgegeben.
-f file Der Ausdruck expr oder die Zeichenketten strings
        werden der Datei file entnommen (egrep, fgrep).
-n      Zeilennummern werden vorangestellt.
```

Alle drei Kommandos setzen eine Statusvariable. Sie bekommt den Wert 0, wenn Zeilen gefunden wurden, den Wert 1, wenn keine Zeilen gefunden wurden, und den Wert 2, wenn syntaktische Fehler auftraten oder Parameterdateien den Zugriff verweigerten.

Das Kommando sort stellt ein sehr flexibles Sortierprogramm dar. Es sortiert die Zeilen der als Parameter angegebenen Dateien file1, file2, ... oder der Standardeingabedatei, falls keine Parameterdateien angegeben sind. Es hat viele Aufrufmöglichkeiten. Einige seien hier wiedergegeben.

```
sort [-cmu][-ofile][-dfinr][-btx][+pos1 [-pos2]]     file1 file2
```

Generell wird lexikographisch nach dem ASCII-Kode sortiert. Mit den Optionen des Kommandos sind verschiedene Modifikationen möglich. Die aufgeführten Optionen haben folgende Bedeutungen:

```
+pos1 [-pos2]
        Begrenzt das Sortiermerkmal. Die Zeichen einer Zeile werden
        in Felder eingeteilt. Erstes Leerzeichen oder erstes <tab>-
        Zeichen dient als Trennzeichen. Weitere Zeichen dieser Art
        gehören zum nächsten Feld. pos1, pos2 sind Zahlen. Sie zäh-
```

len Felder. Fehlt -pos2, so wird das Sortiermerkmal bis zum Zeilenende ausgedehnt. Ohne pos-Optionen ist die ganze Zeile Sortiermerkmal.
+0 -1 +3 -5 bedeutet Feld 1 sowie die Felder 3, 4 und 5 bilden in dieser Reihenfolge verkettet das Sortiermerkmal.

-tx Legt x als Trennzeichen zwischen Feldern fest.
-b Ignoriert führende Leer- oder Tabulatorzeichen.
-d Beim Zeichenvergleich werden nur Buchstaben, Ziffern oder Leerzeichen berücksichtigt.
-f Groß- und Kleinbuchstaben werden identifiziert.
-i Beim nichtnumerischen Vergleich werden Zeichen außerhalb 040-0176 ignoriert.
-n Zeichenfolgen, die sich als Zahlen deuten lassen, werden arithmetisch sortiert.
-r Umkehr der Sortierordnung.
-ofile
Das Ergebnis des Sortiervorgangs gelangt in die Datei file (sonst in die Standardausgabedatei).
-c Die Sortierreihenfolge wird nur überprüft.
-m Die bereits sortierten Eingabedateien werden gemischt.
-u Unterdrückt redundante Ausgaben. Zeilen mit gleichem Sortiermerkmal werden nur einmal ausgegeben.

Verschiedene Kommandos dienen dazu, Dateien in Portionen auf dem Bildschirm sichtbar zu machen. Zu nennen sind

| | |
|---|---|
| more | (WEGA, VMX, XENIX, VENIX) |
| pg | (XENIX, X/OPEN) |
| dog | (WEGA) |

Einen hohen Komfort weist das Kommando pg auf. Mit seiner Hilfe kann man in der Datei blättern, vorwärts, rückwärts, durch Nummern oder durch Suchmuster gesteuert. Alle Kommandos wirken auf die Standardeingabe, wenn keine Dateien als Paramater angegeben sind. Sonst werden die Dateien der Reihe nach jede für sich behandelt.

Das Kommando tail liefert den Schlußteil einer Datei.

```
tail [+-[n][lbc[f]]][file]
```

gibt den Schlußteil der Datei file in die Standardausgabedatei. Ohne die Angabe file wird die Standardeingabe als Datenquelle genommen. Die Optionen haben folgende Bedeutung:

+n Die Kopie beginnt an der Zeile n.
-n Die Kopie beginnt n Zeilen vor dem Dateiende (standardmäßig gilt n=10).
l n bedeutet Zeilen.
b n bedeutet Blöcke zu je 512 Bytes.
c n bedeutet Zeichen.
f Folgeoption.

Die Folgeoption f stellt nach jeweils einer Sekunde n Texteinheiten bereit. Damit können in einer Datei, die etwa ihren Inhaltszuwachs aus einem anderen Prozeß bezieht, jeweils die zuletzt erzeugten n Zeichen oder Zeilen sichtbar gemacht werden.

Das Kommando touch aktualisiert für die angegebenen Dateien die Zeitpunktattribute, den Zeitpunkt des letzten Zugriffs oder der letzten Änderung.

```
touch [-amc][format] file
```

Die Option format legt die Form der Zeitangabe fest. Die anderen Optionen bedeuten:

-a Zeitpunkt des letzten Zugriffs.
-m Zeitpunkt der letzten Modifikation.
-c Einrichten einer leeren Datei.

Das Kommando touch richtet eine leere Datei mit dem Namen file und aktuellen Zeitpunktattributen ein, wenn es keine Datei mit dem Namen file gibt. Standardmäßig sind alle drei Optionen eingeschaltet.

Das Kommando tar dient dem Dateitransfer. Es wird später ausführlich beschrieben und in Anwendungen vorgeführt.

Das Kommando chmod gestattet es, die Zugriffsrechte der als Parameter notierten Dateien zu ändern oder festzulegen.

```
chmod mode file1 file2
```

Zum Verständnis muß beschrieben werden, wie UNIX Zugriffsrechte definiert und verwaltet. UNIX unterscheidet

r Lesezugriff,
w Schreibzugriff,
x Abarbeitung.

Zudem regelt UNIX diese Rechte getrennt für

u Eigentümer,
g Mitglieder der Nutzergruppe, zu der auch der Eigentümer gehört,
o andere Nutzer.

Das ergibt die Kombinationsmöglichkeiten, die bereits mehrfach bei Ausgaben des Kommandos ls -l zu sehen waren.

```
rwxrwxrwx
```

Sie beziehen sich von links nach rechts auf Eigentümer, Gruppenmitglieder und andere Nutzer. Eintrag r, w oder x bestätigt das Zugriffsrecht, sonst erscheint ein Minuszeichen.

Die mode-Angabe kann in der folgenden symbolischen Form angegeben werden:

```
[wer] op erlaubniswert ...
```

Der Vorsatz wer ist eine Zeichenfolge aus u, g und o. Als Standard gilt wer=ugo. Operatoren op sind +, - oder =. Als erlaubniswert kann eine Zeichenfolge aus r, w, x und s zusammengestellt werden.

Hier einige Beispiele:

```
chmod u+x file
    Die Datei file ist für den Eigentümer ausführbar.
chmod ug+x file
    Der Eigentümer und die Mitglieder seiner Gruppe
    dürfen die Datei file starten.
chmod ug+x o= file
    Wie vorher. Den anderen Nutzern werden jegliche
    Zugriffsrechte versagt.
chmod +x file
    Die Datei file ist für alle ausführbar.
chmod o-w file
    Nutzer außerhalb der Gruppe werden Schreibzugriffe
    verwehrt.
chmod u+s file
    Die Ausführungsbedingungen werden modifiziert.
```

Eine ausführbare Datei mit dem Eintrag s für den Eigentümer oder für die Gruppe wird unter diesem Nutzernamen oder Gruppennamen ausgeführt, auch wenn sie von einem beliebigen anderen Nutzer mit Ausführungsberechtigung gestartet wird. Ein typischer Vertreter von Dateien dieses Modus ist das ausführbare Kommando passwd, das alle Nutzer ausführen können, wenn sie ihr Passwort ändern wollen. Sein Eigentümer ist das System. (root, system oder ähnliches findet man in den entsprechenden Ausschriften von ls -l.) Während der Arbeit dieses Kommandos ist das System der effektive Nutzer, obwohl es von einem nicht bevorrechtigten Benutzer gestartet wurde.

Neben der symbolischen Form der mode-Angabe ist eine oktale Notation zulässig. Die Zugriffsrechte für die Nutzerklassen u, g und o werden in drei Oktalziffern verschlüsselt. Die Datei file sei mit Zugriffsrechten 644 ausgestattet. Dies entspricht der Angabe rw-r--r--, wie sie bei ls -l ausgegeben würde. Dann können die ersten Beispiele der obigen Reihe auch durch die folgenden chmod-Kommandos ersetzt werden:

```
chmod 744 file    ≡    rwxr--r--
chmod 754 file    ≡    rwxr-xr--
chmod 750 file    ≡    rwxr-x---
chmod 755 file    ≡    rwxr-xr-x
```

Die s-Angaben werden als linke Ziffer vor den drei Oktalziffern der Nutzerklassen kodiert. Folgende Druckbilder von ls -l und absolute mode-Angaben des Kommandos chmod entsprechen einander. Falls das s-Bit verlangt wird, das x-Bit jedoch fehlt, wird statt s der Buchstabe S geschrieben.

```
rwsr-xr-x   ≡   chmod 4755
rwSr-xr-x   ≡   chmod 4655
rwsr-Sr--   ≡   chmod 6744
```

Das Kommando umask definiert eine Maske für die Fixierung von Zugriffsrechten neu entstehender Dateien.

```
umask [oct]
```

Die Oktalzahl oct ist dreistellig. Die Ziffern gehören der Reihe nach zum Eigentümer, zu dessen Gruppe und zu den anderen Nutzern. Fehlt die Angabe oct, so gibt umask die augenblicklich gültige Maske bekannt. Wenn eine Datei beispielsweise durch den Editor bei der Eingabe eines Textes oder durch einen Compiler oder Lader als ein

ausführbares Programm mit dem Zugriffsrecht

```
    y   erzeugt wird und wenn die durch umask
        festgelegte Maske
    x   ist, so erhält die Datei den Modus
 ~x&y   (~x ist die bitweise Negation von x).
```

Das folgende Terminalskript zeigt einige Beispiele. Das Kommando touch erzeugt leere Dateien mit den Zugriffsrechten 666. Diese entsprechen rw-rw-rw-.

```
Script started on Thu Mar 31 09:42:28 1988
Neue C-Shell
1% alias t 'touch \!*   ls -l \!*'
2% umask
022
3% t d
-rw-r--r--   1 polze    HUB            0 Mar 31 09:44 d
4% umask 027
5% t dd
-rw-r-----   1 polze    HUB            0 Mar 31 09:46 dd
6% umask 000
7% t ddd
-rw-rw-rw-   1 polze    HUB            0 Mar 31 09:47 ddd
8% umask 022
9%
script done on Thu Mar 31 09:48:11 1988
```

## 2.2. Dateisysteme und ihre Struktur

Das Dateiverwaltungssystem des UNIX hat gewiß mit einem beträchtlichen Anteil dazu beigetragen, daß dieses Betriebssystem solche Anerkennung und Verbreitung gefunden hat. Es geht auf Ideen und Ansätze zurück, die bereits mit den ersten Time-Sharing-Systemen von Daley und Neumann (1965) publiziert worden sind.

### 2.2.1. Mehrstufige Verzeichnisse

UNIX unterscheidet vier Sorten von Dateien: einfache Dateien, Gerätedateien, Pipes und Verzeichnisdateien. In diesem Abschnitt beschäftigen wir uns mit einfachen Dateien und mit Verzeichnisdateien. Wir schließen uns an die Betrachtungen im Abschnitt 1. an und erinnern an die Dateien, die an der Arbeit des kleinen PASCAL-Programms beteiligt waren. Mit dem Kommando lc (lc ist eine XENIX-Variante des Kommandos ls)

```
% lc
accu      accu.inp  accu.pas  accu.res  makefile
```

werden ihre Namen aufgelistet. Das Kommando

```
% lc -a
          accu      accu.pas  makefile
          accu.inp  accu.res
```

liefert noch zwei weitere Einträge. Der Eintrag . bezeichnet dieses Verzeichnis als Datei. Sie ist eine Verzeichnisdatei und hat das Zeichen . als Namen. Der Eintrag  bezeichnet eine weitere Verzeichnisdatei.

Wenn nun Verzeichnisse Dateien sind und einen Namen haben, so kann man natürlich Kommandos zur Dateiverarbeitung auf diese Dateien anwenden. Mit dem Dump-Programm od können wir uns den Detailaufbau einer Datei ansehen. Da das Dateiverzeichnis zur PASCAL-Version, in dem wir uns gerade befinden, auch den Namen hat, werden wir das Kommando od auf diese Datei anwenden.

```
% od -c .
00000 373 004      \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00020   P 004   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00040 352 004   a   c   c   u  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00060 356 004   a   c   c   u       i   n   p  \0  \0  \0  \0  \0  \0
00100 353 004   a   c   c   u       p   a   s  \0  \0  \0  \0  \0  \0
00120 357 004   a   c   c   u   .   r   e   s  \0  \0  \0  \0  \0  \0
00140 364 004   m   a   k   e   f   i   l   e  \0  \0  \0  \0  \0  \0
00160
```

Das Druckprotokoll zeigt den zeichenweisen Aufbau dieser Verzeichnisdatei. Es ist sichtbar, daß Gruppen zu je 16 Byte jeweils eine der im Verzeichnis verwalteten Dateien definieren. Maximal 14 Zeichen werden für den Dateinamen genommen. Kürzere Namen werden mit Bytes 0 aufgefüllt. Betrachten wir die Verzeichnisdatei noch einmal wortweise dezimal, so wird sichtbar, daß die jeweils ersten beiden Bytes einer Eintragung Dezimalzahlen sind.

```
% od -cd .
00000   01275   00046   00000   00000   00000   00000   00000   00000
      373 004   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00020   01104   11822   00000   00000   00000   00000   00000   00000
        P 004   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00040   01258   25441   30051   00000   00000   00000   00000   00000
      352 004   a   c   c   u  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
00060   01262   25441   30051   26926   28782   00000   00000   00000
      356 004   a   c   c   u   .   i   n   p  \0  \0  \0  \0  \0  \0
00100   01259   25441   30051   28718   29537   00000   00000   00000
      353 004   a   c   c   u   .   p   a   s  \0  \0  \0  \0  \0  \0
00120   01263   25441   30051   29230   29541   00000   00000   00000
      357 004   a   c   c   u   .   r   e   s  \0  \0  \0  \0  \0  \0
00140   01268   24941   25963   26982   25964   00000   00000   00000
      364 004   m   a   k   e   f   i   l   e  \0  \0  \0  \0  \0  \0
00160
```

Diese Dezimalzahlen indizieren eine Tabelle, in der für jede in einem Dateisystem verwaltete Datei weitere Informationen über sie gesammelt sind. Dateiattribute sind die Adresse des Dateitextes im Speicher, Zeitpunktattribute, Zugriffsrechte und weitere Angaben. Mit dem Kommando

```
% lc -ia
1275            1258 accu       1259 accu.pas   1268 makefile
1104            1262 accu.inp   1263 accu.res
```

können diese Indexzahlen auch direkt ausgegeben werden.

Dateien in UNIX-Dateisystemen bestehen aus dem Text der Datei und einer Beschreibungsinformation. Der Text einer Datei wird blockweise gespeichert und verwaltet. Ein Dateisystem verwaltet von Texten belegte 512-Byte-Blöcke und freie Blöcke. Wenn Dateien gestrichen werden, können die frei gewordenen Blöcke in die Freispeicherverwaltung einbezogen werden. Wenn Dateien entstehen, stellt die Dateiverwaltung Platz aus der Menge der freien Blöcke zur Verfügung.

Die Beschreibungsinformationen heißen Inodes und werden als Tabelle verwaltet. Auch diese Tabelle hat belegte und freie Einträge. Wenn Dateien ihre Existenz beenden und der Tabelleneintrag frei wird, kann er für eine neuentstehende Datei wiederverwendet werden. Für das Verständis ist es vielleicht hilfreich, die Beschreibungsinformation durch eine C-Struktur anzudeuten. Sie verdeutlicht die Bestandteile von Inodes in der Form, wie sie auf einem externen Speichermedium abgelegt werden. Wenn eine Datei durch Zugriffe aktiviert wird, kommt die Inode-Information in den Hauptspeicher und wird dabei Teil einer umfassenderen Datenstruktur, auf die hier nicht weiter eingegangen werden soll. Die Dateiattribute werden jedoch gut sichtbar.

```
typedef unsigned short ushort; /* siehe /usr/include/sys/types.h */
typedef long           off_t;  /* offset in file */
typedef long           time_t; /* a time */

/* Inode structure as it appears on
 * a disk block.
 */

struct dinode   /* siehe /usr/include/sys/ino.h */
    {
      ushort di_mode;          /* mode and type of file */
      short  di_nlink;         /* number of links to file */
      ushort  di_uid;          /* owner's user id */
      ushort  di_gid;          /* owner's group id */
      off_t  di_size;          /* number of bytes in file */
      char   di_addr[40];      /* disk block addresses */
      time_t di_atime;         /* time last accessed */
      time_t di_mtime;         /* time last modified */
      time_t di_ctime;         /* time created */
    };
```

In der Komponente di_mode sind die Zugriffsrechte und der Typ der Datei bitweise verschlüsselt. In anderen Komponenten sind die Länge der Datei, Adressen auf dem Speichermedium, Eigentümernummern, Zeitpunktattribute und die Zahl der Verweise auf diese Datei untergebracht.

Normalerweise gehört zu UNIX ein einziges Dateisystem. Das ist eine Gesamtheit von einfachen Dateien, Verzeichnisdateien sowie Gerätedateien und Pipes. Als Speichermedium ist in der Regel ein Festplattenspeicher oder ein System von Großraumspeichern anzutreffen. Haben diese Speichermedien ein bestimmtes Mindestfassungsvermögen, so können gleichzeitig mehrere Dateisysteme eingerichtet werden. Jedes für sich verwaltet Inodes und Textblöcke sowie die entsprechenden Freilisten. Folgendes Kommando gibt einen Einblick in die vorliegende Situation.

```
% /etc/mount
/dev/root on / read/write on Fri Mar 25 13:00:04 1988
/dev/u on /u read/write on Fri Mar 25 13:00:06 1988
```

Die Ausschrift besagt, es sind zwei Dateisysteme montiert. Das erste Dateisystem residiert auf dem Gerät /dev/root. Dies ist das Dateisystem des UNIX überhaupt. Die darin versammelten Dateien werden durch das Wurzelverzeichnis / erfaßt. Dieses steht an der Spitze der Hierarchie von Dateiverzeichnissen, die UNIX verwaltet. Das zweite Dateisystem, das uns hier begegnet, befindet sich auf dem Gerät /dev/u. Die darin enthaltenen Dateien werden durch das Dateiverzeichnis /u erreicht. Da auch Geräte als Dateien behandelt werden, stehen die zwei Geräte, auf denen die beiden durch das Kommando mount ausgewiesenen Dateisysteme untergebracht sind, als Dateien im Wurzelverzeichnis als /dev/root und /dev/u. Gerätedateien werden im nächsten Abschnitt genauer betrachtet. Einen weiteren Einblick vermittelt das Kommando df, das über freie Blöcke und freie Plätze in den Inode-Tabellen berichtet.

```
% df
/       (/dev/root ):    16060 blocks    4057 i-nodes
/u      (/dev/u    ):    81602 blocks   10619 i-nodes
```

Hier wird die separate Verwaltung der Inodes und der Textblöcke in den beiden Dateisystemen gut sichtbar.

Bleiben wir für die weiteren Erläuterungen vorerst in einem Dateisystem und diskutieren dessen Struktur. Wir betrachten jetzt die beiden Versionen der Akkumulationsprogramme. Eine Version arbeitet mit PASCAL (Abschn. 1.), die andere mit der Sprache C (Abschn. 2.1.). Wir fassen die Dateien für die PASCAL-Version in dem Verzeichnis Paccu zusammen und die Dateien für die C-Version in dem Verzeichnis Caccu. Mit dem Kommando

```
% lc Paccu Caccu
Caccu:
accu        accu.c      accu.inp    accu.res
Paccu:
accu        accu.inp    accu.pas    accu.res    makefile
```

werden sie alle aufgezählt. Da Verzeichnisse wiederum Dateien sind, könnten wir diese beiden Verzeichnisdateien erneut in einem Verzeichnis als Dateien zusammenfassen, das dann genau diese beiden Verzeichnisse Paccu und Caccu als Dateien enthält. Dadurch entsteht eine hierarchische Struktur von Dateinamen. Bild 2.1 zeigt diese Struktur für Paccu.

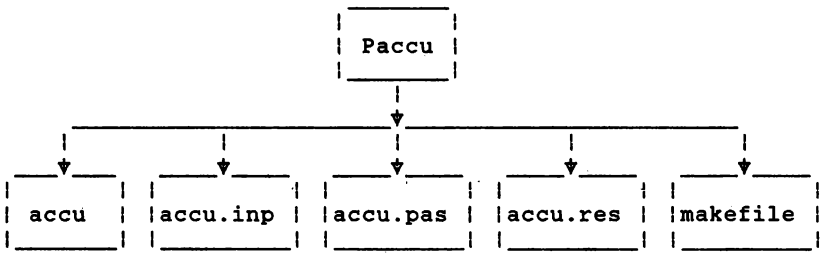



Bild 2.1. Dateiverzeichnis Paccu

Nun könnte man den Spieß umdrehen. Wir tun so, als wären die beiden Programmversionen noch nicht vorhanden. Sie herzustellen ist für das folgende unsere Absicht. Es wäre also ein Dateiverzeichnis für die PASCAL-Version zu schaffen und eines für die C-Version. Einfache Dateien lassen sich wie im Abschnitt 2.1. mit dem Editor ed eingeben. Für das Einrichten von Verzeichnissen benötigt man jedoch ein besonderes Kommando. Das Kommando

```
% mkdir dir1 dir2
```

richtet die Dateien dir1, dir2 usw. als Verzeichnisdateien ein. Für unsere Belange eignet sich das Kommando:

```
% mkdir Paccu Caccu
```

Betrachten wir unsere Beispiele rückblickend, so fanden unsere Manipulationen mit der PASCAL-Version des Abschnitts 1. im Dateiverzeichnis Paccu statt. Die Datei Paccu war dabei Arbeitsverzeichnis. Beim Umgang mit der C-Version war Caccu Arbeitsverzeichnis.

Mit dem Kommando rmdir kann man Dateiverzeichnisse wieder beseitigen. Allerdings löscht das Kommando

```
% rmdir dir1 dir2
```

nur leere Dateiverzeichnisse. Dateinamen in einem mit rmdir zu löschenden Verzeichnis müssen vorher behandelt werden. Rigoros verhält sich in diesem Fall das Kommando rm -r dir. Es löscht die Verzeichnisdatei dir und alle hierarchisch nachfolgenden Einträge.

Bei der Betrachtung mit dem umgedrehten Spieß befinden wir uns in einem Dateiverzeichnis, das nach der Abarbeitung der beiden Kommandos mkdir hierarchisch vor diesen beiden liegt. Mit dem Kommando pwd wird das aktuelle Arbeitsverzeichnis mit allen seinen hierarchisch davorliegenden Namen auf die Standardausgabe geschrieben.

```
% pwd
/u/polze/priv/book/cont/T22
```

Der Schrägstrich trennt die Namen der Dateiverzeichnisse. Als das Manuskript für dieses Buch geschrieben wurde, enthielt T22 die Testbeispiele zum Abschnitt 2.2., in dem wir uns gerade befinden.

Eine Folge von Dateinamen, in der jede Datei als Vorgänger ein Dateiverzeichnis hat, in dem sie enthalten ist, heißt Pfad. Er beginnt in dem zuerst angegebenen Dateiverzeichnis. Steht am Anfang ein Schrägstrich, so beginnt der Pfad an der Hierarchiewurzel der von UNIX verwalteten Dateiverzeichnisse. Solche Namen werden als absolute Pfadnamen bezeichnet. Relative Pfadnamen beginnen in einem beliebigen von der Wurzel / verschiedenen Verzeichnis.

In X/OPEN (1987) sind die Bildungsvorschriften für Pfadnamen in der Form von Syntaxregeln angegeben.

```
pathname    : filename
            | path_prefix filename
            | '/'
path_prefix : rtprefix
            | '/' rtprefix
            | '/'
rtprefix    : dirname '/'
            | rtprefix dirname '/'
```

Grundsymbole in dieser kleinen Grammatik sind filename, dirname und der Schrägstrich / als Trennzeichen. Die Zeichenketten filename und dirname enthalten wenigstens ein und maximal 14 Zeichen, die vom Schrägstrich und vom Nullzeichen verschieden sind. Die Zeichenketten "." und ".." sind also auch als filename oder dirname zulässig. Als dirname werden ausschließlich Verzeichnisdateien bezeichnet. Mit filename können beliebige Dateien als Ziel des Pfades angegeben werden. Andere als die durch diese Grammatik erzeugten Pfadnamen sollen nicht angenommen werden.

Das Kommando lc -R gibt den Inhalt eines Dateiverzeichnisses aus und steigt hierarchisch abwärts, wenn darin erneut Verzeichnisse als Dateien enthalten sind.

```
% lc -R .
Caccu  Paccu
./Caccu:
accu       accu.c    accu.inp  accu.res
./Paccu:
accu       accu.inp  accu.pas  accu.res  makefile
```

Die gleiche Wirkung würde das Kommando

```
% lc -R `pwd`
```

erzielen. Wenn man ein Kommando in <backquote>-Zeichen einschließt, so bedeutet dies seine Ausgabe auf stdout. `pwd` ist daher immer der Name des Arbeitsverzeichnisses, ohne daß man dessen womöglich langen Namen immer hinschreiben muß.

Das Bild 2.2 zeigt diese zweistufige Verzeichnisstruktur der PASCAL- und C-Version unserer vertrauten Programmieraufgabe.

Weil das Betriebssystem UNIX Dateien und Dateiverzeichnisse hierarchisch verwaltet, muß es auch Möglichkeiten geben, in dieser Hierarchie nach oben oder nach unten zu wandern. Dazu gibt es das Kommando cd. Das Kommando

```
% cd Paccu
```

erklärt das Verzeichnis Paccu fortan als Arbeitsverzeichnis. Für die Rückkehr gewinnt die oben mit lc -a sichtbar gewordene zweite Verzeichniseintragung .. Gewicht. Während . das Verzeichnis selbst als Datei bezeichnet, bezeichnet das Vorgängerverzeichnis. Das Kommando

```
% cd
```

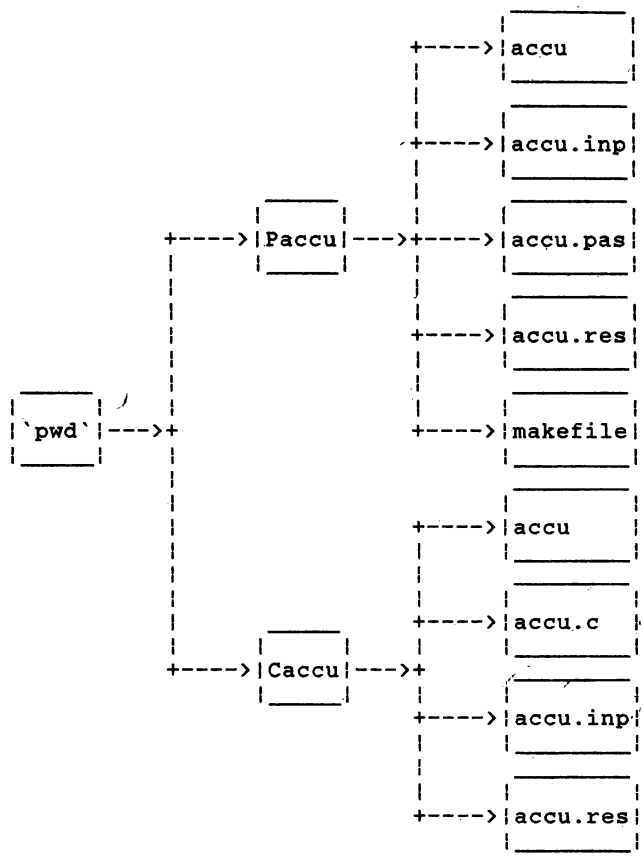



Bild 2.2. Verzeichnisse Paccu Caccu

beendet also den Ausflug in das Verzeichnis Paccu und geht hierarchisch um eine Stufe zurück. Gehen wir nochmals nach Paccu mit dem Kommando:

```
% cd Paccu
```

Jetzt führt uns das Kommando

```
% cd ../Caccu
```

eine Stufe zurück und danach gleich weiter in das Verzeichnis Caccu.

Wenn ein Benutzer an das UNIX-System herantritt, um eine Terminalarbeit zu beginnen, so erhält er einen Knoten aus dem großen UNIX-Verzeichnissystem als Arbeitsverzeichnis zugestellt. Seine bisherige Arbeit kann die Hierarchie an dieser Stelle tief fortgesetzt haben. Wenn er seine Terminalsitzung beginnt, bekommt er dieses Verzeichnis als Arbeitsverzeichnis wieder angeboten. Es soll Heimatverzeichnis heißen.

Ein Aufruf des Kommandos cd ohne Parameter führt ihn aus beliebiger Tiefe auf diese Stufe zurück. Alle Zwischenstufen werden dabei

übergangen. Die Umgebungsvariable HOME hat den Pfadnamen des Heimatverzeichnisses als Wert. In der C-Shell gibt die Kurzform ~ ebenfalls den Dateinamen des Heimatverzeichnisses an.

In der Vielfalt der UNIX-Systeme trifft man Dateisysteme mit ganz unterschiedlich ausgebauten Verzeichnisstrukturen an. Die Standardisierung von UNIX nach X/OPEN (1987) sieht vor, daß einem Nutzer in jedem UNIX-System, das den Anspruch erhebt, dem Standard zu entsprechen, eine Mindestausbaustufe der Dateiverwaltung angeboten wird. Diese Mindestanordnung für die Verzeichnishierarchie ist im Bild 2.3 aufgezeichnet.

| | |
|---|---|
| /bin | Ist Verzeichnisdatei und verweist auf abarbeitbare Fassungen der wichtigsten UNIX-Kommandos. |
| /dev | Ist eine Verzeichnisdatei, in der alle Gerätedateien der UNIX-Konfiguration erfaßt sind. |
| /etc | Enthält die Dateien mit den wichtigsten Systemdaten und Kommandos zu ihrer Verwaltung, beispielsweise das Kommando mount und die Datei passwd mit den Angaben über alle eingetragenen Nutzer. |
| /tmp | Nimmt zeitweise benötigte Dateien auf. Viele UNIX-Kommandos richten zu ihrer Arbeit temporäre Dateien ein, in denen Zwischenresultate abgelegt werden (Editorpuffer, Zwischenkodedateien in Mehrpaßcompilern). |
| /usr/bin | Beinhaltet weitere UNIX-Kommandos, beispielsweise diejenigen des Programmmentwicklungssystems oder zur Textverarbeitung. |
| /usr/tmp | Wird als Platz für temporäre Nutzerdateien empfohlen. |

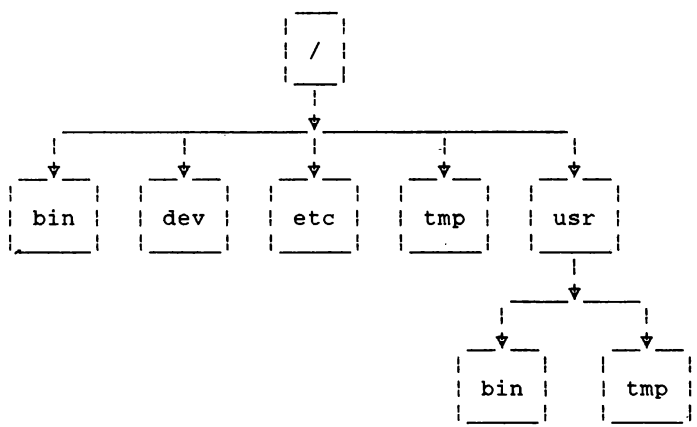



Bild 2.3. Minimale Verzeichnishierarchie

### 2.2.2. Verkettung von Verzeichnissen

Die von UNIX verwalteten Dateiverzeichnisse haben eine streng hierarchisch angeordnete Struktur. Als Graph ist ein Verzeichnis ein Baum. Dateinamen und Verzeichnisnamen, die ja auch Dateinamen sind, bilden die Knoten. Die Relation »Verzeichnis x enthält den Dateinamen y« konstituiert die gerichteten Kanten in diesem Graph.

Dies gilt jedoch nur, wenn man die beiden Verzeichniseinträge und die auch zulässige Dateinamen sind, außer acht

läßt. Nimmt man diese beiden Namen hinzu, so ist der entsprechende Graph nicht mehr kreisfrei, also gewiß kein Baum. Es verbietet sich dann, von einer Hierarchie zu sprechen. Im Bild 2.4, das außerdem die Inodes und die Verbindungen zu den Textblöcken der im Verzeichnis Paccu verwalteten Dateien zeigt, ist der in jedem Verzeichnis vorhandene Kreis deutlich sichtbar.

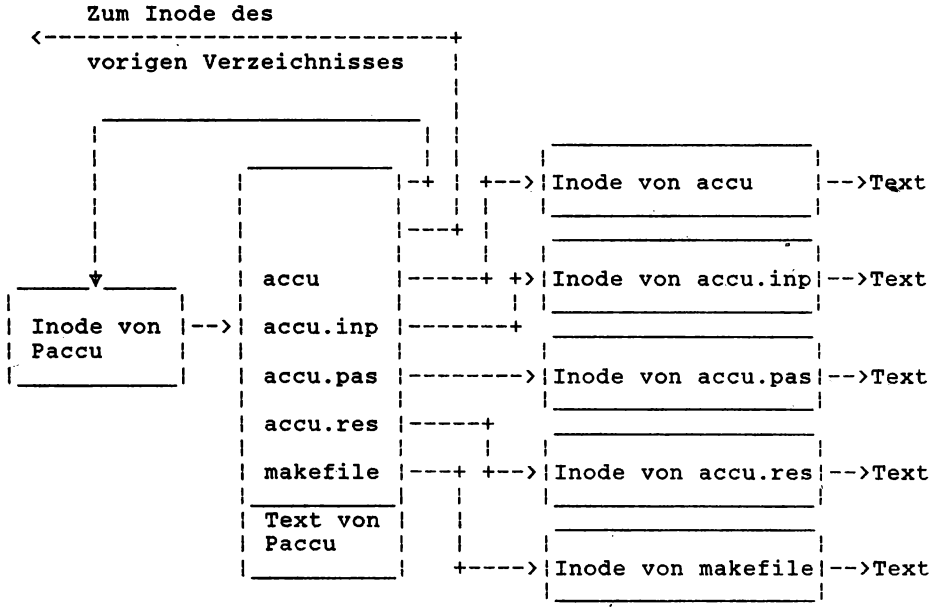



Bild 2.4. Inode-Struktur von Paccu

Auch die folgenden Pfade bewegen sich auf Schleifen in dem Paccu und Caccu vorgelagerten Verzeichnis. Sie zeigen, daß man hier wirklich nicht von einem hierarchischen System sprechen kann.

```
% cd /u/polze/priv/book/cont/T22
% pwd
/u/polze/priv/book/cont/T22
% lc
Caccu  Paccu
% lc ./Caccu
accu      accu.c    accu.inp  accu.res
% lc ./Caccu/..
Caccu  Paccu
% lc ./Caccu/../.
Caccu  Paccu
% lc ./Caccu/.././Paccu/..
Caccu  Paccu
```

Für das Verständis der Dateiverwaltung des UNIX ist es zweckmäßig, das Verzeichnissystem und die unterlegte Gesamtheit von Dateien sorgfältig zu trennen. Dateien treten als Paare oder Tupel aus Inode und Text in Erscheinung. Dateinamen in Verzeichnissen sind Verweise auf die Inodes der Dateien. Bild 2.4 soll diese Verweisstruktur für die Dateien im Verzeichnis Paccu ausdrücken. Das hat den bedeutenden

Vorteil, daß auf eine Datei, also auf ein Inode-Text-Paar, durchaus mehrere Verweise aus dem Verzeichnissystem zeigen können.

Unser Programmbeispiel accu bietet Gelegenheit, diesen Sachverhalt zu demonstrieren. So sind die Testdaten für die beiden Programmversionen in PASCAL oder in C beide Male die gleichen. Es gibt daher überhaupt keinen Grund, die Dateitexte zweimal zu speichern, nur weil die beiden Versionen in unterschiedlichen Arbeitsverzeichnissen aufbereitet werden. Bleiben wir bei der Reihenfolge, wie sie in diesem Buch vorliegt. Zuerst ist die PASCAL-Version behandelt worden. Die Ausgangssituation sei durch folgende Kommandos rekonstruiert:

```
% mkdir Paccu
  ...
% lc Paccu
accu      accu.inp  accu.pas  accu.res  makefile
% mkdir Caccu
% cd Caccu
% ed accu.c
% make accu
```

Jetzt müßten also Testdaten für diese neue Version beschafft werden. Mit dem Kommando ln kann ein Verweis zu einer bestehenden Datei hergestellt werden.

```
% ln ../Paccu/accu.inp accu.inp
```

In der Fassung ln oldfile newfile des Kommandos ln wird ein Verweis auf die bestehende Datei oldfile hergestellt. Dieser Verweis heißt newfile. Nun haben wir auch in dem Verzeichnis Caccu den Namen accu.inp. Dabei ist das dazugehörige Paar Inode-Text nicht neu angelegt worden. Jedoch hat jetzt der Verweiszähler di_nlink in der oben angegebenen C-Struktur den Wert 2 erhalten. Bild 2.5 zeigt die Verhältnisse in den beiden Verzeichnissen Paccu und Caccu. Die gleiche Information läßt sich durch das folgende Kommando gewinnen. Man studiere Inode-Nummern und Verweiszähler.

```
% lc -ialR
total 10
 1104 drwxr-xr-x   4 polze    HUB      144 Apr 11 09:11
  921 drwxr-xr-x  12 polze    HUB     1040 Apr 11 09:12 ..
 1252 drwxr-xr-x   2 polze    HUB      112 Mar 25 14:16 Caccu
 1275 drwxr-xr-x   2 polze    HUB      112 Mar 25 14:16 Paccu
./Caccu:
total 10
 1252 drwxr-xr-x   2 polze    HUB      112 Mar 25 14:16
 1104 drwxr-xr-x   4 polze    HUB      144 Apr 11 09:11 ..
 1265 -rw-r--r--   1 polze    HUB     9536 Mar 25 14:16 accu
 1266 -rw-r--r--   1 polze    HUB      344 Mar 25 14:16 accu.c
 1262 -rw-r--r--   2 polze    HUB       56 Mar 25 14:16 accu.inp
 1263 -rw-r--r--   2 polze    HUB      188 Mar 25 14:16 accu.res
./Paccu:
total 12
 1275 drwxr-xr-x   2 polze    HUB      112 Mar 25 14:16 .
 1104 drwxr-xr-x   4 polze    HUB      144 Apr 11 09:11 ..
 1258 -rw-r--r--   1 polze    HUB    20096 Mar 25 14:16 accu
 1262 -rw-r--r--   2 polze    HUB       56 Mar 25 14:16 accu.inp
 1259 -rw-r--r--   1 polze    HUB      395 Mar 25 14:16 accu.pas
 1263 -rw-r--r--   2 polze    HUB      188 Mar 25 14:16 accu.res
 1268 -rw-r--r--   1 polze    HUB       48 Mar 25 14:16 makefile
```

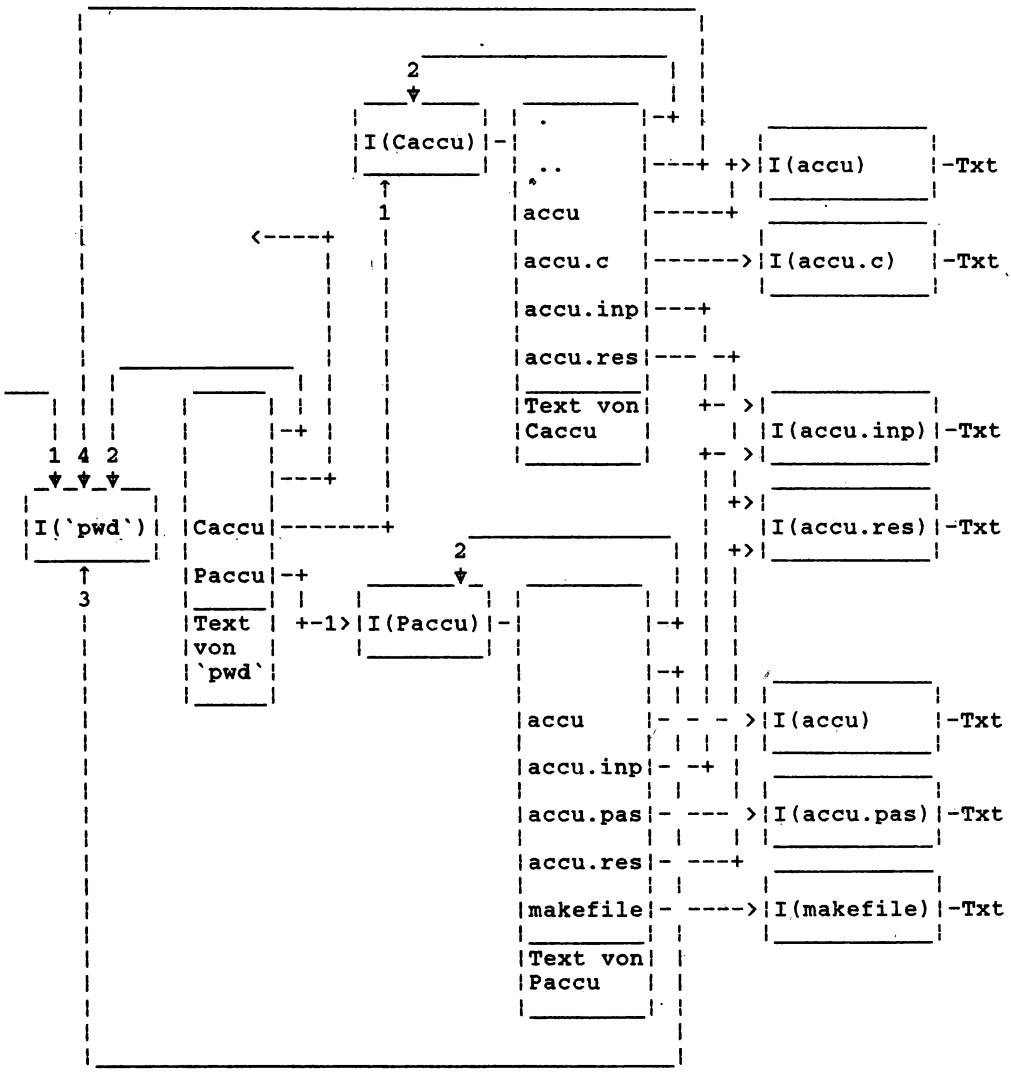



Bild 2.5. Inodes zu Paccu und Caccu

Schließlich sei das Kommando ln vorgestellt. Es hat folgende Aufrufformen:

```
ln [-f] file1 file2
ln [-f] f1 f2 ... fn dir
```

Die erste Form ordnet der vorhandenen Datei, auf die bereits der Name file1 zeigt, den zusätzlichen Verweis file2 zu. Wenn es eine davon verschiedene Datei mit dem Namen file2 bereits gibt, wird diese Datei überschrieben, falls ihre Zugriffsrechte dies erlauben. Wenn die Zugriffsrechte der bestehenden Datei file2 das Schreiben verbieten,

wird die mode-Information von file2 gedruckt und die Antwort des Nutzers abgewartet. Beginnt seine Eingabe mit y, so wird die Aktion von ln auch in diesem Fall vollzogen. Ist die Option -f wirksam, verzichtet ln auf diese Nachfrage.

Die zweite Form des Kommandos trägt die Namen f1 bis fn, die alle auf bestehende Dateien zeigen, auch in das Verzeichnis dir ein.

Es folgt die Tafel 2.1. Sie faßt wichtige Kommandos für die Arbeit in Dateiverzeichnissen zusammen.

Tafel 2.1. Kommandos für Dateien und Verzeichnisse

| | |
|---|---|
| pwd | Arbeitsverzeichnis, absoluter Pfadname nach stdout. |
| mkdir dir | Einrichten von Verzeichnissen dir ... |
| rmdir dir | Streichen der leeren Verzeichnisse dir |
| cd | Neues Arbeitsverzeichnis ist $HOME.<br>Es wird beim Beginn der Terminalarbeit angeboten. |
| cd dir | Neues Arbeitsverzeichnis ist dir. |
| du dir | Gibt die Zahl der belegten Blöcke aller in dir erfaßten Dateien an, einschließlich der Dateien in nachfolgenden Verzeichnissen. |
| rm -r dir | Die Verzeichnisdatei dir wird mit allen ihren hierarchisch nachfolgenden Dateien gestrichen. |
| mv f1 fn dir | Die Verweise f1 ... fn werden ihren bisherigen Verzeichnissen entnommen und nach dir übertragen. |
| cp f1 fn dir | Alle Dateien f1 ... fn werden kopiert. Ihre Namen werden in das Verzeichnis dir aufgenommen. |
| ln f1 fn dir | Die Namen der bestehenden Dateien f1 ... fn werden als zusätzliche Verweise in dir aufgenommen. |
| basename pathname [suffix] | (siehe Regeln für Pfadnamen)<br>Liefert filename [ohne suffix]. |
| dirname pathname | (siehe Regeln für Pfadnamen)<br>Liefert path_prefix ohne '/'. |
| dircmp dir1 dir2 | Vergleicht die Verzeichnisse dir1 und dir2, prüft namensgleiche Dateien auf Textgleichheit. |
| dircmp -d dir1 dir2 | Listet für namensgleiche Dateien mit ungleichem Text die Unterschiede wie das Kommando diff. |

Selbstverständlich können in diesen Kommandos alle Dateinamen als Pfadnamen verstanden werden. Die durch Kopieren neu entstehenden Dateien erhalten die Attribute des Nutzers, der das cp-Kommando ausführt, seine Nutzer- und Gruppennummer sowie Zugriffsrechte, die durch seine umask-Bestimmung vermittelt werden.

Die Zugriffsrechte von Verzeichnisdateien verdienen noch einige weitere Bemerkungen. Sie werden durch die folgende Aufrufform des ls-Kommandos sichtbar.

```
% ls -ld .
drwxr-xr-x  12 polze    HUB          1072 Apr 13 09:41
```

Wieder gehören die rwx-Tripel zum Eigentümer, zu den Mitgliedern der Gruppe und zu den weiteren Nutzern des UNIX-Systems. Häufige Lesezugriffe kommen von Kommandos wie ls oder od. Für die Arbeit von Kommandos wie cp f1 dir oder gar rm -r dir ist die Berechtigung zum Schreiben erforderlich. Die Eintragung x bedeutet für Verzeichnisdateien die Berechtigung zum Suchen nach Dateien, um ihre Abarbeitung zu starten. Lautet zum Beispiel das dritte Tripel --x, so kann kein fremder Benutzer das Verzeichnis besichtigen. Er kann aber abarbeitbare Dateien aus diesem Verzeichnis aufrufen. Lautet das Zugriffstripel r--, so kann ein fremder Nutzer das Verzeichnis ansehen, aber keine Datei starten.

Zugriffsrechte von Verzeichnissen müssen sehr ernst genommen werden. Wir richten ein Verzeichnis a ein und geben fremden Nutzern die Berechtigung zum Schreiben. Anschließend erzeugen wir die Datei b, für die unserer umask-Stellung entsprechend fremde Nutzer keine Schreiberlaubnis erhalten. Das ls-Kommando zeigt die Zugriffsrechtslage für die beiden Dateien a und b.

```
% mkdir a
% cd a
% chmod o=rwx
% touch b
% ls -l
total 4
drwxr-xrwx   2 polze    HUB           48 Apr 13 10:44 .
drwxr-xr-x   5 polze    HUB          192 Apr 13 10:44 ..
-rw-r--r--   1 polze    HUB            0 Apr 13 10:44 b
% logout
```

Nun kommt ein fremder Benutzer und meldet sich als guest an. Er ist wirklich fremd. Mit dem Kommando logname identifiziert er sich demonstrativ. Das ls-Kommando zeigt, er gehört sogar einer anderen Gruppe an. Er besichtigt unser Verzeichnis a und stellt fest, er darf darin schreiben. Mit dem Kommnando rm -r, so weiß er, kann er das mit a beginnende Verzeichnissystem streichen. Das System weist ihn zwar auf die Verletzung der Schreibrechte der Datei b hin, doch er antwortet mit y. Jetzt ist alles geschehen. Hätte er gleich das Kommando rm -rf verwendet, wäre er nicht einmal gefragt worden. Die folgende Ausschrift »no permission« bezieht sich schon auf das vorgelagerte Verzeichnis, in unserem Fall auf die Datei T22, in der wir glücklicherweise die Zugriffsrechte sorgfältiger festgelegt haben. Das folgende ls-Kommando zeigt dem Fremdling, unsere Datei b ist wirklich verschwunden.

```
xenix286!login: guest
   ...
$ logname
guest
$ ls -ld .
drwxr-xr-x   3 guest    group        160 Feb 15 15:16
$ ls -al /u/polze/priv/book/cont/T22/a
total 4
drwxr-xrwx   2 polze    HUB           48 Apr 13 10:44
drwxr-xr-x   5 polze    HUB          192 Apr 13 10:44 ..
-rw-r--r--   1 polze    HUB            0 Apr 13 10:44 b
$ rm -r /u/polze/priv/book/cont/T22/a
/u/polze/priv/book/cont/T22/a/b: 644 mode ? y
rmdir: /u/polze/priv/book/cont/T22/a: no permission
```

```
$ ls -al /u/polze/priv/book/cont/T22/a
drwxr-xrwx   2 polze    HUB           48 Apr 13 10:46
drwxr-xr-x   5 polze    HUB          192 Apr 13 10:44
$ Ctrl-d
```

Nun können wir nur noch den angerichteten Schaden feststellen, wenn wir das nächstemal das Verzeichnis a besichtigen.

```
xenix286!login: polze
Password: ........
  ...
% ls -al a
drwxr-xrwx   2 polze    HUB           48 Apr 13 10:46
drwxr-xr-x   5 polze    HUB          192 Apr 13 10:44
%
```

Der Fremdling hat seine Spuren hinterlassen. Die »schreibgeschützte« Datei a/b ist tatsächlich veschwunden. Die Ermahnung zur Vorsicht ist wirklich berechtigt.

## 2.3. Geräte als spezielle Dateien

### 2.3.1. Geräteverzeichnis /dev

UNIX betrachtet periphere Geräte als Dateien, als spezielle Dateien. Mehrere Besonderheiten sind ihnen eigen. Sie werden in dem Dateiverzeichnis /dev zusammengefaßt. Von den bisherigen Dateien unterscheiden sie sich darin, daß die jeweiligen Bytefolgen nicht als Text gespeichert vorliegen, sondern auf Anforderung erzeugt oder verarbeitet werden. Eine Bytefolge wird erzeugt, indem sie vom peripheren Gerät gelesen wird. Sie wird verarbeitet, indem sie auf ein peripheres Gerät ausgegeben wird. Eine wiederholte Verwendung einer /dev-Datei kann je nach dem Charakter des peripheren Gerätes ganz unterschiedliche Texte als Bytefolgen liefern.

Im vorigen Abschnitt traten zwei Dateisysteme in Erscheinung. Sie befanden sich nach der Auskunft des Kommandos mount auf den beiden Geräten /dev/root und /dev/u, Grund genug für uns, diese beiden Gerätedateien näher zu betrachten. Mit unterschiedlichen Optionen des ls-Kommandos ergibt sich

```
% ls /dev/{root,u}
/dev/root
/dev/u
% ls -il /dev/{root,u}
    4 brw------- 1 sysinfo sysinfo 1, 40 Mar 28 13:07 /dev/root
    8 brw------- 1 sysinfo sysinfo 1, 42 Mar 28 13:11 /dev/u
```

Beide Dateien gehören dem systeminternen Eigentümer sysinfo, der wiederum der Gruppe mit dem gleichen Namen angehört. Wir verzichten im folgenden auf Eigentümer- und Gruppennamen und lassen uns nur den Zeitpunkt des letzten Zugriffs ausgeben. Das gelingt mit dem Kommando

```
% ls -iogu /dev/{root,u}
    4 brw-------   1   1, 40 Mar 28 13:07 /dev/root
    8 brw-------   1   1, 42 Apr 20 11:15 /dev/u
```

Gegenüber dem bisher Bekannten fallen zwei Unterschiede auf: Erstens erscheint auf der Position des Dateimodus ein b (bisher gab es - für einfache Dateien und d für Verzeichnisdateien), und zweitens stehen am Platz der bisherigen Längenangabe zwei Zahlen. Die übrigen Zahlen sind die Inode-Nummer und die Anzahl der Verweise auf die Gerätedatei.

Der Dateimodus b kennzeichnet gepuffert arbeitende Geräte mit einer Blockgröße von 512 Byte oder bei Plattengeräten von 1024 Byte. Davon zu unterscheiden sind zeichenweise arbeitende Geräte. Ihr Dateimodus heißt c. Wenn es für periphere Geräte sinnvoll ist, sind beide Dateiarten als Geräte im Verzeichnis /dev vertreten. Unsere beiden Gerätedateien repräsentieren Plattenspeicher. Dort ist sowohl der blockweise als auch der zeichenweise Transfer möglich. Wir stellen fest, es sind beide Varianten vorhanden.

```
% ls -iog /dev/*{root,u}
    4 brw-------   1   1, 40 Mar 28 13:07 /dev/root
    5 crw-------   1   1, 40 Oct 19  1987 /dev/rroot
    8 brw-------   1   1, 42 Mar 28 13:11 /dev/u
    9 crw-------   1   1, 42 Oct 19  1987 /dev/ru
```

Die beiden Zahlen an der Stelle der Größe einfacher Dateien sind die sogenannte »major device number« und die »minor device number«. Mit der ersten Zahl, der major device number, werden Klassen von peripheren Geräten gebildet. Mit der zweiten Zahl, der minor device number, werden mehrere Exemplare von Geräten der gleichen Klasse unterschieden. Es ist ersichtlich, und das gilt für Plattendateien generell, daß die beiden blockweise und zeichenweise arbeitenden Gerätedateien /dev/root und /dev/u zwar verschiedene Inodes haben, aber in der minor device number nicht unterschieden werden. Die gewählte Numerierung ist hardwareabhängig. Von System zu System sind unterschiedliche Numerierungsschemata anzutreffen. XENIX286 verwendet folgende Numerierung:

```
0   Multiscreening-Instanzen des Bedienterminals
1   Festplattenbereiche
2   Diskettenstationen
3   Bedienterminal
4   Hauptspeicher
5   Terminals (serial ports)
6   Drucker (parallel ports)
```

Folgende Plattendateien sind vorhanden:

```
% ls -iogu /dev/{hd0?,root,swap,u}
   91 brw-------   1   1,  0 Dec 11 12:57 /dev/hd00
   92 brw-------   1   1, 15 Dec 11 12:57 /dev/hd01
   93 brw-------   1   1, 23 Oct 19  1987 /dev/hd02
   94 brw-------   1   1, 31 Oct 19  1987 /dev/hd03
   95 brw-------   1   1, 39 Dec 11 12:57 /dev/hd04
   96 brw-------   1   1, 47 Dec 11 12:57 /dev/hd0a
  164 brw-r-----   1   1, 55 Apr 20 13:30 /dev/hd0d
    4 brw-------   1   1, 40 Mar 28 13:07 /dev/root
    6 brw-------   1   1, 41 Oct 19  1987 /dev/swap
    8 brw-------   1   1, 42 Apr 20 11:15 /dev/u
```

Plattenspeicher werden in vier Bereiche eingeteilt. Die Größe der Bereiche wird bei der Installation des UNIX-Systems festgelegt. Die ganze Platte ist als Gerät hd00 erreichbar. Die Geräte hd01 bis hd04 sind die vier Bereiche. Ein Bereich wird als aktiv ausgewiesen. Das darin befindliche Ladeprogramm startet das Betriebssystem beim Einschalten des Rechners. Das Gerät hd0a gibt den aktiven Plattenbereich. Schließlich kennzeichnet das Gerät hd0d jenen der vier Plattenbereiche, der bei der Installation für das Betriebssystem MS-DOS frei gehalten worden ist. Zusätzlich zu den hd-Geräten befinden sich Plattendateien für das Dateisystem /dev/root sowie für das System der Nutzerdateien /dev/u in der Liste. Das Gerät /dev/swap nimmt Auslagerungen aus dem Hauptspeicher auf. Es bildet die Basis für die Realisierung der konkurrenten Prozesse in der UNIX-Arbeit.

Gewaltig ist die Liste der Geräte, die den verschiedenen Diskettenformaten zugeordnet sind. Hier ein Auszug der XENIX-Floppy-Dateien:

```
% ls -iogu /dev/{fd*,rfd0}
   90 brw-rw-rw-   3   2,  4 Jan  5 15:54 /dev/fd0
   90 brw-rw-rw-   3   2,  4 Jan  5 15:54 /dev/fd048
  150 brw-rw-rw-   1   2, 12 Dec 17 08:34 /dev/fd048ds8
   90 brw-rw-rw-   3   2,  4 Jan  5 15:54 /dev/fd048ds9
  149 brw-rw-rw-   1   2,  8 Oct 19  1987 /dev/fd048ss8
  151 brw-rw-rw-   1   2,  0 Oct 19  1987 /dev/fd048ss9
  161 brw-rw-rw-   2   2, 52 Jan  8 13:31 /dev/fd096
  161 brw-rw-rw-   2   2, 52 Jan  8 13:31 /dev/fd096ds15
  160 brw-rw-rw-   1   2, 36 Mar 10 10:42 /dev/fd096ds9
  155 brw-rw-rw-   3   2,  5 Jan 27 14:05 /dev/fd1
  155 brw-rw-rw-   3   2,  5 Jan 27 14:05 /dev/fd148
  153 brw-rw-rw-   1   2, 13 Dec 17 08:32 /dev/fd148ds8
  155 brw-rw-rw-   3   2,  5 Jan 27 14:05 /dev/fd148ds9
  152 brw-rw-rw-   1   2,  9 Dec 28 08:50 /dev/fd148ss8
  154 brw-rw-rw-   1   2,  1 Nov 18 08:37 /dev/fd148ss9
  163 brw-rw-rw-   2   2, 53 Oct 19  1987 /dev/fd196
  163 brw-rw-rw-   2   2, 53 Oct 19  1987 /dev/fd196ds15
  162 brw-rw-rw-   1   2, 37 oct 29 09:34 /dev/fd196ds9
   89 crw-rw-rw-   3   2,  4 Apr 15 07:12 /dev/rfd0
```

Alle Floppy-Dateien sind auch in der rfd*-Fassung im Verzeichnis /dev vorhanden. Stellvertretend für alle wurde /dev/rfd0 in die obige Liste aufgenommen. Des weiteren bedeuten:

```
fd0     Laufwerk 0
fd1     Laufwerk 1
48 tpi (tracks per inch) gibt 40 Spuren bei 5¼"-Disketten
96 tpi (tracks per inch) gibt 80 Spuren bei 5¼"-Disketten
ds      (double sided) beidseitig beschreibbar
ss      (single sided)  einseitig beschreibbar
 8 Blöcke pro Spur, entspricht  8*512=4096 Byte pro Spur
 9 Blöcke pro Spur, entspricht  9*512=4608 Byte pro Spur
15 Blöcke pro Spur, entspricht 15*512=7680 Byte pro Spur
```

Gebräuchliche Diskettenformate sind:

```
48ds9     mit 512*2*40*9  =  360 KByte
96ds9     mit 512*2*80*9  =  720 KByte
96ds15    mit 512*2*80*15 = 1200 KByte
```

Hier geben die Inode-Nummern und somit auch die minor device numbers weitere Zusammenhänge preis. So zeigen bei der vorliegenden Verkettung die Namen fd0, fd048 und fd048ds9 auf ein und dieselbe Datei, ebenso fd096 und fd096ds15. Entsprechendes gilt für Diskettendateien im Laufwerk 1.

Die Abhängigkeiten von der Systemgestaltung und schließlich von der Hardware werden sichtbar, wenn wir andere UNIX-Systeme betrachten. Nachfolgend ist die Liste der Gerätedateien für Disketten aus dem UNIX-System WEGA des Rechners P8000 abgedruckt. Der Eigentümer der Dateien heißt wega. Der Gruppenname ist system.

```
20 brw-rw-rw- 2 wega system 1, 96 Jul 26 08:00 /dev/fd0
26 brw-rw-rw- 1 wega system 1,128 Jul 26 08:00 /dev/fd048ds9
28 brw-rw-rw- 1 wega system 1,144 Jul 26 08:00 /dev/fd048ss9
24 brw-rw-rw- 1 wega system 1, 80 Jul 26 08:00 /dev/fd096ds16
20 brw-rw-rw- 2 wega system 1, 96 Jul 26 08:00 /dev/fd096ds9
22 brw-rw-rw- 1 wega system 1,112 Jul 26 08:00 /dev/fd096ss9
21 brw-rw-rw- 2 wega system 1, 97 Jul 26 08:00 /dev/fd1
27 brw-rw-rw- 1 wega system 1,129 Jul 26 08:00 /dev/fd148ds9
                ... usw. ...
30 crw-rw-rw- 2 wega system 1, 96 Jul 26 08:00 /dev/rfd0
                ... usw. ...
31 crw-rw-rw- 4 wega system 1, 97 Sep 15 16:42 /dev/tardev
```

Wieder anders erscheint die Namensgebung beim System VENIX. Es folgt eine Auswahl von Gerätedateien für Diskettenformate in diesem Betriebssystem. Hier ist root der Name des Eigentümers der Dateien. Der Gruppenname ist nicht angegeben.

```
148 1 brwxrw-rw- 1 root 0, 24 Jan 14 11:15 /dev/f0
186 1 brw-rw-rw- 1 root 0, 16 Sep 28 11:47 /dev/f0.40.d.8
197 1 brw-rw-rw- 2 root 0, 24 Jan 12 04:09 /dev/f0.40.d.9
 20 0 brw-rw-rw- 1 root 0,  0 Jan 13 16:07 /dev/f0.40.s.8
198 1 brw-rw-rw- 1 root 0,  8 Oct 30 05:17 /dev/f0.40.s.9
197 1 brw-rw-rw- 2 root 0, 24 Jan 12 04:09 /dev/f0.9
184 1 brwxrw-rw- 1 root 0, 57 Jan 15 08:37 /dev/f1
                  usw. ...
```

Hervorgehoben werden sollen einige Gerätedateien der Klassen 3 und 4 (major device numbers 3 und 4).

```
% ls -iogu /dev/{console,systty,tty}
   85 crw-rw-rw-   2   3,  1 Apr 21 06:57 /dev/console
   85 crw-rw-rw-   2   3,  1 Apr 21 06:57 /dev/systty
  108 crw-rw-rw-   1   3,  0 Apr 21 07:41 /dev/tty

% ls -iogu /dev/{mem,kmem,null}
  107 crw-------   1   4,  1 Apr 21 08:08 /dev/kmem
  106 crw-------   1   4,  0 Apr 21 08:08 /dev/mem
   86 crw-rw-rw-   1   4,  2 Jan 14 09:27 /dev/null
```

Die ersten beiden Geräte gehören zum Bedienterminal des Rechners. Die Namen console und systty verweisen auf die gleiche Gerätedatei. Das Gerät /dev/tty ist das Terminal, an dem der Nutzer arbeitet. Die beiden mem-Geräte beziehen sich auf den Hauptspeicher. Bemerkenswert

ist das Gerät /dev/null. In der Ausgabe vernichtet es alle Bytes einer Datei; bei der Eingabe liefert dieses Gerät eine leere Datei. Dies ist wichtig und nützlich, wenn Kommandoprozessoren Arbeiten in den Hintergrund senden und bei der Umlenkung von Dateien erzeugte Bytes der Standardausgaben weggeworfen werden sollen.

Weitere Terminaldateien stehen in der folgenden Liste:

```
115 crw--w--w-   1   0,  0 Apr 20 17:37 /dev/tty01
119 crw--w--w-   1   0,  1 Apr 20 11:15 /dev/tty02

 ..., usw.

128 crw--w--w-   1   0, 10 Nov 27 09:58 /dev/tty11
129 crw--w--w-   1   0, 11 Nov 27 09:58 /dev/tty12
```

Die Geräteklasse 0 enthält die bei der Installation des Systems vorgesehenen Multiscreening-Geräte. Es steht in der Verfügung des Systemadministrators, wie viele der Geräte dieser Art zugeschaltet werden können. Die Zahl der Gerätedateien steigt erheblich; auch weitere Geräteklassen kommen hinzu, wenn etwa fortgeschrittenere Benutzeroberflächen installiert werden. So führt beispielsweise die Aufnahme des MultiView-Pakets, das unter anderem eine Verallgemeinerung des Multiscreening von der Bedienkonsole auf beliebige Terminals darstellt, zu einer beträchtlichen Erweiterung des /dev-Verzeichnisses.

Die Geräteklasse 5 erfaßt die seriell arbeitenden Anschlüsse. Normalerweise stehen zwei serielle Ports zur Verfügung. Beim seriellen Direktanschluß haben sie die Namen tty[12]a, bei einem Anschluß über ein Modem entsprechend tty[12]A.

```
140 crw-rw-rw-   1   5,128 Oct 19  1987 /dev/tty1A
138 crw--w--w-   1   5,  0 Apr 20 16:30 /dev/tty1a
141 crw-rw-rw-   1   5,136 Oct 19  1987 /dev/tty2A
139 crw--w--w-   1   5,  8 Apr 20 17:40 /dev/tty2a
```

Stehen Erweiterungskarten zur Verfügung, so können die Ports 2x, 4x oder 8x aufgeteilt werden. Die dazugehörigen Gerätenamen sind entsprechend ohne oder mit Modem die folgenden:

```
tty[12][ab]          tty[12][AB]
tty[12][a-d]         tty[12][A-D]
tty[12][a-h]         tty[12][A-H]
```

Parallele Ports werden in der Geräteklasse 6 zusammengefaßt. Im Verzeichnis /dev treten solche Geräte beispielsweise mit den Namen /dev/lp[012] auf.

Die Wahl der Numerierung in einer Klasse, also die Vergabe der minor device numbers, folgt stark systemabhängigen, hardewareabhängigen oder vertriebsabhängigen Gesichtspunkten. Als ein Beispiel sei die Vergabe der minor device numbers für die Diskettendateien mitgeteilt, so wie sie in XENIX286 berechnet werden. (Bit 0 steht rechtsbündig.)

```
Bit 1, Bit0: Laufwerksnummer n, 0≤n≤3
       Bit2: einseitig=0, beidseitig=1
Bit 4, Bit3: 00  9 Blöcke je Spur
             01  8 Blöcke je Spur
             10 15 Blöcke je Spur
       Bit5: 48tpi=0, 96tpi=1
```

Zum Beispiel ergibt sich für /dev/fd148ds9 der Wert 000101=5 und für /dev/fd096ds9 der Wert 100100=36.

### 2.3.2. Separate Dateisysteme

UNIX ist darauf eingestellt, daß zu den vorhandenen Dateisystemen weitere hinzugefügt werden können. Ein System von Dateien umfaßt dabei jeweils eine eigene, von den anderen Dateisystemen unabhängige Verwaltung von Inodes, Textblöcken, Freieinträgen der Inodeliste und von Freispeicherbereichen für Textblöcke.

Wir legen ein zusätzliches Dateisystem an. Eine Diskette im Laufwerk 1 dient als Speichermedium. Wir verwenden eine Standarddiskette mit einem Fassungsvermögen von 360 KByte.

Da es sich wirklich um den Neuaufbau eines Dateisystems handeln soll, muß diese Diskette zuerst formatiert werden. Dazu dient das Kommando format. Die Anlage eines neuen Dateisystems erledigt das Kommando mkfs. Nach der Arbeit von mkfs ist die Diskette für die Aufnahme eines Dateisystems hergerichtet. Für UNIX wird dieses Speichermedium erreichbar, nachdem es in die Liste der verwalteten Dateisysteme aufgenommen wurde. Dazu muß das Kommando mount bemüht werden. Gegenläufig dazu dient das Kommando umount dazu, ein Dateisystem aus der UNIX-Dateiverwaltung auszuschließen. Der Aufruf des Kommandos mount ohne Parameter oder der Aufruf des Kommandos df zeigt, daß die beabsichtigten Erweiterungen wirklich vollzogen worden sind. Die Kommandos mkfs, mount und umount befinden sich im Verzeichnis /etc. Übrigens sind die beiden Kommandos mount und umount mit dem Zugriffsmodus s ausgestattet. Sie werden unter der Eigentümerschaft von root abgearbeitet, wer immer sie startet. Das zeigt das Kommando:

```
% ls -l /etc/{mount,umount}
-rws--x--x   1 root     root        15476 Oct 18  1987 /etc/mount
-rws--x--x   1 root     root        10736 Oct 18  1987 /etc/umount
```

Mit einem Terminalskript werden die einzelnen Schritte demonstriert. Am Anfang des Ablaufs werden folgende Kommandos aufgerufen:

```
Script started on Thu May  5 12:06:56 1988
Neue C-Shell
% pwd
/u/polze/priv/book/cont/T23
% set path=(. ~/bin /bin /usr/bin /etc)
% format /dev/rfd1
Insert floppy in drive; hit return when ready
formatting /dev/rfd1 ...
  track 00  head 0
    ... usw. ...
  track 39  head 1
done
```

```
% mkfs /dev/fd1 360 1 9
isize = 80
m/n = 1 9
% mkdir ExDisk
% mount /dev/fd1 `pwd`/ExDisk
% mount
/dev/root on / read/write on Thu May  5 08:18:12 1988
/dev/fd1 on /u/polze/priv/book/cont/T23/ExDisk read/write on Thu
/dev/u on /u read/write on Thu May  5 08:18:14 1988
% df
/       (/dev/root ):    16636 blocks    4040 i-nodes
/u/polze/priv/book/cont/T23/ExDisk(/dev/fd1): 704 blocks 78 i-nds
/u      (/dev/u    ):    79592 blocks   10482 i-nodes
```

Die Kommandos format, mkfs, mount und auch das später noch aufzurufende Kommando umount haben sämtlich die Gerätedateien als Parameter, auf die sich ihre Wirkungen beziehen sollen. Natürlich ergibt sich in unserem Fall eine weitere Flexibilität, indem durchaus mehrere Disketten als auswechselbare externe Speichermedien für das Laufwerk 1 präpariert werden können. Für den Dateitransfer in andere UNIX-Systeme oder in Fragen der Portabilität ist die Einrichtung separater Dateisysteme jedoch nicht geeignet.

Das Kommando format verlangt überdies die zeichenweise arbeitende Variante rfd1 dieser Gerätedatei. Ohne weitere Optionen führt format einen Dialog mit dem Nutzer »insert floppy ...«. Zugleich berichtet es über den Formatiervorgang in einer Zeile der Form:

```
track # head #
```

Auf den Positionen # wechseln Kopfnummern 0 und 1 bzw. Spurnummern 00 bis 39. Ruft man das Kommando format mit der Option -fq auf, so formatiert es die Diskette schweigend ohne jede Äußerungen am Terminal.

Das Kommando mkfs verwendet weitere Parameter. Diese sind systemspezifisch und müssen der gültigen Systemdokumentation entnommen werden. Im Betriebssystem XENIX286 richtet das Kommando

```
% mkfs geraet 360 1 9
```

ein Dateisystem auf einer 360-K-Diskette ein. Je Spur werden 9 Datenblöcke zu 512 Byte und eine Lücke angeordnet. Nach den Listingangaben von %mount (ohne Parameter) oder von %df besitzt das neue Dateisystem 704 Blöcke und 78 Inodes. Man sollte ein kleines Programm schreiben, das die Diskette ausschöpft. Was passiert, wenn die Grenze der 704 Blöcke erreicht wird? Entsprechend erzeugt

```
% mkfs geraet 1200 1 15
```

ein Dateisystem auf einer 1200-K-Diskette.

Das Kommando mount benötigt ein Verzeichnis, das die Dateien im neugebildeten oder neu montierten Dateisystem zugänglich macht. Dieses Verzeichnis sollte bei der Montage des Dateisystems leer sein. Bei der Demontage des Dateisystems durch umount wird es wieder leer

geräumt. Das Verzeichnis gilt für die Zeitperiode, in der das Dateisystem montiert ist. In voneinander getrennten Montageperioden können durchaus verschiedene Verzeichnisse genutzt werden, wie dies am Ende des Terminalskripts sichtbar wird. Im Terminalskript berichtet mount ohne Parameter über alle augenblicklich montierten Dateisysteme.

Der folgende Abschnitt in dem Terminalskript demonstriert einige Arbeitsabläufe mit dem neu eingerichteten, separaten Dateisystem. Dateien werden in das Dateisystem hineinkopiert und anschließend herausgeholt. Das bietet eine Gelegenheit, zugleich zwei Verallgemeinerungen des Kommandos cp vorzuführen. Bemerkt sei, daß aus der Anlage getrennter Dateisysteme - mit eigener Inode- und Textspeicherverwaltung - die Konsequenz erwächst, daß hierbei wirklich kopiert werden muß, Texte also dupliziert werden müssen. Die Wirkungsweise von mv oder ln, an die man hier auch denken könnte, erfordert eine gemeinsame Inodeverwaltung für die Dateien, die mit diesen Kommandos verknüpft werden sollen. Als Verallgemeinerungen von cp werden die beiden Kommandos copy und cpio benutzt. Für die Unterstützung der Arbeit von cpio wird zusätzlich das Kommando find eingesetzt.

```
% copy -rv ../T22 ExDisk
examine directory ../T22/Paccu
copy file ../T22/Paccu/accu
copy file ../T22/Paccu/accu.inp
copy file ../T22/Paccu/accu.pas
copy file ../T22/Paccu/accu.res
copy file ../T22/Paccu/makefile
examine directory ../T22/Caccu
copy file ../T22/Caccu/accu
copy file ../T22/Caccu/accu.c
copy file ../T22/Caccu/accu.inp
copy file ../T22/Caccu/accu.res
% mkdir Newdir
% cd ExDisk
% find . -depth -print   cpio -pdv ../Newdir
10 blocks
../Newdir/Paccu/accu
../Newdir/Paccu/accu.inp
../Newdir/Paccu/accu.pas
../Newdir/Paccu/accu.res
../Newdir/Paccu/makefile
../Newdir/Caccu/accu
../Newdir/Caccu/accu.c
../Newdir/Caccu/accu.inp
../Newdir/Caccu/accu.res
% cd ..
% cmp {../T22,ExDisk}/Paccu/makefile; echo $status
0
% cmp {ExDisk,Newdir}/Paccu/makefile; echo $status
0
% cat Newdir/Paccu/makefile
.SUFFIXES: .pas
.pas:
    pascal $@
    -rm $*.c $*.o
```

```
% umount /dev/fd1
% mount
/dev/root on / read/write on Thu May  5 08:18:12 1988
/dev/u on /u read/write on Thu May  5 08:18:14 1988
%
```

Die Kommandos copy oder cpio gestatten es, Dateien einer Verzeichnishierarchie geschlossen zu transportieren. Wir strapazieren wieder unsere beiden Verzeichnisse Paccu und Caccu, die im Verzeichnis T22 zusammengefaßt sind. Gegenwärtig befinden wir uns im Arbeitsverzeichnis T23 entsprechend der Abschnittsnumerierung in diesem Buch. Das Kommando

```
% copy -rv ../T22
```

kopiert rekursiv (Option -r) mit einem Protokoll (Option -v) alle in T22 erfaßten Dateien in das Verzeichnis ExDisk, das ja das separate Dateisystem verwalten sollte.

Das Kommando copy gibt es nicht in allen UNIX-Systemen. Demgegenüber gehören die Kommandos find und cpio der anschließend angestoßenen zweiten Kopieraktion zum X/OPEN-Standard [XVS87].

Für die zweite Kopierdemonstration wird zuerst ExDisk zum neuen Arbeitsverzeichnis erklärt. Das Kommando

```
% find . -depth -print
```

listet rekursiv (Angabe -depth) die Pfadnamen der im Arbeitsverzeichnis erfaßten Dateien und gibt sie auf die Standardausgabe (Angabe -print). Das Kommando

```
% cpio -pdv ../Newdir
```

reicht Pfadnamen und zugehörige Dateien weiter (Option -p) und transportiert sie mit Protokoll (Option -v) in ein Zielverzeichnis. Falls dazu weitere Verzeichnisse notwendig sind, werden diese von cpio angelegt (Option -d).

Anschließend wird wieder das Abschnittsverzeichnis T23 zum Arbeitsverzeichnis erklärt. Die beiden Verzeichnisse Newdir und ExDisk sind ihm nachgeordnet. ExDisk gehört zum Dateisystem auf der Diskette. Die folgenden Aufrufe von cmp vergleichen die als Beispiel ausgewählte Datei Paccu/makefile in allen drei Verzeichnissytemen, im originären System ../T22 mit ExDisk und in ExDisk mit Newdir. Das Finalprodukt des mehrmaligen Kopierens wird mit cat angezeigt. Die Datei ist mit dem Original aus Abschnitt 1. natürlich identisch. Dies soll die Richtigkeit unserer Kopiermanipulationen belegen.

Am Schluß des Terminalskripts folgt die Demontage des Dateisystems auf dem Gerät /dev/fd1 sowie die nochmalige Montage und Demontage mit einem anderen Dateiverzeichnis. Jetzt könnte die Diskette aus dem Laufwerk genommen werden. Andere Arbeiten mit dem Laufwerk 1 könnten sich anschließen. Nachdem die bisher präparierte Diskette wieder in das Laufwerk 1 eingelegt wurde, berichtet das Terminalskript über die weiteren Aktivitäten. Das Kommando

```
% mount spec dir r
```

montiert das auf dem Gerät spec residierende Dateisystem mit dem Verzeichnis dir in der Zugriffsform r. Dann sind nur Lesezugriffe erlaubt. Der erneute parameterlose Aufruf von mount zeigt diese Situation.

```
% mkdir ~/Ed
% mount /dev/fd1 ~/Ed r
% mount
/dev/root on / read/write on Thu May 5 08:18:12 1988
/dev/fd1 on /u/polze/Ed read only on Thu May 5 12:17:56 1988
/dev/u on /u read/write on Thu May 5 08:18:14 1988
% cmp {~/Ed,Newdir}/Paccu/makefile; echo $status
0
% ls -l ExDisk
total 0
% umount /dev/fd1
% ls -l ~/Ed
total 0
% rmdir ~/Ed
% rmdir ExDisk
% rm -r Newdir
%
script done on Thu May 5 12:21:35 1988
```

Nach der Demontage sind die beiden Verzeichnisse ExDisk und ~/Ed wirklich leer. Sonst würde rmdir protestieren. Mit dem Kommando

```
% rm -fr Newdir
```

werden die Dateiduplikate im ursprünglichen Dateisystem /u/polze/... beseitigt.

Es folgen Kurzbeschreibungen der Kommandos copy, cpio und find mit einigen ausgewählten Optionen. Das Kommando

```
copy opt s1 sn dest
```

kopiert Dateien aus den Quellensembles s1 ... sn in das Zielsystem dest. Als Quellen können einzelne Dateien, auch Verzeichnisdateien, angegeben werden. Ein Verzeichnis erscheint als Zielangabe zweckmäßig. In Parameterkonstellationen, die in den Anwendungsbereich des Kommandos cp fallen, entartet copy zu cp. Einige Optionen werden hervorgehoben.

-v Während des Kopierens erscheint der Dateiname auf dem Terminal.

-r Dateiverzeichnisse als Quellen werden rekursiv durchlaufen.

-o Eigentümer und Gruppenname des Originals bleiben erhalten. Sonst werden Name und Gruppenname desjenigen, der copy aufruft, als Attribute der Kopie genommen.

-m Das Zeitpunktattribut der letzten Modifikation bleibt erhalten. Sonst wird der Kopierzeitpunkt neues Attribut.

-l Wo immer möglich, werden Link-Verbindungen hergestellt, anstatt die Datei zu duplizieren. Dieses Attribut ist ungeeignet, wenn zwischen verschiedenen Dateisystemen kopiert werden soll.

Das Kommando cpio hat die Aufrufformen:

```
cpio -o [opt]
cpio -i [opt][patt]
cpio -p [opt] dir
```

Das Kommando cpio -o liest Pfadnamen von der Standardeingabe stdin und kopiert die so erreichten Dateien zusammen mit den Pfadnamen und mit den Statusinformationen des Inode zur Standardausgabe stdout.

Das Kommando cpio -i liest einen durch cpio -o erzeugten Dateistrom und entnimmt ihm jene Dateien mit den dazugehörigen Attributen, Name und Status, die durch das Muster patt ausgewählt werden. Falls das Kommando kein Auswahlmuster anbietet, werden alle Dateien aus dem Strom angenommen. Dateien werden in die Textblockverwaltung integriert, Namen kommen in die Verzeichnisse entsprechend der Pfadnamenstruktur, Statusinformationen kommen in die Inodes.

Das Kommando cpio -p führt nacheinander cpio -o und sofort anschließend cpio -i aus. Dies entspricht dem obigen copy, erfordert aber die Bereitstellung der Pfadnamenliste für cpio -o. Daher wurde im obigen Demonstrationsbeispiel die Kopplung mit dem Kommando find gewählt. Wieder werden einige Optionen des Kommandos cpio vorgestellt.

-v Während des Transports erscheinen Dateinamen auf dem Bildschirm.

-r Rekursive Arbeit (cpio -i und cpio -p).

-m Modifikationszeitpunkte bleiben erhalten (cpio -i und cpio -p).

-d Verzeichnisse werden angelegt (cpio -i und cpio -p).

-t Die Dateinamen des Eingabestroms werden tabelliert (cpio -i).

-l Wo immer möglich, werden Link-Verbindungen anstelle von Duplikaten hergestellt (cpio -p). Diese Option ist ungeeignet, wenn zwischen verschiedenen Dateisystemen kopiert werden soll.

Das Kommando find hat die Aufrufform:

```
find pf1     pfn exp
```

Nacheinander werden die Pfadnamen pf1 ... pfn und gegebenenfalls ihre rekursiv nachgeordneten Verzeichniseinträge daraufhin überprüft, ob sie den Ausdruck exp erfüllen. Einige Formen des Ausdrucks exp werden angegeben. Die Erklärung beschreibt den Wert true.

-name file Eine Datei hat den Namen file.

-print Gilt immer, der Pfadname wird auf stdout ausgegeben.

-depth Gilt immer, die Suche wird rekursiv fortgesetzt.

-user name Datei gehört dem Nutzer name.

-group name Datei gehört einem Nutzer der Gruppe name.

-atime n Der Zeitpunkt des letzten Zugriffs lag innerhalb der letzten n Tage.

-mtime n Die letzte Modifikation lag innerhalb der letzten n Tage.

-ctime n Die letzte Änderung des Inodes lag innerhalb der letzten n Tage.

-newer file Die Datei ist jünger als file (Modifikationszeit).

-size n[c] Die Datei besteht aus n Blöcken (512 Byte) oder Zeichen.

!exp Negation des Ausdruck exp.

exp1 exp2 Ausdruck exp1 und Ausdruck exp2.

exp1 -o exp2 Ausdruck exp1 oder Ausdruck exp2.

(exp) Klammerung des Ausdrucks exp.

-exec cmd ... {} ... \;
Das Kommando cmd wird abgearbeitet und liefert den Statuswert 0. Argumente {} werden durch den aktuellen Pfadnamen ersetzt. \; beendet das Kommando.

## 2.4. Transfer von Dateien

Kein Rechner steht für sich allein. Die Aufgabe, Informationen von außen in die Rechnerumgebung einzubringen oder Informationen zur weiteren Verarbeitung nach außerhalb zu senden, entsteht nach einer kurzen Kontaktperiode für jeden neuen Rechner und auch für jedes Betriebssystem.

Die Dateiverarbeitung des UNIX spielt sich auf den Speichermedien ab, die im System integriert sind. Zumeist handelt es sich um Plattenspeicher. Die Sicherung der Dateien gegen Störungen oder Verlust ist dabei ein vordringliches Anliegen. Dafür sieht die UNIX-Programmierumgebung unterschiedliche Technologien vor. Regelmäßige Gesamtabzüge der Dateispeicher oder sporadisch durch den Systemverwalter veranlaßte Sicherungskopien ausgewählter Datenbestände werden durch Systemprogramme unterstützt. Wird das UNIX-System an Arbeitsplatzcomputern betrieben, so wird oftmals den Anwendern einzeln die Verantwortung für die Sicherung ihrer Daten zugedacht. Dann werden die zu sichernden Dateien auf Disketten übertragen und stehen eventuell in mehreren Generationen zur Verfügung, wenn im Notfall auf sie zurückgegriffen werden muß. Die Datensicherung wird damit auch zum Transferproblem.

### 2.4.1. Transferprogramm tar

Das Kommando tar löst allgemeine Transferprobleme. Es speichert Dateien auf ein Archivierungsmedium und gestattet es, Dateien aus Archiven zu extrahieren, so daß sie wieder in den Dateisystemen des UNIX oder in den Dateisystemen der Nutzern verwendet werden können. tar ist das am weitesten verbreitete Dienstprogramm für diese Zielstellung. Allerdings existiert es auch in vielen Varianten, die sich zwar nicht in den Hauptfunktionen, jedoch in manchen Details voneinander unterscheiden. Der X/OPEN-Standard empfiehlt als Kommandos für den Dateitransfer neben tar auch das Kommando cpio, das im vorigen Abschnitt vorgestellt wurde.

Wir nehmen an, unser Arbeitsverzeichnis enthält die Datei priv. Das Kommando

```
% tar -cvf /dev/fd096ds15 priv
```

transportiert die Datei priv und alle ihr nachgeordneten Dateien, wenn priv eine Verzeichnisdatei ist, auf die im Laufwerk 0 befindliche 1200-K-Diskette. Mit dem Kommando

```
% tar -xvf /dev/fd096ds15
```

wird die auf der Diskette befindliche Verzeichnishierarchie samt den zugehörigen Dateien in das aktuelle Arbeitsverzeichnis eingefügt. Soll nur der Disketteninhalt tabelliert werden, so leistet dies das folgende Kommando:

```
% tar -tvf /dev/fd096ds15
```

Wenn wir davon ausgehen können, daß, wie im Abschnitt 2.3. beschrieben, weitere Namen auf den gleichen Diskettentreiber verweisen, so lauten die drei Kommandos in kürzerer Form:

```
% tar -cvf /dev/fd096 priv
% tar -xvf /dev/fd096
% tar -tvf /dev/fd096
```

Würden 360-K-Disketten im Laufwerk 1 der hier immer wieder verwendeten UNIX-Installation benutzt, so lauteten die Kommandos entsprechend:

```
% tar cvf /dev/fd1 priv
% tar xvf /dev/fd1 priv/book/cont
% tar tvf /dev/fd1 priv/book/cont
```

Die letzten beiden Kommandos unterscheiden sich von den vorangehenden. Hier werden beim Rücktransport oder bei der Inhaltsangabe nur die Dateien aus dem nachgeordneten Verzeichnis priv/book/cont rekonstruiert bzw. bei der Inhaltsangabe berücksichtigt. Das üblicherweise Optionen einleitende Zeichen - ist bei tar nicht vorgeschrieben.

Nun sollen die wichtigsten Funktionen von tar erläutert werden. Das Kommmando tar hat folgende Aufrufform:

```
tar [fct][opt] file1 file2
```

Funktionen fct und optionale Modifikatoren opt werden unterschieden. In den obigen Beispielen waren c, x und t Funktionen, und v und f waren Modifikatoren. tar erfüllt folgende Funktionen:

c Es wird ein neues Archiv gebildet. Dateien werden in das Archiv kopiert. Das Archiv beginnt am Anfang des Speichermediums. Sein bisheriger Inhalt wird überschrieben.

x Die Dateien file1, file2, ... werden aus dem Archiv extrahiert. Handelt es sich um Verzeichnisdateien, so werden die darin erfaßten Dateien rekursiv ebenfalls herausgeholt. Statusinformationen werden rekonstruiert. Wenn keine Dateinamen angegeben sind, wird das ganze Archiv genommen. Sind im Archiv Dateien mehrfach vorhanden, so überschreibt jede Datei die jeweils vorangegangene. Erhalten bleibt nur die letzte Datei.

t Die Dateinamen werden aufgelistet. Dateien gleichen Namens erscheinen mehrfach. Sind keine Dateinamen angegeben, wird das ganze Archiv tabelliert.

r Die Dateien werden an das Ende des Archivs angehängt. Dabei können Mehrfacheinträge entstehen.

u Die Dateien werden dem Archiv zugefügt, wenn sie noch nicht enthalten oder inzwischen geändert worden sind.

Die folgenden Angaben sind Modifikatoren, mit denen die eben aufgezählten Grundfunktionen beeinflußt werden können. Die Liste bietet nur eine geringe Auswahl der tatsächlich für tar angebbaren Optionen.

v Die Aktivitäten von tar werden protokolliert. Beim Tabellieren bewirkt v die zusätzliche Ausgabe von Statusinformationen.

f Das folgende Argument bezeichnet die Archivdatei. Auch die Standard-Ein-/Ausgabe-Dateien sind zulässig. Sie müssen als - notiert werden. Damit kann tar am Anfang oder Ende von Pipes aufgerufen werden.

0,...,n
Dezimalziffern ($n \leq 9$) beziehen sich auf Eintragungen in der Datei /etc/default/tar. Sie wählen das unter dieser Nummer erfaßte Archivgerät aus. Standardmäßig wird das Gerät 0 angesprochen.

Der zuletzt aufgeführte Modifikator erlaubt eine viel kürzere Notation der eingangs erwähnten Aufrufe von tar. Um dies zu demonstrieren, wird die Datei /etc/default/tar aus der aktuellen XENIX-Umgebung abgedruckt.

```
#   @(#) /etc/default/tar von XENIX
#    device                  block   size     tape
archive0=/dev/rfd048ds9       18      360       n
archive1=/dev/rfd148ds9       18      360       n
archive2=/dev/rfd096ds15      10     1200       n
archive3=/dev/rfd196ds15      10     1200       n
archive4=/dev/rfd096ds9       18      720       n
archive5=/dev/rfd196ds9       18      720       n
archive6=/dev/rct0            20        0       y
archive7=/dev/rctmini         20        0       y
archive8=/dev/rmt0            20        0       y
archive9=/tarfile              1        0       y
```

Dabei bedeutet die Größe size das Fassungsvermögen des Archivs in KByte. size=0 weist auf ein nicht formatierbares Speichermedium hin. Die Angabe tape drückt aus, ob als Archiv ein Magnetband verwendet werden kann. Der Blockungsfaktor steht unter block. Bei archive6 werden 20 Blöcke zu einem Magnetbandblock zusammengefaßt. Wir verwenden diese Datei. Die tar-Aufrufe am Anfang dieses Abschnitts lauten damit:

```
% tar 2cv priv
% tar 2xv
% tar 2tv
```

Entsprechend sehen die Aufrufe mit dem Laufwerk 1 so aus:

```
% tar cv1 priv
% tar xv1 priv/book/cont
% tar tv1 priv/book/cont
```

Nun sollen noch die Dateien /etc/default/tar mitgeteilt werden, die für die UNIX-Systeme WEGA und VENIX aufbereitet wurden, um auch dort die Arbeit mit tar zu erleichtern und zu vereinheitlichen.

```
#   @(#) /etc/default/tar von WEGA
#    device                  block   size    tape
archive0=/dev/fd0             18      720     n
archive1=/dev/fd1             18      720     n
archive2=/dev/rfd0            18      720     n
archive3=/dev/rfd1            18      720     n


#   @(#) /etc/default/tar von VENIX
#    device                  block   size    tape
archive0=/dev/f0.40.s.8        8      320     n
archive1=/dev/f1.80.d.9       18      720     n
archive2=/dev/f0.40.d.9       18      360     n
archive3=/dev/f0.40.s.9        9      360     n
```

Einige Transferrezepte könnten damit folgendermaßen aussehen:

```
XENIX nach WEGA:
    in XENIX:       % tar c4  name
                    % tar 4t [name]
    in WEGA:        % tar t  [name]
                    % tar x  [name]
                    % tar X  [name]

WEGA nach XENIX:
    in WEGA:        % tar c   name
                    % tar t  [name]
    in XENIX:       % tar 4t [name]
                    % tar x4 [name]

XENIX nach VENIX:
    in XENIX:       % tar c1 name   % tar c4 name
    in VENIX:       % tar x2        % tar x1

VENIX nach XENIX:
    in VENIX:       % tar c2 name   % tar c1 name
    in XENIX:       % tar x1        % tar x4
```

Der Transfer mit Standarddisketten 1200K oder 360K im aktuellen XENIX-System verläuft nach dem folgenden einfachen Rezept:

```
XENIX nach N.N.

     1200-K-Diskette, Laufwerk 0:    % tar c2 name
      360-K-Diskette, Laufwerk 1:    % tar c1 name

XENIX von  N.N.

     1200-K-Diskette, Laufwerk 0:    % tar x2
      360-K-Diskette, Laufwerk 0:    % tar x
      360-K-Diskette, Laufwerk 1:    % tar x1
```

### 2.4.2. Transfer überlanger Dateien

Damit sollen solche Dateien gemeint sein, die nicht auf einer Diskette Platz finden oder nicht mehr in den noch freien Platz auf einer Diskette passen. Das folgende Terminalskript berichtet über eine Arbeit im UNIX-System WEGA. Zuerst wird ein kleines Programm mit dem Namen tar_experiment eingegeben und vorgeführt, das Dateien einer vorgegebenen Länge erzeugt. Es druckt L Zeilen (L>1) mit einer festen Zeichenzahl von 16 Byte, das Zeilenendezeichen mitgezählt. Am Ende des zunächst abgedruckten Skriptteils werden die Dateien f1 und f2 mit einer Länge von jeweils 200 KByte und die Datei f3 mit 400 KByte hergestellt.

```
Script started on Fri Jan 29 14:52:04 1988
neue C-Shell
1% set prompt='W-%
W-% ed tar_experiment.c
?tar_experiment.c
a
main(argc,argv)
int argc; char *argv[]; {
int L;
register int j;
sscanf(argv[1],"%d",&L);
printf("L= %12d\n",L);
for (j=2;j<=L;j++)
  printf("Zeile%10d\n",j);
}
.
w
161
q
W-% make tar_experiment
    cc -O  tar_experiment.c -o tar_experiment
W-% tar_experiment 4
L=           4
Zeile        2
Zeile        3
Zeile        4
W-% tar_experiment 4 >f
W-% ls -l f
-rw-rw-rw- 1 polze    mitarbei     64 Jan 29 14:58 f
W-% tar_experiment 12800 >f1
W-% ^f1^f2
tar_experiment 12800 > f2
W-% tail -4 f2
Zeile     12797
Zeile     12798
Zeile     12799
Zeile     12800
W-% cat f1 f2 >f3
W-% ls -l f?
-rw-rw-rw- 1 polze    mitarbei 204800 Jan 29 15:09 f1
-rw-rw-rw- 1 polze    mitarbei 204800 Jan 29 15:10 f2
-rw-rw-rw- 1 polze    mitarbei 409600 Jan 29 15:16 f3
```

Die drei Dateien f1, f2 und f3 zusammen passen nicht auf die WEGA-üblichen 720-K-Disketten. Im folgenden beobachten wir die Arbeit des Kommandos tar in WEGA. Wir formatieren zwei Disketten und rufen tar auf. Während der Übertragung von f3 verlangt das Betriebssystem den Wechsel zur zweiten Diskette und setzt die Arbeit fort.

```
W-% format rfd1
Usage: format fdx [x=0 or 1]
W-% format fd1              # Erste  Diskette  D1
W-% format fd1              # Zweite Diskette  D2

W-% tar cv f[1-3]           # Diskette D1
tar: disk capacity 1440 blocks
a f1 400 blocks
a f2 400 blocks
a f3 800 blocks
insert next disk            # Diskette D2
```

Wir lesen das gesamte eben erzeugte Archiv zur Kontrolle und Bestätigung. Wieder wird während des Lesens von f3 der Wechsel zur zweiten Diskette verlangt. Die drei gelesenen Dateien stimmen mit den Originalen überein, wie die drei Aufrufe des Kommandos cmp zeigen.

```
W-% mkdir Tartst
W-% cd Tartst
W-% tar xv                  # Diskette D1
tar: disk capacity 1440 blocks
Get uid/gid from Archiv
x f1, 204800 bytes, 400 archiv blocks
x f2, 204800 bytes, 400 archiv blocks
x f3, 409600 bytes, 800 archiv blocks
insert next disk :          # Diskette D2
W-% ls -l
total 1600
-rw-rw-rw- 1 polze    mitarbei 204800 Jan 29 15:09 f1
-rw-rw-rw- 1 polze    mitarbei 204800 Jan 29 15:10 f2
-rw-rw-rw- 1 polze    mitarbei 409600 Jan 29 15:16 f3
W-% cmp f1 ../f1
W-% cmp f2 ../f2
W-% cmp f3 ../f3
```

Auch wenn wir nur die Datei f1 aus dem Archiv extrahieren wollen, wird die zweite Diskette angefordert. Setzt sich ein Archiv über mehrere Disketten hinweg fort, können ja mehrere Kopien der gleichen Datei darin enthalten sein. Nach der Kommandobeschreibung von tar soll die Extraktion die zuletzt gefundene Datei liefern. Zugleich wird bei den beiden Suchaktionen der WEGA-typische Unterschied zwischen den Optionen -x und -X sichtbar. Am Ende des Skripts werden die überlangen, aber sonst sinnlosen Dateien f1, f2 und f3 wieder beseitigt.

```
W-% tar Xv f1               # Diskette D1
tar: disk capacity 1440 blocks
Take your own uid/gid
x f1, 204800 bytes, 400 archiv blocks
insert next disk :          # Diskette D2
```

```
W-% ls -l f1
-rw-rw-rw- 1 polze    mitarbei 204800 Jan 29 15:09 f1
W-% cd ..
W-% rm -rf Tartst
W-% rm f*
W-%
script done on Fri Jan 29 15:52:12 1988
```

Jetzt verlassen wir das UNIX-System WEGA und versuchen, die beiden Disketten im UNIX-System XENIX zu lesen. Nehmen wir das Anliegen des Dateitransfers zwischen unterschiedlichen UNIX-Systemen ruhig einmal ganz wörtlich. Unsere ersten Versuche sind entmutigend. Verschiedene Funktionen des Kommandos tar, angewendet auf die Diskette D1, zeigen recht unterschiedliche Ausgabebilder. Allen gemeinsam ist der Hinweis »directory checksum error«. Gleiches zeigt die Diskette D2. Übrigens verträgt die C-Shell des XENIX am Terminal die Texte "# Diskette D1/2" nicht. Das Terminalskript wurde an diesen Stellen nachträglich kommentiert.

```
Script started on Mon Feb  1 11:19:05 1988
Neue C-Shell
1% set prompt='X-%
X-% tar tvf /dev/fd096ds9   #Diskette 1
tar: blocksize = 20
tar: directory checksum error
X-% tar tf /dev/fd096ds9
tar: directory checksum error
f1
f2
f3
X-% tar t4                  #Diskette 1
tar: read error while skipping file
X-% tar t4                  #Diskette 2
tar: directory checksum error
```

Eine Gewaltkur hilft aus der Klemme. Wir kopieren die Disketten nacheinander in die Datei tardat. Diese Datei hat die eindrucksvolle Länge von 1474560=2*720*1024 Byte. Auf diese Datei wenden wir das Kommando tar mit den Funktionen t und danach x an. Jetzt läuft alles reibungslos ab.

```
X-% cat </dev/fd096ds9 >tardat      # Diskette D1
X-% cat </dev/fd096ds9 >>tardat     # Diskette D2
X-% ls -l tardat
-rw-r--r--   1 polze    HUB      1474560 Feb  1 11:38 tardat
X-% tar tf tardat
f1
f2
f3
X-% tar xvf tardat
tar: blocksize = 20
x f1, 204800 bytes, 400 tape blocks
x f2, 204800 bytes, 400 tape blocks
x f3, 409600 bytes, 800 tape blocks
X-% ls -l f?
-rw-r--r--   1 polze  HUB   204800 Jan 29 15:09 f1
-rw-r--r--   1 polze  HUB   204800 Jan 29 15:10 f2
-rw-r--r--   1 polze  HUB   409600 Jan 29 15:16 f3
```

Auch in XENIX besitzen wir das Programm tar_experiment. Wir erzeugen die Dateien ff1 und ff3, 200 KByte und 400 KByte lang, noch einmal und vergleichen sie mit den transferierten Dateien. Der Vergleich ist positiv. Die Spuren der Vorführung können beseitigt werden.

```
X-% make tar_experiment
    cc -O  tar_experiment.c -o tar_experiment
tar_experiment.c
    rm -f tar_experiment.o
X-% tar_experiment 12800 >ff1
X-% cmp ff1 f1
X-% cmp ff1 f2
X-% cat ff1 ff1 >ff3
X-% cmp ff3 f3
X-% echo $status
0
X-% rm ff? f?
X-%
script done on Mon Feb  1 11:47:39 1988
```

Die Beobachtungen am WEGA verleiten zu der Frage, wie XENIX den Wechsel von Disketten organisiert, wenn Dateien an die Grenzen ihres Fassungsvermögens stoßen. Wir verwenden zwei 360-K-Disketten und Dateien zu je 200 KByte. Zuerst werden die beiden Disketten formatiert.

```
Script started on Mon Feb  1 14:00:33 1988
Neue C-Shell
1% set prompt='X-% '
X-% tar_experiment 12800 >f1
X-% cp f1 f2
X-% ls -l f?
-rw-r--r--   1 polze    HUB        204800 Feb  1 14:02 f1
-rw-r--r--   1 polze    HUB        204800 Feb  1 14:02 f2
X-% format /dev/rfd1                 # Diskette D1
Insert floppy in drive; hit return when ready
formatting /dev/rfd1 ...
  track 00  head 0
    ... usw. ...
  track 39  head 1
done
X-% !!                               # Diskette D2
format /dev/rfd1
Insert floppy in drive; hit return when ready
formatting /dev/rfd1
  track 00  head 0
    ... usw. ...
  track 39  head 1
done
```

Jetzt soll ein Archiv kreiert werden. Beginnen wir mit der Diskette 1. Die Datei f1 wird kopiert. Die Datei f2 wird in zwei Teile zu 158 KByte und 42 KByte zerlegt. Nach dem Diskettenwechsel wird Teil 2 auf die zweite Diskette geschrieben. Die Situation wird unmißverständlich beschrieben. Das Zeichen \07 (nachträglich in das Terminalskript eingebaut) unterstützt die Aufforderung zum Diskettenwechsel akustisch.

```
X-% tar cv1 ./f?                        # Diskette D1
tar: large file ./f2 needs 2 extents.
tar: current device seek position = 201K
tar: \07please insert new volume, then press RETURN.
                                        # Diskette D2
Volume ends at 359K, blocking factor = 9K
seek = 0K    a ./f1 200K
+++ a ./f2 158K [extent #1 of 2]
+++ a ./f2 42K [extent #2 of 2]
a ./f2 200K (in 2 extents)
```

Versuchsweise rufen wir tar -t mit der zweiten Diskette auf. Am Terminal erscheint der Hinweis, daß es sich um den zweiten von zwei Teilen der Datei ./f2 handelt. Anschließend extrahieren wir f1 und f2 aus dem zweiteiligen Archiv und vergleichen sie mit den Originalen. Es gibt keine Beanstandungen.

```
X-% tar tv1                             # Diskette D2
rw-r--r--200/200  43008 Feb  1 14:02 1988 ./f2
 [extent #2 of 2] 204800 bytes total
X-% mkdir Tst
X-% cd Tst
X-% tar xv1                             # Diskette D1
tar: ./f2 split across 2 volumes
tar: \07please insert new volume, then press RETURN.
                                        # Diskette D2
x ./f1, 204800 bytes, 200K
+++ x ./f2 [extent #1], 161792 bytes, 158K
+++ x ./f2 [extent #2], 43008 bytes, 42K
x ./f2 (in 2 extents), 204800 bytes, 200K
X-% lc
f1  f2
X-% cmp f1 ../f1
X-% cmp f2 ../f2
X-% echo $status
0
X-% cd ..
X-% tar cv1ek 360 ./f?                  # Diskette D1
Volume ends at 359K, blocking factor = 9K
seek = 0K       a ./f1 200K
tar: \07please insert new volume, then press RETURN.
                                        # Diskette D2
seek = 0K       a ./f2 200K
X-% tar tv1                  # Zweite Diskette D2
rw-r--r--200/200 204800 Feb  1 14:02 1988 ./f2
X-% tar tv1                  # Erste  Diskette D1
rw-r--r--200/200 204800 Feb  1 14:02 1988 ./f1
X-% rm -rf Tst f?
X-%
script done on Mon Feb  1 14:17:11 1988
```

Am Schluß des Skripts zeigen wir noch die Arbeitsweise des Kommandoaufrufs tar -ek 360. Hier wird die Datei f2 nicht zerlegt, sondern als Ganzes auf die zweite Diskette geschrieben. Die beiden Disketten können jetzt unabhängig voneinander mit dem Kommando tar behandelt werden.

### 2.4.3. Transfer mit MS-DOS

Die Zusammenarbeit zwischen XENIX und MS-DOS ist auf verschiedenen Ebenen möglich. Im Vordergrund steht natürlich die Übertragung von Dateien zwischen den beiden Betriebssystemen. Dafür gibt es ein ganzes Paket von Kommandos. Die Entwicklung von MS-DOS-Programmen kann mit den Hilfsmitteln des XENIX jedoch bis zur Erzeugung abarbeitbarer Versionen betrieben werden. Das letzte XENIX-Werkzeug ist dann das Kommando dosld. Das Resultat seiner Arbeit muß aber auch als Datei transferiert werden. Die einzelnen Kommandos haben folgende Aufrufformen:

```
dosdir dir1 dir2
dosls  dir1 dir2 ...
doscat [-r] file1 file2
doscp [-r] file1 file2
doscp [-r] f1 f2 ... fn dir
dosrm file1 file2 ...
dosmkdir dir1 dir2
dosrmdir dir1 dir2 ...
dosld opt file1 file2
dosformat device
```

Diese Kommandos erlauben den Zugriff auf Dateien und Dateiverzeichnisse auf DOS-Disketten. Ebenfalls ist es möglich, auf Dateien zuzugreifen, die sich auf einem dem Betriebssystem MS-DOS zugeordneten Teil von Plattenspeichern befinden. DOS-Dateien werden in der Form

```
device:name
```

beschrieben. Dabei bezeichnet device einen Pfadnamen für die Gerätedatei, mit der die DOS-Diskette oder der DOS-Teil eines Plattenspeichers erreicht wird. Für die kürzere Notation dieser Pfadnamen sind MS-DOS-übliche Gerätenamen eingeführt worden, so daß beispielsweise für die Datei name auf der DOS-Diskette im Laufwerk 1

```
B:name   oder   b:name
```

geschrieben werden kann. Die zulässigen DOS-Gerätenamen sind in der Datei /etc/default/msdos zusammengefaßt. Es folgt ein Listing dieser Datei aus der XENIX-Umgebung.

```
# @(#) /etc/default/msdos von XENIX
#
    A=/dev/fd048ds9
    B=/dev/fd148ds9
    C=/dev/hd0d
    D=/dev/hd1d
    X=/dev/fd096ds15
    Y=/dev/fd196ds15
```

MS-DOS-Textdateien benutzen die Zeichenkombination <carriage return> <newline> für den Zeilenwechsel. Im UNIX steht <newline> allein. Die Transferkommandos erzeugen die DOS-Kombination beim Transfer zum MS-DOS und beseitigen das <carriage return>-Zeichen beim Transfer in das UNIX-System. Wenn bei doscat oder doscp die Option -r verwendet wird, werden Bytefolgen ohne Modifikation übertragen. Die einzelnen Kommandos leisten folgendes:

dosdir Inhaltsangabe im Stil des MS-DOS.

dosls Inhaltsangabe im Stil des UNIX-Kommandos ls.

doscat DOS-Dateien werden nach stdout kopiert.

doscp Kopiert die Datei file1 in die Datei file2 oder die Dateien f1, ... , fn in das Verzeichnis dir. Ziel- und Quelldateien müssen aus verschiedenen Betriebssystemen stammen.

dosrm Beseitigt die angegebenen Dateien.

dosmkdir Bildet DOS-Verzeichnisse.

dosrmdir Beseitigt DOS-Verzeichnisse.

dosld Verbindet die angegebenen Objektdateien und bildet ein im MS-DOS ausführbares Programm. Es steht eine große Auswahl an Optionen zur Verfügung.

dosformat Gestattet die Formatierung von DOS-Disketten.

Im folgenden Terminalskript wird die Arbeit mit einigen MS-DOS-Transferkommandos vorgeführt. Zuerst werden die Verzeichnisse im Stil des MS-DOS und dann in der Form des UNIX ausgegeben. Das Kommando dosls und die folgenden Kommandos benutzen den kurzen Gerätenamen b: für das Diskettenlaufwerk 1. Das Hilfsprogramm uplo ersetzt Großbuchstaben durch Kleinbuchstaben. Das abschließende Kommando ls -l soll den Erfolg der Operation belegen.

```
% pwd
/u/polze/priv/Book/Content/T24
% mkdir Cocol
% cd Cocol
% dosdir /dev/fd1
 Volume in drive /dev/fd148ds9 has no label
 Directory of /dev/fd148ds9:/
COCOLGRA  DEF     3174   0-00-80  12:00a
COCOLLEX  DEF     1714   0-00-80  12:00a
% dosls b:
COCOLGRA.DEF
COCOLLEX.DEF
% foreach i (`dosls b:`)
? echo $i
? doscp b:$i `uplo $i`
? end
COCOLGRA.DEF
COCOLLEX.DEF
% ls -l
total 714
-rw-r--r--   1 polze HUB     3089 May 19 14:15 cocolgra.def
-rw-r--r--   1 polze HUB     7106 May 19 14:17 cocolgra.mod
%
```

# 3. Editoren

## 3.1. Der Editor ed

Der Texteditor ed ist in jedem UNIX-System verfügbar. Er gehört zum Anfangsbestand von UNIX und hat sich über alle Versionswechsel hinweg behauptet. Jüngere Editoren, insbesondere bildschirmorientierte Editoren sind an seine Seite getreten. Sie haben ihre spezifischen Einsatzmöglichkeiten. Die Stärken des ed sind seine Arbeitsgeschwindigkeit, die Verwendung einer recht allgemeinen Klasse von Suchmustern in der Form der regulären Ausdrücke und die interaktionsfreie Steuerung seiner Arbeit durch Skriptdateien, die sogar noch vorangehende Modifikationen durch Kommandointerpreter zulassen. Den regulären Ausdrücken des ed begegnet man auch bei der Beschäftigung mit den gebräuchlichen Kommandos sed und grep sowie in einer erweiterten Gestalt bei awk, egrep und lex. Kenntnisse über den Texteditor ed gehören daher zum Grundwissen über die Arbeit mit UNIX. Mehrere Beispiele in diesem Abschnitt lehnen sich an Kernighan und Pike (1984) an. Ansonsten werden nahezu alle Möglichkeiten des Editors ed nach X/OPEN (1987) dargestellt.

### 3.1.1. Arbeitsweise

Der Aufruf von ed genügt folgender Syntax:

```
ed [-] [-p string] [file]
```

Wird eine Datei als Operand file angegeben, so stellt ed eine Kopie davon in seinen Pufferbereich. Alle Edierkommandos wirken auf diese Kopie. Erst mit dem Schreibkommando w wird die Originaldatei durch die Kopie ersetzt. Wenn der Operand file nicht auf eine bestehende Datei verweist, wird eine neue Datei erzeugt.

Mit der Option -p wird eine Zeichenkette als Aufforderungsprompt für die Kommandoeingaben festgelegt. Der Editor erwartet seine Kommandos von der Standardeingabe. Soll eine Skriptdatei Kommandos liefern, so muß man die Standardeingabe auf diese Datei lenken.

Die Option - unterdrückt einige Editorausgaben, so die Zahl gelesener oder geschriebener Zeichen bei den Editorkommandos e, r und w, Diagnostiktexte bei den Kommandos e und q sowie das Promptzeichen !, wenn ein Exkurs aus dem Editor heraus in den Kommandointerpreter zu Ende geht. Dies ist nützlich, wenn der Editor schweigsam arbeiten soll, vielleicht durch eine Skriptdatei gesteuert. X/OPEN (1987) kündigt an, daß diese Option in späteren Versionen durch -s ersetzt werden kann.

Kommandos haben eine einheitliche syntaktische Struktur.

```
[a1[,a2]]cmd[param]
```

Als Adressen dienen a1 oder a2. Kommandos cmd werden durch einen Buchstaben bestimmt. Parameter sind in unterschiedlicher Zahl, abhängig von den jeweiligen Kommandos, angebbar. Folgende Regeln gelten für die Adressierung in Editorkommandos

1. Das Zeichen . adressiert die aktuelle Zeile. Am Beginn der Editorarbeit ist dies die letzte Zeile im Editorpuffer. Jedes Kommando bestimmt die nächste aktuelle Zeile.

2. Das Zeichen $ adressiert die letzte Zeile des Puffers.

3. Eine Dezimalzahl n adressiert die n-te Zeile des Puffers.

4. 'x adressiert die mit dem Kleinbuchstaben x markierte Zeile. Marken werden mit dem Editorkommando k vergeben.

5. Als Adressen werden Muster /reg/ oder ?reg? mit einem regulären Ausdruck reg akzeptiert (s. Abschn. 3.1.4). Mit dem Muster /reg/ wird vorwärts gesucht. Mit ?reg? verläuft die Suche rückwärts. Die Dateigrenzen werden verbunden. Auf das Dateiende folgt vorwärts die Zeile 1. Rückwärts folgt nach Passieren des Dateianfangs die Zeile $. Die Suche beginnt vorwärts in der Zeile nach der aktuellen, rückwärts in der Zeile vor der aktuellen Zeile.

6. Es sind einfache Adreßrechnungen zulässig. Sei a eine Adresse und n eine positive Dezimalzahl. Dann sind a+n und a-n gültige Adressen. a+n kann auf an verkürzt werden. Beispielsweise sind /reg/+3 und /reg/3 gleiche Adressen. Wenn n=1 gilt, kann n weggelassen werden. Wieder sind /reg/1, /reg/+1 und /reg/+ einander gleich.

7. Plus- oder Minuszeichen vor einer Adresse verknüpfen diese mit der Nummer der aktuellen Zeile. So ist -3 mit der Adresse .-3 identisch. Folgen von Plus- oder Minuszeichen iterieren die entsprechende Verknüpfung.

8. Abkürzend sind die Zeichen , und ; Adreßpaare. Das Zeichen , steht für das Paar 1,$ und das Zeichen ; für das Paar .,$

9. Treten Adressen paarweise auf, so muß die erste Adresse im Editorpuffer vor der zweiten Adresse liegen. Neben Paaren a1,a2 sind auch Adreßpaare a1;a2 zulässig. In diesem Fall wird zuerst die aktuelle Zeile durch a1 bestimmt und danach die zweite Adresse berechnet. So sind /reg/,.+3 und /reg/;.+3 voneinander verschieden, wenn das Muster /reg/ in der aktuellen Zeile nicht vorkommt.

Mit den Kommandos q oder Q wird der Editor ed wieder verlassen. Falls sich der Editorpuffer geändert hat, falls also wirklich ediert und die Datei nicht nur besichtigt wurde, reagiert das Kommando q zunächst mit einer Warnung, wenn der veränderte Text noch nicht ausgeschrieben wurde. Die Wiederholung des Kommandos q beendet dann die Editorarbeit. Das Kommando Q schließt die Editorarbeit ohne solche Warnungen unmittelbar ab.

### 3.1.2. Druckkommandos, Hilferufe und Mustersuche

Das Kommando p druckt die Zeilen a1 bis a2. Danach ist die Zeile a2 die neue aktuelle.

Das Kommando l druckt in gleicher Weise wie das Kommando p; allerdings werden dabei auch sonst nicht sichtbare Zeichen angezeigt, wie das Zeichen <tab> als >, das Zeichen <backspace> als < sowie weitere Sonderzeichen. Nicht deutbare Sonderzeichen werden in ihrem Oktalkode dargestellt.

Das Kommando n druckt ebenso wie das Kommando p die Zeilen a1 bis a2. Dabei wird vor jede Zeile deren Nummer und ein <tab>-Zeichen gesetzt.

Alle drei Druckkommandos können an andere Edierkommandos angehängt werden. Zeilen können auch dadurch gedruckt werden, daß eine Adresse ohne Kommando eingetastet wird. Drücken der <return>-Taste ohne Zeicheneingabe schaltet zur nächsten Zeile weiter. Beide Male erscheint die Zeile in der Ausgabeform des Kommandos p.

Hier sind einige Beispiele. Wir edieren die schon mehrfach verwendete Datei accu.c.

```
% ed accu.c
330
1p          Druckt die Zeile 1 in der Druckart p.
main() {
1n          Druckt die Zeile 1 in der Druckart n.
1   main() {
P
```

Das Kommando P schaltet das Arbeitsregime des Editors um. Jede Kommandoanforderung wird von nun an durch die Ausgabe des Promptzeichens * sichtbar gemacht. Nochmalige Angabe von P schaltet wieder auf den vorherigen Arbeitsmodus zurück.

```
*,p
main() {
int key,val;
int sum,totsum;
printf("Bericht\n");

     usw.

printf("Gesamtsumme %10d\n",totsum);
}
```

Das Kommando ,p steht als Abkürzung für 1,$p. Die ganze im Puffer befindliche Datei wird gedruckt. Jetzt folgen Druckversionen des inneren while-Zyklus aus dem Programm accu.c.

```
*10,15p             - while-Zyklus Druckart p.
    while (key == 0) {
            scanf("%d",&val);
            printf("%10d\n",val);
            sum += val;
            scanf("%d",&key);
    }

*10,15n             - while-Zyklus Druckart n.
10          while (key == 0) {
11                  scanf("%d",&val);
12                  printf("%10d\n",val);
13                  sum += val;
14                  scanf("%d",&key);
15          }

*10,15l             - while-Zyklus Druckart l.
>while (key == 0) {
>>scanf("%d",&val);
>>printf("%10d\n",val);
>>sum += val;
>>scanf("%d",&key);
>}
```

```
*10,15ln            - while-Zyklus Druckart ln.
10>>while (key == 0) {
11>>>scanf("%d",&val);
12>>>printf("%10d\n",val);
13>>>sum += val;
14>>>scanf("%d",&key);
15>>}
```

Mit dem Kommando = kann die Nummer der aktuellen Zeile abgefragt werden. Ohne Adressenangabe wird die Nummer der letzten Zeile geliefert, und damit also die Zahl der im Editorpuffer gesammelten Zeilen.

```
.=          Nummer der aktuellen Zeile.
 =          Nummer der letzten Zeile.
```

Bisweilen reagiert der Editor mit der Ausgabe eines Fragezeichens. Mit dem Kommando h kann man Hilfe erbitten. Der Editor gibt dann eine kurze Erklärung ab. Falls man den Diagnosetext bei jedem Fragezeichen wünscht, steht das Kommando H zur Verfügung. Es schaltet den Diagnosemodus ein und bei nochmaliger Angabe wieder aus. Der Editor beginnt seine Arbeit mit ausgeschalteter Diagnose.

```
% ed -p '*' f                %ed -p '*' g
?f                           ?
*h                           *h
cannot open input file       illegal character in input file
*a                           *q
Text                         %
.
*q                           %ed -p '*' gg
?                            ?
*h                           *h
warning: expecting `w'       line too long
*q                           *Q
%                            %
```

Die drei Beispiele zeigen einige Diagnosefälle. Die Datei f wird neuerstellt. Die Datei g beinhaltet ein abarbeitungsfähiges Programm. Die Datei gg enthält eine Zeile mit mehr als 256 Zeichen.

Jetzt suchen wir in der Datei accu.c nach Zeilen, in denen fragliche Texte vorkommen. Texte werden durch einfache Muster charakterisiert. Der Aufruf des Editors zeigt außerdem die Option p. Damit fordert er neue Kommandos mit dem Zeichen : als Prömptzeichen.

```
% ed -p      accu.c
330
:/print/    Gibt es eine Zeile mit dem Muster print?
printf("Bericht\n");
:/print/    Suche nach der nächsten Zeile.
    printf("%10d\n",val);
://         Suche mit dem gleichen Muster.
  printf("Gruppensumme%10d\n",sum);
://         Noch einmal.
printf("Gesamtsumme %10d\n",totsum);
:??         Suche in der umgekehrten Richtung.
  printf("Gruppensumme%10d\n",sum);
```

Mit dem Muster /print/ suchen wir Zeilen, in denen dieser Text vorkommt. Die Suche beginnt in der Zeile, die auf die aktuelle folgt. Zu Beginn der Editorarbeit ist die letzte Zeile der vorhandenen Datei

accu.c die aktuelle Zeile. Da das Dateiende mit dem Anfang verbunden wird, beginnt die Suche mit der ersten Zeile der Datei. Wird eine geeignete Zeile gefunden, so wird sie gedruckt. Die wiederholte Eingabe des Musters findet die nächste Zeile, die das Muster enthält. Der Editor speichert das Suchmuster. Die Eingabe // verwendet das gespeicherte Muster. Mit der Eingabe ?? wird auch das alte Muster wiederverwendet. Die Suche verläuft jedoch rückwärts. Sie beginnt mit der Zeile vor der aktuellen.

Die zuletzt gefundene Zeile ist jetzt aktuelle Zeile der Datei. Mit dem Muster ?while? suchen wir rückwärts eine Zeile, die diesen Text enthält, und dies gleich mehrere Male. Dabei passieren wir die Dateigrenze und finden mit der letzten Zeile die erste while-Zeile noch einmal.

```
:?while?    Gibt es eine Zeile mit dem Muster while?
  while (key == 0) {
:??         Suche nach der vorigen Zeile mit gleichem Muster.
while (key == 1) {
:??         Noch einmal.
  while (key == 0) {
```

Anstelle der Muster /print/ oder ?while? hätten die kürzeren Formen /print oder ?while zum gleichen Ergebnis geführt.

### 3.1.3. Eingabemodus, Löschen und Transport von Zeilen

Die Kommandos a, i und c schalten den Editor in den Eingabemodus um. Der nachfolgend eingegebene Text wird unbesehen entgegengenommen. Der Text wird mit der Zeile

<newline>

abgeschlossen. Diese Schlußzeile besteht also aus genau zwei Zeichen. Die maximale Länge einer Textzeile ist 256. Das sie abschließende <newline>-Zeichen wird dabei mitgezählt.

Das Kommando a hängt den Text an die aktuelle Zeile an. Die letzte Zeile des eingegebenen Textes wird zur neuen aktuellen. Falls kein Text gekommen ist, bleibt die Zeile aktuell, die vor dem Kommando a bereits aktuelle Zeile war. Die Adresse 0 ist zulässig. Der Text wird dann vor der ersten Zeile plaziert.

Das Kommando i legt den Text vor der aktuellen Zeile ab. Auch hier wird die letzte Textzeile die neue aktuelle Zeile. Bei einem leeren Text ändert sich die aktuelle Zeile nicht. Die Adresse 0 ist nicht erlaubt.

Das Kommando c löscht die Zeilen von der Adresse a1 bis einschließlich Adresse a2. Danach wird der Eingabetext eingefügt. Die letzte Eingabezeile wird aktuell. Bei Beendigung des Eingabemodus ohne die Eingabe von Zeilen wird die Zeile a1-1 aktuell oder eben die Zeile 1, falls das Kommando c mit der Adresse a1=1 gegeben war.

Das Kommando d löscht die adressierten Zeilen aus dem Editorpuffer. Die nachfolgende Zeile wird dabei zur aktuellen Zeile erklärt. Falls keine Zeile folgt, so ist die letzte vorhandene Zeile fortan aktuell.

Die zwei Transportkommandos m und t bewegen ganze Zeilengruppen. Das Kommando m verschiebt den durch die beiden Adressen a1 und a2 angegebenen Bereich an eine neue Position. Das Kommando t kopiert den Bereich an eine andere Stelle.

| | |
|---|---|
| a1,a2mb | Verlagerung der Zeilen a1 bis a2 nach b. |
| a1,a2tb | Einfügen einer Kopie nach der Zeile b. |

Werden Adressen weggelassen, so werden sie jeweils durch die aktuelle Position ersetzt. Der transportierte Text wird an die Zeile mit der Zieladresse b angehängt. Die Zieladresse b=0 ist zulässig. Der Text kommt dann an den Anfang der Datei. Bei dem Kommando m darf die Zieladresse b nicht in den zu transportierenden Text zeigen. Verstöße dagegen führen zu einer Fehlermeldung. Nach der Abarbeitung von Transportkommandos ist die letzte Zeile das transportierten Textes die neue aktuelle Zeile.

Hier einige Beispiele von Transportkommandos:

| | |
|---|---|
| m+ | Anhängen der aktuellen Zeile an die folgende (Austauschen zweier aufeinanderfolgender Zeilen). |
| m-2 | Einfügen der aktuellen Zeile vor der vorigen (also auch Zeilentausch). |
| m-- | Dies ist mit dem vorigen Kommando identisch. |
| m- | Dieses Kommando ist wirkungslos. |
| m$ | Transport der aktuellen Zeile an das Dateiende. |
| m0 | Transport der aktuellen Zeile an den Dateianfang. |
| t. | Anhängen einer Kopie der aktuellen Zeile an die aktuelle Position (Verdopplung der Zeile). |
| 1,$t$ | Verdoppeln der ganzen Datei. |
| ,t$ | Dies ist mit dem vorigen Kommando identisch. |
| g/^/m0 | Die Reihenfolge der Zeilen wird umgekehrt. |
| g/$/m0 | Dies ist mit dem vorigen Kommando identisch. |

### 3.1.4. Substitutionen, reguläre Ausdrücke, globale Kommandos

Gewiß ist das Substitutionskommando s das am häufigsten benutzte Kommando des Editors ed. Es ersetzt ein als ersten Parameter angegebenes Textmuster durch ein anderes, das als zweiter Parameter des Kommandos folgt. Das Kommando s läßt viele Modifikationen zu. Ohne Adressenangaben a1 oder a2 bezieht es sich auf die aktuelle Zeile.

| | |
|---|---|
| s/alt/neu/ | Ersetzt in der aktuellen Zeile den erstmals auftretenden Text alt durch den Text neu. |
| s/alt/neu/p | Leistet dasselbe, druckt jedoch die geänderte Zeile. |
| s/alt/neu | Ist mit dem vorigen Kommando identisch. |
| s/alt/neu/ln | Wie das vorige Kommando, jedoch nach der Druckart ln. |

Wenn der zu ersetzende Text in mehreren aufeinanderfolgenden Substitutionskommandos gleichbleibt, kann er durch // abgekürzt werden. Das zeigen die folgenden Kommandos:

| | |
|---|---|
| s/alt/neu1/ | Ersetzt den ersten Text alt durch den Text neu1. |
| s//neu2/ | Ersetzt den ersten Text alt durch den Text neu2. |

Wenn der Ersetzungstext in mehreren aufeinanderfolgenden Substitutionskommandos gleichbleibt, kann er durch das Zeichen % abgekürzt werden. Dies verdeutlichen die Kommandos

```
s/alt1/neu/   Ersetzt den ersten Text alt1 durch den Text neu.
s/alt2/%/     Ersetzt den ersten Text alt2 durch den Text neu.
s/alt3/\%/    Ersetzt den ersten Text alt3 durch das Zeichen %.
```

Die Belegung des Zeichens % bleibt bestehen, bis ein anderer Ersetzungstext erscheint, der danach als Wert für das Sonderzeichen % gespeichert wird. Ein vorangestelltes <backslash>-Zeichen hebt die Sonderbedeutung von % auf.

Auch das Begrenzungszeichen für Textmuster kann modifiziert werden. Als Grenzzeichen dient das auf den Kommandobezeichner s folgende Zeichen. Mögen die Muster alt und neu beide den Buchstaben X nicht enthalten. Dann sind auch folgende Kommandos untereinander gleich:

```
s/alt/neu/
sXaltXneuX    Alle diese Kommandos ersetzen den ersten Text alt
sXXneuX       durch den Text neu.
sXX%X
s//%/
```

Sind die gewählten Textmuster erst einmal durch modifizierte Grenzzeichen beschrieben, so kann auch der übliche Schrägstrich / wieder verwendet werden. Das zeigt das letzte Kommando der obigen Reihe.

Mit dem Modifikationsparameter g wird erreicht, daß alle Vorkommnisse der Zeichenreihe alt durch den Text neu ersetzt werden. Wieder ist das Kommando mit den verschiedenen Druckarten kombinierbar.

```
s/alt/neu/g   Ersetzt alle Texte alt durch den Text neu.
s/alt/neu/gp  Leistet dasselbe, druckt jedoch die geänderte Zeile.
s/alt/neu/gl  Wie das vorige Kommando, jedoch nach der Druckart l.
s/alt/neu/gn  Wie das vorige Kommando, jedoch nach der Druckart n.
```

Durch die Angabe einer Zahl i ($1 \le i \le 512$) wird das i-te Auftreten der Zeichenreihe alt gesucht und dieses durch den Text neu ersetzt.

```
s/alt/neu/2   Ersetzt den zweiten Text alt durch den Text neu.
s/alt/neu/2p  Leistet dasselbe, druckt jedoch die geänderte Zeile.
s/alt/neu/2l  Wie das vorige Kommando, jedoch nach der Druckart l.
s/alt/neu/2n  Wie das vorige Kommando, jedoch nach der Druckart n.
```

Damit man irrtümlich gesendete Editorkommandos rückgängig machen kann, wurde das Kommando u bereitgestellt. Es annulliert das zuletzt abgearbeitete Editorkommando. Allerdings kann sich seine Wirkung nur auf den Editorpuffer beziehen. Sollte eine falsch geänderte Datei mit dem Kommando w schon geschrieben worden sein, ist alles zu spät, wenn man nicht in der Arbeitsumgebung vorausschauend Vorsorge getroffen hatte. Das Kommando u ist mit den bekannten Druckkommandos kombinierbar.

```
u     Rücksetzen auf den Zustand vor dem letzten Kommando.
up    Leistet dasselbe, druckt die aktuelle Zeile.
ul    Druckart l.
un    Druckart n.
```

Zweckmäßig ist es, das Substitutionskommando mit Adressen auszustatten. So kann durch eine Adresse zuerst die zu ändernde Zeile gesucht werden. Wenn das Suchmuster für die Adressierung mit dem zu ersetzenden Text übereinstimmt, kann die Zeichenreihe muster verkürzt angegeben werden. Dies zeigen die beiden folgenden Kommandos:

```
/muster/s/muster/neu  Sucht Text muster und ersetzt ihn durch neu.
/muster/s//neu        Wie das vorige Kommando.
```

Außerdem kann mit zwei Adressen a1 und a2 ein Zeilenbereich fixiert werden, in dem dann Ersetzungen vorgenommen werden. Es werden alle Zeilen berücksichtigt, die das Muster alt enthalten. Wir wählen a1=1 und a2=$.

```
1,$s/alt/neu/    Ersetzt in allen Zeilen das erste alt durch neu.
1,$s/alt/neu/p   Wie vorher, druckt die zuletzt behandelte Zeile.
1,$s/alt/neu     Wie vorher.
,s/alt/neu       Wie vorher. 1,$ läßt sich als , abkürzen.
,s/alt/neu/g     Ersetzt in allen Zeilen alle Texte alt durch neu.
,s/alt/neu/gp    Wie vorher, druckt die zuletzt behandelte Zeile.
,s/alt/neu/2     Ersetzt in allen Zeilen das zweite alt durch neu.
,s/alt/neu/2p    Wie vorher, druckt die zuletzt behandelte Zeile.
```

Falls die zu ersetzende Zeichenreihe als Bestandteil der neuen verwendbar ist, lohnt es sich, von der Abkürzung & für den alten Text Gebrauch zu machen. Dies zeigen die nächsten Kommandos.

```
s/alt/ganz &  Identisch mit   s/alt/ganz alt/p
s/hui/& &     Identisch mit   s/hui/hui hui/p
s/Klammer/(&) Identisch mit   s/Klammer/(Klammer)/p
```

Das Zeichen <backslash> hebt die Bedeutung von Sonderzeichen auf. Das Kommando

```
s/und/\&
```

ersetzt das Wort und durch das Zeichen &. Die Sonderbedeutung des <backslash>-Zeichens wird durch Verdoppelung außer Kraft gesetzt. Mit

```
s/backslash/\\
```

wird das Wort backslash durch das Sonderzeichen <backslash> ersetzt. Auch der Punkt . ist ein Sonderzeichen. Soll in einem Text ein Punkt durch ein Ausrufungszeichen ersetzt werden, so leistet dies das Kommando:

```
s/\./!
```

Reguläre Ausdrücke; wie sie der Editor ed verarbeiten kann, heißen nach X/OPEN (1987) einfache reguläre Ausdrücke. Erweiterte reguläre Ausdrücke können durch die Programme lex, awk oder egrep verarbeitet werden.

Ein regulärer Ausdruck legt eine Menge von Zeichenketten fest. Eine Zeichenkette erfüllt den Ausdruck reg, wenn sie Element der Menge reg ist. Muster sind ebenfalls Mengen von Zeichenketten. Sie werden als reguläre Ausdrücke notiert. Zeilen in Textdateien werden durch Muster adressiert, wenn sie eine Zeichenkette enthalten, die das Muster als regulären Ausdruck erfüllt. Es folgt die Tafel 3.1. Sie stellt die Regeln für die Bildung regulärer Ausdrücke zusammen.

Tafel 3.1. Regeln zur Bildung regulärer Ausdrücke

| | |
|---|---|
| c | Bedeutet sich selbst, wenn c nicht eines der folgenden Sonderzeichen ist. |
| \c | Bedeutet c, wenn c ein Sonderzeichen ist. |
| ^ | Kennzeichnet den Anfang der Zeile. |
| $ | Kennzeichnet das Ende der Zeile. |
| . | Bedeutet ein beliebiges Zeichen. |
| [M] | Bedeutet ein Zeichen aus der nichtleeren Menge M. Die Menge M wird als Folge ihrer Elemente notiert. Folgen von Zeichen nach der ASCII-Kodierung können als Bereich c1-c2 angegeben werden. Das Zeichen ] muß als erstes in der Folge M stehen. Das Zeichen - kann als erstes oder letztes in M aufgestellt sein. Das Zeichen ^ kann irgendwo in M vorkommen. Wenn es am Anfang steht, hat es folgende Sonderbedeutung. |
| [^M] | Bedeutet ein beliebiges Zeichen, das nicht zu M gehört. |
| x* | Bedeutet null- oder mehrmaliges Auftreten des regulären Ausdrucks x. |
| xy | Bedeutet die Aufeinanderfolge oder Verkettung der beiden regulären Ausdrücke x und y. |
| x\{m,n\} | Bedeutet mehrmaliges Auftreten des regulären Ausdrucks x, mindestens m-mal, höchstens n-mal. |
| x\{m\} | Bedeutet genau m-maliges Auftreten von x. |
| x\{m,\} | Bedeutet mindestens m-maliges Auftreten. |
| \(x\) | Bedeutet den regulären Ausdruck x. Die Klammern \( und \) werden ignoriert. |
| \[1-9] | Sei n eine Zahl zwischen 1 und 9, dann bedeutet \n den vorangegangenen n-ten Ausdruck x, der in Klammern \( und \) eingeschlossen wurde. |

Die folgenden Beispiele zeigen einige reguläre Ausdrücke:

| | |
|---|---|
| /^$/ | Ist eine leere Zeile. Sie hat Anfang und Ende, sonst nichts, ist genau ein Zeichen lang (<newline>-Zeichen). |
| /./ | Eine Zeile mit wenigstens einem beliebigen von <newline> verschiedenen Zeichen. |
| /^/ | Eine beliebige Zeile. |
| /$/ | Ebenfalls eine beliebige Zeile. |

| | |
|---|---|
| /dies/ | Zeile, die dies enthält. |
| /^dies/ | Zeile beginnt mit dies. |
| /dies$/ | Zeile endet mit dies. |
| /^dies$/ | Zeile enthält nur dies. |
| /dies.$/ | Am Ende steht dies und noch ein Zeichen. |
| /dies\.$/ | Am Ende steht dies und ein Punkt. |
| /\/dies\// | Zeile, die /dies/ enthält. |
| /[dD]ies/ | Zeile, die dies oder Dies enthält. |
| /dies[0-9]/ | Auf dies folgt eine Ziffer. |
| /dies[^0-9]/ | Auf dies folgt ein Zeichen. Dieses Zeichen ist keine Ziffer. |
| /dies[0-9][^0-9]/ | Auf dies folgt eine Ziffer und ein anderes Zeichen. |
| /dies.*das/ | Auf dies folgen beliebige Zeichen und das, auch diesdas ist dabei. |

```
/dies *das/           Auf dies folgen beliebig viele Leerzeichen,
                      dann das. Wieder ist diesdas dabei.
/dies .*und .*das/    Nacheinander kommen dies und das vor.
                      Dazwischen stehen Leerzeichen und
                      beliebig viele andere Zeichen.
/^dies .*und .*das$/ Wie vorher, dies am Anfang, das am Ende.

/\(buchstaben\)/      Zeile enthält das Wort buchstaben.
/Gross\(buchstaben\) und Klein\1/
                      Sucht nach dem Text
                      Grossbuchstaben und Kleinbuchstaben.
```

Man kann Zeilen mit Kleinbuchstaben markieren und diese Marken dann in beliebigen anderen Kommandos als Adressen verwenden. Markiert man mit dem Buchstaben x, so bezeichnet die Adresse 'x die markierte Zeile. Seien reg1 und reg2 reguläre Ausdrücke.

```
/reg1/ka              Sucht die nächste Zeile, die reg1
                      enthält, und markiert sie mit a.
/reg2/kb              Sucht die nächste Zeile, die reg2
                      enthält, und markiert sie mit b.
'a,'bs/alt/neu/       Ersetzt in allen Zeilen von der
                      Marke 'a bis zur Marke 'b das erste
                      alt durch neu.
```

Die Kommandos g und v sind globale Kommandos. Sie laufen in zwei Schritten ab. Zuerst werden alle Zeilen markiert, die im Falle des Kommandos g ein Muster zum regulären Ausdruck reg enthalten oder die im Falle des Kommandos v kein Muster zum regulären Ausdruck reg enthalten. Anschließend wird für alle markierten Zeilen die Kommandofolge cmdlist abgearbeitet. Kommandolisten sind Kommandos oder Kommandolisten, auf die mit <backslash><newline> ein weiteres Kommando folgt.

```
m,ng/reg/cmdlist      Für alle Zeilen zwischen m und n
                      mit einem Muster zum Ausdruck reg
                      wird die Liste cmdlist abgearbeitet.
m,nv/reg/cmdlist      Für alle Zeilen zwischen m und n
                      ohne ein Muster zum Ausdruck reg
                      wird die Liste cmdlist abgearbeitet.
```

Im folgenden sind drei Kommandos cmd1, cmd2 und cmd3 als Liste in einem globalen Kommando zusammengefaßt, das auf die ganze Datei angewendet wird.

```
1,$g/reg/cmd1\
cmd2\
cmd3
```

Auch die Kommandos a, i oder c sind für die Listenbildung zugelassen. Die dazugehörigen Textzeilen werden mit <backslash><newline> abgeschlossen. Falls die Abschlußzeile .<newline> die Kommandoliste cmdlist beenden würde, kann sie wegbleiben. Es folgen einige Beispiele:

```
m,ng/reg/p  Drucke alle Zeilen zwischen m und n,
            die ein Muster von reg enthalten.
1,$g/reg/p  Drucke alle Zeilen mit einem Muster von reg.
g/reg/p     Ist mit dem vorigen Kommando identisch.
g/reg/d     Streiche alle Zeilen mit einem Muster von reg.
```

```
g/reg/s//neu/    Ersetze das erste Muster aus reg durch neu.
g/reg/s//neu/p   Wie eben und drucke alle betroffenen Zeilen.
g/reg/s//neu     Dies ist mit dem vorigen Kommando identisch.

/reg/            Suche nach einer Zeile mit einem Muster aus reg.
;g//s///g        Von hier bis zum Dateiende soll das Muster
                 jedesmal gelöscht werden.

g/reg/s//neu/gp      In der ganzen Datei sollen Muster aus reg
                     überall durch neu ersetzt werden.
                     Die betroffenen Zeilen werden gedruckt.
g/reg/s/alt/neu/     In Zeilen mit Mustern aus reg wird der
                     Text alt erstmals durch neu ersetzt.
g/reg/s/alt/neu/g    In Zeilen mit Mustern aus reg wird der
                     Text alt überall durch neu ersetzt.
g/reg/s/alt/neu/gp   Wie vorher, aber alle betroffenen Zeilen
                     werden gedruckt.
v/reg/s/alt/neu/gp   Wie vorher, aber alle die Zeilen sind jetzt
                     betroffen, die kein Muster aus reg haben.

v/^$/p           Dieses Kommando druckt alle nichtleeren Zeilen.
g/^$/d           Dieses Kommando löscht alle Leerzeilen.
```

Zu den Kommandos g und v gibt es die interaktiv arbeitenden Kommandos G und V. Sie haben die Form:

```
m,nG/reg/
m,nV/reg/
```

Auch hier werden im ersten Schritt alle angesprochenen Zeilen markiert. Danach werden der Reihe nach die markierten Zeilen zu aktuellen Zeilen erklärt, und der Nutzer wird aufgefordert, Kommandos einzugeben. Die Kommandos a, i und c sind dabei nicht zulässig. Nach der Abarbeitung des Kommandos wird die nächste markierte Zeile gedruckt, und der Nutzer kann ein dafür geeignetes Kommando eingeben. Eingabe des Zeichens <newline> bewirkt nichts und schaltet zur nächsten markierten Zeile weiter. Das vorangegangene Kommando kann durch Eingabe des Zeichens & abgekürzt wiederverwendet werden.

Die Kommandos G oder V sind zu Ende, wenn alle markierten Zeilen behandelt wurden. Vorher kann die Arbeit dieser interaktiven Kommandos durch ein Interruptsignal (<delete>-Taste) abgebrochen werden. Der Editor reagiert in diesem Falle mit einem Fragezeichen. Mit dem Kommando h kann die Begründung erfragt werden.

### 3.1.5. Zugriff auf Dateien

Mehrere Kommandos stellen Verbindungen zum Dateisystem her. Mit dem Kommando f kann der aktuelle Dateiname abgefragt oder gesetzt werden.

```
f            Der aktuelle Dateiname wird ausgegeben.
f file       Der Name file ist von nun an aktueller Dateiname.
```

Das Kommando w schreibt aus dem Editorpuffer heraus. Bei erfolgreichem Ablauf wird die Zahl der transportierten Byte mitgeteilt. Die aktuelle Zeile wird durch das Kommando w nicht verändert.

```
a1,a2w       Der adressierte Bereich wird als Datei mit dem
             aktuellen Dateinamen gespeichert. Eine eventuell
             vorhandene Datei gleichen Namens wird gelöscht.
```

| | |
|---|---|
| a1,a2w file | Es wird in die Datei mit dem Namen file geschrieben. Wenn es noch keinen aktuellen Dateinamen gibt, wird der Name file dafür eingesetzt. Sonst bleibt der aktuelle Dateiname erhalten. |
| a1,a2w !cmd | Dies führt zu einem Exkurs in den Kommandointerpreter. Der adressierte Bereich wird auf die Standardeingabe des Shell-Kommandos cmd gelenkt. |
| w | Diese Kommandos wirken wie die vorangehenden. |
| w file | Es wird aber der gesamte Inhalt des Editorpuffers ausgegeben. |
| w !cmd | |

Das Kommando r liest in den Editorpuffer hinein. Bei erfolgreichem Ablauf wird die zuletzt gelesene Zeile die neue aktuelle. Die Zahl der transportierten Byte wird mitgeteilt.

| | |
|---|---|
| .r file | Die Datei mit dem Namen file wird an die aktuelle Zeile angehängt. Falls es noch keinen aktuellen Dateinamen gibt, ist file von nun an aktueller Name. Sonst wird der aktuelle Dateiname nicht geändert. |
| .r | Die aktuelle Datei wird an die aktuelle Zeile angehängt. |
| .r !cmd | Dies führt zu einem Exkurs in den Kommandointerpreter. Die Standardausgabe des Shellkommandos cmd liefert die neuen Zeilen. |
| ar file | Diese Kommandos arbeiten wie die vorigen. |
| ar | Der gelesene Text wird jedoch an die |
| ar !cmd | Zeile mit der Adresse a angehängt. Die Adresse 0 ist zulässig. |
| r file | Diese Kommandos hängen an die Adresse $ an. |
| r | Unbedacht benutzt, widerspricht dies häufig |
| r !cmd | den Intentionen des Nutzers. |

Das Kommando e füllt den Editorpuffer neu. Der alte Inhalt wird überschrieben. Falls er sich während der vorherigen Editorarbeit geändert haben sollte, wird der Benutzer gewarnt. Nochmaliges Senden des Kommandos e überschreibt den Puffer wirklich. Die letzte Zeile des neuen Textes wird zur aktuellen. Die Zahl der gelesenen Byte wird mitgeteilt.

| | |
|---|---|
| e file | Die Datei mit dem Namen file kommt in den Editorpuffer. Der Name file wird zugleich als aktueller Dateiname gewählt. |
| e | Die aktuelle Datei kommt in den Puffer. |
| e !cmd | Dies führt zu einer Exkursion in den Kommandointerpreter. Die Ausgabe des Shell-Kommandos cmd füllt den Puffer des Editors. |

Das Kommando E leistet dasselbe wie das Kommando e, unterläßt jedoch die dort erwähnten Warnungen, falls der Puffer vorher geändert wurde.

### 3.1.6. Exkursionen zum Kommandointerpreter /bin/sh

Man kann den Editor kurzzeitig verlassen, indem man das Kommando ! absetzt. Damit erreicht man den Kommandointerpreter /bin/sh. Zeichen, die auf das Ausrufungszeichen folgen, werden als Kommandos interpretiert. Erscheint im Kommandotext ein Zeichen %, so wird es durch den aktuellen Dateinamen ersetzt. Beginnt der Kommandotext mit

einem Zeichen !, so wird dieses Zeichen durch den Kommandotext der vorangehenden Exkursion ersetzt. Das Editorkommando !! wiederholt also die vorige Exkursion ohne Änderung. Hier ein Beispiel:

```
% ed f.c                 Anlegen der Datei f.c.
?f.c
*a
main(){}
.
*w                       Transport ins Dateisystem.
9
*!ls -i f.c              Ausgabe des i-Attributs.
 1247 f.c
!
*!wc %                   % bezeichnet die aktuelle Datei.
wc f.c
       1       1       9 f.c
!
*!make f                 Aktivierung des C-Übersetzers.
    cc -O  f.c -o f
f.c
    rm -f f.o
!
*s/{}/{;}                Erweiterung des Programms.
main(){;}
*w                       Transport ins Dateisystem.
10
*!! -s                   Jetzt soll make schweigsam arbeiten.
make f -s
f.c
!
*q
```

Das Beispiel zeigt, wie mit sehr einfachen Mitteln abwechselnd mit dem Editor und einem Übersetzungssystem gearbeitet werden kann. Für die Lokalisierung und Beseitigung einfacher Quelltextfehler ist dies ein durchaus angemessenes Korrekturverfahren.

### 3.1.7. Umorganisieren innerhalb von Zeilen

Das Kommando j erlaubt es, mehrere Zeilen zu einer Zeile zusammenzufassen. Die resultierende Zeile wird zur neuen aktuellen Zeile erklärt.

```
a1,a2j      Die Zeilen von a1 bis a2 werden zu einer
            Zeile zusammengefaßt.
j           Die aktuelle und die folgende Zeile werden
            zusammengenommen.
.j          Dieses Kommando bewirkt gar nichts.
```

Anders herum muß man vielleicht Zeilen in aufeinanderfolgende Teilzeilen zerlegen. Das ist mit folgendem Kommando möglich.

```
s/ErsterTeilZweiterTeil/ErsterTeil\
ZweiterTeil/
```

Und noch ein weiteres Beispiel:

```
s/ /\       Dieses Kommando zerlegt die aktuelle Zeile.
/g          Jedes Wort kommt in eine Zeile für sich.
```

Tafel 3.2. Liste der Editorkommandos

| | |
|---|---|
| .a | Anhängen von Text, Abschluß mit einer Zeile .<newline>, maximale Zeilenlänge 256 Zeichen, Eingabemodus. |
| .c | Austausch des adressierten Textes durch den eingegebenen, Eingabemodus wie bei Kommando a. |
| .,.d | Streichen des adressierten Textes. |
| e file | Reinitialisierung des Editorpuffers. Falls der alte Puffer geändert wurde, erhält man eine Warnung. |
| E file | Reinitialisierung des Editorpuffers ohne Warnung. |
| f | Ausgabe des Namens der bearbeiteten Datei. |
| f file | Der Name file wird fortan als Dateiname verwendet. |
| 1,$g | Globale Anwendung der als Parameter gegebenen Kommandoliste auf alle Zeilen, die das angegebene Muster enthalten. |
| 1,$G | Das Globalkommando arbeitet interaktiv. Beendigung, falls die Datei ausgeschöpft ist, oder durch Interruptsignal. |
| h | Ausgabe eines Diagnostiktextes, falls Kommandos zurückgewiesen wurden oder ein Interruptsignal behandelt wurde. |
| H | Ein- oder Ausschalten des Diagnostikregimes. |
| .i | Einfügen von Text vor der aktuellen Zeile, Eingabemodus wie bei Kommando a. |
| .,.+1j | Zusammenfügen aufeinanderfolgender Zeilen zu einer Zeile. |
| .kx | Markierung einer Zeile mit der Marke x. Mit 'x kann die Marke x benutzt werden. |
| .,.l | Druckt Zeilen mit Berücksichtigung von Sonderzeichen. |
| .,.m | Transportiert Zeilen an eine andere Position. |
| .,.n | Druckt Zeilen mit ihren Nummern. |
| .,.p | Druckt Zeilen. |
| P | Ein- oder Ausschalten des Prompt-Regimes. Das Zeichen * dient als Prompt-Zeichen. |
| q | Das quit-Kommando beendet die Arbeit des Editors. Falls der Editorpuffer geändert wurde, erhält man eine Warnung. |
| Q | Beendigung der Editorarbeit ohne Warnung. |
| $r | Liest die bearbeitete Datei als Ganzes und hängt sie am Ende an den Puffer an. |
| $r file | Liest die angegebene Datei und hängt sie an der angegebenen Adresse an. |
| .,.s | Substitutionskommando. |
| .,.t | Kopiert Zeilen an eine angegebene Position. |
| u | Annullierung des vorher abgearbeiteten Kommandos. |
| 1,$v | Globale Anwendung der als Parameter gegebenen Kommandoliste auf alle Zeilen, die das angegebene Muster nicht enthalten. |
| 1,$V | Das Globalkommando arbeitet interaktiv wie das Kommando G. |
| 1,$w | Der adressierte Text oder der Editorpuffer ersetzt die bearbeitete Datei. |
| 1,$w file | Der adressierte Text oder der Editorpuffer wird als Datei mit dem Namen file geschrieben. Eine etwa vorhandene Datei wird überschrieben. |
| $= | Die Nummer der adressierten Zeile wird mitgeteilt. |
| !cmd | Das Kommando cmd wird dem Kommandointerpreter /bin/sh zur Bearbeitung übergeben. |
| .+1 | Eine Adresse allein bewirkt die Ausgabe der Zeile gemäß p. |

In der Tafel 3.2 sind die Editorkommandos alphabetisch aufgelistet. Die Adressen geben die voreingestellten Standardwerte an.

### 3.1.8. Kommandofolgen als Skriptdateien

Der Editor ed erwartet Kommandos von der Standardeingabe. Lenkt man diese auf eine Skriptdatei um, so kann man die Editoraktionen vorbereiten und wiederholt in der gleichen Reihenfolge ausführen lassen.

Zur Demonstration stellen wir uns die Aufgabe, das kleine C-Programm accu.c zu modifizieren. Die Eingabe- und Ausgabefunktionen scanf und printf sollen durch die Funktionen fscanf und fprintf ersetzt werden. Natürlich müssen zu diesem Zweck auch einige Deklarationen hinzugenommen werden. Das folgende Editorskript zeigt eine Lösung:

```
% cat edscript
0a
#include <stdio.h>
.
1a
FILE *out = stdout;
FILE *inp = stdin;
.
,s/printf(/fprintf(out,/
,s/scanf(/fscanf(inp,/
w accuf.c
q
```

Rufen wir den Editor ed mit dieser Skriptdatei auf, so entsteht die neue Datei accuf.c. Hier ist es angebracht, die Editoroption - zu fordern. Der Editor arbeitet jetzt ohne Ausgabeinformationen, solange er keine Fehler findet. Die folgenden Kommandos zeigen das Resultat der Editorarbeit und belegen zugleich die Funktionstüchtigkeit des neu hergestellten Programms accuf.

```
% ed - accu.c <edscript
% cat accuf.c
#include <stdio.h>
FILE *out = stdout;
FILE *inp = stdin;
main() {
int key,val;
int sum,totsum;
fprintf(out,"Bericht\n");
totsum = 0;
fscanf(inp,"%d",&key);
while (key == 1) {
        sum = 0;
        fscanf(inp,"%d",&key);
        while (key == 0) {
                fscanf(inp,"%d",&val);
                fprintf(out,"%10d\n",val);
                sum += val;
                fscanf(inp,"%d",&key);
        }
        totsum += sum;
        fprintf(out,"Gruppensumme%10d\n",sum);
}
fprintf(out,"Gesamtsumme %10d\n",totsum);
}
```

```
% make accuf
        cc -O  accuf.c -o accuf
accuf.c
        rm -f accuf.o
% accuf <accu.inp
Bericht
        10
        20

      usw.

        39
Gruppensumme         48
Gesamtsumme         200
```

### 3.1.9. Editorskriptdateien in Shell-Skripts

Zusätzliche Flexibilität läßt sich gewinnen, indem Skriptdateien für den Editor ed in Shell-Skripts integriert werden. Die Skriptdateien für den Editor liegen dann im Eingabestrom des Kommandointerpreters. Damit werden auch die Editorkommandos durch den Kommandointerpreter vorverarbeitet.

Zur Demonstration wird die Datei accuf.c, die im vorigen Abschnitt entstanden war, nochmals modifiziert. Die Pointervariablen inp und out werden durch die fopen-Funktion dynamisch initialisiert. Die Dateinamen werden mit dem Editorskript eingetragen. Als Dateinamen werden aber Bezeichnungen aus der Umgebung angegeben, die erst durch die Arbeit des Kommandointerpreters endgültig durch Dateinamen ersetzt werden. Es folgt das Shell-Skript. In der UNIX-Literatur werden Dateien im Eingabestrom auch als Hier-Dokumente bezeichnet. Daher soll das Shell-Skript herescript heißen.

```
% cat herescript
:   Bourne-Shell /bin/sh
PATH=:$HOME/bin:/bin:/usr/bin
ed - accuf.c <<end
/fprintf/i
out = fopen("$output","w");
inp = fopen("$input","r");
.
w accuff.c
q
end
cat accuff.c
make accuff
ls -l accuff
accuff
cat $outputt
```

Zuerst belegen wir die Umgebungsvariablen input und output mit den Dateinamen accu.inp und accu.res. Anschließend starten wir den Kommandointerpreter sh mit der Option -x und dem Shell-Skript herescript. Die Option -x bewirkt, daß alle Kommandos des Shell-Skripts so protokolliert werden, wie sie schließlich ausgeführt werden. Die Modifikationen, die nacheinander der Editor und der Kommandointerpreter durchgeführt haben, sind in den Druckprotokollen ablesbar. Die Abarbeitung des resultierenden Porgramms accuff soll wieder seine Funktionsfähigkeit belegen.

```
% setenv input accu.inp
% setenv output accu.res
% sh -x herescript

+ : Bourne-Shell /bin/sh

PATH=:/u/polze/bin:/bin:/usr/bin

+ ed - accuf.c
/fprintf/i
out = fopen("accu.res","w");
inp = fopen("accu.inp","r");
.
w accuff.c
q

+ cat accuff.c
#include <stdio.h>
FILE *out = stdout;
FILE *inp = stdin;
main() {
int key,val;
int sum,totsum;
out = fopen("accu.res","w");
inp = fopen("accu.inp","r");
fprintf(out,"Bericht\n");
totsum = 0;
fscanf(inp,"%d",&key);
while (key == 1) {

      usw. wie accuf.c

}
fprintf(out,"Gesamtsumme %10d\n",totsum);
}

+ make accuff
       cc -O  accuff.c -o accuff
accuff.c
       rm -f accuff.o

+ ls -l accuff
-rwxr-xr-x   1 polze    HUB        11114 Feb 16 10:55 accuff
+ accuff
+ cat accu.res
Bericht
        10
        20

      usw.

        39
Gruppensumme         48
Gesamtsumme         200
%
```

## 3.2. Der Stream-Editor sed

### 3.2.1. Arbeitsweise und Kommandos

Der Stream-Editor sed, McMahon (1978), bearbeitet Textdateien in einem Durchlauf. Zeile für Zeile wird zyklisch mit vorgegebenen Mustern verglichen und die zugeordneten Editorkommandos oder Folgen von Editorkommandos werden ausgeführt.

Der Aufruf von sed hat folgende Form:

```
sed [-n][-e script][-f scrfile] file1 file2
```

Die Dateien file1, file2 usw. werden nacheinander behandelt und im Ergebnis der Arbeit von sed in die Standardausgabedatei stdout transportiert. Wenn keine Dateinamen als Parameter angegeben sind, arbeitet sed als Filter und entnimmt den zu bearbeitenden Text der Standardeingabe.

Die Option -n unterdrückt die Ausgabe auf die Standardausgabedatei im Regelfall. Hier werden nur solche Zeilen auf die Standardausgabe gesandt, bei deren Bearbeitung dies ausdrücklich verlangt wurde.

Bei der Option -e steht script für eine Folge von Editorkommandos, eins je Zeile. Falls script mehrere Editorkommandos enthält, muß es in Apostrophe eingefaßt werden.

Die Option -f name besagt, daß eine Folge von Editorkomandos der Datei name entnommen werden soll.

Treten e-scripts oder f-Skriptdateien mehrfach auf, wird sed von der akkumulierten Gesamtfolge gesteuert.

```
sed -e script1 -f scrfile1 -f scrfile2 -e script2
```

arbeitet als Filter. Die Editorkommandos stehen der Reihe nach als script1 explizit im sed-Kommando, in den Dateien scrfile1 und scrfile2. Den Schluß bilden die explizit als script2 notierten Editorkommandos. Falls nur ein einziges Editorkommando cmd aktiviert werden soll, kann kurz

```
sed cmd file
```

geschrieben werden. Der Aufruf von sed behandelt die Datei file mit dem Kommando cmd. Als Beispiel für eine Kurzform beseitigt das Kommando

```
sed '/^$/d' file
```

alle Leerzeilen aus der Datei file und schreibt das Resultat auf die Standardausgabe. Die Apostrophierung verbirgt dem Kommandointerpreter das $-Zeichen.

Anders als der Editor ed verwaltet der Stream-Editor sed keinen Editorpuffer. Daher können Kommandos nur nacheinander angewendet werden, nacheinander im Sinne der Reihenfolge der Eingabezeilen. Edierkommandos und der zu edierende Text werden über die Zeilennummern synchronisiert. Rückgriffe sind unmöglich, Vorgriffe nur in einem sehr eingeschränkten Sinn, indem ganze Dateibereiche übersprungen werden können.

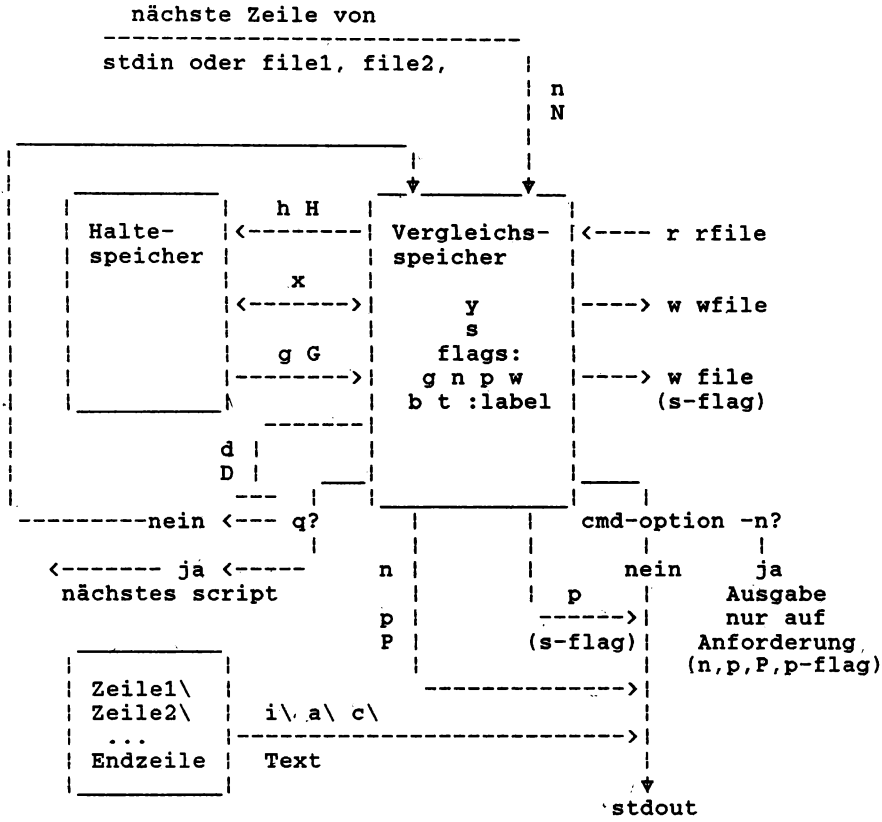



Bild 3.1. Arbeitsweise von sed

Zur Bearbeitung werden die Zeilen in einen Vergleichsspeicher gebracht (pattern space). Wenn sie in den Einflußbereich von Editorkommandos fallen, werden die entsprechenden Editorfunktionen ausgeführt. Anschließend kommen sie auf die Standardausgabe, falls nicht -n gewählt wurde, und der Vergleichsspeicher wird gelöscht (Ausnahme Kommando D). Zusätzlich zum Vergleichsspeicher verwaltet sed noch den Haltespeicher (hold space), der es gestattet, Textteile aufzubewahren. Bild 3.1 skizziert die Arbeitsweise des Stream-Editors.

Die Kommandos des Stream-Editors sed haben folgende Form:

```
[a1[,a2]]fct[arg]
```

Die Adressen a1 oder a2 sind als Dezimalzahlen Zeilennummern der Eingabedateien. Diese werden fortlaufend über alle angegebenen Dateien hinweg numeriert. $ bezeichnet die Nummer der letzten Zeile in der letzten Datei.

| | | |
|---|---|---|
| a1 = n | (1 ≤ n ≤ $) | Die Editorfunktion fct wird auf die Zeile n angewendet. |
| a1,a2 = m,n | (1 ≤ m ≤ n ≤ $) | Die Funktion fct wirkt auf die Zeilen m bis n einschließlich. |

Wenn die Zeile n den Vergleichsspeicher passiert hat, werden Editorkommandos mit Adressen ≤ n nicht mehr berücksichtigt.

Außerdem können als Muster auch reguläre Ausdrücke als Adressen a1 oder a2 angegeben werden. Zeilen, die das Muster erfüllen, werden gemäß der angegebenen Editorfunktion bearbeitet. Das sind Zeilen, die eine Zeichenkette enthalten, die Element des regulären Ausdrucks ist (s. Abschn. 3.1.).

Das Muster a1 eröffnet einen Dateibereich, wenn eine einkommende Zeile das Muster erfüllt. Ihre Zeilennummer charakterisiert die erste Zeile dieses Dateibereichs. Die zu diesem Kommando gehörende Editorfunktion wird auf Zeilen bis zur Nummer a2 einschließlich angewendet. Auch die Adresse a2 kann durch einen regulären Ausdruck festgelegt werden.

Nach dem Ende eines Bereichs a1,a2 steht das Muster a1 wieder zur Verfügung, um vielleicht weitere Bereiche zu eröffnen. Wenn alle Zeilen behandelt sind, ist die Arbeit von sed zu Ende. Einen zyklischen Neubeginn wie bei ed gibt es nicht.

Kommandos mit nur einer Adresse a1 adressieren jeweils eine Zeile. Kommandos ohne Adressen gelten für alle Zeilen.

Tafel 3.3. Liste der sed-Kommandos

---

Substitutionsfunktionen

---

s/reg/new/flags
: Ersetze Zeichenketten aus reg durch new. Erstesmal.
  Mögliche Flags:
  n : (1 ≤ n ≤ 512), ersetze beim n-ten Mal.
  g : Ersetze global.
  p : Drucke den Vergleichsspeicher, falls substituiert wurde.
  w wfile : Hänge den Vergleichsspeicher an die Datei wfile an, falls substituiert wurde.

y/string1/string2/
: Transformation aller Zeichen aus string1 durch entsprechende Zeichen aus string2. string1 und string2 sind gleich lang.

---

Einfügen von Text

---

Das Argument text besteht aus einer oder mehreren Zeilen.
Die Zeilen außer der letzten enden mit einem <backslash>-Zeichen.

a\
text
: Anhängen. Der Text geht nach stdout, bevor die nächste Zeile gelesen wird.

```
i\
text    Einfügen. Der Text geht nach stdout.
c\
text    Ändern. Lösche den Vergleichsspeicher. Übergehe weitere
        Zeilen des adressierten Dateibereichs. Text nach stdout.
```

Streichen von Zeilen, Drucken

```
d       Lösche den Vergleichsspeicher. Fortsetzung.
D       Lösche erste Zeile im Vergleichsspeicher. Fortsetzung.
n       Kopiere den Vergleichsspeicher nach stdout. Ersetze
        den Vergleichsspeicher durch die nächste Eingabezeile.
N       Anhängen der nächsten Eingabezeile an den Vergleichsspeicher.
p       Drucke. Kopiere den Vergleichsspeicher nach stdout.
P       Drucke die erste Zeile. Kopiere die erste Zeile des
        Vergleichsspeichers nach stdout.
l       Wie p, Sonderzeichen werden sichtbar.
```

Arbeit mit dem Haltespeicher

```
h       Ersetze den Haltespeicher durch den Vergleichsspeicher.
H       Anhängen des Vergleichsspeichers  an den Haltespeicher.
g       Ersetze den Vergleichsspeicher durch den Haltespeicher.
G       Anhängen des Haltespeichers an den Vergleichsspeicher.
x       Austausch der Inhalte von Vergleichs- und Haltespeicher.
```

Lesen und Schreiben von Dateien

```
r rfile     Lies die Datei rfile. Gib sie nach stdout,
            bevor die nächste Eingabezeile geholt wird.
w wfile     Anhängen des Vergleichsspeichers an die Datei wfile.
```

Steuerungsfunktionen

```
q           Beendigung von script.
{fct's}     Fasse die Funktionen fct's zu einer neuen Funktion
            zusammen.
!           Anwendung der Funktion nur auf Zeilen,
            die nicht zum adressierten Bereich gehören.
=           Ausgabe der Zeilennummer als Zeile.
:label      Markierung, sonst keine Wirkung.
b label     Überspringe Editorkommandos. Nächstes Kommando ist
            mit :label markiert. Gibt es :label nicht,
            springe an das Ende von script.
t label     Überspringe Kommandos bedingungsabhängig. Springe,
            wenn seit dem letzten Lesen einer Eingabezeile oder
            seit dem letzten t substituiert wurde.
```

### 3.2.2. Ein Beispiel

Wir kommen auf die schon angekündigte Möglichkeit zurück, das Ergebnis der Arbeit des Kommandos diff zum Vergleich zweier Dateien für die Steuerung des Stream-Editors sed aufzubereiten. Als Dateien verwenden wir die beiden C-Texte accu.c und accuff.c, die sich bereits am Ende des vorigen Abschnitts bei der Vorführung von Editorskripts als nützlich erwiesen haben.

Das ganze Ansinnen wäre durch folgende Vorstellung zu motivieren: Mehrere Versionen von Dateien unterscheiden sich in einer begrenzten Zahl von Zeichen oder Zeilen voneinander. Dann könnte es rentabel sein, eher eine Dateiversion aufzubewahren und ebenfalls alle Änderungen gegenüber der anderen Version als beide einander ähnlichen Dateien. Diese Vorstellung wird übrigens in dem Dateiverwaltungssystem SCCS (Source Code Control System) mit hoher Perfektion verwirklicht. Hier zunächst das Resultat des Kommandos diff.

```
% diff accu.c accuff.c >diffscript
% cat diffscript
0a1,3
> #include <stdio.h>
> FILE *out = stdout;
> FILE *inp = stdin;
4c7,9
< printf("Bericht\n");
---
> out = fopen("accu.res","w");
> inp = fopen("accu.inp","r");
> fprintf(out,"Bericht\n");
6c11
< scanf("%d",&key);
---
> fscanf(inp,"%d",&key);
9c14
<   scanf("%d",&key);
---
>   fscanf(inp,"%d",&key);
11,12c16,17
<           scanf("%d",&val);
<           printf("%10d\n",val);
---
>           fscanf(inp,"%d",&val);
>           fprintf(out,"%10d\n",val);
14c19
<           scanf("%d",&key);
---
>           fscanf(inp,"%d",&key);
17c22
<   printf("Gruppensumme%10d\n",sum);
---
>   fprintf(out,"Gruppensumme%10d\n",sum);
19c24
< printf("Gesamtsumme %10d\n",totsum);
---
> fprintf(out,"Gesamtsumme %10d\n",totsum);
```

Die Ausgabe von diff könnte als Vorlage für manuelles Edieren der Datei accu.c mit ed genutzt werden, so daß im Ergebnis accuff.c her-

gestellt würde. Diese Möglichkeit liegt der Wahl des Ausgabeformats des Kommandos diff zugrunde. Das folgende Terminalskript zeigt einen anderen Weg, der auf die Anwendung von sed gerichtet ist. Die soeben erzeugte Datei diffscript dient als Ausgangspunkt.

```
Script started on Fri Mar 11 10:49:22 1988
Neue C-Shell
1% unalias ed
2% ed - diffscript
g/^</d
g/---/d
v/^>/s/\([acd]\).*$/\1/
,s/\\/\\\\/
,s/$/\\/
v/^>/i\
>
1d
$a
>
.
,s/^> //
/^0a/s//1i/
,n
1   1i\
2   #include <stdio.h>\
3   FILE *out = stdout;\
4   FILE *inp = stdin;\
5
6   4c\
7   out = fopen("accu.res","w");\
8   inp = fopen("accu.inp","r");\
9   fprintf(out,"Bericht\\n");\
10
11  6c\
12  fscanf(inp,"%d",&key);\
13
14  9c\
15          fscanf(inp,"%d",&key);\
16
17  11,12c\
18                  fscanf(inp,"%d",&val);\
19                  fprintf(out,"%10d\\n",val);\
20
21  14c\
22                  fscanf(inp,"%d",&key);\
23
24  17c\
25          fprintf(out,"Gruppensumme%10d\\n",sum);\
26
27  19c\
28  fprintf(out,"Gesamtsumme %10d\\n",totsum);\
29
w sed.scr
q
3% sed -f sed.scr accu.c >f.c
4% sed '/^$/d' f.c >ff.c
5% cmp accuff.c ff.c
6% echo $status
0
7%
script done on Fri Mar 11 10:55:54 1988
```

Für die mehrfach einzufügenden Texte nach a\-, c\- oder i\-Kommandos wurden Endezeilen ohne <backslash>-Zeichen benötigt. Hier wurden kurzerhand Leerzeilen (zuerst Zeilen '<'-Zeichen, Leerzeichen, <newline>) einsortiert. Wenn die Dateien selbst Leerzeilen enthalten, müssen andere markante Schlußzeilen erfunden werden. Bemerkt sei außerdem, daß die in den Einfügetexten möglicherweise vorhandenen <backslash>-Zeichen verdoppelt werden müssen.

Die Schritte zur Aufbereitung des diff-Resultats könnten auch als Skriptdatei für ed formuliert werden.

```
% cat diff_ed.scr
g/^</d
g/---/d
v/^>/s/cd]).*$/1/
,s/\/\\/
,s/$/\/
v/^>/i    >
1d
$a
>
.
,s/^> //
/^0a/s//1i/
w sed.scr
q
```

Mit dieser Skriptdatei für ed könnten dann die folgenden Kommandos eingegeben werden. Bemerkt werden muß, daß die im Kommando 3% verwendete Datei sed.scr im Ergebnis des vorherigen Kommandos entsteht.

```
1% diff accu.c accuff.c >diffres
2% ed - diffres <diff_ed.scr
3% sed -f sed.scr accu.c >f.c
4% sed '/^$/d' f.c >ff.c
5% cmp accuff.c ff.c
6% echo $status
```

## 3.3. Bildschirmorientierte Editoren

Für die Eingabe umfangreicher Texte empfiehlt sich der bildschirmorientierte Editor vi. Sein Leistungsspektrum umfaßt die Möglichkeiten von ed. Die Bearbeitung des Textes spielt sich in einem Editorpuffer ab. Auf dem Bildschirm ist ein Ausschnitt des Puffers gewissermaßen als ein Fenster sichtbar. Zur Orientierung im Fenster können viele Kommandos angewendet werden, die den Tasten der Terminaleingabe zugeordnet sind. Insbesondere können die Kursorpositioniertasten ausgenutzt werden, um den Edierpunkt zu adressieren.

In der Arbeitsweise von vi unterscheidet man zwei Modi,

- den direkten Kommandomodus und
- den Zeilenkommandomodus.

Beim Start von vi wird der direkte Kommandomodus eingeschaltet. Hier werden eingegebene Zeichen als Kommandos interpretiert und unmittelbar verarbeitet. Die Eingabe des Zeichens : schaltet den Zeilenkommandomodus ein. In ihm werden nahezu alle Kommandos und weitere, wie sie auch ed kennt, in analoger Weise wirksam. Es können Zeichenketten gesucht, substituiert oder transportiert werden.

Die abzuarbeitenden Kommandos des Zeilenkommandomodus werden in der letzten Bildschirmzeile protokolliert. Im direkten Kommandomodus steht dort eine Statistikzeile. Das Erscheinungsbild am Terminal sieht so aus:

```
Die Kommandos i oder a schalten den Einfügemodus ein,
i vor der Kursorposition, a anschließend
an die Kursorposition.

Es können mehrere Textzeilen durch <newline> begrenzt
eingegeben werden. Die Esc-Taste (<escape>-Zeichen) beendet
den Einfügemodus.

Es ist sehr langweilig, alle vi-Kommandos der Reihe nach
zu beschreiben. Die Handhabe bildschirmorientierter Editoren
muß man sich als Fertigkeit im täglichen Umgang aneignen.
~
~
~
~
~
"demotxt" [new file]
```

Unbelegte Zeilen des Fensters werden durch die Tilde ~ angezeigt. Nicht akzeptierbare Zeichen werden durch vi hörbar zurückgewiesen. Die Positionierungsmöglichkeiten des vi sind wesentlich vielfältiger als die des ed. Es können Einzelzeichen, Wörter, Zeilen oder Zeilengruppen angesprochen werden. Für die Zwischenlagerung von Textteilen sind Puffer vorgesehen, die für die Umordnung in Dateien gute Dienste leisten. Eine hilfreiche Leistung von vi ist die Abkürzung von Editorskripts in der Form von Makrodefinitionen. Mehrfach sich wiederholende Editorkommandos oder Folgen von Editorkommandos können mit ihrem Makronamen aufgerufen werden.

Wie bei ed sind Exkursionen in den Kommandointerpreter möglich. Allerdings wird hier konsequenterweise derjenige Interpreter aktiviert, der durch die Umgebungsvariable SHELL festgelegt ist. (Der Editor ed leitet alle Exkursionen in die Bourne-Shell, ganz gleich, in welcher Interpreterumgebung er selbst aufgerufen wurde.)

Als Exkursionen sind wie bei ed die Abarbeitung von Kommandos :! cmd, die Ausgabe von Bereichen des Editorpuffers in die Standardeingabe von Kommandos :w! cmd und die Entgegennahme der Standardausgabe von Kommandos :r! cmd vorgesehen. Zusätzlich ist es möglich, Bereiche des Editorpuffers einem anderen Kommando als Eingabe anzubieten range:! . Dessen Ausgabe ersetzt dann den adressierten Bereich. So kann beispielsweise mit dem Kommando sort (s. Abschn. 2.1.) im Editorpuffer sortiert werden.

Bemerkenswert sind Schutzmechanismen von vi. Das Kommando :w [file] schreibt den Editorpuffer in die Datei file. Falls sie schon vorhanden ist, wird der Benutzer gewarnt, bevor die alte Datei überschrieben wird, eine segensreiche Einrichtung.

Manche Systeme heben den Texteditor ex hervor, der etwa dem Zeilenkommandomodus des vi entspricht, und dokumentieren den bildschirmgeprägten Teil und die Zeilenverarbeitung getrennt, so auch X/OPEN (1987). Im Testsystem XENIX sind vi und ex Dateibezeichnungen, die auf den gleichen Inode führen. Bemerkenswert ist aber der Größenvergleich der verschiedenen Editoren untereinander, wie im

Abschnitt 2.1. bei der Demonstration des Kommandos size sichtbar wurde. Der bildschirmorientierte Editor vi belegt fünfmal soviel Platz wie ed.

Man kann auf mehrfache Weise die Umgebung der vi-Arbeit gestalten. Ähnlich den Anfang- und Endeabläufen der C-Shell (Abschn. 4.1.3.) wird hier eine Datei .exrc wirksam. Ist sie im aktuellen Arbeitsverzeichnis oder im Heimatverzeichnis vorhanden, wird sie bei jedem Start von vi abgearbeitet. Eine Kommandofolge zur Editorinitialisierung kann weiterhin als Wert der Umgebungsvariablen EXINIT vorgegeben werden (s. Abschn. 4.3.). Alle diese Varianten sind dazu geeignet, sich eine mehr oder weniger sichere Editorumgebung zu präparieren. Beispielsweise kann man es so einrichten, daß generell nur in einer Dateikopie ediert wird, und erst wenn man des Erfolgs ganz sicher ist, wird das alte Original durch die neuere Version ersetzt. Mehrere Generationen von Altversionen können auf diese Weise gerettet, verwaltet und gepflegt werden.

Schließlich kann beim Aufruf von vi mit der Option +command bereits ein Editorkommando vorgelagert werden. Beispielsweise bewirkt die Angabe eines Suchmusters, daß das zuerst gezeigte Fenster mit der adressierten Zeile beginnt.

```
vi +/muster file  Die erste Zeile, die muster enthält, erscheint
                  in der ersten Fensterzeile.
vi +25 file       Zeile 25 der Datei file ist erste Fensterzeile.
```

Weitere Aufrufoptionen sind:

```
vi -R file  Die Datei file wird in keinem Fall überschrieben
            (Read-only-Arbeit).
view file   Ist mit dem vorigen Kommando identisch.
vi -r file  Wiederherstellung der Datei file
            im Störungsfall.
```

Wie breit das Leistungsspektrum von vi auch immer angesehen werden könnte, es hat niemals aktive Nutzer- oder Firmengruppen davon abgehalten, weitere Varianten bildschirmorientierter Editoren anzubieten. Und tatsächlich zeigt es sich immer wieder, daß Einfaches durch noch Einfacheres oder Bequemeres ersetzt werden kann. So muß es aussichtslos bleiben, auch nur dem Gedanken nachzuhängen, etwa den besten bildschirmorientierten Editor zu finden und diesen dann zu beschreiben.

# 4. Kommandointerpreter

## 4.1. Prinzipielles über csh und sh

Bereits die erste Begegnung mit UNIX bedarf der unaufdringlichen Mitwirkung eines Kommandointerpreters. Alle Beispiele der vorangegangenen Abschnitte machen fleißig von Kommandointerpretern Gebrauch. Die Interpreter nehmen vom Terminal Eingabezeilen entgegen, suchen darin nach dem Namen des abzuarbeitenden Programms, stellen die aufgeführten Parameter bereit und starten schließlich die beabsichtigte Programmabarbeitung.

Bisher dominierte der dialogfreundliche Kommandointerpreter csh. Einige Male, vor allem bei der Mitteilung von Kommandodateien, wurde der Standardinterpreter sh verwendet. Er wird sehr oft mit dem Namen seines Urhebers verknüpft und als »Bourne-Shell« bezeichnet (Bourne 1978). Zu den beiden Kommandointerpretern csh und sh gesellten sich seither bildschirmorientierte Shells, beispielsweise vsh, als Visual Shell bezeichnet, oder mview, Multi View, das mit mehreren Fenstern auf einem Bildschirm arbeiten kann. Bemerkenswert ist es, daß der Standardisierungsvorschlag X/OPEN (1987) konsequent nur von einem einzigen Kommandointerpreter spricht und damit die Bourne-Shell meint. Alle anderen Kommandointerpreter weisen mannigfache Systembindungen auf, so daß im Rahmen der Portabilitätsfrage auf den Interpreter sh Bezug genommen werden sollte. Aus diesem Grunde werden in diesem Abschnitt die beiden Kommandointerpreter sh und csh nebeneinandergestellt.

Beide Kommandointerpreter fordern durch die Ausgabe von Promptzeichen zur Eingabe des nächsten Kommandos auf. Diese Promptzeichen sind einstellbar. Für die Gegenüberstellungen in diesem Abschnitt werden die Standardprompts $ für sh und % für csh verwendet.

### 4.1.1. Aufbau von Kommandos

Einfache Kommandos sind Folgen von Wörtern, die durch Leerzeichen oder durch <tab>-Zeichen voneinander getrennt sind. Das erste Wort ist der Kommandoname. Es folgen Argumente. Die Abarbeitung eines Kommandos liefert einen Wert. Bei normaler Beendigung ist dies der Wert, der beim Verlassen des Programms über den exit()-Systemruf vermittelt wird. Dieser Wert heißt status. Bei abnormaler Beendigung eines Kommandos entsteht als Wert die Summe aus status und einer systemabhängigen Größe, oktal 0200 (X/OPEN), dezimal 1000 (XENIX86) oder oktal 01000 (XENIX286).

Einfache Kommandos des Interpreters sh sind auch Ein-/Ausgabe-Umlenkungen und Zuweisungen an Variable. Im Anhang 3 werden Kommandos der Bourne-Shell syntaktisch beschrieben.

Kommandos werden aus einfachen Kommandos aggregiert. Unterschiedliche Verknüpfungsoperatoren stehen zur Verfügung.

cmd1 ; cmd2
: Sequentielle Ausführung. Das Kommando cmd2 wird gestartet, wenn die Arbeit von cmd1 beendet ist.

cmd &
: Asynchrone Ausführung. Für die Abarbeitung des Kommandos cmd wird ein neuer Prozeß gebildet. Seine Nummer wird bekanntgegeben. Die Arbeit am Terminal kann unmittelbar fortgesetzt werden.

cmd1 && cmd2
Kommando cmd2 wird nur dann gestartet, wenn cmd1 erfolgreich abgearbeitet werden konnte.

cmd1 || cmd2
Kommando cmd2 wird nur dann gestartet, wenn cmd2 nicht erfolgreich abgearbeitet werden konnte.

cmd1 | cmd2
Die Kommandos cmd1 und cmd2 bilden ein Pipe. Die Standardausgabe von cmd1 wird mit der Standardeingabe von cmd2 verbunden. Die beiden Kommandos werden in separaten Prozessen abgearbeitet. Beide Prozesse werden durch das UNIX-Betriebssystem synchronisiert. Der Kommandointerpreter wartet auf die Beendigung des letzten Kommandos. Als exit-Wert wird derjenige des letzten Kommandos genommen.

( cmd ) - nur sh -
Startet eine neue Instanz des Interpreters sh, die das Kommando cmd entschlüsselt und startet.

{ cmd[;] } - nur sh -
Das Kommando cmd wird als Unterprogramm des Kommandointerpreters gestartet, ohne dafür einen neuen Prozeß zu erzeugen. Die Zeichen { und } werden als Wörter und als solche als Kommandonamen verstanden. Daher sind Leerzeichen verbindlich. Das Semikolon kann entfallen, wenn } am Beginn der neuen Zeile steht.

name(){ cmd[;] } - nur sh
Definiert eine Shell-Funktion. Sie wird unter der Bezeichnung name erreicht und als Unterprogramm des Kommandointerpreters gestartet. { cmd } ist der Funktionskörper. Folgen dem Aufruf name Argumente, so werden diese als Positionsparameter $1, $2, usw. zur Verfügung gestellt. Auch hier müssen die Klammern { } wie im vorangegangenen Fall abgetrennt werden.

Die Verknüpfungsoperatoren wurden in der Reihenfolge ihrer Priorität angeordnet. Die Operatoren ; und & haben gleiche Priorität, ebenso die beiden Operatoren && und ||. Wenn durch Verknüpfungen Kommandos entstehen, die mehrere Zeilen füllen, so kann durch \<newline> in der nächsten Zeile fortgesetzt werden. Ein derart geschütztes <newline>-Zeichen ist wirkungslos.

Bis hierher konnten beide Kommandointerpreter csh und sh nahezu gemeinsam abgehandelt werden. Aber schon die folgenden Bemerkungen signalisieren erhebliche und bedenkliche Differenzen. Die Kommandos cmd1 && cmd2 und cmd1 || cmd2 benötigten zu ihrer Erklärung den Terminus »erfolgreiche« Abarbeitung.

Zur Demonstration verwenden wir die Kommandos true und false, deren schlichte Leistung darin besteht, einen bestimmten exit-Wert zu liefern. Die beiden kleinen, nachträglich kommentierten Skripts zeigen dies.

```
% true    echo $status      # Die C-Shell stellt den exit-Wert
0                           # in einer Variablen namens status
% false   echo $status      # bereit.
255                         # Mit $status ist diese Variable
%                           # ansprechbar.
```

```
$ true    echo $?            # Die Bourne-Shell liefert
0                            # den exit-Wert in der
$ false ; echo $?            # Variablen $?
255
$
```

Die Kommandos true und false leisten also in beiden Kommandointerpretern dasselbe. Die Bourne-Shell versteht eine Kommandoabarbeitung als erfolgreich, wenn der exit-Wert 0 geliefert wird. Das wird im folgenden Skript demonstriert.

```
$ true  && echo 'ja \c'   echo ende
ja ende
$ false && echo 'ja \c'   echo ende
ende
$ true  || echo 'nein \c'   echo ende
ende
$ false || echo 'nein \c'   echo ende
nein ende
$
```

Dagegen interpretiert die C-Shell den exit-Wert im Sinne der Programmiersprache C. Dort steht der Wert 0 für den logischen Wert false und jeder von 0 verschiedene Wert für den logischen Wert true. Das führt zum folgenden, dem vorangegangenen total entgegengesetzten Resultat.

```
1% true  && echo 'ja \c'   echo ende
ende
2% false && echo 'ja \c'   echo ende
ja ende
3% true  || echo 'nein \c'   echo ende
nein ende
4% false || echo 'nein \c'   echo ende
ende
5%
```

Ebenso nachhaltig wirkt sich dieser Effekt auch im Verständnis der in beiden Interpretern vorhandenen Kommandos zur Ablaufsteuerung aus. Mit den Möglichkeiten der Bourne-Shell sieht das so aus:

```
$ if true
> then echo ja
> else echo nein
> fi
ja
$ if false
> then echo ja
> else echo nein
> fi
nein
$
```

Um mit der C-Shell ähnliches zu zeigen, muß man schon Kommandodateien bilden, für die solche Steuerkommandos auch erst ihren eigentlichen Sinn erhalten. Darauf kommen wir im Abschnitt 4.4. zurück.

```
% cat true.scr
#
if (`true;echo $status`) then
  echo ja
else
  echo nein
endif
% true.scr
nein
%
```

Die Ablaufkommandos der C-Shell werden durch Ausdrücke gesteuert. Deshalb steht dort eine aufwendigere Konstruktion, um zu dem exit-Wert als Wert eines Ausdrucks zu kommen. Wie die C-Shell das Dilemma zu beheben gestattet, zeigt die modifizierte Kommandodatei true1.scr.

```
% cat true1.scr
#
if { true } then
  echo ja
else
  echo nein
endif
% true1.scr
ja
%
```

In der C-Shell ist die Konstruktion { cmd } ein Ausdruck mit dem Wert 1, wenn cmd den exit-Wert 0 liefert. Der Ausdruck hat den Wert 0, wenn cmd einen von 0 verschiedenen exit-Wert erzeugt. Ganz nebenbei erkennt man hier auch den Grund dafür, daß in der obigen Aufzählung von Kommandoverknüpfungen am Ende Einschränkungen auf die Bourne-Shell notwendig waren.

Bemerkt sei, daß das Kommando false bei VENIX den Wert 1 liefert. Bourne-Shell und C-Shell verhalten sich bei interaktivem Betrieb gleichwertig. Kommandodateien zeigen dagegen gleichfalls die hier berichteten Differenzen.

### 4.1.2. Variable und Ersetzungen

Naturgemäß benötigen die Kommandointerpreter ein gehöriges Ensemble von Daten zur Verwaltung ihrer eigenen Arbeit. Die Benutzer können individuell weitere Variable definieren und bei Bedarf darauf Bezug nehmen. Die Namen werden aus Buchstaben (einschließlich <underscore>-Zeichen) und Ziffern gebildet. Die maximale Länge ist systemabhängig, beispielsweise ≤ 20. Variable haben Zeichenketten als Werte. Dabei trennen Leerzeichen oder <tab>-Zeichen Wörter voneinander ab. Oft werden Variable zur abgekürzten Notation mehrfach wiederkehrender Zeichenketten benutzt. Wenn im Eingabetext von csh oder sh

```
$name        oder
${name}
```

erscheinen, so wird name als Bezeichnung einer Variablen gedeutet und durch die zugehörige Zeichenkette ersetzt, falls die Variable für den Interpreter erklärt wurde. Auch Umgebungsvariable werden in dieser Form angesprochen. Kann keine Definition von name gefunden werden, erscheint eine Fehlernachricht. Die zweite Form gestattet es, Variable in den fortlaufenden Text eines Wortes einzufügen.

Variable erhalten durch folgende Definitionskommandos Werte:

% set name
Die csh-Variable name wird definiert und hat die leere Zeichenkette als Wert.

% set name = word
Die Zeichenkette word wird als Wert der csh-Variablen name festgehalten.

% set name = (wordlist)
Eine Liste von Wörtern muß in Klammern eingefaßt werden. Die Klammern sind im entstehenden Wert von name nicht enthalten.

% set name[index] = word
Ein vorhandenes Wort mit der Nummer index ≥0 kann getrennt vom Rest der Zeichenkette ersetzt werden.

$ name=value
Die sh-Variable name erhält die Zeichenkette value als Wert.

Unterschiedlich für die beiden Kommandointerpreter gibt es im Umgang mit Variablen weitere Ersetzungsvarianten. Das folgende gilt für die C-Shell. Sie hat ihren eigenen Reichtum an Ausdrucksformen.

$#name
${#name}
Liefert die Anzahl der Wörter im Wert von name.

$0
Gibt den Namen der Datei, von der Kommandos gelesen werden. Kommen Kommandos vom Terminal, so heißt es »No file for $0«.

$zahl
${zahl}
Identisch mit $argv[zahl]

$*
Identisch mit $argv[*]

Die Substitution von Variablen kann durch angehängte Modifikatoren beeinflußt werden. Von Dateinamen können Suffixe entfernt werden. Bei Pfadnamen können Pfadpräfix oder der Dateiname am Schluß extrahiert werden. Die dafür zuständigen Modifikatoren r, t und h werden im Abschnitt 4.2.1. über History-Substitutionen beim Terminaldialog mit der C-Shell erörtert.

Weitere Sonderformen der Variablensubstitution in der C-Shell, die allerdings keine modifizierenden Anhänge zulassen, sind die folgenden:

$?name
${?name}
Liefert 1, wenn name definiert ist, sonst 0.

$?0
Liefert 1, wenn die Kommandoquelldatei bekannt ist, sonst 0. (Wenn Kommandos vom Terminal kommen, entsteht 0.)

$$ Prozeßnummer der aktuellen C-Shell.

Die Bourne-Shell bietet dem Nutzer andere Möglichkeiten an.

${name:-value}
Liefert den Wert der Variablen name, falls es sie gibt und ihr Wert nicht die leere Zeichenkette ist. Andernfalls wird value substituiert, ohne dadurch eine Variable name zu definieren.

${name:=value}
Wie vorher, jedoch wird die Variable name mit dem Wert value definiert, wenn es sie bisher nicht gegeben hatte. Falls sie leer war, erhält sie jetzt value als Wert.
${name:?text}
${name:?}
Wie vorher, jedoch, wenn es die Variable nicht gibt oder wenn sie die leere Zeichenkette als Wert hat, wird die Ausschrift text erzeugt, und die Bourne-Shell wird verlassen. Fehlt der Zusatz text, so wird die Ausschrift »parameter null or not set« generiert.
${name:+value}
Wenn es die Variable name gibt und ihr Wert ist nicht leer, wird value substituiert. Andernfalls wird nichts substituiert.

Die soeben aufgezählten Möglichkeiten gibt es in Versionen ohne den Doppelpunkt : als Trennzeichen. Dann entfällt überall die Frage danach, ob Werte leer sind. ${name-value} ersetzt ${name} nur dann durch value, wenn es keine Variable name gibt. Gegebenenfalls werden also auch leere Zeichenketten substituiert.

Überdies werden im Sprachgebrauch der Bourne-Shell Variable auch als Parameter bezeichnet. Parameter, die durch explizite Wertzuweisungen der Form name=value entstanden sind, heißen Schlüsselwortparameter. Argumente von Kommandos werden als Variable der Form $n ($0≤n≤9) berücksichtigt. Sie heißen Positionsparameter. Die Variablen $* und $@ haben die Liste aller Argumente als Werte. Es gilt:

"$*" bedeutet "$1 $2 ...
"$@" bedeutet "$1" "$2"

Die Bourne-Shell verwaltet die Parameter $#, $-, $?, $$ und $! als Sondervariable mit den folgenden Bedeutungen:

$# Zahl der Positionsparameter (dezimal).
$- Aufrufoptionen der Bourne-Shell oder durch set erzeugte Optionen.
$? Ergebniswert des letzten Kommandos.
$$ Prozeßnummer der aktuellen Shell.
$! Prozeßnummer des zuletzt asynchron gestarteten Kommandos.

Die beiden Kommandointerpreter melden sich mit Promptzeichen, die als Variable definiert sind und daher auch von den Benutzern individuell gewählt werden können. Die Bourne-Shell gestattet sogar individuelle Trennzeichen zwischen den einzelnen Wörtern.

prompt - csh -
Standardprompt ist das Zeichen %.
PS1 - sh -
Primäres Promptzeichen ist $.
PS2 - sh -
Sekundäres Promptzeichen ist >.
IFS - sh -
Feldseparator, voreingestellt: Leerzeichen, <tab>, <newline>.

Beide Kommandointerpreter csh und sh erzeugen Dateinamen aus Kurzbezeichnungen.

```
*    Steht für beliebige Zeichenketten, auch für leere.
?    Steht für ein einzelnes Zeichen.
[...]
     Steht für eins der aufgezählten Zeichen.
[c-d]
     Steht für ein Zeichen aus dem Intervall c bis d.
[!c]        - nur sh -
     Steht für ein Zeichen ungleich c.
{v,w}       - nur csh -
     Erzeugt zwei Texte, einen mit dem Substituten+v und einen
     mit w.
            - nur csh -
     Steht für die Umgebungsvariable HOME.
~uname      - nur csh -
     Steht für die Umgebungsvariable HOME des Nutzers uname.
```

Ebenso ist in beiden Kommandointerpretern csh und sh eine Substitution durch Kommandoaufrufe vorgesehen. Der Text

```
`cmd`
```

wird durch die Produktion der Standardausgabe des Kommandos cmd ersetzt. Normalerweise wird der Ergebnistext bei Leerzeichen, <tab>-Zeichen und <newline>-Zeichen in Wörter zerlegt.

Die Flut von Zeichen mit Sonderbedeutungen läßt erwarten, daß Vorkehrungen für die normale Verwendung der Zeichen getroffen werden müssen. Das geschieht auf unterschiedlichem Niveau. Das Fluchtsymbol \ (<backslash>-Zeichen) hebt die etwaige Bedeutung des folgenden Zeichens auf. Einschluß von Zeichenketten in einfache Apostrophzeichen verhindert jegliche Interpretation von Sonderzeichen. Der Einschluß in Doppelapostrophe erlaubt die Ersetzung von Variablen und die Kommandosubstitution. Andere Sonderzeichen haben keine Wirkung. Das Fluchtsymbol \ schützt $ und `.

### 4.1.3. Anfangs- und Endeabläufe

Jeder Aufruf des Kommandointerpreters csh führt zuerst zu der Frage, ob es im Heimatverzeichnis des Benutzers eine Datei mit dem Namen .cshrc gibt. Wenn sie als abarbeitbare Datei vorhanden ist, wird sie gestartet. Damit sind individuelle Initialisierungen der Kommandointerpreter möglich. Gewöhnlich werden Variable gesetzt und Alias-Bezeichnungen festgelegt. Letztere dienen der kürzeren Niederschrift oft benötigter Kommandos. Sie werden im Abschnitt über die Dialogarbeit mit der C-Shell erklärt. Als Beispiel folgen Auszüge aus einer .cshrc-Datei.

Enthält das Heimatverzeichnis keine Datei mit dem Namen .cshrc, so wird in einigen UNIX-Systemen die Datei /etc/cshrc gestartet.

```
set noclobber                    # '>' soll nicht überschreiben
set history=20                   # 20 Kommandos aufheben
if ($?prompt) then
    set prompt = (\!%\ )         # Prompttext wählen
endif
alias print 'pr -n \!:* | lpr'   # Druckkommando
alias ed 'ed -p "*"'
alias h 'history | tail -10'
alias H history
alias m more
alias p 'pr -172 \!* | pf >/dev/lp1'
```

```
alias P 'pr -t \!* | pf >/dev/lp1'
alias N 'nroff -rL72 -rW69 -mm'
alias NR 'NRoff \!* &'
alias pg 'pg -n'
echo Neue C-Shell
```

Bei der Aufnahme von Interessenten in den Kreis der Nutzer eines UNIX-Systems wird festgelegt, welcher Kommandointerpreter angeboten werden soll, wenn sich der Nutzer am Terminal meldet. Dieser wird im folgenden als login-Shell bezeichnet. Ist eine C-Shell zugleich login-Shell, so vollzieht sie nach der Abarbeitung von ~/.cshrc einen weiteren Initialisierungsschritt. Wenn es im Verzeichnis ~ eine abarbeitbare Datei .login gibt, wird sie gestartet. Es folgt ein Auszug aus einer Datei .login, wie sie für eine reale Arbeit präpariert sein könnte.

```
set ignoreeof                       # Ctrl-d führt nicht zu logout
set path = (. $home/bin /bin /usr/bin)  # Suchpfad
setenv      MAIL    /usr/spool/mail/`logname`
set noglob
set term = \
(`tset -r -S -Q -m ansi:ansi -m wy60:wy60 -m: \?ansi`)
setenv      TERM    $term[1]
setenv      TERMCAP $term[2]    # Terminalbeschreibung
unset term noglob
umask 022                           # Dateierzeugungmaske
setcolor hi_white                   # Individuelle Farben
setcolor -b lt_blue                 # Hintergrundfarbe
setcolor -r yellow magenta          # Umkehrfarben
echo Neues login
```

Wenn die csh-Variable ignoreeof definiert ist, kann die Arbeit von csh nicht durch end of file und am Terminal nicht versehentlich durch Ctrl-d abgebrochen werden. Statt dessen muß das Kommando logout explizit aufgerufen werden, um die Tätigkeit einer login-Shell zu beenden. Das Kommando logout kann nur login-Shells beenden.

Bei der Ausführung eines Kommandos stehen mehrere Verzeichnisse zur Auswahl, in denen das zu startende Kommando enthalten sein könnte. Die csh-Variable path zählt die Verzeichnisse auf und ordnet sie in der Reihenfolge an, in der in den Verzeichnissen nach dem abzuarbeitenden Kommando gesucht werden soll.

Die csh-Variable term bekommt zwei Wörter als Wert. Das erste Wort ist die Bezeichnung des Terminaltyps. Es wird der Umgebungsvariablen TERM zugewiesen. Das zweite Wort ist die Beschreibung der Manipulationsmöglichkeiten des Terminals. Es wird zum Wert der Umgebungsvariablen TERMCAP erklärt. Das hier benutzte System verfügt über eine Konsole nach dem ANSI-Standard und über WYSE-Terminals. Das Kommando tset ermittelt den Terminaltyp und stellt die Zeichenkette für die csh-Variable term zusammen. Nachdem die Umgebungsvariablen belegt sind, wird die csh-Variable term nicht mehr benötigt. Ebenso die csh-Variable noglob. Sie verhinderte Fehldeutungen des byteweise verschlüsselten TERMCAP-Texts, indem sie mögliche Interpretationen als Dateinamen ausschaltete.

Beim Verlassen einer login-Shell durch das Kommando logout oder durch Ctrl-d, falls die csh-Variable ignoreeof nicht gesetzt ist, wird als Aufräumungskommando ~/.logout gestartet, insofern es vorhanden ist. Darin könnten folgende Kommandos vorkommen:

```
echo 'Soll eine Sicherheitskopie hergestellt werden? (y) \c'
if { y_question } then
                # y_question gibt exit-Wert 0 bei Antwort 'y'
  echo '\'tar 2u priv/book/cont\' gestartet'
  tar 2u priv/book/cont
else
  echo Es wurde keine Kopie angefertigt!
endif
setcolor -n     # Rücksetzen der individuellen Farben
echo Adieu!
```

Bei der Bourne-Shell ist der Komfort für die Anfangs- oder Endegestaltung beträchtlich bescheidener. Hier kann man im Heimatverzeichnis die Datei .profile anlegen. Sie wird gestartet, wenn die Bourne-Shell als login-Shell fungiert. Auch hierfür ein Beispiel:

```
:
PATH=:$HOME/bin:/bin:/usr/bin           # Suchpfad
MAIL=/usr/spool/mail/`logname`          # Datei für mail-Texte
eval `tset -r -s -Q`
umask 022                               # Dateimaske
export PATH MAIL
m() more $*                             # Shell-Funktion
echo Neue Bourne-Shell
```

Zwei sh-Variable werden belegt. Das Terminal wird erkundet, und die Dateierzeugungsmaske wird mit 022 fixiert. Das export-Kommando sorgt dafür, daß die beiden sh-Variablen PATH und MAIL als Umgebungsvariable behandelt werden.

Es folgt die Definition einer Shell-Funktion. In dieser Shell kann m als Abkürzung für more benutzt werden. Dies entspricht den Alias-Abkürzungen in der C-Shell. Wenn jedoch eine Nachfolge-Shell gestartet wird, muß more als Bezeichnung wieder ausgeschrieben werden. Nachfolge-Shells enststehen beispielsweise bei Exkursionen aus den Editoren ed oder vi, aus dem Debugger adb, bei der Arbeit von make und bei weiteren Gelegenheiten. Namen von Shell-Funktionen werden nicht an Nachfolgeprozesse vererbt. An dieser Stelle schneidet also die Bourne-Shell im Vergleich mit der C-Shell unglücklich ab.

### 4.1.4. Ausführung von Kommandos

Bevor ein Kommando mit seiner Arbeit beginnt, werden folgende Schritte durchlaufen:

(1) Erledigung aller Substitutionen.

(2) Wenn das Kommando als Unterprogramm des Interpreters vorhanden ist, wird dieses aktiviert. Die so nutzbaren Programme werden auch als Spezialkommandos (Built-in-Kommandos) bezeichnet.

(3) - nur sh -
Wenn das Kommando als Shell-Funktion identifiziert werden kann, wird es im gleichen Prozeß, in dem der Interpreter selbst läuft, abgearbeitet.

(4) Es wird ein Nachfolgeprozeß eingerichtet. Dieser übernimmt die Abarbeitung des Kommandos. Er verliert seine Existenz, wenn das Kommando zu Ende gekommen ist, sei es erfolgreich oder durch einen vorzeitigen Unfall.

Wenn nach der Datei mit dem Kommandonamen gesucht werden muß, die das abzuarbeitende Programm enthält, helfen die beiden Variablen

```
path        ( - csh - )      und
PATH        ( - sh - ).
```

PATH ist zugleich Umgebungsvariable. Werte von path bzw. PATH sind Listen von Verzeichnissen. In ihnen wird nacheinander nach dem gefragten Dateinamen gesucht.

```
% echo $path
. /u/polze/bin /bin /usr/bin
% echo $PATH
:/u/polze/bin:/bin:/usr/bin
```

Die Trennzeichen unterscheiden sich. Ansonsten werden die Variablen durch die C-Shell abgestimmt verwaltet. Änderungen von PATH ändern zugleich path und umgekehrt. In der Bourne-Shell gibt es nur die Variable PATH für die Fixierung der Suchreihenfolge. Bei PATH bedeutet ein leerer Eintrag das Arbeitsverzeichnis, bei path muß das Arbeitsverzeichnis mit seinem Namen explizit angegeben werden.

```
% echo $path ; echo $PATH
. /u/polze/bin /bin /usr/bin
:/u/polze/bin:/bin:/usr/bin
% set path = (/bin /usr/bin $HOME/bin .)
% echo $path ; echo $PATH
/bin /usr/bin /u/polze/bin .
/bin:/usr/bin:/u/polze/bin:
% setenv PATH ":$PATH/etc"    # siehe Fußnote 1)
% echo $path ; echo $PATH
. /bin /usr/bin /u/polze/bin /etc
:/bin:/usr/bin:/u/polze/bin:/etc
```

Wenn im Kommandonamen der Schrägstrich / als Trennzeichen für Pfadnamen vorkommt, insbesondere wenn absolute Pfadnamen vorliegen, werden die Variablen path bzw. PATH nicht benutzt.

Die Kommandointerpreter sammeln oder berechnen die Positionen der erreichbaren Kommandos. Dazu dienen die Spezialkommandos:

```
$ hash [-r][name     ]     ( - sh - )
% rehash               '    ( - csh - )
```

Die Bourne-Shell sammelt Positionen aufgerufener Kommandos. Sie vergißt sie wieder, wenn die Variable PATH ihren Wert ändert und ebenfalls durch den Aufruf von $ hash -r. Das Kommando $ hash name ... berechnet die Positionen der angegeben Dateien. Diese müssen ausführbar sein. Ohne Parameter berichtet $ hash alle gesammelten Positionen und schreibt die Anzahl der Aufrufe und die Zahl der Suchschritte aus.

---

1) Meine C-Shell tut sich schwer bei der Ersetzung von Umgebungsvariablen. % echo ${HOME} ${PATH}. Daher sind in diesen Beispielen nicht beliebige Kombinationen vorführbar. (Der Autor.)

Zur Demonstration folgt ein Beispiel:

```
$ PATH=/bin:/usr/bin::/u/polze/bin:/etc
$ csh -f true.scr
nein
$ hash ed make mount accu
$ hash
hits    cost    command
1       1       /bin/csh
0       1       /bin/ed
0*      5       /etc/mount
0       1       /bin/make
0*      3       accu
$ hash -r
$ hash
hits    cost    command
$
```

Die C-Shell berechnet bei ihrem Start eine Hash-Tabelle für alle erreichbaren Kommandonamen. Der Aufruf von % rehash bewirkt erneut die Berechnung dieser Tabelle.

Weitere Spezialkommandos sind für die Ablaufsteuerung vorgesehen. Sie werden im Abschnitt 4.4. zusammengestellt, da sie vor allem für die Formulierung von Kommandodateien wichtig sind. Hier sollen noch einige Spezialkommandos mitgeteilt werden, die, wie bemerkt, als Unterprogramme der Interpreter realisiert sind.

: - sh, csh -
Nullkommando, ohne Wirkung. Der exit-Wert ist 0. Am Anfang von Kommandodateien dient es als Hinweis dafür, daß die Datei durch die Bourne-Shell interpretiert werden muß.

\# - csh -
Nullkommando ohne Wirkung. Am Anfang von Kommandodateien dient es als Hinweis dafür, daß die Datei durch die C-Shell interpretiert werden muß. Des weiteren kennzeichnet es in Kommandodateien den folgenden Zeilenrest als Kommentar. Bei der interaktiven Arbeit kennt die C-Shell keine Kommentare.

\# - sh -
Kommentar. Der Zeilenrest wird nicht interpretiert.

. name - sh -
source name - csh -
Die Datei name enthält Kommandos. Sie werden interpretiert.

exec cmd - sh, csh -
Das Kommando cmd wird im Prozeß des Kommandointerpreters gestartet.

exit [n] - sh, csh -
Das Kommando beendet die Arbeit des Interpreters. Der exit-Wert ist n. Wenn nichts angegeben wurde, wird der exit-Wert des zuletzt interpretierten Kommandos genommen. Die C-Shell akzeptiert für n Ausdrücke.

eval [arg ... ] - sh -
Argumente werden interpretiert. Die daraus entstehende Zeichenkette wird als Kommando gestartet.

read [name ... ] - sh -
Eine Zeile wird gelesen. Die Variable name erhält das erste Wort als Wert. Wenn nicht gerade end of file erkannt wurde, ist der exit-Wert 0.

```
set -v              - sh -
set verbose         - csh -
    Der Interpreter protokolliert fortan Eingaben, wie er sie
    vorfindet.
set -x              - sh -
set echo            - csh -
    Der Interpreter protokolliert Kommandos mit ihren Argumenten,
    so wie er sie abarbeitet, gegebenenfalls nach Substitutionen.

set -k              - sh -
    Schlüsselwortparameter werden in die Umgebung aufgenommen,
    auch wenn sie als Argumente nach dem Kommandonamen
    erscheinen.
setenv name value   - csh -    ( ohne '='-Zeichen! )
    Die Umgebungsvariable name erhält den Wert value.  Falls es
    sie noch nicht gibt, wird sie hierdurch definiert.
times               - sh -
time                - csh -
time cmd            - csh -
    Ohne Argumente wird die bisher durch den Interpreter kon-
    sumierte Zeit ausgegeben.  Mit angehängtem Kommando cmd wird
    die durch dieses Kommando benötigte Zeit mitgeteilt.
```

Hier ein Beispiel für die Leistung von time:

```
% time nroff -rL72 -rW69 -mm lit.mm > litu
21.3u 4.1s 0:27 94%
% time
208.8u 81.4s 2:23:06 3%
```

Die Zahlen sind CPU-Zeiten in Sekunden für den Nutzer (u), für das System (s) Wartezeit am Terminal und prozentuale Auslastung.

## 4.2. Dialog und Umlenkungen

### 4.2.1. Terminaldialog mit der C-Shell

Die C-Shell bietet einige vorteilhafte Möglichkeiten für den Dialog am Terminal an. Bei einer entsprechenden Anfangsbelegung ihrer Variablen speichert sie die zuletzt eingegebenen Kommandos. Man kann darauf zurückgreifen. Dabei sind mannigfache Modifikationen möglich.

Ein Ausrufungszeichen vermittelt den Einstieg in den sogenannten History-Mechanismus. Andere Benutzungen des Ausrufungszeichens müssen durch das Fluchtsymbol \ geschützt werden. Auch wenn auf ein Ausrufungszeichen ein Leerzeichen folgt, verliert es seine Rolle als Bezugssymbol auf die Vorgeschichte. Zunächst sei auf das Kommando

```
% history
```

hingewiesen. Es druckt die augenblicklich noch bekannten Kommandos aus. Die Zahl der jeweils gespeicherten Kommandos wird als Wert der C-Shell-Variablen history gefunden.

Kommandozeiger verweisen auf gespeicherte Kommandozeilen. Mit Hilfe von Wortzeigern können einzelne Wörter eines Kommandos angesprochen werden. Angehängte Modifikatoren gestatten es, die ausgesuchten Wörter zu verändern.

Kommandozeiger sind absolute oder relative Nummern oder Zeichenketten, die eine Zeile adressieren.

```
!!  Voriges Kommando.
!#  Laufendes Kommando.
!n  Kommando mit der Nummer n.
!-n Vom laufenden Kommando beginnend n Kommandos zurückgezählt.
!letter
    Letztes Kommando, das mit dem Buchstaben letter begann.
!?string?
    Letztes Kommando, das als Muster string enthielt.
```

Wortzeiger verweisen in ein bereits lokalisiertes Kommando. Sie haben die Form

```
!c[:]w
```

mit dem Kommandozeiger c und dem Zeiger w. Bleibt der Kommandozeiger c weg, so beziehen sich Wortzeiger auf das vorige Kommando.

```
!c^    Erstes Argument (nach dem Kommandonamen).
!c$    Letztes Argument.
!c*    Alle Argumente.
!c-y   Wörter 0 bis y.
!c%    Wort, das zum vorangegangenen ?string? gehörte.

!c:0   Nulltes Wort, Kommandoname.
!c:n   Wort Nummer n.
!c:x-y Wörter x bis y.
!c:x*  Wörter x bis $.
!c:x-  Wie :x* aber ohne das Wort $.
```

Folgende Modifikatoren sind vorgesehen:

```
:s/alt/neu[/]
    Die Zeichenkette alt (nicht reguläre Ausdrücke) wird durch
    den Text neu ersetzt. Im Text neu bedeutet & den Text alt.
    Die Form :s//neu ersetzt das vorige alt oder das ?string?
    aus dem Kommandozeiger. Andere Begrenzer als / sind
    zulässig, der letzte kann entfallen, wenn <newline> folgt.
:gs/alt/neu[/]
    Ersetzt alt durch neu in allen Wörtern des ausgewählten Kom-
    mandos.
:p  Druckt das modifizierte Kommando, ohne es zu starten.
:t  Entfernt führende Pfadnamenkomponenten (path_prefix),
    (pref/fname -> fname).
:h  Entfernt die letzte Pfadnamenkomponente
    (pref/fname -> pref).
:r  Entfernt das letzte Suffix
    (name.xxx -> name).
```

Die Modifikatoren :t, :h und :r können auch als Anhänge bei der Ersetzung von csh-Variablen benutzt werden. Dort gibt es zudem noch einige Varianten.

```
:g[thr]
    Die Modifikationen werden in allen Wörtern der Zeichenkette
    vorgenommen, die als Wert der Variablen vorliegt.
:q  Bei der Substitution verlieren Sonderzeichen ihre Bedeutung
    (wie beim Einschluß in Quote-Zeichen).
:x  Wie :q, aber der Text wird wortweise geschützt.
```

Beim Substitutionsmodifikator :s erleichert die Abkürzung

für !:s^ und damit also

^alt^neu für !:s^alt^neu

die Anwendung dieser oft benötigten Substitution. Des weiteren hebt die Klammerung { } History-Substitutionen von darauf folgenden Zeichen ab.

Mit einigen Beispielen sollen History-Bezüge demonstriert werden.

Am Ende einer Terminalsitzung müssen Sicherungskopien der geleisteten Arbeit angefertigt werden, und zwar zwei Stück, weil besonders wertvolle Taten vollbracht wurden. Das erste Kommando muß explizit formuliert werden. Nach dem Einlegen der zweiten Diskette kann man sich sehr kurz fassen.

```
% tar cvf /dev/fd096ds15          # Erste  Diskette
% !!                              # Zweite Diskette
```

Berücksichtigt man die Möglichkeiten des Abschnitts 2.4., so kann das Ganze mit

```
% tar c2                    # Erste  Diskette
% !!                        # Zweite Diskette
```

noch kürzer ausgedrückt werden.

Bei der Arbeit am Manuskript dieses Abschnitts wiederholt sich folgender Ablauf.

```
% cd ~/priv/book/cont
            # Einstellen des Arbeitsverzeichnisses
% ed ch42.mu
            # Arbeit am Text von Abschnitt 4.2.
% upr <ch42.mu >ch42.mm
            # Behandlung der Ersatztexte für Umlaute
            # und Sonderzeichen
% nroff -mm -rL72 -rW69 ch4.mm >ch4u
            # Textaufbereitung Abschnitt 4.
% upo <ch4u >ch4uu
            # Ersetzen der von nroff erzeugten Spezial-
            # konstrukte durch druckbare Zeichen
```

Das folgende Terminalskript zeigt Bezugnahmen auf die History-Verwaltung.

```
Script started on Tue Jun  7 11:36:02 1988
Neue C-Shell
1% cd ~/priv/book/cont
2% ed ch42.mu
11856

     usw.

*q
3% upr <ch41.mu >! ch41.mm
4% upr <ch42.mu >! ch42.mm
```

```
5% upr <ch43.mu >! ch43.mm
6% upr <ch4.mu >! ch4.mm
7% nroff -rL72 -rW69 -mm ch4.mm >! ch4u
8% upo <ch4u >! ch4uu
```

Bis hierher handelt es sich um herkömmliche Kommandoaufrufe. Die Eingabe des Kommandos history reproduziert den vorangegangenen Ablauf. Danach wird der History-Apparat vorgeführt.

```
9% history
     1      cd ~/priv/book/cont
     2      ed ch42.mu
     3      upr < ch41.mu > ! ch41.mm
     4      upr < ch42.mu > ! ch42.mm
     5      upr < ch43.mu > ! ch43.mm
     6      upr < ch4.mu > ! ch4.mm
     7      nroff -rL72 -rW69 -mm ch4.mm > ! ch4u
     8      upo < ch4u > ! ch4uu
     9      history
10% !1
cd ~/priv/book/cont
11% !-3
upo < ch4u > ! ch4uu
12% !?upo?
upo < ch4u > ! ch4uu
13% !1:1:p
~/priv/book/cont
14% !1:$:p
~/priv/book/cont
15% !1:1:h:p
~/priv/book
16% !1:1:t:p
cont
17% cd {!1:1:h}/cont
cd {~/priv/book}/cont
18% pwd
/u/polze/priv/book/cont
```

Nun geben wir die anfangs vorgestellte Kommandofolge noch einmal an und benutzen History-Substitutionen.

```
19% !1
cd ~/priv/book/cont
20% ed ch42.mu
11856

     usw.

*q
21% upr < !$ >! !$:r.mm
upr < ch42.mu > ! ch42.mm
22% !!:gs/2/1
upr < ch41.mu > ! ch41.mm
23% !-2:gs/2/3
upr < ch43.mu > ! ch43.mm
24% !u:gs/3/
upr < ch4.mu > ! ch4.mm
25% nroff -rL72 -rW69 -mm !$ >! !$:ru
nroff -rL72 -rW69 -mm ch4.mm > ! ch4u
26% upo < !$ >! !$u
upo < ch4u > ! ch4uu
```

Und hier noch einige beachtenswerte Details der History-Substitution bei Kommandos.

```
27% ^uu^vv
upo < ch4u > ! ch4vv
28% !u:0:p •
upo
29% !27:1:p
<
30% !27:2:p
ch4u
31% !27:3:p
>
32% !27:4:p
!
33% !27:5:p
ch4vv
```

Die folgenden Skriptzeilen zeigen die Wirkung von Modifikatoren bei der Ersetzung von csh-Variablen.

```
34% cd ~/priv/Book/Content
35% upr < ch42.mu > ! ch42.mm
36% set v=(!35:*)
set v= ( < ch42.mu > ! ch42.mm )
37% echo $v
< ch42.mu > ! ch42.mm
38% echo $v:r
< ch42 > ! ch42.mm
39% echo $v:gr
< ch42 > ! ch42
40% cat ch42.mu

       Text

41% !40:s/mu/$zzz/
cat ch42.$zzz
zzz: Undefined variable.
42% !40:s/mu/$zzz/:q
cat ch42.$zzz
cat: cannot open ch42.$zzz
43%
script done on Tue Jun  7 11:56:55 1988
```

Der Kommandointerpreter C-Shell bietet eine weitere Unterstützung des Dialogs am Terminal an, indem er Abkürzungen für häufig verwendete Kommandos zuläßt. Dies leisten die sogenannten Alias-Ersetzungen. Möge man sich daran stören, wenn man immer den »langen« Namen history schreiben muß, um den eben erörterten History-Apparat in Bewegung zu setzen. Mit

```
% alias H history
```

erreicht man, daß der Buchstabe H allein schon genügt, um das Kommando history zu starten.

Ruft man history oder nunmehr H auf, so hat man anschließend den ganzen Bildschirm voller zurückliegender Information. Die eigentliche Arbeit, mit der man gerade beschäftigt war, ist schon nicht mehr zu sehen. Dagegen hilft es, wenn man ein weiteres Kommando einrichtet,

das vielleicht nur die letzten 10 Zeilen der History-Information sichtbar macht. Dafür bilden wir einen anderen Alias-Namen mit

```
% alias h 'history | tail -10'      oder
% alias h 'H | tail -10'
```

Dem kann man entgegenhalten, man hätte nur der csh-Variablen history den Wert 10 geben sollen. Was aber, wenn man manchmal doch weiter in die Vorgeschichte zurückblicken muß? Hier wurde ein Kompromiß gefunden, indem man wahlweise H oder h aufrufen kann.

Bei der Interpretation einer Kommandozeile verfährt die C-Shell in folgenden Schritten: Zuerst wird das Kommando in einfache Kommandos zerlegt. Diese werden einzeln behandelt. Wenn das erste Wort eines einfachen Kommandos eine Alias-Bezeichnung ist, wird der zugehörige Alias-Text noch einmal gelesen. Dabei werden History-Ersetzungen vorgenommen, so als wäre dieses zu behandelnde einfache Kommando das zuletzt interpretierte.

Als weitere Beispiele für Alias-Abkürzungen nutzen wir wieder solche aus dem Manuskriptentwurf dieses Buches.

```
% alias N 'nroff -rL72 -rW69 -mm'
% alias Nr 'N \!^.mm | tee \!^u | upo >! \!^uu'
```

Nehmen wir an, die Dateien ch4*.mm seien schon vorhanden, beginnen wir also nach der Behandlung der Sonderzeichen. Dann bleibt der Aufruf

```
% Nr ch4
```

übrig. Nr ist eine Alias-Bezeichnung. Dazu gehört der eben angegebene Text. Jedesmal wird !^ durch ch4, das erste (und einzige) Argument aus dem Kommando % Nr ch4, ersetzt. Beim Lesen des Alias-Textes findet der Interpreter drei einfache Kommandos vor, die einzeln zu behandeln sind. Das erste davon führt erneut auf einen Alias-Namen. Der dazu gehörende Text nimmt nicht mehr auf Früheres Bezug. Er wird ungeändert eingefügt.

Das Kommando tee kopiert die Standardeingabe in die Standardausgabe und fertigt eine Zweitkopie an, die in der als Parameter angegebenen Datei abgelegt wird.

```
% alias p 'pr -l72 \!* | pf >/dev/lp1'
% alias P 'pr -t \!* | pf >/dev/lp1'
```

Mit p und P sind weitere Alias-Namen eingeführt worden. Sie umgehen die Druckpufferverwaltung und führen direkt auf das Gerät /dev/lp1. Dazwischen liegt noch ein privater Filter pf. Das Kommando p besorgt die Auslastung der 72 Zeilen des gewöhnlich verfügbaren Endlosformularpapiers. Das Kommando P druckt bereits 72zeilig vorgefertigte Texte ohne die Standardkopfleiste des pr-Kommandos.

Die zwei folgenden Alias-Bildungen zeigen Modifikationen von Dienstprogrammrufen, wenn bestimmte Optionen regelmäßig oder sogar ausschließlich gewünscht werden.

```
% alias ed 'ed -p "*"'
% alias pg 'pg -n'
```

Mit dem Kommando unalias können Alias-Ersetzungen rückgängig gemacht werden. Die Kommandos

```
% unalias ed
% unalias pg
```

stellen die originale Form der beiden Dienstprogramme wieder her. Will man die augenblicklich gültigen Alias-Bezeichnungen wissen, so kann man durch das Kommando

```
% alias
```

ohne Parameter die Liste aller eingeführten Alias-Namen erhalten.

### 4.2.2. Ein-/Ausgabe-Umlenkung

Die C-Shell sieht ausschließlich die Umlenkung der Standarddateien stdin und stdout sowie eine Sonderbehandlung von stderr vor.

`< name`
: Die Datei name wird eröffnet und mit stdin identifiziert.

`<< word`
: Der folgende Text wird als inline-Datei in der C-Shell lokal gesammelt und der Standardeingabe zugeführt, wenn das entprechende Kommando gestartet wird (wird auch als Hier-Dokument bezeichnet). Eine Zeile, die genau aus der Zeichenreihe word besteht, beendet den Text. Im Text werden Variable ersetzt. Kommandoaufrufe `cmd` werden ausgeführt. Dabei können die Zeichen $, \ und ` durch das Fluchtsymbol \ geschützt werden. Zuvor wird jedoch die Endezeile word gesucht. Wenn die Zeichenreihe word Zeichen \ , ", ' oder ` enthält, so werden keine Ersetzungen vorgenommen.

Bei der Ausgabe wird die Umlenkung durch die C-Shell-Variable noclobber beeinfluβt.

`> name`
`>! name`
`>& name`
`>&! name`
: Die Datei name wird mit sdtout identifiziert. Wenn es sie noch nicht gibt, wird sie eingerichtet. Wenn die Variable noclobber definiert ist, stoppt die Interpretation des Kommandos, wenn es die Datei name schon gibt. Sonst wird eine bestehende Datei überschrieben. Enthält der Umlenkungsoperator das Zeichen !, so entfällt dieser Test. Zeichenweise organisierte Gerätedateien unterliegen diesem Test nicht.
  Mit dem Zeichen & im Umlenkungsoperator werden die Standardausgabedateien stdout und stderr beide in die Datei name gelenkt.

`|&`
: In dieser Kombination wird der gemischte Datenstrom von stdout und stderr an die Standardeingabe stdin des auf den Pipe-Operator folgenden Kommandos weitergereicht.

`>> name`
`>>! name`
`>>& name`
`>>&! name`
: Hier wird die Standardausgabedatei stdout oder beide Standardausgabedateien stdout und stderr zusammen an die Datei name angehängt. Wenn die Variable noclobber definiert ist, stoppt die

Kommandointerpretation, falls es die Datei name noch nicht gibt. Wiederum entfällt dieser Test, wenn der Umlenkungsoperator das Zeichen ! enthält.

Die Bourne-Shell organisiert die Ein-/Ausgabe-Umlenkung ein bißchen anders. Eine Variable namens noclobber hat hier keinen Einfluß. Dafür lassen sich alle programmintern benutzten Dateien, auch diejenigen, die zu Deskriptornummern n>2 gehören, voneinander unabhängig auf vorgegebene Dateinamen umlenken. Zunächst ähnelt die Umlenkung dem Vorangegangenen.

< name
Die Datei name wird mit stdin (Deskriptor 0) identifiziert.

> name
Die Datei name wird mit stdout (Deskriptor 1) identifiziert. Gibt es die Datei nicht, so wird sie eingerichtet. Eine bestehende Datei wird überschrieben.

>> name
Die Datei name wird mit stdout (Deskriptor 1) identifiziert. Die Ausgabe wird an die Datei name angehängt. Wenn es die Datei name noch nicht gibt, wird sie eingerichtet.

<< word
<<- word
Der folgende Text wird in der Bourne-Shell lokal als inline-Datei gesammelt. Die Zeichenreihe word in einer separaten Zeile beendet den Text. Enthält der Umlenkungsoperator das Zeichen -, so werden im Text und in der Endezeile führende <tab>-Zeichen ignoriert. Die Textzeilen werden interpretiert, es finden Textersetzungen statt. Die Zeichen \<newline> werden ignoriert. Das Zeichen \ schützt die Zeichen \, $, ' sowie das erste Zeichen von word. Eine Ausnahme bildet der Fall, wenn in der Zeichenreihe word Zeichen geschützt wurden. Dann finden keine Textersetzungen statt.

< &m
> &m
Der Deskriptor m wird dupliziert. Die Umlenkung führt auf die Kopie des Deskriptors m.

< &-
> &-
Die umzulenkende programminterne Datei wird abgeschlossen.

Allgemeiner ist die Umlenkung nach folgenden Mustern zulässig.

```
n uop name
n uop &m
```

lenkt die programminterne Datei, die durch den Deskriptor n ($0 \le n \le 9$) gesteuert wird, gemäß dem Umlenkungsoperator uop auf die Datei name oder auf die Kopie des Deskriptors m um.

Häufig findet man in der UNIX-Literatur das Umlenkungsbeispiel

```
$ cmd >name 2>&1
```

mit dieser Erklärung: Der Deskriptor 1 wird dupliziert. Auf die Kopie wird der Fehlerausgang stderr umgelenkt. Vor dem Duplizieren wurde der Deskriptor 1 mit der Datei name verbunden. Also erscheinen die Regelausgaben und die Fehlerausgaben des Programms cmd zusammengemischt in der Datei name. Sie erscheinen tatsächlich alle in derselben Datei name. Da aber normalerweise die Ausgabe in das Dateisystem in Blöcken zu je 512 Bytes vonstatten geht, wechseln also »Blöcke«

von Regelausgaben und »Blöcke« von gesammelten Fehlernachrichten einander ab. Das steht gewiß nur selten im Einklang mit den Absichten des Nutzers.

Es folgt ein Beispielskript für die Eingabe von Hier-Dokumenten mit den Möglichkeiten der Bourne-Shell. Die Abläufe erklären sich selbst.

```
Script started on Fri Jun  3 11:42:38 1988
$ cat << eof
> abcdefghij
> $HOME
> `pwd`
> eof
abcdefghij
/u/polze
/u/polze/priv/book/cont/T42
$ cat << eof
> \\escape
> \$HOME
> \`pwd\`
> \eof
> eof
\escape
$HOME
`pwd`
eof
$ cat << \eof
> \eof
> $HOME
> `pwd`
> abcdefgij
> eof
\eof
$HOME
`pwd`
abcdefgij
$
script done on Fri Jun  3 11:47:36 1988
```

Die Bourne-Shell sh hat eine weitere bemerkenswerte Eigenschaft. Die mit der Dateiumlenkung verbundenen Aktivitäten laufen unabhängig vom eventuell später folgenden Start des interpretierten Kommandos ab. Die Angabe eines Kommandos ist für den Ablauf von Umlenkungsmaßnahmen gar nicht erforderlich. So führt die folgende Eingabezeile

```
$ > name
```

dazu, daß eine leere Datei name eingerichtet wird. Das ist die einfachste Art, eine Datei anzulegen. Dies ist aber nur mit sh möglich. Auch die Reienfolge von Umlenkungen und Kommandoaufruf ist willkürlich. So kopieren all drei Kommandos

```
$ cat <eingabe >ausgabe
$ <eingabe cat >ausgabe
$ <eingabe >ausgabe cat
```

die Datei eingabe in die Datei ausgabe.

Beide Kommandointerpreter csh und sh lenken bei asynchron gestarteten Kommandos

```
cmd &
```

die Standard-Ein-/Ausgabe-Dateien stdin und stdout auf die spezielle Datei /dev/null um. Als Gerätedatei kann /dev/null beliebig oft angesprochen werden. Die Diagnosedatei stderr ist von dieser Regelung nicht betroffen.

### 4.2.3. Shell-Funktionen

Wir kommen dazu noch einmal auf die Umlenkungsoperatoren der Bourne-Shell zurück. Die separate Umlenkung der Dateideskriptoren 0-9 ist von Belang, wenn Programme verschiedene Datenströme erzeugen, ohne daß dafür programminterne Dateinamen festgelegt werden sollen. Bei der Beschreibung des Terminaldialogs mit der C-Shell hatte die nroff-Textverarbeitung geeignete Beispiele geboten. Hier bietet sie Anlaß, gegen die C-Shell zu argumentieren.

Das Programm nroff verwendet den Dateideskriptor 2 nicht nur zur Ausgabe von Fehlernachrichten, sondern auch zur Abspaltung von Texten von dem zu verfertigenden Dokument. Dies läßt sich gut ausnutzen, um neben der Textaufbereitung gekennzeichnete Schlüsselwörter mit ihren Standortangaben automatisch abzuzweigen, um sie danach zu sortieren und als Sachwortverzeichnis zu sammeln.

Mit der Bourne-Shell leistet dies das Kommando

```
$ nroff -rL72 -rW69 -mm ch4.mm 2> ch4.key > ch4u
```

oder als Pipe verknüpft:

```
$ nroff -rL72 -rW69 -mm ch4.mm 2> ch4.key \
> | tee ch4u | upo > ch4uu
```

Die C-Shell würde solche Kommandos mit der Ausschrift »Ambiguous output redirect« zurückweisen, weil sie annähme, der Dateideskriptor 1 sollte zweimal, und zwar auf unterschiedliche Dateien, umgelenkt werden.

Damit entzieht sich diese Niederschrift des nroff-Aufrufs dem Alias-Apparat der C-Shell. Will man trotzdem eine Abkürzung des unhandlich langen Kommandotextes haben, bleibt nur die Formulierung einer kleinen Kommandodatei, etwa in der folgenden Weise, oder die Definition einer Shell-Funktion. Zuerst zeigen wir eine Kommandodatei, die aus der C-Shell heraus aufgerufen werden kann.

```
% cat NRoff
:       /bin/sh
upr <$1.mm  \
| nroff -rL72 -rW69 -mm 2>$1.key  \
| tee $1u   \
| upo > $1uu
echo NRoff fertig >/dev/tty
% NRoff ch4
```

Der Parameter ch4 ersetzt den Text $1 überall in der Datei NRoff. Das anfangs stehende Nullkommando : erzwingt die Verarbeitung der Datei durch die Bourne-Shell.

Nun folgt noch die Notation als Shell-Funktion NRoff(). Sie ruft intern die ebenfalls zu definierende Funktion N() auf. Das alles kann nur innerhalb der Bourne-Shell aufgerufen werden.

```
$ N() { nroff -rL72 -rW69 -mm $*; }
$ NRoff() {
> upr < $1.mm \
> | N 2> $1.key \
> | tee $1u \
> | upo > $1uu;
> echo NRoff fertig > /dev/tty
> }
$ NRoff ch4
```

Shell-Funktionen werden benutzbar, wenn sh ihre Definition abarbeitet. Man könnte fragen, ob es zweckmäßig sei, die Definitionen in die Datei $HOME/.profile zu schreiben. Das würde der Vorgehensweise bei den Alias-Definitionen der C-Shell gleichen, deren Unterbringung in der Datei ~/.cshrc angemessen erschien. Nun wird aber die Datei $HOME/.profile nur abgearbeitet, wenn die Bourne-Shell auch login-Shell ist. Auch in Nachfolge-Shells stehen die Funktionen dann nicht zur Verfügung.

Als Empfehlung könnte man eine Datei .func einrichten, die alle Funktionsdefinitionen enthält. Bei jedem Start einer Nachfolge-Shell müßte man zu Beginn die Datei .func abarbeiten, beispielsweise die folgende.

```
m() more $*
M() m $*
Ed() { ed -p "*" $; }
Pg() {
     pg -n $*
     }
```

Es müssen neue Namen gewählt werden. Das Beipiel zeigt zugleich verschiedene Notationsformen, die bei der einfachen Sachlage alle zulässig wären.

## 4.3. Die Umgebung eines Prozesses

Jede Rechneraktivität im Betriebssystem UNIX vollzieht sich in einem Prozeß. Wenn das Betriebssystem gestartet wird, richtet es für jedes Terminal einen Prozeß ein. Tritt ein Nutzer an ein UNIX-Terminal, so findet er diesen Prozeß vor. Nach den erfolgreichen Anmeldungsformalitäten bietet ihm dieser Prozeß eine Benutzeroberfläche an, indem er einen neuen Prozeß einrichtet. Als Programm des neuen Prozesses wird ein Kommandointerpreter gestartet. Der Nutzerbeschreibung kann entnommen werden, welcher Interpreter verwendet werden soll. Wenn der Nutzer am Terminal sein erstes Promptzeichen findet, das ihn zur Eingabe von Kommandos ermuntert, sind also bereits mehrere Prozesse für ihn tätig. Jeder dieser Prozesse hat eine Umgebung.

### 4.3.1. Gestaltung der Umgebung

Die Umgebung eines Prozesses ist ein Satz von Variablen im Zeichenkettenformat. Neben dem aktuellen Dateiverzeichnis und den Informationen zur Unterbrechungsbehandlung ist die Umgebung eines der wichtigsten Prozeßattribute. Darauf wurde bereits im Abschnitt 1. hingewiesen (Bild 1.2). Der X/OPEN-Standard schreibt einen Mindestsatz von Umgebungsvariablen vor. Welche Umgebung durch das verwendete UNIX-System angeboten wird, kann durch die Kommandos

```
printenv            oder
env
```

sichtbar gemacht werden. Die Kommandos gelten für beide Interpreter.

```
% printenv
HOME=/u/polze
PATH=:/u/polze/bin:/bin:/usr/bin
TERM=ansi
HZ=50
TZ='mez-1mesz;W13,W39'
SHELL=/bin/csh
MAIL=/usr/spool/mail/polze
TERMCAP=li|ansi:al=\E[L:am:bs:cd=\E[J:ce=\E[K:
```

Das nächste Kommando zeigt den gleichen Sachverhalt an einem anderen Terminal.

```
% env
HOME=/u/polze
PATH=:/u/polze/bin:/bin:/usr/bin
TERM=wy60
HZ=50
TZ='mez-1mesz;W13,W39'
SHELL=/bin/csh
MAIL=/usr/spool/mail/polze
TERMCAP=w7|wy60|wyse60:is=\E`\072\Ee(\EO\Ee6\Ec41\E 4\Ec21\Ed/:..
```

Man kann die Umgebung des aktuellen Prozesses erweitern. Die C-Shell bietet dafür das Kommando

```
% setenv name value
```

an. Damit wird die Umgebung des Prozesses, in dem die C-Shell läuft, durch die Variable name ergänzt.

```
% setenv heute Donnerstag
% printenv
HOME=/u/polze

     usw.

TERMCAP=w7|wy60|wyse60:is=\E`\072\Ee(\EO\Ee6\Ec41\E 4\Ec21\Ed/:..
heute=Donnerstag
```

Die Ausschriften haben die Form:

```
name=value
```

So stehen die Umgebungsdaten auch im Speicher.

Wenn ein Kommandointerpreter einen Nachfolgeprozeß startet, etwa zum Zwecke der Abarbeitung eines eingegebenen Kommandos, gibt er diesem Prozeß eine Kopie seiner Umgebung als neue Prozeßumgebung mit auf den Weg. Mit dem Kommando

```
env [-][[name=value]...] cmd
```

kann die Umgebung des Nachfolgeprozesses, der für die Arbeit des Kommandos cmd gestartet wird, speziell modifiziert werden. Die Option - veranlaßt, daß der neue Prozeß eine leere Umgebung erhält. Die Paare name=value werden in die Umgebung des Nachfolgeprozesses zusätzlich aufgenommen.

```
% env - printenv
% env - morgen=Freitag printenv
morgen=Freitag
%
```

Die Bourne-Shell sieht neben dem Kommando env weitere Möglichkeiten zur Modifikation der Umgebung von Nachfolgeprozessen vor. Zunächst erst einmal wird durch das Kommando export geregelt, welche der vorhandenen Variablen des Kommandointerpreters in die Umgebung von Nachfolgeprozessen aufgenommen werden sollen. Der Aufruf des Kommandos export ohne Parameter zählt die exportierten Variablen auf.

```
$ heute=Donnerstag
$ set
HOME=/u/polze
HZ=50
IFS=

MAIL=/usr/spool/mail/polze
MAILCHECK=600
PATH=:/u/polze/bin:/bin:/usr/bin
PS1=$
PS2=>
SHELL=/bin/csh
TERM=wy60
TERMCAP=w7|wy60|wyse60:is=\E`\072\Ee(\EO\Ee6\Ec41\E 4\Ec21\Ed/:..
TZ='mez-1mesz;W13,W39'
heute=Donnerstag
$ env
HOME=/u/polze
HZ=50
MAIL=/usr/spool/mail/polze
PATH=:/u/polze/bin:/bin:/usr/bin
SHELL=/bin/csh
TERM=wy60
TERMCAP=w7|wy60|wyse60:is=\E`\072\Ee(\EO\Ee6\Ec41\E 4\Ec21\Ed/:..
TZ='mez-1mesz;W13,W39'
$ export heute
$ env
HOME=/u/polze

     usw.

TZ='mez-1mesz;W13,W39'
heute=Donnerstag
```

```
$ export
export heute
$ unset heute
$ env
HOME=/u/polze

     usw.

TZ='mez-1mesz;W13,W39'
$
```

Die letzten Zeilen des Skripts zeigen, wie man Variable aus der Umgebung entfernen kann. Ein analoges Kommando für die C-Shell taucht in den Dokumentationen unter dem Namen unsetenv name gelegentlich auf, steht aber nicht durchgängig zur Verfügung.

Die Bourne-Shell gestattet die Notation von Schlüsselwortparametern. Sie haben ebenfalls die Gestalt:

```
name=value
```

Schreibt man Schlüsselwortparameter vor ein Kommando, so gelangen sie als Variable in die Umgebung des Prozesses, der das Kommando abarbeitet. Wenn die Bourne-Shell mit der Option -k arbeitet, werden Zeichenfolgen der Form name=value auch nach dem Kommandonamen als Schlüsselwortparameter interpretiert und in die Umgebung eingefügt.

```
$ heute=Donnerstag cmd arg1     argn
$ set -k
$ cmd arg1     heute=Donnerstag     argn
$
```

### 4.3.2. Programmabarbeitung in Prozessen

Nun schließt sich die Frage an, wie man in Prozessen mit den Umgebungsvariablen arbeiten kann. Das Betriebssystem stellt Programmen, die als Kommandos aufgerufen werden sollen, aktuelle Parameter nach dem folgenden Muster zur Verfügung:

```
main(argc,argv,envp)
int argc;
char **argv, **envp;
{}
```

Entsprechend der Programmierung in der Sprache C müssen diese Parameter nur angegeben werden, wenn sie benötigt werden. Am Ende der Liste aktueller Parameter können unbenutzte weggelassen werden. Deshalb waren in vorangegangenen Beispielen allenfalls die ersten beiden Parameter vorhanden. Wollen wir auf die Umgebungsvariablen zugreifen, so hilft uns der dritte Parameter envp weiter. Er verweist auf eine Liste von Zeigern auf die Zeichenketten name=value, die unsere Umgebung bilden. Des weiteren wird beim Start eines Prozesses die globale Variable environ auf die Zeigerliste der Umgebung gesetzt. Wie man mit beiden arbeiten kann, zeigt das folgende kleine Beispiel envt1.c. Das Bild 4.1 stellt die Datenstruktur graphisch dar. Es demonstriert die Parameteranordnung bei allen Systemrufen exec(), die zum Start oder Neustart von Prozessen verwendet werden können.

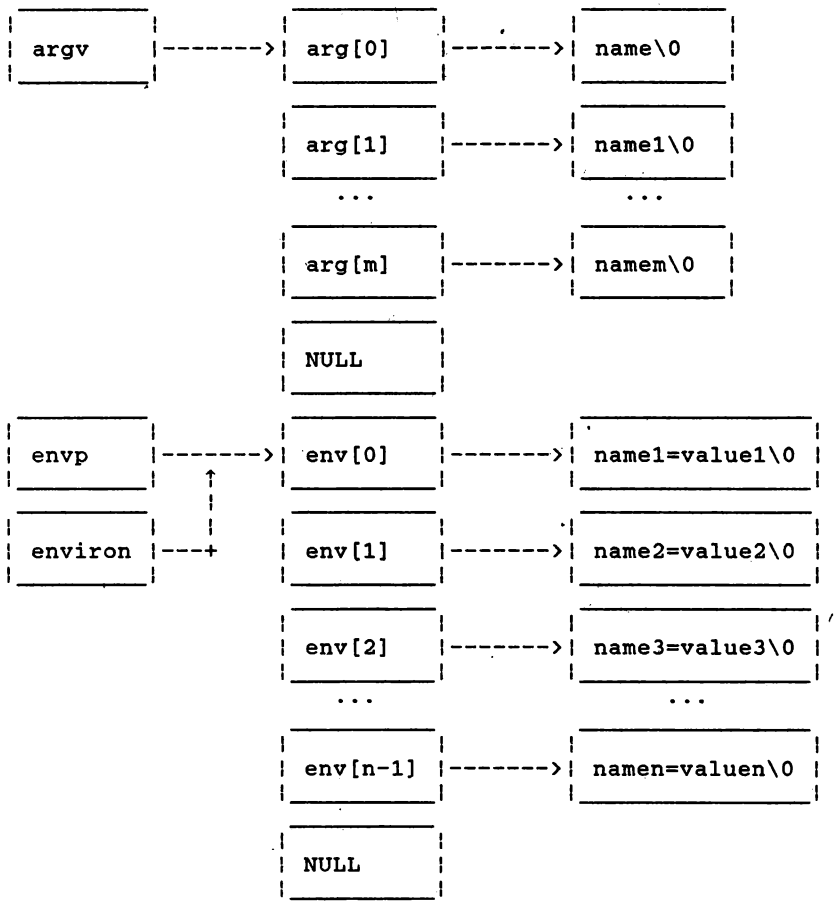



Bild 4.1. Parameter der exec-Systemrufe

```
% cat envt1.c
#include <stdio.h>
main(argc,argv,envp)
int argc;
char **argv, **envp;
{
extern char **environ;
printf("argc= %3d\n",argc);
while (*argv)
  printf("nextarg= %s\n",*argv++);
while (*envp)
  printf("nextenv= %s\n",*envp++);
printf("firstenv= %s\n",*environ);
}
% make -s !$:r
make -s envt1
% !$ file1 -opt1 -opt2 file2
envt1 file1 -opt1 -opt2 file2
```

```
argc=   5
nextarg= envt1
nextarg= file1
nextarg= -opt1
nextarg= -opt2
nextarg= file2
nextenv= HOME=/u/polze
nextenv= PATH=:/u/polze/bin:/bin:/usr/bin
nextenv= TERM=wy60
nextenv= HZ=50
nextenv= TZ='mez-1mesz;W13,W39'
nextenv= SHELL=/bin/csh
nextenv= MAIL=/usr/spool/mail/polze
nextenv= TERMCAP=w7|wy60|wyse60:is=
firstenv= HOME=/u/polze
%
```

Wenn man das vorige Programm geringfügig modifiziert, kann man sich einen Überblick über die Speicherverteilung der Umgebungsstruktur am konkreten System verschaffen (envt2.c).

```
% cat envt2.c
#include <stdio.h>
main(argc,argv,envp)
int argc;
char **argv, **envp;
{
extern char **environ;
printf("argc= %3d\n",argc);
while (*argv)
  printf("nextarg= %s\n",*argv++);
printf("environ= %d, *environ= %d\n",environ,*environ);
while (*envp) {
  printf("envp= %d, *envp= %d, nextenv= %s\n",envp,*envp,*envp);
  envp++;
}
printf("envp= %d, *envp= %d\n",envp,*envp);
printf("firstenv= %s\n",*environ);
}
```

### 4.3.3. Bibliotheksfunktionen

Die Unterprogrammbibliothek des C-Systems stellt für die Arbeit mit der Umgebung zwei Routinen bereit:

```
char *getenv(name)
char *name;

int putenv(string)
char *string;
```

Die Routine getenv() erwartet als Parameter den Namen einer Umgebungsvariablen und liefert im Ergebnis einen Zeiger auf den Wert der Variablen, falls sie in der Umgebung gefunden werden konnte. Andernfalls liefert die Routine einen Nullzeiger als Resultat.

Die Routine putenv() verlangt als Parameter eine Zeichenkette der Form "name=value". Gibt es die Umgebungsvariable name bereits, so erhält sie den neuen Wert value. Sonst wird die Variable name in die Umgebung aufgenommen. Ergebniswert 0 bedeutet erfolgreiche Arbeit.

Die folgenden Programme heute.c und exenvvar.c demonstrieren die Wirkungsweise der Routine getenv(). Sie zeigen zudem, daß man Umgebungsvariable auch als »verdeckte Argumente« auffassen kann, durch die der Arbeitsablauf eines Programms beeinflußt werden kann, ohne dies im Kommando explizit zu notieren.

```
% cat heute.c
/* Das Programm fragt nach der Umgebungsvariablen heute
   und druckt deren Wert. */
#include <stdio.h>
main()
{
char *p, *getenv();
if (p=getenv("heute"))
  puts(p);
else
  puts("heute?, weiss ich nicht.");
}
% make -s !$:r
make -s heute
heute.c
% !$
heute
heute?, weiss ich nicht.
% setenv heute Donnerstag
% heute
Donnerstag

% cat exenvvar.c
/* Das Programm fragt nach den als Parameter angegebenen
   Umgebungsvariablen und druckt deren Werte. */
main(argc,argv)
int argc;
char **argv;
{
char *p, *getenv();
while (++argv,--argc)
  if (p=getenv(*argv))
    printf("%s = %s\n",*argv,p);
  else
    printf("%s ist unbekannt\n",*argv);
}
% make -s !$:r
make -s exenvvar
exenvvar.c
% !$ gestern heute morgen
exenvvar gestern heute morgen
gestern ist unbekannt
heute = Donnerstag
morgen ist unbekannt
%
```

Nun wollen wir noch die Routine putenv() vorführen. Hier muß man sich ein weiteres Mal darüber im klaren sein, daß es sich um eine Manipulation der Umgebung des Prozesses handelt, in dem das gestartete Programm arbeitet, und nicht etwa um eine Manipulation der Umgebung des Kommandointerpreters. Es wäre also fehl am Platze, Änderungen an Umgebungen, die mit putenv() vorgenommen wurden, mit Kommandos der Sorte printenv oder env kontrollieren zu wollen.

Dagegen eignet sich putenv() sehr gut dazu, bei der Programmierung von Überlagerungsstrukturen, Daten aus dem einen in das folgende Programm zu transportieren. Dabei kann man davon profitieren, daß bei jedem Systemruf exec() die bestehende Prozeßumgebung gesammelt und nach dem Muster von Bild 4.1 dem nächsten Programm übermittelt wird. Dies zeigt das folgende Beispiel. Die Funktion execve() ist ein Vertreter aus der Familie der exec()-Systemrufe.

Die Routine putenv() ist noch nicht in allen UNIX-Systemen verfügbar. Daher wurden in das nachfolgende Programm Ergänzungszeilen und Modifikationen in der Form von Kommentaren aufgenommen. Das geänderte Programm übergibt beim Wechsel der Überlagerungen die gleiche Umgebung, nur müssen Neubelegungen von Variablen der laufenden Umgebung unterbleiben.

```
Script started on Fri Jul  1 14:12:26 1988
Neue C-Shell
1% cat tagfolge.c
#include <stdio.h>

extern char **environ;
char *pn, *getenv();
char nq[20];

char *Tage[] = {
        "Montag",
        "Dienstag",
        "Mittwoch",
        "Donnerstag",
        "Freitag",
        "Sonnabend",
        "Sonntag",
        (char *)0};

char *tev(ev)
char *ev; {
  char *q;
  if (!(q=getenv(ev)))
    printf("fct tev: %s nicht gefunden\n",ev);
  return q;
}

main() {
  char *q; int i, nn;
  char *nargv[2];
              /* Zusatz: char *nenvp[3]; */

  nargv[0]="tagfolge";
  nargv[1]=(char *)0;
              /* Zusatz: nenvp[2]=(char *)0; */

  q=tev("heute");
  if (!q)  {
    puts("tagfolge: Umgebungsvariable \"heute\" nicht auffindbar");
    exit(1);
  }
  i=0; while ((strcmp(q,Tage[i])) && (i<7)) i++;
  if (i==7) {
    printf("tagfolge: %s ist kein Tagesname.\n",q);
    exit(1);
  }
```

```
    printf("\ngestern war %s\n",Tage[(i+6)%7]);
    printf("heute ist %s\n",Tage[i]);
    printf("morgen ist %s\n\n", Tage[(i+1)%7]);

    sprintf(nq,"heute=%s",Tage[(i+1)%7]);
    putenv(nq);
                /* Ersatz: nenvp[1]=nq; */

    pn=tev("n");
    if (pn) {
      sscanf(pn,"%d",&nn);
      if(--nn) {
        sprintf(pn,"n=%1d",nn);
        putenv(pn);
                /* Ersatz: nenvp[0]=pn; */

        printf("n=%s, noch ein Tag:\n",tev("n"));
                /* Ersatz:
                 printf("%s, noch ein Tag:\n",pn); */

        execve("./tagfolge",nargv,environ);
                /* Ersatz:
                 execve("./tagfolge",nargv,nenvp); */

        perror("tagfolge: execve geerdet");
                /* Diese Zeile wird nicht ereicht,
                 wenn execve() gelingt. */
      }
    }
}
2% tagfolge
fct tev: heute nicht gefunden
tagfolge: Umgebungsvariable "heute" nicht auffindbar
3% setenv heute Freitag
4% tagfolge

gestern war Donnerstag
heute ist Freitag
morgen ist Sonnabend

fct tev: n nicht gefunden
5% setenv n 4
6% tagfolge

gestern war Donnerstag
heute ist Freitag
morgen ist Sonnabend

n=3, noch ein Tag:

       usw.

n=1, noch ein Tag:

gestern war Sonntag
heute ist Montag
morgen ist Dienstag

7%
script done on Fri Jul  1 14:13:51 1988
```

UNIX bietet mehrere Varianten des exec()-Systemrufs an. Alle orientieren sich an der Parameterstruktur gemäß Bild 4.1. Generell wird durch einen exec()-Systemruf der laufende Prozeß durch einen neuen Programmtext überlagert.

```
#define NULL (char *)0

int execl(path,arg0,arg1,...,argn,NULL)
char *path, *arg0, *arg1, ..., *argn;

int execv(path,argv)
char *path, **argv;

int execle(path,arg0,arg1,...,argn,NULL,envp)
char *path, *arg0, *arg1,      *argn, **envp;

int execve(path,argv,envp)
char *path, **argv, **envp;

int execlp(file,arg0,arg1,...,argn,NULL)
char *file, *arg0, *arg1, ..., *argn;

int execvp(file,argv)
char *file, **argv;

extern char **environ;
```

Mit dem Argument path wird ein Pfadname der Datei angegeben, die den neuen Programmtext enthält. Bei den p-Varianten des exec()-Systemrufs wird der Dateiname file einem Suchverfahren wie bei der Ausführung von Programmen durch Kommandointerpreter unterzogen. Der Suchablauf wird dabei durch die Variable PATH oder path gesteuert. Die l-Varianten des exec()-Systemrufs geben die Argumente einzeln an. Die aktuellen Parameter arg0, ... ,argn bilden die Argumenteliste entsprechend Bild 4.1. Die v-Varianten des exec()-Systemrufs verwenden eine Variable mit der Anfangsadresse der Zeigerliste als Wert. Die e-Varianten des exec()-Systemrufs gestatten die Übergabe einer aktuell aufbereiteten Umgebung. In jedem Fall zeigt die externe Variable environ auf die für die neue Programmstruktur gültige Umgebung.

## 4.4. Kommandodateien

### 4.4.1. Aufruf der Interpreter

Bisher erhielten Kommandointerpreter die zu interpretierenden Eingabetexte von einem Terminal. Mit den Kommandos

```
sh  options sh_file          oder
csh options csh_file
```

werden die Interpreter veranlaßt, Kommandos den Dateien sh_file oder csh_file zu entnehmen. Werden Kommandodateien mit dem Zugriffsrecht x

```
chmod +x sh_file
chmod +x csh_file
```

ausgestattet, so können sie direkt als Kommandonamen verwendet werden. Zur Interpretation der in der Datei enthaltenen Kommandos wird eine neue Version einer C-Shell oder einer Bourne-Shell gestartet. Die C-Shell startet nur dann wieder eine C-Shell zur Interpretation der Kommandodatei, wenn diese mit einem Nullkommando oder einer Kommentarzeile der C-Shell (#-Zeichen am Zeilenanfang) beginnt. Sonst wird die Kommandodatei durch eine Bourne-Shell interpretiert. Üblich ist es, Dateien mit dem Nullkommando der Bourne-Shell zu beginnen, wenn sie durch diesen Interpreter bearbeitet werden sollen.

Beim Aufruf der Kommandointerpreter sind folgende Optionen zulässig:

-c string
    Die Zeichenkette string wird als Kommando interpretiert.
-e  Der Interpreter wird verlassen, wenn ein Kommando abnormal endet oder einen exit()-Wert ungleich null liefert.
-f  - nur csh -
    Schnellstart, Suche und Abarbeitung von .cshrc unterbleiben.
-n  Kommandos werden analysiert, aber nicht abgearbeitet.
-s  Kommandos werden von stdin geholt.
-t  Beendigung der Interpretation nach jeweils einem Kommando.
-v  Protokolliert die zu interpretierenden Kommandos.
-V  - nur csh -
    Auch Kommandos aus .cshrc werden protokolliert.
-x  Protokolliert Kommandos nach eventuellen Ersetzungen, unmittelbar vor ihrer Abarbeitung.
-X  - nur csh -
    Auch Kommandos aus .cshrc werden nach Ersetzungen protokolliert.

Nach der Bewilligung der Ausführungserlaubnis (chmod +x) können Namen von Kommandodateien in zu interpretierenden Kommandos notiert werden. Selbstverständlich müssen die Kommandodateien lesbar sein. Gemeinhin folgen auf den Kommandonamen Parameter, Argumente oder Optionen. Um in der Kommandodatei darauf zugreifen zu können, wurden dafür die Variablen $0, $1, ... , $9 und $* reserviert. Der Kommandoname, also Name der Kommandodatei, ist Wert von $0. Die nächsten Parameter werden entsprechend ihrer Stellung numeriert. Schließlich bedeutet $* die Liste der Parameter beginnend mit $1. Eine flexiblere Lösung ist durch die Steueroperation shift möglich. Sie verschiebt die Parameterliste um eine Position nach links. Der Parameter $0 bleibt davon unberührt, ansonsten wird der bisherige Parameter $2 zum neuen Parameter $1 erklärt. Damit stellt die Numerierung der Parameter durch eine Ziffer keine drückende Begrenzung dar.

### 4.4.2. Steuerkommandos

Die Kommandointerpreter sehen Möglichkeiten zur Ablaufsteuerung vor, wie sie in höheren Programmiersprachen üblicherweise anzutreffen sind. Aus Gründen der Portabilität sollte man sich bei der Formulierung von Kommandodateien an der Bourne-Shell orientieren. Nur diese ist Bestandteil des X/OPEN-Portability Guide.

Die folgende Aufzählung stellt Steuerstrukturen beider Interpreter nebeneinander. Sie erklären sich selbst. Die C-Shell-Konstrukte orientieren sich an der Programmiersprache C. Bei der switch-Struktur gibt es Abweichungen. Die Bourne-Shell knüpft Abbruch- oder Verzweigungsbedingungen an den exit-Wert eines Kommandos oder des letzten einer Folge von Kommandos. Die C-Shell benutzt Ausdrücke.

| sh | Konstruktion von Zyklen | csh |
|---|---|---|

```
for var in word_list                    foreach name (word_list)
do                                        cmd_list
  cmd_list                              end
done

for var                                 label:
do                                        cmd_list
  cmd_list                                goto label
done

while cmd_list                          while (expression)
do                                        cmd_list
  cmd_list                              end
done

until cmd_list
do
  cmd_list
done

break [n]                               break

continue [n]                            continue

shift                                   shift [var]
```

Das Kommando break bricht Zyklen ab, bei sh über n Aufrufstufen hinweg. Das Kommando continue verzweigt zur Abbruchbedingung zurück, bei sh wiederum über n Stufen. Das Kommando shift verschiebt die Parameter um eine Stelle nach links. Der Parameter $0 wird davon nicht betroffen. Bei csh vollzieht sich die Verschiebung in der Variablen argv oder in der angegebenen Variablen var.

| sh | Verzweigungstabellen | csh |
|---|---|---|

```
case string in                          switch (word)
  pattern ) cmd_list                    case string1:
    ...                                     cmd_list
  pattern ) cmd_list ;;                     breaksw
esac                                         ...
                                        case stringn:
                                            cmd_list
                                            breaksw
                                        default:
                                            cmd_list
                                            breaksw
                                        endsw
```

Die Marken pattern können in gleicher Weise gebildet werden, wie es die Dateinamengenerierung von sh erlaubt. Gleiches gilt für die

case-Zeichenketten bei csh. Hier müssen die Zweige mit breaksw begrenzt werden. Sonst wird die Kommandofolge bis endsw durchlaufen.

| sh | Binäre Verzweigungen | csh |
|---|---|---|

```
if cmd_list                          if (expression) then
then cmd_list                          cmd_list
fi                                   endif

if cmd_list                          if (expression) then
then cmd_list                          cmd_list
else                                 else
cmd_list                               cmd-list
fi                                   endif

if cmd-list                          if (expression) then
then cmd_list                          cmd_list
elif cmd_list                        else if (expression) then
then cmd_list                          cmd_list
  ...                                    ...
elif cmd_list                        else if (expression) then
then cmd_list                          cmd_list
else                                 else
cmd_list                               cmd_list
fi                                   endif
```

Aufgrund der Festlegungen des X/OPEN-Standards ist es gewiß nutzlos, für C-Shell-Kommandodateien zu werben. Die Steuerstrukturen der C-Shell sollte man dennoch im Blickfeld behalten, nachdem man sich entschlossen hat, die C-Shell zur Terminalsteuerung auszuwählen, wie das in diesem Buch durchgängig verfolgt wurde. Bisweilen kommt man auch im Terminaldialog in Situationen, in denen kleine Kommandozyklen hilfreich sein können. Das zeigen die folgenden Beispiele.

Zuerst sei vermerkt, daß manche UNIX-Systeme beispielsweise nicht den vollen Umfang des Kommandos size realisieren, so wie er im Abschnitt 2.1. beschrieben worden ist. Das Kommando size soll für Archivdateien eine Liste der Größen der einzelnen Archiveinträge liefern. Wenn sich size nur auf Dateien anwenden läßt, so leistet der folgende Zyklus das Gewünschte. Er beschreibt die kleine Version des Routinenarchivs zum Programmgenerator lex.

```
% set LIB=/lib/Slibl.a
% foreach i (`ar t $LIB`)
? ar x $LIB $i
? echo "${i}: \c"; size $i
? rm $i
? end
__.SYMDEF: size: __.SYMDEF: not an object file
allprint.o: 210 + 19 + 0 = 229 = 0xe5
main.o: 21 + 0 + 0 = 21 = 0x15
reject.o: 280 + 0 + 0 = 280 = 0x118
yyless.o: 92 + 0 + 0 = 92 = 0x5c
yywrap.o: 15 + 0 + 0 = 15 = 0xf
%
```

Das zweite Beispiel zeigt einen nroff-Anwendungsfall. Im Verzeichnis Froe seien die nroff-Texte (Makropaket ms) der Abschnitte eines Buches erfaßt. Das Verzeichnis Froe soll keine weiteren Dateien beinhalten. Die Dateien werden in dem folgenden Zyklus hemdsärmelig und ohne Netz mit nroff -ms behandelt und mit dem Namenszusatz _u in das Verzeichnis Fnroff geschrieben.

```
% cd
% cat Fnroff/fum
.ds A \o'"A'
.ds a \o'"a'
.ds O \o'"O'
.ds o \o'"o'
.ds U \o'"U'
.ds u \o'"u'
% cd Froe
% echo '.so fum' > ../Fnroff/v
% foreach i (*)
? echo ".so ../Froe/${i}" >> ../Fnroff/v
? end
% cd ../Fnroff
% nroff -ms v > vu
%
```

Die nroff-Texte benutzen Zeichenkettenvariable für die Notation deutscher Umlaute, so die Variable *a für ä. Dazu wird jedem nroff-Text die Datei fum vorangesetzt. Konstrukte der Form .so file sind nroff-Steuerkommandos für den Texteinschluß. Die Datei fum definiert nroff-Überdruckfunktionen in der Form \o'"a' für ä als Werte für die Zeichenkettenvariablen, also für *a. Der Textformatierer nroff produziert daraus die Zeichenketten "\"\ba". Aus diesen Überdruckfolgen stellt das Programm upo Zeichen des erweiterten Zeichensatzes her, so daß mit dem folgenden Druckkommando das ganze Buch gedruckt werden kann.

```
% upo < vu | pr -ft -166 | lpr
%
```

Als Kommando zur Formulierung der Abbruch- oder Verzweigungsbedingungen in den Steuerkommandos der Kommandointerpreter wird häufig test verwendet. Es hat die Aufrufformen

```
test expr
[ expr ]
```

Der Ausdruck expr wird berechnet. Der Wert true führt zum exit-Wert 0. Der Wert false liefert ebenso wie ein Aufruf ohne Parameter einen exit-Wert ≠0. Die folgende Liste beschreibt die Ergebniswerte true. Bezüge auf Dateien liefern immer false, wenn es die Datei nicht gibt.

```
-r file    Die Datei ist lesbar.
-w file    Schreiberlaubnis.
-x file    Abarbeitungserlaubnis.
-f file    Reguläre Datei.
-d file    Verzeichnisdatei.
-c file    Zeichenorientierte Gerätedatei.
-b file    Blockorientierte Gerätedatei.
-p file    Die Datei ist ein Pipe.
-u file    Das s-Bit für den Eigentümer ist gesetzt.
-g file    Das s-Bit für die Gruppe ist gesetzt.
```

```
-s file      Die Länge ist >0.
-t [filedescriptor]
             Die Datei mit der angegebenen Deskriptornummer ist  einem
             Terminal zugeordnet (Standard ist 1).
-z string    Die Länge der Zeichenkette string ist 0.
-n string    Die Länge der Zeichenkette string ist ≠0.
s1 = s2      Die Zeichenketten s1 und s2 sind gleich.
s1 != s2     Die Zeichenketten s1 und s2 sind nicht gleich.
str          Die Zeichenkette str ist nicht leer.
n1 -op n2    Algebraischer Vergleich der  ganzen  Zahlen  n1  und  n2.
             Zulässige Operatoren op sind eq, ne, gt, ge, lt und le.
!            Negation.
-a           Konjunktive Verknüpfung (and).
-o           Disjunktive Verknüpfung (or).
( expr )     Klammerung.
```

### 4.4.3. Zwei Beispiele

Nun sollen zwei Beispiele seriöser Kommandodateien folgen. Die erste Datei testscript nach Schreiner und Friedman (1985) realisiert eine systematische Teststrategie, wenn etwa nacheinander neue Versionen eines Programms getestet werden sollen. Zur Validierung einer neuen Version möchte man erreichen, daß bisher erfolgreich abgearbeitete Testbeispiele wieder korrekt verarbeitet werden. Das Verzeichnis Tstinp möge Testdateien enthalten. Die Resultatdateien sollen mit gleichem Namen im Verzeichnis Tstout gesammelt werden. Der Testablauf könnte für einen Testfall zum Programm tstprog folgendermaßen aussehen:

```
$ tstprog optionen argumente < Tstinp/tstf1 > Tstout/tstf1
$ echo 'exit() Code' $? >> Tstout/tstf1
```

Neben den eigentlichen Ergebnisdaten soll jede Datei in Tstout noch eine Zeile mit dem exit-Wert von tstprog in diesem standardisierten Format enthalten. Für den Fall, daß es das Verzeichnis Tstout noch nicht gibt, richtet die Kommandodatei testscript zuerst ein solches Verzeichnis ein. Es wird ein Testprotokoll hergestellt. Bei identischen Resultaten wird die neue Ergebnisdatei weggeworfen. Bei unterschiedlichen Ergebnissen wird zusätzlich ein Protokoll nach dem Muster des Kommandos diff angefertigt. Die alte Datei wird mit dem Namenszusatz _old aufgehoben.

```
$ cat testscript
:           /bin/sh
if test ! -d $OD            # wenn kein Verzeichnis Tstout,
then    mkdir $OD           # dann Einrichten.
fi

for i in $ID/*              # sei $i=Tstinp/f
do  file=`basename $i       # file=f
    "$@" <$i >$OD/\&$file   #       ######
                            # testprog opt arg <Tstinp/f >Tstout/&f
    echo 'exit() Code' $? >>$OD/\&$file
                            # echo 'ex..' $? >Tstout/&f
```

```
    if test -r $OD/$file                    # gibt es f bereits?
    then  if cmp -s $OD/\&$file $OD/$file # ja, ist &f=f?
          then    echo "$1 $file unveraendert"     # ja
                  rm $OD/\&$file                   # rm &f
          else    echo "$1 $file veraendert: <neu >alt"
                                                   # nein, &f≠f
                  diff $OD/\&$file $OD/$file       # diff &f f
                  mv $OD/$file $OD/${file}_old     # f --> f_old
                  mv $OD/\&$file $OD/$file         # &f --> f
          fi
    else  echo "$1 $file erzeugt"           # nein, f gibt es nicht.
          mv $OD/\&$file $OD/$file          # &f --> f
    fi
  done
  exit 0
  $
```

Der Aufruf der Kommandodatei testscript erfordert folgende Schritte:

```
$ ID=Tstinp
$ OD=Tstout
$ export ID OD
$ testscript tstprog optionen argumente
```

Soll die Datei testscript auf Argumentdateien angewendet werden, für Programme des Aufrufmusters

```
$ tstprog optionen arg1     Tstinp/tstfile     > Tstout/tstfile
```

also, so muß in der Zeile (######) die Eingabeumlenkung unterbleiben. Dies ist beim Testen von Compilern der Regelfall (Schreiner und Friedman 1985).

Das nächste Beispiel einer Kommandodatei stellt zugleich eine Einstimmung auf den Abschnitt 5. dar. Ein größeres Programm setzt sich modular aus mehreren C-Quelltexten zusammen. Bezüge zwischen den Moduln werden in Einschlußdateien beschrieben. Mit der C-Präprozessoranweisung #include können importierende Moduln darauf zugreifen. So sollen die Programmtexte aussehen:

```
zge.c:
   # include "bi.h"
   # include "pu.h"
   # include "pr.h"
   # include "processes.h"
        weiterer Programmtext,
        keine weiteren include-Anweisungen

bi.c:                                bi.h:
   # include "pr.h"                     keine include-Anw.
   # include "processes.h"
        weiterer Programmtext,
        keine include-Anw. ...

pu.c:                                pu.h:
   # include "pr.h"                     keine include-Anw.
   # include "processes.h"
    ... weiterer Programmtext,
        keine include-Anw. ...
```

```
pr.c:                                pr.h:
   keine include-Anw.                   keine include-Anw.

processes.c:                         processes.h:
   # include "processes.h"              # include "system.h"
        weiterer Programmtext,              ... weiterer Programmtext,
        keine include-Anw. ...                  keine include-Anw. ...
```

Die Kommandodatei makescript stellt nach Bach u.a. (1987) folgende Abhängigkeiten her:

```
zge.o: bi.h pu.h pr.h processes.h system.h
bi.o: pr.h processes.h system.h
pu.o: pr.h processes.h system.h
pr.o:
processes.o: processes.h system.h
all: zge.o bi.o pu.o pr.o processes.o
```

Dieses Ergebnis wird durch den Aufruf

```
makescript zge.c bi.c pu.c pr.c processes.c
```

erzeugt. Es folgt eine Version der Datei makescript. Dabei erweist sich der im Abschnitt 3.2. vorgestellte Stream-Editor sed als ein sehr nützliches Werkzeug.

```
$ cat makescript
: /bin/sh

while test $1
do
  # Verarbeitung eines Parameters bis shift

  ziel= echo $1 | sed 's/.c$/.o/'`
  # Umwandlung von Parameter f.c --> f.o
  all="$all $ziel"
  # Akkumuliere all:-Liste

  dep=`sed -n '
    /^#[\t ]*include[\t ]*"/  {
        s|^#[\t ]*include[\t ]*"\([^"]*\)".*$|\1|p
    }' $1 `
  # Suche nach Zeilen # include "pathname"
  # Akkumuliere Pfadnamen von #include-Dateien der ersten
  # Abhängigkeitsstufe in $dep

  more=$dep
  while test "$more"
  do
  # Suche nach weiteren #include-Dateien
  # Solche, die sich in einer Datei aus $more befinden,
  # werden in $sub gesammelt.
  # Alle werden in $save aufgenommen.
  # $save bildet die nächste Abhängigkeitsstufe.
```

```
        save=
        for i in $more
        do
          sub=`sed -n
            /^#[\t ]*include[\t ]*"/    {
              s|^#[\t ]*include[\t ]*"\([^"]*\)".*$| \1|p
            }' $i `
          save="$save $sub"
        done
        dep="$dep $save"              ######
        more=$save
      done

      depend="$ziel: $dep"
      # Konstruktion der endgültigen Abhängigkeitszeile

      echo $depend
      shift
    done
    echo "all:$all"
    $
```

Beim näheren Studium dieses Programms wird bestimmt sichtbar, daß Dateien, die in mehreren #include-Dateien wieder als Einschlußdateien vorkommen, mehrfach in die Abhängigkeitszeilen aufgenommen werden, was letztlich zwar nicht stört, aber auch nicht gerade befriedigt. Ersetzt man die mit ###### markierte Akkumulationsanweisung durch den folgenden Zyklus, so werden nur solche Dateien in die Variable dep aufgenommen, die noch nicht darin enthalten sind.

```
    for j in $save
    do
      for k in $dep
      do
        if test "$j" = "$k"
        then continue 2
        fi
      done
      dep="$dep $j"
    done
```

Die jetzt folgende Version makescript1 zeigt andere Möglichkeiten von sed. Sie verwendet temporäre Dateien für die Sammlung der Pfadnamen von #include-Dateien. Dabei lenkt sie unser Interesse bereits auf die anschließend zu erörternde Problematik der Behandlung von Unterbrechungen.

```
$ cat makescript1
: /bin/sh

# Version mit temporären Dateien

ret=0
trap 'ret=1; exit' 1 2 3 15
trap 'rm -f /usr/tmp/$$incl; exit $ret' 0
# Das Kommando trap wird im Abschnitt 4.5. erklärt.
```

```
while test $1
do
  ziel=`echo $1 | sed 's/.c$/.o/'`
  all="$all $ziel"
  sed -n '
    /^#[\t ]*include[\t ]*"/  {
    s|^#[\t ]*include[\t ]*"\([^"]*\)".*$|\1|w '/usr/tmp/$$incl'
    }' $1
  more=`cat /usr/tmp/$$incl`
  dep=$more
  while test "$more"
  do
    save=
    for i in $more
    do
      sed -n
        /^#[\t ]*include[\t ]*"/    {
      s|^#[\t ]*include[\t ]*"\([^"]*\)".*$| \1|w '/usr/tmp/$$incl'
        }' $i
      sub=`cat /usr/tmp/$$incl`
      save="$save $sub"
    done
    dep="$dep $save"
    more=$save
  done
  depend="$ziel: $dep"
  echo $depend
  shift
done
echo "all:$all"
$
```

Alle Kommentare und die Bemerkung zur Vermeidung von wiederholten Abhängigkeitsbezügen gelten für diese Version in gleicher Weise.

## 4.5. Unterbrechungsbehandlung

### 4.5.1. Signale

Jeder Prozeß besitzt in seinem Deskriptor eine Variable vom Typ long (32 Bit), in der anhängige Unterbrechungssignale registriert werden. Sie beeinflussen seinen weiteren Ablauf wesentlich und entscheiden über seinen Fortbestand als zyklisch arbeitender Prozessor gemäß dem grundlegenden Architekturmodell nach J.v.Neumann, wie es schon einmal im Abschnitt 1. angedeutet wurde. Unterbrechungen werden durch äußere Ereignisse oder auch aus der Programmabarbeitung selbst heraus verursacht. Wann solche Ereignisse eintreten, ist für den betroffenen Prozeß völlig unbestimmt. Nun schöpfen derzeit die UNIX-Systeme den Vorrat unterscheidbarer Unterbrechungen längst nicht aus. In der Datei /usr/include/sys/signal.h findet man eine Zusammenstellung der möglichen Signale eines Systems. Es folgt ein Auszug aus dieser Datei mit Kommentaren, die auf das auslösende Ereignis deuten. Bei der Behandlung der Signale 3-8 und 10-12 wird ein Speicherabzug hergestellt. Er wird unter dem Namen core im Arbeitsverzeichnis hinterlegt und kann mit einem der Debugger adb oder sdb analysiert werden.

```
#define SIGHUP  1  /* hangup */
#define SIGINT  2  /* interrupt (rubout, delete) */
#define SIGQUIT 3  /* quit (ASCII FS, Ctrl-\) */
#define SIGILL  4  /* illegal instruction */
#define SIGTRAP 5  /* trace trap (not reset when caught) */
#define SIGIOT  6  /* IOT instruction */
#define SIGEMT  7  /* EMT instruction */
#define SIGFPE  8  /* floating point exception */
#define SIGKILL 9  /* kill (cannot be caught or ignored) */
#define SIGBUS  10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS  12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD  18 /* death of a child */
#define SIGPWR  19 /* power-fail restart */

#define NSIG    20

#define SIG_DFL (int (*)())0

#ifdef  lint
#define SIG_IGN (int (*)())0
#else
#define SIG_IGN (int (*)())1
#endif
```

In der Unterprogrammbibliothek des C-Systems befindet sich die interessante und leistungsfähige Routine signal. Mit der Parameterdeklaration hat sie folgende Gestalt:

```
int (*signal(sig,func())()
int sig;
int (*func)();
```

Damit kann man für ein Signal sig aus der obigen Liste festlegen, daß beim Auftreten des entsprechenden Unterbrechungsereignisses die Funktion func abgearbeitet werden soll. Der zweite Parameter von signal ist ein Zeiger auf diese Funktion. Vom gleichen Typ ist auch das Resultat, das bei der Abarbeitung von signal entsteht. Der Resultatzeiger verweist auf die Behandlungsroutine, die vor der Abarbeitung von signal gültig war. Dadurch wird die Aufbewahrung des Zustandes ermöglicht, der vor dem Aufruf von signal() vorlag. Hervorzuheben sind die Signalbehandlung nach dem Systemstandard und die Möglichkeit, eintreffende Signale kurzerhand zu ignorieren. Diesem Zweck dienen die beiden Bezeichnungen SIG_DFL für den Systemstandard und SIG_IGN für das Ignorieren von Signalen. Es sind Konstanten, die zwangsweise auf den geforderten Typ Zeiger auf eine Funktion eingestellt wurden.

Interessant ist die Frage, wie sich die Kommandointerpreter Unterbrechungssignalen gegenüber verhalten. Das aus Schreiner (1987) übernommene Programm demonstriert dies.

```
41% cat sigshow.c
#include <signal.h>
#include <stdio.h>
#define SIG_ERR ((int (*)()) -1)
```

```
int sigshow(fp) /* Teilt Signalnummern mit, die auf SIG_IGN
           führen oder nicht mit SIG_DFL reagieren, und
           ermittelt die Nummer des letzten Signals. */
FILE * fp;
{  register int s = 1, (*f)();

   for (s=1;; ++s)
     if (s == SIGKILL) continue;
         /* Signal 9 muß ausgelassen werden. */
     else {
       f = signal(s,SIG_IGN);
         /* f zeigt auf die bisherige Unterbrechungsfunktion. */
       if (f == SIG_ERR) break;  /* Endetest */
       else {
         signal(s,f);
         /* Rücksetzen auf die frühere Funktion */
         if (f == SIG_IGN)      /* Ignorieren */
           fprintf(fp,"%d: ignored\n",s);
         else if (f != SIG_DFL) /* Abfangen */
           fprintf(fp,"%d: caught\n",s);
       }
     }
   return s-1;
}
main() {
  printf("%d: last number\n",sigshow(stdout));
  return 0;
}
42% csh
Neue C-Shell
1% sigshow
16: ignored
19: last number
2% !! &
sigshow &
148
3% 1: ignored
2: ignored
3: ignored
16: ignored
19: last number
exec sigshow
3: ignored
15: ignored
16: ignored
19: last number
43% sh
$ sigshow
16: ignored
19: last number
$ sigshow &
153
$ 2: ignored
3: ignored
16: ignored
19: last number
exec sigshow
16: ignored
19: last number
44%
```

Hervorzuheben ist der Wechsel der Kommandointerpreter csh und sh untereinander. Er ist erforderlich, weil die Verwendung des Kommandos exec die jeweils bisherige Shell verdrängt. Im Übrigen ist festzustellen, daß sich die beiden Kommandointerpreter auch in diesem Zusammenhang unterschiedlich verhalten.

Nun wenden wir uns vor allem den Signalen zu, die für die Arbeit am Terminal von Belang sind. Das sind die Signale 1, 2, 3 und 15, 16, 17. Die ersten drei Signale werden durch Tasteneingaben ausgelöst, das Signal 15 gehört zur Voreinstellung im Kommando kill, und die Signale 16 und 17 können durch den Nutzer freizügig verwendet werden. Bereits mehrfach wurde darauf hingewiesen, daß im Gefolge einer Terminalarbeit immer eine ganze Kollektion von Prozessen aktiviert wird. UNIX spricht von Prozeßgruppen. Ein Prozeß in der Gruppe ist leitender Prozeß. Dies ist in vielen Fällen derjenige Prozeß, der das Terminal steuert, in dem der dialogführende Kommandointerpreter agiert.

Gibt der Nutzer am Terminal ein Signal an das System, so gelangt es zuerst an den Interpreterprozeß. Dieser reicht es an synchron gestartete Prozesse weiter und erwartet deren normale oder abnormal verlaufene Beendigung. Das Ganze geht rekursiv durch die hierarchisch aufgefächerte Prozeßgruppe hindurch. Schließlich steht vor dem Interpreter die Frage, wie er seine Arbeit fortsetzen soll. Für diesen Zweck hat die Bourne-Shell das Kommando trap und die C-Shell das Kommando onintr. Beide sind Spezialkommandos der Interpreter. Das Kommando trap hat folgende Aufrufform:

```
trap [arg] [signr]
```

Trifft ein Signal mit der Nummer signr ein, so wird das als Kommando zu interpretierende Argument arg ausgeführt. Wichtig sind die Sonderfälle:

trap signr
: Leere Zeichenkette für arg. Das Signal signr soll ignoriert werden.

trap arg 0
: Das Kommando arg wird einmal ausgeführt, wenn die Shell verlassen wird, gegebenenfalls in der Folge des Kommandos exit.

trap
: Ohne Parameter. Gültige trap-Aktionen werden aufgelistet.

Nach der Unterbrechungsbehandlung durch trap, also nach der Abarbeitung des darin vorgesehenen Kommandos, setzt sh die Arbeit mit der Interpretation des folgenden Kommandos fort. Soll die Bearbeitung einer Kommandodatei im Unterbrechungsfall abgebrochen werden, so muß dies mit dem exit-Kommando explizit verlangt werden.

Die Argumente des trap-Kommandos müssen sorgfältig mit Quote-Zeichen umschlossen werden, weil Variablen- oder Kommandosubstitutionen in den Argumenten von trap auch dann vorgenommen werden, wenn sh das trap-Kommando interpretiert, ohne durch ein Unterbrechungsereignis darauf hingeführt zu werden. Dies kann mehrfach der Fall sein, genauso wie Unterbrechungen mehrfach auf das gleiche trap-Kommando zugreifen können.

Mit dem Kommando kill kann man vom Terminal aus Signale an Prozesse senden. Es hat die Aufrufform

```
kill [-signr] pid
```

mit der Signalnummer signr und Prozeßdeskriptornummern der gewünschten Prozesse. Das Kommando

```
kill -9 pid
```

beendet den Prozeß pid im Wortsinn des Kommandonamens. Dagegen kann sich kein Prozeß schützen. Allerdings erfolgt die Signalbehandlung jeweils am Anfang einer Prozeßaktivierung. Kommt eine Prozeß während einer Zeitscheibe zum Stehen, kann er auch mit kill nicht mehr erreicht werden. Mit dem Kommando

```
kill pid
```

wird das Signal 15 (SIGTERM) gesendet.

Für beide Kommandos kill und trap müssen Signalnummern als Zahlen notiert werden. Die Signalnamen aus #include <signal.h> werden nicht verstanden. Bedauerlicherweise liegt hier eine gewisse Systembindung vor. Im Laufe der Weiterentwicklung von UNIX soll das bereinigt werden, X/OPEN (1987). Der bisherige Standard der X/OPEN-Gruppe schreibt lediglich die Signale 1, 2, 3 und 15 fest.

### 4.5.2. Berücksichtigung in Kommandodateien

In Kommandodateien ist der Einsatz des trap-Kommandos besonders wichtig, wenn temporäre Dateien benutzt werden. Wenn die Interpretation einer Kommandodatei mißlingt oder wenn sie vom Terminal aus abgebrochen wird, bleiben temporäre Dateien unter Umständen übrig, und keiner denkt daran, sie zu beseitigen, bis die Speicherressourcen schwinden.

Das nachfolgende kleine Beispiel hilft für den Fall, daß die Standardausgabe nicht auf Eingabedateien von Transformationsprogrammen gelenkt werden kann. Systemprogramme wie cat oder sed, aber auch nroff sind beispielsweise so beschaffen. Hier kann man, Bach u.a. (1987) folgend, die Kommandodatei into einsetzen. Sie richtet die in diesen Fällen benötigte Zwischenspeicherdatei als temporäre Datei $Neu ein. Mit der Shell-Variablen $$ bekommt man die Deskriptornummer des Prozesses, der die Datei into interpretiert. Solche Nummern werden systemweit eindeutig vergeben. Dadurch ist die Wahl dieser Bezeichnung unstrittig. Die Standardeingabe von into gelangt in die Datei $Neu. Mit der letzten Kommandozeile von into wird die Datei $Neu in die Parameterdatei $1 kopiert, und diese kann durchaus mit der Ausgangsdatei der Transformation übereinstimmen.

Folgender Aufruf ist möglich, wenn aus der Datei text sämtliche Leerzeilen gestrichen werden sollen.

```
% sed '/^$/d' text   into text
```

Die Datei into enthält folgende Kommandos:

```
% cat into
: /bin/sh
# Aufrufe:
# % filter <name   into name
# % transf  name   into name
```

```
trap 'ret=1; exit' 1 2 3 15
trap 'rm -f $Neu; exit $ret' 0
ret=0
Neu=/usr/tmp/$$into
cat >$Neu
trap '' 1 2 3 15
cp $Neu $1
%
```

Erhält ein Interpreter, der in einem Nachfolgeprozeß into abarbeitet, Signale mit den Nummern 1, 2, 3 oder 15, so wird er anschließend das Kommando exit starten und die Kommandodatei into verlassen. Für den exit-Ablauf ist sowohl beim normalen Ende der Abarbeitung als auch im Unterbrechungsfall ein weiteres trap-Kommando vorhanden, das die temporäre Zwischendatei $Neu beseitigt. Der exit-Wert wird über die Shell-Variable ret vermittelt. Während des Kopierens der Datei $Neu in die Resultatdatei werden keine Unterbrechungen zugelassen. Dies besorgt das dritte trap-Kommando in der Datei into.

In analoger Weise würden auch die temporären Dateien in der Kommandodatei makescript1 im vorigen Abschnitt 4.4. behandelt.

Zum Vergleich sei es gestattet, diese kleine Kommandodatei noch einmal mit den Mitteln der C-Shell wiederzugeben.

```
% cat c_into
# /bin/csh
  set Neu=/usr/tmp/$$c_into
  onintr rem
  cat >$Neu
  onintr -
  cp $Neu $1
  rm $Neu; exit(0)
rem:
  rm $Neu; exit(1)
%
```

Hier steuert das Kommando onintr den Interpreterlauf, wenn Unterbrechungen vorkommen. Die Form onintr - läßt Unterbrechungen ignorieren, onintr rem verzweigt zum markierten Kommando.

Bei der Signalbehandlung muß auf eine Ausnahme hingewiesen werden. Spezialkommandos der Bourne-Shell sind als Unterprogramme von sh organisiert. Es gibt also gar keinen synchron gestarteten Prozeß zur Kommandoabarbeitung, an den das Signal weitergereicht werden könnte. Die Bearbeitung des Signals erfolgt in diesen Fällen nach der Rückkehr aus dem Unterprogramm und damit nach der Beendigung des Spezialkommandos. Spezialkommandos sind also in diesem Sinne nicht unterbrechbar. Dies kann man mit dem folgenden Terminalskript nachvollziehen:

```
% sh
$ trap 'echo Interrupted!' 2
$ trap
2: echo Interrupted!
$ cp file1 file2
       !! <delete>-Taste drücken!
Interrupted!
```

```
$ hash
       !! <delete>-Taste drücken!
hits    cost    command
1       1       /bin/cp
$ pwd
/u/polze/priv/book/cont
Interrupted!
$
       !! <delete>-Taste drücken!
  lc
    ... Text
Interrupted!
$ hash
hits    cost    command
1       1       /bin/cp
1       1       /bin/lc
$ Ctrl-d %
```

Die Aufrufe von $ hash zeigen, daß nur die beiden Kommandos cp und lc aus dem Interpreter herausführten. Am Ende ist erkennbar, daß die Aktion aus dem trap-Kommando erst nach der Beendigung des für lc eingerichteten, synchron abgelaufenen Prozesses wirksam wird.

Erneut eine Ausnahme in dieser Situation ist das Spezialkommando read. Es ist unterbrechbar, wie man beim Experimentieren mit der folgenden Kommandodatei beobachten kann. Das aktuelle Arbeitsverzeichnis möge weitere Verzeichnisdateien enthalten. Sie werden der Reihe nach angesprochen. Eingabe von Ctrl-d (end of file) bricht den while-Zyklus ab. Mit dem Verzeichnisnamen als Aufforderungsprompt wird der Nutzer angeregt, ein Kommando einzugeben. Mit dem Kommando eval der Bourne-Shell wird der Eingabetext abgearbeitet. Diese Datei ist auf das Unterbrechungssignal SIGINT eingestellt. Wird es während des Lesens gesendet, bricht die Verarbeitung ab. Die Auswertung eines eingegebenen Kommandos ist jedoch nicht unterbrechbar.

```
% cat read_demo
: /bin/sh
d=`pwd`
for i in *
do
  if test -d $d/$i
  then cd $d/$i
    while echo "$i:"
          trap exit 2
          read x
    do
          trap : 2
          # SIG_INT ignorieren.
          eval $x
    done
  fi
done
%
```

Das folgende Terminalprotokoll schließt die Betrachtungen zur Unterbrechungsbehandlung ab. Es zeigt eine kleine Kommandodatei, die endlos zyklisch nichts tut. Man muß sie also abbrechen. Glücklicherweise ist sie darauf vorbereitet. Zuerst wird sie synchron gestartet. Sie reagiert dann nur auf die Unterbrechungstaste des Terminals. Der asynchrone Start läßt die Signale SIGTERM und SIGUSR2 auf den Nachfolgeprozeß einwirken, dagegen ignoriert er das

Signals SIGUSR1. Der Aufruf des Kommandos wait ohne Parameter zeigt, daß noch Hintergrundprozesse arbeiten. Betätigung der Unterbrechungstaste führt zur Ausschrift der Nummer unseres Nachfolgeprozesses.

```
1% cat traptst
: /bin/sh
rm -f file
trap 'echo Sig17 `date` >>file; echo Signal 17 >/dev/tty; exit' 17
trap 'echo Sig16 `date` >>file; echo Signal 16 >/dev/tty' 16
trap 'echo Sig15 `date` >>file; echo Signal 15 >/dev/tty' 15
trap 'echo Sig2  `date` >>file; echo Signal 2  >/dev/tty; exit' 2
trap 'echo Ende  `date` >>file; echo Ende!!    >/dev/tty; exit' 0
date; date >>file
while
:
do
:
done
2% traptst
Tue Aug  9 14:09:59 mesz 1988
    !! <delete>-Taste drücken!
Signal 2
Ende!!
3% cat file
Tue Aug  9 14:10:00 mesz 1988
Sig2 Tue Aug 9 14:10:04 mesz 1988
Ende Tue Aug 9 14:10:04 mesz 1988
4% traptst &
531
5% Tue Aug  9 14:10:21 mesz 1988
5% wait
    !! <delete>-Taste drücken!
   531  traptst
wait: Interrupted.
6% kill 531
Signal 15
7% kill -15 531
Signal 15
8% kill -16 531
9% kill -17 531
Signal 17
Ende!!
10% cat file .
Tue Aug  9 14:10:22 mesz 1988
Sig15 Tue Aug 9 14:10:48 mesz 1988
Sig15 Tue Aug 9 14:10:59 mesz 1988
Sig17 Tue Aug 9 14:11:27 mesz 1988
Ende Tue Aug 9 14:11:27 mesz 1988
11%
```

# 5. Das Technologieprogramm make

## 5.1. Elementare Leistungen

### 5.1.1. Das Grundanliegen von make

Softwaretechnologie will große oder komplexe Programme beherrschen helfen. Einer ihrer wirklich bewährten und vielleicht wesentlichsten Ansätze besteht darin, komplexe Aufgabenstellungen in Teile zu zerlegen, um durch deren Bewältigung das Gesamtproblem zu lösen. Programmiersprachen und ihre Implementationen tragen dem in unterschiedlicher Weise Rechnung. Sprachen wie Modula-2 oder Ada enthalten eigene Sprachelemente zur Abgrenzung von Moduln. Dort regeln sprachliche Mittel den Informationsaustausch zwischen den Moduln. Andere Sprachen - so auch die Programmiersprache C - gestatten die Übersetzung von Programmteilen, ohne die Bezüge zwischen Übersetzungseinheiten streng zu reglementieren. In allen Fällen schließen sich weitere Verarbeitungsschritte an, in denen etwa durch einen Programmverbinderlauf die getrennt bearbeiteten Teile zu einem startfähigen Programm zusammengefügt werden.

Daraus resultiert eine für größere Softwareprojekte typische Situation. Es müssen Objekte mit unterschiedlichem Bearbeitungsstand nebeneinander verwaltet werden. Hier liegt der Ansatzpunkt von make. Das Dienstprogramm make (Feldman 1978) agiert in einer Gesamtheit von Objekten. Objekte sind Dateien, reguläre Dateien, auch leere Dateien oder nur projektbezogene Bezeichnungen, die als Ziele von Bearbeitungsabläufen fungieren. Die Gesamtheit der von make verwalteten Dateien wird durch das aktuelle Arbeitsverzeichnis erschlossen. Grenzen für die Einflußnahme von make gibt es dabei kaum. Einen äußeren Rahmen stellt natürlich das Dateisystem des UNIX überhaupt dar.

Objekte können von Quellen abhängen. Sollen Objekte hergestellt werden, so müssen ihre Quellen vorhanden sein oder eben auch hergestellt werden. In einer Beschreibungsdatei makefile wird durch Regeln ausgedrückt, wie Objekte von ihren Quellen abhängen. Solchen Regeln können Kommandozeilen folgen, die Objekte aus ihren Quellen konstruieren. Das Ganze unterliegt dem Raster fortschreitender Zeit. Objekte mit Quellen sind genau dann gültig, wenn alle Quellen vorhanden, gültig, und älter als das Objekt sind. Objekte ohne Quellen sind gültig. Änderung oder Beseitigung einer Quelle macht das Objekt ungültig. Bei Dateien bezieht sich das Zeitraster auf die letzte Modifikation. Der Aufruf

```
% make ziel
```

stellt das Objekt mit dem Namen ziel her, wenn es nicht als gültiges Objekt vorhanden ist. Dazu bildet make eine Kommandofolge, die alle ungültigen Quellen rekonstruiert und schließlich das Objekt ziel erzeugt. Gültige Quellen werden so, wie sie sind, verwendet.

Damit ist die Nützlichkeit dieses technologischen Werkzeuges offenkundig. Man spart mit ihm Übersetzungszeit. Bei der Herstellung von Dokumentationen kann man damit Druckzeit und Papier sparen. Wenn man in seinem Softwareprojekt mit wenig Aufwand korrekte Verhältnisse schaffen muß, bietet make seine hilfreichen Dienste an. Man könnte blauäugig verlangen: An den Anfang eines jeden Softwareprojekts gehört ein makefile! Ohne so zu übertreiben, muß aber doch bereits in einem frühen Entwurfsstadium das technologische Konzept fixiert werden. Großproduzenten von Software können mit den Mitteln von make

technologische Standards vorschreiben. Die schrittweise Verfeinerung des Projektentwurfs erstreckt sich dann auch auf die iterierte Überarbeitung der make-Beschreibungsdatei.

Bemerkenswert ist es, daß der Gedanke des UNIX-make auch außerhalb dieses Betriebssystems eine weite Verbreitung gefunden hat. Implementationen von Programmiersprachen mit Modularisierungsmöglichkeiten bieten verwandte technologische Dienstprogramme an (m2make für Modula-2, Programmierumgebung von Ada). Und wenn Sie aufmerksam in diesem Buch zurückblättern, werden Sie feststellen, daß Ihnen das Programm make bereits im Abschnitt 1. begegnet ist.

### 5.1.2. Ein Beispiel, Beschreibungsdateien

Wir kommen auf ein Programmsystem zurück, das im Abschnitt 4.4. bereits den Anlaß für die Formulierung einer Kommandodatei gegeben hatte. Es handelt sich um ein Koroutinensystem, das innerhalb eines einzigen UNIX-Prozesses arbeiten soll. Es orientiert sich an dem Koroutinenkonzept von Modula-2. Die meisten Teilprogramme sind in C geschrieben. Die Umschaltung von einer Koroutine in eine andere erfordert hardwarenahe Maßnahmen und wurde daher, Comer (1984) folgend, in der Assemblersprache verfaßt. Folgende Dateien sind beteiligt:

| | |
|---|---|
| zge.c | Steuerndes Hauptprogramm. |
| bi.c pu.c pr.c | Problemorientierte Teilprogramme. |
| processes.c | Koroutinenverwaltung. |
| system.s | Umschaltapparat von einer zur anderen Koroutine. |
| *.h | #include-Dateien für den geordneten Bezeichner-Import. |

Weitere Objekte in diesem Softwareprojekt sind die getrennt übersetzten Teile *.o und als aufrufbares Programm zge selbst. Jetzt folgt ein Ausschnitt aus einer Datei makefile, wie sie in realer Umgebung nötig war. Die Zeilen wurden nachträglich numeriert.

```
(1)     SHELL = /bin/sh
(2)     ASFLAGS = -Mx
(3)     obj = zge.o bi.o pu.o pr.o processes.o system.o
(4)     .s.o:
(5)     <tab>   $(AS) $(ASFLAGS) $<
(6)
(7)     zge:            # Erzeuge Hauptprogramm zge
(8)     zge: $(obj)
(9)     <tab>   cc -o zge $(CFLAGS) $(obj)
(10)
(11)    zge.o: bi.h pu.h pr.h processes.h system.h
(12)    bi.o: pr.h processes.h system.h
(13)    pu.o: pr.h processes.h system.h
(14)    processes.o: processes.h system.h
```

Bevor wir den Aufbau dieser makefile-Datei näher studieren, wollen wir make aufrufen. Wir nehmen an, das Arbeitsverzeichnis erreicht folgende Dateien und keine weiteren: zge.c bi.[ch] pu.[ch] pr.[ch] processes.[ch] und system.[sh]. Dann erzeugt der Aufruf von make zge folgende Kommandos und arbeitet sie ab. Offenbar stammt nur das Letzte aus der Beschreibungsdatei. Hätte es noch gültige Dateien *.o gegeben, wären die entsprechenden C-Übersetzungen weggeblieben. Ein nochmaliger Aufruf von make mit dem gleichen Objekt als Ziel erbringt den Hinweis, das Objekt zge ist noch gültig.

```
% make zge
        cc -O -c zge.c
zge.c
        cc -O -c bi.c
bi.c
        cc -O -c pu.c
pu.c
        cc -O -c pr.c
pr.c
        cc -O -c processes.c
processes.c
        as -Mx system.s
        cc -o zge -O zge.o bi.o pu.o pr.o processes.o system.o
% make zge
`zge' is up to date.
%
```

Betrachten wir den obigen Auszug aus einer make-Beschreibungsdatei. Offensichtlich besitzt make einen Makromechanismus. Die Bezeichnung obj ist in Zeile (3) erklärt und steht für die Liste der *.o-Dateien. Mit $(obj) wird die Bezeichnung verwendet. Andere Makrobezeichnungen wie AS oder CFLAGS sind in der Datei nicht erklärt worden, sie sind vordefiniert. Wie das spätere Abarbeitungsprotokoll zeigt, ist AS=as und CFLAGS=-O. Der Schalter ASFLAGS wählt eine Option des Assemblers aus, die für die hier erörterten technologischen Zusammenhänge ohne Belang ist. Die Zeile (7) zeigt einen Kommentar. Kommentartexte werden durch # und <newline> begrenzt. In den Zeilen (8) und (11) bis (14) stehen Abhängigkeitsregeln. Betrachten wir diese etwas genauer.

Regeln haben die generellen Formen:

```
Objekte       Quellen

Objekt   [ Quellen ...]    Kommando

Objekt   [ Quellen ...] [; Kommando]
<tab>    Kommando
   ... usw. ...
<tab>    Kommando
```

Objekte dürfen mehrfach auf der linken Seite von Regeln dieser Art vorkommen. Höchstens einmal kann sich eine Kommandofolge anschließen, die das Objekt konstruiert. Diese Kommandofolge beginnt eventuell mit einem Kommando noch in der gleichen Zeile und kann aus mehreren Kommandozeilen bestehen. Eine zweite Sorte von Regeln hat zwei Doppelpunkte als Trennzeichen.

```
Objekte        Quellen ...

Objekte ... :: Quellen ... [; Kommando ]
<tab>    Kommando
   ... usw. ...
<tab>    Kommando
```

Hier dürfen für unterschiedliche Quellen zum gleichen Objekt verschiedene Kommandofolgen angegeben werden. Allerdings können Objekte immer nur in einer Regelsorte auf der linken Seite vorkommen. Die ::-Regeln werden im Abschnitt 5.2. vorgeführt.

Kommandozeilen im Anschluß an Regeln müssen mit einem <tab>-Zeichen beginnen. Sie können mit dem <backslash>-Zeichen fortgesetzt werden. Jede Kommandozeile cmd wird protokolliert, danach einem Kommandointerpreter als Zeichenkette überreicht und mit sh -ce cmd oder csh -ce cmd verarbeitet. make wählt den Interpreter, der durch die Umgebungsvariable SHELL erreicht wird. In den makefile-Dateien muß daher der Interpreter angegeben werden, für den die Kommandozeilen formuliert sind. In dem kleinen Beispiel ist das belanglos, weil die einfachen Kommandozeilen nicht an einen bestimmten Interpreter gebunden sind. Zudem kürzt make den Weg in die Compiler cc und as nochmals ab.

Die Regeln in den Zeilen (11) bis (14) gehen auf die #include-Dateien zurück, von denen die Objektkodedateien abhängen. Sie wurden bereits im Abschnitt 4.4. durch die Kommandodatei makescript erstellt. Hervorzuheben ist allerdings, daß zwar die Dateien *.o von den #include-Dateien abhängen, nicht jedoch die *.c-Dateien, wenngleich die #include-Präprozessoranweisungen in diesen Dateien stehen.

Es fällt überhaupt auf, daß alle Abhängigkeiten auf *.o-Dateien münden. Diese Dateien sollte es nach den früher genannten Voraussetzungen im Arbeitsverzeichnis gar nicht geben. Es müssen also noch weitere vordefinierte Regeln Berücksichtigung finden, nach denen *.o-Dateien aus den vorhandenen *.c-Dateien oder aus system.s gebildet werden.

In der Tat verfügt make über einen ganzen Satz eingebauter Transformationsregeln, die suffixgesteuert Objekte aus Quellen herstellen, wenn sie nicht gültig sind. In unserem Fall sind dies die Transformationsregeln

```
.c.o:
        $(CC) $(CFLAGS) -c $<
.s.o:
        $(AS) $(ASFLAGS) $<
```

Der interne Makroaufruf $< meint das Objekt, das transformiert werden muß, hier also beispielsweise pu.c oder system.s.

Welche Suffixe Transformationen von Objekten zulassen, wird durch die make-eigene Liste

```
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~
```

überwacht. Suffixe mit dem Zeichen ~ gehören zu Dateien aus dem Quelltextverwaltungssystem SCCS. Ansonsten kennzeichnen

| | |
|---|---|
| .o | Objektkodedateien, |
| .c | Quelltexte in C, |
| .y | Quelltexte in yacc, |
| .l | Quelltexte für lex, |
| .s | Quelltexte der Assemblersprache, |
| .sh | Kommandodateien, |
| .h | Einschlüsse, Spezifikationen oder ähnliches, |
| .a | Bibliotheken (mehrere Objektmoduln). |

Das folgende Bild 5.1 gemäß Bach u.a. (1986) verdeutlicht Abhängigkeiten, für die es in make eingebaute Transformationsregeln gibt.

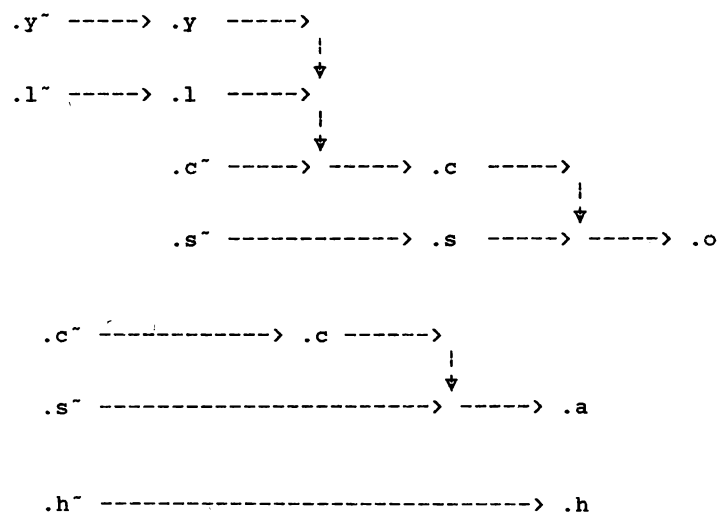



Bild 5.1. Suffix-Abhängigkeiten

Der aufmerksame Leser wird aber immer noch die Frage unbeantwortet wissen, warum in den Zeilen (4) und (5) der anfänglichen Beschreibungsdatei die Transformation s.o. nochmals aufgeführt wurde. Ganz allgemein setzen explizit formulierte Transformationsregeln die make-eigenen außer Kraft, was hier unmotiviert erscheint. Leider ist auch UNIX nicht frei von Unvollkommenem. So sind die eingebauten Regeln für die Transformation von Assemblertexten einfach falsch. (Das betrifft sowohl WEGA als auch XENIX286. Auch im make des XENIX386 sind diese und weitere Fehler enthalten.) Will man sich einen genauen Überblick über die make-eigenen Regeln verschaffen, so muß man folgende Kommandos starten:

```
% sh
$ env - make -fp /dev/null 2> /dev/null > built-in_rules
$ more built-in_rules
```

Für XENIX286 befindet sich das Protokoll dieser Kommandos im Anhang 4. Im Übrigen ist dies wieder eine Gelegenheit, bei der man auf den Umlenkungsapparat der Bourne-Shell angewiesen ist.

### 5.1.3. Makros und Ersetzungen

In dem anfänglichen Beispiel kamen bereits Makrodefinitionen und die Verwendung von Makrobezeichnungen vor. Eintragungen der Form

```
string1 = string2
```

sind Makrodefinitionen. Die Zeichenkette string2 reicht bis zum Beginn eines Kommentars oder bis zum nächsten gültigen <newline>-Zeichen. Die beiden Zeichen <backslash><newline> stellen ein <newline>-Zeichen in der Kette string2 dar. Anwendungen von Makros haben die Formen:

```
$(string1)
$(string1:substr1=[substr2])
```

In der ersten Form können die Klammern wegbleiben, wenn string1 aus nur einem Zeichen besteht. In der zweiten Form werden in string2 global Teilzeichenketten substr1 durch substr2 oder durch die leere Kette ersetzt.

Die Reihenfolge von Definitionen und Anwendungen der Makrobezeichner ist nicht vorgeschrieben. In einem ersten Durchlauf werden alle Makros gesammelt und danach erst substituiert. Bei mehrfachen Definitionen ist nur die letzte wirksam. Vordefinierte Makros erhalten durch explizite Makrodefinitionen in der Beschreibungsdatei eine neue Bedeutung.

Beim Aufruf von make werden die Umgebungsvariablen ebenfalls als Makrobezeichnungen verfügbar. Zudem werden Schlüsselwortparameter in die Makromenge aufgenommen.

Diese Vielfalt von Möglichkeiten zur Einführung von Makros erfordert Regeln für das Überschreiben bereits definierter Makros.

Abhängig von der Aufrufoption -e gelten die folgenden Prioritäten. Höchsten Vorrang haben die Schlüsselwortparameter.

| % make ( ohne -e ) | % make -e |
|---|---|
| 1. Schlüsselwortparameter | 1. Schlüsselwortparameter |
| 2. Definition explizit | 2. Umgebungsvariable |
| 3. Umgebungsvariable | 3. Definition explizit |
| 4. Vordefinition intern | 4. Vordefinition intern |

Es folgt ein kleines Terminalskript, in dem die Ersetzungsverhältnisse offengelegt werden. Am Anfang des Terminaldialogs soll es keine Umgebungsvariable CFLAGS geben. Die Option -n beim Aufruf von make bewirkt, daß die auszuführenden Kommandos zwar protokolliert, aber nicht gestartet werden.

```
% cat >! makefile << ende
x: x.c ; cc -c \$(CFLAGS) x.c
ende
% touch x.c
% make -n x
    cc -c -O x.c
% setenv CFLAGS "-LARGE -S"
% make -n x
    cc -c -LARGE -S x.c
% cat - makefile >! f << ende; mv f makefile
CFLAGS = -I path
ende
% make -n x
    cc -c -I path x.c
% make -en x
    cc -c -LARGE -S x.c
% make -n x CFLAGS='-S -O'
    cc -c -S -O x.c
%
```

Das sind zwar klare Verhältnisse. Wenn jedoch die C-Shell als standardmäßiger Kommandointerpreter für den Terminaldialog gewählt wurde, hat die Umgebungsvariable SHELL den Wert /bin/csh. Dann schaltet make -e eben auch auf diesen Interpreter um. Da dieses vermutlich als störend empfunden wird, hilft nur noch der Alias-Apparat der C-Shell, wie das folgende Skript demonstriert:

```
% echo $SHELL
/bin/csh
% cat makefile
SHELL=/bin/sh
h: ; @echo 'make benutzt $(SHELL)'
% make
make benutzt /bin/sh
% make -e
Neue C-Shell
make benutzt /bin/csh
% alias make 'make SHELL=/bin/sh'
% make -e
make benutzt /bin/sh
%
```

Wenn übrigens im Aufruf von make das Zielobjekt fehlt, wird als Ziel das erste Objekt der ersten Abhängigkeitsregel gebildet. Das Ampersandzeichen vor dem Kommando echo schaltet den Protokollmodus für diese Kommandozeile ab.

### 5.1.4. Interne Makros

Das make-Programm verwaltet intern fünf Kurzschriftmakros, die für die Gestaltung von Abhängigkeits- und Transformationsregeln recht hilfreich sind. Um beispielsweise das Terminalprotokoll im Anhang 4 zu verstehen, ist die Kenntnis dieser Makros notwendig.

**$@**
**$(@D) $(@F)**
Steht für den Namen des Zielobjekts. Der Zusatz D verkürzt auf den Verzeichnisteil (path_prefix), der Zusatz F extrahiert den Dateinamen (filename), wenn $@ ein entsprechender Pfadname ist.

**$?** - nur in Abhängigkeitsregeln -
Steht für die Liste aller ungültigen Quellen. Zusätze D oder F sind hier nicht möglich.

**$<** - nur in Transformationsregeln -
**$(<D) $(<F)**
Steht für das transformierbare Objekt. In Regeln der Form c.o.: wird $< durch object.c ersetzt. Die Zusätze D oder F wirken wie bei $@.

**$*** - nur in Transformationsregeln -
**$(*D) $(*F)**
Vom Namen des Objekts, das die Transformation auslöst, wird das Suffix entfernt. Der Rest ersetzt $*. Wird die Regel c.o.: durch das Objekt object.o ausgelöst, so steht $* für den Namen object. Die Zusätze D oder F wirken wie bei $@.

```
$%      - Zielobjekt in einer Bibliotheksdatei -
$(%D) $(%F)
    Hat das Ziel die Form lib.a(file.o), so bezeichnet $@ die
    Datei lib.a und $% die Datei file.o. Die Zusätze D oder F
    wirken wie vorher.
```

Von den Möglichkeiten interner Makros infiziert, überarbeiten wir gleich unsere anfängliche Beschreibungsdatei makefile. Die Zeilen (7) bis (9) dieser Datei ersetzen wir durch die folgenden:

```
zge:            # Erzeuge Hauptprogramm zge
zge: $(obj)
        cc -o $@ $(CFLAGS) $(obj)

files= zge.c bi.[ch] pu.[ch] pr.[ch] processes.[ch] system.[sh]

doc:            # Dokumentiere Quelltexte
doc: $(files)
        @-pr $? | lp
        @touch doc

.PRECIOUS: doc
```

Sofort überschaubar ist der Einsatz des Makros $@ als Zielobjekt des Programmverbinderlaufs, aufgerufen durch cc -o ...

Die weiteren Zeilen zeigen, wie man mit make Druckzeit und auch Papier sparen kann. Dazu wird die leere Datei doc verwaltet. Nur der Zeitpunkt ihrer letzten Modifikation, eines ihrer Attribute, ist von Interesse. Sind erst einmal alle Dateien gedruckt, so werden später nur solche ausgewählt, die sich seitdem geändert haben. Dies leistet der Makro $?. Er sucht ungültige Quellen.

Gleichzeitig wurde hier das Scheinobjekt .PRECIOUS eingesetzt. Die ihm anvertrauten Quellen werden gewissermaßen veredelt. Wenn die Arbeit von make unterbrochen wird, beispielsweise durch Signale SIGINT oder SIGQUIT, gehen die gerade bearbeiteten Zielobjekte verloren. Sie bleiben jedoch erhalten, wenn .PRECIOUS von ihnen abhängt. Nun ist die Datei doc wohl nur eine leere, aber ihre Zeitpunktattribute sind überaus wertvoll und teuer. Sie muß vor Verlust geschützt werden.

Es folgt wieder ein Skript, in dem mit dem Kommando touch fortschreitende Zeit vorgegaukelt wird.

```
% touch *.[csh]
% make -n doc
    pr zge.c bi.h bi.c      processes.c system.s system.h   lp
    touch doc   .
% touch doc
% touch zge.c system.h
% make -n doc
    pr zge.c system.h   lp
    touch doc
%
```

### 5.1.5. Zur Arbeitsweise von make

Beschreibungsdatei

Ein Aufruf von make verlangt eine Beschreibungsdatei, in der Abhängigkeitsregeln, Kommandozeilen oder weitere Informationen versammelt sind. Ohne die Option -f sucht make im Arbeitsverzeichnis nach Dateien mit den Namen makefile oder Makefile bzw. s.makefile oder s.Makefile, immer in dieser Reihenfolge. Die beiden letzten Namen führen in das Quelltextverwaltungssystem SCCS. Bei Aufrufen der Form

```
% make -f name1 -f name2
```

wird die Verkettung der Dateien name1, name2, ... als Beschreibungsdatei genommen. Mit -f - kann auch die Standardeingabe als Quelle für Beschreibungsinformationen dienen.

Gestaltung von make-Protokollen

Wenn man keine besonderen Vorkehrungen trifft, werden alle Kommandozeilen protokolliert, die aufgrund der Abhängigkeitsverhältnisse abgearbeitet werden müssen.

Kommandozeilen, die mit dem Zeichen @ beginnen, werden nicht ausgeschrieben. Beim Aufruf mit der Option -n werden sie jedoch sichtbar gemacht. Aufrufe make -n erzeugen dadurch fast fertige Kommandodateien, mit denen dann das Zielobjekt auch ohne make konstruiert werden kann.

Wird make mit der Option -s aufgerufen, werden Kommandozeilen überhaupt nicht protokolliert. In gleicher Weise schweigsam arbeitet make, wenn es in der Beschreibungsdatei die Regel .SILENT: mit dem Scheinobjekt .SILENT gibt, das natürlich von keiner Quelle abhängig ist.

Verhalten bei Fehlern

Wenn bei der Abarbeitung von Kommandos Fehler auftreten, also exit-Werte $\neq 0$ zurückgegeben werden, bricht make die Arbeit ab. Der verursachende exit-Wert wird in der Fehlernachricht mitgeteilt.

Beginnt eine Kommandozeile mit einem Minuszeichen, so wird die Arbeit auch bei einem Fehler in einem Kommando dieser Zeile fortgesetzt.

Aufrufe von make mit der Option -i garantieren die Fortsetzung nach Fehlern an beliebiger Stelle. Es werden ebenfalls alle Fehler ignoriert, wenn die Beschreibungsdatei die Regel .IGNORE: mit dem Scheinobjekt .IGNORE enthält.

Anders beeinflußt die Option -k das Verhalten bei Fehlern. Hier wird die fehlerbefallene Kommandoabarbeitung zur Konstruktion eines Objekts abgebrochen. Wenn jedoch zur Herstellung des Zielobjekts weitere ungültige Quellen rekonstruiert werden müssen, setzt make die Bearbeitung mit dem nächsten Quellobjekt fort.

Kommandoprotokollierung und Fehlerverhalten können gemeinsam durch Vorsätze an der Kommandozeile beeinflußt werden. Beide Kombinationen @- und -@ sind möglich und gleichwertig.

Scheinobjekt .DEFAULT

Wenn Abhängigkeitsregeln auf Objekte führen, die es gar nicht gibt, reagiert make mit der Ausschrift »Don't know how to make ...«. Man kann jedoch das Scheinobjekt .DEFAULT in die Beschreibungsdatei aufnehmen und Kommandos formulieren, die zur Bildung von unbekannten Objekten abgearbeitet werden sollen. Hierfür zeigen wir ein kleines Beispiel:

```
% cat makefile
a : b
% make
Make:  Don't know how to make b.  Stop.
% cat - makefile >! f <<ende ; mv f makefile
.DEFAULT:
<tab>    cp project/archive/\$< .
<tab>    @echo project/archive/\$< nach . kopiert.
ende
% make
        cp project/archive/b .
project/archive/b nach . kopiert.
%
```

Übersicht über die Zieleinträge

Den Beschreibungsdateien für make kommt eine technologische Schlüsselrolle zu, wenn es darum geht, größere Softwareprojekte betriebstüchtig zu gestalten. Wie schlimm mag es sich ausnehmen, wenn man nach längerem die ehemals sorgfältig gewählten Bezeichnungen für Zielobjekte nicht mehr ganz genau weiß. Bevor es richtig losgeht, muß dann erst einmal in den Beschreibungsdateien nachgelesen werden.

Attraktiv und wirkungsvoll ist daher ein Vorschlag von Schreiner (1987), als erste Regel in einem makefile immer ein Hilfeziel zu erklären:

```
help:   ;@egrep '^[^:;=.]*::?[ <tab>]*#' [mM]akefile
```

Wenn man dann alle Zielobjekte konsequent kommentiert hat, erreicht man mit dem ersten Aufruf von make ohne Parameter eine aussagekräftige Übersicht über alle vorbereiteten Ziele. Die Gefahr, versehentlich Schaden zu stiften, ist dadurch gewiß etwas gemindert. Ohne Aufhebens wurde die Beschreibungsdatei des einleitenden Beispiels bereits nach diesem Muster gestaltet. So wäre bei ihrer zweiten Fassung folgender Ablauf zu beobachten:

```
% make
zge:            # Erzeuge Hauptprogramm zge
doc:            # Dokumentiere Quelltexte
% make zge
`zge' is up to date.
%
```

Aufruf von make und Zusammenstellung der Optionen:

Das Kommando

```
% make options ziel1 ziel2
```

konstruiert der Reihe nach die Objekte ziel1, ziel2 usw. Wenn kein Ziel angegeben ist, bildet make das erste Zielobjekt der ersten Abhängigkeitsregel. Folgende Optionen beeinflussen die Arbeitsweise von make. In der Argumentliste können Optionen und Ziele gemischt angeordnet sein.

-f name
Die Datei name enthält Beschreibungsdaten.

-p Ausgabe aller Makrodefinitionen und Objektbeschreibungen.

-i Fortsetzung bei Fehlern.

-k Überbrückung fehlerhafter Objekte, Fortsetzung mit der nächsten Quelle.

-s Ohne Kommandoprotokoll.

-r Ausschalten interner Regeln.

-n Kommandoprotokoll ohne Abarbeitung (statisches Protokoll).

-e Umgebungsvariable überschreiben Makrodefinitionen.

-t Aktualisiere Zielobjekte.

-q Fragemodus. make gibt den exit-Wert 0 zurück, wenn das Zielobjekt gültig ist, andernfalls einen Wert ≠ 0.

-b Auch auf ältere Versionen von make abgestimmte Beschreibungsdateien werden akzeptiert.

### 5.1.6. Installation von Software

UNIX selbst benutzt den Apparat von make zur Installation von Softwarepaketen. Der Installation einer Version des Programmgenerators yacc liegt die Beschreibungsdatei /usr/src/cmd/yacc/makefile zugrunde. Sie wird als Beispiel auszugsweise mitgeteilt. Leider sind in den neueren UNIX-Versionen die Quelltexte der Kommandos und Systemprogramme nicht mehr abrufbar, so daß das Studium anderer ebenso informativer makefile-Dateien nicht so ohne weiteres möglich ist.

```
all: yacc liby.a
        :

cmp: all
        cmp yacc /bin/yacc
        ls -l liby.a /lib/liby.a
        rm yacc *.o liby.a

cp:  all
        cp yacc /bin/yacc
        cp liby.a /lib
        rm yacc liby.a *.o

yacc: y1.o y2.o y3.o y4.o
        cc $(CFLAGS) -s -o yacc y?.o

y1.o y2.o y3.o y4.o: dextern files
```

```
liby.a:
        cc -c $(CFLAGS) lib/main.c
        cc -c $(CFLAGS) lib/yyerror.c
        rm -f liby.a
        ar rvc liby.a main.o yyerror.o
        rm main.o yyerror.o
```

Die Quelltexte für yacc befinden sich in den Dateien y[1-4].c. Dazu kommen zwei Einschlüsse dextern und files sowie das Verzeichnis lib mit den beiden C-Quelltexten für die späteren Bibliotheksfunktionen main() und yyerror().

Die Beschreibungsdateien für die Installationsaktivitäten der alten UNIX-Werkzeuge sind natürlich lange Zeit vor der Schreinerschen Kommentierungsidee entstanden, so daß einem hier sozusagen die klassische Form von makefiles gegenübersteht.

In solchen makefiles nimmt immer das fiktive Zielobjekt all die Führungsposition ein. Hier erzwingt eine Konstruktion von all die Herstellung der beiden Quellobjekte yacc und liby.a. Das erste Objekt ist die abarbeitungsfähige Form des Kommandos yacc, das zweite ist die yacc-eigene Bibliothek liby.a, auf die mit dem Aufruf cc -ly Bezug genommen werden kann.

In späteren UNIX-Versionen sind diese Dinge in ihrem Umfang beträchtlich gewachsen. So gibt es die Bibliothek in drei Versionen [LMS]liby.a, und auch das Kommando selbst kann als yacc oder als yaccL aufgerufen werden.

In dem technologischen Konzept der Installations-makefiles findet man meist wie hier die Einträge cmp und cp.

Das Objekt cmp ist ein Testeintrag, gut geeignet für den Fragemodus beim make-Aufruf. Man geht davon aus, daß das Kommando yacc normalerweise in dem zentralen Verzeichnis /bin und die Bibliothek liby.a im Verzeichnis /lib verwaltet wird. Wenn das möglicherweise neuhergestellte Programm yacc mit dem Programm in /bin byteweise übereinstimmt und beide Bibliotheken als Dateien vorhanden sind, werden alle (!) Dateien *.o und die Objekte yacc und liby.a wieder beseitigt. Wenn die Kommandozeile cmp eine Abweichung feststellt, bricht make die weitere Arbeit am Objekt cmp ab, ebenso, wenn eine der beiden Bibliotheksdateien fehlt. (ls bringt den exit-Wert 2, wenn es gefragte Dateien nicht findet.) Nach einem Abbruch liefert make cmp -q einen exit-Wert ≠ 0. Letzteres kann auch passieren, wenn es gar keine Dateien *.o gibt. Da wäre es günstiger, rm -f zu notieren, und wenn make gar die C-Shell bemüht, würde erst -rm helfen.

Mit dem Eintrag cp werden die neuhergestellten Objekte in die Verzeichnisse /bin und /lib kopiert. Für die Aufräumzeile gelten die gleichen Bemerkungen wie im Falle von cmp.

Eine Anwendung dieser makefile-Datei könnte so aussehen:

```
$ cd /usr/src/cmd/yacc
$ make cmp -q || make cp
$
```

### 5.1.7. Erzeugung von Objekten aus suffigierten Quellen

Solange beim Aufruf von Compilern wiederkehrende Standardabläufe ausreichen, kann man erneut make strapazieren. Bei den mannigfaltigen Beispielen in diesem Buch ist überhaupt noch kein Compiler explizit bei seinem Namen gerufen worden. Sollte ein C-Programm prog.c abarbeitbar aufbereitet werden, so reichte das Kommando make prog. In den Abschnitten 1. und 2.2. wurde das PASCAL-Programm accu.pas mit dem Kommando make accu in eine ausführbare Datei umgeformt. Dazu war freilich schon die Datei Paccu/makefile nötig. Bei der Bearbeitung von C-Programmen braucht man nicht einmal eine Beschreibungsdatei.

make kann ohne eine Beschreibungsdatei Objekte herstellen, wenn es dafür Dateien mit Namenssuffixen aus der make-eigenen Suffixliste gibt. Für das Suffix selbst muß es eine einstellige Transformationsregel geben.

Die interne Suffixliste und eingebaute Transformationsregeln können mit dem Kommando make -p sichtbar gemacht werden. Noch einmal wird auf den Anhang 4 verwiesen, der einen kompletten Ausdruck für das XENIX-make zeigt.

Für die Transformation von C-Quelltexten ist die Regel

```
.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
        -rm -f $*.o
```

zuständig. Sie läßt noch vieles offen. Selbst der Compiler kann als Makroname vorgegeben werden. Für die Belegung der Optionen kann der ganze Ersetzungsapparat von make bewegt werden. Schlüsselwortparameter, vielleicht mit alias versteckt, oder Umgebungsvariable bieten einen breiten Spielraum.

Die Verarbeitung von PASCAL-Programmen oder von Programmen anderer höherer Programmiersprachen ist beispielsweise im XENIX-make nicht vorgesehen. Dann geht es nicht ohne makefile-Dateien. Für bescheidene Ansprüche leistet aber schon die folgende Datei beim Aufruf des PASCAL-Systems eine akzeptable Verkürzung.

```
% cat Makefile
PASCAL=pascal
PASCFLAGS=

.SUFFIXES: .pas
.pas:
    $(PASCAL) $(PASCFLAGS) $*
    -rm $*.c $*.o
%
```

Wie bereits im Abschnitt 1. vermerkt wurde, handelt es sich hier um eine PASCAL-Implementation, die einen Zwischentext in der Sprache C erzeugt, der am Ende wieder beseitigt werden muß, genauso wie der entsprechende Objektmodul. Die Beschreibungsdatei zeigt das Vorgehensschema. Für andere Implementationen muß man passende Regeln aufschreiben. Mit dem Scheinobjekt .SUFFIXES wird die make-eigene Suffixliste verlängert. Die einstellige Transformationsregel .pas: erweitert die Menge der bereits vorhandenen Transformationsregeln.

Mit der Abhängigkeitszeile

```
.SUFFIXES:
```

mit einer leeren rechten Seite wird die alte Suffixliste als ungültig erklärt.

make ist auch darauf eingestellt, daß man Kommandodateien mit suffigierten Namen bezeichnet. In der internen Suffixliste befinden sich die Einträge .sh und .sh~. Für das Suffix .sh gibt es die Transformationsregel:

```
.sh:
        cp $< $@
```

Die zweite Form greift wieder auf das Quelltextverwaltungssystem SCCS zurück. make kopiert die Originaldatei suffixfrei und startet sie erst dann als Kommandodatei. Das folgende Fragment demonstriert die Sachlage:

```
% ls cmd*
-rwxr-xr--  ...  cmd.sh
% cat makefile
a   cmd
        cmd ... a ...
%
```

### 5.1.8. Eine Beschreibungsdatei für dieses Buch

Wie schon in früheren Abschnitten bietet die Erarbeitung des Manuskripts für dieses Buch mit Hilfe von UNIX die Gelegenheit, das Werkzeug make anzuwenden und eben auch zu demonstrieren. Dieser Abschnitt 5. wird in der Datei ch5.mu beschrieben.

```
.nr H1 5
.so chhead.mm
.so ch51.mm
.so ch52.mm
.TC
```

Sie drückt in der ersten Zeile aus, daß es sich um den Abschnitt 5. handelt. Die letzte Zeile erzeugt das Inhaltsverzeichnis. Es bleiben die drei nroff-Einschlußanweisungen .so name. Die erste hat diverse Steueranweisungen zum Inhalt, die anderen beiden enthalten den Text für die Abschnitte 5.1. und 5.2. Nun war im Laufe des Abschnitts 4. schon des öfteren davon die Rede, daß für die Verwendung von Umlauten und Sonderzeichen zuerst der Präprozessor upr den originalen Text durchmustert. Mit den Kommandos

```
% upr <ch51.mu >ch51.mm
% upr <ch52.mu >ch52.mm
```

erzeugt er die nroff-gerechten Einschlußdateien. Die Textteile müssen also vor ihrer Formatierung durch nroff selbst erst einmal transformiert werden. Der formatierte Text ist von den transformierten Teilen abhängig. Am Ende werden auf das nroff-Resultat Postfilter aufgesetzt, die den verschiedenen Ausgabegeräten, Druckern oder Terminals, passend zuarbeiten.

Dieser Situation ist das Leistungsvermögen von make geradezu auf den Leib geschnitten. Und was für den Abschnitt 5. gut ist, gilt für die anderen Manuskriptteile schon lange. Also gehen wir daran, die Beschreibungsdatei gleich so zu gestalten, daß die Abschnittsnummer als Parameter oder Umgebungsvariable vorgegeben werden kann.

Für die Arbeit des Präprozessors upr führen wir die Suffixe .mm und .mu zusammen mit einer entsprechenden Transformationsregel ein. Als Zielobjekte wählen wir u für das Formatierungsprodukt ohne Umlautpostfilter und als ein Beispiel das Objekt uu für die Aufbereitung des Textes im erweiterten ASCII-Zeichenkode.

Zugleich gibt dieses Ansinnen die Gelegenheit, rekursive Aufrufmöglichkeiten von make zu demonstrieren. Es werden die beiden Makrobezeichnungen MAKE und MAKEFLAGS verwendet. MAKE ist mit make vordefiniert. Verwendet man diesen Makro, so wird beim Aufruf make -n die Kommandozeile mit $(MAKE) nicht nur ausgegeben, sondern auch gestartet. MAKEFLAGS sammelt alle Optionen, die beim Passieren von Rekursionsstufen angetroffen werden. Würde man statt $(MAKE) das Kommando make selbst angeben, so entfiele diese durchreichende Protokollverarbeitung.

Es folgt eine makefile-Datei für die Abschnitte 1. bis 5. dieses Buches.

```
% cat makefile
SHELL=/bin/sh
NR=nroff -rL72 -rW69 -mm ch$(c).mm 2> ch$(c).key

N1=
N2=ch21.mm ch22.mm ch23.mm ch24.mm
N3=ch31.mm ch32.mm ch33.mm
N4=ch41.mm ch42.mm ch43.mm ch44.mm ch45.mm
N5=ch51.mm ch52.mm

.SUFFIXES: .mm .mu
.mu.mm:
        upr < $< > $@

help:

u:      # Formatiere Kapitel ch$(c)u (ohne Umlautpostfilter)
u:      ; $(MAKE) -$(MAKEFLAGS) ch$(c)u

uu:     # Formatiere Endfassung ASCII-ext ch$(c)uu
uu:     ; $(MAKE) -$(MAKEFLAGS) ch$(c)uu

ch$(c)u: ch$(c).mm chhead.mm
        $(NR) > $@
        rm -f ch$(c)*.mm

ch$(c)uu: ch$(c)u
        upo < ch$(c)u > $@

ch1u: $(N1)
ch2u: $(N2)
ch3u: $(N3)
ch4u: $(N4)
ch5u: $(N5)
```

```
help: ; @echo'        # c=$c'
        @egrep '[:;=.]*::?[ <tab>]*#' [mM]akefile
%
```

Man muß feststellen, daß die Einschlußabhängigkeiten denen gleichen, die bei #include-Dateien in C-Quelltexten auftreten. So sind auch hier nicht die Dateien ch?.mm von den ch?*.mm abhängig, sondern die Endprodukte ch?u, wenngleich sich die .so ch?*mm-Anweisungen in den Dateien ch?.mm befinden. Die Arbeit gestaltet sich sehr bequem, wenn wir einige der make-Möglichkeiten ausnutzen.

```
% setenv c 5
% make
        # c=5
u:      # Formatiere Kapitel ch$(c)u (ohne Umlautpostfilter)
uu:     # Formatiere Endfassung ASCII-ext ch$(c)uu
% make uu
        make -b ch5uu
        upr < ch5.mu > ch5.mm
        upr < ch51.mu > ch51.mm
        upr < ch52.mu > ch52.mm
        nroff -rL72 -rW69 -mm ch5.mm 2> ch5.key > ch5u
        rm -f ch5*.mm
        upo < ch5u > ch5uu
%
```

## 5.2. Einige Sonderleistungen

### 5.2.1. Umgang mit Bibliotheken

Bibliotheken vereinen mehrere Objektmoduln in einer Datei. Wenn beim Laden oder Verbinden von Moduln Bibliotheken einbezogen werden, stehen alle darin auffindbaren externen Symbole für die Bindung von Referenzen zur Verfügung.

Zur Demonstration der Verhältnisse greifen wir auf das einführende Beispiel von Abschnitt 5.1.2. zurück. Wir nehmen an, daß im Arbeitsverzeichnis alle Objektkodedateien zge.o, bi.o, pu.o, pr.o, processes.o und system.o vorhanden sind. Dann haben die folgenden Kommandos Sinn:

```
% ar r lib.a {bi,pu,pr,processes,system}.o
% ranlib lib.a
% cc -o zge zge.o lib.a
```

Mit dem ersten Kommando wird die Bibliothek lib.a erzeugt. Sie nimmt alle Moduln auf, die dem steuernden Hauptprogramm unseres kleinen Beispiels nachgeordnet sind. Das Kommando ranlib stellt für die eben gebildete Bibliothek eine Symboltabelle her, auf die der im letzten Kommando verdeckt aufgerufene Lader ld zurückgreifen kann. Nun sollen diese Dinge von make berücksichtigt werden.

In Beschreibungsdateien werden Objekte aus Bibliotheken durch die Konstrukte bibl-name(objm-name) ausgewählt. Die Abhängigkeitsregeln müssen entsprechend gestaltet werden. Das Hauptprogramm hängt vom Objektmodul zge.o und von der Bibliothek lib.a ab. Am Anfang sind jedoch im Arbeitsverzeichnis keine .a-Dateien vorhanden. Statt dessen gibt es für die Quellen von lib.a einige passende .c-Dateien und die Datei system.s. Zu den eingebauten Transformationsregeln gehört die Regel .c.a: (s. Anhang 4). Sie hier aufzuschreiben ist

also eigentlich überflüssig. Dagegen fehlt die Regel .s.a: im make-eigenen Transformationsapparat, so daß diese jedenfalls in der Beschreibungsdatei erscheinen muß.

```
1% cat makefile
SHELL = /bin/sh
ASFLAGS = -Mx

.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
.s.a:
        $(AS) $(ASFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o

files = zge.c bi.[ch] pu.[ch] pr.[ch] processes.[ch] system.[sh]

help:
zge:            # Erzeuge Hauptprogramm zge
zge: zge.o lib.a
        ranlib lib.a
        cc -o zge $(CFLAGS) $(LDFLAGS) zge.o lib.a

doc:            # Dokumentiere Quelltexte
doc: $(files)
        @-more $?
        @touch doc

zge.o: bi.h pu.h pr.h processes.h system.h

lib.a(bi.o): pr.h processes.h system.h
lib.a(pu.o): pr.h processes.h system.h
lib.a(processes.o): processes.h system.h

lib.a: lib.a(bi.o) lib.a(pu.o) lib.a(pr.o) lib.a(processes.o)
lib.a: lib.a(system.o)

help:   ;@egrep '^[^:;=.]*::?[ <tab>]*#' [mM]akefile

.PRECIOUS: lib.a
```

Das anschließende Skript zeigt einige Aufrufprotokolle. Mit dem Kommando touch wird die zeitliche Änderung der Situation provoziert. Die Protokollausschriften stammen vom C-Compiler und vom Kommando ar. (Die Aufrufe von ar wurden in den Transformationsregeln mit der Option -v ausgestattet. a - kommt von appends, r - von replaces.) Das Skript wurde nachträglich kommentiert.

```
2% make zge
        cc -O -c zge.c
zge.c
        cc -c -O bi.c
bi.c
        ar rv lib.a bi.o
ar: creating lib.a
a - bi.o
        rm -f bi.o
        cc -c -O pu.c
pu.c
```

```
        ar rv lib.a pu.o
a - pu.o
        rm -f pu.o
        cc -c -O pr.c
pr.c
        ar rv lib.a pr.o
a - pr.o
        rm -f pr.o
        cc -c -O processes.c
processes.c
        ar rv lib.a processes.o
a - processes.o
        rm -f processes.o
        as -Mx system.s
        ar rv lib.a system.o
a - system.o
        rm -f system.o
        ranlib lib.a
        cc -o zge -O  zge.o lib.a
3% make -q zge ; echo $status
0
4% touch system.[sh]
5% make -s zge
zge.c                       # wegen zge.o: system.h
bi.c                        #
r - bi.o                    # wegen bi.o: system.h
pu.c                        #
r - pu.o                    # wegen pu.o: system.h
processes.c                 #
r - processes.o             # wegen processes.o : system.h
r - system.o                # wegen lib.a : system.s
6% !3
make -q zge ; echo $status
0
7% touch bi.[ch]
8% !5
make -s zge
zge.c                       # wegen zge.o: bi.h
bi.c                        #
r - bi.o                    # wegen lib.a: bi.c
9% !3
make -q zge ; echo $status
0
10%
```

In einer Variation der Beschreibungsdatei verwenden wir Abhängigkeitsregeln mit dem zweifachen Doppelpunkt. Mit solchen Regeln können für die einzelnen Abhängigkeiten passend formulierte Kommandofolgen verbunden werden. Bibliotheken müssen keine Suffixe tragen. Transformationsregeln werden in dieser Variante überhaupt nicht benötigt.

Die Abhängigkeiten für lib:: werden in der Reihenfolge betrachtet, wie sie in der Beschreibungsdatei erscheinen. Wenn die Quellen in einer Regel gültig sind, entfällt die Aktivierung der angehängten Kommandofolge. Beachtenswert ist es, daß die letzte Regel bei jeder Änderung von Quellen der Bibliothek lib das Kommando ranlib startet. Das ist auch notwendig, solange nicht bei jedem Aufruf von ar auch die zugehörige Symboltabelle aktualisiert wird. Im Laufe der UNIX-Weiterentwicklung ist dies zwar angekündigt, aber noch nicht durchgängig realisiert.

```
1% cat makefile
SHELL = /bin/sh
ASFLAGS = -Mx

libfiles = bi.[ch] pu.[ch] pr.[ch] processes.[ch] system.[sh]
files = zge.c $(libfiles)

help:

r:              # rm -f zge zge.o lib
        rm -f zge zge.o lib

doc:            # Dokumentiere Quelltexte
doc: $(files)
        @-more $?
        @touch doc

zge.o: bi.h pu.h pr.h processes.h system.h

lib::           # Korrekturen der Bibliothek
lib:: system.s
        $(AS) $(ASFLAGS) system.s
        ar r lib system.o && rm -f system.o
lib:: bi.c pr.h processes.h system.h
        $(CC) -c $(CFLAGS) bi.c
        ar r lib bi.o && rm -f bi.o
lib:: pu.c pr.h processes.h system.h
        $(CC) -c $(CFLAGS) pu.c
        ar r lib pu.o && rm -f pu.o
lib:: pr.c
        $(CC) -c $(CFLAGS) pr.c
        ar r lib pr.o && rm -f pr.o
lib:: processes.c system.h
        $(CC) -c $(CFLAGS) processes.c
        ar r lib processes.o && rm -f processes.o
lib:: $(libfiles)
        ranlib lib

zge:            # Erzeuge Hauptprogramm
zge: zge.o lib
        cc -o zge zge.o lib

help:   ;@egrep '^[^:;=.]*::?[ <tab>]*#' [mM]akefile

.PRECIOUS: lib
```

Wieder werden mit einem Skript einige make-Aufrufe vorgeführt. Das Kommando ar arbeitet diesesmal schweigsam. So ist das Terminalprotokoll ein biβchen kürzer.

```
2% make zge
        cc -O -c zge.c
zge.c
        as -Mx system.s
        ar r lib system.o && rm -f system.o
ar: creating lib
        cc -c -O bi.c
bi.c
        ar r lib bi.o && rm -f bi.o
```

```
        cc -c -O pu.c
pu.c
        ar r lib pu.o && rm -f pu.o
        cc -c -O pr.c
pr.c
        ar r lib pr.o && rm -f pr.o
        cc -c -O processes.c
processes.c
        ar r lib processes.o && rm -f processes.o
        ranlib lib
        cc -o zge zge.o lib
3% make -q zge ; echo $status
0
4% touch pr.h
5% make zge
        cc -O -c zge.c                  # wegen
zge.c                                   # zge.o: pr.h
        cc -c -O bi.c                   #
bi.c                                    # wegen
        ar r lib bi.o && rm -f bi.o     # lib:: bi.c pr.h
        cc -c -O pu.c                   #
pu.c                                    # wegen
        ar r lib pu.o && rm -f pu.o     # lib:: pu.c pr.h
        ranlib lib
        cc -o zge zge.o lib
6% !3
make -q zge   echo $status
0
7%
```

### 5.2.2. Rückgewinnung von Speicherplatz

Wir kommen jetzt auf die Beschreibungsdatei für dieses Buch zurück, so wie sie im Abschnitt 5.1.8. gezeigt wurde. Originale Eingabetexte wurden zuerst transformiert, danach formatiert und anschließend nochmals umgeformt. Dabei entstanden die Zwischentexte ch5*.mm, die mit der Kommandozeile rm -f ... in der Regel ch$(c)u: ... wieder beseitigt wurden, um den ja nicht unbeträchtlichen Plattenspeicherplatz wieder freizugeben.

In früheren Bezügen auf die Manuskriptentstehung, im Abschnitt 4., wurden die Transformationstexte über Pipes geleitet. Diese belegen außerhalb ihrer aktiven Arbeitsphase keinen Speicherplatz. Dafür können sie aber nur einen Ausgang und einen Eingang bedienen. Hier mußten wir mehrere Quellobjekte berücksichtigen, um ein ganzes Buchkapitel zu formatieren. Also benutzten wir make.

Die Beseitigung der *.mm-Dateien hat aber die unangenehme Nebenwirkung, daß dadurch die konstruierten Objekte ch?u ungültig werden. Startet man make mit dem Ziel uu, so beginnt die Arbeit doch wieder von vorn, einschließlich der zeitaufwendigeren Formatierung, auch wenn die Dateien ch?u noch oder schon vorhanden sind. Wir sollten daher make überlisten.

Das gelingt mit der folgenden Beschreibungsdatei. Wiederum werden die Zwischentexte beseitigt, aber sofort danach werden sie als leere Dateien wiederhergerichtet. Am Ende werden noch die Zeitstempel in der richtigen Reihenfolge aufgedrückt, so daß alle Objekte der Abhängigkeitshierarchie wieder gültig sind.

Zur Beseitigung auch der leeren Dateien, vielleicht am Ende einer Terminalsitzung, wurde das Zielobjekt r in die Beschreibungsdatei aufgenommen. Leider hat aber die Sache einen weiteren Haken. Wenn man einen Unterabschnitt ediert, bleiben die anderen Unterabschnitte und damit auch die Objekte, für die sie Quellen sind, weiterhin gültig. make bildet sie daher nicht neu, und der Formatierer schließt leere Dateien ein. Als Ausweg wurde hierfür noch das Zielobjekt t formuliert, mit dem man mit Sicherheit alle Objekte ungültig macht, die von den Originaldateien abhängen. Das darf man aber nicht vergessen.

```
% cat makefile
SHELL=/bin/sh
NR=nroff -rL72 -rW69 -mm ch$(c).mm 2> ch$(c).key

N1=
N2=ch21.mm ch22.mm ch23.mm ch24.mm
N3=ch31.mm ch32.mm ch33.mm
N4=ch41.mm ch42.mm ch43.mm ch44.mm ch45.mm
N5=ch51.mm ch52.mm

.SUFFIXES: .mm .mu
.mu.mm:
        upr < $< > $@

u:      # Formatiere Kapitel ch$(c) (ohne Umlautpostfilter)
u:     ; $(MAKE) -$(MAKEFLAGS) ch$(c)u

uu:     # Formatiere Endfassung ASCII-ext ch$(c)
uu:    ; $(MAKE) -$(MAKEFLAGS) ch$(c)uu

t:      # touch ch$(c)*.mu
        touch ch$(c)*.mu

r:      # rm -f ch$(c)*.mm  # (ev. leere Dateien)
        rm -f ch$(c)*.mm

ch$(c)u: ch$(c).mm chhead.mm

ch1u: $(N1)
        $(NR) > $@
        rm -f $(N1) ch1.mm; > ch1.mm
        touch $(N1) $@
ch2u: $(N2)
        $(NR) > $@
        rm -f $(N2) ch2.mm; > ch2.mm
        touch $(N2) $@

        usw.

ch5u: $(N5)
        $(NR) > $@
        rm -f $(N5) ch5.mm; > ch5.mm
        touch $(N5) $@

ch$(c)uu: ch$(c)u
        upo <ch$(c)u > $@
%
```

Daß diese Lösung manuelle Zusätze benötigt, gereicht ihr sicher zum Nachteil. Aber auch formale Gründe rechtfertigen eine weitere Modifikation. War doch die erste Variante im Abschnitt 5.1.8. über die Numerierung der Kapitel parametrisiert. In der vorgestellten Lösung mußten für jedes Kapitel einzelne Abhängigkeitsregeln aufgeschrieben werden. Der Grund dafür liegt in den bescheidenen Möglichkeiten des Ersetzungsapparats von make. Die Aufzählung der Unterabschnitte in jedem Kapitel würde eine zweite Ersetzungsstufe erfordern, wie man halt generell für die Zugriffe auf Listen von Listen indirekte Techniken benötigt.

In der folgenden Beschreibungsdatei teilen sich make und sh in diese zweistufige Arbeit. Die Bezeichnungen N2..N5 sind Makronamen von make, dagegen sind T2..T5 Shell-Variable. Beim Aufruf $(MAKE) mit c=5 hat make bereits die Bezeichnung T5 gebildet. Die Bourne-Shell wählt T5 aus der vorangestellten Reihe der Shell-Variablen T2..T5.

Im Übrigen erfordert diese Variante die oben erörterten manuellen Eingriffe nicht mehr. Mit dem Kommando find wird die Gültigkeitslage erkundet, und entsprechende Maßnahmen werden ausgelöst. Mit dem Kommando test werden allerschlimmste Folgen verhindert, falls man make u oder make uu aufruft, ohne der Variablen c einen Wert zugewiesen zu haben.

```
% cat makefile
SHELL=/bin/sh
NR=nroff -rL72 -rW69 -mm ch$(c).mm 2> ch$(c).key

N2=ch21.mm ch22.mm ch23.mm ch24.mm
N3=ch31.mm ch32.mm ch33.mm
N4=ch41.mm ch42.mm ch43.mm ch44.mm ch45.mm
N5=ch51.mm ch52.mm

TT=\
        T2='>ch21.mm;>ch22.mm;>ch23.mm;>ch24.mm;' \
        T3='>ch31.mm;>ch32.mm;>ch33.mm;' \
        T4='>ch41.mm;>ch42.mm;>ch43.mm;>ch44.mm;>ch45.mm;' \
        T5='>ch51.mm;>ch52.mm;'

.SUFFIXES: .mm .mu
.mu.mm:
        upr < $< > $@

help:

u:      # Formatiere Kapitel ch$(c) (ohne Umlautpostfilter)
        @test "$c"
        @find ch[1-9]*.mm -name 'ch$(c)*.mm' -size 0c \
        -exec touch ch$(c)*.mu \; 2> /dev/null
        @$(TT) \
        ; $(MAKE) -$(MAKEFLAGS) T=$${T$(c)} ch$(c)u
        # Bemerkung zu find:
        # Wenn eine Datei *.mm geleert wurde,
        # aktualisiere alle *.mu-Dateien.
        # Diagnosen von find werden ignoriert.
```

```
uu:     # Formatiere Endfassung ASCII-ext ch$(c)
        @test "$c"
        @find ch$(c)*.mu chhead.mm -newer ch$(c)u \
        -exec touch ch$(c)*.mu \; 2> /dev/null ||touch ch$(c)*.mu
        @$(TT) \
        ; $(MAKE) -$(MAKEFLAGS) T=$${T$(c)} ch$(c)uu
        # Bemerkung zu find:
        # Wenn eine Quelldatei neuer ist als ch$(c)u
        # oder wenn es ch$(c) gar nicht gibt,
        # aktualisiere alle Dateien ch$(c)*.mu.
        # Diagnosen von find werden ignoriert.

p:      # pg ch$(c)uu
p:    ; @make -s uu > /dev/null
        # Die Nachricht "`...' is up to date." soll verschwinden.
        @pg -n ch$(c)uu

t:      # touch ch$(c)*.mu
        touch ch$(c)*.mu

r:      # rm -f ch$(c)*.mm  # (ev. leere Dateien)
        rm -f ch$(c)*.mm

ch2u: $(N2)
ch3u: $(N3)
ch4u: $(N4)
ch5u: $(N5)

ch$(c)u: ch$(c).mm chhead.mm
        $(NR) > $@
        $T >ch$(c).mm
        touch $@

ch$(c)uu: ch$(c)u
        upo < ch$(c)u > $@

help: ; @echo '         # c=$c'
        @egrep '^[^:;=.]*::?[ <tab>]*#' [mM]akefile
%
```

Zum Abschluß der kleinen Leistungsschau des Technologieprogramms make folgt ein Terminalskript, in dem verschiedene Aufrufmöglichkeiten der eben vorgestellten Lösung mit ihren Protokollen gezeigt werden. Weitere Beispiele für die Arbeit mit make, die einer eingehenden Betrachtung wert sind, befinden sich im Abschnitt 6. und im Anhang 6.

```
1% setenv c 5
2% make
        # c=5
u:      # Formatiere Kapitel ch$(c) (ohne Umlautpostfilter)
uu:     # Formatiere Endfassung ASCII-ext ch$(c)
p:      # pg ch$(c)uu
t:      # touch ch$(c)*.mu
r:      # rm -f ch$(c)*.mm  # (ev. leere Dateien)
3% make u
        upr < ch51.mu > ch51.mm
        upr < ch52.mu > ch52.mm
        upr < ch5.mu > ch5.mm
        nroff -rL72 -rW69 -mm ch5.mm 2> ch5.key > ch5u
        >ch51.mm;>ch52.mm; >ch5.mm
        touch ch5u
```

```
4% make uu
        upo < ch5u > ch5uu
5% touch ch52.mu
6% make uu
        upr < ch51.mu > ch51.mm
        upr < ch52.mu > ch52.mm
        upr < ch5.mu > ch5.mm
        nroff -rL72 -rW69 -mm ch5.mm 2> ch5.key > ch5u
        >ch51.mm;>ch52.mm; >ch5.mm
        touch ch5u
        upo < ch5u > ch5uu
7% make r
        rm -f ch5*.mm
8% make u
        upr < ch51.mu > ch51.mm
        upr < ch52.mu > ch52.mm
        upr < ch5.mu > ch5.mm
        nroff -rL72 -rW69 -mm ch5.mm 2> ch5.key > ch5u
        >ch51.mm;>ch52.mm; >ch5.mm
        touch ch5u
9% make -n u
        find ch[1-7]*.mm -name 'ch5*.mm' -size 0c \
        -exec touch ch5*.mu \; 2> /dev/null
        T2='>ch21.mm;>ch22.mm;>ch23.mm;>ch24.mm;' \
        T3='>ch31.mm;>ch32.mm;>ch33.mm;' \
        T4='>ch41.mm;>ch42.mm;>ch43.mm;>ch44.mm;>ch45.mm;' \
        T5='>ch51.mm;>ch52.mm;' \
        ; make -bn T=${T5} ch5u
 ch5u' is up to date.
10% make -n uu
        @find ch5*.mu chhead.mm -newer ch5u \
        -exec touch ch5*.mu \; 2> /dev/null || touch ch5*.mu
        T2='>ch21.mm;>ch22.mm;>ch23.mm;>ch24.mm;' \
        T3='>ch31.mm;>ch32.mm;>ch33.mm;' \
        T4='>ch41.mm;>ch42.mm;>ch43.mm;>ch44.mm;>ch45.mm;' \
        T5='>ch51.mm;>ch52.mm;' \
        ; make -bn T=${T5} ch5uu
        upo < ch5u > ch5uu
11% make p -n
        make -s uu > /dev/null
        pg -n ch5uu
12% make uu
        upo < ch5u > ch5uu
13%
```

# 6. Die Programmgeneratoren yacc und lex

Der Name yacc steht für »yet another compiler compiler« (Johnson 1978) yacc ist eine von den ältesten UNIX-Komponenten und ist in einer Zeit entstanden, in der Compiler-Generatoren im aktuellen Blickfeld der Informatikforschung standen. Heutzutage gibt es weitere Compiler-Compiler, die jedoch nicht oder noch nicht den Weg in den Standardbestand des UNIX gefunden haben, siehe Rechenberg und Mössenböck (1985). Das Werkzeugprogramm lex von Lesk und Schmidt (1978) erzeugt Programme für die lexikalische Analyse, wie man sie für Compiler benötigt. Beide Werkzeugprogramme sind aufeinander abgestimmt. Sie sollen daher gemeinsam vorgestellt werden.

Nun werden die Anwender von UNIX kaum täglich vor der Aufgabe stehen, Compiler anfertigen zu müssen. Das könnte zu dem eiligen Vorurteil führen, daß diese Werkzeuge so wichtig wohl nicht sein können. Um dem zu begegnen, sollen im Abschnitt 6.1. zunächst einige Ausführungen über die Methode des syntaxorientierten Programmierens angeboten werden. Dabei werden yacc und lex schon benutzt. Vielleicht besitzt der Zugang zu diesen Werkzeugen über die Beschäftigung mit einzelnen Beispielen überhaupt die größte Überzeugungskraft.

## 6.1. Syntaxorientiertes Programmieren

In der anfänglichen Begegnung mit UNIX (Abschnitt 1.) war das schlichte Programmm accu geeignet, erste Schritte im Umgang mit diesem Betriebssystem zu demonstrieren. Das kleine Beispiel ist keineswegs zu trivial, um nicht auch zur Einführung in das syntaxorientierte Programmieren gute Dienste zu leisten.

Eine Eingabedatei beschrieb ein zweistufig gegliedertes Datenmaterial. Die Eingabedaten bilden eine Folge von Schlüsselwörtern. Schlüssel 1 charakterisiert den Beginn einer Gruppe. Schlüssel 0 verweist auf ein Objekt in der Gruppe. Ihm folgt ein numerisches Attribut. Ziel der Verarbeitung ist ein Bericht, der die numerischen Werte gruppenweise akkumuliert.

Die folgenden Ableitungsregeln beschreiben diesen Sachverhalt syntaktisch. Sie schlagen eine Brücke zur Welt der formalen Grammatiken. Die ersten beiden Regeln drücken aus, daß der Bericht aus einem Titel und dem eigentlichen Inhalt besteht. Der Titeltext bleibt zunächst offen. Der Berichtsinhalt (reportrest) ist die Beschreibung einer Gruppe oder ein schon als reportrest akzeptierter Text, auf den noch eine weitere Gruppenbeschreibung folgt. Eine Gruppe wird durch das explizit notierte Schlüsselwort "1" eingeleitet. Darauf folgt der eigentliche Teil als grouprest. Dieser ist ein Detail oder eine Folge von Details. Details werden explizit durch "0" angezeigt.

```
report: title reportrest
title: .
reportrest: group | reportrest group
group: key grouprest
key: "1" .
grouprest: detail   grouprest detail
detail: "0"
```

Die Eingabedaten für das Programm accu, wie sie im Abschnitt 1. durch die Datei accu.inp gewählt wurden, gehören zu der Schlüsselfolge:

```
1 0 0 0 0 1 0 0 1 0 0 <eof>
```

Die numerischen Attribute der Objekte haben keine syntaktische Relevanz. Sie wurden weggelassen. In der Terminologie der formalen Grammatiken bildet die Gesamtheit der von accu verwertbaren Eingabedateien eine Sprache. Die obigen Ableitungsregeln sind Teil der Beschreibung einer Grammatik für diese Sprache. Per definitionem umfaßt eine Grammatikbeschreibung vier Teile:

```
G = ( N, T, s, R )
```

Hierin bedeuten:

```
T terminale Symbole, im Beispiel T={0,1},
N Hilfssymbole, Nichtterminale, im Beispiel NT={report,title,
  reportrest,group,key,grouprest,detail},
s Satzsymbol, ein ausgezeichnetes Nichtterminal, s=report,
R Menge der Regeln.
```

Regeln sind Paare (l,r) mit der linken Seite l aus der Menge der Nichtterminale und der rechten Seite r, einer Folge von Elementen aus dem Vokabular der Grammatik. Als Vokabular V wird gemeinhin die Vereinigung der beiden Mengen N und T verstanden. Die Regelmenge R wurde in dem Beispiel in der EBNF-Notation angegeben, wie sie aus den Beschreibungen von Programmiersprachen geläufig ist.

Fast ungeändert geben wir die Ableitungsregeln als Datei a.y ein und bieten sie dem Programmgenerator yacc an. Er zeigt mit Exitwert 0 an, daß er die Datei als yacc-Quelltext akzeptiert hat. Die Datei y.tab.c entsteht im Ergebnis seiner Arbeit.

```
1% mkdir Yaccu
2% cd Yaccu
3% cat >a.y <<ENDE
%%
report: title reportrest
title:
reportrest: group
          | reportrest group
group: key grouprest
key: '1'
grouprest: detail
         | grouprest detail
detail: '0'
ENDE
4% yacc a.y
5% lc
a.y    y.tab.c
6%
```

Übrigens gab es in diesem Buch bereits weitere Berührungen mit yacc. Im Abschnitt 2.2.1. wurden Pfadausdrücke der Dateiverwaltung durch Syntaxregeln beschrieben. Die Datei path.y

```
% cat path.y
%token filename dirname
%%
pathname: filename
        | path_prefix filename
  ... usw. ...
rtprefix : dirname '/'
         | rtprefix dirname '/'
```

ist als Quelltext für yacc geeignet. Ebenso ist Anhang 3, der die Syntax von Kommandos der Bourne-Shell wiedergibt, ein korrekt notierter yacc-Quelltext.

yacc erzeugt Programme, die Eingangstexte nach dem LALR(1)-Verfahren, siehe Loeper u. a. (1987), analysieren. Der Eingangstext wird Symbol für Symbol sequentiell durchmustert. Dazu gleitet ein Fenster über den Text. Das darin sichtbare Symbol heißt Arbeitssymbol. Es steuert die Analyse. Der Analyseprozeß verläuft in einzelnen Schritten, denen Zustände des Analyseautomaten entsprechen. Zustände werden numeriert und durch ihre Nummer gekennzeichnet. In jedem Zustand wählt das Arbeitssymbol eine von vier Aktionen aus:

shift s
: Die Nummer des bestehenden Zustandes wird in den Analysekeller gegeben. Der Parameter s kennzeichnet den nächsten Zustand. Das folgende Eingangssymbol wird als Arbeitssymbol sichtbar.

reduce r
: Hier wird nach der Regel mit der Nummer r reduziert. Das heißt, es werden so viele Elemente aus dem Analysekeller entfernt, wie durch die Länge der rechten Seite dieser Regel abgezählt werden. Die linke Seite der Regel r, also ein Nichtterminal, wird Arbeitssymbol. Dieses bestimmt zusammen mit der Zustandsnummer im Analysekeller den neuen Zustand.

error
: Fehleraktion. Der bestehende Zustand und das Arbeitssymbol sind miteinander unverträglich. Die Analyse kann erst nach einer Fehlerbehandlung fortgesetzt werden. Schlimmstenfalls muß sie abgebrochen werden.

accept
: Der Eingangstext wurde als Satz der Sprache erkannt, die Analyse ist beendet.

Durch mehrere shift-Aktionen sei folgende Belegung der oberen Plätze im Analysekeller entstanden. Inhalt der Kellerplätze sind Zustandsnummern. Ihnen entsprechen Symbole des Vokabulars der Grammatik.

```
    s0  s1  s2        sn      as  restl. Eingangstext.
        r1  r2        rn
```

Arbeitssymbol as und Analysezustand sn führen zu einer Reduktion mit der Regel (l,r1 r2 ... rn). Dann schrumpft der Keller. Was sich im Keller als Abbild der rechten Seite der Regel wiederfindet, wird durch die linke Seite ersetzt. In der Folge der reduce-Aktion entsteht der neue Zustand s'. Nach der Reduktion bleibt folgende Kellerbelegung übrig:

```
          l
    s0  s1'                 as  restl. Eingangstext.
```

Zustand und Arbeitssymbol sind Indizes der steuernden LR-Tabelle. Es gibt viele Möglichkeiten, diese Tabelle zu implementieren. Sie einfach als Matrix anzulegen, in der die Aktionen als Elemente erscheinen, stößt schnell an die Grenzen einer sinnvollen Speicherverwaltung. yacc verwirklicht daher ein ausgeklügeltes System von Zeigern auf ein kompakt gespeichertes lineares Feld, indem beispielsweise Matrixzeilen mit identischen Aktionen oder die in ihrer Anzahl dominierenden Fehlereinträge zusammengedrängt werden.

Außerdem werden Reduktionsaktionen und anschließende Zustandsübergänge zu Einzelaktionen zusammengefaßt.

Wesentlich für das Verständnis von yacc ist die Verbindung von Reduktionsaktionen mit sogenannten semantischen Aktionen. Jeder Regel kann eine C-Anweisung als semantische Aktion beigegeben werden. Wird beim Ablauf der Analyse eine Aktion reduce r gestartet, so kann die Regelnummer r Anweisungen aus einer C-switch-Konstruktion auswählen. Diese Anweisungen können beliebige, zunächst global deklarierte Daten verknüpfen.

Alle Elemente des Zyklus, der für die tabellengesteuerte Arbeit anhand der Analysetabelle erzeugt wurde, befinden sich in der Funktion yyparse(). Sie ist neben der Analysetabelle selbst Bestandteil der Datei y.tab.c, die als Endprodukt eines erfolgreichen yacc-Laufes vorliegt.

Zudem muß man berücksichtigen, daß vor dem Eingang in die syntaktische Analyse erst einmal terminale Symbole konstruiert werden müssen. In Compilern ist dies die Aufgabe der lexikalischen Analyse. Sie stellt aus geeigneten Zeichenketten der Eingabetextdatei Symbole der syntaktischen Analyse her, aus Ziffernfolgen entstehen Zahlen, Wortsymbole werden erkannt, oder Zeichenfolgen werden als Operatoren weitergegeben. Wenn yacc ein neues Arbeitssymbol benötigt, um das Fenster darauf zu schieben, ruft es die Funktion yylex() auf. Dies ist dann auch gleich die ideale Gelegenheit, Daten für die Operanden in semantischen Aktionen bereitzustellen, damit sie bei Reduktionen wirksam werden können.

Nun zeigen wir die Fassung accu.y. Sie verwendet die eben erörterten Möglichkeiten von yacc. Die Funktion yylex() wurde als C-Funktion programmiert und hinten angehängt. Sie liefert die steuernden Schlüsselwörter '1' und '0'. Im Falle von Details ('0') erhält die Variable val einen numerischen Wert entsprechend der bekannten Aufgabenstellung. Das Skript zeigt die Datei accu.y und ihre Verarbeitung bis hin zum Test mit den Eingabedaten aus Abschnitt 2.2.1.

```
11% cat accu.y
%{
# include <stdio.h>
int totsum,sum,val;
%}
%%
report: title reportrest
          {printf("Gesamtsumme%11d\n",totsum);
           printf("======================\n");}
title: {printf("        Bericht\n");}
reportrest: group  {totsum=sum;}
          | reportrest group  {totsum+=sum;}
group: key grouprest
       {printf("Gruppensumme%10d\n",sum);
        printf("            ==========\n");}
key: '1'
grouprest: detail  {sum=val;}
         | grouprest detail  {sum+=val;}
detail: '0'
           {printf("%22d\n",val);}
%%
yylex() {
int d;
scanf("%d",&d);
```

```
if (d==0) {
  scanf("%d",&val); return '0';
  }
if (d==1) return '1';
return EOF;
}
12% yacc accu.y
13% cc -o accu y.tab.c -ly
14% ln ~/priv/book/cont/T22/Paccu/accu.inp accu.inp
15% accu <accu.inp
        Bericht
                    10
                    20
                   usw.
Gruppensumme         52
            ==========
                     9
                    39
Gruppensumme         48
            ==========
Gesamtsumme         200
======================
```

Im Abschnitt 5. haben wir die guten Dienste des Technologieprogramms make schätzen gelernt. Wir richten uns eine Datei Makefile ein, die es erlaubt, die Arbeit von yacc und die anschließende C-Übersetzung in einen make-Aufruf zu komprimieren, zumindestens für solche Quelltexte, die alle benötigten Informationen in einer einzigen Datei versammeln.

```
21% cat Makefile
SHELL=/bin/sh
LDFLAGS=-s -ly -ll
.y:
    $(YACC) $(YFLAGS) $<
    $(CC) -o $@ $(CFLAGS) y.tab.c $(LDFLAGS)
    rm -f y.tab.c
22% make accu
    yacc  accu.y
    cc -o accu -O y.tab.c -s -ly -ll
y.tab.c
    rm -f y.tab.c
23% accu <accu.inp
        Bericht
        usw.
24%
```

Die semantischen Aktionen der yacc-Regeln brauchen sich nicht mit der Verarbeitung explizit deklarierter Datenobjekte zu begnügen. Auch die Symbole des Vokabulars können als Attribute Werte mit sich führen. yacc verwaltet zu diesem Zweck gleichlaufend mit dem Analysekeller einen Wertekeller, in dem die Attribute der Symbole gespeichert werden.

In einer Regel (li,r1 r2 ... rn) werden die zu den Symbolen li, r1, rn gehörenden Attribute mit $$, $1, $2, $n adressiert.

So kann man auf die Variablen totsum, sum und val ganz und gar verzichten und statt dessen die Symbole reportrest, group und '0' attributieren. Je nach der Plazierung in den Regeln werden sie als entsprechende $i-Variable verknüpft. Terminale Symbole werden durch

yylex() erst im Laufe der Analyse sichtbar. yacc deklariert für ihre Attribute die Variable yylval. Standardgemäß sind die Elemente des Wertekellers und auch die Variable yylval vom Typ int. Die Funktion yylex() muß dies berücksichtigen. So entsteht eine weitere Modifikation accus.y der anfänglichen kleinen Aufgabe.

```
31% cat accus.y
%{
# include <stdio.h>
%}
%%
report: title reportrest
          {printf("Gesamtsumme%11d\n",$2);
           printf("======================\n");}
title: {printf("        Bericht\n");}
reportrest: group  {$$=$1;}
          | reportrest group  {$$=$1+$2;}
group: '1' grouprest  {$$=$2;
       printf("Gruppensumme%10d\n",$2);
       printf("             ==========\n");}
grouprest: detail  {$$=$1;}
         | grouprest detail  {$$=$1+$2;}
detail: '0'
          {printf("%22d\n",$1);}
%%
yylex() {
int d;
scanf("%d",&d);
if (d==0) {
  scanf("%d",&yylval); return '0';
  }
if (d==1) return '1';
return EOF;
}
32% make -s accu
y.tab.c
33% accu <accu.inp >accu.res
34%
```

Mit der Verarbeitung von Attributen ordnet sich yacc in den Sichtkreis der attributierten Grammatiken ein, vgl. Rechenberg (1984). Attributierte Grammatiken eignen sich als Leithilfen für die Programmentwicklung. Eins der bekanntesten Demonstrationsprogramme ist das Telegrammbeispiel von Henderson. Hier wird es vorgeführt, um ein weiteres Mal eine Berührung mit yacc anzubieten.

> »Ein Strom von einlaufenden Telegrammen ist zu verarbeiten. Jedes Telegramm wird durch die Zeichenfolge 'ZZZZ' abgeschlossen. Der Telegrammstrom ist beendet, wenn ein leeres Telegramm gefolgt von 'ZZZZ' eintrifft. Die Wörter eines Telegramms sollen gezählt werden. Die überlangen Wörter mit mehr als 12 Zeichen sollen extra gezählt werden. Die ermittelten Anzahlen sind hinter jedem Telegramm auszudrucken. Überflüssige Leerzeichen und Zeilenwechsel sollen eliminiert werden. Die Telegramme werden gelesen und anschließend zeilenweise ausgegeben. Das längste zugelassene Wort hat 20 Zeichen, längere Wörter sollen zum Programmabbruch führen.«

Neben den ASCII-Literalen erlaubt yacc es auch, terminale Symbole zu bezeichnen und solche Bezeichnungen in Regeln zu verwenden. Durch %token-Deklarationen werden die drei Namen LETT, BL und ZZZZ als

Bezeichner für terminale Symbole eingeführt. Entsprechend bedeuten sie einen Buchstaben, eine Folge von Leerzeichen, <tab>-Zeichen oder <newline>- Zeichen und das Schlußsymbol.

```
11% cat tel.y
/* yacc-Quelltext für das Telegrammbeispiel
        ohne semantische Aktionen */
%token LETT BL ZZZZ
%%
telstr  : leer
        | tele leer
tele    : tel
        | tele tel
tel     : text endw
text    : textw
        | text textw
textw   : letters BL
letters : LETT
        | letters LETT
leer    : endw
endw    : ZZZZ
%%
#include "lex.yy.c"
12%
```

Des weiteren haben wir keine explizite Funktion yylex() programmiert. An ihre Stelle tritt der Einschluß #include "lex.yy.c". Die Datei lex.yy.c entsteht in der Arbeit des Programmgenerators lex, den wir hiermit erstmals vorführen. Seine Aufgabe und Leistung ist es gerade, die für yacc notwendige Funktion yylex() zu erzeugen. Er benötigt als Eingabetext eine Beschreibung der Intentionen des Anwenders, wie er die lexikalische Analyse ablaufen lassen will. Genauer heißt das, es müssen Zeichenreihen des Quelltextes definiert werden, die jeweils als ein terminales Symbol der yacc-Analyse wirken sollen.

lex-Quellen bestehen aus Deklarationen und Regeln. Routinen können folgen. Die Datei tel.l beschreibt die lexikalische Analyse für das Telegrammbeispiel. Das Markierungszeichen %% trennt Deklarationen und Regeln voneinander. Eine Regel besteht aus einem Ausdruck, einer Verallgemeinerung der regulären Ausdrücke, wie sie beim Editor ed im Abschnitt 3.1.4. beschrieben wurden, und einer Aktion, einer C-Anweisung, die im wesentlichen die Aufgabe erfüllen muß, ein Terminalsymbol mit einem Attribut an yacc zu liefern. Im Deklarationsteil können Bezeichnungen für Ausdrücke eingeführt werden. Hier bt für Leer-/<tab>-Zeichenfolgen und btn für ein oder mehrere Zeilenwechsel, umgeben von Leerzeichen oder <tab>-Zeichen. Die Deklaration %start txtw kennzeichnet txtw als Startbedingung.

```
12% cat tel.l
  /* lex-Quelltext für das Telegrammbeispiel */
bt      [ \t]+
btn     ([ \t]*\n[ \t]*)+
%start txtw
%%
[,.A-Za-z]      {BEGIN txtw; yylval = *yytext; return LETT;}
<txtw>{btn}     {BEGIN 0; yylval = '\n'; return BL;}
<txtw>{bt}      {BEGIN 0; yylval = ' ';  return BL;}
```

```
ZZZZ            {return ZZZZ;}
\n              |
                ;
13%
```

Folgende Regeln wurden formuliert: Buchstaben und Interpunktionszeichen schalten die Bedingung txtw (Textwort) ein. Als terminales Symbol wird LETT zurückgegeben, als Attribut das Zeichen selbst. Generell zeigt die Variable yytext auf ein Feld, das gerade die durch den Ausdruck ermittelte Zeichenkette enthält. Hier ist die Zeichenkette genau ein Zeichen lang. Wenn auf ein Textwort eine Zeichenkette folgt, die den Ausdruck bt erfüllt, so wird das Terminal BL mit einem Leerzeichen als Attribut zurückgegeben. Wenn der Ausdruck btn erfüllt wird, entsteht ein <newline>-Zeichen als Attribut. Die Zeichenkette ZZZZ liefert das ebenso genannte Terminalsymbol. Der Ausdruck . bedeutet ein beliebiges Zeichen, das von <newline> verschieden ist, '\n' bedeutet ein <newline>-Zeichen. Beiden Ausdrücken entspricht die leere C-Anweisung ; als Aktion. Das Zeichen | besagt: Nimm die Aktion der nächsten Regel!

Der Aufruf

```
% lex tel.l
```

startet den Programmgenerator lex. Dabei entsteht die Datei lex.yy.c.

Wieder erinnern wir uns an das Technologieprogramm make. Das folgende Fragment einer Datei Makefile vereint die beiden bisher aufgetretenen Fälle: yacc-Queltext mit explizit programmierter Funktion yylex() und yacc-Text mit dem Einschluß lex.yy.c. Für den Fall, daß es die Datei tel.l gibt, wird zusätzlich noch lex gestartet. Das Kommando test wurde im Abschnitt 4.4.2. beschrieben. Sehr schnell stößt das lex-Produkt lex.yy.c an die Grenzen der C-Compiler-Version. Deshalb ist es ratsam, gleich von vornherein die Option -LARGE zu bemühen.

```
SHELL=/bin/sh
LDFLAGS=-ly -ll
.y:
        -test -r $*.l && $(LEX) $(LFLAGS) $*.l
        $(YACC) $(YFLAGS) $<
        $(CC) -o $@ $(CFLAGS) -LARGE y.tab.c $(LDFLAGS)
        rm -f y.tab.c lex.yy.c
```

Falls man in seinem Projekt ausschließlich yacc-lex-Aufgaben lösen muß, kann man mit einem einfacheren Makefile-Fragment auskommen.

```
SHELL=/bin/sh
LDFLAGS=-ly -ll
.DEFAULT:
        -test -r $<.l && $(LEX) $(LFLAGS) $<.l
        $(YACC) $(YFLAGS) $<.y
        $(CC) -o $@ $(CFLAGS) -LARGE y.tab.c $(LDFLAGS)
        rm -f y.tab.c lex.yy.c
```

Abschließend zeigen wir die mit semantischen Aktionen angereicherte Fassung des Telegrammbeispiels mit make-Protokoll und einem Strom von drei Telgrammen als Testdaten.

```
21% cat tels.y
%{
/* yacc-Quelltext für das Telegrammbeispiel
    mit semantischen Aktionen */
#include <stdio.h>
short l,m,n;
%}
%token LETT BL ZZZZ
%%
telstr    leer
          {puts("Telegrammstrom:");} tele leer
tele      tel
          tele tel
tel       {puts("Telegrammtext:");} text endw
          {printf("\nWortanzahl=%d, davon lange: %d\n",n,m);}
text      textw  {n=1; m=(l>12)? 1 : 0;}
          text textw  {n++; if (l>12) m++;}
textw     letters BL  {putchar(yylval);}
letters   LETT  {l=1; putchar(yylval);}
          letters LETT  {if (++l>20) YYABORT; putchar(yylval);}
leer      endw
endw      ZZZZ
%%
#include "lex.yy.c"
22% make tels
        test -r tels.l && lex tels.l
        yacc  tels.y
        cc -o tels -O -LARGE y.tab.c -ly -ll
y.tab.c
        rm -f y.tab.c lex.yy.c
23% tels <tel.inp
Telegrammstrom:
Telegrammtext:
Dies ist das erste Telegramm.
Wortanzahl=5, davon lange: 0
Telegrammtext:
Das zweite enthält ein langes Wort, und zwar abcdefghijklm.
Wortanzahl=9, davon lange: 1
Telegrammtext:
Ansonsten enthalten die Telegramme wenig Überraschungen.
Wortanzahl=6, davon lange: 1
24%
```

## 6.2. Generieren von Programmen

### 6.2.1. Die Vielfalt der beteiligten Objekte

Für die Einarbeitung in die Problematik der automatischen Generierung von Programmen ist es sehr wichtig, ständig die Übersicht über die beteiligten Objekte zu behalten. Schon bei kleinen Projekten wächst die Anzahl der Objekte recht schnell an. Im Sinne seines Namens erzeugt yacc Compiler für eine Sprache lang, deren Grammatik als yacc-Quelltext lang.y bereitgestellt wird. yacc erzeugt einen Parser, dessen Eingabesymbole durch eine abgestimmte lexikalische Analyse angeboten werden müssen (lang.l). Dieser Parser kann Sätze in der Sprache lang analysieren, beispielsweise den Satz exempl.lang. Wenn die semantischen Aktionen zudem noch Kode generieren, etwa den Zwischenkode exempl.z, so kann dieser anschließend in eine ladbare Form gebracht werden. Es entsteht das Programm exempl, das mit Eingaben exempl.inp Resultate exempl.res liefert. Dieses zeigt Bild 6.1.

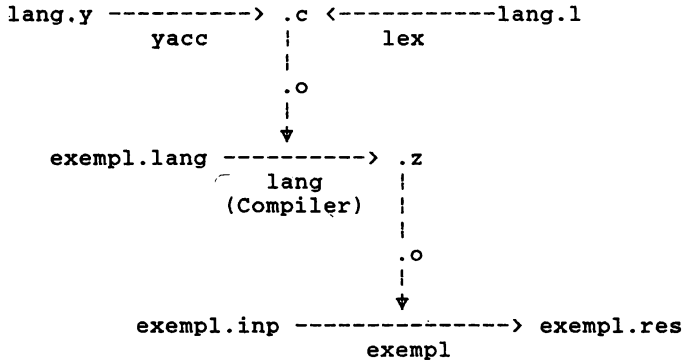



Bild 6.1. Compilergenerator

Eine Verkürzung der Situation ergibt sich, wenn Sätze der Sprache lang alle zu verarbeitenden Daten in sich bergen, vielleicht als Mischungen von Sprachelementen und Attributen oder einfach als generative Programme, in denen die Ausgangsgrößen von Berechnungen statisch als Konstanten integriert sind. In dieser Variation arbeitet yacc als Interpretergenerator (Bild 6.2). Die Beispiele des syntaxorientierten Programmierens gehören hierher, ebenfalls das im Abschn. 6.2.2. folgende Beispiel eines Programms zur syntaktisch gesicherten Interpretation von Ausdrücken einer kleinen Ringarithmetik.

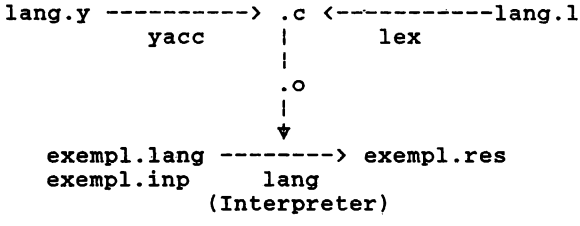



Bild 6.2. Interpretergenerator

In einer dritten Variante erzeugt yacc Syntaxprüfer. Es interessiert nur der exit-Wert von yyparse(). Ein Beispiel für diese Form der yacc-Arbeit wird im Abschnitt 6.5. mit dem Programm myaccg angeboten (Bild 6.3).

### 6.2.2. Ein klassisches Beispiel

Zur Gestaltung eines Interpreters für arithmetische Ausdrücke ist das Beispiel deskcalc.y bis auf geringfügige Modifikationen bereits in der Originalarbeit von Johnson (1978) zu finden. Der »Tischrechner« soll multiplizieren, addieren und subtrahieren und geklammerte Ausdrücke berechnen. Ein Minuszeichen vor einem Ausdruck kehrt sein Vorzeichen um. Der »Tischrechner« hat 26 Speicherregister, die mit den Buchstaben a bis z bezeichnet werden. Er verarbeitet Dezimalzahlen und Oktalzahlen. Oktalzahlen beginnen mit der Ziffer '0'.

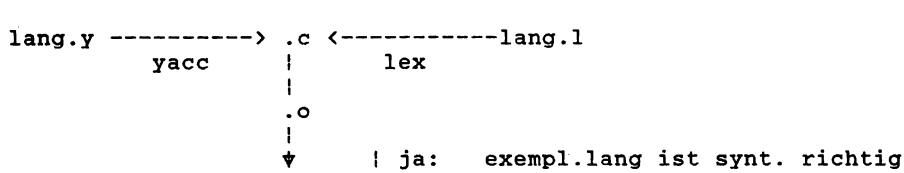



Bild 6.3. Generieren eines Syntaxprüfers

Die Operatoren '+' '-' '*' UMINUS sind linksassoziativ und haben in der Reihenfolge ihrer Deklaration wachsenden Vorrang. Ein Ausdruck ist höchstens eine Zeile lang. Syntaktisch falsche Ausdrücke sind zulässig. Nach dem fehlerstabilisierenden <newline>-Zeichen wird die Analyse und Interpretation fortgesetzt. Die lexikalische Analyse ist explizit als C-Funktion yylex() angehängt.

```
11% cat deskcalc.y
%{
# include <stdio.h>
# include <ctype.h>
int regs[26];
int base;
%}
%token DIGIT ':=' 270  LETTER
%left '+' '-'
%left '*'
%left UMINUS
%%
list :
     | list '\n'
     | list stat '\n'
     | list error '\n' {yyerrok;}
     ;
stat : expr  {printf("....%d\n",$1);}
     | LETTER ':=' expr  {regs[$1]=$3;}
     ;
expr : '(' expr ')'       {$$=$2;}
     | expr '+' expr      {$$=$1+$3;}
     | expr '-' expr      {$$=$1-$3;}
     | expr '*' expr      {$$=$1*$3;}
     | '-' expr %prec UMINUS  {$$=(-$2);}
     | LETTER  {$$=regs[$1];}
     | number
     ;
number : DIGIT  {$$=$1; base= ($1==0) ? 8   10;}
       | number DIGIT  {$$=base*$1+$2;}
       ;
%%
yylex(){
  int c,d;
  while ((c=getchar())==' ');
  if (islower(c)){
    yylval=c-'a'; return LETTER;
  }
  if (isdigit(c)){
    yylval=c-'0'; return DIGIT;
```

```
    }
    if (c==':')
      if ((d=getchar())=='=')
        return 270;
        else ungetc(d,stdin);
    return(c);
}
main(){
  exit(yyparse());
}
```

Als zweite Version folgen die Realisierung von yylex() durch den Programmgenerator lex und schließlich Aufrufe mit Testkommandos.

```
12% cat deskcalc.l
%%
[ \t]*          ;
[0-9]           {yylval=(*yytext - '0'); return DIGIT;}
[a-z]           {yylval=(*yytext - 'a'); return LETTER;}
":="            return 270;
\n              |
.               return *yytext;
13% make -s deskcalc
14% deskcalc
a := -3
(a + 012)*5
....35
Ctrl-d
15%
```

Die folgenden Programmzeilen zeigen eine Passage aus der Datei lex.yy.c, wie sie für die Datei deskcalc.l von lex generiert wird. Bemerkt werden muß, daß die eigentliche lexikalische Analyse, also die Suche nach Zeichenreihen, die Ausdrücke der lex-Regeln erfüllen, in der Funktion yylook() verborgen ist. Sie liefert die Nummer des Ausdrucks, der durch eine Zeichenreihe erfüllt wird. Die verbleibende Arbeit von yylex() ist die switch-Verzweigung anhand dieser Nummer. Wie der Ausschnitt aus lex.yy.c zeigt, werden die Aktionen der lex-Regeln ohne Änderung in die switch-Anweisung eingebaut. Der Text klärt mehr Details, insbesondere Einzelheiten der Dateiendebehandlung, als durch verbale Erläuterungen verdeutlicht werden könnten.

```
yylex(){
int nstr; extern int yyprevious;
    /* xxxxxxxxx */
while((nstr = yylook()) >= 0)
yyfussy: switch(nstr){
case 0:
if(yywrap()) return(0); break;
case 1:

break;
case 2:
         {yylval=(*yytext - '0'); return DIGIT;}
break;
case 3:
         {yylval=(*yytext - 'a'); return LETTER;}
break;
case 4:
          return 270;
```

```
    break;
    case 5:
                case 6:
                 return *yytext;
    break;
    case -1:
    break;
    default:
    fprintf(yyout,"bad switch yylook %d",nstr);
    } return(0); }
    /* end of yylex */
```

Eine Bemerkung verdient der Sonderfall nstr=0. Er tritt immer dann ein, wenn die Funktion yylook() das Dateiende erkennt. Ersichtlich wird der weitere Fortgang von yylex() durch die Funktion yywrap() beeinflußt. Sie ist Bestandteil der Bibliotheken /lib/[LMS]libl.a, die durch die Laderoption -ll erreicht werden. Die darin angebotene Standardversion lautet:

```
    yywrap() {return 1;}
```

Die Funktion kann aber auch Anwenderwünschen entsprechend programmiert werden, beispielsweise die Eröffnung einer folgenden Eingabedatei veranlassen, eine Statistik der lexikalischen Analyse der beendeten Datei berechnen oder anderes. Hervorgehoben sei, daß yacc Ergebnisse ≤0 von yylex() immer als das Ende der Eingabe auffaßt, demgemäß Fehler signalisiert, wenn der gelesene Text noch nicht akzeptiert wurde oder wenn während einer Fehlerbehandlung der Text zu Ende geht.

## 6.3. Programmieren mit yacc und lex

### 6.3.1. Ausdrucksmittel von yacc

yacc-Programme bestehen aus drei Teilen, die jeweils durch Marken %% voneinander getrennt werden. Der letzte Teil einschließlich der vorangehenden Marke kann wegbleiben.

```
    Deklarationsteil
    %%
    Regelteil
    %%
    Routinenteil
```

Deklarationsteil

`%{ C-Text %}`
: Der in Klammern eingeschlossene C-Text wird ungeändert in die Datei y.tab.c übernommen.

`%start name`
: Kennzeichnet das Nichtterminal als Satzsymbol der Grammatik. Fehlt diese Deklaration, so wird die linke Seite der ersten Regel als Satzsymbol genommen.

`%union { member-declaration-list }`
: Vereinbart den Typ der Elemente des Wertekellers. Sonst wird dieser Typ als int fixiert.

%token <comp> name numb
Bezeichner oder Zeichenketten name werden als terminale Symbole deklariert. Mehrere Namen können in einer Deklarationszeile erscheinen. Ihnen werden durch #define-Anweisungen Nummern ≥257 zugeordnet. Mit der optionalen Ziffernfolge numb kann eine andere Numerierung erzwungen werden. (Allerdings berücksichtigt yacc bei seiner internen Zählung die Nummern numb nicht. Bei einer gemischten Numerierung, wie sie in deskcalc.y gezeigt wurde, ist daher Sorgfalt geboten.) Der ebenfalls optionale Zusatz <comp> legt den Wertetyp des Terminals fest, indem er eine Variable aus der Liste in der %union-Deklaration auswählt.

%type <comp> name
Wählt die Komponente comp aus der %union-Deklaration. Ihr Typ bestimmt den Wertetyp des Nichtterminals name.

%left op
%right op
%nonassoc op
Legt Assoziativität und Priorität der Operatoren op fest. Mehrere Operatoren in einer Deklarationszeile haben gleiche Priorität. Ansonsten bestimmt die Reihenfolge der Deklarationszeilen wachsende Prioritäten der jeweiligen Operatorklassen.

Regelteil

Regeln (l, r11 r12 ) oder (l, r21 r22 ) werden in der Form

```
l   r11 r12
l   r21 r22
```

aufgeschrieben. Mehrere Regeln mit gleicher linker Seite können als Sequenz von Alternativen notiert werden.

```
l   r11 r12
    r21 r22
```

Am Ende von Regeln können semantische Aktionen angehängt werden. Das sind in Klammern { } eingefaßte Anweisungen der Sprache C. Gelegentlich ist es wünschenswert, zwischen zwei Symbolen auf der rechten Seite einer Regel eine weitere semantische Aktion einzuordnen. yacc führt diesen Fall auf den Normalfall zurück, indem es die Konstruktion

```
l       r {action} r'
```

durch die beiden Regeln

```
l : ... r q r' ...
q : /*leer*/ {action}
```

ersetzt. Die Regel q ist eine künstlich durch yacc generierte Regel. Mit dem Zusatz %prec id können Priorität oder Assoziativität einer Regel vorgeschrieben werden. Der Bezeichner id muß dazu an passender Stelle in der Prioritätenliste des Deklarationsteils vorkommen.

Routinenteil

Der Routinenteil enthält wichtige Routinen und Kommunikationsbezeichnungen.

```
#include <stdio.h>
yyerror(s)
char *s; {
  fprintf(stderr,"%s\n",s);
}
main() {
  return yyparse();
}
```

Die Objektkodes dieser beiden Routinen befinden sich in den Bibliotheken /lib/[LMS]liby.a, die durch die Laderoption -ly angesprochen werden. Freundlicher und oftmals ausreichend ist folgende Variante des Hauptprogramms (auch für XENIX386 geeignet):

```
main() {
short res;
  res = yyparse();
  printf("yyparse()=%d\n",res);
  exit(res);
}
```

Einige Kommunikationsbezeichnungen erscheinen einfach als Präprozessor-Definitionsanweisungen in der Datei y.tab.c.

```
# define YYERRCODE 256
# define YYERROR   goto yyerrlab
# define YYACCEPT  return(0)
# define YYABORT   return(1)
# ifndef YYSTYPE
# define YYSTYPE.  int
# endif
# define yyerrok   yyerrflag = 0
# define yyclearin yychar = -1
```

Für den Aufruf von yacc gibt es die drei Optionen -dtv.

-t In der Datei y.tab.c wird die Präprozessorbezeichnung YYDEBUG mit dem Wert 1 definiert, wenn sie nicht bereits vordefiniert war. Es werden Druckanweisungen für ein Spurverfolgungsprotokoll generiert, die im endgültigen Lademodul durch Wertzuweisung an die Variable yydebug≠0 wirksam geschaltet werden können.
-v Die Analysetabelle, die yacc erzeugt, wird in lesbarer Form in der Datei y.output abgelegt.
-d yacc erzeugt zusätzlich zu y.tab.c die Datei y.tab.h. Sie enthält Definitionsanweisungen für die Tokennamen. Wenn im Deklarationsteil eine %union-Vereinbarung angegeben war, kommt die daraus erzeugte Typdefinition auch in y.tab.h vor.

Zur Illustration folgen die Datei y.tab.h und der Anfang von y.output für das Beispiel deskcalc aus Abschnitt 6.2.2.

```
% yacc -dv deskcalc.y
% cat y.tab.h
# define DIGIT 257
# define LETTER 259
# define UMINUS 260
% cat y.output
state 0
    $accept   _list $end
    list : _    (1)
       reduce 1
```

```
        list  goto 1
    state 1
        $accept :  list_$end
        list :  list_
        list :  list_stat
        list :  list_error
        $end  accept
        error  shift 4          #####
        DIGIT  shift 10
        LETTER  shift 6
        -  shift 8
        \n  shift 2
        (  shift 7
        .  error
        stat  goto 3
        expr  goto 5
        number  goto 9
    state 2
            usw.
```

### 6.3.2. Ausdrucksmittel von lex

Auch lex-Programme bestehen aus drei Teilen. Wieder werden sie durch Marken %% voneinander abgetrennt. Der letzte Teil kann wegbleiben.

```
Deklarationen
%%
Regeln
%%
Routinen
```

Ein Teil der Zeilen in lex-Quelltexten bstimmt die Arbeit von lex, ein anderer Teil wird unverändert in die Resultatdatei lex.yy.c übergeben. Die eingeklammerten Zeilen

```
%{<newline>
text
%}<newline>
```

werden ohne Änderung übertragen. Ebenso verfährt lex mit

```
<blank>text
<tab>text
```

Stehen diese Zeilen vor dem ersten Markierungszeichen %%, so werden sie in der Resultatdatei auf globalem Niveau einsortiert. Folgen sie unmittelbar auf dieses Markierungszeichen, so kommen sie an das Ende des lokalen Deklaratonsteils von yylex(). (Diese Stelle wurde in dem zu deskcalc.l gehörenden C-Text mit xxxxxxxxx markiert, siehe Abschn. 6.3.2.) Wenn solche Zeilen an anderer Stelle gefunden werden, gelangen sie wie die Aktionen-C-Texte in die Resultatdatei, also in das Innere der switch-Anweisung zur Aktionenauswahl. Dies kann für Kommentarzeilen sinnvoll sein.

Die Zeilen des Routinenteils werden ebenfalls unverändert übertragen. Andere Zeilen werden als Deklarationen oder Regeln für lex interpretiert.

Deklarationsteil

Hier können Bezeichner erklärt werden, die später in regulären Ausdrücken des Regelteils abkürzend für womöglich mehrfach wiederverwendete andere reguläre Ausdrücke eintreten können. Die Deklarationen haben die Form

```
id          ausd
```

und beginnen mit dem ersten Zeichen der Zeile. Später vertritt {id} den definierten Ausdruck ausd. Mit der Zeile

```
%start name1 name2
```

werden die Namen name1, name2 usw. als Bezeichnungen für Startbedingungen deklariert.

Regelteil

Regeln bestehen aus einem Ausdruck und einer Aktion. Als Aktion muß eine C-Anweisung formuliert werden. Ein Leer- oder ein <tab>-Zeichen reicht aus, Ausdruck und Aktion voneinander zu trennen. Der Ausdruck beschreibt eine Menge von Zeichenketten. Er beginnt mit dem ersten Zeichen der Zeile und kann aus mehreren Teilen bestehen. Folgende Zeichen dienen als Verknüpfungsoperatoren oder lösen spezielle Funktionen bei der Interpretation von Ausdrücken aus:

```
" \ [ ] ^ - ?   * +   ( ) $ / { } % < >
```

Unmittelbar aufeinanderfolgende Ausdrücke beschreiben die Verkettung von Zeichenreihen, die beide Ausdrücke einzeln erfüllen. Die weiteren Verknüpfungen sind als Liste zusammengestellt:

```
x       Zeichen x ohne Sonderfunktion.
"x"     Zeichen x, auch wenn es ein Operatorzeichen ist.
\x      Zeichen x, auch wenn es ein Operatorzeichen ist.
[xy]    Ein Zeichen, das Zeichen x oder das Zeichen y.
[x-y]   Ein Zeichen zwischen x und y einschließlich.
[^x]    Ein beliebiges Zeichen, aber nicht x.
.       Ein beliebiges Zeichen, aber nicht <newline>.
^x      Zeichenkette x am Zeilenanfang.
<y>x    Zeichenkette x, wenn die Startbedingung y gilt.
x$      Zeichenkette x am Zeilenende.
x?      0- oder einmaliges Auftreten von x.
x*      0- oder mehrmaliges Auftreten von x.
x+      1- oder mehrmaliges Auftreten von x.
x|y     Zeichenkette x oder y.
(x)     Klammerung, Zeichenkette x.
x/y     Zeichenkette x, wenn darauf y folgt.
{id}    Zeichenkette entsprechend der Deklaration von id.
x{m,n}  Zeichenkette x, m- bis n-mal.
```

Es ist möglich, daß Zeichenketten des Eingabetextes mehrere Ausdrücke in den Regeln eines lex-Textes erfüllen. Dann wird ein Ausdruck ausgewählt, der es gestattet, eine möglichst lange Zeichenreihe zu absorbieren. Gibt es immer noch mehrere Ausdrücke dieser Art, so wird der erste entsprechend der Reihenfolge im lex-Text genommen.

Routinenteil

Auch hier enthält der Schlußteil wichtige Routinen und Kommunikationsbezeichnungen:

```
char *yytext;
```
Zeichenkette, die den ausgewählten Ausdruck erfüllt hat.
```
int yyleng;
```
Länge dieser Zeichenkette. Die Zeichenkette befindet sich in den Feldelementen:
```
    yytext[0], ..., yytext[yyleng-1]
    yytext[yyleng]=='\0'
int yylineno;
```
Zeilennummer des Eingabetextes, Nummer der Zeile, in der der Text endet.

ECHO
Aktion zur Reproduktion der abgebildeten Zeichenkette auf stdout.

BEGIN
Einschalten von Startbedingungen. BEGIN name schaltet die Bedingung name ein und vorher gültige aus. BEGIN 0 schaltet eine gerade gültige Startbedingung ab.

```
yymore()
```
Der folgende Eingabetext wird noch einmal mit dem eben erfüllten Ausdruck konfrontiert. Gegebenenfalls verlängert sich yytext dabei.
```
yyless(n) int n;
```
Die Analyse des Eingabetextes wird auf yytext[n] zurückgesetzt, (0≤n<yyleng).

REJECT
Aktion, die den im Regelteil folgenden Ausdruck erneut mit dem Eingabetext konfrontiert. Sie wird durch die Bibliotheksroutine reject() realisiert.
```
yywrap() {return 1;}
main() {return yylex();}
```
Bibliotheksversionen von yywrap() und main(), siehe Abschnitt 6.2.2.
```
input(), yyinput()                - Makro, Funktion -
```
Liefert das nächste Zeichen des Eingabtextes.
```
output(c), yyoutput(c) char c;    - Makro, Funktion -
```
Gibt das Zeichen c nach stdout.
```
unput(c), yyunput(c) char c;      - Makro, Funktion -
```
Gibt das Zeichen c in den Eingabetext zurück.

Für das Programm lex gibt es die beiden Aufrufoptionen -tv. Der Aufruf

```
% lex -t src.l
```

schreibt die sonst entstehende Datei lex.yy.c unmittelbar auf stdout. Mit

```
% lex -v src.l
```

werden Statistikdaten mitgeliefert. Wenn keine Quelldatei src.l angegeben wurde, erwartet lex den Quelltext von stdin. lex arbeitet dann als Filter.

## 6.4. Konflikte, Fehlerbehandlung und Spurverfolgung

### 6.4.1. Grammatikkonflikte

Wenn yacc für eine gegebene Grammatik die Analysetabelle berechnen soll, können zwei Konfliktsituationen auftreten: Bei der Berechnung einer reduce-Aktion stellt yacc fest, daß es für das gleiche Tupel (Zustand, Arbeitssymbol) bereits vorher eine Aktion berechnet hat. Lag eine shift-Aktion vor, so handelt es sich um einen shift/reduce-Konflikt. Befindet sich eine vorher berechnete reduce-Aktion auf dem Platz in der Analysetabelle, so ist es ein reduce/reduce-Konflikt. Beide Konfliktsituationen belegen wir mit Beispielen.

Wir beschäftigen uns zuerst mit shift/reduce-Konflikten. yacc löst diese Konflikte, indem es die shift-Aktion favorisiert. Nur sie findet in der Analysetabelle Platz. Ein geradezu klassisches Beispiel eines solchen Konflikts geht aus der verkürzten Fassung der if-then-else-Konstruktion hervor. Es führt uns also an den Anfang der Ära höherer Programmiersprachen. Bekannt geworden ist dieses Sprachkonstrukt schon durch ALGOL 60. Die gleiche Geschichte findet man auch in PASCAL und in C. Wir zeigen diesen Konflikt in der Sprache C und rüsten dazu den Sachverhalt auf das Wesentliche ab, indem wir sonst weiterführende syntaktische Strukturen kurzerhand in den Rang von terminalen Symbolen erheben.

```
11% cat a1.y
%token If Else cond simple
%%
stat : simple
   | If cond stat
   | If cond stat Else stat
%%
#include "lex.yy.c"
main(argc,argv)
int argc; char **argv; {
extern int yydebug;
if (argc>1) yydebug=atoi(argv[1]);
printf("yyparse()=%d\n",yyparse());
}
12% yacc -v a1.y
conflicts: 1 shift/reduce
13%
```

Der Routinenteil zeigt zusätzlichen Luxus anderer Art, der für das weitere ganz nützlich ist. Wir zeigen die Analysetabelle als Datei y.output. Dort bekommen wir den Zustand 5 vorgeführt, der die Konfliktsituation in sich birgt.

```
13% cat y.output   sed '/^$/d'   sed '/terminals/,$d'
state 0
    $accept : _stat $end
    If  shift 3
    simple  shift 2
    .  error
    stat  goto 1
state 1
    $accept :  stat_$end
    $end  accept
    .  error
state 2
    stat    simple_    (1)
```

```
      .  reduce 1
state 3
    stat :  If_cond stat
    stat :  If_cond stat Else stat
    cond  shift 4
    .  error
state 4
    stat :  If cond_stat
    stat :  If cond_stat Else stat
    If  shift 3
    simple  shift 2
    .  error
    stat  goto 5
5: shift/reduce conflict (shift 6, red'n 2) on Else
state 5
    stat :  If cond stat_    (2)
    stat :  If cond stat_Else stat
    Else  shift 6
    .  reduce 2
state 6
    stat :  If cond stat Else_stat
    If  shift 3
    simple  shift 2
    .  error
    stat  goto 7
state 7
    stat :  If cond stat Else stat_    (3)
       reduce 3
14%
```

Wir demonstrieren die Verarbeitung dieses Satzes durch yyparse().

```
If cond If cond simple Else simple
```

Natürlich benötigen wir dazu eine lexikalische Analyse. Wir haben die Datei a1.l präpariert.

```
14% cat a1.l
%{
#include "y.tab.h"
%}
id          [A-Za-z][0-9A-Z_a-z]*
%%
[ \n\t]*      ;
{id}        {if (strcmp(yytext,"If")     ==0) return If;
             if (strcmp(yytext,"Else")   ==0) return Else;
             if (strcmp(yytext,"cond")   ==0) return cond;
             return simple;}
.           return yytext[0];
15%
```

Zur Erleichterung im Umgang mit dem Listing y.output sei bemerkt, daß eine Reduktionsaktion so viele Kellerplätze räumt, wie die Regel Symbole in ihrer rechten Seiten besitzt. Der dadurch aufgedeckte Zustand enthält eine goto-Tabelle. Mit der linken Seite der Reduktionsregel liefert der goto-Eintrag die nächste Zustandsnummer. Sie wird neues oberes Kellerelement.

Nun verfolgen wird die Spur des Analyseprozesses, schon auf die Mittel aus dem Abschnitt 6.4.3. vorgreifend. Rechts wird die Belegung des Analysekellers mit Zustandsnummern gezeigt.

```
15% make -s a1 YYFLAGS=-dt
conflicts: 1 shift/reduce
16% a1 1 <<ENDE
If cond If cond alpha Else beta
ENDE
  ...                                  Stackbelegung:
State 0                                ¦ 0
Received token If                      ¦
State 3                                ¦ 0 3
Stack: If                              ¦
Received token cond                    ¦
State 4                                ¦ 0 3 4
Stack: If cond                         ¦
Received token If                      ¦
State 3                                ¦ 0 3 4 3
Stack: If cond If                      ¦
Received token cond                    ¦
State 4                                ¦ 0 3 4 3 4
Stack: If cond If cond                 ¦
Received token simple                  ¦
State 2                                ¦ 0 3 4 3 4 2
Stack: If cond If cond simple          ¦ 0 3 4 3 4
Reduce by (1) "stat : simple"          ¦ goto 5
State 5                                ¦ 0 3 4 3 4 5
Stack: If cond If cond stat             ¦
Received token Else                    ¦
State 6                                ¦ 0 3 4 3 4 5 6
Stack: If cond If cond stat Else       ¦
Received token simple                  ¦
State 2                                ¦ 0 3 4 3 4 5 6 2
Stack: If cond If cond stat Else simple ¦ 0 3 4 3 4 5 6
Reduce by (1) "stat : simple"          ¦ goto 7
State 7                                ¦ 0 3 4 3 4 5 6 7
Stack: If cond If cond stat Else stat    0 3 4
Reduce by (3) "stat : If cond stat Else stat"  goto 5
State 5                                  0 3 4 5
Stack: If cond stat
Received token [end of file]             0
Reduce by (2) "stat : If cond stat"      goto 1
State 1, token [end of file]             0 1
Stack: stat                              0 --> accept
yyparse()=0
17%
```

Aus allem geht hervor, daß die Lösung shift-vor-reduce, die yacc standardmäßig auswählt, den üblichen Verlautbarungen verbaler Semantik bei ALGOL, PASCAL oder C entspricht. Fragen wir nach dem Grund für die Konfliktsituation, so hilft eine Rückbesinnung auf das entsprechende Syntaxdiagramm im Bild 6.4.

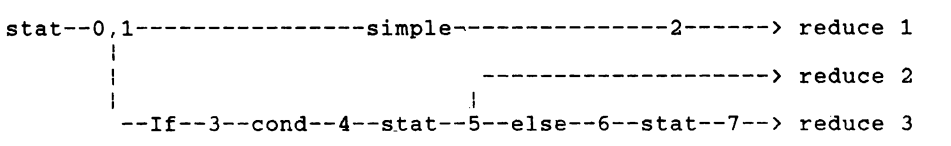



Bild 6.4. if-then-else-Anweisung

Der Konflikt wird durch den ungeschützten Ausgang im Zustand 5 verursacht. Stellen wir einen Wächter an diesen Ausgang, indem wir eigens dafür ein terminales Symbol sep erfinden, so verschwindet der Konflikt.

```
17% cat a2.y
%token If Else cond simple sep
%%
sent : stat
stat : if cond stat sep
stat : if cond stat else stat
stat : simple
18% yacc a2.y
19%
```

In dieser Weise wurde das alte if-then-else-Problem bei Modula-2 aus der Welt geschafft. Dort wird allerdings auch am else-Zweig ein END angebracht, was für die Konfliktbewältigung schon gar nicht nötig wäre. Anwärter für Konflikte sind Regeln mit leeren rechten Seiten oder Regeln, in denen die rechten Seiten Anfangsstücke anderer Regeln sind. Nicht zwangsläufig führen diese Konstruktionen zu Konflikten. Nicht jeder offene Ausgang muß in den Regeln selbst abgedichtet werden. Konflikte sind aber nicht zu umgehen, wenn in der umhüllenden Regelmenge Ausgänge offen bleiben. Theoretisch scheitert der übliche Algorithmus zur Berechnung der Analysetabelle, sobald Konflikte auftreten. Um dem zu begegnen, schließt yacc »sämtliche« Ausgänge vorsichtshalber selbst, indem es allen Regeln voran die künstliche Regel

```
$accept: startsymbol $end
```

generiert, wie dies in der Datei y.output sichtbar ist.

Wenden wir uns nun den reduce/reduce-Konflikten zu. In diesem Fall reduziert yacc standardmäßig nach der Regel mit der kleineren Nummer. Wieder finden wir ein Beispiel in der Welt der klassischen Programmiersprachen. In Modula-2 gibt es die EBNF-Regeln:

```
qualident  = ident {"." ident}.
designator = qualident {"." ident | "[" exprlist "]"   ""}.
```

Kleiden wir sie in das Gewand von yacc, so könnten wir diesen yacc-Text erhalten.

```
21% cat r1.y
%token ident exprlist
%start designator
%%
qualident  : ident
           | qualident '.' ident
designator : qualident
           | qualident designator_tail
designator_tail : designator_elem
                | designator_tail designator_elem
designator_elem : '.' ident
                | '[' exprlist ']'
                | '^'
22% yacc r1.y
conflicts: 4 reduce/reduce
23%
```

In der Datei y.output finden wir den Zustand 9 mit folgenden Hinweisen:

```
9: reduce/reduce conflict (red'ns 2 and 7 ) on $end
9: reduce/reduce conflict (red'ns 2 and 7 ) on .
9: reduce/reduce conflict (red'ns 2 and 7 ) on [
9: reduce/reduce conflict (red'ns 2 and 7 ) on ^
state 9
        qualident :  qualident . ident_     (2)
       designator_elem :    ident_     (7)
          reduce 2
```

Wir verdeutlichen uns die Verhältnisse, indem wir ein bißchen Modula-2-Kontext besichtigen. Folgende Zeichenreihen sind zweifelsohne Sätze unserer designator-Sprache:

```
... A.B.C       A.B.C.alpha       Regel 2 ist richtig.
... A.B[ ]      A.B[ ].alpha      Regel 7 ist richtig.
... A.B^        A.B^.alpha        Regel 7 ist richtig.
```

Nur für die erste Zeile ist also die von yacc gewählte Standardlösung richtig. Ersichtlicherweise läßt sich also die Situation mit verfügbarem Kontext eines konkreten Satzes von Modula-2 klären. Diese Informationen besitzt aber yacc nicht, wenn es die Analysetabelle zur vorgegebenen Grammatik ausrechnen soll. yacc hat für diese Aufgabe nur die Grammatik.

Aber auch dieser Konflikt läßt sich durch eine bessere Umsetzung der EBNF-Regeln in yacc-Regeln beheben. Die Inkonsequenz des ersten Ansatzes wurde mit der Bescheinigung von reduce/reduce-Konflikten gewissermaßen bestraft. Der folgende yacc-Text realisiert ebenfalls die beiden EBNF-Regeln:

```
23% cat r2.y
%token ident exprlist
%start designator
%%
qualident  : ident
           | qualident     ident
designator : qualident
           | designator '.' ident
           | designator '[' exprlist ']'
           | designator ''
24% yacc r2.y
conflicts: 1 shift/reduce
25%
```

Ganz allgemein kann man feststellen: Während shift/reduce-Konflikte oftmals durchaus beabsichtigte Verhältnisse richtig wiedergeben, sollte man reduce/reduce-Konflikte in jedem Fall zum Anlaß nehmen, die Grammatik sorgfältig zu überprüfen. Die Standardlösung von yacc ist eher ein willkürbehafteter Ausweg. Oftmals zeigt die Analyse von reduce/reduce-Konflikten Fehler in der Grammatik, die bisher unerkannt geblieben sind.

Schließlich soll auf den Einfluß von Prioritäten und Assoziativitäten bei shift/reduce-Konflikten hingewiesen werden. Prioritäten oder Assoziativitäten können terminalen Symbolen durch entsprechende Deklarationen am Anfang von yacc-Programmen zugeeignet werden. Auch Regeln können Prioritäten und Assoziativitäten besitzen. Dies ist der Fall, wenn rechts in einer Regel ein terminales Symbol vorkommt,

das seinerseits Priorität und Assoziativität hat. Priorität und Assoziativität des am weitesten rechts stehenden terminalen Symbols sind dann schließlich Priorität und Assoziativität der Regel. Prioritäten sind Zahlen pr=1, 2, 3, ... , die durch die Reihenfolge der Deklarationen entstehen. Als Assoziativitäten sind left, right und nonassoc zu unterscheiden. Regeln können zudem mit dem Sondermerkmal %prec Attribute zugeordnet bekommen. Es überschreibt anders bestimmte Prioritäten und Asoziativitäten. Klar ist, weder jedes Terminal noch jede Regel verfügt über diese Attribute.

Diskutieren wir Konflikte unter den möglichen Attributierungskombinationen:

(1) Die Regel r oder das Arbeitssymbol s haben keine Prioritäts- oder Assoziativitätsattribute:
Der shift/reduce-Konflikt wird protokolliert, und als Standard wird die shift-Aktion ausgewählt.

(2) Die Regel r und das Arbeitssymbol s haben Priorität und Assoziativität:
Hier sind mehrere Fälle zu unterscheiden.

```
pr(s)>pr(r)                        --------> shift
                Beispiel:    expr + expr    |   *
pr(s)<pr(r)                        --------> reduce r
                Beispiel:    expr * expr    |   +
pr(s)=pr(r), ass(s)=left           --------> reduce r
                Beispiel:    expr + expr    |   +
pr(s)=pr(r), ass(s)=right          --------> shift
                Beispiel:     var = var     |   =
                (Zuweisungen mit mehreren linken Seiten)
pr(s)=pr(r), ass(s)=nonassoc       --------> error
                Beispiel:    expr > expr    |   >
                (Vergleichsoperatoren)
```

In dieser Weise behebbare Konfliktsituationen werden nicht protokolliert. Hier zeigt sich letztlich der Grund dafür, daß yacc auch mit bis zu einem gewissen Grade mehrdeutigen Grammatiken fertig wird. Das Beispiel deskcalc im Abschnitt 6.2.2. besitzt in der Grammatik mehrere Regeln, die mehrdeutig sind.

### 6.4.2. Gestaltung der Fehlerbehandlung in generierten Programmen

In den generierten Parsern muß die Behandlung von Fehlern einsetzen, wenn in einem Analysestadium sa das anstehende Arbeitssymbol as als Eintrag in der Analysetabelle eine error-Aktion vorfindet. Der Analysekeller hat vielleicht folgende Belegung:

```
0 s1     si     sa     as  restl. Eingangstext.
```

Als ersten Schritt in der Behandlung des Fehlers wird im Keller rückwärts nach einem Zustand gesucht, in dem es eine shift-Aktion zum Fehlersymbol gibt. Als ein Beispiel für einen solchen Zustand sei auf das Listing y.output im Abschnitt 6.3.1. hingewiesen. Dort enthält der Zustand state 1 ein »error-shift« (Markierung #####). Unter Umständen wird bei dieser Rückwärtssuche der Keller leergeräumt. Dann zwingt die versuchte Fehlerbehandlung zum Abbruch der ganzen Analyse. Es entsteht yyparse()==1. Nehmen wir also an, der Zustand si enthält die Eintragung:

```
error  shift sn
```

Dann wird diese Aktion ausgeführt, und wir erhalten als neue Belegung des Analysekellers

```
0 s1     sn     as  restl. Eingangstext.
```

Nun kann es sein, daß das Arbeitssymbol as mit dem Zustand sn keine fehlerlose Fortsetzung zuläßt. Dann wird im Eingangstext so lange vorwärts gesucht, bis sich ein Symbol einfindet, das eine shift- oder reduce-Aktion ermöglicht. Wenn dabei das Ende der Datei erreicht wird, hilft natürlich auch hier nur noch der Abbruch mit yyparse()==1.

Die Abläufe zur Fehlerbehandlung werden durch einen Auszug aus der Datei /usr/lib/yaccpar verdeutlicht. yacc nimmt sie als Schlußteil zur Resultatdatei y.tab.c hinzu. Das Anfangsstück von y.tab.c enthält die Analysetabelle, die aus dem Text der Quelldatei name.y hergestellt wird. Der algorithmische Teil, insbesondere die Funktion yyparse(), bildet das Schlußstück. Es ist nicht von name.y abhängig und kann daher aus /usr/lib einfach kopiert werden. Erst anschließend werden die semantischen Aktionen eingefügt, die allerdings wieder aus name.y entnommen werden müssen.

C-Anweisungen, die den Zugriff auf die Analysetabelle realisieren, wurden durch verbale Erklärungen ersetzt, die hier in »Sonderklammern« |* und *| eingefaßt wurden. Ansonsten wird die Fehlerbehandlung durch die Variable yyerrflag gesteuert. Ihr Anfangswert ist 0. Die Variable yynerrs zählt die entdeckten Fehler.

```
switch (yyerrflag) {
  case 0: /*neuer Fehler*/
          yyerror("syntax error, ...");
          yynerrs++;
  case 1:
  case 2: yyerrflag=3;
          while (|*Keller nicht leer*|)
|* Enthält der aktuelle Zustand einen error-shift-Eintrag?
ja:   Führe die shift-Aktion aus, Fortsetzung der Analyse
      mit dem alten Arbeitssymbol.
nein: Nimm den Zustand vom Keller, das nächste
      Kellerelement wird aktueller Zustand. *|
          YYABORT;
  case 3:
          if (yychar==0) YYABORT;
          yychar=(-1);
|*Fortsetzung mit dem nächsten Durchlauf des Analysezyklus.*|
          } /*Ende switch*/
```

In gleicher Weise wird das Anfangsstück des Analysezyklus angegeben. Hier ist die Variable yychar wichtig. Sie hat das gerade behandelte Arbeitssymbol als Wert. Sobald es ausgedient hat, erhält die Variable yychar den Wert -1. Dies signalisiert den Bedarf des nächsten terminalen Symbols. Der Anfangswert von yychar ist ebenfalls -1.

```
if ((yychar<0)&&((yychar=yylex())<0))
  yychar=0;
/*Es gilt: yychar >0: Ein Terminal wurde gefunden.
           yychar >0: Dateiende erreicht.*/
|*Bestimme die auszuführende Aktion.
  Ist sie eine shift-Aktion, so wird die Anweisung
  if (yyerrflag>0) yyerrflag--;
  abgearbeitet. *|
```

Beide Programmfragmente zusammen zeigen die sogenannte Drei-Token-Regel. Treten nach einem behandelten Fehler erneut Fehler auf, so werden sie nicht protokolliert, wenn nicht bereits drei »legal«-shift-Aktionen abgelaufen sind. Das sind solche shift-Aktionen, die durch das Paar (Analysezustand, Arbeitssymbol) aufgrund des Tabelleneintrags shift ausgelöst wurden.

Betrachten wir noch einmal das Programm deskcalc aus Abschnitt 6.2.2. Es enthält als Regel Nr. 4 eine Regel zur Fehlerbehandlung.

```
list   list error '\n'  {yyerrok;}
```

Lassen wir diese Regel zunächst erst einmal ganz weg (deskcalc_1.y).

```
% make -s deskcalc_1
% deskcalc_1
a:=b)3;4
[error 1] line 1 near ")": syntax error, expecting '\n'
%
```

Die Eingabe enthält zwei Fehler bei ')' und ';'. Nur der erste wird angezeigt. Das Programm bricht ab.

Jetzt nehmen wir als Regel Nr. 4 diese hinzu:

```
list   list error '\n'
```

also noch ohne {yyerrok;}. Nennen wir dieses Programm deskcalc_2.y.

```
% deskcalc_2
a:=b)3
[error 1] line 1 near ")": syntax error, expecting: '\n'
;4
5+010
....13
;4
[error 2] line 4 near ";": syntax error,
          expecting: '\n' '(' - DIGIT LETTER
Ctrl-d
%
```

Der erste Fehler wird angezeigt und behandelt. Der zweite Fehler wird auch stabilisiert, aber nicht protokolliert. Erst nach der richtigen Eingabe 5+... werden wieder Fehler angezeigt.

Rechnen wir dagegen mit dem Originalprogramm aus Abschnitt 6.2.2., so erhalten wir:

```
% deskcalc
a:=b)3
[error 1] line 1 near ")": syntax error, expecting: '\n'
;4
[error 2] line 2 near ";": syntax error,
          expecting: '\n' '(' - DIGIT LETTER
Ctrl-d
%
```

Die Beobachtungen an diesen drei Versionen von deskcalc ermuntern dazu, immer dann, wenn aufgrund eines terminalen Symbols (hier '\n') eine geklärte Ausgangsposition vorliegt, durch die Schaltanweisung yyerrok; die Fehlerbehandlung in ihren Initialzustand zurückzustellen.

Nun soll noch ein etwas attraktiveres Schaustück einer Fehlerbehandlung vorgeführt werden. Wir wählen die bedingten Anweisungen aus Modula-2, wobei wir aber wieder Ausdrücke und einfache Anweisungen zu terminalen Symbolen degradieren. Wir beschreiben sie für yacc und lex mit den Dateien m2.y und m2.l.

```
        % cat m2.y
        %token If Then Elsif Else End expr simple
        %%
(1)     stat: simple
(2)         | if_pref stat_seq End {yyerrok;}
(3)         | if_pref stat_seq if_rest End {yyerrok;}
(4)         | error stat_catch {yyerrok; yyless(0);}

(5)     stat_seq: stat
(6)             | stat_seq ';' stat {yyerrok;}
(7)             | stat_seq error stat {yyerrok;}
(8)             | stat_seq error stat_catch {yyerrok; yyless(0);}

(9)     if_pref: If expr Then {yyerrok;}

(10)    if_rest: elsif_seq
(11)           | else_part
(12)           | elsif_seq else_part {yyerrok;}

(13)    elsif_seq: elsif_elem
(14)             | elsif_seq elsif_elem {yyerrok;}

(15)    elsif_elem: elsif_pref stat_seq

(16)    elsif_pref: Elsif expr Then {yyerrok;}

(17)    else_part: Else stat_seq

(18-21) stat_catch:     | Elsif | Else | End ;

        %%
        #include "lex.yy.c"
        main(argc,argv)
        int argc; char **argv; {
        extern int yydebug;
        if (argc>1) yydebug=atoi(argv[1]);
        printf("yyparse()=%d\n",yyparse());
        }
        % cat m2.l
        %{
        #include "y.tab.h"
        %}
        id          [A-Za-z][0-9A-Z_a-z]*
        %%
        [ \n\t]*     ;
        {id}        {if (strcmp(yytext,"If")    ==0) return If;
                     if (strcmp(yytext,"Then")  ==0) return Then;
                     if (strcmp(yytext,"Elsif") ==0) return Elsif;
                     if (strcmp(yytext,"Else")  ==0) return Else;
```

```
                if (strcmp(yytext,"End")   ==0) return End;
                if (strcmp(yytext,"expr")  ==0) return expr;
                return simple;}
              return yytext[0];
        % make -s m2 YFLAGS=-dt
        %
```

Für die Fehlerstabilisierung wurden die Fangregeln (18-21) gebildet. In ihren rechten Seiten stehen die Nachfolgesymbole für Anweisungen bzw. Anweisungsfolgen (stat, stat_seq). Wurde ein Stabilisierungssymbol gefunden, so wird die lexikale Analyse darauf zurückgesetzt (yyless(0);).

Auf eine fällige Fehlerbehandlung in den Konstruktionen if_pref und elsif_pref wurde verzichtet, weil sie nicht typisch wäre. Nur durch unsere künstliche Beschränkung sind diese Gebilde aus drei aufeinanderfolgenden terminalen Symbolen entstanden. In einem realen Zusammenhang müßte expr mit einer eigenen Fehlerstabilisierung ausgestattet werden. Wollte man hier dennoch eine Fehlerbehandlung organisieren, so wäre das durch die Regeln

```
    if_pref   If expr Then {yyerok;}
              If error expr Then {yyerrok;}
              If expr error Then {yyerrok;}
```

zu erledigen und analog auch für elsif_pref.

Wir zeigen Auszüge aus Spurverfolgungsprotokollen zweier fehlerhafter Eingangstexte.

```
    % cat tsttxt1
    If expr Then a a End
    % m2 < tsttxt1
    [error 1] line 1 near "a": syntax error,
                               expecting: ';' Elsif Else End
    yyparse()=0
    % m2 1 < tsttxt1
      ...
    State 7
    Stack: if_pref stat
    Reduce by (5) "stat_seq   stat"
    State 6
    Stack: if_pref stat_seq
    Received token simple
    [error 1] line 1 near "a": syntax error,
                               expecting: ';' Elsif Else End
    State 18, token simple
    Stack: if_pref stat_seq [error]
    State 2
    Stack: if_pref stat_seq [error] simple
    Reduce by (1) "stat : simple"
    State 30
    Stack: if_pref stat_seq [error] stat
    Reduce by (7) "stat_seq : stat_seq error stat"
    State 6
    Stack: if_pref stat_seq
    Received token End
    State 15
    Stack: if_pref stat_seq End
    Reduce by (2) "stat : if_pref stat_seq End"
    State 1
```

```
Stack: stat
Received token [end of file]
yyparse()=0
%
```

Hier wird nach der Regel (7) stabilisiert. Das nächste Beispiel demonstriert die anderen beiden Fehlerregeln (4) und (8). Zugleich wird hier die Rückwärtssuche im Analysekeller sichtbar, um einen Zustand mit einer »error-shift«-Aktion zu finden. Anschließend wird im Eingangstext vorwärts nach einem Fortsetzungsterminal gesucht.

```
% cat tsttxt2
If expr Then a
  Elsif expr Then Elsif *
  Else x End
% m2 < tsttxt2
[error 1] line 2 near "Elsif": syntax error, expecting: If simple
[error 2] line 2 near "*": syntax error, expecting: expr
yyparse()=0
% m2 1 < tsttxt2   sed '/State [0-9][0-9]*/d'
  ...
Stack: if_pref stat_seq elsif_pref
Received token Elsif
[error 1] line 2 near "Elsif": syntax error, expecting: If simple
Stack: if_pref stat_seq elsif_pref [error]
Stack: if_pref stat_seq elsif_pref [error] Elsif
Reduce by (22) "stat_catch : Elsif"
Stack: if_pref stat_seq elsif_pref [error] stat_catch
Reduce by (4) "stat : error stat_catch"
Stack: if_pref stat_seq elsif_pref stat
Reduce by (5) "stat_seq : stat"
Stack: if_pref stat_seq elsif_pref stat_seq
Received token Elsif
Reduce by (17) "elsif_elem : elsif_pref stat_seq"
Stack: if_pref stat_seq elsif_elem
Reduce by (15) "elsif_seq : elsif_elem"
Stack: if_pref stat_seq elsif_seq
Stack: if_pref stat_seq elsif_seq Elsif
Received token '*'
[error 2] line 2 near "*": syntax error, expecting: expr
Error recovery pops state 24, uncovers state 19
               pops symbol Elsif , uncovers symbol elsif_seq
Error recovery pops state 19, uncovers state 6
               pops symbol elsif_seq   uncovers symbol stat_seq
Stack: if_pref stat_seq [error]
Error recovery discards token '*'
Received token Else
Stack: if_pref stat_seq [error] Else
Reduce by (23) "stat_catch : Else"
Stack: if_pref stat_seq [error] stat_catch
Reduce by (8) "stat_seq : stat_seq error stat_catch"
Stack: if_pref stat_seq
  ...
Stack: if_pref stat_seq if_rest End
Reduce by (3) "stat : if_pref stat_seq if_rest End"
Stack: stat
Received token [end of file]
yyparse()=0
%
```

Die Fehlerstabilisierung auf der Basis der eben gezeigten Fangmengen stößt in komplexeren Zusammenhängen an Grenzen, wenn mehrere Mengen gebildet werden müssen, die gemeinsame Elemente besitzen. Da jedoch Fangmengen ausschließlich terminale Symbole produzieren, können sie ganz und gar in die Semantik verlagert werden. Anstelle des Nichtterminals stat_catch wäre eine Routine stat_catch() als C-Funktion zu definieren.

```
stat_catch() { int c;
c=yychar;
while (c>0 && c!=';' && c!=Elsif && c!= Else && c!=End)
  c=yylex();
yyerrok; yyclearin; yyless(0);
}
```

Die Anweisung yyclearin; veranlaßt yyparse(), das nächste Terminal abzufordern. Dies ist das durch yyless(0); an yylex() zurückgegebene Stabilisierungssymbol. Nun müssen in m2.y die modifizierten Regeln (4a) und (8a) eingefügt werden. Die Regeln (18-21) entfallen.

```
(4a)      stat : error  {stat_catch();}
(8a)  stat_seq : stat_seq error  {stat_catch();}
```

Nochmals abkürzend kann auch die Funktion stat_catch1() benutzt werden.

```
stat_catch1() {
while (yychar>0 && yychar!=';' && ... )
  if ((yychar=yylex())<0) yychar=0;
yyerrok;
}
```

### 6.4.3. Spurverfolgung bei der Arbeit generierter Programme

Für den Test von generierten Programmen ist die ´Spurverfolgung ein unentbehrliches Hilfsmittel. Die diesbezüglichen Vorkehrungen sind in der Datei /usr/lib/yaccpar zu finden. Da diese Datei zu den wenigen öffentlich zugänglichen C-Quellen von UNIX-Werkzeugen gehört, sind die Spurverfolgungseinschübe ständig Gegenstand von nutzerspezifischen Modifikationen und Verbesserungen gewesen. Ein deutlicher Schritt in der Qualifizierung des Spurverfolgungsservices wurde von Schreiner und Friedman (1985) veröffentlicht. Seitdem UNIX-System V mit einer wiederum verbesserten Spurverfolgungsunterstützung verfügbar wurde, hat Schreiner (1987) beides miteinander vereint. In diesem Buch wird eine Spurverfolgung verwendet, die einen weiteren Zusatz erhielt, so daß in vielen Fällen Rückgriffe auf die Datei y.output entbehrlich werden.

Die Spurverfolgung erfordert den Aufruf von yacc mit der Option -t oder eine Übersetzung des resultierenden C-Programms mit der Option -DYYDEBUG=1. Wir modifizieren unser Hintergrund-Makefile.

```
% cat Makefile
SHELL=/bin/sh
LDFLAGS=-ly -ll
YFLAGS=-dt
.y: ;   -test -r $*.l && $(LEX) $(LFLAGS) $*.l
        $(YACC) $(YFLAGS) $<
        sed -f /usr/lib/yaccsed.script y.tab.c > $*.c
        rm -f y.tab.c
```

```
        $(CC) -o $@ $(CFLAGS) -LARGE $*.c $(LDFLAGS)
        rm -f $*.c lex.yy.c
```

Die Datei /usr/lib/yaccsed.script wird im Anhang 5 vollständig abgedruckt. Sie ist zugleich ein demonstratives Anwendungsbeispiel für die Arbeit mit dem Stream-Editor sed, der im Abschnitt 3.2. vorgestellt wurde. Die Spurverfolgung zum Quelltext source.[ly] wird wirksam, wenn die Variable yydebug mit einem Wert ≠0 besetzt wird. Das kann durch

```
% adb -w source
* yydebug/w1
_yydebug:       0.=     1.
* $q
% source
yydebug = 1
debug-mode: States, Stack:   1
max. number of stack-items:  8
            Received token:  1
            Pops, uncovers:  1
            Discards token:  1
            Reduce by rule:  1
State 0
  ...
% adb -w source
* yydebug/w0
_yydebug:       1.=     0.
* $q
%
```

erreicht oder wieder rückgängig gemacht werden. Eine andere Lösung wurde in den m2-Beispielen im Abschnitt 6.4.2. gezeigt. Dort wurde die Wertbelegung von yydebug in das Hauptprogramm verlagert.

Spurverfolgungszusätze gibt es für fünf Positionen innerhalb des Analysezyklus von yyparse().

(1) Protokoll des erreichten Analysezustands mit einer Übersicht über die Belegung der oberen Kellerplätze mit steuernden Symbolen. Die Maximalzahl der protokollierten Kellerplätze ist durch die C-Compiler-Option -DYYDEBST=max wählbar. Voreingestellt ist YYDEBST=8.
    State nummer
    Stack:  symbol symbol

(2) Protokoll des aktuell gerade bereitgestellten Arbeitssymbols.
    Received Token terminales Symbol (Name oder ASCII-Kette)

(3) Protokoll der Zyklusschritte bei der Rückwärtssuche im Analysekeller, um einen Zustand mit »error-shift« zu finden.
    Error recovery pops state nummer, uncovers state nummer
                      symbol name,  uncovers symbol name

(4) Protokoll der Zyklusschritte bei der Vorwärtssuche im Eingangstext, um ein Eingabesymbol zu finden, das eine stabilisierte Fortsetzung erlaubt.
    Error recovery discards token symbol (Name oder ASCII-Kette)

(5) Protokoll von Reduktionen.
    Reduce by (nummer) "Regel-Text"

Da der Umfang der entstehenden Protokolle sehr schnell ins Uferlose wächst, wurde durch Schreiner (1987) eine Auswahl eingeführt, die abhängig vom Wert von yydebug bei den fünf aufgezählten Möglichkeiten Protokollpunkte ein- oder ausschaltet. Die Variable yydebug wird als

```
x0 + 2*y(1) + 4*y(2) + 8*y(3) + 16*y(4) + 32*y(5) == yydebug
```

dargestellt und hat folgende Wirkungen (y(i)=0¦1):

x0=1: Alle Protokollpunkte sind aktiv, jedoch wird mit y(i)=1 die Druckposition i ausgeschaltet.

x0=0: Alle Druckpositionen sind ausgeschaltet, jedoch mit y(i)=1 wird der Protokollpunkt i aktiviert.

Die listige Funktion, die dieses Verhalten ermöglicht, ist:

```
# define z yydebug
# define y(i) ((z^z>>i)&1)
```

Mit yydebug als Argument hat sie folgende Wertetabelle:

| | yydebug | y(1) | y(2) | y(3) | y(4) | y(5) |
|---|---|---|---|---|---|---|
| 0 + 0 + 0 + 0 + 0 + 0 | = 0 | 0 | 0 | 0 | 0 | 0 |
| 1 + 0 + 0 + 0 + 0 + 0 | = 1 | 1 | 1 | 1 | 1 | 1 |
| 0 + 2 + 0 + 0 + 0 + 0 | = 2 | 1 | 0 | 0 | 0 | 0 |
| 1 + 2 + 0 + 0 + 0 + 0 | = 3 | 0 | 1 | 1 | 1 | 1 |
| 0 + 0 + 4 + 0 + 0 + 0 | = 4 | 0 | 1 | 0 | 0 | 0 |
| 1 + 0 + 4 + 0 + 0 + 0 | = 5 | 1 | 0 | 1 | 1 | 1 |
| 0 + 2 + 4 + 0 + 0 + 0 | = 6 | 1 | 1 | 0 | 0 | 0 |
| 1 + 2 + 4 + 0 + 0 + 0 | = 7 | 0 | 0 | 1 | 1 | 1 |
| ... | | | | ... | | |
| 1 + 0 + 4 + 8 +16 +32 | = 61 | 1 | 0 | 0 | 0 | 0 |
| 0 + 2 + 4 + 8 +16 +32 | = 62 | 1 | 1 | 1 | 1 | 1 |
| 1 + 2 + 4 + 8 +16 +32 | = 63 | 0 | 0 | 0 | 0 | 0 |

Interessant sind diese Werte von yydebug.

| yydebug | |
|---|---|
| 0,63 | kein Protokoll |
| 1,62 | alle Protokolldrucke |
| 2,61 | Kellerentwicklung |
| 4,59 | Tokenverarbeitung |
| 31,32 | Reduktionen |
| 24,39 | Fehlerbehandlung |

## 6.5. Beispiel einer Programmentwicklung

### 6.5.1. Lexikalische und syntaktische Analyse

In der schon einmal zitierten Arbeit von Johnson (1978) wird eine Grammatik der yacc-Eingabesprache angegeben und mit den Ausdrucksmitteln von yacc formuliert. Wir stellen sie an den Anfang einer Programmentwicklung und ergänzen sie um jene Teile, die für ihre Umsetzung in ein abarbeitbares Programm erforderlich sind. Damit soll die Möglichkeit geschaffen werden, beliebige yacc-Eingabetexte auf ihre syntaktische Richtigkeit hin zu überprüfen.

```
11% cat myaccg.y
    /*Grammatik fuer yacc-Quelltexte*/
    /*Deklarationsteil*/
%token IDENTIFIER   /*umfasst Bezeichner und Literale*/
%token C_IDENTIFIER /*Bezeichner mit nachfolgendem Doppelpunkt*/
%token NUMBER       /*[0-9]+*/
    /*reservierte Woerter: %type=>TYPE, %left=>LEFT, usw.*/
%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION
%token MARK  /* %%-Marke*/
%token LCURL /* %{ Anfang eines C-Textes*/
%token RCURL /* %} Ende eines C-Textes*/
    /*ASCII-Zeichen bedeuten sich selbst als Terminal*/
%start spec

%%

spec  : defs MARK rules tail
      ;
tail  : MARK {eat_up();} /*Uebergehe den Rest des Quelltextes*/
      | /*leer, die zweite %%-Marke ist optional*/
      ;
defs  : /*leer*/
      | defs def
      ;
def   : START IDENTIFIER
/*    | UNION {Kopiere das Innere der union-Definition} */
/*    | LCURL {Kopiere C-Text in die Datei y.tab.c} RCURL */
      | rword tag nlist
      ;
rword : TOKEN
      | LEFT
      | RIGHT
      | NONASSOC
      | TYPE
      ;
tag   : /*leer*/
      | '<' IDENTIFIER '>'
      ;
nlist : nmno
      | nlist nmno
      | nlist ',' nmno
      ;
nmno  : IDENTIFIER
      | IDENTIFIER NUMBER
      ;
    /*Regelteil*/

rules : C_IDENTIFIER rbody prec
      | rules rule
      ;
rule  : C_IDENTIFIER rbody prec
      | '|' rbody prec
      ;
rbody : /*leer*/
      | rbody IDENTIFIER
      | rbody act
      ;
act   : '{' {copy_act();} '}' /*Uebergib semantische Aktionen*/
      ;
```

```
prec  : /*leer*/
      | PREC IDENTIFIER
      | PREC IDENTIFIER act
      | prec ';'
      ;
%%
#include <stdio.h>
#include "lex.yy.c"
eat_up() { int c;
while ((c=getchar())!=EOF); unput(c);
}
copy_act() { int c;
while ((c=input())!='}'); unput(c);
}
12%
```

Wenn myaccg als Programm startklar hergestellt werden soll, benötigen wir selbstverständlich die Funktion yylex(). Wir lassen sie durch lex generieren und verwenden dazu dieses lex-Programm:

```
12% cat myaccg.l
id                  [A-Za-z][0-9A-Z_a-z]*
%start cid
%%
"/*"[^*]*"*/"       |   /* Kommentare eliminieren */
"%{"[^%]*"%}"       ;   /* Einschlüsse streichen */
{id}/[ \t]*:        {BEGIN cid; return C_IDENTIFIER;}
<cid>[ \t]*:        {BEGIN 0;}
{id}                {return IDENTIFIER;}
[0-9]+              return NUMBER;
"%%"                return MARK;
"%{"                return LCURL;
"%}"                return RCURL;
"%start"            return START;
"%token"            return TOKEN;
"%left"             return LEFT;
"%right"            return RIGHT;
"%nonassoc"         return NONASSOC;
"%type"             return TYPE;
"%union"            return UNION;
"%prec"             return PREC;
\'[^']+\'           return IDENTIFIER;
[ \t\n]*            ;
.                   return *yytext;
%%
yywrap(){return 1;}
main(){printf("myaccg: yyparse()=%d\n",yyparse());}
13%
```

Hervorgehoben werden muß, daß der yacc-spezifische Komfort im Umgang mit dem Semikolon als Trennzeichen eine besondere Aufmerksamkeit erfordert. Johnson weist schon darauf hin, daß die angegebene yacc-Eingabegrammatik eigentlich vom Typ LR(2) ist. Seinem Vorschlag folgend beheben wir diese Schwierigkeit durch die Unterscheidung von IDENTIFIER und C_IDENTIFIER bei den deklarierten Terminalsymbolen.

Nun können wir in bekannter Weise myaccg anlegen lassen und starten. Wir wählen den yacc-Text myaccg.y zugleich als Testeingabedatei für myaccg.

```
13% make myaccg -s
14% myaccg < myaccg.y
myaccg: yyparse()=0
15%
```

Jetzt soll das Progamm myaccg mit weiterer Semantik angereichert werden. Es soll eine yacc-ähnliche Numerierung der Symbole entstehen. Das heißt, Nichtterminale werden fortlaufend negativ gezählt, ASCII-Zeichen behalten ihren Kodierungswert und Terminale werden mit 257 beginnend numeriert. Dazu ist eine Bezeichnerverwaltung erforderlich. In einem allerersten Ansatz dient diesem Zweck eine rückwärts verkettete Liste. Das dafür aufbereitete Programmaterial befindet sich im Anhang 6 in den Dateien myaccy.y, myacclex.l und sym.c. Dort gibt es die Datei makefile, die alle Bedürfnisse der im weiteren verfolgten Programmentwicklung befriedigt. Die numerierende Variante nennen wir myaccn. Wir zeigen ihre Funktionsweise mit einer Grammatik von Loeper u.a. (1980, 1987). Sie wurde als yacc-Text in die Datei loe.y geschrieben. Hier sind die Resultate der einzelnen Schritte:

```
11% cat loe.y
%token if then else fi id print
%%
Satz      : Anweisung
Anweisung: print id
          | if Clausel fi
Clausel   : Dann else Anweisung
          | Dann
Dann      : Bedingung then Anweisung
Bedingung: id '<' id
12% make -s myaccn
13% myaccn < loe.y
myaccn: yyparse()=0
nbr=  -5  name= "Bedingung"
nbr=  -4  name= "Dann"
nbr=  -3  name= "Clausel"
nbr=  -2  name= "Anweisung"
nbr= 262  name= "print"
nbr= 261  name= "id"
nbr= 260  name= "fi"
nbr= 259  name= "else"
nbr= 258  name= "then"
nbr= 257  name= "if"
nbr=  -1  name= "Satz"
14%
```

Mit diesen Nummern nimmt die Grammatik aus loe.y folgende Gestalt an:

```
-1 :   -2
-2 :  262   261
-2 :  257    -3   260
-3 :   -4   259    -2
-3 :   -4
-4 :   -5   258    -2
-5 :  261    60   261
```

Der weitere Gang unserer Programmentwicklung führt zur Berechnung einer Analysetabelle. Wir werden einen stark vereinfachten Weg wählen, um zu diesem Ziel zu kommen. Einschränkende Annahmen sind dazu unabdingbar. Der erreichbare Gewinn besteht in der Transparenz der Lösung. Einschränkungen betreffen die verwertbaren Grammatiken.

Sie dürfen keine Regeln mit leeren rechten Seiten enthalten. Das Satzsymbol darf genau einmal vorkommen, und zwar als linke Seite der ersten Regel. Assoziativitäten und Vorränge von Operatoren müssen entfallen. Wir verzichten auch auf den Wertekeller und können daher %union- oder %type-Deklarationen weglassen.

Um aber doch auch unseren mit deskcalc.y konstruierten »Tischrechner« als Testfall einbeziehen zu können, passen wir seine Grammatik an die Einschränkungen an und erhalten dafür die Datei pc.y.

```
21% cat pc.y
%token DIGIT LETTER quit
%%
sent : list quit
list : stat '\n'
     | list stat '\n'
stat : expr
     | LETTER '=' expr
expr : term
expr : expr '+' term
expr : expr '-' term
term : factor
term : term '*' factor
factor : LETTER
       | number
factor : '(' expr ')'
number : DIGIT
       | number DIGIT
22%
```

### 6.5.2. Tabellengesteuerte Abarbeitung

Zur Grammatik loe.y berechnet das im Abschnitt 6.5.3. noch zu beschreibende Verfahren folgende Analysetabelle als Matrix mit Zustandsnummern als Zeilenindex und den fortlaufend numerierten Symbolen des Vokabulars als Spaltendindex. Sie stimmt bis auf die Numerierung der Zustände mit der vom Original-yacc berechneten überein.

| j,v | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| j,sy | | -1 | -2 | -3 | -4 | -5 | 60 | 257 | 258 | 259 | 260 | 261 | 262 |
| | eof | Satz | Anw | Clau | Dann | Bed | < | if | then | else | fi | id | print |
| 1 | 0 | 30 | 2 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 4 |
| 2 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 3 | 0 | 0 | 0 | 5 | 6 | 7 | 0 | 0 | 0 | 0 | 0 | 8 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | -5 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 13 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 |
| 10 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | -3 |
| 11 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 4 |
| 12 | 0 | 0 | 15 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 4 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 0 |
| 14 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 |
| 15 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 | -6 |
| 16 | -7 | -7 | -7 | -7 | -7 | -7 | -7 | -7 | -7 | -7 | -7 | -7 | -7 |

Zu so gespeicherten Analysetabellen gehört ein einfacher Parsing-Algorithmus. Das komplette C-Programm dafür befindet sich in der Datei parse.c im Anhang 6. Der Kellerspeicher für die Analyse wird als Feld myys mit einer vorgegebenen Maximallänge angelegt. Entsprechend dem Grundprinzip der LR-Analysetechnik wählt der Parsing-Algorithmus unter vier Fällen aus. Als Matrixelement findet er die Aktion act:

act=0
    Syntaxfehler, Abbruch.
act=NST_MAX
    Akzeptieren als Satz. (NST_MAX ist die maximal mögliche Zustandsnummer, hier wurde NST_MAX=30 gewählt.)
act=n>0 shift-Aktion.
    Der nächste Zustand ist n. Wenn die shift-Aktion nicht als Folge einer Reduktion abläuft, muß das nächste Arbeitssymbol bereitgestellt werden.
act=r<0 reduce-Aktion.
    Reduktion entsprechend Regel -r. Die linke Seite dieser Regel wird zum neuen Arbeitssymbol.

Mit dem Satz

```
11% cat tstif
if alpha < beta then print gamma fi
12%
```

erhält man wiederum als Vorgriff das folgende Arbeitsprotokoll von myacc. In der Datei parse.c im Anhang 6 müssen dazu Druckanweisungen, die dort in Kommentarklammern eingefaßt sind, wirksam gemacht werden.

```
12% myacc < loe.y tstif
yyparse()=0
actsy=257 actv=  7  shift  3  STACK: 1
actsy=261 actv= 11  shift  8  STACK: 1 3
actsy= 60 actv=  6  shift 13  STACK: 1 3 8
actsy=261 actv= 11  shift 16  STACK: 1 3 8 13
actsy=258 actv=  8 reduce -7  STACK: 1 3 8 13 16
          Regel  7: Bedingung: id < id
actsy= -5 actv=  5  shift  7  STACK: 1 3
actsy=258 actv=  8  shift 12  STACK: 1 3 7
actsy=262 actv= 12  shift  4  STACK: 1 3 7 12
actsy=261 actv= 11  shift  9  STACK: 1 3 7 12 4
actsy=260 actv= 10 reduce -2  STACK: 1 3 7 12 4 9
          Regel  2: Anweisung: print id
actsy= -2 actv=  2  shift 15  STACK: 1 3 7 12
actsy=260 actv= 10 reduce -6  STACK: 1 3 7 12 15
          Regel  6: Dann: Bedingung then Anweisung
actsy= -4 actv=  4  shift  6  STACK: 1 3
actsy=260 actv= 10 reduce -5  STACK: 1 3 6
          Regel  5: Clausel: Dann
actsy= -3 actv=  3  shift  5  STACK: 1 3
actsy=260 actv= 10  shift 10  STACK: 1 3 5
actsy=  0 actv=  0 reduce -3  STACK: 1 3 5 10
          Regel  3: Anweisung: if Clausel fi
actsy= -2 actv=  2  shift  2  STACK: 1
actsy=  0 actv=  0 reduce -1  STACK: 1 2
          Regel  1: Satz: Anweisung
actsy= -1 actv=  1 accept
myyparse()=0
```

### 6.5.3. Generierung von Analysetabellen

Der Anhang 6 enthält auch die Programme, mit denen die Analysetabelle im Abschnitt 6.5.2. testfallmäßig berechnet wurde. Mit ihnen kann das Programm myacc hergestellt werden. Alle dafür notwendigen Vorkehrungen und Abhängigkeiten sind in der Datei makefile zu finden. Das Programm myacc, mit den Mitteln von Johnsons yacc hergerichtet, hat inzwischen so viele Gemeinsamkeiten mit dem Original gewonnen, daß wir es berechtigtermaßen »Mini-yacc« nennen können.

Beschäftigen wir uns zu Beginn mit der Berechnung der shift-Aktionen. Wir folgen dabei weitgehend den Ansätzen von Loeper u.a. (1980, 1987). Alle möglichen shift-Aktionen einer Analysetabelle berechnen heißt zuerst einmal, alle zu unterscheidenden Zustände zu finden. Bausteine für Zustände sind Paare (r,p) mit einer Regel r und der Position p. Beispielsweise befinden sich in der Regel 4 von loe.y die 4 Paare (r,p):

```
(4,0) ≡  -3 ≡ Clausel
(4,1) ≡  -4 ≡ Dann
(4,2) ≡ 259 ≡ else
(4,3) ≡  -2 ≡ Anweisung
```

Zustände des Parsing-Algorithmus sind Mengen von solchen Paaren. Am Anfang steht das Paar (1,0). Es gehört zum Satzsymbol der Grammatik. Wurde erst einmal ein Zustand als Menge von Paaren (r,p) abgegrenzt, so bildet er den Ausgangspunkt für den Versuch, weitere Zustände zu formulieren. Das Verfahren bricht ab, wenn sich alle vermeintlich neuen bereits unter den bekannten Zuständen befinden.

Zustände sind Vereinigungen von zwei Mengen, die nacheinander entstehen. Die erste heißt Zustandsbasis. Bei der Suche nach einem neuen Zustand entsteht sie als neue Zustandsbasis. Ist sie von den Basismengen der bekannten Zustände verschieden, handelt es sich wirklich um einen neuen Zustand. Er wird zu Ende berechnet, indem zu der gefundenen Basismenge als zweite Menge die Zustandshülle bestimmt wird.

Dieser Ablauf zerfällt in folgende Einzelschritte:

```
Die erste Basismenge bekommt das Paar (1,0) als einziges Element.
Bilde dazu die Zustandshülle.

Für alle bekannten Zustände stati,
für alle Symbole sy:
  Versuche die Basismenge eines Nachfolgezustandes statk
  zu konstruieren.
  Ist diese Menge von den Basismengen
  der bekannten Zustände statjj verschieden?
    ja:
      Es entsteht ein neuer Zustand, bilde seine Hülle.
      Trage in die Analysetabelle, Zeile stati, Spalte sy,
      die Aktion »shift statk« ein.
    nein:
      Der Zustand ist als statjj bekannt.
      Trage in die Analysetabelle, Zeile stati, Spalte sy,
      die Aktion »shift statjj« ein.
      Entferne die Paare, die für statk berechnet wurden.
```

Bild 6.5 zeigt die verwendete Datenstruktur. Es handelt sich um ein Feld von Paaren (r,p), das in den Programmen als Variable Statest

deklariert wurde. Für jeden Zustand stati zeigt Stateind[stati] auf seinen Anfang im Feld Statest. Nstatebase[stati] ist die Zahl der Paare seiner Zustandsbasis. Entstehen Paare (r',p'), so werden sie durch die Funktion stateappend() an die betreffende Menge angehängt. Alle für die Berechnung der shift-Aktionen nötigen Funktionen stehen in der Datei shift.c des Anhangs 6.

Im Zyklus »Für alle stati, für alle sy« gilt: Für alle Elemente (r,p) des Zustands stati sind die Paare (r,p+1) Elemente der Basismenge eines Nachfolgezustands, falls in der Regel r auf Platz p+1 das Symbol sy steht.

Gehört das Paar (r,p) zu einer Menge von Zustandselementen und steht auf Platz p+1 der Regel r das Nichtterminal sy, so werden für alle Regeln r' mit der linken Seite sy die Paare (r',0) in die Zustandshülle aufgenommen. Die Hüllenbildung wird transitiv abgeschlossen.

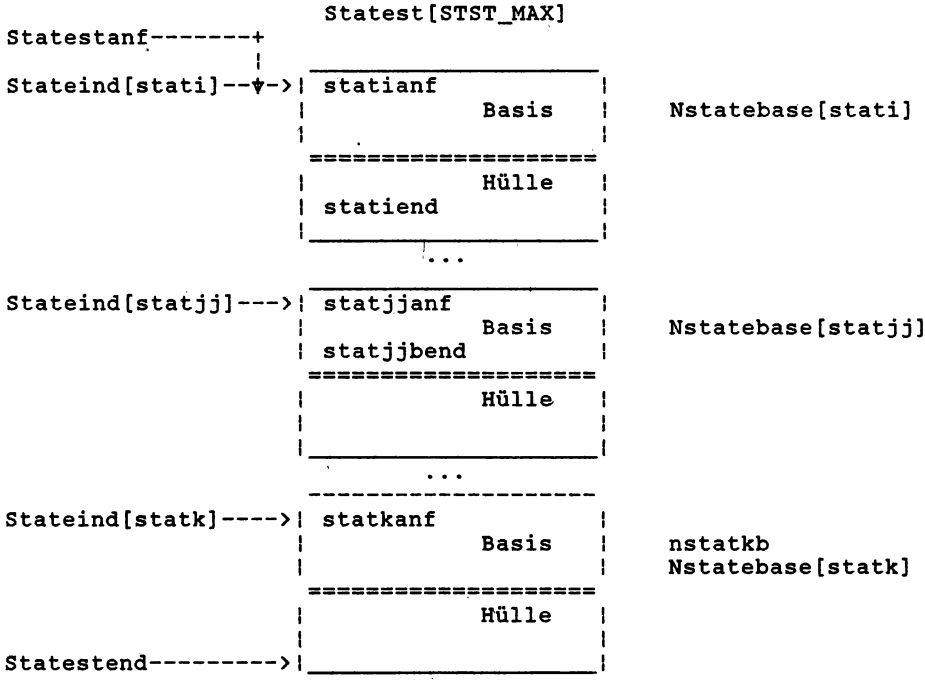



Bild 6.5. Datenstruktur Statest

Auf die Berechnung der shift-Aktionen folgt die Bestimmung der reduce-Aktionen. Die dafür erforderlichen Programme befinden sich in der Datei reduce.c im Anhang 6.

Zuerst wird nach vollständigen Zustandselementen gesucht. Das sind solche Paare (r,p), bei denen p das letzte Symbol auf der rechten Seite der Regel r indiziert.

Wenn es für den Zustand stati genau ein vollständiges Zustandselement gibt und für ihn noch keine shift-Aktion berechnet wurde, handelt es sich um einen reinen Reduktionszustand. In der Analysetabelle wird die ganze Zeile stati mit »reduce r« ausgefüllt. Im anderen Fall wird zum Arbeitssymbol tsy für das vollständige Zustandselement (r,p) mit der Regel r=(li,re1 re2 ... ) die Aktion »reduce r« eingetragen, wenn tsy zur Nachfolgemenge des Nichtterminals li gehört. Es kann sein, daß der Platz in der Analysetabelle bereits ≠0 besetzt ist. Dann handelt es sich um eine konfliktbelastete Grammatik. Die ordnungsgemäße Arbeit mit der berechneten Analysetabelle kann nicht zugesichert werden.

Für die Entscheidung, ob ein Symbol in der Nachfolgemenge eines anderen enthalten ist, wird bedauerlicherweise die Nachfolgerelation der Grammatik benötigt, zu deren Berechnung wiederum die Anfangsrelation gebraucht wird. Die Anfangsrelation gibt an, mit welchen Symbolen Satzteile beginnen können, die sich auf ein gegebenes Nichtterminal reduzieren lassen. Die Nachfolgerelation legt fest, welche Symbole auf Satzteile folgen können, die sich auf ein gegebenes Nichtterminal reduzieren lassen. Die Programme zur Berechnung der Relationen befinden sich in der Datei relgen.c im Anhang 6. Für die Beispielgrammatik loe.y berechnen diese Programme folgende Relationen:

Anfangsrelation

| tsy | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|
| | 60 | 257 | 258 | 259 | 260 | 261 | 262 |
| sy | < | if | then | else | fi | id | print |
| -1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| -2 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| -3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| -4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| -5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Nachfolgerelation

| 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|
| 60 | 257 | 258 | 259 | 260 | 261 | 262 |
| < | if | then | else | fi | id | print |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Relationen werden als Folgen von Paaren gespeichert, die Anfangsrelation als Feld Inirel und die Nachfolgerelation als Feld Sucrel. Die Rolle eines Zwischenspeichers spielt die Relation Rrel.

Anfangsrelation Inirel:
    Paare (nt,t) mit dem Nichtterminal nt und dem terminalen
    Symbol t:
    Für alle Regeln (l,r1 r2 ... ) kommen die Paare (l,r1) zu Inirel.
    Bildung der transitiven Hülle.
    Weglassen der Paare (a,e) mit nichtterminalem e.
Relation Rrel:
    Für alle Regeln (l,r1 ... rn) kommen die Paare (rn,l) zu Rrel
    (rn ist rechtes Zwischensymbol von l).
    Bildung der transitiven Hülle.
Nachfolgerelation Sucrel:
    Paare (nt,t) mit dem Nichtterminal nt und dem terminalen
    Symbol t:
    Für alle Regeln (l,r1 ... r r' ... rn) mit nichtterminalem r:
      1. r'=t terminales Symbol: (r,t) kommt zu Sucrel.
      2. r' ist Nichtterminal: (r,t) kommt zu Sucrel, falls
                               (r',t) zu Inirel gehört.
      3. Falls (re,ra) zu Rrel und (re,t) bereits zu Sucrel
         gehört, so kommt auch (ra,t) zu Sucrel hinzu.

### 6.5.4. Vorführung des vollständigen Programms

Mit den in mühevoller Kleinarbeit zusammengetragenen Teilen kann nunmehr zu einer gegebenen Grammatik eine LR-Analysetabelle berechnet werden. Nur geben wir uns damit noch immer nicht zufrieden. Mit einem passend organisierten Parsing-Algorithmus soll die eben berechnete Analysetabelle ihre Funktionstüchtigkeit zeigen. Er befindet sich in der Datei parse.c des Anhangs 6.

Original-yacc erzeugt die Datei y.tab.c, die sich mit der Funktion yylex() zusammen als generiertes Programm verselbständigen läßt. Wir scheuen den dazu erforderlichen Aufwand an Ein-/Ausgabe-Programmierung und binden alles zusammen in das Programm myacc. Allerdings entsteht dabei eine Verdopplung der lexikalischen Analyse. Zusätzlich zu myacclex.l brauchen wir mit loe.l oder pc.l eine lexikalische Analyse für die beiden ausgewählten oder für weitere Testgrammatiken. An die Seite von yylex(), das aus myacclex.l hervorgegangen ist, stellt sich myylex() zur lexikalischen Analyse für die Testgrammatiken. Kollisionen zwischen yylex() und myylex() verhindert der Zwischenfilter lexsed.script (s. Anhang 6).

Um für alle diese Teile getrennte Dateien zuzulassen, erzeugt unser Generator myacc ähnlich dem Original-yacc eine Datei my.tab.h, die in loe.l oder pc.l eingeschlossen werden kann. Die folgenden Kommandos zeigen den Arbeitsablauf:

```
1% myacci <loe.y
2% myacc <loe.y tstif
yyparse()=0
Quelltext: tstif
myyparse()=0
```

Die Testarbeit mit tstif verwendet die lexikalische Analyse loe.l. Wir zeigen .l-Programme für loe.y und für pc.y.

```
3% cat loe.l
%{
#include "my.tab.h"
%}
id          [A-Za-z][0-9A-Z_a-z]*
%%
[ \t]*      ;
{id}        {if (strcmp(yytext,"if")    ==0) return tokenif;
             if (strcmp(yytext,"then")  ==0) return tokenthen;
             if (strcmp(yytext,"else")  ==0) return tokenelse;
             if (strcmp(yytext,"fi")    ==0) return tokenfi;
             if (strcmp(yytext,"print") ==0) return tokenprint;
             return tokenid;}
\n          ;
.           return yytext[0];
%%
yywrap() {return 1;}
4% cat pc.l
%{
#include "my.tab.h"
%}
%%
[ \t]*      ;
quit        return tokenquit;
[a-z]       return tokenLETTER;
[0-9]       return tokenDIGIT;
```

```
\n          |
            return yytext[0];
%%
yywrap() {return 1;}
5%
```

Insgesamt ist die Zahl der beteiligten Objekte ganz schön angewachsen. Nur die bedachte Organisation auf der Grundlage der Datei makefile kann hierbei Ordnung halten. Das Studium der Datei makefile aus Anhang 6 wird zugleich die Kenntnisse über make bereichern, die im Abschnitt 5. dieses Buches angeboten wurden. Die weiteren Kommandos benutzen diese Datei auch zum Test der generierten Parser.

Vorher soll das Bild 6.6 zeigen, wie die beteiligten Dateien dieses Beispiels zusammenwirken. Eine Variante der Programmgenerierung wurde im Abschnitt 6.2. als Syntaxprüfer beschrieben (Bild 6.3). Das hier entstandene »Mini-yacc« wird getestet, indem damit Syntaxprüfer für Sprachen hergestellt und gestartet werden, die durch »Mini-yacc«-Eingabegrammatiken beschrieben werden. Es handelt sich um die Eingabegrammatiken loe.y und pc.y. Sie berücksichtigen den gegenüber dem Original-yacc eingeschränkten Sprachumfang. Keine rechte Seite ist leer, das Satzsymbol steht links in der ersten Regel, sie sind konfliktfrei, enthalten keine Semantik usw. Andererseits wurde »Mini-yacc« mit den Mitteln von yacc hergestellt. So wurde dem Syntaxprüfer gemäß Bild 6.3 noch eine Generierungsstufe vorgelagert. Gerade das soll das Bild 6.6 verdeutlichen.

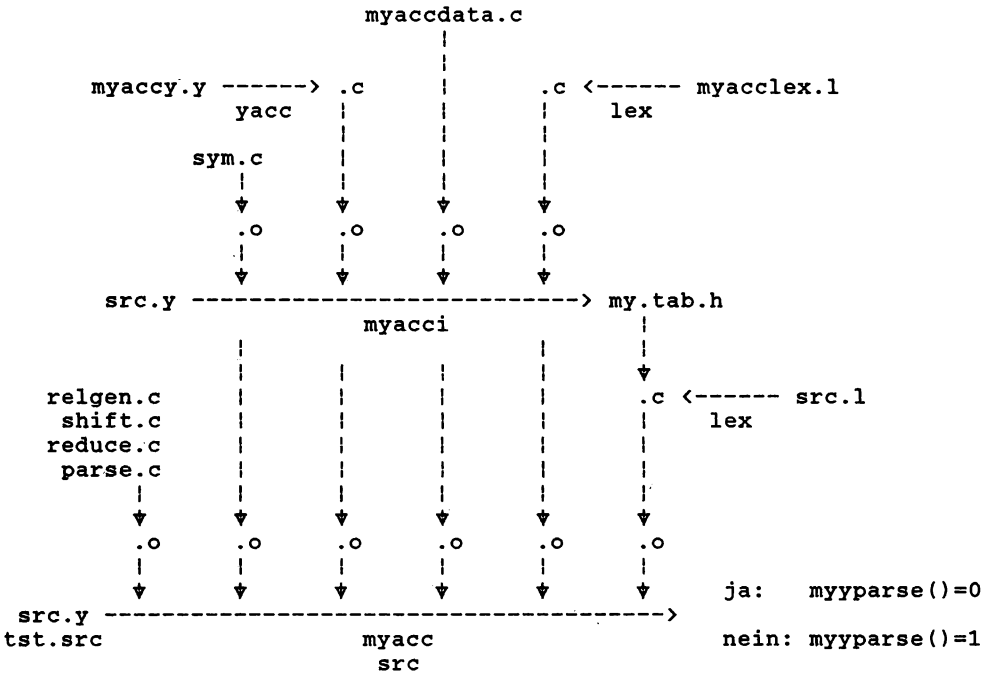



Bild 6.6. Mini-yacc

Die Skizze eines Syntaxprüfers, wie sie als Bild 6.3 angeboten wurde, ist ganz und gar in den letzten waagerechten Pfeil zusammengeflossen. Der Dateiname src steht für die Testsprache lang des Bildes 6.3.

```
5% cat tstif tsterr tstprint
if alpha < beta then print gamma fi
if a < then print b
print alpha
6% make tst
        myacc < loe.y tstif tsterr tstprint > loe.res
        cat loe.res
yyparse()=0
Quelltext: tstif
myyparse()=0
Quelltext: tsterr
syntax error
myyparse()=1
Quelltext: tstprint
myyparse()=0
```

Jetzt wechseln wir die Testgrammatik. make bemerkt diesen Wechsel und reagiert richtig, indem es eine neue Analysefunktion myylex() einarbeitet, bevor die Arbeit mit der Testgrammatik pc.y beginnt.

```
7% cat pctst
1+3
a=425
b=(a*2-3)*4
quit
8% make tst TEST=pc TESTtxt=pctst
        lex  -t pc.l | \
           sed -f lexsed.script > myylex.c
        cc -O -c -LARGE myylex.c
myylex.c
        rm myylex.c
        cc -o myacc myaccdata.o myaccy.o myacclex.o sym.o main.o
            parse.o relgen.o shift.o reduce.o myylex.o -ly
        myacc < pc.y pctst > pc.res
        cat pc.res
yyparse()=0
Quelltext: pctst
myyparse()=0
```

Um noch ein weiteres Mal die Schlagkraft von make beobachten zu können, wollen wir am Gebäude unseres »Mini-yacc« ein biβchen rütteln und einige Dateien beseitigen. Sie werden prompt wiederhergestellt, bevor die Testarbeit richtig losgeht.

```
9% rm {,m}[xy].tab.h
10% make tst TESTtxt=tstif
        yacc -d myaccy.y
        rm y.tab.c
        lex  -t myacclex.l >myacclex.c
        cc -O -c -LARGE myacclex.c
myacclex.c
        rm myacclex.c
        cc -o myacci myaccdata.o myaccy.o myacclex.o sym.o
            maini.o -ly
        myacci < loe.y
        lex  -t loe.l | \
           sed -f lexsed.script > myylex.c
        cc -O -c -LARGE myylex.c
myylex.c
        rm myylex.c
        cc -o myacc myaccdata.o myaccy.o myacclex.o sym.o main.o
            parse.o relgen.o shift.o reduce.o myylex.o -ly
        myacc < loe.y tstif > loe.res
        cat loe.res
yyparse()=0
Quelltext: tstif
myyparse()=0
11%
```

## 6.6. Texttransformation mit lex

Im Anhang des DUDEN werden die Vorschriften für den deutschen Schriftsatz festgelegt. Dort kann man lesen: »Im deutschen Satz ist es unzulässig, für die Umlaute Ä, Ö, Ü behelfsweise Ae, Oe, Ue oder gar AE, OE, UE und für die Umlaute ä, ö, ü behelfsweise ae, oe, ue zu setzen. Ausgenommen sind Personen-, Familien- und Vornamen sowie geographische Namen.«

Daher muß ein Filter hergestellt werden, der die tastaturgeprägten Eingabetexte in dudengerechte Satzmuster umformt. Die Aufgabe zerfällt in drei Schritte.

1. Umformung der Behelfsumlaute Ae, ae, Oe, oe, Ue und ue in nroff-String-Variable, die aus den Behelfsumlauten Einzelzeichen, beispielsweise Überdrucke, herstellen.

2. Satzaufbereitung durch nroff.

3. Ersetzen der nroff-Überdrucke durch Zeichen aus dem erweiterten Zeichenvorrat oder durch passende Escape-Sequenzen.

```
% cat upr.l
%{
/*
        Kommandodatei make_upr:
        % lex upr.l
        % cc -o upr -O -LARGE lex.yy.c -ll
        % rm lex.yy.[co]
        % mv upr ~/bin
        % rehash
*/
```

```
# define out output('\\');output('*')
# define inp input();input()
# define inp2 inp;inp
%}
%%
ie/ss           |       /* schliessen */
ei/ss           |       /* heissen */
eu/ss           |       /* Preussen */
[Aa]u/sse       |
[Gg]ro/ss       |
[Ss]to/ss       |
[BFbf]u/ss      |
[Ss]tra/sse     {ECHO;inp;out;output('s');}
[Aa]euss        {out;output(*yytext);
                 output('u');
                 out;output('s');}
[Mm]/aessig     {ECHO;inp2;out;output('a');out;output('s');}
[Vv]erst/oess   |
[Gg]r/oess      {ECHO;inp2;out;output('o');out;output('s');}
[BFbf]/uess     {ECHO;inp2;out;output('u');out;output('s');}
ss/\n           |
ss/[-abdfghlnrstvz !),.?^] {out;output('s');}
sz              {out;output('s');}
aussehen        |
nteressant      |
szei            |       /* Minuszeichen */
szu             |       /* auszugeben */
ungsz           |       /* Ausrufungszeichen */
ungss           |       /* Steuerungsstruktur */
soeben          |
au              |       /* bauen */
eu              |       /* neuer */
ndividuell      |
ventuell        |
[Mm]anuell      |
[Aa]ktuell      |
[Zz]uerst       |
[Ss]equen       |
[Qq]uel         ECHO;
[AOU]E          |
[AaOoUu]e       {out;output(*yytext);}
\<\"            {out;output('{');}
\>\"            {out;output('}');}
\<\=\=          {out;output('<');}
\>\=\=          {out;output('>');}
\^1\/4          {out;output('4');}
\^1\/2          {out;output('2');}
\=\=\=          {out;output('=');}
%%
/* Offen bleiben: Masse
                  Masze gleichermaszen anmaszen besaszen
*/
%
```

Das Programm upr erzeugt beispielsweise aus »fuer« die Zeichenkette »f*ur«. Das Textverarbeitungsprogramm nroff macht daraus »f"\bur«. Das folgende Programm upo ersetzt den Überdruck ü durch das Zeichen ü aus dem erweiterten ASCII-Zeichensatz, so daß die Zeichenkette »für« gedruckt wird. Die Programme upr und upo wurden bei der Aufbereitung des Manuskripttextes für dieses Buch ständig benutzt.

```
% cat upo.c
#include <stdio.h>
main(){
register int c,d,f;
while ((c=getchar())!=EOF) {
  if (c!='"') putchar(c);
    else if ((d=getchar())==EOF) {putchar(c); return;}
      else if (d!='\b') {putchar(c); putchar(d);}
        else if ((f=getchar())==EOF)
          {putchar(c); putchar(d); return;}
          else switch (f) {
      case 'A': putchar(142); break;   /* oktal 0216 */
      case 'O': putchar(153); break;   /* oktal 0231 */
      case 'U': putchar(154); break;   /* oktal 0232 */
      case 'a': putchar(132); break;   /* oktal 0204 */
      case 'o': putchar(148); break;   /* oktal 0224 */
      case 'u': putchar(129); break;   /* oktal 0201 */
      case 's': putchar(225); break;   /* oktal 0341 */
      case '{': putchar(174); break;   /* oktal 0256 */
      case '|': putchar(175); break;   /* oktal 0257 */
      case '<': putchar(243); break;   /* oktal 0363 */
      case '>': putchar(242); break;   /* oktal 0362 */
      case '=': putchar(240); break;   /* oktal 0360 */
      case '2': putchar(171); break;   /* oktal 0253 */
      case '4': putchar(172); break;   /* oktal 0254 */
      default: putchar(c); putchar(d); putchar(f);
    }
  }
exit(0);
}
%
```

Postfilter ähnlicher Bauart mußten gleichfalls für die Darstellung in der EPSON-Kodierung oder für die Ausgabe auf unterschiedliche Terminals angefertigt werden.

## Anhang 1. Mustertext aus dem Programmierhandbuch Teil 1

Das Kommando script sammelt alle Terminalausgaben in einer Datei. Bei der Gestaltung dieses Buches wurde es häufig dazu verwendet, Kommandoabläufe originalgetreu wiederzugeben.

Im Betriebssystem WEGA auf P8000 sind die Texte des Programmierhandbuchs Teil 1 auch on-line abrufbar. Für das Kommando script müßte

```
% man script
```

eingegeben werden. Auch das Kommando

```
% nroff -man /usr/man/man1/script.1
```

führt zu diesem Ziel.

---

SCRIPT(1) WEGA SCRIPT(1)

**NAME**

script - Anfertigen einer Dateikopie von allen Terminaldialogen.

**SYNTAX**

**script** [ **-a** ] [ **-q** ] [ **-S** shell ] [ *file* ]

**BESCHREIBUNG**

**script** sammelt alle Zeichen, die auf dem Terminal ausgegeben werden, in der Datei *file*. Ist kein Dateiname angegeben, werden die Terminalausgaben in der Datei *typescript* aufbewahrt.

Eingabe von Ctrl-d beendet **script**. Dabei ergeht ein end-of-file an alle gestarteten Prozesse. Dazu wird die Eingabe von Ctrl-d entsprechend vervielfacht.

**OPTIONEN**

**-a** Die Terminalausgaben werden an die Datei *file* oder an *typescript* angehängt. Es wird keine neue Skriptdatei erzeugt.

**-q** Die Ausschriften "Script started" und "script done" werden unterdrückt.

**-S** Ermöglicht die Spezifikation des gewünschten Kommandointerpreters. Standardmäßig wird der in der Datei /etc/passwd für den Nutzer eingetragene Kommandointerpreter zugrunde gelegt.

**BEISPIEL**

```
Script started on Thu Oct 30 13:21:23 1986
$ cat list1
boat
boathouse
boatload
$ rev list1
taob
esuohtaob
daoltaob
$
script done on Thu Oct 30 13:21:55 1986
```

**EINSCHRÄNKUNGEN**

**script** startet für seine Arbeit ein Pipe. Da es keine Möglichkeit gibt, ein end-of-file in ein Pipe zu schreiben, ohne es damit abzuschließen, beendet jedes eingegebene Ctrl-d auch gleichzeitig **script**.

**script** startet eine neue Shell. Diese erhält ihre Standardeingabe von dem gestarteten Pipe und nicht vom Terminal. Dadurch funktionieren die Programme **stty** und **ttyname** nicht. Programme wie **ls** erzeugen kein mehrspaltiges Format, da sie auf ein Pipe schreiben. Im allgemeinen arbeitet **script** nicht richtig, wenn ein Programm Systemmöglichkeiten benutzt, die andere als zeichenorientierte Ein- oder Ausgaben verwenden.

Wenn der Nutzer einen Druckprozeß unterbricht, versucht **script** die Ausgabe zu entfernen, die sich im Pipe zur Sicherung befindet. Gewöhnlich wird dabei das nächste Prompt ebenfalls entfernt.

## Anhang 2. Zeichenersetzung bei einigen UNIX-Kommandos

Reguläre Ausdrücke der einfachen Art oder erweiterte reguläre Ausdrücke bewähren sich in vielen UNIX-Kommandos zur Beschreibung von Suchmustern oder Ersetzungsaktionen. Mit dieser Tabelle werden einige wichtige Operationen mit regulären Ausdrücken zusammengestellt und ihre Anwendbarkeit in verschiedenen Kommandos aufgelistet.

| Aktion | csh | sh | ed<br>sed | ex<br>vi | grep | egrep<br>lex<br>awk |
|---|---|---|---|---|---|---|
| verketten | --- immer erreichbar durch Aneinanderreihen --- | | | | | |
| vervielfachen | {,} | | | | | |
| Klammerung zur Strukturierung | | | | | | ( ) |
| wahlweise | | | | | | ¦ |
| mehrfach(≥0) | * | * | * | * | * | * |
| mehrfach(≥1) | | | | | | + |
| mehrfach(0 ¦1) | | | | | | ? |
| mehrfach(=m) | | | \{m\} | \{m\} | | |
| mehrfach(≥m) | | | \{m,\} | \{m,\} | | |
| mehrfach(≥m und ≤n) | | | \{m,n\} | \{m,n\} | | |
| ein Zeichen | ? | ? | . | . | . | . |
| Zeichenklasse | [-] | [-] | [-] | [-] | [-] | [-] |
| Komplement | | [!] | [^] | [^] | [^] | [^] |
| Zeilenanfang | | | ^ | ^ | ^ | ^ |
| Zeilenende | | | $ | $ | $ | $ |
| Wortanfang | | | | \< | | |
| Wortende | | | | \> | | |
| Fluchtsymbol | \ | \ | \ | \ | \ | \ |

## Anhang 3. Syntax eines Kommandos für /bin/sh

```
%start command
%token if then else elif fi case esac
       for in do done while until
       '||' '&&' ';;' '<<' '>>'
       word name value  /* syntaktisch nicht aufgelöste
                                   Bausteine von sh-Kommandos */
%%
item:             word
|                 input_output
|                 name '=' value
simple_command:   item
|                 simple_command item
command:          simple_command
|                 '(' command_list ')'
|                 '{' command_list '}'
|                 for name do command_list done
|                 for name in word do command_list done
|                 while command_list do command_list done
|                 until command_list do command_list done
|                 case word in case_part esac
|                 if command_list then command_list else_part fi
pipeline:         command
|                 pipeline     command
andor:            pipeline
|                 andor '&&' pipeline
|                 andor '||' pipeline
command_list:     andor
|                 command_list ';'
|                 command_list '&'
|                 command_list ';' andor
|                 command_list '&' andor
input_output:     '>' file
|                 '<' file
|                 '<<' word
|                 '>>' file
|                 digit '>' file
|                 digit '<' file
|                 digit '>>' file
file:             word
|                 '&' digit
|                 '&' '-'
case_part:        pattern ')' command_list
pattern:          word
|                 pattern '|' word
else_part:        elif command_list then command_list else_part
|                 else command_list
|                 empty
empty:
digit:            '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'
```

## Anhang 4. Interne Suffixabhängigkeiten von make (XENIX286)

Terminalbild der folgenden Kommandos:

```
% sh
$ env - make -pf /dev/null 2> /dev/null \
  | sed '/^$/d' | more
```

Es zeigt vordefinierte Makros, suffixgesteuerte Transformationsregeln und Erzeugungsregeln von Objekten aus suffigierten Quellen. In den Zeilen, die mit #### markiert sind, sind die Optionen -o umstritten. Gegebenenfalls muß man die entsprechenden Transformationsregeln außer Kraft setzen.

```
        GFLAGS =
        GET = get
        ASFLAGS =
        AS = as
        CFLAGS = -O
        CC = cc
        LDFLAGS =
        LD = ld
        LFLAGS =
        LEX = lex
        YFLAGS =
        YACC = yacc
        MAKE = make
        $ = $
        MAKEFLAGS = b
.h~.h:
        $(GET) $(GFLAGS) -p $< > $*.h
.s~.a:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s        ####
        ar rv $@ $*.o
        -rm -f $*.[so]
.c~.a:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) -c $(CFLAGS) $*.c
        ar rv $@ $*.o
        rm -f $*.[co]
.c.a:
        $(CC) -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
.l.c:
        $(LEX) $<
        mv lex.yy.c $@
.y~.c:
        $(GET) $(GFLAGS) -p $< > $*.y
        $(YACC) $(YFLAGS) $*.y
        mv y.tab.c $*.c
        -rm -f $*.y
.y.c:
        $(YACC) $(YFLAGS) $<
        mv y.tab.c $@
```

```
.l~.o:
        $(GET) $(GFLAGS) -p $< > $*.l
        $(LEX) $(LFLAGS) $*.l
        $(CC) $(CFLAGS) -c lex.yy.c
        rm -f lex.yy.c $*.l
        mv lex.yy.o $*.o
.l.o:
        $(LEX) $(LFLAGS) $<
        $(CC) $(CFLAGS) -c lex.yy.c
        rm lex.yy.c
        mv lex.yy.o $@
.y~.o:
        $(GET) $(GFLAGS) -p $< > $*.y
        $(YACC) $(YFLAGS) $*.y
        $(CC) $(CFLAGS) -c y.tab.c
        rm -f y.tab.c $*.y
        mv y.tab.o $*.o
.y.o:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) -c y.tab.c
        rm y.tab.c
        mv y.tab.o $@
.s~.o:
        $(GET) $(GFLAGS) -p $< > $*.s
        $(AS) $(ASFLAGS) -o $*.o $*.s        ####
        -rm -f $*.s
.s.o:
        $(AS) $(ASFLAGS) -o $@ $<            ####
.c~.c:
        $(GET) $(GFLAGS) -p $< > $*.c
.c~.o:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
.c.o:
        $(CC) $(CFLAGS) -c $<
.sh~:
        $(GET) $(GFLAGS) -p $< > $*.sh
        cp $*.sh $*
        -rm -f $*.sh
.sh:
        cp $< $@
.c~:
        $(GET) $(GFLAGS) -p $< > $*.c
        $(CC) $(CFLAGS) $(LDFLAGS) $*.c -o $*
        -rm -f $*.c $*.o
.c:
        $(CC) $(CFLAGS) $(LDFLAGS) $< -o $@
        -rm -f $*.o
.SUFFIXES: .o .c .c~ .y .y~ .l .l~ .s .s~ .sh .sh~ .h .h~
```

## Anhang 5. Skriptdatei für eine modifizierte Spurverfolgung bei yacc

Die Datei yaccsed.script ist eine Skriptdatei für den Stream-Editor sed, der im Abschnitt 3.2. ausführlich beschrieben worden ist. Mit ihr kann das Resultat der Arbeit von yacc, die Datei y.tab.c, so geändert werden, daß bessere Fehlermeldungen entstehen und eine bessere Spurverfolgung möglich wird. Die Skriptdatei orientiert sich an Kommentaren der Datei /usr/lib/yaccpar, die bei der Arbeit von yacc als Teil von y.tab.c einbezogen wird. Die Skriptdatei wurde an die yacc-Version von UNIX System V angepaßt. Für frühere yacc-Versionen kann die prinzipielle Vorgehensweise aufgegriffen werden. Allerdings lassen sich Verbindungen zwischen Analysezuständen und bestimmenden Symbolen nicht so einfach wie hier herstellen.

Mit der Skriptdatei, sei sie auch im Verzeichnis /usr/lib erreichbar, ist folgender Ablauf ratsam:

```
yacc -dt source.y
sed -f /usr/lib/yaccsed.script y.tab.c > source.c
cc -o source $(CFLAGS) source.c $(LDFLAGS)
```

Anmerkung: Im folgenden Skripttext steht <tab> für das Tabulatorzeichen \t≡011.

yaccsed.script

```
/^typedef struct {.*} yytoktype;/{
   n
   N
   N
   N
   N
   h
   d
    }
/"-unknown-".*\/\* ends search \*\//{
   N
   p
   g
    }
/^char \* yyreds\[] =/,/^};/s/\\/&&/g
/yaccpar\.src/i\
#include <stdio.h>\
#include <ctype.h>\
static char *yydisplay(ch)\
register int ch; {  /* Verdeutlichung der terminalen Symbole */\
register yytoktype *p;  /*nach Schreiner*/\
static char buf[15];\
if (ch==0) return "[end of file]";\
if (ch<0) return "[none]";\
for (p=yytoks;p->t_val>=0;++p) if (p->t_val==ch) return p->t_name;\
switch(ch) {\
  case YYERRCODE: return "[error]";\
  case '\\b': return "'\\\\b'";\
  case '\\f': return "'\\\\f'";\
  case '\\n': return "'\\\\n'";\
  case '\\r': return "'\\\\r'";\
```

```
  case '\\t': return "'\\\\t'";\
  case '\\v': return "'\\\\v'";\
  }\
if (isascii(ch) && isprint(ch)) sprintf(buf,"'%c'",ch);\
else if (ch<256) sprintf(buf,"char %4.3o",ch);\
else             sprintf(buf,"token %d",ch);\
return buf;\
}\
yyerror(s)  /* Verbesserte Fehlermeldungen */\
register char *s; {  /*nach Schreiner*/\
extern int yynerrs;\
static int list = 0;\
if (!list) {\
  printf("[error %d] ", yynerrs+1); yywhere();\
  if (list=s[strlen(s)-1]==':') fputs(s,stdout);\
    else puts(s);\
  } else if (*s!='\\n') putchar(' '),fputs(s,stdout);\
    else putchar('\\n'),list=0;\
}\
extern char yytext[];\
extern int yyleng,yylineno;\
yywhere() {  /* Fehlerplatzierung durch Bezug auf yytext */\
register int colon=0;  /*nach Schreiner*/\
if (yylineno>0) {\
  if (colon) fputs(", ",stdout);\
   printf("line %d",yylineno-((*yytext=='\\n')||(!*yytext)));\
   colon=1;\
  }\
switch (*yytext) {\
  case '\\0':\
  case '\\n': break;\
  default: if (colon) putchar(' ');\
    printf("near \\"%.*s\\"",yyleng>20 ? 20 :\
           yytext[yyleng-1]=='\\n' ? yyleng-1 : yyleng,yytext);\
    colon=1;\
  }\
if (colon) fputs(": ",stdout);\
}\
yyputsy(ind)  /* Zuordnung Zustand-Symbol im Analysekeller */\
int ind; {\
int i,j; register char *cp;\
if ((i=yychk[ind])>=0) printf("%s ",yydisplay(i));\
  else if (-i<=YYNPROD) { char c;\
    j=0; while (yyr1[++j]!=(-i)); cp=yyreds[j];\
    do fputc((c=(*(cp++))),stdout); while (c!=' ');\
    } else fputs("-- ",stdout);\
}\
#define yydbg(x) ((yydebug^yydebug>>x)&1)
/^#define YYRECOVERING/a\
#ifndef YYDEBST\
#       define YYDEBST  8\
#endif
/^int yydebug.*;/{
   s//int yydebug = 0;/
   s/to 1/> 0/
    }
/\*\* Initialize externals/{
   N
   a\
#if YYDEBUG\
if ( yydebug ) {\
```

```
  printf("yydebug = %d\\n",yydebug);\
  printf("debug-mode: States, Stack: %3d\\n",yydbg(1));\
  if ( yydbg(1) )\
    printf("max. number of stack-items:%3d\\n",YYDEBST);\
  printf("            Received token:%3d\\n",yydbg(2));\
  printf("            Pops, uncovers:%3d\\n",yydbg(3));\
  printf("            Discards token:%3d\\n",yydbg(4));\
  printf("            Reduce by rule:%3d\\n",yydbg(5));\
  }\
#endif /* YYDEBUG */
    }
/\*\* we have a new state -/{
   N
   a\
#if YYDEBUG\
if ( yydbg(1) ) {\
  register int *yy_i,i;\
  printf("State %d", yy_state);\
  if (yychar>=0) printf(", token %s",yydisplay(yychar));\
  putchar('\\n');\
  i=0; yy_i=yy_ps;\
  while ((yy_i>yys) && (i<YYDEBST)) {i++;yy_i--;}\
  if (i) {\
  printf("Stack: ");\
  for(;yy_i<yy_ps;) yyputsy(*(++yy_i));\
  putchar('\\n');\
  }\
}\
#endif /* YYDEBUG */
    }
/\/\* simple state \*\//a\
#if YYDEBUG\
yytmp=yychar<0;\
#endif /* YYDEBUG */
/<tab><tab><tab>yychar = 0;<tab><tab>\/\* reached EOF \*\//a\
#if YYDEBUG\
if ( yydbg(2) && yytmp )\
  printf("Received token %s\\n",yydisplay(yychar));\
#endif /* YYDEBUG */
/yydef\[ yy_state ] ) == -2/{
   N
   a\
#if YYDEBUG\
yytmp=yychar<0;\
#endif /* YYDEBUG */
    }
/case 0:<tab><tab>\/\* new error \*\//a\
if ((yy_n=yypact[yy_state])>YYFLAG && (yy_n<YYLAST)) {\
  register int x,y=0;\
  for (x=yy_n>0 ? yy_n : 0; x<YYLAST; ++x)\
    if ((yychk[yyact[x]]==x-yy_n) && (x-yy_n!=YYERRCODE)) {\
      if (!y) {\
        yyerror("syntax error, expecting:"); y=1;\
        }\
      yyerror(yydisplay(x-yy_n));\
      }\
  if (y) yyerror("\\n");\
    else yyerror("syntax error");\
  }\
  else
/yynerrs++;/{
```

```
   h
   d
    }
/skip_init:/{
   p
   g
    }
/\*\* "error", pop stack/{
   N
   a\
#if YYDEBUG\
if ( yydbg(3) ) {\
  printf("Error recovery pops state %d",*yy_ps);\
  if (yy_ps>yys) printf(", uncovers state %d\\n", yy_ps[-1]);\
  else putchar('\\n');\
  printf("                pops symbol "); yyputsy(*yy_ps);\
  if (yy_ps>yys) {\
    printf(", uncovers symbol "); yyputsy(yy_ps[-1]);\
    } else printf(", stack is empty");\
  putchar('\\n');\
  }\
#endif /* YYDEBUG */
    }
/\/\* no shift yet; eat a token \*\//a\
#if YYDEBUG\
if ( yydbg(4) )\
  printf("Error recovery discards token %s\\n",yydisplay(yychar));\
#endif /* YYDEBUG */
/\*\* put stack tops, etc\. so /{
   N
   a\
#if YYDEBUG\
if ( yydbg(5) )\
  printf( "Reduce by (%d) \\"%s\\"\\n",yy_n,yyreds[yy_n]);\
#endif
    }
/^#if YYDEBUG/,/^#endif/d
```

## Anhang 6. Quelldateien für Mini-yacc

makefile

```
TEST=loe
TESTtxt=tstif tsterr tstprint
SHELL=/bin/sh
YFLAGS= -dt
LDFLAGS= -ly
FILES=makefile lexsed.script myacc.h myaccdata.c myaccy.y myacclex.l\
main.c maini.c mainn.c parse.c sym.c relgen.c shift.c reduce.c
SFILES=loe.y loe.l pc.y pc.l
OBJS=myaccdata.o myaccy.o myacclex.o sym.o
OBJSo=parse.o relgen.o shift.o reduce.o
OBJSc=$(OBJS) main.o $(OBJSo) myylex.o

.l.o:           # Ausschalten von Standardabläufen
.l.c:           # dto.
.y.o:           # dto.
        $(YACC) $(YFLAGS) $<
        sed -f /usr/lib/yaccsed.script y.tab.c > $*.c
        $(CC) $(CFLAGS) -c $*.c
        rm -f y.tab.c $*.c
                # Bei WEGA muß die Regel y.o.: wegbleiben.
.PRECIOUS: pr tsttxt

help:   @echo           # TEST=$(TEST)'
        @echo           # TESTtxt=$(TESTtxt)'
        @egrep '^[^:;=.]*::?[ <tab>]*#' [mM]akefile

myacci: # Installation von myacci,
myacci: # Initialisierung myacci < src.y erzeugt my.tab.h
myacci: $(OBJS) maini.o
        $(CC) -o $@ $(OBJS) maini.o $(LDFLAGS)

myacc: $(OBJSc)
        $(CC) -o $@ $(OBJSc) $(LDFLAGS)

myaccc: # Installation von myacc für eine neue Grammatik
myaccc: # rm my.tab.h; make myacc
myaccc:  ; rm -f my.tab.h
        make myacc TEST=$(TEST)

myaccn: # myaccn < src.y, yacc-artige Numerierung der Symbole
myaccn: $(OBJS) mainn.o
        $(CC) -o $@ $(OBJS) mainn.o $(LDFLAGS)

tst:    # myacc < $(TEST).y $(TESTtxt) > $(TEST).res
tst: myacc $(TESTtxt)
        myacc < $(TEST).y $(TESTtxt) > $(TEST).res
        cat $(TEST).res

$(OBJS): myacc.h
main.o maini.o mainn.o $(OBJSo): myacc.h
```

```
myacclex.o: myacclex.l x.tab.h
        $(LEX) $(LFLAGS) -t myacclex.l > myacclex.c
        $(CC) $(CFLAGS) -c -LARGE myacclex.c
                # Bei WEGA entfällt die Option -LARGE.
        rm myacclex.c

x.tab.h: y.tab.h      -@cmp -s $@ $? || cp $? $@
y.tab.h: myaccy.y
        $(YACC) -d myaccy.y
        rm -f y.tab.c

myylex.o:: tsttxt
myylex.o:: $(TEST).l mx.tab.h
        $(LEX) $(LFLAGS) -t $(TEST).l | \
           sed -f lexsed.script > myylex.c
        $(CC) $(CFLAGS) -c -LARGE myylex.c
                # Bei WEGA entfällt die Option -LARGE.
        rm myylex.c

mx.tab.h: my.tab.h  ;  -@cmp -s $@ $? || cp $? $@
my.tab.h: myacci $(TEST).y
        myacci < $(TEST).y

tsttxt:: myacci $(TEST).y
        myacci < $(TEST).y
        @echo $(TEST) > tsttxt

tsttxt:: txt
        -@cmp -s tsttxt txt || \
           myacci < $(TEST).y
        @mv txt tsttxt

txt:    @echo $(TEST) > txt

rm:     # rm *.o; rm *.tab.h
        rm -f $(OBJSc) main[in].o
        rm [xy].tab.h m[xy].tab.h

pr:     # Drucke geänderte Dateien.
pr: $(FILES) $(SFILES)
        @p $?
        @touch pr
```

---

lexsed.script

---

```
/yy/s//myy/g
/myyout/s//yyout/g
/yyoutput/s//myyoutput/g
/FILE/s/=//g
/FILE/s/{stdin}//
/FILE/s/{stdout}//
```

myacc.h

```
#include <stdio.h>
#define RNO_MAX 50
#define RLGTH_MAX 20
#define NVOC_MAX 30
#define TOKENNO_MAX 300
#define NST_MAX 30
#define STST_MAX 200
#define STACKDEPTH_MAX 20
#define REL_MAX 100
typedef struct reltup {
       short a,e; } reltup;
typedef struct rppaar {
       short r,p; } rppaar;
typedef struct nametype {
       char *name;
       short nbr;
       struct nametype *next; } nametype;
extern short     Rno,Rlgth[],Reg[][RLGTH_MAX];
extern short     Nontno,Tokenno;
extern short     Declpart,Startid;
extern nametype *Nameptr;
extern short     Asciic[];
extern short     Nvoc,Voc[];
extern rppaar    Statest[];
extern short     Statestanf,Statestend;
extern short     Nstates,Nstatebase[];
extern short     Stateind[],Shiftaction[];
extern short     Lrtab[][NVOC_MAX];
extern reltup    Inirel[],Rrel[],Sucrel[];
extern short     Ninirel,Nrrel,Nsucrel;
extern FILE     *myyin, *erfptr, *tabhptr;
```

myaccdata.c

```
#include "myacc.h"
short     Rno,Rlgth[RNO_MAX],Reg[RNO_MAX][RLGTH_MAX];
short     Nontno=0,Tokenno=256;
short     Declpart=1,Startid=0;
nametype *Nameptr;
short     Asciic[128];
short     Nvoc,Voc[NVOC_MAX];
rppaar    Statest[STST_MAX];
short     Statestanf,Statestend;
short     Nstates,Nstatebase[NST_MAX];
short     Stateind[NST_MAX],Shiftaction[NST_MAX];
short     Lrtab[NST_MAX][NVOC_MAX];
reltup    Inirel[REL_MAX],Rrel[REL_MAX],Sucrel[REL_MAX];
short     Ninirel=0,Nrrel=0,Nsucrel=0;
FILE     *erfptr={stdout};
FILE     *tabhptr;
```

myaccy.y

```
%{   /*grammar for the input to yacc*/
#include "myacc.h"
short rl,rlft;
%}

/*basic entities*/

%token IDENTIFIER   /*includes identifiers and literals*/
%token C_IDENTIFIER /*identifier followed by colon*/
%token NUMBER       /*[0-9]+ */

/*reserved words: %type=>TYPE, %left=>LEFT, etc.*/

%token LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION
%token MARK  /*the %% mark*/
%token LCURL /*the %{ mark*/
%token RCURL /*the %} mark*/

/*ascii character stand for themselves*/

%start spec

%%

spec : defs MARK {Declpart=0;} rules tail
tail : MARK {eat_up();} /*Eat up the rest of file*/
     | /*empty : the second MARK is otpional*/
defs : /*empty*/
     | defs def
def  : START {Startid=1;} IDENTIFIER
/*   | UNION {Copy union definition to output} */
/*   | LCURL {Copy C code to output file} RCURL */
     | rword tag nlist
rword: TOKEN
     | LEFT
     | RIGHT
     | NONASSOC
/*   | TYPE */
tag  : /*empty*/
/*   | '<' IDENTIFIER '>' */
nlist: nmno
     | nlist nmno
     | nlist ',' nmno
nmno : IDENTIFIER
     | IDENTIFIER NUMBER {if (yylval>=TOKENNO_MAX) {
        fprintf(erfptr,"Tokennummer zu gross, TOKENNO_MAX=%d\n",
                TOKENNO_MAX); YYABORT;}
        Nameptr->nbr=Tokenno=yylval;}


/*rules section*/

rules: C_IDENTIFIER {Rno=1;} rbody prec
       {if (!$3) { epsabb(); YYABORT;}
        Rlgth[Rno]=$3;Reg[Rno][0]=rlft=$1;}
       rules rule
```

```
rule   C_IDENTIFIER {if (rnoincr()) YYABORT;} rbody prec
       {if (!$3) { epsabb(); YYABORT;}
        Rlgth[Rno]=$3;Reg[Rno][0]=rlft=$1;}
     | '|' {if (rnoincr()) YYABORT;} rbody prec
       {if (!$3) { epsabb(); YYABORT;}
        Rlgth[Rno]=$3;Reg[Rno][0]=rlft;}
rbody: /*empty*/ {rl=0;$$=rl;}
     | rbody IDENTIFIER
       {if (rlgthincr()) YYABORT;Reg[Rno][rl]=$2;$$=rl;}
       rbody act
act    '{' {copy_act();} '}'
prec   /*empty*/
/*     PREC IDENTIFIER
       PREC IDENTIFIER act */
       prec ';'
%%
eat_up(){ int c;
while ((c=getchar())!=EOF); yyunput(c);
}
copy_act(){ int c;
while ((c=yyinput())!='}'); yyunput(c);
}
epsabb() {
fprintf(erfptr,"Regel erzeugt EPSILON!\n");}
rnoincr() {
short rc=0;
if (++Rno>=RNO_MAX) { rc=1;
  fprintf(erfptr,"Zu viele Regeln, RNO_MAX=%d\n",RNO_MAX);}
return rc;
}
rlgthincr() {
short rc=0;
if (++rl>= RLGTH_MAX) { rc=1;
  fprintf(erfptr,"Zu lange Regel, RLGTH_MAX=%d\n",RLGTH_MAX);}
return rc;
}
```

myacclex.l

```
%{
#include "myacc.h"
#include "y.tab.h"
#define YYERRCODE 256
extern int yylval;
short lexerr;
nametype *look(), *install();
%}
id              [A-Za-z][0-9A-Z_a-z]*
%start cid
%%
"/*"[^*]*"*/"       /* Kommentare eliminieren */
"%{"[^%]*"%}" ;     /* Einschlüsse streichen */
{id}/[ \t]*:   {BEGIN cid;nameincl();
                if (!(Nameptr->nbr))
                  Nameptr->nbr=(-(++Nontno));
                yylval=Nameptr->nbr;
                return C_IDENTIFIER;}
<cid>[ \t]*:   {BEGIN 0;}
{id}           {ident();return lexerr ? YYERRCODE : IDENTIFIER;}
[0-9]+         {sscanf(yytext,"%d",&yylval);return NUMBER;}
"%%"           return MARK;
"%{"           return LCURL;
"%}"           return RCURL;
"%start"       return START;
"%token"       return TOKEN;
"%left"        return LEFT;
"%right"       return RIGHT;
"%nonassoc"    return NONASSOC;
\'\\n\'        {yylval='\n';Asciic[yylval]=1;return IDENTIFIER;}
\'\\t\'        {yylval='\t';Asciic[yylval]=1;return IDENTIFIER;}
\'[^']+\'      {if (yyleng==3) {
                  yylval=yytext[1];Asciic[yylval]=1;
                  } else {short i;
                  yytext[yyleng-1]='\0';
                  for (i=0;i<=yyleng-2;i++)
                    yytext[i]=yytext[i+1];
                  ident();}
                return lexerr ? YYERRCODE   IDENTIFIER;}
[ \t\n]*       ;
.              return *yytext;
%%
yywrap(){return 1;}
nameincl() {
  nametype *nptr;
  if ((nptr=look(yytext))==0)
    nptr=install(yytext);
  Nameptr=nptr;
}
ident() {
  lexerr=0; nameincl();
  if (Declpart) { /* Deklarationsteil */
    if (Startid) {
        Nameptr->nbr=(-(++Nontno));Startid=0;
      } else {
      if (++Tokenno>=TOKENNO_MAX) { lexerr=1;
```

```
        fprintf(erfptr,"Tokennummer zu gross, TOKENNO_MAX=%d\n"
                ,TOKENNO_MAX);return;}
      Nameptr->nbr=(Tokenno);
      fprintf(tabhptr,"#define token%s %d\n",
                      Nameptr->name,Nameptr->nbr);
      }
    } else /* Regelteil */
    if (!(Nameptr->nbr)) Nameptr->nbr=(-(++Nontno));
  yylval=Nameptr->nbr;
}
```

main.c

```
#include "myacc.h"
main(argc,argv)
int argc; char **argv; {
short argi=0;
tabhptr=fopen("my.tab.h","w");
printf("yyparse()=%d\n",yyparse());
vocgen();
genini(Inirel,&Ninirel);
gensuc(Sucrel,Rrel,Inirel,&Nsucrel,&Nrrel,&Ninirel);
Statestanf=Statestend=1;
shiftact();
Lrtab[1][1]=NST_MAX;
reduceact();
while (++argi<argc) {
  printf("Quelltext: %s\n",argv[argi]);
  myyin=fopen(argv[argi],"r");
  printf("myyparse()=%d\n",myyparse());
  fclose(myyin);
  }
exit(0);
}
```

maini.c

```
#include "myacc.h"
main() {
tabhptr=fopen("my.tab.h","w");
exit(yyparse());
}
```

mainn.c

```
#include "myacc.h"
main() {
tabhptr=fopen("my.tab.h","w");
printf("myaccn: yyparse()=%d\n",yyparse());
symprot();
exit(0);
}
```

sym.c

```
#include "myacc.h"
static nametype *namelist=0;
              /*Wörterbuch als rückwärts verkettete Liste*/
nametype *look(s)
char *s; {    /*Suche nach einer Bezeichnung*/
nametype *nptr;
for (nptr=namelist; nptr!=(nametype *)0; nptr=nptr->next)
  if (strcmp(nptr->name,s)==0) return nptr;
return 0;
}
nametype *install(s)
char *s; {    /*Aufnahme  einer neuen Bezeichnung*/
nametype *nptr;
char *malloc();
nptr=(nametype *)malloc(sizeof(nametype));
nptr->name=malloc(strlen(s)+1);
strcpy(nptr->name,s);
nptr->nbr=0;
if (strcmp(s,"error")==0) nptr->nbr=256;
nptr->next=namelist;
namelist=nptr;return nptr;
}
symprot() {  /*Drucken des Wörterbuches*/
nametype *nptr;
for (nptr=namelist; nptr!=(nametype *)0; nptr=nptr->next)
  printf("nbr= %3d name= \"%s\"\n",nptr->nbr,nptr->name);
}
vocgen() {    /*Konstruktion und Aufzählung des Grammatikvokabulars*/
short i,ii,j;                 /*1. Nichtterminale, 2. ASCII-Zeichen,*/
nametype *nptr;               /*3. deklarierte Terminale.*/
short imax,vocscr[NVOC_MAX];
short min,imin;
for (j=1;j<=Nontno;j++) Voc[j]= -j;
for (i=0;i<=127;i++) if (Asciic[i]) Voc[j++]=i;
for (i=1,nptr=namelist;nptr!=(nametype *)0;
              nptr=nptr->next)
  if (nptr->nbr>0) vocscr[i++]=nptr->nbr;
imax=i-1;     /*Mit gespreizten Fingern wird das Vokabular*/
for (i=1;i<=imax;i++) {          /*geordnet und numeriert.*/
  min=TOKENNO_MAX;
  for (ii=1;ii<=imax;ii++)
    if (vocscr[ii] && (vocscr[ii]<min)) {
      min=vocscr[ii];imin=ii;}
  vocscr[imin]=0;Voc[j++]=min;}
Nvoc=j-1;
}
syno(sn)
short sn; {  /*Rekonstruktion der "yacc"-Numerierung*/
short v=0;   /*aus den Positionen im Vokabular*/
while (v<=Nvoc)
  if (Voc[++v]==sn) break;
if (v>Nvoc) v=0;  /*unbekanntes Terminalsymbol*/
return v;
}
```

parse.c

```
#include "myacc.h"
myyparse() {
short myys[STACKDEPTH_MAX];
short *sptr,*myyps,act,actv,actsy,actsy_old,myystate=1,ende=0;
short shift_aft_red=0;

myyps=myys;*myyps=1;
actsy=myylex();
if (!actsy||(actsy==EOF)) ende=1;
while (!ende) {
  actv=syno(actsy);
  act=Lrtab[myystate][actv];

/*
printf("actsy=%3d actv=%3d ",actsy,actv);
if (act==NST_MAX) fputs("accept",stdout);
  else if (!act) fputs("error!",stdout);
    else {
    if (act>0) fputs(" shift",stdout);
    if (act<0) fputs("reduce",stdout);
    printf("%3d  STACK:",act);
    }
if ((act<NST_MAX)&&act)
  for (sptr=myys;sptr<=myyps;) printf(" %d",*sptr++);
putchar('\n');
if (act<0) printf("          Regel%3d:\n",-act);
*/

  switch (act) {
      case 0: myyerror("syntax error");  /*error*/
              return 1;
      case NST_MAX: return 0;            /*accept*/
     default: if (act>0) {               /*shift*/
                myystate=act;
                if (++myyps>=&myys[STACKDEPTH_MAX]) {
                   myyerror("myacc: stack overflow");
                   return 1;
                   }
                *myyps=myystate;
                if (!shift_aft_red) {
                   actsy=myylex();
                  } else {
                  actsy=actsy_old; shift_aft_red=0;
                  }
                } else {                 /*reduce*/
                shift_aft_red=1;
                actsy_old=actsy;
                myystate=(*(myyps-=Rlgth[-act]));
                actsy=Reg[-act][0];
                }
      }
    }
  }
  myyerror(s)
  char *s; { puts(s); }
```

relgen.c

```
#include "myacc.h"

genini(irel,nirel)
reltup *irel;              /*Bildung der Anfangsrelation*/
short *nirel; {
short i,j,ri;
reltup p;

*nirel=0;
for (ri=1;ri<=Rno;ri++) {  /*Für alle Regeln (l,r1 r2     rn)*/
  p.a=Reg[ri][0];          /*Setze p = (l,r1)*/
  p.e=Reg[ri][1];
  relappend(irel,nirel,p);/*Hinzunahme des Elements p*/
  }
transclos(irel,nirel);     /*Konstruktion der transitiven Hülle*/
for (j=0,i=1;i<=(*nirel);i++)
  if (irel[i].e>0)         /*Terminales Symbol*/
    irel[++j]=irel[i];
*nirel=j;      /*Weglassen der Paare (a,e) mit nichtterminalem e*/
}

gensuc(srel,rrel,irel,nsrel,nrrel,nirel)
reltup *srel,*rrel,*irel;  /*Bildung der Nachfolgerelation*/
short *nsrel,*nrrel,*nirel; {
reltup p;
short pend;
short i,j,jmax;
short ri,sp;

   /*Zuerst: Bildung der Relation "rechtes Zwischensymbol"*/
for (ri=1;ri<=Rno;ri++) {  /*Für alle Regeln (l,r1 r2     rn)*/
  p.e=Reg[ri][0];          /*Setze p = (l,rn)*/
  p.a=Reg[ri][Rlgth[ri]];
  if (p.a<-1)  /*Zwischensymbol*/
    relappend(rrel,nrrel,p); /*Hinzunahme des Elements p*/
  }              /*zur Relation "rechtes Zwischensymbol"*/

transclos(rrel,nrrel);     /*Konstruktion der transitiven Hülle*/

for (ri=1;ri<=Rno;ri++) {
  for (sp=1;sp<=Rlgth[ri]-1;sp++) { /*Für alle Regeln*/
    p.a=Reg[ri][sp];               /*(l,r1 ... r r' ... rn)  */
    p.e=Reg[ri][sp+1];             /*Setze p = (r,r') */
    if (p.a<-1)              /*Falls r ein Zwischensymbol ist:*/
      if (p.e<-1) {              /*r' ist Zwischensymbol: */
        pend=p.e;
        for (i=1;i<=(*nirel);i++)/*Nimm für alle Paare (r',t),*/
          if (pend==irel[i].a) { /*die zur Anfangsrelation gehören,*/
            p.e=irel[i].e;       /*das Paar (r,t)*/
            relappend(srel,nsrel,p);/*zur Nachfolgerelation hinzu.*/
            }
        } else                   /*r' ist nicht Zwischensymbol: */
        relappend(srel,nsrel,p);
    }          /*Hinzunahme von (r,r') zur Nachfolgerelation*/
  }
```

```
jmax=(*nsrel);
for (i=1;i<=(*nrrel);i++)
  for (j=1;j<=jmax;j++)
    if (rrel[i].e==srel[j].a) {
      p.a=rrel[i].a; /*Falls re rechtes Zwischensymbol von ra, und*/
      p.e=srel[j].e; /*(re,t) bereits zur Nachfolgerelation gehört,*/
      relappend(srel,nsrel,p); /*so nimm auch (ra,t) hinzu.*/
      }
}

transclos(rel,nrel)
reltup *rel;        /*Bildung der transitiven Hülle der Relation rel*/
short *nrel; {
short i,j,imin=0,imax=0;
short elneu=1;      /*Neues Element gefunden*/
reltup p;

while (elneu) {
  elneu=0;imin=imax+1;imax=(*nrel);
  for (i=imin;i<=imax;i++)
    for (j=1;j<=(*nrel);j++)
      if (rel[i].e==rel[j].a) {
        p.a=rel[i].a;p.e=rel[j].e;
        if (relappend(rel,nrel,p)) elneu=1;
        }
  }
}

relappend(rel,nrel,tupel)
reltup *rel,tupel; /*Hinzunahme des Paares tupel zur Relation rel*/
short *nrel; {
short i=1;
short elneu=1;      /*ELement ist neu*/

while (elneu&&(i<=(*nrel))) {
  elneu=((tupel.a!=rel[i].a)||(tupel.e!=rel[i].e));i++;}
if (elneu) {
  if ((++*nrel)>=REL_MAX)  {
    fprintf(erfptr,"Zu viele Relationen, REL_MAX=%d\n",REL_MAX);
    exit(255);}
  rel[*nrel]=tupel;
  }
return elneu;
}
```

shift.c

```
#include "myacc.h"

shiftact() {  /*Berechnung der shift-Aktionen*/
rppaar stel;
short stati,statianf,statiend;
short statkanf,nstatkb;
short statjj,statjjanf,statjjbend;
short i,j,jj,v,sy,ri,sp,ungl;

Nstates=stati=statianf=1;Statestend=0;
Stateind[Nstates]=statianf;
stel.r=1;stel.p=0;
stateappend(statianf,stel);
Nstatebase[stati]=1;
stateclos(statianf);  /*Bilde die Hülle zum ersten Zustandselement*/
Shiftaction[stati]=0;
while (stati<=Nstates) { /*Für alle bisher erkannten Zustände:*/
  for (v=2;v<=Nvoc;v++) {  /*Für alle Symbole sy:*/
    sy=Voc[v];
    statianf=Stateind[stati];
    statiend=stati==Nstates ? Statestend : Stateind[stati+1]-1;
    statkanf=Statestend+1;            /*Versuche die Basismenge*/
    nstatkb=0;  /*eines Nachfolgezustands statk zu konstruieren*/
    for (i=statianf;i<=statiend;i++) { /*Für alle Zustandselemente*/
      ri=Statest[i].r;sp=Statest[i].p; /*(ri,sp) im Zustand stati:*/
      if ((sp+1)<=Rigth[ri])
        if (sy==Reg[ri][sp+1]) { /*Falls sy in der Regel ri*/
          stel.r=ri;              /*auf Platz sp+1 steht:*/
          stel.p=sp+1;            /*Setze stel = (ri,sp+1)*/
          if (stateappend(statkanf,stel)) nstatkb++;
          }  /*Hinzunahme von stel zur Zustandsbasis von statk*/
      }
      if (nstatkb) {/*Es wurde eine nichtleere Basismenge gebildet.*/
        ungl=1; /*Ist sie von denen bekannter Zustände verschieden?*/
        for (jj=2;(jj<=Nstates)&&ungl;jj++) {
          statjj=jj; /*Vom Laufindex entkoppeln*/
          if (Nstatebase[statjj]==nstatkb) {
            statjjanf=Stateind[statjj];
            statjjbend=statjjanf+Nstatebase[statjj]-1;
            for (ungl=0,i=statjjanf;(i<=statjjbend)&&!ungl;i++)
              for (ungl=1,j=statkanf;(j<=Statestend)&&ungl;j++)
                ungl=((Statest[i].r!=Statest[j].r)
                      ||(Statest[i].p!=Statest[j].p));
          }
        }
      if (ungl) { /*Es wurde eine neue Basismenge gebildet.*/
        if (++Nstates>=NST_MAX) {
         fprintf(erfptr,"Zu viele Zustände, NST_MAX=%d\n",NST_MAX);
          exit(255);}
        Nstatebase[Nstates]=nstatkb;
        Stateind[Nstates]=statkanf; /*Als neuen Zustand aufnehmen*/
        Shiftaction[Nstates]=0;
        Lrtab[stati][v]=Nstates; /*shift-Aktion zum neuen Zustand*/
        Shiftaction[stati]=1;
        stateclos(statkanf);
        } else {/*Die Menge gleicht der Basis von Zustand statjj.*/
```

```
          Lrtab[stati][v]=statjj; /*shift-Aktion dorthin*/
          Shiftaction[stati]=1;
          Statestend=statkanf-1; /*Wegwerfen von statk*/
          }
        }
    }
    stati++;
  }
}

stateclos(sta)
short sta; {    /*Bilden der Zustandshülle*/
short sy;
rppaar stel;    /*state_element*/
short stelgf=1; /*neues state_element gefunden*/
short imin,imax;
short i,rr,ri,sp;

imax=sta-1;
while (stelgf) {
  stelgf=0;
  imin=imax+1;imax=Statestend;
  for (i=imin;i<=imax;i++) { /*Für alle Zustandselemente (ri,sp):*/
    ri=Statest[i].r;         /*Sei ri=(li,r1 ... r r' ... rn)*/
    sp=Statest[i].p;         /*Auf Position sp steht Symbol r*/
    if (sp+1<=Rlgth[ri]) {
      sy=Reg[ri][sp+1];
      if (sy<-1) { /*sy ist Zwischensymbol. Falls es eine Regel*/
        for (rr=2;rr<=Rno;rr++) { /*rr=(lr,rr1 ...) mit*/
          if (sy==Reg[rr][0]) {   /*lr=r' gibt, so nimm das*/
            stel.r=rr;stel.p=0;   /*Zustandselement (rr,0) hinzu.*/
            if (stateappend(sta,stel)) stelgf=1;
            }
          }
        }
      }
    }
  }
}

stateappend(sta,stel)
rppaar stel;   /*Hinzunahme des Zustandselements stel*/
short sta; {
short elneu=1; /*Element stel ist neu*/
short i;

i=sta;
while (elneu&&(i<=Statestend)) {
  elneu=((stel.r!=Statest[i].r)||(stel.p!=Statest[i].p));i++;}
if (elneu) {
  if (++Statestend>=STST_MAX) {
    fprintf(erfptr,"Zu viele Zustandselemente, STST_MAX=%d\n",
            STST_MAX); exit(255);}
  Statest[Statestend]=stel;
  }
return elneu;
}
```

reduce.c

```
#include "myacc.h"

reduceact() {  /*Berechnung der reduce-Aktionen*/
short i,j,v,sy,tsy,ri,sp;
short sy_suc_tsy;
short stati,statianf,statibend;
short ncomp,stcomp[NST_MAX];

for (stati=2;stati<=Nstates;stati++) {  /*Für alle Zustände:*/
  statianf=Stateind[stati];
  statibend=statianf+Nstatebase[stati]-1;
  ncomp=0;  /*Suche nach vollständigen Zustandselementen (ri,sp)*/
  for (i=statianf;i<=statibend;i++) { /*d.h. mit sp=Rlgth[ri]*/
    ri=Statest[i].r;sp=Statest[i].p;
    if (sp==Rlgth[ri]) stcomp[++ncomp]=ri;
    }
  for (i=1;i<=ncomp;i++) { /*Für alle vollständigen*/
    ri=stcomp[i];          /*Zustandselemente:*/
if (!Shiftaction[stati]&&(ncomp==1)) { /*LR(0)*/

  /*Es gibt genau ein vollständiges Element, und es wurde
  **keine shift-Aktion ermittelt: Unabhängig vom Arbeitssymbol
  **wird entsprechend Regel ri reduziert.*/

      for (v=0;v<=Nvoc;) Lrtab[stati][v++]=(-ri);
      } else { /*SLR(1)*/

  /*Für das vollständige Zustandselement (ri,sp) mit der Regel
  **ri=(sy,...) wird zum Arbeitssymbol tsy=Voc[v] eine Reduktion
  **entsprechend Regel ri eingetragen, wenn (sy,tsy) zur
  **Nachfolgerelation gehört, tsy also Nachfolger von sy ist.*/

      sy=Reg[ri][0];
      for (v=Nontno+1;v<=Nvoc;v++) { /*terminale Symbole*/
        tsy=Voc[v];
        for (j=1;j<=Nsucrel;j++)
          if (sy_suc_tsy=((Sucrel[j].a==sy)&&
                          (Sucrel[j].e==tsy))) break;
        if (sy_suc_tsy) {
          if (!Lrtab[stati][v]) Lrtab[stati][v]=(-ri);
            else {
            fprintf(erfptr,
                    "Die Grammatik enthält Konlikte!\n");
            exit(255);
            }
          }
        }
      }
    }
}
```

## Literaturverzeichnis

Bach, F.; Baur, A.; Jansen, C.; Spies, G. (1986):
UNIX-Tabellenbuch. München, Wien: Carl Hanser Verlag

Bach, F.; Domann, P.; Weng-Beckmann, U. (1987): UNIX Handbuch zur Programmentwicklung. München, Wien: Carl Hanser Verlag

Banahan, M.; Rutter, A. (1984): UNIX lernen, verstehen, anwenden.
München, Wien: Carl Hanser Verlag

Bothe, K.; Hohberg, B.; Horn, C.; Wikarski, O. (1987):
PASCAL=>C Portabilität durch Quelltexttransformation.
edv-aspekte 4(1987)

Bourne, S.R. (1978): An Introduction to the UNIX Shell.
UNIX Programmer's Manual, 7th Ed., Vol. 2A.
Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

Claßen, L.; Oefler, U. (1987): UNIX und C - Ein Anwenderhandbuch.
Berlin: VEB Verlag Technik; Heidelberg: Dr. Alfred Hüthig Verlag

Clauß, M.; Fischer, G. (1988): Programmieren mit C.
Berlin: VEB Verlag Technik; Heidelberg: Dr. Alfred Hüthig Verlag

Comer, D. (1984): Operating System Design, the XINU approach.
London: Prentice-Hall International, Inc.

Daley, R.C.; Neumann, P.G. (1965): A general-purpose file system for secondary storage. Proc. AFIPS Fall Joint Computer Conference, 27, part 1, pp. 213-29

Feldman, S.I. (1978): Make - A Program for Maintaining Computer Programs. UNIX Programmer's Manual, 7th Ed., Vol. 2A.
Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

Hartwig, M.; Stein, E.; Strobel, R. (1988): Die Programmiersprache Ada als Kern von Software-Produktions-Umgebungen.
rechentechnik/datenverarbeitung 8(1988), S. 14

Haviland, K.; Salama, B. (1987): UNIX system programming.
Wokingham, England: Addison-Wesley Publishing Company, Ltd.

Jackson, M.A. (1975): Principles of program design.
London, New York: Academic Press

Johnson, S.C. (1978): YACC: Yet Another Compiler-Compiler.
UNIX Programmer's Manual, 7th Ed., Vol. 2A.
Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

Kernighan, B.W.; Pike, R. (1984): The UNIX Programming Environment.
Englewood Cliffs: Prentice Hall; Deutsche Ausgabe (1987):
Der UNIX-Werkzeugkasten. München, Wien: Carl Hanser Verlag

Lesk, M.E.; Schmidt, E. (1978): LEX - A Lexical Analyzer Generator.
UNIX Programmer's Manual, 7th Ed., Vol. 2A.
Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

Loeper, H.; Bachmann, P. (1980): Theorie und Technik der Übersetzerprogramme höherer Programmiersprachen. Schriftenreihe Informationsverarbeitung. Leipzig: BSB B.G. Teubner Verlagsgesellschaft

Loeper, H.; Jäkel, H.-J.; Otter, W. (1987): Compiler und Interpreter für höhere Programmiersprachen. Berlin: Akademie-Verlag

McMahon, L.E. (1978): SED - A Non-interactive Text Editor. UNIX Programmer's Manual, 7th Ed., Vol. 2A. Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

Rechenberg, P. (1984): Attributierte Grammatiken als Methode der Softwaretechnik. Elektronische Rechenanlagen 26(1984), S.111

Rechenberg, P.; Mössenböck, H. (1985): Ein Compiler-Generator für Mikrocomputer. München, Wien: Carl Hanser Verlag

Ritchie, D.M.; Thompson, K. (1974): The UNIX Time-Sharing System. Comm. ACM, 17, No. 7 (July 74), pp. 365-375

Schreiner, A.-T. (1984): System-Programmierung in UNIX. Band 1. Stuttgart: Teubner Verlag

Schreiner, A.-T. (1986): System-Programmierung in UNIX. Band 2. Stuttgart: Teubner Verlag

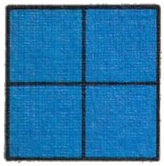
Schreiner, A.-T. (1987): Professor Schreiners UNIX Sprechstunde. München, Wien: Carl Hanser Verlag

Schreiner, A.-T.; Friedman, H.G.jr. (1985): Compiler bauen mit UNIX. München, Wien: Carl Hanser Verlag

WEGA (1987): WEGA-Software Programmierhandbuch, Teile 1 und 2. Berlin: Kombinat Elektro-Apparate-Werke »Friedrich Ebert«

X/OPEN (1987): X/OPEN Portability Guide, Bände 1 bis 5. Amsterdam: Elsevier Science Publishers B.V.

**Sachwörterverzeichnis**
(*Kursiv*: Kommandos)



UNIX ist ein standardisiertes Betriebssystem mit vielfältigen Werkzeugen zur Programmentwicklung.
Das Buch behandelt die Themenkomplexe: das Dateiverwaltungssystem mit Werkzeugen zur Dateibehandlung, die Kommandointerpreter Bourne-Shell und C-Shell, das technologische Werkzeug make und die Programmgeneratoren yacc und lex.

ISBN 3-341-00865-9