robotron

Systemhandbuch

MUTOS – 8000

Teil II Abschnitt 1

Kommandointerpreter Shell

robotron

Programmtechnische Beschreibung
fuer Buerocomputer A5120.16

SYSTEMHANDBUCH

MUTOS 8000

Teil II

Abschnitt 1 - Kommandointerpreter Shell

VEB Robotron-Buchungsmaschinenwerk
Karl-Marx-Stadt
Stand: 12/85

Die vorliegende Dokumentation entspricht dem Stand 12/85.
Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Im Interesse einer staendigen Weiterentwicklung werden alle Leser gebeten, ihre Vorschlaege bzw. Hinweise dem
VEB Robotron-Buchungsmaschinenwerk
9010 Karl-Marx-Stadt
Annabergerstrasse 93
mitzuteilen.

Fuer das Betriebssystem MUTOS 8000 wurden folgende Dokumentationen erarbeitet:

Anwendungsbeschreibung MUTOS 8000

Systemhandbuch MUTOS 8000 Teil I
Abschnitt 1 Kommandos
Abschnitt 2,3 Systemrufe, Bibliotheksfunktionen
Abschnitt 4, 5, 7, 8 Special Files, Fileformate, Makropakete, Systemunterstuetzung

Systemhandbuch MUTOS 8000 Teil II
Abschnitt 1 Kommandointerpreter Shell
Abschnitt 2 Editor
Abschnitt 3 Testhilfsprogramm adb

Systemhandbuch MUTOS 8000 Teil III
Abschnitt 1 Textverarbeitung nroff

Sprachbeschreibungen
Assemblersprachbeschreibung U 8000
Sprachbeschreibung C-Sprache

Das Sachwortverzeichnis ist als Anlage in der Anwenderbeschreibung enthalten.

Zur Programmentwicklung und Programmtestungen fuer den Prozessor U 8000 werden zusaetzlich folgende Dokumentationen angeboten:
Monitortestsystem MON8000

CROSS-Software U8000 zum Betriebssystem UDOS
Assembler U8000ASM
Binder ZLINK
Absolutbinder IMAGER
Preprozessor INCLUDE

Inhaltsverzeichnis

Kommandointerpreter Shell

1. Einfuehrung

Die Kommunikation der meisten Nutzer mit dem Betriebssystem MUTOS 8000 erfolgt ueber die Kommandosprache Shell. Der Gesamtumfang der Kommandosprache ist relativ gross. Dafuer bietet Shell aber vielfaeltige Moeglichkeiten, dem Nutzer die Arbeit unter MUTOS 8000 zu erleichtern.

Shell realisiert u.a.:

- verschiedene Elemente, wie sie aus algorithmischen Sprachen bekannt sind:
  - Programmablauf - Steuerkonstrukte (for, case, if, while)
  - Variable
  - Parameteruebergabe
- Kommando- und Parametersubstitution
- Filenamen-Generierung
- Umlagerung der E/A-Richtungen
- Modifikation der Umgebung, in der ein Prozess ablaeuft
- Signalbehandlung
- Datenuebergabe zwischen verschiedenen Prozessen ueber Pipes
- Suchen von Files entlang vom Nutzer definierter Wege innerhalb des MUTOS 8000 - Filesystems.

Die Shell-Syntax zeichnet sich durch eine knappe Notation aus, die aus leicht verstaendlichen und einpraegsamen Zeichen aufgebaut ist. Unter Ausnutzung der anwenderfreundlichen Eigenschaften von MUTOS 8000, wie der Transparenz des Filesystems, der Dienstprogramme, die vom Standard-Eingabefile stdin lesen und zum Standard-Ausgabefile stdout ausgeben sowie des Pipe-Konzeptes, bietet Shell dem Nutzer bei wenig Eingabearbeit viel an Systemleistung. Dabei unterstuetzt der Kommandointerpreter sowohl die interaktive Arbeit, d.h. die Eingabe einzelner Kommandozeilen ueber die Tastatur, als auch das Abarbeiten von Shell-Prozeduren, d.h. von Files, die Shell-Kommandos enthalten. Beide Verarbeitungsarten werden im wesentlichen gleich behandelt.

Das Erlernen der Shell-Programmierung wird dem Nutzer dadurch erleichtert, dass es zu Beginn genuegt, wenige Elemente der Kommandosprache und die Wirkungsweise einiger MUTOS 8000 - Dienstprogramme zu kennen und spaeter bei Bedarf weitere Moeglichkeiten von Shell hinzuzulernen.

Dem Rechnung tragend ist die vorliegende Schrift so aufgebaut, dass im zweiten Kapitel vornehmlich auf solche Bestandteile von Shell eingegangen wird, die man als Grundelemente des interaktiven Verkehrs bezeichnen kann. Das dritte Kapitel stellt Moeglichkeiten vor, die besonders die Arbeit mit Shell-Prozeduren unterstuetzen. Im vierten Abschnitt werden weiterfuehrende und zur Ergaenzung der beiden ersten Kapitel notwendige Bestandteile von Shell erlaeutert. Der Anhang gibt schliesslich einen zusammenfassenden Ueberblick der gesamten Shell-Grammatik.

Bemerkung:

Die Kommandosprache ist durch einen Interpreter implementiert (/bin/sh). Dieser hat keinen besonderen Status innerhalb der Bestandteile von MUTOS 8000 und ist genauso auslagerbar wie alle anderen Dienstprogramme. Es ist auch moeglich, anstelle von Shell fuer einzelne Nutzer ein anderes Programm als Kommandointerpreter arbeiten zu lassen. Dazu ist lediglich eine Aenderung des Eintrags fuer den betreffenden Nutzer im Password-File (/etc/passwd) erforderlich. In einer bestimmten Umgebung (z.B. zur Datenerfassung) koennte der Editor die Stelle von Shell einnehmen.

2. Grundlagen der interaktiven Shell-Arbeit

2.1. Einfache Kommandos

In der einfachsten Form besteht ein Kommando aus einer Folge von Worten, die durch Leerzeichen getrennt sind. Das erste dieser Worte ist der 'Name' des Kommandos, d.h. eines Files, das ein ausfuehrbares Programm enthaelt. Die verbleibenden Worte werden durch Shell als Argumente an das auszufuehrende Programm weitergegeben. Das Kommando wird durch 'Newline' abgeschlossen und dann von Shell interpretiert. Unter 'Newline' wird das durch Betaetigen der Taste <ENTER> ausgeloeste Zeichen verstanden. So wird nach Eingabe der Kommandozeile

```
comname arg1 arg2 ... argn
```

ein File mit dem Namen comname gesucht und ueber einen **execute** - Systemruf dessen Ausfuehrung eingeleitet, wobei die Argumente arg1, arg2, ... argn als Parameter des **execute**-Systemrufes an das auszufuehrende Programm uebergeben werden.

Ist comname ein Pfadname, der nicht mit '/' beginnt, in dem aber mindestens einmal das Zeichen '/' enthalten ist, so wird ein einmaliger Versuch unternommen, das File comname vom aktuellen Directory aus ueber den angegebenen Pfadnamen zu erreichen. Beginnt comname mit '/', so wird der Pfadname als vollstaendig bezeichnet, und die Suche nach dem auszufuehrenden Programm beginnt in der Wurzel (root) des gesamten Filesystems.

Ist comname nur ein einfacher Name, in dem kein Zeichen'/' enthalten ist, so sucht Shell ein File mit diesem Namen entlang eines Suchpfades, der vom Nutzer veraendert werden kann (siehe *PATH in 3.1.). Als Standard wird versucht, das File comname zuerst im aktuellen Directory zu finden, falls erfolglos, dann in /bin und schliesslich in /usr/bin. Die beiden letztgenannten Directories enthalten normalerweise die MUTOS 8000-Dienstprogramme .

2.2. Hintergrund-Kommandos

Zur Ausfuehrung eines Kommandos wird von Shell i.a. ein neuer Prozess eroeffnet, auf dessen Beendigung, d.h. das Ende der Kommandoabarbeitung, dann gewartet wird. Solange die Kommandoabarbeitung andauert, ist fuer den Nutzer keine Interaktion mit Shell moeglich. Wird ein Kommando jedoch mit dem Operator '&' abgeschlossen, so wird der Wartemechanismus fuer dieses Hintergrund-Kommando ausser Kraft gesetzt, und die interaktive Arbeit kann unmittelbar nach dem Eingeben des Kommandos fortgesetzt werden. Die Eingabe der Kommandozeile

```
cc source.c &
```

bewirkt damit z.B, dass der Nutzer waehrend der Uebersetzung des C-Programms source.c andere Arbeiten an seinem Geraet erledigen kann.

2.3. Umlagerung der E/A-Richtungen

Zu jedem Prozess unterhaelt MUTOS 8000 eine Menge von Filedeskriptoren 0, 1, 2, ..., die der Identifizierung von Files waehrend der Durchfuehrung von E/A-Operationen dienen. Nachdem ein Nutzer seine Arbeit unter MUTOS 8000 begonnen hat (login), sind fuer seinen laufenden Shell-Prozess drei Files, denen die Deskriptoren 0, 1 und 2 zugeordnet sind, geoeffnet. Das File, dem der Deskriptor 0 (stdin) zugeordnet ist, wird als Standard-Input, das File, dem der Deskriptor 1 (stdout) zugeordnet ist, wird als Standard-Output und das File, dem der Deskriptor 2 (stderr) zugeordnet ist, wird als Standard-Nachrichten-File bezeichnet. Die drei geoeffneten Files entsprechen zu Beginn der Arbeit Tastatur und Bildschirm (Spezial-File /dev/tty1).

Die meisten MUTOS 8000-Dienstprogramme sind so beschaffen, dass sie Eingaben von stdin lesen und Ausgaben nach stdout liefern. Um Ein-/Ausgaben von/nach einem anderen Geraet bzw. File als dem Terminal zu realisieren, kann mit Hilfe der Umlagerungszeichen `<', `>', `<<' und `>>' eine Aenderung der E/A-Richtung fuer die Dauer der Ausfuehrung eines Kommandos erreicht werden. Die Umlagerungszeichen werden mit den ihnen nachgestellten Filenamen von Shell selbst interpretiert und nicht als Argumente an die aufgerufenen Programme uebergeben. Dies wird an den MUTOS 8000 - Dienstprogramme cat und pr in den folgenden Beispielen gezeigt. Die folgenden Kommandos bewirken somit:

`cat text`	(ohne Umlagerung) Ausgabe des Files text auf Bildschirmterminal
`cat text >outtext`	Ausgabe von text in das File outtext. Falls outtext noch nicht existierte, wird dieses File durch Shell erstellt. Ansonsten wird der alte Inhalt ueberschrieben.

Da die Ein- und Ausgaben zu peripheren Geraeten (repraesentiert durch die zugehoerigen `Spezial Files') in MUTOS 8000 genauso behandelt werden wie zu gewoehnlichen Files, laesst sich durch

```
pr text >/dev/lp
```

die Ausgabe von Text auf dem Zeilendrucker erreichen. Das Anfuegen von Daten an ein bereits existierendes File wird durch

```
cat text >> oldfile
```

erreicht. Die Eingaberichtung kann mit `<' umgelagert werden. Der Deskriptor stderr ist von diesen E/A-Umlagerungen nicht betroffen. Die Benutzung von `<<' wird im Abschnitt 3.3. naeher erlaeutert.

2.4. Pipelines und Filter

Das Bereitstellen des Pipe-Mechanismus durch MUTOS 8000 und die einfache Handhabung, die Shell dazu bietet, ermoeglichen auf unkomplizierte Weise die Kombination von MUTOS 8000 - Dienstprogrammen untereinander und mit Anwenderprogrammen.

Das Verknuepfen von Kommandos durch den Pipe-Operator`|' wie in

```
anycommand |  sort | pr
```

bewirkt, dass die Kommandos der so geschaffenen 'Pipeline' simultan abgearbeitet werden, wobei der Standard-Output jedes Kommandos als Standard-Input des folgenden Kommandos benutzt wird.
Damit realisiert die oben angegebene Kommandozeile, dass die Ausgaben des (Anwender-) Programms anycommand mit Hilfe des MUTOS 8000 - Dienstprogramms sort sortiert und anschliessend entsprechend pr formatiert ausgegeben werden. Ohne Benutzung des Pipe-Operators muessten zum Erreichen des gleichen Ergebnisses folgende Kommandos abgearbeitet werden:

```
anycommand > temp1
sort       < temp1 >temp2
pr         < temp2
rm  temp1  temp2
```

Der Shell-Pipe-Operator ist ueber den Systemruf pipe(2) implementiert. Dementsprechend erfolgt auch die Synchronisation der Ablaeufe in Pipelines durch den Betriebssystemkern und nicht durch Shell. Es ist zu beachten, dass bei MUTOS 8000 maximal nur 3 Kommandos in einer Pipeline angeordnet werden koennen.

Programme wie sort(1) , die von stdin lesen, die eingelesenen Daten in irgendeiner Weise bearbeiten und das Ergebnis dieser Bearbeitung (nach stdout) ausgeben, werden 'Filter' genannt. Eine Reihe der von MUTOS 8000 zur Verfuegung gestellten Dienstprogramme sind derartige Filter.

2.5. Filenamen-Generierung

Vielen MUTOS 8000 - Dienstprogrammen werden neben den Optionen zur Auswahl bestimmter Teilfunktionen des Kommandos vor allem Filenamen als Argumente uebergeben. Um die Arbeit mit den Filenamen effektiv zu gestalten, d.h. um die noetige Schreibarbeit zu verringern und gleichzeitig einen Suchmechanismus nach bestimmten Filenamen zur Verfuegung zu haben, werden von Shell die Filenamen - Erweiterungszeichen '*', '?' und '[...]' unterstuetzt. Wird ein Filename innerhalb eines Kommandos nicht vollstaendig, sondern in Form eines Musters angegeben, gebildet unter Benutzung der Filenamen-Erweiterungszeichen, so ermittelt Shell alle dem Muster entsprechenden Filenamen und setzt diese, alphabetisch geordnet, als einzelne Argumente anstelle des Musters in die Kommandozeile ein. Das Zeichen '*' steht dabei fuer jede beliebige Zeichenkette,'?' fuer ein beliebiges Zeichen, und anstelle von '[...]' kann im zu ermittelnden Filenamen irgendeines der in den Klammern angebenen Zeichen auftreten.

Daher listet das Kommando

```
ls -l *.c
```

alle Filenamen im aktuellen Directory, die mit .c' enden (in einer gemaess der Option -l 'langen' Version der Liste) und

```
ls -l /dev/rk?
```

liefert die Namen aller Disketten-Laufwerke, fuer die in **/dev** ein Eintrag vorhanden ist.

Werden in **[...]** zwei Zeichen durch ein Minus-Zeichen verbunden, so entsprechen diesem Muster alle Zeichen, die lexikographisch zwischen dem so gebildeten Paar liegen. Dementsprechend erhaelt man durch

```
ls -l /bin/[m-p]*
```

die Namen aller MUTOS 8000-Dienstprogramme unter /bin, die mit den Buchstaben m, n, o und p beginnen. Filenamen, die mit einem `.' beginnen, werden bei der Filenamen-Generierung nach solchen Mustern nicht mit erfasst. Der `.' muss in einer entsprechenden Kommandozeile explizit angegeben werden.

Das MUTOS 8000 - Dienstprogramm echo gibt seine Argumente, durch Leerzeichen getrennt, auf Standard-Output aus. Durch

```
echo .*
```

erhaelt man die Namen aller Files im aktuellen Directory, die mit `.' beginnen, waehrend

```
echo *
```

gerade die Namen aller anderen Files in diesem Directory liefert.

Diese Sonderbehandlung des Punktes als erstes Zeichen im Filenamen wird durchgefuehrt, um das Miterfassen der Namen `.' (fuer das aktuelle Directory selbst) und `..' (fuer sein Parent-Directory) bei der Filenamengenerierung zu vermeiden.

2.6. Apostrophieren

Die Zeichen, die eine besondere Bedeutung fuer Shell haben, werden Metazeichen genannt. Neben den bisher schon erwaehnten Zeichen <, >, *, ?, |, & gibt es noch weitere Metazeichen, die in den folgenden Kapiteln erlaeutert werden. Im Anhang befindet sich eine Zusammenstellung aller Metazeichen.
Soll eines der Metazeichen in einer Kommandozeile ohne seine besondere Bedeutung fuer Shell benutzt werden (z.B. in

Filenamen), so muss dieses Zeichen apostrophiert werden, indem ein '\' vorangestellt wird.
Das '\' wird von Shell aus der Kommandozeile entfernt, so dass

```
echo \*
```

als Ausgabe '*' liefert. Das Zeichen '\' kann auch selbst apostrophiert werden.

```
echo \\
```

liefert somit '\'.

Auch das'Newline'-Zeichen (Taste ENTER) kann auf diese Weise apostrophiert werden, so dass die Folge '\Newline' von Shell nicht als Kommandozeilenabschluss erkannt wird. Lange Kommandozeilen koennen sich auf diese Weise ueber mehrere Bildschirmzeilen erstrecken.

Das umstaendliche und fehleranfaellige Apostrophieren laengerer Zeichenketten nach der soeben geschilderten Methode ist nicht notwendig. Eine Zeichenkette kann, um die Bearbeitung darin enthaltener Metazeichen durch Shell zu vermeiden, in Apostrophe eingeschlossen werden.

```
echo '???'
```

ergibt

```
???
```

Die apostrophierte Zeichenkette kann 'Newlines' enthalten, die analog zum Apostrophieren mit '\' nicht als Zeilenende-Zeichen interpretiert werden. Apostrophe selbst duerfen in der so apostrophischen Zeichenkette nicht enthalten sein.

Schliesslich existiert noch ein dritter Mechanismus zum Apostrophieren, der mit Anfuehrungsstrichen arbeitet und nicht alle eingeschlossenen Shell-Metazeichen von der Behandlung durch Shell ausschliesst. Die Erlaeuterungen dazu finden sich im Abschnitt 4.5.

2.7. Beginn der Shell-Arbeit

Der Kommandointerpreter beginnt seine Arbeit unmittelbar nachdem sich ein Nutzer am Bildschirm eingetragen hat (login(1)). Falls im 'Home-Directory' dieses Nutzers ein File mit dem Namen **.profile** enthalten ist, wird zunaechst dieses File von Shell eingelesen. Es wird als Shell-Prozedur (siehe Kapitel 3) interpretiert, d.h. **.profile** sollte Kommandozeilen enthalten, von denen der Nutzer moechte, dass sie jedesmal vor Beginn seiner Arbeit am Bildschirmterminal ausgefuehrt werden. Ein einfaches Beispiel

eines solchen **.profile** koennte folgendermassen aussehen:

```
stty length 23 width 80
echo 'Aktuelles Datum und Zeit:'
date
PS2='weiter: '
```

Mit dem Kommando stty(1) lassen sich bestimmte Terminalfunktionen setzen. In unserem Fall wird die Laenge einer Bildschirmseite auf 23 Zeilen und die Zeilen auf 80 Spalten gesetzt. Alle Ausgaben auf dem Bildschirm erscheinen in Seiten zu 23 Zeilen in einer Breite von 80 Zeichen. Die jeweils naechste Seite wird nach Betaetigen der Leertaste ausgegeben. Das Kommando echo(1) liefert als Ausgabe auf dem Bildschirm die ihm als Argument uebergebene Zeichenkette. Von date(1) wird anschliessend das aktuelle Datum und die Uhrzeit ausgegeben.

Ist Shell bereit, von der Tastatur eine Kommandozeile einzulesen, so zeigt es dem Nutzer diese Bereitschaft durch Ausgabe von '$' an. Falls Shell nach Einlesen eines 'Newline'-Zeichens feststellt, dass die vorangegangenen Eingaben noch nicht vollstaendig sind und zur Abarbeitung des gegebenen Kommandos weitere Eingaben benoetigt werden (z.B. wurde der abschliessende Apostroph fuer eine Zeichenkette vergessen), so teilt es dies dem Nutzer durch Ausgabe von '>' auf dem Bildschirm mit. Beide Zeichenketten (Prompt-Zeichenkette) kann der Nutzer je nach seinem Geschmack veraendern. In unserem Beispiel fuer ein '**.profile**' geschah dies fuer die zweite Prompt-Zeichenkette, so dass anstelle von '>' auf dem Bildschirm dieses Nutzers '**weiter:**' erscheinen wuerde. Analog liesse sich der erstgenannte Prompt-String zum Beispiel durch

```
PS1='fertig: '
```

veraendern.

3. Die Arbeit mit Shell-Prozeduren

Der Kommandointerpreter Shell kann selbst in einem Kommando aufgerufen werden. Damit besteht die Moeglichkeit (nicht interaktiv), Kommandos abzuarbeiten, die in einem File, einer sogenannten Shell-Prozedur, enthalten sind. Ein Aufruf der Form

```
sh options procname arg1 arg2 ...
```

bewirkt, dass die Kommandos, die in der Shell-Prozedur procname enthalten sind, nacheinander gelesen und ausgefuehrt werden. Die Argumente arg1, arg2, ... sind dabei in der angegebenen Reihenfolge den Stellungsparametern $1, $2, ... zugeordnet und unter diesen Namen in der Shell-Prozedur verfuegbar. Das Kommando sh kann unter Wirkung einer ganzen Reihe von options (Schalter) ausgefuehrt werden, die zum Beispiel Mechanismen zur Kontrolle der Abarbeitung der Shell-

Prozedur bereitstellen (siehe Abschnitt 3.5. sowie sh(1) fuer die vollstaendige Beschreibung aller Optionen). Enthaelt das File compil z.B. die Kommandos

```
cc *1
mv a.out outprog
outprog
```

so fuehrt die Eingabe der Kommandozeile

```
sh compil test.c
```

zum sukzessiven Abarbeiten der Kommandos in der Shell-Prozedur compil, wobei anstelle von *1 test.c eingesetzt wird, d.h. das C-Programm test.c wird uebersetzt (einschliesslich Laden), wobei das ausfuehrbare Programm a.out entsteht. Dieses wird durch das Kommando mv in outprog umbenannt und anschliessend abgearbeitet.

Falls dem File compile (mit dem Kommando chmod) das Attribut 'ausfuehrbar' (executable) zugewiesen wurde, so kann diese Shell-Prozedur ohne Aufruf des Kommandointerpreters sh abgearbeitet werden. Die Kommandozeile

```
compil test.c
```

hat dann die gleiche Wirkung wie oben beschrieben.

Shell-Prozeduren sind ein wirksames Mittel zum Vereinfachen der Arbeit am Rechner. Da sie keiner Compilation bedueerfen, sind sie leicht zu erstellen und zu verwalten. Mit Hilfe der Shell-Optionen wird dem Nutzer das Testen seiner Prozeduren erleichtert. Der Standard-Input und der Standard-Output einer Shell-Prozedur bleiben waehrend der Abarbeitung der Prozedur unveraendert. Damit koennen solche Prozeduren als Filter benutzt werden.

3.1. Shell-Variable

Shell bietet die Moeglichkeit, Variablen sowohl in Prozeduren als auch waehrend der interaktiven Arbeit, Zeichenketten zuzuweisen. Auf diese Weise laesst sich z.B. das Schreiben haeufig benutzter Namen verkuerzen. So weist die Eingabezeile

```
source=/usr/sys/cmd
```

der Shell - Variablen source den Wert **/usr/sys/cmd** zu. In einem folgenden Kommando kann diese Zeichenkette ueber die Variablensubstitution *source, also durch Voranstellen eines '*' vor den Namen der Variablen, erreicht werden. Das Kommando

```
echo *source
```

liefert dann als Ausgabe den Standard-Output

```
/usr/sys/cmd.
```

Eine leere Zeichenkette wird einer Variablen zum Beispiel durch

```
empty=
```

zugewiesen.

Variablennamen muessen mit einem Buchstaben beginnen und koennen sonst aus Buchstaben, Ziffern und dem Unterstrich bestehen. Shell-Variable koennen innerhalb von Kommandozeilen in eine Zeichenkette eingefuegt werden. Zur Begrenzung des Variablen- (oder Parameter-) Namens dienen geschweifte Klammern. Im Beispiel

```
file=/mnt/bwk/bin/
cp a.out ${file}mutos
```

wird das File a.out aus dem aktuellen Directory nach **/mnt/bwk/bin/mutos** kopiert. Haette die zweite Zeile die Gestalt,

```
cp a.out $filemutos
```

so wuerde Shell versuchen, als zweites Argument den Wert der Variablen filemutos einzusetzen und diesen an cp zu uebergeben.

Durch Shell werden dem Nutzer spezielle Parameter bzw. Variablen zur Verfuegung gestellt, deren Bedeutung kurz erlaeutert werden soll.

Unter **$0** ist der Wert des nullten Arguments des execute-Systemrufes verfuegbar, der die Ausfuehrung des laufenden Prozesses bewirkte. Dies ist i.a. der Name des auszufuehrenden Programms selbst.

$*

steht, ebenso wie **$@**, fuer die Folge aller Stellungsparameter ab **$1**. Ein Unterschied zwischen beiden Parametern besteht beim Apostrophieren mit Anfuehrungszeichen (siehe 4.5.). Die Wirkung von "$*" ist gleich "$1 $2 ..." waehrend gilt: "$@" ist gleich "$1" "$2" ... .

$#

gibt (dezimal) die Anzahl der Stellung parameter einer Shell-Prozedur an.

$-

liefert die fuer den laufenden Shell-Prozess gesetzten Optionen (z.B. -x, -v).

$$

liefert den Prozessidentifikator des gerade laufenden Prozesses (dezimal). Jeder Prozess hat eine ihm vom Betriebssystem eindeutig zugeordnete Prozessnummer. Diese Eindeutigkeit ermoeglicht z.B. die Benutzung der Prozessnummer bei der Bildung von Namen fuer temporaer benoetigte Files. Damit wird erreicht, dass dieser Name einmalig ist, auch wenn das Programm in mehreren Prozessen gleichzeitig abgearbeitet wird. Dieses Prinzip wird auch von vielen MUTOS 8000 - Dienstprogrammen benutzt. Der Inhalt eines Directory aesst sich z.B. folgendermassen in einem temporaeren File zwischenspeichern:

```
ls      )/tmp/dir$$
...
rm      /tmp/dir$$.
```

$?

liefert als dezimalen Wert den Rueckkehrwert (Exit-Status) des zuletzt abgearbeiteten Kommandos. Bei den meisten Kommandos ist dieser Wert gleich Null, wenn das Kommando erfolgreich abgearbeitet wurde, und ungleich Null beim Auftreten eines Fehlers. Diese Rueckkehrwerte dienen als Testgroessen in **if**- und **while**-Konstrukten, wie in den Abschnitten 3.2.1. und 3.2.2. beschrieben.

$!

liefert den dezimalen Prozessidentifikator des letzten Prozesses, der im Hintergrund gestartet wurde.

Neben diesen durch Shell mit Werten belegten Variablen gibt es weitere, die eine besondere Bedeutung fuer Shell haben. Sie sollten nur in der fuer sie vorgesehenen Weise benutzt werden.

$HOME

Der Wert der Variablen HOME stellt den Pfadnamen des 'Home-Directorys' eines Nutzers dar. In der Ebene der Kommandosprache erfolgt der Zugriff auf ein File durch Angabe seines Namens in der entsprechenden Kommandozeile. Der Name wird als sogenannter Pfadname angegeben und symbolisiert die 'Lage' des bezeichneten Files innerhalb der Filesystem-Hierarchie.
Der Standardwert, mit dem **HOME** belegt ist, ist das im Password-File **etc/passwd** fuer jeden Nutzer eingetragene login-Directory, welches auch immer zu Beginn der Arbeit dieses Nutzers (nach login) das aktuelle Directory fuer ihn ist.

PATH

Unter diesem Namen sind Directories aufgelistet, in denen nach Files gesucht werden soll. Die Ausfuehrung eines Kommandos erfordert, dass zuerst das ausfuehrbare File mit dem angegebenen Namen (dem

Kommandonamen) innerhalb der Filesystem-Hierarchie gesucht wird, um dann ueber execute(2) dessen Ausfuehrung einzuleiten. Damit die Kommandonamen nicht unbedingt als vollstaendige Pfadnamen, d.h. ausgehend von / angegeben werden muessen, gibt es Standard-Suchpfade, die immer durchlaufen werden, wenn ein Kommando abgearbeitet werden soll und nicht durch ein Neusetzen von PATH etwas anderes vorgeschrieben wurde. Diese Standardpfade sind das **aktuelle Directory, /bin** und **/usr/bin.** Durch Neusetzen von PATH kann jeder Nutzer eine ihm genehme Aenderung dieses Standards vornehmen. Bei MUTOS 8000 wird durch eine Eintragung in .profile PATH zu Beginn der Arbeit mit dem System gewoehnlich wie folgt gesetzt:

PATH=:/mnt/bin:/usr/bin:/bin:/etc:/mnt/etc

Die Kommandos werden dann in der festgelegten Reihenfolge in den durch `:' getrennten Directories gesucht; zuerst also im aktuellen Directory (dargestellt durch die leere Zeichenkette vor dem ersten `:'), dann in **/mnt/bin,** wo die privaten Dienstprogramme des Nutzers untergebracht sein koennen, dann weiter in **/usr/bin** sowie in **/bin,** den Directories, in denen normalerweise die MUTOS 8000 - Dienstprogramme enthalten sind, und schliesslich in /etc und /mnt/etc.

Diese Suchstrategie ist ausser Kraft gesetzt, wenn der Kommandoname (nicht an erster Stelle) ein `/' enthaelt. In diesem Falle wird nur ein einfacher Zugriff vom aktuellen Directory ausgefuehrt.

***PS1**

Diese Zeichenkette (Prompt-Zeichenkette) wird auf dem Bildschirm ausgegeben, wenn Shell zur interaktiven Arbeit bereit ist. Standard ist `#'.

***PS2**

Wird diese Zeichenkette durch Shell auf dem Bildschirm ausgegeben, so sind weitere Eingaben des Nutzers erforderlich, um bereits gemachte Eingaben gemaess der Shell-Syntax zu vervollstaendigen. Standard ist `>'.

***IFS**

Diese Variable enthaelt die Zeichen, die Shell zusaetzlich zu den Shell-Metazeichen als Trennzeichen zwischen den Worten in Kommandozeilen interpretieren soll. Standard sind Leerzeichen, Tabulator und `Newline' (siehe auch Abschnitt 4.5.).

3.2. Programmablauf-Steuerung

Shell stellt vier Verzweigungs- bzw. Schleifenmechanismen zur Steuerung des Ablaufs der Kommandoabarbeitung (vor allem innerhalb von Shell-Prozeduren) bereit. Abhaengig vom Rueckkehrcode (Exit-Status) des zuletzt abgearbeiteten Kommandos wird im **if**- und im **while**-Konstrukt ueber den weiteren Ablauf entschieden, waehrend der **case**- und der **for**-Konstrukt eine daten- bzw. zeichengesteuerte Verzweigung bzw. Schleife darstellen.

3.2.1. Die if-Verzweigung

Die allgemeine Form des **if**-Kommandos hat die Gestalt:

```
if Kommandoliste1
then Kommandoliste2
else Kommandoliste3
fi
```

Der **else**-Zweig hat optionalen Charakter, muss also nicht angegeben werden.

Dabei steht Kommandoliste1 jeweils fuer eine Folge von ein oder mehreren Kommandos, die durch Semikolon oder 'Newline' voneinander getrennt sind. Ist der Rueckkehrwert des letzten einfachen Kommandos aus der Kommandoliste1 gleich Null, so wird die Kommandoliste2 abgearbeitet, anderenfalls die Kommandoliste3. Ein einfaches Beispiel fuer die Benutzung des **if**-Kommandos ist

```
if test -d $1
then ls -l $1 >/dev/lp
fi
```

Test ist ein Anwenderprogramm, das prueft, ob das angegebene Argument ein Directory oder ein regulaeres File ist. Dabei wird als Rueckkehrwert Null geliefert, wenn das Argument ($1) ein Directory ist. Andernfalls wird ein von Null verschiedener Wert zurueckgegeben.
Fuer geschachtelte **if**-Konstrukte der Form

```
if ...
then ...
else

        if ...
        then ...
        else ...

             ...

        fi

fi
```

gibt es eine die Schreibarbeit reduzierende Modifikation des if-Kommandos

```
if...
then...
elif...
then...
elif

...
fi
```

Eng verwandt mit dem if-Konstrukt sind die Funktionen der Shell-Metazeichen && und ||.

So laesst sich die Kommandofolge

```
if Kommandoliste1
then Kommandoliste2
fi
```

auch schreiben als

```
Kommandoliste1 && Kommandoliste2
```

Waehrend durch

```
Kommandoliste1 || Kommandoliste2
```

ein Arbeiten von Kommandoliste2 nur dann erfolgt, wenn die Kommandoliste1 nicht erfolgreich abgearbeitet wurde, d.h. einen Rueckkehrwert ungleich Null hatte. Der Rueckkehrwert einer Kommandoliste ist jeweils der des zuletzt abgearbeiteten einfachen Kommandos.

3.2.2. Die while-Schleife

Ebenso wie beim if-Kommando bildet beim **while**-Kommando der Test eines Rueckkehrwertes die Grundlage fuer die Steuerung des weiteren Programmablaufes. Die allgemeine Form dieses Konstruktes ist

```
while Kommandoliste1
do Kommandoliste2
done
```

Falls der Rueckkehrwert des letzten einfachen Kommandos aus der Kommandoliste1 gleich Null ist, wird die Kommandoliste2 abgearbeitet und anschliessend wieder die Kommandoliste1 mit dem entsprechenden Test des Rueckkehrwertes. Diese Schleife wird solange durchlaufen, bis beim Abarbeiten der Kommandoliste1 ein Rueckkehrwert ungleich Null geliefert wird. Ohne die Kommandoliste2 nochmals abzuarbeiten, ist in diesem Falle die Schleife beendet.

Eine Modifikation des **while**-Kommandos ist die **until**-Schleife. Hier erfolgt das Beenden der Schleife im entgegengesetzten Fall, d.h., wenn der Rueckkehrwert der Kommandoliste1 gleich Null ist.

3.2.3. Die for-Schleife

Diese Moeglichkeit fuer die Programmierung einer Schleife wird sehr haeufig in Shell-Prozeduren genutzt. Das **for**-Kommando hat die allgemeine Form

```
for Name in Wort1 Wort2 ...
do Kommandoliste
done
```

Der Shell-Variablen Name werden nacheinander die Zeichenketten Wort1, Wort2 usw. zugewiesen und fuer jede dieser Belegungen wird einmal die Kommandoliste abgearbeitet.

Enthaelt z.B. die Shell-Prozedur mulpr die Kommandos,

```
for count in 1 2 3 4 5
do pr -h "%count. Exemplar von %1" %1 >/dev/lp
done
```

so realisiert der Aufruf

```
mulpr ein1.1
```

das fuenfmalige Drucken des Files ein1.1 auf dem Zeilendrucker mit Numerierung der Exemplare in der durch pr(1) erzeugten Ueberschrift.

Die 'eigentliche' Anwendung des **for**-Kommandos liegt aber wohl im Durchlaufen aller Argumente einer Shell-Prozedur. Dafuer laesst sich die verkuerzte Schreibweise

```
for Name
do ...
done
```

verwenden. Bei Weglassen des Teiles **in** Wort1 Wort2 ... setzt Shell als Standard **in** %* ein, so dass zum Beispiel durch die Prozedur compall mit dem Inhalt

```
for filename
do cc %filename -c
done
```

eine Uebersetzung aller als Argumente uebergebener C-Quellen erreicht wird. Der Aufruf

```
compall file1.c file2.c
```

liefert, fehlerfreie Quellen file1.c und file2.c

vorausgesetzt, die zugehoerigen Objekt-Files 'file1.o' und 'file2.o'.

3.2.4. Die case-Verzweigung

Der **case**-Konstrukt bietet die Moeglichkeit der Mehrwegverzweigung in Shell-Prozeduren in Abhaengigkeit des Musters bestimmter Zeichenketten. Die allgemeine Form des **case**-Kommandos ist:

```
case Wort in

     Muster1 ) Kommandoliste1 ;;
     Muster2 ) Kommandoliste2 ;;

     ...

esac
```

Die Zeichenkette Wort wird in der angegebenen Reihenfolge nacheinander mit den Zeichenketten Muster1, Muster2 usw. verglichen. Wird eine Uebereinstimmung der verglichenen Zeichenketten festgestellt, so wird die zugehoerige Kommandoliste abgearbeitet. Damit ist die Ausfuehrung des **case**-Kommandos beendet, d.h. eventuelle weitere Uebereinstimmungen von Wort mit einem der Muster werden nicht beachtet. Die Anwendung der for-Schleife und der case-Verzweigung soll folgende Shell-Prozedur zeigen:

```
mt1=/etc/mount; mt2=/etc/mount
dev1=/dev/rk0; dev2=/dev/rk1
name1=usr; name2=mnt
for i
do case $i in
        -m)         ;;
        -m0|-0)     mt2=; dev2=; name2=;;
        -m1|-1)     mt2=; dev2=; name2=; name1=mnt;
                    dev1=/dev/rk1;;
        -u|-u[01]) name1=; name2=;
                    mt1=/etc/umount; mt2=/etc/umount;
                    case $i in
                         -u0)   mt2=; dev2=;;
                         -u1)   mt2=; dev2=;
                                dev1=/dev/rk1;;
                    esac;;
        -*)         echo "unkown flag $i"; exit;;
        *)          name1=$i;;
        esac
        done
        $mt1 $dev1 $name1
        $mt2 $dev2 $name2
```

Da der Buerocomputer A5120.16 als Externspeicher nur ueber Disketten verfuegt, ist es haeufig notwendig die Filesysteme, die sich auf Diskette befinden, ein- bzw. auszugliedern. Dazu

stehen die Kommandos mount(1) und umount(1) zur Verfuegung.

Die angegebene Shell-Prozedur soll den Aufwand fuer die Tastatureingabe reduzieren. Es wird davon ausgegangen, dass rk0 hauptsaechlich ueber den Pfad /usr und rk1 hauptsaechlich ueber den Pfad /mnt eingegliedert wird. Durch den Aufruf

```
mo
```

soll das Eingliedern beider Filesysteme durchgefuehrt werden. Das Ausgliedern beider Filesysteme erfolgt durch:

```
mo  -u
```

Ein einzelnes Filesystem ist auf rk0 oder rk1 eingliederbar durch:

```
mo  -m0 [name]
```

bzw.

```
mo  -m1 [name]
```

Das Ausgliedern erfolgt analog.
Die beiden letzten Zeilen der Shellprozedur sind die einfachen Kommandos, deren Namen und Argumente aus Variablen zusammengesetzt werden. Die Variablen werden als erstes entsprechend dem Eingliedern beider Filesysteme (Aufruf der Prozedur ohne Argumente) initialisiert. Werden diese Werte nicht veraendert, ergibt sich nach der Substitution fuer die beiden letzten Zeilen

```
/etc/mount /dev/rk0  usr
/etc/mount /dev/rk1  mnt
```

Mit der for-Schleife werden die Argumente der Variablen i nacheinander zugewiesen. Durch die aeussere case-Verzweigung werden diese Argumente getestet und Variablen neue Werte zugewiesen.
Die Schreibweise

```
-m0|-0)
```

bedeutet, dass die nachfolgende Kommandoliste abgearbeitet wird, wenn als Argument '**-m0**' oder '**-0**' gelesen wird (Eingliedern des Filesystems auf rk0). Die Kommandoliste loescht die Werte der Variablen fuer die letzte Kommandozeile. Damit wird durch diese Zeile kein Kommando aufgerufen. Zur Verkuerzung der Shellprozedur werden die Operationen

```
-u
-u0
-u1
```

zunaechst gemeinsam behandelt. In dem Muster des case-Konstruktes wird dazu das Metazeichen benutzt, das im Abschnitt 2.5. als Filenamen-Generierung beschrieben wurde.

Eine Unterscheidung der Operationen fuer das Ausgliedern erfolgt durch eine weitere case-Verzweigung. Eine nicht vorgesehene Optionsangabe wird durch das Muster '-*' (Bindestrich und beliebige Zeichen) ermittelt. In diesem Fall wird mit echo(1) eine Fehlernachricht gesendet und die Prozedur mittels exit(1) beendet. Eine Uebereinstimmung mit einer beliebigen Zeichenkette (Pfadname) wird durch das Muster '*' erreicht.

3.3. Eingabepakete

Einige MUTOS 8000 - Dienstprogramme verlangen waehrend ihrer Abarbeitung Eingaben ueber Standard-Input. Sollen solche Programme in Shell-Prozeduren integriert werden, ist es angebracht, ihnen die geforderten Eingaben als 'Eingabepaket' zur Verfuegung zu stellen. Der folgende Editor-Aufruf innerhalb einer Shell-Prozedur ist ein Beispiel fuer die Benutzung von Eingabepaketen.

```
ed text <<ende
1,\*s/Mutos/MUTOS/g
w
ende
```

Die Zeilen ab <<ende bis zum erneuten Auftreten von ende werden durch Shell als Standard-Input fuer das Kommando ed zur Verfuegung gestellt. Die Beispielprozedur realisiert damit, dass im File text jede Zeichenkette Mutos in MUTOS umgewandelt wird. Anstelle von ende kann dabei irgendeine beliebige Zeichenkette zum Begrenzen des Eingabepaketes benutzt werden. Die begrenzende Zeichenkette muss zum Abschluss des Eingabepaketes jedoch allein auf einer Zeile (und am Zeilenanfang) stehen. Bevor ein Eingabepaket Standard-Input des zugehoerigen Kommandos wird, fuehrt Shell gegebenenfalls Parametersubstitution aus. Deshalb ist das Zeichen fuer die letzte Zeile '*' durch Voranschreiben von '\' zu apostrophieren. Eine Prozedur edi koennte die Zeilen

```
ed *3 <<!!
1,\*s/*1/*2/g
w
!!
```

enthalten. Das Kommando

```
edi Mutos MUTOS text
```

wuerde zum gleichen Resultat fuehren, wie das zuerst angefuehrte Beispiel.

Soll die Substitutionsausfuehrung fuer das gesamte Eingabepaket unterdrueckt werden, so ist die das Eingabepaket begrenzende Zeichenkette wie in der Prozedur edall zu apostrophieren:

```
for filnam
do  ed *filnam <<\@
    1,*s/string1/string2/g
    1,*s/string3/string4/g
    w
    @
done
```

Fuer alle beim Aufruf von edall als Argumente uebergebenen Files werden die in dem Eingabepaket beschriebenen Substitutionen ausgefuehrt.

3.4. Kommando-Gruppen

Kommandos koennen auf zwei Arten zu Kommandogruppen zusammengefuegt werden: durch Einschliessen mehrerer Kommandos in geschweifte Klammern

{Kommandoliste}

sowie durch Einschliessen in runde Klammern.

(Kommandoliste)

Im ersten Fall wird die Kommandoliste ohne Besonderheiten abgearbeitet. Im zweiten Fall wird zum Abarbeiten der eingeklammerten Kommandos ein eigener Prozess eroeffnet - oder anders ausgedrueckt: die in runden Klammern eingeschlossenen Kommandos werden in einem eigenen (Unter-) Shell-Prozess abgearbeitet.

Durch die Kommandofolge

```
{cd dir1 ; mv file1 file1a}
```

wird das File file1 im Directory dir1 in file1a umbenannt. Das aktuelle Directory wird das durch cd erreichte dir1.

Die Kommandofolge

```
(cd dir1 ; mv file1 file1a)
```

dagegen haette zwar bezueglich der Umbenennung von file1 dieselbe Wirkung wie oben, aber nach Abarbeiten dieser Kommandos haette sich das aktuelle Directory des Nutzers nicht veraendert, da das Kommando cd, ausgefuehrt im Unter-Shell-Prozess, keine Wirkung in der aufrufenden Shell-Ebene hat.

3.5. Das Testen von Shell-Prozeduren

Zur Unterstuetzung des Tests von Shell-Prozeduren sind zwei Mechanismen verfuegbar, die ein Verfolgen des Programmablaufs innerhalb der Shell-Prozeduren gestatten (Tracing). Der erste Mechanismus realisiert, dass jede eingelesene Kommandozeile

der Shell-Prozedur auf den Bildschirm ausgegeben wird. Damit wird besonders das Ueberpruefen der syntaktischen Richtigkeit unterstuetzt. In Gang gesetzt wird dieser Mechanismus durch Angabe der Option -v beim Aufruf einer Shell-Prozedur, wie zum Beispiel in

```
sh -v procname ...
```

Soll der Tracing-Mechanismus nicht fuer die gesamte Shell-Prozedur, sondern nur fuer bestimmte Teile gueltig sein, so kann er innerhalb der Prozedur an einer beliebigen Stelle durch Einfuegen des Kommandos

```
set -v
```

aktiviert werden. Dieser, wie auch der zweite Tracing-Mechanismus, wird durch das Kommando

```
set -
```

an einer anderen Stelle der Shell-Prozedur wieder ausser Kraft gesetzt.

Fuer den zweiten Mechanismus gelten die gleichen Regeln zum Ein- bzw. Ausschalten. Anstelle von 'v' ist 'x' zu schreiben. Ist dieser zweite Tracing-Mechanismus fuer eine Shell-Prozedur aktiv, so werden auf dem Bildschirm die Kommandos ausgegeben (mit einem vorangestellten '+'), die gerade von Shell ausgefuehrt werden. Dabei sind bereits alle Parametersubstitutionen erfolgt, so dass man sich mit Hilfe dieses Mechanismus Klarheit ueber das Aussehen der tatsaechlich ausgefuehrten Kommandos verschaffen kann.

In Verbindung mit der Option -v ist die Option -n von Bedeutung. Ist -n gesetzt, so wird die Ausfuehrung aller folgenden Kommandos unterdrueckt. Daher sollte man diese Option auch nicht fuer den am Bildschirmterminal laufenden Shell-Prozess angeben. Wird das Kommando

```
set -n
```

am Terminal eingegeben, koennen danach eingegebene Kommandos nicht mehr ausgefuehrt werden. Nur durch Eingabe der 'End-of-File'-Folge <CTRL/Z> kann dieser Zustand aufgehoben werden.

4. Weitere Moeglichkeiten von Shell

Im Abschnitt 3. und besonders unter 3.1. wurde bereits einiges dazu gesagt, wie Parameter bzw. Variablen in Shell-Prozeduren zu behandeln sind. Hier sollen nun einige weitere Moeglichkeiten der Arbeit mit Parametern vorgestellt werden.

4.1. Parameteruebergabe

Einer Shell-Prozedur koennen bei deren Aufruf, wie schon unter 3. geschildert, Argumente uebergeben werden, die innerhalb der Prozedur den Stellungsparametern $1, $2 usw. zugeordnet sind. Variablen koennen einer Shell-Prozedur aber auch durch 'Exportieren' erschlossen werden. Das Kommando

```
export source
```

legt z.B. fest, dass die Variable source eine solche exportierbare Variable ist. Allen spaeteren aufgerufenen Prozeduren werden Kopien dieser Variablen uebergeben. Modifikationen der Werte exportierter Variablen haben keinen Einfluss auf den Wert der entsprechenden Variablen in der aufrufenden Ebene (beim Exporteur). Dies ordnet sich in das allgemeingueltige Prinzip der Shell-Arbeit ein, wonach Aktionen in einer aufgerufenen Prozedur keinen Einfluss auf den Zustand der aufrufenden Prozedur haben, es sei denn, dass die aufgerufene Ebene ausdruecklich derartige Veraenderungen anfordert.

Soll verhindert werden, dass der Wert irgendeiner Variablen durch spaetere Zugriffe veraendert wird, so ist es moeglich, diese Variable als 'nur lesbar' zu vereinbaren. Das entsprechende Kommando bezueglich der Variablen source hat die Form

```
readonly source
```

und bewirkt, dass alle spaeteren Versuche, dieser Variablen Werte, d.h. Zeichenketten, zuzuweisen, erfolglos bleiben.

4.2. Kennwortparameter

Eine weitere Art der Uebergabe von Werten an Shell-Prozeduren wird realisiert, indem eine Wertzuweisung der Form Name = Wert dem Prozeduraufruf unmittelbar vorangestellt wird. Ein Beispiel dafuer ist:

```
mon=August days
```

Dadurch wird der Variablen mon noch vor Aufruf der Prozedur days der Wert August zugewiesen. Mit diesem Wert ist die Variable dann innerhalb days belegt. In der aufrufenden Shell-Ebene erfolgt durch dieses Kommando keine Wertzuweisung an mon. Die wie mon uebergebenden Parameter heissen Kennwortparameter. Durch Angabe von -k als Option an Shell werden alle Wertzuweisungen der Form Name = Wert, unabhaengig von ihrer Position in der Kommandozeile, als Kennwortparameter an das Kommando uebergeben. Durch

```
set -k
days mon=August arg1 arg2 ...
```

wird die selbe Zuweisung erreicht wie im obigen Beispiel. Die uebrigen Argumente arg1, arg2, ... sind in days als Stellungsparameter $1, $2, ... verfuegbar.

Auch die Stellungsparameter koennen noch auf eine andere Art mit Werten belegt werden. Innerhalb der Shell-Prozeduren werden den Stellungsparametern durch das **set**-Kommando Werte zugewiesen. Nach dem Kommando

```
set file1 file2
```

ist als $1 der Name file1 und als $2 der Name file2 verfuegbar.

```
set - *
```

weist den Stellungsparametern die Namen aller Files aus dem aktuellen Directory zu. Das Minuszeichen als erstes Argument soll verhindern, dass Shell die Angabe einer Option vermutet, falls der erste Filename mit einem '-' beginnt.

4.3. Parametersubstitution

In Shell gibt es einige Operatoren (- = + ?), die in Anhaengigkeit davon, ob ein bestimmter Parameter mit einem Wert belegt ist oder nicht, unterschiedliche Substitutionen veranlassen. Das Kommando

```
echo $var
```

bzw.

```
echo ${var}
```

liefert keine Ausgabe, falls die Variable var nicht mit einem Wert belegt ist. Fuer eine nichtbelegte Variable wird also die Null-Zeichenkette substituiert. Es ist aber auch moeglich, anstelle eines nichtbelegten Parameters eine angebbare Zeichenkette zu substituieren. Dies geschieht z. B. durch:

```
echo ${var-NOTSET}
```

Ist die Variable var mit einem Wert belegt, so wird dieser substituiert und durch das Kommando echo ausgegeben. Falls var kein Wert zugeordnet war, wird die Zeichenkette NOTSET substituiert und von echo ausgegeben. Statt der zu substituierenden Zeichenkette koennte in diesem Kommando auch der Wert einer anderen Variablen oder eines Stellungsparameters angegeben werden.

```
echo ${var-$1}
```

liefert als Ausgabe den Wert von $1, falls var kein Wert zugewiesen ist. Die bereits genannten Regeln zum Apostrophieren gelten auch hier, so dass

```
echo ${var-'$1'}
```

die Zeichenkette $1 ausgeben wuerde, falls var kein Wert zugewiesen waere.

Das Kommando

```
echo ${var=NOTSET}
```

liefert das gleiche Resultat wie

```
echo ${var-NOTSET}
```

Der Unterschied besteht darin, dass bei Benutzung des Operators '=' der Variablen var die Zeichenkette NOTSET als Wert zugewiesen wird, falls var vor der Ausfuehrung dieses Kommandos noch nicht mit einem Wert belegt war. Stellungsparameter duerfen bei Benutzung des Operators '=' nicht anstelle von var auftreten !

Im Gegensatz zu beiden bisher erlaeuterten Operatoren wird bei Benutzung von '+' gerade dann eine andere Zeichenkette substituiert, wenn die getestete Variable mit einem Wert belegt war.

```
echo ${var+WASSET}
```

liefert als Ausgabe die Zeichenkette WASSET, falls der Variablen var irgendein Wert zugeordnet ist. Anderenfalls wird nichts substituiert und das Kommando liefert keine Ausgabe.

Der Operator '?' schliesslich fuehrt zum Abbruch einer Shell-Prozedur, falls die getestete Variable nicht mit einem Wert belegt ist.

```
echo ${var?STOP}
```

liefert als Ausgabe den Wert der Variablen var, falls ein solcher vorhanden ist, und sonst die Meldung STOP. Der zweite Fall bedeutet auch den Abbruch der Shell-Prozedur nach diesem Kommando. Falls keine Zeichenkette wie STOP angegeben ist, erfolgt vor dem Abbruch der Shell-Prozedur die Ausgabe einer Standard-Meldung.

Soll das Abarbeiten einer Shell-Prozedur davon abhaengig gemacht werden, ob bestimmte Variablen gesetzt sind oder nicht, so bietet sich als erstes Kommando der Prozedur zum Beispiel folgendes an:

```
: ${var1?} ${var2?}
```

Der Doppelpunkt ist der Name eines speziellen Shell-Kommandos (wie set, export u.a.m. - siehe sh(1)), das keine Aktion ausloest. Es wird beispielsweise zum Einfuegen von Kommentarzeilen in Shell-Prozeduren benutzt. In unserem Beispiel wird nur getestet, ob die Variablen var1 und var2 mit Werten belegt sind (und der Prozedur in irgendeiner Form uebergeben wurden - sei es durch **export** oder als

Kennwortparameter). Ist nur eine der Variablen nicht belegt, so beginnt die eigentliche Prozedurabarbeitung gar nicht erst.

4.4. Kommandosubstitution

Eine Zeichenkette, die in Akzentzeichen (`) eingeschlossen ist, wird als Kommando betrachtet, das vor Abarbeiten der zugehoerigen Kommandozeile ausgefuehrt und durch seine nach Standard-Output gerichtete Ausgabe ersetzt wird. Das Kommando pwd(1) zum Beispiel liefert als Ausgabe auf Standard-Output den Namen des aktuellen Directory. Ist nun **/usr/wrk/bwk** das aktuelle Directory eines Nutzers, so weist die Kommandozeile

```
curdir=`pwd`
```

der Shell-Variablen curdir den Wert **/usr/wrk/bwk** zu. Die Kommandosubstitution schafft damit die Moeglichkeit, Programme, die Zeichenketten verarbeiten, unkompliziert in Shell-Prozeduren einzubeziehen. Durch die Kommandosprache selbst werden das Zusammenfuegen von Zeichenketten und das Mustererkennen und -ausfuellen im Zusammenhang mit der Filenamengenerierung realisiert.

Ein anderes Beispiel einer Kommandosubstitution koennte sein:

```
set`date`
echo Uhrzeit: $1
```

Dadurch wird die Ausgabe 'Uhrzeit : 12 :00 :00' erzeugt, falls es nach der Rechnerzeit gerade zwoelf Uhr mittags ist.

4.5. Die Substitutionen und das Apostrophieren

Fuer die Argumente von Kommandos kann Shell drei verschiedene Arten von Ersetzungen ausfuehren - Parametersubstitution, Kommandosubstitution und die Filenamengenerierung. Die Reihenfolge ihrer Ausfuehrung und die Auswirkungen der verschiedenen Arten des Apostrophierens sollen in diesem Abschnitt besprochen werden.

Vor dem Ausfuehren jeglicher Ersetzungen wird ein Kommando auf seine Richtigkeit hinsichtlich der einzuhaltenden Grammatik (siehe Anhang 1) ueberprueft. Werden hierbei keine Fehler festgestellt, werden die notwendigen Ersetzungen in folgender Reihenfolge vorgenommen:

1.) Parametersubstitution (wie fuer **$var**)

2.) Kommandosubstitution (wie fuer **`date`**)

Beide Substitutionen werden jeweils nur einmal durchgefuehrt. Daher liefert das Kommando

```
echo $a
```

als Ausgabe die Zeichenkette *b, falls der Wert von a gleich der Zeichenkette *b ist. Falls auch noch die Substitution des Wertes von b gewuenscht wird, ist die Benutzung des speziellen Shell-Kommandos eval erforderlich, das weiter unten beschrieben wird.

3.) Interpretation der Freiraeume

Nachdem die genannten Substitutionen fuer ein Kommando ausgefuehrt sind, wird die entstandene Zeichenkette (der 'Wortlaut' des Kommandos) in einzelne Worte aufgeteilt. Ein Wort ist dabei eine von Freiraeumen freie Zeichenkette. Freiraeume in diesem Sinne sind die in der Shell-Variablen IFS (siehe 3.1.) angegebenen Zeichen, im Standardfall Leerzeichen, Tabulator und 'Newline'. Eine leere Zeichenkette wird nicht als Wort interpretiert, es sei denn, sie ist apostrophiert.

```
echo ''
```

erzeugt eine leere Zeichenkette als erstes Argument fuer echo, waehrend durch

```
echo *empty
```

ein Aufruf von echo ohne Argument erfolgt, falls der Variablen empty eine leere Zeichenkette zugewiesen wurde bzw. falls empty ueberhaupt nicht mit einem Wert belegt ist.

4.) Filenamengenerierung

Jedes der nach den Schritten 1, 2 und 3 entstandenen Worte wird zum Abschluss aller Substitutionen nach eventuell vorhandenen Filenamen-Erweiterungszeichen durchsucht. Werden in einem Wort eines oder mehrere der Zeichen '*', '?', '[...]' festgestellt, so wird dieses Wort durch eine alphabetisch geordnete Liste aller Filenamen ersetzt, die in das vorgegebene Muster passen. Jeder dieser Filenamen stellt eines der Argumente des zugehoerigen Kommandos dar.

Die hier beschriebenen Ersetzungen werden alle auch fuer die innerhalb des for-Konstruktes durch

```
in Wort1 Wort2 ...
```

vorgegebene Liste von Worten Wort1, Wort2 usw. ausgefuehrt. Fuer das im case-Konstrukt durch

```
case Wort in
```

angegebene Wort ist keine Filenamengenerierung moeglich. Hier koennen lediglich Parameter- und Kommandosubstitutionen durchgefuehrt werden.

Neben den bereits beschriebenen Moeglichkeiten des Apostrophierens einzelner Zeichen durch '\' bzw. ganzer Zeichenketten mittels '...' existiert noch eine dritte Form. Dieser dritte Mechanismus dient ebenfalls dem Apostrophieren von Zeichenketten. Diese werden dazu in Anfuehrungsstriche eingeschlossen. Der Unterschied zu den vorher beschriebenen Apostrophierungsmechanismen besteht darin, dass in einer durch Anfuehrungsstriche eingeschlossenen Zeichenkette Parameter- und Kommandosubstitutionen ausgefuehrt werden. Eine Interpretation von Freiraeumen und eine Filenamen-Generierung erfolgt nicht. Durch

```
a=string
echo 'Fall1 : a = $a'
echo "Fall2 : a = $a"
```

werden auf dem Bildschirm die Ausgaben

```
Fall1 : a = $a
Fall2 : a = string
```

erzeugt. Besondere Bedeutung innerhalb von Anfuehrungsstrichen haben demnach die Zeichen:

```
$   fuer Parametersubstitution
`   fuer Kommandosubstitution
"   Beenden der apostrophierten Zeichenkette
\   Apostrophieren der Zeichen $, ` und ", die damit
    ihre besondere Bedeutung verlieren
```

Die folgende Tabelle gibt eine Uebersicht, welchen Metazeichen bei den einzelnen Apostrophierungsarten sowie bei der Kommandosubstitution eine besondere Bedeutung zukommt.

Metazeichen

	'	"	`	$	*	\
'	e	-	-	-	-	-
"	-	e	b	b	-	b
`	-	-	e	-	-	b

Dabei gilt:

```
e   Endezeichen
b   besondere Bedeutung
-   keine besondere Bedeutung
```

Bei der intensiveren Arbeit mit Shell-Prozeduren kann man schnell an einen Punkt gelangen, an dem es nicht mehr genuegt, dass bei Substitutionen jeweils nur ein einfaches Ersetzen

durchgefuehrt wird. Wurde der Variablen b die Zeichenkette string zugewiesen und der Variablen a die Zeichenkette *b, so liefert (wie oben bereits gesehen)

```
echo *a
```

die Ausgabe *b. Soll nun aber der Wert von b als Ausgabe erscheinen, so wird zum Ausfuehren der notwendigen doppelten Substitution das spezielle Shell-Kommando eval benutzt.

```
eval echo *a
```

liefert dann als Ausgabe string. Das Kommando eval bewirkt, dass die ihm zugeordneten Argumente (fuer die durch Shell bereits eine Bearbeitung, wie zu Beginn dieses Abschnittes beschrieben, durchgefuehrt wurde) als Kommando an Shell uebergeben werden, wodurch ein zweites Bearbeiten des eigentlichen Kommandos ausgeloest wird. Dadurch ist nach dem Setzen von

```
spr='eval anycommand | sort | pr'
```

auch ein Kommando der Art

```
*spr
```

moeglich. Dies bewirkt das gleiche wie:

```
anycommand | sort | pr
```

Beide Kommandos sortieren und formatieren die Ausgabe von anycommand. Das Benutzen von eval in diesem Beispiel ist noetig, da nach einer Substitution (*spr) keine Interpretation von Metazeichen (wie |) mehr erfolgt.

4.6. Kommandoausfuehrung

Sind in einem Kommando, wie in 4.5. beschrieben, alle notwendigen Ersetzungen vorgenommen, so wird dieses Kommando abgearbeitet. Dazu wird von Shell unter Benutzung des Systemrufes fork ein neuer Prozess geschaffen. Dieser neue Prozess ist Child-Prozess des urspruenglich arbeitenden Shell-Prozesses und enthaelt in seiner Ausfuehrungsumgebung die gleichen geoeffneten Files, wie der Parent-Prozess (also mindestens stdin, stdout, stderr) sowie den gleichen Status fuer die einzelnen MUTOS 8000 - Signale (siehe 4.7.).

Sind im abzuarbeitenden Kommando E/A-Umlagerungen angegeben, werden diese erst im Child-Prozess wirksam, d.h. nach dem Ausfuehren von fork. Somit haben E/A-Umlagerungen auch keinen Einfluss auf den urspruenglichen Shell-Prozess und gelten immer nur fuer die Dauer der Abarbeitung des zugehoerigen Kommandos, d.h. in dem gestarteten Child-Prozess.

Als Ergaenzung zu den im Abschnitt 2.3. vorgestellten elementaren Moeglichkeiten von E/A-Umlagerungen sollen nun einige weiterfuehrende Bemerkungen zu diesem Thema folgen.

Fuer die hinter einem Umlagerungszeichen angegebene Zeichenkette, die die neue Ein- bzw. Ausgaberichtung angibt, koennen Parameter- und Kommandosubstitutionen ausgefuehrt werden. Eine Filenamen-Generierung und eine Interpretation von Freiraeumen erfolgt fuer diesen Teil eines Kommandos nicht. Somit wird durch

```
cat text >> *.t
```

der Inhalt des Files text nicht etwa an alle Files im aktuellen Directory angefuegt, die auf '.t' enden, sondern nur einmal in ein File mit dem Namen ***.t** geschrieben.

Kommandosubstitutionen koennen analog zu den im Abschnitt 3.3. bereits erwaehnten Parametersubstitutionen auch in Eingabepaketen benutzt werden.
Weitere Moeglichkeiten der E/A-Umlagerung werden durch folgende Formen realisiert:

>&digit Der Filedeskriptor digit (eine Ziffer) wird unter Benutzung des Systemrufes dup(2) dupliziert und das entstandene Ergebnis dem Standard-Output (Filedeskriptor 1) zugeordnet, d.h. Ausgaben ueber Standard-Output erfolgen danach zu demselben File, dem der Filedeskriptor digit zugeordnet ist.

<&digit Analog oben werden Eingaben von Standard-Input nun von dem File gelesen, dem der Filedeskriptor digit zugeordnet ist.

>&- Der Standard-Output wird geschlossen.

<&- Der Standard-Input wird geschlossen.

Fuer alle Formen von E/A-Umlagerungen ist es moeglich, dem Umlagerungszeichen eine Ziffer voranzustellen. Dies bewirkt, dass sich die auszufuehrende Umlagerung nicht auf den Standard-Input bzw. -Output bezieht, sondern auf den Filedeskriptor, der durch die angegebene Ziffer dargestellt ist. So wird zum Beispiel durch

```
command 2>errfile
```

erreicht, dass beim Abarbeiten des Kommandos command die Meldungen, die ueber stderr (Filedeskriptor 2) ausgegeben werden, nicht wie im Standardfall zum Bildschirmterminal des Nutzers gerichtet sind, sondern in das File errfile geschrieben werden. Durch

```
command 2>&1
```

wird erreicht, dass die Ausgaben ueber stderr und stdin zum gleichen File gerichtet sind, wie es im Standardfall auch ist, wo beide Filedeskriptoren dem Bildschirmterminal des Nutzers zugeordnet sind. Das Zusammenlegen beider Ausgabestroeme wird dadurch erreicht, dass Filedeskriptor 2 ein Duplikat des Filedeskriptors 1 wird.

Fuer die Umgebung von Hintergrund-Kommandos wird durch Shell bezueglich der E/A-Richtungen eine Modifikation vorgenommen. Der Standard-Input solcher Kommandos ist immer das leere File /dev/null. Damit wird verhindert, dass die parallel laufenden Prozesse, das Hintergrund-Kommando und Shell versuchen, die selben Eingaben zu lesen. Ein solches Teilen der Eingaben koennte zu einem Durcheinander fuehren, wenn im Hintergrund ein Programm versehentlich ohne E/A-Umlagerung gestartet ist, das waehrend seiner Arbeit Eingaben ueber Standard-Input verlangt.

4.7. Signalbehandlung

Im Betriebssystem MUTOS 8000 werden 15 verschiedene Signale behandelt (siehe signal(2) fuer ausfuehrlichere Erlaeuterungen). Der folgende kurze Ueberblick gibt die Signale und ihre Bedeutung an.

1	hangup	Modem; Verlust der Traegerfrequenz
2	interrupt	Unterbrechung (wird durch das gleichzeitige Betaetigen der Tasten <CTRL> und <C> erzeugt)
3*	quit	Quittierung
4*	illegal instruction	illegaler Befehl
5*	trace trap	Signal fuer Programmverfolgung
6*	IOT trap	
7*	EMT trap	
8*	floating point exception	Gleitkommaverletzung
9	kill	Programmabbruch
10*	bus error	Hardware-Fehler
11*	segmentation violation	Segmentierungsverletzung (Hardware-Fehler)
12*	bad argument to system call	Falsche Argumente bei einem Systemruf
13	pipe	Schreiben in eine Pipe, die nicht gelesen wird
14	alarm clock	Signal der Echtzeituhr
15	software termination signal	Software-Beendigungssignal

Die mit einem '*' gekennzeichneten Signale fuehren zum Entstehen eines Speicherabzugs (Core-Files), falls sie vom betreffenden Prozess nicht abgefangen werden.

Ein MUTOS-Signal kann von einem Prozess auf drei verschiedene Arten behandelt werden. Die Standardbehandlung ist, dass der Prozess beim Empfang eines Signal abgebrochen wird, ohne zuvor

noch irgendwelche Aktionen auszufuehren. Eine zweite Moeglichkeit besteht darin, Signale zu ignorieren. In diesem Falle wird der Prozess fortgesetzt, als haette es das Ereignis, welches zum Senden des Signals gefuehrt hat, nicht gegeben. Die dritte Moeglichkeit ist das Auffangen von Signalen. Hierzu ist es notwendig, dass durch den Prozess festgelegt wird, welche Aktionen auszufuehren sind, wenn ein bestimmtes Ereignis eintritt, d.h. das zugehoerige Signal an den Prozess gesendet wurde.

Diese Moeglichkeiten der Signalbehandlung werden dem Shell-Benutzer auch auf der Ebene der Kommandosprache geboten. Die fuer eine Behandlung durch Shell relevanten Signale sind vor allem 1, 2, 3, 14 und 15. Shell selbst ignoriert das Signal 3, das einzige externe Signal, das zum Entstehen eines Speicherabzugs fuehren kann. Zur Signalbehandlung existiert das spezielle Shell-Kommando trap. Normalerweise wird das Abarbeiten einer Shell-Prozedur abgebrochen, wenn an den zugehoerigen Prozess ein Signal gesendet wird (z.B. ein Interrupt-Signal von der Tastatur des Nutzers). Durch

```
trap action 2
```

wird das Signal 2 (Terminal-Interrupt) jedoch abgefangen. Empfaengt der Prozess dieses Signal, so wird das Abarbeiten der Shell-Prozedur nicht abgebrochen, sondern nur unterbrochen und das Kommando action ausgefuehrt. Anschliessend wird die Shell-Prozedur an der Stelle fortgesetzt, wo sie unterbrochen wurde. Wurden durch eine Shell-Prozedur beispielsweise Files angelegt, die temporaeren Charakter haben sollen, so muss dafuer gesorgt werden, dass diese Files nicht nur bei normaler Beendigung der Prozedur sondern auch bei einem Abbruch nach Empfang eines Signals, geloescht werden. Dazu koennte folgendes Kommando dienen:

```
trap 'rm /tmp/ddir**; exit' 1 2 3 15
```

Beim Empfang eines der Signale 1, 2, 3 oder 15 wird das temporaere File **/tmp/ddir**** geloescht. Durch das spezielle Shell-Kommando exit wird die Shell-Prozedur danach beendet.

Ein Signal 0 gibt es in MUTOS 8000 nicht. Fuer das trap-Kommando wird diese Signalnummer dazu benutzt, Aktionen zu definieren, die bei Beendigung einer Shell-Prozedur auszufuehren sind.

```
trap endfile 0
```

sorgt dafuer, dass nach Beendigen der Prozedur, in der dieses Kommando erscheint, endfile abgearbeitet wird.
Das Ignorieren von Signalen wird mit dem trap-Kommando dadurch erreicht, dass eine leere Zeichenkette als erstes Argument, d.h. als Name des beim Empfang der angegebenen Signale abzuarbeitenden Kommandos, angegeben wird.

Wird trap fuer bestimmte Signale ohne Angabe einer auszufuehrenden Aktion aufgerufen, so wird beim Empfang dieser Signale wieder die Standardaktion, d.h. Abbruch des Prozesses, ausgefuehrt. Ein Aufruf wie

```
trap 2
```

dient daher dem Ruecksetzen frueher getroffener Festlegungen bzgl. des Signals 2. Durch

```
trap
```

wird eine Liste der den Signalen aktuell zugeordneten Aktionen ausgegeben.

Als abschliessendes Beispiel zur Signalbehandlung unter Shell soll die Prozedur scan dienen:

```
d=`pwd`
for name in *
do   if  test     -d $name
     then   cd    $d/$name
            while echo "$name:"
                  trap exit 2
                  read com
                  dotrap : 2
                  eval $com
            done
     fi
done
```

Ausgehend vom aktuellen Directory wird nacheinander in alle Subdirectories verzweigt. Nach dem Verzweigen wird jeweils der Name des Subdirectory ausgegeben. An dieser Stelle kann der Nutzer durch ein Interrupt das Abarbeiten von scan beenden. Ansonsten wartet scan durch das spezielle Shell-Kommando read auf eine Eingabe ueber Standard-Input. Wurde eine Eingabezeile gelesen, so wird deren Wortlaut der Variablen com zugewiesen, als Kommando aufgefasst und abgearbeitet. Waehrend der Abarbeitung dieses Kommandos ist scan nicht durch ein Terminal-Interrupt unterbrechbar. Wird End-of-File eingegeben, so erfolgt die Verzweigung ins naechste Subdirectory, da read in diesem Falle einen Rueckkehrwert ungleich Null liefert. Dies ist fuer while das Kriterium zum Beenden der Schleife.

Fuer Hintergrundkommandos wird durch Shell eine spezielle Signalbehandlung organisiert. Solche Kommandos ignorieren die Signale 2 und 3, so dass diese Signale vom Terminal aus gesendet werden koennen, ohne die Arbeit der im Hintergrund laufenden Prozesse zu beeinflussen.
Falls eine Shell-Prozedur im Hintergrund abgearbeitet wird, kann ein in dieser Prozedur auftretendes trap-Kommando die Behandlung der Signale 2 und 3 nicht veraendern.

4.8. Fehlerbehandlung

Die Behandlung von Fehlern, die waehrend der Ausfuehrung von Kommandos festgestellt werden, haengt von der Art des Fehlers ab sowie davon, ob der laufende Shell-Prozess interaktiv ist oder nicht. Als interaktiv wird ein Shell-Prozess bezeichnet, dessen Ein- und Ausgaben an Tastatur bzw. Bildschirm gebunden sind. Auch alle durch

```
sh -i ...
```

aufgerufenen Shell-Prozesse gelten in diesem Sinne als interaktiv.
Ursachen fuer Fehler beim Ausfuehren eines Kommandos koennen sein:

- Das Kommando, identifiziert durch seinen Namen, existiert nicht oder kann nicht ausgefuehrt werden (Zugriffsrechte).

- Umlagerungen der E/A-Richtungen koennen nicht durchgefuehrt werden, z. B. wenn ein angegebenes File nicht existiert bzw. nicht geoeffnet oder erstellt werden kann.

- Die Kommandoausfuehrung wird durch den Empfang eines Signals (z.B. Signal 10 - Ausschrift "bus error") vorzeitig beendet.

- Die Kommandoausfuehrung wird normal beendet, aber der Rueckkehrwert des Kommandos ist ungleich Null.

Fehler, die aus den bisher aufgefuehrten Ursachen resultieren, bewirken (bis auf den letzten Fall) die Ausgabe einer Fehlernachricht und den Uebergang zum Abarbeiten des naechsten Kommandos in Shell-Prozeduren bzw. das Warten auf weitere Eingaben im interaktiven Fall.

Weitere Fehler koennen sein:

- Syntaxfehler, z. B. das Vergessen von done zum Beenden einer Schleife oder von ;; zum Abgrenzen der einzelnen Faelle eines case-Konstruktes.

- Empfang eines Signals wie 2 (Tastatur-Interrupt).

- Fehlerhaftes Ausfuehren eines der speziellen Shell-Kommandos (cd, eval, exec, ...).

Fehler dieser Art bewirken den Abbruch des Abarbeitens einer Shell-Prozedur (nach Ende des gerade ausgefuehrten Kommandos) bzw. im interaktiven Fall das Warten auf eine weitere Eingabe (bei Syntaxfehlern mit Ausgabe von > (*PS2) auf dem Bildschirm.

Durch Angabe der Option -e beim Aufruf eines Shell-Prozesses oder in einem set-Kommando wird erreicht, dass der Prozess immer abgebrochen wird, wenn ein Fehler festgestellt wurde.

4.9. Der Aufruf von Shell

Beginnt das Argument Null beim Aufruf von Shell (mittels eines execute-Systemrufes) mit einem Minuszeichen, wie das beim Shell-Aufruf nach der Login-Prozedur der Fall ist, so wird als erstes Kommando aus dem File **.profile** im Home-Directory des jeweiligen Nutzers gelesen und ausgefuehrt.
Danach erfolgt die weitere Arbeit von Shell wie beschrieben. Durch Angabe der folgenden Optionen beim Aufruf von Shell kann diese Arbeit modifiziert werden.

-c name Shell liest Kommandos aus dem File name ein.

-s Shell liest Kommandos vom Standard-Input ein. Shell-Ausgaben sind nach Standard-Error (Filedeskriptor 2) gerichtet. Diese Festlegungen gelten auch, wenn beim Shell-Aufruf keine Argumente angegeben wurden.

-i Ein so aufgerufenes Shell gilt als interaktiv. Das gleiche trifft auch dann zu, wenn der Standard-Input und der Standard-Output eines Shell-Prozesses zu einem Terminal gerichtet sind. Als Terminal gilt, was durch einen gtty(2)-Ruf als solches identifiziert wird. Interaktive Shell-Prozesse ignorieren das Signal 15, so dass z. B. durch das Kommando

```
kill 0
```

ein interaktives Shell nicht mit beendet wird. Das Signal 2 wird abgefangen, aber keine zugehoerige Aktion ausgefuehrt. Dadurch kann ein Wartezustand, ausgeloest durch das spezielle Shell-Kommando wait(1), mit dem Signal 2 unterbrochen werden.

Anhang 1 - Grammatik

Kommando:
- Einfaches Kommando
- (Kommandoliste)
- {Kommandoliste}
- **for** Name **do** Kommandoliste **done**
- **for** Name **in** Wort ... **do** Kommandoliste **done**
- **case** Wort **in** Musterteil **esac**
- **while** Kommandoliste **do** Kommandoliste **done**
- **until** Kommandoliste **do** Kommandoliste **done**
- **if** Kommandoliste **then** Kommandoliste Else-Teil **fi**

Musterteil:
- Muster) Kommandoliste ;;

Muster:
- Wort
- Muster | Wort

Else-Teil:
- **elif** Kommandoliste **then** Kommandoliste Else-Teil
- **else** Kommandoliste
- Leer

Einfaches Kommando:
- Einheit
- Einfaches-Kommando Einheit

Einheit:
- Wort
- Input/Output
- Name=Wort

Kommandoliste:
- Und/Oder
- Kommandoliste ;
- Kommandoliste &
- Kommandoliste ; Und/Oder
- Kommandoliste & Und/Oder

Und/Oder:	Pipeline Und/Oder && Pipeline Und/Oder || Pipeline
Pipeline:	Kommando Pipeline | Kommando
Input/Output:	> File < File >> Wort << Wort
File:	Wort &Ziffer &-
Leer:	
Wort:	Zeichenfolge ohne Worttrennzeichen
Worttrennzeichen:	Leerzeichen, Tabulator, Newline ; & | < > ( )
Name:	Zeichenfolge, die aus Buchstaben, Ziffern oder Unterstrichen bestehen kann
Ziffer:	0 1 2 3 4 5 6 7 8 9

Anhang 2 - Metazeichen und reservierte Worte

a) Syntax

;	Trennung von Kommandos
&	Hintergrund-Kommandos
|	Pipe-Symbol
( )	Kommandogruppe
;;	Begrenzung eines Musterteils
&&	'Und'-Verknuepfung
||	'Oder'-Verknuepfung
<	Eingabe-Umlagerung
>	Ausgabe-Umlagerung
<<	Eingabepaket
>>	Anfuegen von Ausgaben

b) Substitutionen

$*{...}	Shell-Variable
`...`	Kommandos

c) Muster

*	steht fuer jede beliebige Zeichenfolge
?	steht fuer ein beliebiges einzelnes Zeichen
[...]	steht fuer jedes der eingeschlossenen Zeichen

d) Apostrophieren

\	apostrophiert das naechstfolgende Zeichen
'...'	apostrophiert die eingeschlossene Zeichenkette (ausser ')
"..."	apostrophiert die eingeschlossene Zeichenkette (ausser * ' \ ")

e) reservierte Worte

if then else elif fi

case in esac

for while until do done

{ }

Die reservierten Worte muessen jeweils als erstes Wort eines Kommandos erscheinen. Ausnahmen sind **in** und **}**.

Kv 462/86 WV/6/1-10 3109a