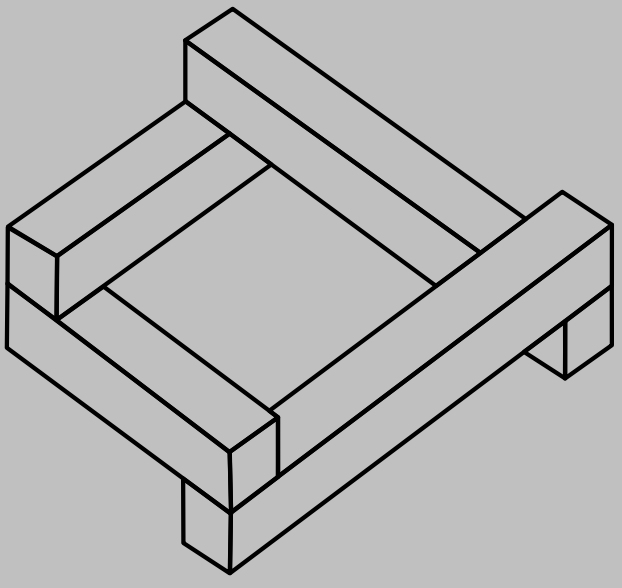
R. Fößmeier

Die Schnittstellen von UNIX-Programmen

Tips zur Programm-Organisation unter UNIX



Springer-Verlag

Vorwort

Das Betriebssystem UNIX[1] erfreut sich wachsender Verbreitung auf Rechnern aller Klassen, vom Einplatzsystem zum Supercomputer. Es ist nach einer Reihe von Grundsätzen aufgebaut, die möglichst große Übersichtlichkeit, Flexibilität und Portabilität sowohl der System-Programme als auch zu erstellender Benutzerprogramme ermöglichen. Diese Grundsätze, zu denen u. a. die bevorzugte Verwendung kleiner, aber universeller Werkzeuge gehört, sind z. B. bei Buschlinger (1985) formuliert und werden häufig als UNIX-*Philosophie* bezeichnet.

UNIX hat eine längere Entwicklung hinter sich, an der, oft unabhängig voneinander, viele Menschen beteiligt waren. Eine Beschreibung des Systems gleicht daher in vieler Hinsicht der Arbeit eines Linguisten oder Grammatikers, der a posteriori nach Regeln in einer Sprache sucht, die ohne bewußte Absicht evolutionär entstanden sind. Solche Regeln haben die Grundphilosophie im Laufe der Zeit ergänzt und bis zu einem gewissen Grad formalisierbar gemacht.

Die UNIX-Philosophie hat zur Entwicklung einer Reihe von Standardwerkzeugen geführt, die sich durch Standard-Schnittstellen miteinander verbinden lassen, wodurch viele Probleme ohne das Schreiben eigener Programme lösbar sind. Diese Standardwerkzeuge sind in den Systemhandbüchern ausreichend dokumentiert und in vielen Büchern gut beschrieben. Dem Benutzer, der Einzelheiten zu einem bestimmten Kommando lernen oder nachschlagen will, steht damit reichhaltiges Material zur Verfügung. In der Praxis gibt eine Aufgabe jedoch keine unmittelbaren Hinweise auf die Werkzeuge, mit denen sie sich lösen läßt. Hier ist der Programmierer auf seine Intuition angewiesen, die ihn vertraute Grundstrukturen, insbesondere Schnittstellen, wiedererkennen läßt und auf diesem Wege dann auch zu geeigneten Werkzeugen hinführt. Erst im letzten Stadium dieses Prozesses setzen die meisten Lehr- und Nachschlagewerke ein.

Dieses Buch versucht früher anzusetzen und den Programmierer mit bestimmten Schnittstellen vertraut zu machen, die sich in der Entwicklung und im Gebrauch von UNIX herauskristallisiert haben. Anhand einer detaillierten Beschreibung dieser Schnittstellen wird das Zusammenwirken von Werkzeugen untereinander und auch mit Benutzerprogrammen erklärt. Dieser Ansatz berücksichtigt stärker den mehr daten-orientierten Zugang eines modernen Software-Entwurfs; er kann den Leser, der schon Grundkenntnisse über UNIX besitzt, zu einem tieferen Verständnis der UNIX-Philosophie führen, als er es durch das Studium einzelner Kommandos und Systemaufrufe erwerben könnte.

---

[1] UNIX ist ein in den USA und anderen Ländern eingetragenes Warenzeichen der Bell-Laboratorien von AT&T

Ein so unbestimmter Begriff wie die *Philosophie* eines Betriebssystems läßt sich natürlich nicht rein theoretisch vermitteln, deshalb habe ich in allen Abschnitten darauf geachtet, ausreichend Beispiele zu bringen; sowohl einfache, um die Funktion zu erläutern, als auch realistischere, um die Anwendung zu demonstrieren. Wo es möglich ist, füge ich Erklärungen direkt als Kommentare in den Kommando- oder Programmtext ein. Praktisch interessante oder umfassendere Beispiele, die sich auf mehrere Abschnitte beziehen, sind im Anhang B zusammengefaßt. Ein kleines Glossar erklärt schließlich wichtige Fachausdrücke, die vielleicht kein Allgemeingut sind oder speziell in diesem Buch eingeführt werden.

Die Beschreibungen in diesem Buch setzen voraus, daß der Leser bereits erste Kontakte mit einem UNIX-System geknüpft hat. Es ist daher weder als Einführung für den UNIX-Neuling gedacht noch als Nachschlagewerk für Kommandosyntax oder -optionen konzipiert; dazu sind die Systemhandbücher da, die es schon vom Umfang her nicht ersetzen kann. Ich habe versucht, wesentliche Eigenschaften, Möglichkeiten und Besonderheiten aufzuführen, und verweise an geeigneten Stellen auf weiterführende Literatur.

Da es nicht überall möglich war, Versions-Abhängigkeiten auszuklammern, beziehe ich mich auf die UNIX-Version *System V*; gelegentlich verweise ich jedoch auf Unterschiede zu anderen UNIX-Versionen oder UNIX-ähnlichen Systemen. Alle Beispiele in diesem Buch wurden u. a. unter UNIX System V auf einem am Institut für Informatik der TU München entwickelten PAX-Rechner mit Prozessor INTEL 80286/7 getestet. Soweit Ausführungszeiten angegeben sind, beziehen sie sich auf diese Konfiguration.

Ein kleines Hindernis beim Verfassen dieses Buches war die teilweise noch verkümmerte deutsche Terminologie auf dem Gebiet UNIX und auf dem Gebiet der Datenverarbeitung allgemein. Obwohl es eigentlich kein Problem sein sollte, für Ausdrücke wie *shell* oder *pipe* ein deutsches Wort zu finden, zeigt die deutsche Sprache dabei sehr wenig Lebendigkeit (vgl. Probst 1989). Ich fühle mich hier nicht zu einem Alleingang berufen, möchte aber den Leser ermutigen, diese Situation nicht als unabänderlich hinzunehmen. Einen guten Anfang macht das System-V-Handbuch (1988), das einen erfrischend klaren Wortschatz, jedoch teilweise eine stark amerikanisierte Grammatik verwendet.

Ich möchte an dieser Stelle allen danken, die mit mir die Geheimnisse des UNIX-Systems erforscht haben, vor allem meinem Kollegen Ulrich Rüde, der mich auf viele interessante Aspekte aufmerksam gemacht hat. Besonderen Dank schulde ich weiterhin Dietmar Friede, durch den ich vor vielen Jahren mit UNIX in Kontakt gekommen bin. Dem Springer-Verlag gebührt Dank für seine Hinweise für die Gestaltung wie auch zu inhaltlichen Details des Buches.

München, im November 1990 *Reinhard Fößmeier*

Inhaltsverzeichnis

Anhang A: Tabellen

Tabelle A.1. Allgemeine Referenz-Version des 8-Bit-Codes (ARV8) nach DIN 66303 (ISO 8859/1)
Die Code-Nummern der Zeichen sind in hexadezimaler Form angegeben; die Zahl links entspricht den niedererwertigen vier Bits, die Zahl oben den höherwertigen.

	2	3	4	5	6	7
0		0	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	´	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	{
C	,	<	L	\	l	|
D	−	=	M	]	m	}
E	.	>	N	^	n	~
F	/	?	O	_	o	

A	B	C	D	E	F
	°	À	Ð	à	ð
¡	±	Á	Ñ	á	ñ
¢	²	Â	Ò	â	ò
£	³	Ã	Ó	ã	ó
¤	´	Ä	Ô	ä	ô
¥	µ	Å	Õ	å	õ
¦	¶	Æ	Ö	æ	ö
§	·	Ç	×	ç	÷
¨	¸	È	Ø	è	ø
©	¹	É	Ù	é	ù
ª	º	Ê	Ú	ê	ú
«	»	Ë	Û	ë	û
¬	¼	Ì	Ü	ì	ü
	½	Í	Ý	í	ý
®	¾	Î	Þ	î	þ
¯	¿	Ï	ß	ï	ÿ

In der deutschen Referenz-Version (DRV8) tauschen folgende Zeichen ihre Plätze:

↕	@	[	\	]	{	|	}	~
	§	Ä	Ö	Ü	ä	ö	ü	ß

Folgende Steuerzeichen besitzen in UNIX-Programmen eine einheitliche Interpretation sowie eine besondere Ersatzdarstellung (vgl. Abschnitt 2.4.2):

Tabelle A.2. Steuerzeichen

hex.	okt.	C	Name
8	10	\b	Rückwärtsschritt *(backspace)*
9	11	\t	Tabulator
A	12	\n	Zeilenvorschub (LF)
B	13	\v	Vertikaler Tabulator
C	14	\f	Neue Seite
D	15	\r	Wagenrücklauf (CR)

Diakritische Zeichen (Über- und Unterzeichen) in Spalte C (12) der Mehr-Byte-Version (MBV8, ISO 6937), jeweils mit einem Beispiel:

Tabelle A.3. Diakritische Zeichen (MBV8)

		4	˜ñ	8	¨ä	C	_a
1	`à	5	¯ā	9	.ş	D	˝ő
2	´é	6	˘ŭ	A	°å	E	˛ę
3	ˆĉ	7	˙ċ	B	¸ç	F	ˇš

Die folgende Tabelle führt zu allen abdruckbaren Sonderzeichen auf, welche besondere Funktion sie für bestimmte UNIX-Programme haben, insbesondere auf der Kommmandoebene, also beim Aufruf oder im Dialog mit der *Shell*. Aufgrund der Vielzahl der Programme und der raschen Entwicklung kann die Liste leider keinen Anspruch auf Vollständigkeit erheben. Die Abkürzung „RA“ bedeutet, daß ein Zeichen in regulären Ausdrücken vorkommt.

Tabelle A.4. Besondere Funktionen von Sonderzeichen

Zeichen	Funktion
!	Verzweigt aus vielen Kommandos in eine *Shell*, z. B. aus Editoren *Shell:* komplementiert Zeichenklassen in [] *csh:* wiederholt Kommandos
"	*Shell, troff:* Zitierklammer

Zeichen	Funktion
#	*sh, m4 ...:* leitet Kommentare bis zum Zeilenende ein *vi:* Abkürzung für letzten Dateinamen
$	*Shell:* Wert einer Variable bestimmen RA: Zeilenende
%	*vi:* Abkürzung für den Namen der editierten Datei *printf* u. a: Format-Bezeichnung *NLSPATH:* Spezial-Komponenten *mail:* sekundärer Adreß-Trenner (vgl. @)
&	*Shell:* Ausführung im Hintergrund *sed:* Platzhalter für ein erkanntes Ersetz-Muster
'	*Shell:* Zitierklammer *troff:* Kommandosymbol am Zeilenanfang *m4:* als rechtes Anführungszeichen voreingestellt.
()	*Shell:* Kommandos in Unter-*Shell* ausführen RA: gruppieren
*	RA: beliebige Wiederholung (auch *Shell*)
+	RA: mindestens einmalige Wiederholung *sort:* leitet Feldnummern ein
,	*getopts*: Trennung mehrerer Werte bei Wertoption */etc/passwd:* Felder nochmals unterteilen
−	K-Schnittstelle: Option einleiten
.	RA: beliebiges Zeichen *troff:* Kommandosymbol am Zeilenanfang 64er-Zahlen: Ziffer 0 (s. Abschnitt 2.7.5) Dateiname: trennt Grundnamen und Endung; .. = Elterverzeichnis, . = eigenes Verzeichnis *Shell:* Kommandos in einer Datei ausführen *mail:* trennt Komponenten in Adressen
/	Dateinamen: Verzeichnisse im Pfadnamen trennen 64er-Zahlen: Ziffer 1
:	*/etc/passwd* u. a: Felder trennen *rcp, mount:* Rechnernamen abschließen *Shell:* wirkungsloses Kommando
;	*Shell, find:* Anweisungen trennen
<	*Shell:* Eingabe umlenken
=	*Shell:* Variablen setzen
>	*Shell:* Ausgabe umlenken
?	*Shell:* beliebiges Zeichen in Dateinamen ersetzen RA: 0- oder 1-maliges Vorkommen
@	*mail, talk, mount:* Rechnernamen einleiten
[]	RA: Zeichenklassen bilden (auch*Shell*)
\	vielfach: Spezialbedeutung des folgenden Zeichens aufheben (Zitieren); Zeichen in Oktaldarstellung angeben, u. a. m.
^	RA: Zeilenanfang; Komplement einer Zeichenklasse *Shell:* altes Zeichen für *Pipe* (|)

Zeichen	Funktion
_	vielfach neben Buchstaben in Namen zugelassen
`	*Shell:* setzt Kommandoausgabe in *Shell*-Eingabe ein *m4:* als linkes Anführungszeichen voreingestellt.
{ }	*Shell:* mehrere Kommandos gruppieren, Ausgabe verketten Variablennamen klammern: `${..}`
|	*Shell: Pipe* RA: Alternative
~	*ksh, csh:* `$HOME`-Verzeichnis *mail:* Sonderfunktionen aufrufen

Tabelle A.5. Standardfunktionen zur Bearbeitung von Zeichenreihen

`char * strcat (s1,s2)`	hängt `s2` an `s1` an
`int strcmp (s1,s2)`	v ergleicht `s2` und `s1`
`char * strcpy (s1,s2)`	k opiert `s2` in `s1`
`int strlen (s)`	berechnet die Länge v on `s`
`char * strchr (s,c)`	zeigt auf das erste `c` in `s`
`char * strpbrk(s1,s2)`	zeigt auf das erste Zeichen aus `s2` in `s`
`int strspn (s1,s2)`	berechnet die Länge des Anfangsstücks von `s1`, das *nur* Zeichen aus `s2` enthält
`int strcspn(s1,s2)`	bzw. das *keine* Zeichen aus `s2` enthält
`char strtok (s1,s2)`	zerle gt `s1` nach Trennern aus `s2` (s. u.)
`char * s, s1, s2` `int c`	*(Typen der Parameter)*

Zu *strcat*, *strcpy* und *strcmp* gibt es Varianten, deren Namen mit *strn* beginnen. Sie haben einen dritten Parameter vom Typ *int*, der die Länge der kopierten oder verglichenen Zeichenreihe beschränkt. Bei *strncpy* wird *genau* die angegebene Zahl von Bytes kopiert, auch wenn vorher ein *\0* auftritt.

Tabelle A.6. Standardfunktionen zur Typfeststellung und -umwandlung von Zeichen (sprachabhängig unter NLS!)

Name	Bedeutung	Manual
`int isalpha (c)`	ist `c` Buchstabe?	*ctype (3)*
`int isupper (c)`	– Großb uchstabe?	
`int islower (c)`	– Kleinb uchstabe?	
`int isdigit (c)`	– Zif fer?	
`int isxdigit(c)`	– Hexadezimalziffer?	
`int isalnum (c)`	– Buchst. od. Zif fer?	
`int isspace (c)`	– Leerraum (`\t\r\n\f\v`)?	
`int isprint (c)`	– abdruckbar (32–126)?	
`int isgraph (c)`	– sichtbar ? (`isprint`, aber nicht Leerzeichen)	
`int ispunct (c)`	– Sonderzeichen ? (`isgraph`, aber nicht `isalnum`)	
`int iscntrl (c)`	– Steuerzeichen (`DEL` oder < 32)?	
`int isascii (c)`	– ASCII-Zeichen (< 128)?	
`int toupper (c)`	`c`→ Großb uchstabe	*conv (3)*
`int tolower (c)`	`c`→ Kleinb uchstabe	
`int toascii (c)`	höchstes Bit v on `c` wird 0	

Die Zeichenklassen sind in einem externen Vektor festgelegt, der in der Datei <*ctype.h*> vereinbart ist. Diese Datei muß mit *#include* in das Programm eingebunden werden.
Unter NLS (siehe Abschnitt 6) hängt die Klasse eines Zeichens von der eingestellten Sprache ab, ebenso die Entsprechung von Groß- und Kleinbuchstaben.

Tabelle A.7. Standardfunktionen zur Umw andlung zwischen Zeichenreihen und anderen Typen

Name	Bedeutung	Manual

`long strtol (s,p,ba)`
`long atol (s)`
`int atoi (s)`
	g anze Zahlen zur Basis *ba*
	} Basis 10
		strtol (3)

`double atof (s,p)`
`double strtod (s,p)`
`char * [efg]cvt (...)`
	Gleitk omma-Zahlen
	- mit Weiterschalten von *p*
	- formatgesteuert
		strtod (3)
		ecvt (3)

`int sscanf (s,f,...)` `int sprintf (s,f,...)`	universell, formatgesteuert *(f)*	*scanf (3)* *printf (3)*
`long a64l (s)` `char * l64a (l)`	Zahlen zur Basis 64 (s. 2.7.5)	*a64l (3)*
`char *s, **p; long l; int ba;` Nach Aufruf von `strto[dl]` zeigt `*p` auf das erste Zeichen in `s`, das nicht zur verarbeiteten Zahl gehört.		

Reguläre Ausdrücke: Die Programme *ed*, *grep* und *sed* verarbeiten folgendermaßen aufgebaute reguläre Ausdrücke:

Tabelle A.8a. Elementare reguläre Ausdrücke

Ausdr.	paßt zu	Beispiel	paßt zu
`.`	jedem Zeichen	`.NIX`	`UNIX`, `ENIX`, ...

`[...]`
–
	einem der Zeichen in ...
	Bereiche können dabei in der Form *anfang-ende* angegeben werden
		`[awk]`
		`[a-z]`
		`[A-Z]`
		`[1-4]`
			`a`, `w` oder `k`
			jedem Kleinbuchstaben
			jedem Großbuchstaben
			`1`, `2`, `3` oder `4`

`[^...]`	einem Zeichen außer ...	`[^A-Za-z]`	jedem Nicht-Buchstaben

x`*`	jeder Anzahl von *x*	`A*`
		`.*NIX`
		`[1-9][0-9]*`
		`.*`
			ε, `A`, `AA`, `AAA`, ...
			`NIX`, `UNIX`, `XENIX`, ...
			positiven Dezimalzahlen
			allem

`^`	Zeilenanfang	`^Wort`	„`Wort`“ am Zeilenanfang
`$`	Zeilenende	`Wort$` `^$`	„`Wort`“ am Zeilenende leere Zeile

Ausdrücke mit einer erweiterten Syntax werden von den Programmen *egrep* und *awk* erkannt (α und β stehen für reguläre Ausdrücke, die nötigenfalls geklammert sind; ε steht für die leere Zeichenreihe):

Tabelle A.8b. Erweiterte reguläre Ausdrücke

Ausdruck	paßt zu
α+	α oder mehrere Wiederholungen davon
α?	ε (leer) oder α
$\alpha \mid \beta$	α oder β (statt „ | “ auch `NL`)
(α)	α (zum Gruppieren)

Für die C-Unterprogramme *regcmp* und *regex* gelten noch einige Erweiterungen. Die Funktion des Fragezeichens fällt allerdings weg; sie wird durch *{0,1}* ersetzt:

Tabelle A.8c. Erweiterungen bei *regcmp*

α{*m*}	α *m*-mal hintereinander
α{*m*,}	α *m*-mal oder öfter hintereinander
α{*m*,*n*}	α *m*- bis *n*-mal hintereinander
`\n`	`NL` (Zeilenwechsel), `$` ist Ende der Zeichenreihe

Tabelle A.9. (Ungefähre) Entsprechungen zwischen K ommandos und C-Unterprogrammen

„+“ = wird innerhalb der*Shell* ausgeführt. C-Unterprogramme aus Sektion (2) entsprechen Systemaufrufen, solche aus Sektion (3) Bibliotheksroutinen.

sh	C (Manual)	Beschreibung
+cd	chdir(2)	Arbeitsverzeichnis ändern
chown	chown(2)	Eigentümer einer Datei ändern
chgrp	chown(2)	Eigentümergruppe einer Datei ändern
chmod	chmod(2)	Zugriffsrechte einer Datei ändern
date	time(2) ctime(3)	Datum und Uhrzeit
+exec	exec(2)	Neuen Prozeß starten
+exit	exit(2)	eigenen Prozeß beenden
find	ftw(3)	Dateibaum durchsuchen
kill	kill(2)	Signal an Prozeß senden
ln	link(2)	Zusätzlichen Namen für eine Datei erzeugen
ls	stat(2)	Informationen über eine Datei abfragen
mknod	mknod(2)	Verzeichnis-Eintrag erzeugen
rm	unlink(2)	Verzeichnis-Eintrag löschen
sleep	sleep(3)	Prozeß „nicht rechenwillig“ setzen
sort	qsort(3)	Sortieren
time	clock(3)	Rechenzeit eines Prozesses
+times		eigene Rechenzeit (der *Shell*)
touch	utime(2)	Zugriffsdatum einer Datei ändern
+trap +onintr	signal(2/3)	Unterbrechungssignale abfangen
tty	ttyname(3)	Name des eigenen Sichtgerätes
+ulimit	ulimit(2)	Dateigröße begrenzen
+umask	umask(2)	Einschränkung von Zugriffsrechten einstellen
+wait	wait(2)	auf Ende eines Prozesses warten

Anhang B: Beispiele

Beispiel B.1. Ausschreiben von Schecks
Zu einer Liste von Namen und Beträgen sind Schecks auszudrucken. Dabei muß der Betrag auch in Worten angegeben werden. Soweit möglich, soll die korrekte Anrede aus dem Vornamen ermittelt werden; dazu steht eine Liste von Vornamen zur Verfügung.
Es werden die UNIX-Datenbankwerkzeuge verwendet; die auftretenden Tabellen (Relationen) seien folgendermaßen aufgebaut:

Eingabeschnittstelle:

```
Name : Vorname : Betrag
Huber:Siegfried:310,16
Meier:Barbara Ulrike:380,50
Schulze:Hans-Joachim:425,00
 ...
```

Vornamenliste
(Datei *genus.dat*):

```
Vorname : Geschlecht
Barbara:w
Hans:m
Siegfried:m
Uli:?
 ...
```

1. Zwischenschnittstelle:

```
Name : Vorname : Betrag : Geschlecht
Huber:Siegfried:310,16:m
Meier:Barbara:380,50:w
Schulze:Hans-Joachim:425,00::m
 ...
```

2. Zwischenschnittstelle:

```
Name : Vorname : Betrag : Geschlecht : Betrag in Worten
Huber:Siegfried:310,16:m:dreihundertzehn
Meier:Barbara:380,50:w:dreihundertachtzig
Schulze:Hans-Joachim:425,00:m:vierhundertfuenfundzwanzig
 ...
```

Zum Übergang von der Eingabeschnittstelle auf die erste Zwischenschnittstelle (Geschlecht zum Vornamen bestimmen) kann ein **awk-Programm** dienen: Zuerst wird ein Feld *gen* angelegt, in dem zu jedem Vornamen die Geschlechtsangabe gespeichert ist. Anschließend werden etwaige Doppel-Vornamen getrennt und nach dem ersten Teil das Geschlecht bestimmt. *awk* le gt das Feld *gen* nach dem Streuspeicherverfahren ab, so daß die Zugriffszeiten kurz sind.

```
awk -F: '
FILENAME ~ /genus.dat$/          # Vornamenliste
        { gen[$1] = $2; next }
        {                        # Eingabe
# -->   Name    Vorname Betrag
        split ($2, DOPNAME, " ")
        split (DOPNAME[1], VORNAME, "-")
        print $1,DOPNAME,$3,gen[VORNAME[1]] }
# -->   Name    Vorname Betrag  Geschlecht
'  OFS=":"  genus.dat  -
```

Der Übergang auf die erste Zwischenschnittstelle kann auch mit dem Datenbank-Programm *join* erfolgen. Dazu müssen die beiden Listen aber nach ihrem verbindenden Element, nämlich den Vornamen, geordnet sein. Wenn wir das für die Vornamenliste *genus.dat* voraussetzen, können wir die folgenden Kommandos verwenden:

```
LJ="-j1 2 -j2 1"                # V erbindungsfelder
LO="-o 1.1 1.2 2.2"             # Ausg abefelder
sort -t: +1 | \                 # nach 2. Feld sortieren
join -t: $LJ $LO - genus.dat    # und v ereinigen
```

Anschließend muß die Liste wieder so sortiert werden, wie sie vorher war, z. B. nach Familiennamen. Beachten Sie, daß das zweite Feld für*sort* die Nummer *1* hat! Dagegen zählt *join* seine Felder beginnend mit 1, so daß die Ausgabefelder die Nummern 1 und 2 haben.

Abbildung 56 zeigt das **awk-Programm** zum Übergang von der ersten auf die zweite Zwischenschnittstelle (Betrag in Worten ausschreiben); als Kommentarzeilen sind der Aufbau der der Ein- und Ausgabe-Schnittstelle angegeben.
Das **Drucken des Schecktextes** erfolgt in einem *Shell*-Skript; es könnte auch mit Kommandos wie *awk* oder *nroff* erfolgen:

```
IFS=":" while read ZEILE; do
        set $ZEILE
        echo "Zahlen Sie gegen diesen Scheck"
        echo "$5 deutsche Mark **DM $3**\n"
        case $4 in
                m) echo "an Herrn $2 $1" ;;
                w) echo "an Frau $2 $1" ;;
                *) echo "an Firma/Frau/Herrn $2 $1" ;;
        esac done
```

```
awk -F: '
# Eingabe:   Name    Vorname Betrag  Geschlecht
BEGIN {
    xe="ein zwei drei vier fuenf sechs sieben acht neun"
    xz="zehn zwanzig dreissig vierzig fuenfzig" \
       "sechzig siebzig achtzig neunzig"
    xh="elf zwoelf dreizehn vierzehn fuenfzehn" \
       "sechzehn siebzehn achtzehn neunzehn"
    split (xe, we, " ")
    split (xz, wz, " ")
    split (xh, wh, " ")
}
{
    printf "%s:%s:%s:%s:", $1,$2,$3,$4
    z = "00"$3 ; split(z, zahl, ",")
    lz=length(zahl[1])
    hs=substr(zahl[1],lz-2,1)
    zs=substr(zahl[1],lz-1,1)
    es=substr(zahl[1],lz  ,1)

    if (hs > 0) printf "%shundert", we[hs]
    if (zs == 0) {
        if (es == 1) printf "eins"
        else if (es > 1) printf "%s", we[es]
    }
    else if (zs == 1) printf "%s", w1[es]
    else if (es > 0) printf "%sund%s", we[es], wz[zs]
    printf "\n"
# Ausgabe:   Name  Vorname  Betrag  Geschlecht  Betr. i. W.
} '
```

Abb. 56. *awk*-Programm zum Umsetzen von Zahlen in Worte

Ergebnis:

```
Zahlen Sie gegen diesen Scheck
dreihundertzehn deutsche Mark **DM 310,16**
an Herrn Siegfried Huber
```

```
Zahlen Sie gegen diesen Scheck
dreihundertachtzig deutsche Mark **DM 380,50**
an Frau Barbara Meier
```

```
Zahlen Sie gegen diesen Scheck
vierhundertfuenfundzwanzig deutsche Mark **DM 425,00**
an Herrn Hans-Joachim Schulze
```

Unschön ist noch die Darstellung von *ü* durch *ue* usw. Bessere Methoden zur Behandlung von Umlauten sind in Abschnitt 5 beschrieben.

calprog.sh	Erklärung
`M1="[Jj]an";  M2="[Ff]eb";  M3="[Mm]ar"`	
`M4="[Aa]pr";  M5="[Mm]ay";  M6="[Jj]un"`	
`M7="[Jj]ul";  M8="[Aa]ug";  M9="[Ss]ep"`	
`M10="[Oo]ct"; M11="[Nn]ov"; M12="[Dd]ec"`	
`VOR="(^|[ (,;])"`	# Vorschalt-Muster
`NCH="([^0-9]|\$)"`	# Nachschalt-Muster
`set 'date "+%d %m %y %w"'`	# heutiges Datum ist ?
`TG='expr $1 + 0'`	# führende Nullen weg
`MO='expr $2 + 0'`	
`JHR=$3 ; WTG=$4`	# Jahr, Wochentag
`case $MO in`	
`  2)  if expr $JHR % 4 >/dev/null`	# Februar hat
`     then ZT=28; else ZT=29`	# 28 oder 29 Tage
`     fi ;;`	
`  4|6|9|11) ZT=30 ;;`	# Apr/Jun/Sep/Nov
`  *)  ZT=31 ;;`	# andere Monate
`esac`	
`if  [ $4 -eq 5 ];  then LST='3 2 1 0'`	# am Freitag 4 Tage ...
`elif [ $4 -eq 6 ];  then LST='2 1 0'`	# am Samstag 3 Tage ...
`                    else LST='1 0'; fi`	# sonst 2 Tage voraus
`S='(($MNAM[^ ]* *\|0*$MO/|\*/)0*($TAGE))'`	# Schema für's Datum
`MNAM=`	
`EGREP=`	# Skript noch leer
`for n in $LST ; do`	
`  if [ "$MNAM" = "" ]`	# falls kein Monatsname:
`  then TAGE="$TG"`	
`       eval MNAM='$M'$MO`	# dann Namen bestimmen
`  else TAGE="$TAGE\|$TG"`	
`  fi`	
`  if [ "$n" = "0" ]; then break; fi`	
`  if [ $TG -ge $ZT ]`	# falls Monatsletzter:
`    then eval "EGREP=\"$S|\""`	# dann Schema einsetzen
`         MO='expr "(" $MO % 12 ")" + 1'`	# neuen Monat bestimmen
`         MNAM=`	
`         TG=1`	
`    else TG='expr $TG + 1'`	# sonst weiterzählen
`  fi`	
`  done`	
`eval "EGREP=\"$VOR($EGREP$S)$NACH\""`	# Skript zusammensetzen
`echo $EGREP`	# und ausgeben

Abb. 57. Erzeugung des *egrep*-Skripts zur Datumssuche durch ein *Shell*-Programm

Beispiel B.2. Automatischer Terminkalender *(calendar)*
Das UNIX-Programm *calendar* durchsucht eine gleichnamige Datei nach dem Auftreten des heutigen oder morgigen Datums und informiert den Benutzer über die entsprechenden Termine (die ganze Zeile mit dem Datum wird ausgegeben). *calendar* verwendet das Programm *egrep*, um verschiedene (amerikanische) Formen des Datums zuzulassen. Beispiel für den 16. April:

```
(^|[ (,;])(([Aa]pr[^ ]* *|0*4/|\*/)0*16)([^0123456789]|$)
(^|[ (,;])(([Aa]pr[^ ]* *|0*4/|\*/)0*17)([^0123456789]|$)
```

Als Monatsangabe ist somit *Apr* oder *apr* (mit nachfolgenden Zeichen, also auch *April*) sowie auch {0}4 oder * (jeder Monat) zugelassen, aber nur am Zeilenanfang (^) oder nach Klammer, Komma, Punkt oder Leerzeichen. Es folgt der Tag, u. U. mit führenden Nullen; er muß von einer Nicht-Ziffer gefolgt sein oder am Zeilenende ($) stehen.
Da dieses *egrep*-Skript von einem C-Programm *(calprog)* erzeugt wird, läßt es sich nicht an sprachliche, regionale oder persönliche Eigenheiten anpassen, wenn man nicht über die C-Quellen verfügt. Das Skript ist außerdem sehr aufwendig; insbesondere an Freitagen, wenn auch das Datum des folgenden Sonntags und Montags geprüft wird und dabei noch der Monat wechselt, liegen die Rechenzeiten für *egrep* auf kleinen Anlagen im Bereich mehrerer -zig Sekunden (!). Das Beispiel eines *Shell*-Programms in Abbildung 57 zeigt, wie ein besseres *egrep*-Skript erzeugt werden kann. Außerdem ist ein *Shell*-Programm natürlich flexibler und läßt sich leichter an besondere Wünsche anpassen (siehe auch Fößmeier 1989).
Das erzeugte *egrep*-Skript lautet z. B. für dasselbe Datum

```
(^|[ (,;])((([Aa]pr[^ ]* *|0*4/|\*/)0*(16|17)))([^0-9]|$)
```

Es ist kürzer und wird durchschnittlich etwa doppelt so schnell ausgeführt. Das Programm kann leicht z. B. an die deutsche Datumsform angepaßt werden, indem das Schema *S* anders besetzt wird; zum Beispiel wird die Form „Tag.Monat“ erkannt durch

```
S='(($TAGE)\.($MNAM|$MO))'
```

Nachteilig ist, daß alles Rechnen durch Aufrufe von *expr* geschehen muß. Bei Verwendung der *Korn-Shell* könnte die Arithmetik von der *Shell* selbst erledigt werden.

Beispiel B.3. C-Unterprogramm **`zerleg`** zur Zerle gung von feldstrukturierten S-Schnittstellen in ihre Felder. Der Parameter *s2* enthält die (gleichberechtigten) Trennzeichen, *s1* wird zerlegt. Ein Aufruf mit*s1=NULL* liefert das nächste Feld aus dem letzten *s1*. Ist kein Feld mehr vorhanden, so ist das Ergebnis *NULL*. Das Programm arbeitet ähnlich wie *strtok*, zwei Trenner hintereinander oder ein Trenner am Anfang oder am Ende begrenzen jedoch ein *leeres* Feld.
Die hier verwendeten Techniken (Gruppierung von Zeichen durch Aneinanderreihen, Abschluß durch *\0*, Gebrauch des *NULL*-Zeigers) sind typisch für die Bearbeitung von alphanumerischen Daten in C.

```
char *
zerleg(s1,s2)
    char *s1, *s2;
{
    static char *feld;                          /* merkt alten Rest */
    char *s;

    if (! s1) s1 = feld;                        /* Rest v erwenden */
    if (! s1) return NULL;                        /* nichts zu tun */
    for (s = s1; *s; s++) {                              /* Suche */
        char *t;
        for (t=s2; *t; t++)                    /* nach allen Trennern */
            if (*s == *t) break;
        if (*t) break;                        /* bis einer gefunden ist */
    }
    if (*s == '\0') feld = NULL;                 /* oder s zu Ende */
    else            feld = s+1;                 /* Rest merk      en */
    *s = '\0';                                /* durch \0 abschließen */
    return s1;                                          /* abliefern */
}
```

Die doppelte Schleife über `s` und `t` wird verlassen, wenn in der inneren Schleife ein Trennzeichen gefunden ist. Da die Anweisung *break* in C nur eine einzige Schleife verlassen kann, könnte hier ein *goto* Verwendung finden; üblich ist aber die Abfrage einer geeigneten Bedingung nach der inneren Schleife, die feststellt, ob die Schleife auf normalem Weg beendet wurde. Dazu müßte in diesem Fall **t* den Wert 0 haben; bei anderen Werten wird also auch die äußere Schleife beendet.

„Telefonbuch“

```
# Sortieren, Umlaute gelten
# wie ae, oe, ue;
# Jeder Umlaut wird durch
# Transskription+# ersetzt;
# ß wird durch ss+# ersetzt
#
SED1='
s/Ä/Ae#/g
s/Ö/Oe#/g
s/Ü/Ue#/g
s/ä/ae#/g
s/ö/oe#/g
s/ü/ue#/g
s/ß/ss#/g
'
SED2='
s/Ae#/Ä/g
s/Oe#/Ö/g
s/Ue#/Ü/g
s/ae#/ä/g
s/oe#/ö/g
s/ue#/ü/g
s/ss#/ß/g
'
```

„Wörterbuch“

```
# Umlaute gelten wie der Grundvokal.
# Jeder Umlaut wird zu Grundvokal+#.
# an das Wort wird ein #
# und ein Leerzeichen angehängt,
# dadurch Einordnen nach einem
# gleichlautenden Wort ohne Umlaut
SED1='
s/Ä\([A-ß]*\)/A#\1# /g
s/Ö\([A-ß]*\)/O#\1# /g
s/Ü\([A-ß]*\)/U#\1# /g
s/ä\([A-ß]*\)/a#\1# /g
s/ö\([A-ß]*\)/o#\1# /g
s/ü\([A-ß]*\)/u#\1# /g
s/ß/ss#/g
'
SED2='
s/A#/Ä/g
s/O#/Ö/g
s/U#/Ü/g
s/a#/ä/g
s/o#/ö/g
s/u#/ü/g
s/ss#/ß/g
s/# //g
'
```

```
sed "$SED1" | sort -d | sed "$SED2"
```

Abb. 58. Alphabetisches Sortieren deutscher Texte mit *sed* und *sort*

Beispiel B.4. Abbildung 58 zeigt *sed*-Skripten für die Umk odierung deutscher Texte zum alphabetischen **Sortieren**, für die beiden in Abschnitt 6.4 beschriebenen Arten der Sortierung. Deutsche Texte werden zunächst so kodiert (`SED1`), daß die Sortierung mit dem Programm*sort* erfolgen kann, und anschließend wieder zurückkodiert (`SED2`). Die Buchstaben*ÄÖÜäöüß* können z. B. durch *[\]{|}˜* dargestellt sein.

Beispiel B.5. Programm **`look.c`** zum schnellen Suchen in sortierten Listen (vgl. Abschnitt 3.3.3).

```
#include <stdio.h>
#include <ctype.h>                          /* für „isalnum“ ...*/
#include <sys/types.h>
#include <sys/stat.h>                          /* für „stat“ */

#define DLISTE  "/usr/dict/words"
#define BLO     512
#define istlex(c) (isalnum(c) || isspace(c))
```

Einige globale Variablen sind für mehrere Unterprogramme wichtig:

```
char *komname;
int lex = 0, fol = 0;
FILE * liste;

main(pz,pw)
    int pz;
    char **pw;
{
    int lm, v, c, gefunden = 0;
    char *muster, *name, *m, *s;
    struct stat stpuf;
    long p0, p1;

    extern char *optarg;                    /* für „getopt“ */
    extern int optind;
```

Optionen werden verarbeitet, globale Variablen werden besetzt:

```
    komname = pw[0];                        /* für „fehler“ */
    while ((c = getopt(pz,pw, "fl")) != EOF)
        switch (c) {
        case 'f':    fol = 1; break;
        case 'l':    lex = 1; break;
        }

    if (optind >= pz)
        fehler (2, "Suchmuster ?", ""); /* Kein Suchmuster! */
    muster = pw[optind++];

    if (optind >= pz) {
        name = DLISTE;                       /* Standardwert */
        fol = 1;
    }
    else name = pw[optind];
```

Das Suchmuster wird entsprechend den Optionen vorbereitet:

```
    for (s = m = muster; *s; s++) {      /* Muster aufbereiten */
        if (lex)                        /* Sonderz. überspringen */
            while (! istlex(*s)) s++;
        if (fol)                          /* Kleinbuchstaben */
            *s = tolower(*s);
        if (*s == '\\') continue;
        else if (*s == '$') *m++ = '\n';   /* $ = Zeilenende */
        else *m++ = *s;
    }
    *m = '\0';
    lm = strlen(muster);
```

Eröffnen der Datei, Bestimmen der Länge:

```
    if ((liste = fopen(name, "r")) == NULL)
        fehler (2, "Datei %s nicht gefunden\n", name);
    if (fstat(liste->_file, &stpuf))          /* Länge = ? */
        fehler (2, "Fehler in stat(%s)", name);
```

Die untere und obere Grenze für das Bisektionsverfahren werden initialisiert: *p0* zeigt immer vor die Suchstelle, *p1* immer dahinter.

```
    p0 = 0;
    p1 = stpuf.st_size;

    while (p1 > p0+1) {                 /* Intervallschachtelung */
        long pos = (p0+p1)/2;

        v = suche (pos, muster, lm);
        if (v < 0)
            p0 = pos;
        else {
            p1 = pos;
            if (v == 0) gefunden = 1;
        }
    }
    if (! gefunden)
        fehler(1, "Muster `%s' nicht gefunden", muster);
    drucke(p0, muster, lm);
    exit(0);                                  /* alles in Ordnung */
}
```

Das Unterprogramm *suche* übernimmt die eigentliche Sucharbeit:

```
int
suche(pos, muster, l)                    /* sucht in Datei „liste“ */
    long pos;                                /* ab Position „pos“ */
    char *muster;                              /* das „muster“ */
    int l;                                      /* der Länge „l“ */
{
    int e, c;
    char puf[BLO];

    fseek(liste, pos, 0);                        /* Positionieren */
    fgets(puf, BLO, liste);                    /* Zeile überlesen */
    fgets(puf, BLO, liste);                       /* Zeile lesen */
    e = vgl(puf, muster, l);                    /* vergleichen */
    return e;                                     /* Ergebnis */
}
```

Das Unterprogramm *drucke* gibt die gefundenen Zeilen aus:

```
drucke(pos, muster, l)                  /* druckt aus Datei „liste“ */
    long pos;                               /* ab Position „pos“ */
    char *muster;                /* alle Zeilen mit Anfang „muster“ */
    int l;                                          /* (Länge l) */
{
    char puf[BLO];

    fseek (liste, pos, 0);
    fgets(puf, BLO, liste);                    /* Zeile überlesen */
    do {
        fgets(puf, BLO, liste);
    } while (vgl(puf, muster, l));       /* falsche Z. überlesen */
    do {
        fputs(puf, stdout);                 /* Zeilen ausdrucken */
        fgets(puf, BLO, liste);
    } while (!vgl(puf, muster, l));       /* solange sie passen */
}
```

Das Unterprogramm *vgl* untersucht, ob die Zeichenreihe *s1* kleiner, gleich oder größer als *s2* ist. Es werden *n* Zeichen betrachtet. Beim Aufruf in *suche* ist *s2* das Suchmuster.

```
int
vgl(s1, s2, n)                                 /* vergleicht „s1“ */
    char *s1, *s2;                            /* mit Muster „s2“ */
    int n;                                      /* der Laenge „n“ */
{
    int c1=0, c2=0;
    while (n--) {
        if (lex) while (*s1 && ! isalnum(*s1)) s1++;
        if (fol) c1 = tolower(*s1);
        else c1 = *s1;
```

```
        c2 = *s2;
        if (c1 != c2) break;
        s1++; s2++;
    }
    return c1 - c2;
}
```

Das Unterprogramm *fehler* druckt im Bedarfsfall eine Meldung, die auch den Kommandonamen umfaßt, und bricht das Programm ab.

```
fehler(r, f, s)
    int r;
    char *f, *s;
{                                                          /* druckt */
    fprintf(stderr, "%s: ", komname);/* den Kommandonamen, */
    fprintf(stderr, f, s);          /* Meldung "s" im Format "f" */
    fprintf(stderr, "\n");
    exit(r);                                          /* und bricht ab */
}
```

Beispiel B.6. Shell-K ommandoskript **`diff.sh`** zur Verarbeitung von FIFOs (benannten *Pipes*) durch *diff*:

diff (vgl. Abschnitt 3.2.5) will seine Eingabeströme mehrmals lesen; FIFOs müssen daher in temporäre Dateien kopiert werden. Dabei ist Verschiedenes zu berücksichtigen:

1. Die möglichen Optionen für *diff* müssen abgefragt werden; es könnte ja z. B. ein FIFO mit dem Namen „`-h`“ existieren, das sonst bei *diff -h* versehentlich mitkopiert würde. Das Kommando *getopt* trennt Optionen und Namensparameter durch „`--`“. Das Auftreten von „`--`“ wird in der Variablen *VGL* gemerkt, die anzeigt, daß alle Optionen verarbeitet und somit alle weiteren Parameter Vergleichsoperanden sind, die im Falle von FIFOs (erkennbar durch „*test -p*“ bzw. „*[-p]*“) kopiert werden müssen. Alle Parameter werden in *PAR* gesammelt.
2. Für die Zwischendatei wird ein unverwechselbarer Name benötigt, der mit Hilfe der Prozeßnummer (`$$`) in der Variablen *TMP* gebildet wird. Nach dem Kopieren wird in *PAR* der neue Name eingesetzt. Für den Fall, daß auch der zweite Operand ein FIFO ist, muß ein zweiter Name bereitgestellt werden.
3. Da bei zwei FIFOs nicht feststeht, in welcher Reihenfolge sie beschrieben werden, muß das Kopieren asynchron, also im Hintergrund, geschehen. Dafür sorgt die *Shell* durch das Zeichen „`&`“. Vor der Ausführung von *diff* muß durch *wait* das Ende der Kopierprozesse abgewartet werden.
4. Falls das Kommando durch ein Signal unterbrochen wird, müssen die Kopierprozesse beendet und die Hilfsdateien gelöscht werden. Daher ist es nötig, die Prozeßnummern (`$!`) und die Dateinamen für die Kommandos *kill* und *rm* zu speichern, wozu die Variablen *KILL* und *RM* dienen. Das Kommando *trap* sorgt für das Abfangen der Unterbrechungssignale.

diff.sh

```
VGL=                                          # Vergleichsobjekt?
PAR=                                          # Parameter für diff
TMP=/tmp/dif$$                                # Name f. temp. Datei
KILL=                                         # Parameter für kill
RM=                                           # Parameter für rm
trap 'kill $KILL; rm -f $RM; exit' \
    1 2 3                                     # Signale abf angen
for P in `getopt "efbh" $*`
do  if   [ "$P" = "--" ]                      # Optionen erledigt?
      then VGL="ja"
    elif [ "$VGL" = "ja" -a -p "$P" ]         # FIFO ?
      then cp $P $TMP &                       # K opieren, im HGr.
           KILL="$KILL $!"                    # für kill merk en
           P="$TMP"                           # Neuer Parameter
           RM="$RM $TMP"                      # zum Löschen merk en
           TMP=${TMP}2                        # neuer /tmp-Name
    fi
    PAR="$PAR $P"                             # P arameter sammeln
done
wait                                          # Kopieren abwarten
diff $PAR                                     # eigentliches diff
rm -f $RM                                     # Reste löschen
```

Beispiel B.7. Erzeugen von Verweisen in Texten mit *troff* und *m4*.

Will man mit dem Textformatierer *troff* symbolische Marken in einen Text setzen und sich in Querverweisen auf diese beziehen, so stört die Beschränkung der Registernamen auf eine Länge von zwei Zeichen sehr. Es liegt daher nahe, *troff* zu diesem Zweck mit dem Makro-Expandierer *m4* zu kombinieren.

Damit auch Voraus-Verweise möglich sind, müssen die Marken in einer Datei gesammelt werden, die beim nächsten Durchlauf vorgeschaltet wird (s. Abbildung 59). Wie im Abschnitt 5.1.3 beschrieben, kann *troff* nur auf Standard- und Fehlerkanal ausgeben. Benutzt man den Fehlerkanal zur Erzeugung der Makro-Definitionen, so sind Fehlermeldungen nicht mehr möglich. Vorteilhafter ist es, die

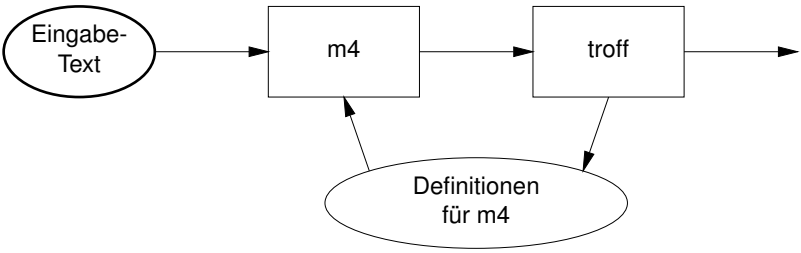


Abb. 59. Querverweise in *troff* mit *m4*

Definitionen als Kommentare (mit #) in die Standardausgabe einzustreuen und anschließend daraus zu extrahieren. Das Einfügen in die Ausgabe erfolgt durch die *troff*-Anweisung `\!`; dahinter folgt eine Kennung *m4*, die ein Extrahieren mit dem Kommando *grep* erlaubt. Eine solche Definitionszeile lautet also in der Eingabe an *troff* etwa:

```
\!#m4 define(marke_kapitel_troff,5)
```

Ein *troff*-Makro macht die Benutzung eleganter, besonders wenn in einem Register die jeweilige Kapitelnummer zur Verfügung steht. Abbildung 60 zeigt ein solches Makro *Ma*; das Register für die Kapitelnummer heiße *$n*. Das Kommando *grep* sucht die Definitionen aus der Ausgabe, *cut* entfernt das Kommentarsymbol # und die Kennzeichnung *m4*. Nach jeder Änderung sind natürlich zwei Durchläufe erforderlich, bis auch die geänderten Marken in der Ausgabe erscheinen.

```
.de Ma \" erzeugt eine m4-Definition
\!#m4\t'define(\\$1,\\*($n)dnl'
..
.Ma marke_kapitel_troff
```

Ergebnis:

```
#m4     define(marke_kapitel_troff,5)
```

Kommandos:

```
m4 marken text | troff ... >tr.aus
grep '^#m4' tr.aus | cut -f2- >marken
```

Abb. 60. *troff*-Makro für Querverweise

Eine Verbesserung der beschriebenen Technik bietet einen Schutz gegen falsch geschriebene Marken: Marken werden nicht durch sich selbst, sondern durch ein spezielles *m4*-Makro *_ref_* aufgerufen, das prüft, ob die Marke als Makro definiert ist. Zur Sicherheit gegen zufälliges Auftreten des Markennamens wird jeder Marke das Wort *mArKe* vorangestellt. Abbildung 61 zeigt die nötigen Definitionen. Im Aufruf von *ifdef* müssen sowohl die Marke als auch der Aufruf von *errprint* in Anführungszeichen eingeschlossen werden; sonst würde statt des Markennamens der Marken*wert* eingesetzt, und *errprint* würde bereits bei der Vereinbarung des Makros *_ref_* ausgeführt. Aus demselben Grund werden bei der Definition des *troff*-Makros *Ma* die Abstriche bei den Parametern *\\$1* und *\\$2* verdoppelt; auch sie sollen ja erst bei Ausführung, nicht bei Definition des Makros wirksam werden.

```
define(_ref_,`ifdef(`mArKe_$1',mArKe_$1,dnl
   `errprint(Marke $1 ist nicht definiert
)')')dnl
.de Ma \" erzeugt eine m4-Definition
\!#m4\t`define(mArKe_\\$1,\\*($n)dnl'
..
.Ma kapitel_troff
 ...
Im Abschnitt _ref_(kapitel_troff) ...
```

Ergebnis:

```
#m4     define(mArKe_kapitel_troff,5)
 ...
Im Abschnitt 5 ...
```

Abb. 61. Verbesserte Makros für Querverweise

Die beschriebene Technik ist sehr ähnlich der im Formatierer LaTeX verwendeten; da *troff* weder lange Makro-Namen kennt noch mehrere Ausgabedateien schreiben kann, müssen andere UNIX-Werkzeuge wie *m4*, *grep* und *cut* einspringen.

Glossar

Abschirmen
: Aufheben der Sonderbedeutung eines Steuerzeichens, durch Davorstellen eines dazu bestimmten anderen Steuerzeichens; UNIX-Programme verwenden häufig den → Abstrich.

Abstrich
: der umgekehrte Schrägstrich „\“; er wird häufig als Steuerzeichen, z. B. als Kommando-Einleitung oder zum Abschirmen (→ Zitieren) anderer Steuerzeichen verwendet.

Anführungszeichen
: 1. Zeichen in Form doppelter Hochkommas (Doppelapostroph, `"`)
  2. =→ Zitierklammer

Antwort-Kode
: → Ende-Status.

Ende-Status
: Eine ganze Zahl, die ein Prozeß bei seiner Beendigung an den aufrufenen Prozeß zurückgibt (engl.*r eturn code*). Diese Zahl gehört zur R-Schnittstelle. Der Ausdruck *Ende-Status* wird im System-V-Handbuch (1988) benutzt; Wolek (1990) (Abschnitt 5.3); verwendet die Bezeichnung *Antwortcode*.

Fehlerkanal
: auch Diagnosekanal: Der Ausgabekanal Nummer 2 eines UNIX-Programms, auf den normalerweise Fehler- und andere Meldungen geschrieben werden, die den Benutzer auch beim umgelenkter Standardausgabe (Kanal 1) direkt erreichen sollen.

FIFO
: *(first in, first out)* Ausdruck, der das Prinzip der Warteschlange kennzeichnet; in UNIX: vom System verwalteter Pufferspeicher, der die Ausgabe eines Programms mit der Eingabe eines anderen verbindet (benannte *Pipe*). Verfügt im Gegensatz zur einfachen *Pipe* über einen Dateiknoten (N-Schnittstelle) und einen Namen.

Fließband
: *(pipeline)* eine Kette von Kommandos, deren Standardausgabe über *Pipes* mit der Standardeingabe des jeweils nächsten verbunden ist.

Gegenschrägstrich
: = → Abstrich

Katalog
: 1. Liste von Meldungstexten in NLS (s. Abschnitt 6).
  2. =→ V erzeichnis.

Klient
: ein Programm, das von einer anderen, passiv wartenden Einheit (→ *Server*) aktiv einen Dienst in Anspruch nimmt; i. w. S. auch der Rechner, auf dem dieses Programm läuft.

Metazeichen
: Steuerzeichen in regulären Ausdrücken, das nicht für sich selbst, sondern für andere Zeichen steht.

Neue Zeile, `NL`
: (vom engl. *new line*, DIN 66 003) Steuerzeichen, das am Sichtgerät oder Drucker einen Vorschub um eine Zeile und einen Wagenrücklauf bewirkt; in UNIX dargestellt durch das Zeichen *LF* mit der ASCII-Nummer 10, in C dargestellt durch *\n*. UNIX-Treiberprogramme ergänzen bei der Ausgabe einen Wagenrücklauf (`CR`, *\r*), so daß am Anfang der nächsten Zeile weitergeschrieben wird. *NL* wird v on vielen UNIX-Programmen als Trenner von Datensätzen (Zeilen) verwendet.

Pipe
: engl. Wort für eine Verbindung zwischen zwei Prozessen, bei der die Ausgabe des ersten zur Eingabe des zweiten wird. Vgl. Abschnitt 2.4.5. Benannte *Pipe (named pipe)*: = FIFO.

Server
: ein Programm oder Gerät, das passiv eine Dienstleistung anbietet, indem es auf einen Auftrag eines → Klienten wartet. Beispiele sind Datenbank-*Server* oder *Server* für das grafische Fenster-System X.

Shell
: engl. Ausdruck für einen UNIX-Kommandointerpretierer, ein Programm, das Eingaben des Benutzers in Kommando-Aufrufe umsetzt. Vgl. Abschnitt 3.1.

Steuerzeichen
: Zeichen, das die Übertragung oder die Verarbeitung von Zeichen beeinflußt (DIN 44 300, Nr. 2.2.08).

Trenner
: Sammelbegriff für → Trennzeichen und → Trennraum.

Trennzeichen, Informationstrennzeichen
: ein Steuerzeichen, das eine Folge von Zeichen begrifflich gliedert (DIN 44 300, Nr. 2.2.07), also z. B. gleichrangige Strukturelemente (Felder, Datensätze) in einem Datenstrom trennt. Das Trennzeichen gehört nicht zu den angrenzenden Strukturen. Zwei aufeinanderfolgende Trennzeichen begrenzen ein leeres Feld bzw. einen leeren Datensatz.

Trennraum
: Folge von einem oder mehreren Zeichen, von denen jedes als Trenner dienen könnte, die aber zusammen nur einen Trenner darstellen. Wenn eine Struktur Trenn*raum* statt Trenn*zeichen* benutzt, kann sie keine leeren Felder enthalten. Meist besteht Trennraum aus Leerzeichen und Tabulatoren; man spricht dann auch von *Leerraum* oder *Zwischenraum*.

Verzeichnis, Dateiverzeichnis
: UNIX-Datei, die aus Verweisen auf andere Dateien besteht; Nicht-Blatt im Dateibaum.

Zitieren
: Anführen eines Textstückes zur „wörtlichen“, nicht interpretierenden Übernahme. Zitieren kann erfolgen durch Abschirmen aller Steuerzeichen durch ein besonderes Steuerzeichen (in UNIX meist der Abstrich). In vielen Schnittstellen existieren aber auch → Zitierklammern (Anführungszeichen), die die Sonderbedeutungen aller geklammerten Zeichen aufheben. Vgl. Abschnitt 2.4.2.

Zitierklammer
: auch *Zitat*klammer: Jedes von zwei (gleichen oder unterschiedlichen) Zeichen (oder Zeichenfolgen), die Sonderbedeutungen der Zeichen aufheben, die von den beiden Zitierklammern eingeschlossen (geklammert) sind, die also die eingeschlossene Zeichenreihe → zitieren; = *Anführungszeichen 2*. Bei Verwendung von Zitierklammern ergibt sich das Problem, wie ein Text zitiert wird, der eine (rechte) Zitierklammer enthält. Es kann durch Verdoppeln oder Abschirmen der Klammer oder gar nicht gelöst sein.

Literatur

Aho AV, Kernighan BW, and Weinberger PJ, *Awk — A pattern scanning and processing language*.

Baber RL, *Softwarereflexionen — Ideen und Konzepte für die Praxis,* Springer-Verlag, Berlin, 1986.

Bach F, Domann P, and Weng-Beckmann U, *UNIX. Handbuch zur Programmierung,* Carl Hanser Verlag, München, 1987.

Banahan M and Rutter A,*UNIX lernen, ver stehen, anwenden,* Carl Hanser Verlag, München, 1984.

Bauer FL, *Informatik I,* HTB 80, Springer-Verlag, Berlin, 1973.

Bourne SR, *Das UNIX-System,* Addison-Wesley, Bonn, 1985.

Buschlinger E, *Software-Entwicklung mit UNIX,* Teubner, Stuttgart, 1985.

Corbin J and Silveri C, Open network programming, *UNIX WORLD*, Nr. 1989/12, S. 115–128.

Detering R, *UNIX-Handbuch,* Sybex-Verlag, Düsseldorf, 1984.

DIN 44 300, *Informationsverarbeitung, Begriffe,* Deutsches Institut für Normung.

DIN 66 003, *Informationsverarbeitung, 7-Bit-Code,* Deutsches Institut für Normung.

DIN 66 303, *Informationsverarbeitung, 8-Bit-Code,* Deutsches Institut für Normung.

Foxley E, *UNIX für Super-User,* Addison-Wesley, Bonn, 1988.

Fößmeier R, Verarbeitung deutscher Texte ohne NLS, *GUUG-Nachrichten*, Nr. 14, S. 9–15, Juni 1988.

Fößmeier R, Better and faster calendar (in: Wizard's Grabbag, R Thomas), *UNIX/World*, Nr. 9/1989, S. 115–116.

Gulbins J, *UNIX. Eine Einführung in Begriffe und Kommandos von UNIX – Version 7, bis System V.3,* Springer-Verlag, Berlin, 3. Aufl. 1988.

Jänsch Chr, Rüde U, and Schnepper K, Macro expansion, a tool for the systematic development of scientific software, TUM-I8814, Technische Universität München, München, 1988.

Johnson SC, Yacc: yet another compiler-compiler, Comp. Sci. Tech. Rep. Nr. 32, Bell Laboratories, Murray Hill, New Jersey, Juli 1975. (bei Kernighan u. McIlroy, 1978)

Kernighan B and Pike R, *Der UNIX-Werkzeugkasten,* Carl Hanser Verlag, München, 1987.

Kernighan BW and McIlroy MD, *UNIX programmer's manual,* Bell Laboratories, 7th edition 1978.

Kühn E and Schikuta E, *UNIX FORUM III – 87,* Österreichische Computer-Gesellschaft, Wien, 1987.

Lesk ME, Lex — A lexical analyzer generator, Comp. Sci. Tech. Rep. Nr. 39, Bell Laboratories, Murray Hill, New Jersey, Oktober 1975. (bei Kernighan u. McIlroy, 1978)

Lesk ME, Tbl — a program to format tables, Bericht Nr. 49, Bell Laboratories, 1976. (in der Troff-Beschreibung)

Leut J, Mein UNIX kommt mir spanisch vor, *unix/mail*, Nr. 2/90, S. 63–65.

Palmer JF and Morse SP, *Die mathematischen Grundlagen der Numerik-Prozessoren 8087/80287,* te-wi Verlag, München, 1985.

Peterson JL and Silberschatz A, *Operating system concepts,* Addison-Wesley, Reading, 1985.

Probst A, *Ami-Deutsch,* Fischer-Taschenbuch-Verlag, Frankfurt, 1989.

Rüde U and Zenger Chr, *A workbench for multigrid methods,*Institut f. Informatik, T echn. Univ. München, 1986.

Roddy KP, *UNIX nroff/troff user's guide,* Holt, Rinehart and Winston, New York, 1987.

Schreiner AT and Friedman HG, *Compiler bauen mit UNIX,* Carl Hanser Verlag, München, 1985.

UNIX System V user's reference manual, deutsche Ausgabe, Prentice Hall, (Carl Hanser, München), 1988.

Wolek H, *OS/2. Einführung,* Oldenbourg-Verlag, München, 1990.

Tabellenverzeichnis

Sachverzeichnis*

* Kursive Seitenzahlen verweisen auf Definitionen oder ausführliche Beschreibungen von Kommandos.

Fößmeier Die Schnittstellen von UNIX-Programmen

Dieses Buch vermittelt die grundlegende Philosophie der Problemlösung im Betriebssystem UNIX. UNIX bietet dem Benutzer eine gut durchdachte und bewährte Grundmenge universell verwendbarer Werkzeuge, durch deren Kombination viele Probleme mit sehr geringem eigenem Programmieraufwand gelöst werden können. Zugleich sind die so entstehenden Lösungen meist übersichtlich und gut portabel.

Während übliche Bücher über UNIX nach Werkzeugen gegliedert sind, ist dieses an den Schnittstellen orientiert, die Werkzeuge verbinden. Dieser mehr datenorientierte Zugang entspricht der Vorgehensweise der modernen Programmiertechnik und hilft dem Programmierer, bei neuen Problemen bekannte Strukturen wiederzuerkennen und die geeigneten Werkzeuge zu ihrer Bearbeitung auszuwählen. Auch der erfahrene UNIX-Kenner kann von der hier gebotenen neuen Sichtweise profitieren.

Eine große Zahl von Beispielen erleichtert das Verständnis. Tabellen, die Wissenswertes über Schnittstellen in übersichtlicher Form zusammenstellen, machen das Buch auch zu einem interessanten Nachschlagewerk.