ANWENDER-DOKUMENTATION	Anleitung für den Bediener	MOS
11/87	Kommandointerpreter Shell	MUTOS 1700

Programmtechnische Beschreibung Teil 2

Anleitung für den Bediener

Kommandointerpreter Shell

AC A 7100/7150

VEB Robotron-Projekt Dresden

C 8015-0002-1 M 3030

Ausgabe: 11/87

Die Ausarbeitung dieser Dokumentation erfolgte durch ein Kollektiv des
VEB Robotron-Elektronik Dresden Stammbetrieb des VEB Kombinat Robotron.

Im Interesse einer ständigen Weiterentwicklung werden alle Leser gebeten, Hinweise zur Verbesserung der Dokumentation dem Herausgeber mitzuteilen.

Herausgeber:

VEB Robotron-Projekt Dresden
Leningrader Str. 9
Dresden 8010

Kurzreferat

Mit der Kommandosprache Shell wird dem Benutzer des Betriebssystems MUTOS eine Schnittstelle zum System zur Verfügung gestellt, die hohen Anforderungen gerecht wird. Die Shell verbindet Mechanismen algorithmischer Programmiersprachen wie Schleifen und Verzweigungen, Variablen und Parameterübergabe mit solchen Bestandteilen von Kommandosprachen, die der änderung der Ein- und Ausgaberichtung von Programmen dienen, die Kommandosubstitutionen ermöglichen, eine Signalbehandlung realisieren und vieles anderes mehr.
In der vorliegenden Schrift wird versucht, dem Leser, oft anhand kleiner Beispiele, eine Einführung in die Benutzung der Shell zu geben und ihn anzuregen, bei seiner Arbeit unter MUTOS die Möglichkeiten der Shell weitgehend auszunutzen.

Inhaltsverzeichnis Seite

1. Einleitung

Die Kommunikation der meisten Nutzer mit dem Betriebssystem MUTOS erfolgt über die Kommandosprache Shell. Der volle Umfang der Kommandosprache ist relativ groß. Dafür bietet die Shell aber die vielfältigsten Möglichkeiten, um dem Nutzer die Arbeit unter dem Betriebssystem MUTOS zu erleichtern.

Die Shell realisiert u.a.:

- verschiedene Elemente, wie sie aus algorithmischen Sprachen bekannt sind:
  - Programmablauf-Steuerkonstrukte(for, case, if, while)
  - Variablen
  - Parameterübergabe
- Kommando- und Parametersubstitution
- File-Namen-Generierung
- änderung der E/A-Richtungen
- Modifikation der Umgebung, in der ein Prozeß abläuft
- Signalbehandlung
- Datenübergabe zwischen verschiedenen Prozessen über Pipes
- Suchen von Files entlang vom Nutzer definierter Wege innerhalb des MUTOS-File-Systems.

Die Shell-Syntax zeichnet sich durch eine knappe Notation aus, die aus leicht verständlichen und einprägsamen Zeichen aufgebaut ist. Unter Ausnutzung der anwenderfreundlichen Eigenschaften von MUTOS wie der Transparenz des File-Systems, der vielen Dienstprogramme, die vom Standard-Eingabe-File stdin lesen und zum Standard-Ausgabe-File stdout ausgeben sowie des Pipe-Konzeptes, bietet die Shell dem Nutzer bei wenig Eingabearbeit viel an Systemleistung. Dabei unterstützt der Kommandointerpreter sowohl die interaktive Arbeit, d.h. die Eingabe einzelner Kommandozeilen über das Bildschirmterminal des Nutzers, als auch die Abarbeitung 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 völlig genügt, einige wenige Elemente der Kommandosprache und die Wirkungsweise einiger Dienstprogramme von MUTOS zu kennen, und später bei Bedarf nach und nach weitere Möglichkeiten der Shell hinzuzulernen.

Dem Rechnung tragend ist die vorliegende Schrift so aufgebaut, daß im zweiten Kapitel vornehmlich auf solche Bestandteile der Shell eingegangen wird, die man als Grundelemente des interaktiven Verkehrs bezeichnen könnte. Das dritte Kapitel stellt Möglichkeiten vor, die besonders die Arbeit mit Shell-Prozeduren unterstützen. Im vierten Abschnitt werden alle weiterführenden und zur Ergänzung der beiden ersten Kapitel notwendigen Bestandteile der Shell erläutert. Der Anhang gibt schließlich einen zusammenfassenden überblick der gesamten Shell-Grammatik.

Bemerkung:
Die Kommandosprache ist durch einen Interpreter implementiert (/bin/sh). Dieser hat keinen besonderen Status innerhalb der Bestandteile der MUTOS und ist genauso auslagerbar wie alle anderen Dienstprogramme. Es ist auch möglich, anstelle der Shell für einzelne Nutzer ein anderes Programm als Kommandointerpreter arbeiten zu lassen. Dazu ist lediglich eine änderung des Eintrags für den

betreffenden Nutzer im Kennwort-File (/etc/passwd) erforderlich. In einer bestimmten Umgebung (z.B. zur Datenerfassung) könnte der Editor die Stelle der 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 ausführbares Programm enthält. Die verbleibenden Worte werden durch die Shell als Argumente an das auszuführende Programm weitergegeben. Das Kommando wird durch ein Zeilenende-Zeichen abgeschlossen und dann von der Shell interpretiert. Unter einem Zeilenende-Zeichen wird das durch Betätigen der Tasten Wagenrücklauf (carriage return, <CR>) ausgelöste Zeichen verstanden. So wird nach Eingabe der Kommandozeile

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

ein File mit dem Namen comname gesucht und über einen execute-Systemruf dessen Ausführung eingeleitet, wobei die Argumente arg1, arg2, ...argn als Parameter des execute-Systemrufes an das auszuführende Programm übergeben 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 von der aktuellen Directory aus über den angegebenen Pfadnamen zu erreichen. Beginnt comname mit "/", so wird der Pfadname als vollständig bezeichnet, und die Suche nach dem auszuführenden Programm beginnt in der Wurzel (root) des gesamten File-Systems.

Ist comname nur ein einfacher Name, in dem kein Zeichen "/" enthalten ist, so sucht die Shell ein File mit diesem Namen einen Suchpfad entlang, der vom Nutzer verändert werden kann (siehe $PATH im Abschnitt 3.1.). Als Standard wird versucht, das File comname zuerst in der aktuellen Directory zu finden, falls erfolglos, dann in /bin und schließlich in /usr/bin enthalten die MUTOS-Dienstprogramme (siehe Programmtechnische Beschreibung Teil 1, Abschnitt 1.).

2.2. Hintergrundkommandos

Zur Ausführung eines Kommandos wird von der Shell ein neuer Prozeß eröffnet, auf dessen Beendigung, d.h. das Ende der Kommandoabarbeitung, dann gewartet wird. Solange die Kommandoabarbeitung andauert, ist für den Nutzer keine Interaktion mit der Shell möglich. Wird ein Kommando jedoch mit dem Operator "&" abgeschlossen, so wird der Wartemechanismus für dieses Hintergrundkommando 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 zum Beispiel, daß der Nutzer während der übersetzung des C-Programms source.c andere Arbeiten an seinem Bildschirmterminal erledigen kann.

2.3. änderung der E/A-Richtungen

Zu jedem Prozeß unterhält MUTOS eine Menge von File-Deskriptoren "0", "1", die der Identifizierung von Files während der Durchführung von E/A-Operationen dienen. Nachdem ein Nutzer seine Arbeit unter MUTOS begonnen hat (login) sind für seinen laufenden Shell-Prozeß drei Files, denen die Deskriptoren "0", "1" und "2" zugeordnet sind, geöffnet. 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 geöffneten Files entsprechen zu Beginn der Arbeit dem Bildschirmterminal des Nutzers (Spezial-File /dev/tty*).
Die meisten MUTOS-Dienstprogramme sind so beschaffen, daß sie Eingaben von stdin lesen und Ausgaben nach stdout liefern. Um Ein-/Ausgaben von/nach einem anderen Gerät bzw. File als dem Terminal zu realisieren, kann mit Hilfe der Zeichen "<", ">", "<<" und ">>" eine änderung der E/A-Richtung für die Dauer der Ausführung eines Kommandos erreicht werden. Die Zeichen zur Richtungsänderung werden mit den ihnen nachgestellten File-Namen von der Shell selbst interpretiert und nicht als Argumente an die aufgerufenen Programme, wie die MUTOS-Dienstprogramme cat und pr in den folgenden Beispielen, übergeben. Die folgenden Kommandos bewirken somit:

cat text	(ohne Richtungsänderung) Ausgabe des Files text auf dem Bildschirmterminal
cat text >outtext	Ausgabe von text in das File outtext. Falls outtext noch nicht existiert, wird dieses File durch Shell erstellt, ansonsten wird der alte Inhalt überschrieben.

Da die Ein- und Ausgaben zu peripheren Geräten (repräsentiert durch die zugehörigen Spezial-Files) in MUTOS genauso behandelt werden wie zu gewöhnlichen Files, läßt sich durch

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

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

```
cat text >> oldfile
```

erreicht. Die Eingaberichtung kann mit "<" geändert werden. Der Deskriptor stderr ist von diesen E/A-Richtungsänderungen nicht betroffen. Die Benutzung von "<<" wird im Abschnitt 3.3. näher erläutert.

2.4. Pipelines und Filter

Die Bereitstellung des Pipe-Mechanismus durch MUTOS und die einfache Handhabung, die die Shell dazu bietet, ermöglichen auf unkomplizierte Weise die Kombination von MUTOS-Dienstprogrammen untereinander und mit Nutzerprogrammen.
Das Verknüpfen 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 Programms anycommand mit Hilfe des MUTOS-Dienstprogramms sort sortiert und anschließend entsprechend pr formatiert ausgegeben werden. Ohne Benutzung des Pipe-Operators müßten zum Erreichen des gleichen Ergebnisses folgende Kommandos abgearbeitet werden:

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

Der die Shell-Pipe-Operator ist über den Systemruf pipe(2) implementiert. Dementsprechend erfolgt auch die Synchronisation der Abläufe in Pipelines durch den Betriebssystemkern und nicht durch die die Shell.
Programme wie sort, 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 zur Verfügung gestellten Dienstprogramme sind derartige Filter.

2.5. File-Namen-Generierung

Vielen MUTOS-Dienstprogrammen werden neben den Optionen zur Auswahl von bestimmten Teilfunktionen des Kommandos vor allem File-Namen als Argumente übergeben. Um die Arbeit mit den File-Namen effektiv zu gestalten, d.h. um die nötige Schreibarbeit zu verringern sowie gleichzeitig einen Suchmechanismus nach bestimmten File-Namen zur Verfügung zu haben, werden von die Shell die File-Namen-Erweiterungszeichen "*" "?" und "[...]" unterstützt. Wird ein File-Name innerhalb eines Kommandos nicht vollständig sondern in Form eines Musters, gebildet unter Benutzung der File-Namen-Erweiterungszeichen, angegeben, so ermittelt die Shell alle dem Muster entsprechenden File-Namen und setzt diese, alphabetisch geordnet, als einzelne Argumente anstelle des Musters in die Kommandozeile ein. Das Zeichen "*" steht dabei für jede beliebige Zeichenkette, "?" für ein beliebiges Zeichen und anstelle von "[...]" kann im zu ermittelnden File-Namen irgendeines der in den Klammern angebenen Zeichen auftreten.
Daher listet das Kommando

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

alle File-Namen der aktuellen Directory, die mit .c enden (in einer gemäß der Option -l "langen" Version der Liste)und .sp

```
ls -l /dev/tty?
```

liefert die Namen aller Terminals, für 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 erhält man durch

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

die Namen aller MUTOS-Dienstprogramme, die mit den Buchstaben m, n, o bis z beginnen. File-Namen, die mit einem "." beginnen, werden bei der File-Namen-Generierung nach solchen Mustern nicht mit erfaßt. Der "." müßte in einer entsprechenden Kommandozeile explizit angegeben werden.
Das MUTOS-Dienstprogramm echo gibt seine Argumente, durch Leerzeichen getrennt, auf Standard-Output aus. Durch

```
        echo .*
```

erhält man die Namen aller Files in der aktuellen Directory, die mit "." beginnen, während

```
        echo *
```

gerade die Namen aller anderen Files in dieser Directory liefert.
Diese Sonderbehandlung des Punktes als erstem Zeichen im File-Namen wird durchgeführt, um das Miterfassen der Namen "." (für die aktuelle Directory selbst) und ".." (für seine Parent-Directory) bei der File-Namen-Generierung zu vermeiden.

2.6. Apostrophieren

Die Zeichen, die eine besondere Bedeutung für die Shell haben, werden Sonderzeichen genannt. Neben den bisher schon erwähnten Zeichen "<", ">", "*", "?", "|", "&" gibt es noch weitere Sonderzeichen, die in den folgenden Kapiteln erläutert werden. Im Anhang findet sich nochmals eine Zusammenstellung aller Sonderzeichen.
Soll eines der Sonderzeichen in einer Kommandozeile ohne seine besondere Bedeutung für die Shell benutzt werden (z. B. in File-Namen), so muß dieses Zeichen apostrophiert werden, indem ein "\" vorangestellt wird. Der "\" wird von der Shell aus der Kommandozeile entfernt, so daß

```
        echo \*
```

als Ausgabe "*" liefert. Das Zeichen "\" kann auch selbst apostrophiert werden. Es kann also mit

```
        echo
```

ausgegeben werden.
Auch das Zeilenende-Zeichen (Taste <CR>) kann auf diese Weise apostrophiert werden, so daß die Folge "<CR>" von der Shell nicht als Kommandozeilenabschluß erkannt wird. Lange Kommandozeilen können sich somit über mehrere Bildschirmzeilen erstrecken.
Das umständliche und fehleranfällige Apostrophieren längerer Zeichenketten nach der soeben geschilderten Methode ist nicht notwendig. Eine Zeichenkette kann, um die Bearbeitung darin enthaltener Sonderzeichen durch die Shell zu vermeiden, in Apostrophe "'...'" eingeschlossen werden.

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

Die apostrophierte Zeichenkette kann Zeilenende-Zeichen (<CR>) enthalten, die analog zum Apostrophieren mit "\" nicht als solche interpretiert werden. Apostrophe selbst dürfen in der so apostrophierten Zeichenkette nicht enthalten sein.
Schließlich existiert noch ein dritter Mechanismus zum Apostrophieren, der mit Anführungsstrichen arbeitet und nicht alle eingeschlossenen Shell-Sonderzeichen von der Behandlung durch die Shell ausschließt. Die Erläuterungen dazu befinden 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). Falls in der "Home-Directory" dieses Nutzers ein File mit dem Namen .profile enthalten ist, so wird zunächst dieses File von der Shell eingelesen. Es wird als die Shell-Prozedur (siehe Kapitel 3) interpretiert, d. h. .profile sollte Kommandozeilen enthalten, von denen der Nutzer möchte, daß sie jedesmal vor Beginn seiner Arbeit am Bildschirmterminal ausgeführt werden. Ein einfaches Beispiel eines solchen .profile könnte folgendermaßen aussehen:

```
stty length 24
echo -n 'Aktuelles Datum und Zeit: '
date
PS2='weiter: '
```

Mit dem Kommando stty(1) lassen sich bestimmte Terminalfunktionen setzen. In unserem Fall wird die Länge einer Bildschirmseite gleich 24 Zeilen gesetzt. Alle Ausgaben auf dem Bildschirmterminal dieses Nutzers erscheinen von nun an in Seiten zu je 24 Zeilen. Die jeweils nächste Seite wird nach Betätigen der Taste <Leerzeichen> ausgegeben. Das Kommando echo(1) liefert als Ausgabe auf dem Bildschirm die ihm als Argument übergebene Zeichenkette und von date(1) wird anschließend das aktuelle Datum und die Uhrzeit ausgegeben.
Wenn die Shell bereit ist, vom Terminal eine Kommandozeile einzulesen, so zeigt sie dem Nutzer diese Bereitschaft durch Ausgabe von "$" an. Falls die Shell nach Einlesen eines Zeilenende-Zeichens feststellt, daß die vorangegangenen Eingaben noch nicht vollständig sind und zur Abarbeitung des gegebenen Kommandos weitere Eingaben benötigt werden (z.B. wurde der abschließende Apostroph für eine Zeichenkette vergessen), so teilt es dies dem Nutzer durch Ausgabe von ">" auf dem Bildschirm mit. Beide Zeichen bzw. Zeichenketten für Eingabeanforderungen (Promt-Strings) kann der Nutzer nach seinem Belieben verändern. In unserem Beispiel für ein .profile geschah dies für den zweiten Prompt, so daß anstelle von ">" auf dem Bildschirm dieses Nutzers weiter: erscheinen würde. Analog ließe sich der erstgenannte Prompt zum Beispiel durch

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

verändern.

3. Arbeit mit Shell-Prozeduren

Der Kommandointerpreter Shell kann selbst in einem Kommando aufgerufen werden. Damit besteht die Möglichkeit (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 ausgeführt werden. Die Argumente arg1, arg2, ... sind dabei in der angegebenen Reihenfolge den Stellungsparametern "$1", "$2", ... zugeordnet und unter diesen Namen in der Shell-Prozedur verfügbar. Das Kommando sh kann unter Wirkung einer ganzen Reihe von options (Optionen) ausgeführt werden, die zum Beispiel Mechanismen zur Kontrolle der Abarbeitung der Shell-Prozedur bereitstellen (siehe Abschnitt 3.5. sowie sh(1) für die vollständige Beschreibung aller Optionen). Enthält das File compil z. B. die Kommandos

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

so führt die Eingabe der Kommandozeile

```
sh compil test.c
```

zur sukzessiven Abarbeitung der Kommandos in der Shell-Prozedur compil,,, wobei anstelle von "$1" test.c eingesetzt ist, d.h. das C-Programm test.c wird übersetzt (einschließlich Laden), wobei das ausführbare Programm a.out entsteht. Dieses wird durch das Kommando mv in outprog umbenannt und anschließend abgearbeitet.
Falls dem File compil (mit dem Kommando chmod))) das Attribut "ausführbar" (executable) zugewiesen wurde, so kann diese Shell-Prozedur ohne Aufruf des Kommandointerpreters ssshhh abgearbeitet werden. Die Kommandozeile

```
compil test.c
```

hat dann die gleiche Wirkung wie oben beschrieben.
Shell-Prozeduren sind ein wirksames Mittel zur Vereinfachung der Arbeit am Rechner. Da sie keiner Compilation bedürfen, 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 während der Abarbeitung der Prozedur unverändert. Damit können solche Prozeduren als Filter benutzt werden.

3.1. Shell-Variable

Die Shell bietet die Möglichkeit, Variablen, sowohl in Prozeduren als auch während der interaktiven Arbeit, Zeichenketten zuzuweisen. Auf diese Weise läßt sich z.B. das Schreiben häufig benutzter Namen verkürzen. So weist die Eingabezeile

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

der die source den Wert /usr/src/sys/cmd zu. In einem folgenden Kommando kann diese Zeichenkette über die Variablensubstitution $source, also durch Voranstellen eines "$" vor den Namen der Variablen, erreicht werden. Das Kommando

```
echo $source
```

liefert dann als Ausgabe nach Standard-Output

```
/usr/src/sys/cmd
```

Eine leere Zeichenkette wird einer Variablen zum Beispiel durch

```
empty=
```

zugewiesen.
Variablennamen müssen mit einem Buchstaben beginnen und können sonst aus Buchstaben, Ziffern und dem Unterstrich bestehen. Shell-Variablen können innerhalb von Kommandozeilen in eine Zeichenkette eingefügt werden. Zur Begrenzung des Variablen- oder Parameternamens dienen dabei geschweifte Klammern. Im Beispiel

```
mybin=/usr/wrk/th/bin/
cp a.out ${mybin}plp
```

wird das File a.out aus der aktuellen Directory nach /usr/wrk/th/bin/plp kopiert. Hätte die zweite Zeile die Gestalt

```
cp a.out $mybinplp
```

gehabt, so würde Shell versuchen als zweites Argument den Wert der Variablen mybinplp einzusetzen und diesen an cccppp zu übergeben.

Durch die Shell werden dem Nutzer einige spezielle Parameter bzw. Variablen zur Verfügung gestellt, deren Bedeutung nun kurz erläutert werden soll.
Unter "$0" ist der Wert des nullten Arguments des execute-Systemrufes verfügbar, der die Ausführung des laufenden Prozesses bewirkte. Dies ist i.a. der Name des auszuführenden Programms selbst.

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

`$#` gibt (dezimal) die Anzahl der Stellungsparameter einer die Shell-Prozedur an.

`$-` liefert die für den laufenden die Shell-Pozeß gesetzten Optionen (z.B. -x, -v).

$$ liefert die Prozeßnummer des gerade laufenden Prozesses (dezimal). Jeder Prozeß hat eine ihm vom Betriebssystem eindeutig zugeordnete Prozeßnummer. Diese Eindeutigkeit ermöglicht zum Beispiel die Benutzung der Prozeßnummer bei der Bildung von Namen für temporär benötigte Files. Dieses Prinzip wird auch von vielen MUTOS-Dienstprogrammen benutzt. Das Inhaltsverzeichnis eines mit dump(1) beschriebenen Magnetbandes läßt sich zum Beispiel folgendermaßen auf einem temporären File zwischenspeichern

```
dumpdir  >/tmp/ddir$$
.........
rm  /tmp/ddir$$.
```

Jedes MUTOS-Kommando liefert nach Beendigung seiner Arbeit einen Rückkehrwert (Exit-Status). Bei den meisten Kommandos ist dieser Wert gleich Null, wenn das Kommando erfolgreich abgearbeitet wurde, und ungleich Null beim Auftreten eines Fehlers. Diese Rückkehrwerte dienen beispielsweise als Testgrößen für die weitere Abarbeitung in den if- und while-Konstrukten, wie in den Abschnitten 3.2.1. und 3.2.2. beschrieben. Unter dem Namen "$?" ist jeweils der dezimale Rückkehrwert des zuletzt abgearbeiteten Kommandos als Zeichenkette verfügbar.
Neben diesen durch die Shell mit Werten belegten Variablen, gibt es weitere, die eine besondere Bedeutung für die Shell haben. Sie sollten nur in der für sie vorgesehenen Weise benutzt werden.

$HOME Der Wert der Variablen HOME stellt den Pfadnamen der "Home-Directory" 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 File-System-Hierarchie.

```
cat /usr/wrk/th/text
```

bewirkt, daß der Inhalt des Files text auf dem Bildschirm ausgegeben wird. Die Lage von text innerhalb des Filesystems wird dabei durch die Folge der Directories angegeben, die von der Wurzel "/" des Filesystems aus durchlaufen werden müßten, um zu text zu gelangen. Beginnt der Pfadname eines Files in einer Kommandozeile nicht mit "/", so erfolgt der Zugriff zu diesem File von der aktuellen Directory aus, d.h. von der Stelle innerhalb des File-Systems, an der sich der Nutzer gerade befindet. Eine Bewegung innerhalb des File-Systems ist mit dem Kommando cd(1) möglich. Der Pfadname der aktuellen Directory wird durch das Kommando pwd(1) geliefert. Das gleiche Ergebnis, wie die oben angebene Kommandozeile liefert

```
cd usr/wrk/th
cat text
```

Das Standardargument des Kommandos cd ist $HOME. Wird also cd ohne Argument aufgerufen, so entspricht dies dem Kommando

```
cd $HOME
```

und falls an irgendeiner Stelle (z.B. im .profile des Nutzers) gesetzt wurde

```
HOME=/usr/wrk/th
```

so läßt sich die Ausgabe des Inhaltes von text realisieren durch

```
cd
cat text
```

Der Standardwert, mit dem HOME belegt ist, ist die im Kennwort-File etc/passwd für jeden Nutzer eingetragene Login-Directory, welche auch immer zu Beginn der Arbeit dieses Nutzers (nach login) die aktuelle Directory für ihn ist.

$PATH Unter diesem Namen sind Directories aufgelistet, in denen nach ausführbaren Files gesucht werden soll. Die Ausführung eines Kommandos erfordert, daß zuerst das ausführbare File (Programm) mit dem angegebenen Namen (dem Kommandonamen) innerhalb der File-System-Hierarchie gesucht wird, um dann über execute(2) dessen Ausführung einzuleiten. Damit die Kommandonamen nicht unbedingt als vollständige Pfadnamen, d.h. ausgehend von "/" angegeben werden müssen, gibt es Standardsuchpfade, die immer durchlaufen werden, wenn ein Kommando abgearbeitet werden soll, und wenn nicht durch ein Neusetzen von PATH etwas anderes vorgeschrieben wurde. Diese Standardpfade sind die aktuelle Directory, /bin und /usr/bin. Durch Neusetzen von PATH kann jeder Nutzer eine ihm genehme änderung dieses Standards vornehmen. Nach dem Setzen von

```
PATH=:/usr/wrk/th/bin:/bin:/usr/bin
```

werden die Kommandos in der festgelegten Reihenfolge in den durch ":" getrennten Directories gesucht, zuerst also in der aktuellen Directory (dargestellt durch die leere Zeichenkette vor dem ersten ":"), dann in /usr/wrk/th/bin, wo zum Beispiel die privaten Dienstprogramme des Nutzers untergebracht sind, und schließlich in /bin sowie in /usr/bin, den Directories in denen die MUTOS-Dienstprogramme enthalten sind.
Diese Suchstrategie ist außer Kraft gesetzt, wenn der Kommandoname (nicht an erster Stelle) ein "/" enthält. In diesem Falle wird nur ein einfacher Zugriff von der aktuellen Directory aus gemacht.

$PS1 Diese Zeichenkette (Prompt-String) wird auf dem Bildschirm als Eingabeanforderumg (Prompt) ausgegeben, wenn die Shell zur interaktiven Arbeit bereit ist. Standard ist "$".

$PS2 Wird diese Zeichenkette durch Shell auf dem Bildschirm als Eingabeanforderung ausgegeben, so sind weitere Eingaben des Nutzers erforderlich, um bereits gemachte Eingaben gemäß der der Shell-Syntax zu vervollständigen. Standard ist ">".

$IFS Diese Variable enthält die Zeichen, die die Shell zusätzlich zu den die Shell-Sonderzeichen als Trennzeichen zwischen den Worten in Kommandozeilen interpretieren soll. Standard sind Leerzeichen, Tabulator und Zeilenende-Zeichen (siehe auch Abschnitt 4.5.).

$MAIL Diese Variable kann als Wert den Namen eines Files bekommen, in welches Mitteilungen von anderen an den betreffenden Nutzer abgelegt werden sollen. Während des interaktiven Verkehrs mit dem Nutzer überprüft die Shell vor jeder Ausgabe der Prompt-Zeichenkette "$", ob das in der Variablen MAIL festgelegte File seit der letzten überprüfung modifiziert worden ist. Wird eine Modifikation festgestellt, d.h. wurde inzwischen eine Nachricht in diesem File abgelegt, so erfolgt vor der Ausgabe von "$" die Ausschrift "you have mail" auf dem Bildschirm. Eine Standardbelegung von MAIL gibt es nicht. Wird dieser Variablen durch den Nutzer kein File-Name zugeordnet, so ist dieser Mitteilungsmechanismus nicht aktiviert. Soll er jedoch genutzt werden, so empfiehlt es sich, die Variable MAIL im .profile zu setzen.
Wird zum Beispiel

```
MAIL=/usr/spool/mail/th
```

gesetzt, so kontrolliert die Shell das vom MUTOS-Dienstprogramm mail(1) benutzte Nachrichten-File des Nutzers th , und informiert ihn darüber, ob während seiner Arbeit Nachrichten für ihn eingegangen sind. über die durch mail(1) eingegangenen Nachrichten wird sonst nur nach Ausführung von login(1) informiert.

3.2. Programmablauf-Steuerung

Die Shell stellt vier Verzweigungs- bzw. Schleifenmechanismen zur Steuerung des Ablaufs der Kommandoabarbeitung (vor allem innerhalb von Shell-Prozeduren) bereit. Abhängig vom Rückkehrcode (Exit-Status) des zuletzt abgearbeiteten Kommandos wird im if- und im while-Konstrukt über den weiteren Ablauf entschieden, während der case- und der for-Konstrukt eine daten- bzw. zeichenkettengesteuerte Verzweigung bzw. Schleife darstellen.

3.2.1. if-Verzweigung

Die allgemeine Form des if-Kommandos hat die Gestalt:

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

wobei der else-Zweig optionalen Charakter hat, also nicht angegeben werden muß. Dabei steht Kommandoliste jeweils für eine Folge von einem oder mehreren Kommandos, die durch Semikolon oder Zeilenende-Zeichen voneinander getrennt sind. Wenn der Rückkehrwert des letzten einfachen Kommandos aus der Kommandoliste1 gleich Null ist, so wird die Kommandoliste2 abgearbeitet, anderenfalls die Kommandoliste3.
Ein einfaches Beispiel für die Benutzung des if-Kommandos ist

```
if test -d $1
then echo "$1 is a directory"
fi
```

Das aufgerufene Programm test ist ein MUTOS-Dienstprogramm, das speziell für die Benutzung in Shell-Prozeduren geschaffen wurde. In unserem Beispiel wird durch die Option -d festgelegt, daß test den Rückkehrwert Null liefert, falls "$1"

eine Directory ist, d.h. falls der Shell-Prozedur, in der die im Beispiel angeführten Kommandozeilen auftreten, als erstes Argument der Name einer Directory übergeben wurde.
Für 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-Sonderzeichen "&&" und "||" .
So läßt sich die Kommandofolge

```
if Kommandoliste1
then Kommandoliste2
fi
```

auch schreiben als

```
Kommandoliste1 && Kommandoliste2
```

Während durch

```
Kommandoliste1 || Kommandoliste2
```

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

3.2.2. while-Schleife

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

```
while Kommandoliste1
do Kommandoliste2
done
```

Falls der Rückkehrwert des letzten einfachen Kommandos aus der Kommandoliste1 gleich Null ist, so wird die Kommandoliste2 abgearbeitet und anschließend wieder

die Kommandoliste1 mit dem entsprechenden Test des Rückkehrwertes. Diese Schleife wird solange durchlaufen, bis bei der Abarbeitung der Kommandoliste1 ein Rückkehrwert 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 die Beendigung der Schleife im entgegengesetzten Fall, d. h., wenn der Rückkehrwert der Kommandoliste1 gleich Null ist. Damit kann zum Beispiel durch

```
until test -d directory
do sleep 300
done
cp *.c directory
```

erreicht werden, daß in Abhängigkeit von den Ergebnissen anderer Prozesse (von anderen Nutzern) bestimmte Kommandos ausgeführt werden. Mit Pausen von fünf Minuten wird immer wieder überprüft, ob (evtl. durch einen anderen Nutzer) die Directory directory eingerichtet wurde und erst wenn dies geschehen ist, werden mit cp(1) alle auf .c endenden Files aus der aktuellen Directory nach dort kopiert.

3.2.3. for-Schleife

Diese Möglichkeit für die Programmierung einer Schleife findet sehr häufig in Shell-Prozeduren Anwendung. 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 für jede dieser Belegungen wird einmal die Kommandoliste abgearbeitet.
Enthält zum Beispiel 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 einl.1
```

das fünfmalige Drucken des Files einl.1 auf dem Zeilendrucker mit Nummerierung der Exemplare in der durch pr(1) erzeugten überschrift.
Die eigentliche Anwendung des for-Kommandos liegt aber wohl im Durchlaufen aller Argumente einer Shell-Prozedur. Dafür läßt sich die verkürzte Schreibweise

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

verwenden. Bei Weglassen des Teiles iiinnn Wort1 Wort2 in $* ein, so daß zum Beispiel durch die Prozedur compall mit dem Inhalt

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

eine übersetzung aller als Argumente übergebener C-Quellen erreicht wird. Der Aufruf

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

liefert, fehlerfreie Quellen file1.c und file2.c vorausgesetzt, die zugehörigen Objekt-Files file1.o und file2.o.

3.2.4. case-Verzweigung

Der case-Konstrukt bietet die Möglichkeit der Mehrwegverzweigung in Shell-Prozeduren in Abhängigkeit vom Aussehen bestimmter Zeichenketten. Die allgemeine Form des case-Kommandos ist

```
case Wort in
     Muster1 ) Kommandoliste1 ;;
     Muster2 ) Kommandoliste2 ;;
esac
```

Die Zeichenkette Wort wird in der angebenen Reihenfolge nacheinander mit den Zeichenketten Muster1, Muster2 usw. verglichen. Wird eine übereinstimmung der verglichenen Zeichenketten festgestellt, so erfolgt die Abarbeitung der zugehörigen Kommandoliste. Damit ist die Ausführung des case-Kommandos beendet, d.h. eventuelle weitere übereinstimmungen von Wort und einem der Muster werden nicht beachtet.
Illustrationsbeispiel für diesen Konstrukt sei das folgende Fragment der Shell-Prozedur mtsor :

```
.........
ISB=false
ISV=false
case $1 in
     -b | -B  )   ISB=true ;;
     -v | -V  )   ISV=true ;;
           *  )   echo "unknown flag $1" ;;
esac

.........

case $2 in
     lp | tty )   if $ISB
                  then    ...
                  fi

                  if $ISV
                  then    ...
                  fi      ;;

                  .........

esac
.........
```

Dieser Shell-Prozedur können zwei Argumente übergeben werden, von denen das erste der Auswahl bestimmter möglicher Funktionen von mtsor dient. Die erlaubten Optionen sind -b bzw. -B sowie -v bzw. -V. Zwischen Groß- und Kleinbuchstaben soll also nicht unterschieden werden. Daher sind die entsprechenden Muster im

ersten case-Konstrukt auch als Alternativmuster angegeben. Durch "|" getrennt, können in case-Konstrukten verschiedene Muster angegeben werden, deren übereinstimmung mit der Vergleichszeichenkette (in unserem Beispiel "$1") zur Abarbeitung derselben Kommandoliste führt. Das Merken der angegebenen Option geschieht durch Setzen der Variablen ISB bzw. ISV. Dabei sind true und false keine logischen Werte sondern Namen von MUTOS-Dienstprogrammen, deren einzige Funktion darin besteht, stets die Rückkehrwerte Null bzw. ungleich Null zu liefern. Diese Eigenschaft wird bei dem späteren Aufruf dieser Standard-Kommandos durch

```
        if $ISB
```

bzw.

```
        if $ISV
```

ausgenutzt.
In den Mustern des case-Konstruktes können die Sonderzeichen benutzt werden, die im Abschnitt 2.5. als File-Namen-Erweiterungszeichen beschrieben wurden. So hätte

```
        -b | -B ) ...
```

auch als

```
        -[bB] ) ...
```

geschrieben werden können. Daraus ergibt sich auch die Möglichkeit ein Standard-Muster anzugeben. Wird "*" als letztes Muster in einem case-Konstrukt verwendet, so ist gesichert, daß die zugehörige Kommandoliste immer abgearbeitet wird, falls bei keinem der anderen Muster eine übereinstimmung mit der vorgegebenen Zeichenkette festgestellt wurde. In unserem Beispiel erfolgt in diesem Falle eine Ausschrift mittels echo(1). Sollen in einem Muster die Sonderzeichen "*", "?" und "[...]" frei von ihrer Bedeutung als Shell-Sonderzeichen auftreten, so müssen sie, wie bereits in Abschnitt 2.6. beschrieben, apostrophiert werden.

3.3. Eingabepakete

Einige MUTOS-Dienstprogramme verlangen während ihrer Abarbeitung Eingaben über Standard-Input. Sollen solche Programme in Shell-Prozeduren integriert werden, so ist es angebracht, ihnen die geforderten Eingaben als Eingabepaket zur Verfügung zu stellen. Der folgende Editor-Aufruf innerhalb einer Shell-Prozedur ist ein Beispiel für die Benutzung von Eingabepaketen.

```
        ed text <<ende
        g/Mutos/s//MUTOS/g
        w
        ende
```

Die Zeilen ab <<ende bis zum erneuten Auftreten von ende werden durch die Shell als Standard-Input für das Kommando ed zur Verfügung gestellt. Die Beispielprozedur realisiert damit, daß im File text jedes Vorkommen der Zeichenkette Mutos in MUTOS umgewandelt wird. Anstelle von ende kann dabei irgendeine beliebige Zeichenkette zur Begrenzung 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 zugehörigen Kommandos wird, erfolgt durch die Shell ggf. eine Parametersubstitution. So könnte eine Prozedur edi die Zeilen

```
        ed $3 <<!!
        g/$1/s//$2/g
        w
        !!
```

enthalten. Das Kommando

```
        edi Mutos MUTOS text
```

würde dann zum gleichen Resultat führen, wie das zuerst angeführte Beispiel. Die Parametersubstitution kann verhindert werden, wenn "$" durch Voranschreiben von "\" apostrophiert wird. So ließe sich die Shell-Prozedur edi auch schreiben als

```
        ed $3 <<@
        1,\$s/$1/$2/g
        w
        @
```

Die Wirkung ist dieselbe wie oben, nur daß durch das veränderte Editor-Kommando ein "?" ausgegeben wird, falls im File "$3" keine Zeichenkette, die gleich "$1" ist, gefunden wurde. Soll die Substitutionsausführung für das gesamte Eingabepaket unterdrückt 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
```

Für alle beim Aufruf von edall als Argumente übergebenen Files werden die in dem Eingabepaket beschriebenen Substitutionen ausgeführt.

3.4. Kommandogruppen

Kommandos können auf zwei Arten zu Kommandogruppen zusammengefügt werden:

- durch Einschließen mehrerer Kommandos in geschweifte Klammern:

```
        { Kommandoliste }
```

- durch Einschließen in runde Klammern:

```
        ( Kommandoliste )
```

Im ersten Fall wird die Kommandoliste ohne Besonderheiten abgearbeitet. Im zweiten Fall wird zur Abarbeitung der eingeklammerten Kommandos ein eigener Prozeß eröffnet, oder anders ausgedrückt, die in runde Klammern eingeschlossenen Kommandos werden in einem eigenen Shell-Unterprozeß abgearbeitet.
Durch die Kommandofolge

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

würde das File file1 in der Directory dir1 umbenannt in file1a und die aktuelle

Directory für den Nutzer, der diese Kommandofolge eingegeben hat, wäre das durch cd erreichte dir1.
Die Kommandofolge

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

dagegen, hätte zwar bezüglich der Umbenennung von file111 dieselbe Wirkung wie oben, aber nach Abarbeitung dieser Kommandos hätte sich die aktuelle Directory des Nutzers nicht verändert, da das Kommando cd, ausgeführt im Shell-Unterprozeß, keine Wirkung in der aufrufenden Shell-Ebene hat.

3.5. Testen von Shell-Prozeduren

Zur Unterstützung des Tests von Shell-Prozeduren stehen zwei Mechanismen zur Verfügung, die eine Verfolgung des Programmablaufs innerhalb der Shell-Prozeduren gestatten (tracing). Der erste Mechanismus realisiert, daß jede eingelesene Kommandozeile der Shell-Prozedur auf dem Bildschirm ausgegeben wird. Damit wird besonders die überprüfung der syntaktischen Richtigkeit unterstützt. 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 für die gesamte Shell-Prozedur sondern nur für bestimmte Teile gültig sein, so kann er innerhalb der Prozedur an einer beliebigen Stelle durch Einfügen des Kommandos

```
set -v
```

in Gang gesetzt werden. Dieser, wie auch der zweite Tracing-Mechanismus, wird durch das Kommando

```
set -
```

an einer anderen Stelle der Shell-Prozedur wieder außer Kraft gesetzt.
Für den zweiten Mechanismus gelten die gleichen Regeln, um ihn ein- bzw. auszuschalten. Anstelle von v ist dabei x zu schreiben. Ist dieser zweite Tracing-Mechanismus für eine Shell-Prozedur aktiv, so werden auf dem Bildschirm die Kommandos ausgegeben (mit einem vorangestellten +), die gerade von der Shell ausgeführt werden. Dabei sind bereits alle Parametersubstitutionen erfolgt, so daß man sich mit Hilfe dieses Mechanismus Klarheit über das Aussehen der tatsächlich ausgeführten Kommandos verschaffen kann.
In Verbindung mit der Option -v ist die Option -n von Bedeutung. Wenn -n gesetzt ist, so wird die Ausführung aller folgenden Kommandos unterdrückt. Daher sollte man diese Option auch nicht für den am Bildschirmterminal laufenden Shell-Prozeß angeben, denn das Kommando

```
set -n
```

am Terminal eingegeben, verhindert die Ausführung aller danach eingegebenen Kommandos. Man kann sich dann nur durch Eingabe der End-of-File-Folge (CTRL-Z) aus dieser Misere retten.

4. Weitere Möglichkeiten der Shell

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

4.1. Parameterübergabe

Einer Shell-Prozedur können bei deren Aufruf, wie schon im 3. Kapitel geschildert, Argumente übergeben werden, die dann innerhalb der Prozedur den Stellungsparametern "$1", "$2" usw. zugeordnet sind. Variablen können einer Shell-Prozedur aber auch durch "Exportieren" zugänglich gemacht werden. Das Kommando

```
export source
```

legt zum Beispiel fest, daß die Variable source eine solche exportierbare Variable sein soll. Allen später aufgerufenen Prozeduren werden Kopien dieser Variablen übergeben. Modifikationen der Werte von exportierten Variablen haben keinen Einfluß auf den Wert der entsprechenden Variablen in der aufrufenden Ebene (beim Exporteur). Dies ordnet sich in das allgemeingültige Prinzip der Shell-Arbeit ein, nachdem Aktionen in einer aufgerufenen Prozedur keinen Einfluß auf den Zustand der aufrufenden Prozedur haben, es sei denn, daß die aufrufende Ebene ausdrücklich derartige Veränderungen anfordert.
Soll verhindert werden, daß der Wert irgendeiner Variablen durch spätere Zugriffe verändert wird, so ist es möglich, diese Variable als "nur lesbar" zu vereinbaren. Das entsprechende Kommando bezüglich der Variablen source hat die Form

```
readonly source
```

und bewirkt, daß alle späteren Versuche, dieser Variablen Werte, d.h. Zeichenketten, zuzuweisen, erfolglos bleiben.

4.2. Kennwortparameter

Eine weitere Art der übergabe von Werten an Shell-Prozeduren wird realisiert, indem eine Wertzuweisung der Form Name=Wert dem Prozeduraufruf unmittelbar vorangestellt wird, wie in dem Beispiel

```
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 übergebenen Parameter heißen Kennwortparameter. Durch Angabe von -k als Option an die Shell werden alle Wertzuweisungen der Form Name=Wert, unabhängig von ihrer Position in der Kommandozeile, als Kennwortparameter an das Kommando übergeben. Durch

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

wird dieselbe Zuweisung erreicht, wie im obigen Beispiel. Die übrigen Argumente arg1, arg2, ... sind in days als die Stellungsparameter "$1", "$2", ... verfügbar.
Auch die Stellungsparameter können noch auf eine andere Art mit Werten belegt werden. Innerhalb von 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 verfügbar.

```
        set - *
```

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

4.3. Parametersubstitution

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

```
        echo $var
```

bzw.

```
        echo ${var}
```

liefert nichts als Ausgabe, falls die Variable var nicht mit einem Wert belegt ist. Für eine nichtbelegte Variable wird also die Null-Zeichenkette substituiert. Es ist aber auch möglich, anstelle eines nichtbelegten Parameters eine angebbare Zeichenkette zu substituieren. Dies geschieht zum Beispiel 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, so wird die Zeichenkette NOTSET substituiert und von echo ausgegeben. Statt der zu substituierenden Zeichenkette könnte 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 würde, falls var kein Wert zugewiesen wäre.

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 Ausführung dieses Kommandos noch nicht mit einem Wert belegt war. Stellungsparameter dürfen bei Benutzung des Operators "=" nicht anstelle von var auftreten!
Im Gegensatz zu beiden bisher erläuterten 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 führt 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 würde auch den Abbruch der Shell-Prozedur nach diesem Kommando bedeuten. Falls keine Zeichenkette wie STOP angegeben ist, erfolgt vor dem Abbruch der Shell-Prozedur die Ausgabe einer Standard-Meldung.
Soll die Abarbeitung einer Shell-Prozedur davon abhängig 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. - siehe sh(1), das keine Aktion auslöst. Es wird beispielsweise zum Einfügen 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 übergeben wurden - sei es durch "export" oder als Kennwortpameter). 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 Abarbeitung der zugehörigen Kommandozeile ausgeführt und durch seine nach Standard-Output gerichtete Ausgabe ersetzt wird. Das Kommando pwd, zum Beispiel, liefert als Ausgabe auf Standard-Output den Namen des aktuellen Directories. Wenn nun /usr/wrk/th die aktuelle Directory eines Nutzers ist, so weist die Kommandozeile

```
curdir=`pwd`
```

der Shell-Variablen curdir den Wert /usr/wrk/th zu. Die Kommandosubstitution schafft damit die Möglichkeit, Programme, die Zeichenketten verarbeiten, unkompliziert in Shell-Prozeduren einzubeziehen. Durch die Kommandosprache selbst werden das Zusammenfügen von Zeichenketten und das Mustererkennen und -ausfüllen im Zusammenhang mit der File-Namen-Generierung realisiert.
Kommandosubstitution kann überall dort durchgeführt werden, wo auch Parametersubstitution möglich ist (zum Beispiel auch in Eingabepaketen). Beide Arten von Substitutionen werden vor Beginn der File-Namen-Generierung für ein Kommando ausgeführt.
Ein weiteres Kommando, das sich zur Benutzung innerhalb von Kommandosubstitutionen anbietet, ist basename(1). Dieses Kommando entfernt von Zeichenketten einen angebbaren Suffix sowie den ggf. vorhandenen, auf "/" endenden, Präfix. Auf Pfadnamen angewendet, wird durch basename der "Stammname" des betreffenden Files auf Standard-Output geliefert. Die Shell-Prozedur

```
for filnam
do   cc $filnam
     mv a.out $HOME/bin/`basename $filnam .c`
done
```

bewirkt die übersetzung aller ihr als Argumente übergebenen C-Quell-Files und benennt das jeweils entstehende File a.out um. Ist /usr/th/cmd/pri.c eines der Argumente der Shell-Prozedur und /usr/th die Home-Directory des Nutzers, so entsteht nach der (fehlerfreien) übersetzung des C-Programms das ausführbare File usr/th/bin/pri.

Ein anderes Beispiel einer Kommandosubstitution könnte sein:

```
set`date`
echo Uhrzeit: $4
```

wodurch die Ausgabe Uhrzeit : 12 :00 :00 erzeugt wird, falls es nach der Rechnerzeit gerade zwölf Uhr mittags ist.

4.5. Substitutionen und Apostrophieren

Für die Argumente von Kommandos kann die Shell drei verschiedene Arten von Ersetzungen ausführen - Parametersubstitution, Kommandosubstitution und die File-Namen-Generierung. Die Reihenfolge ihrer Ausführung und die Auswirkungen der verschiedenen Arten des Apostrophierens dabei, sollen in diesem Abschnitt besprochen werden.

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

1.) Parametersubstitution (wie für $var)

2.) Kommandosubstitution (wie für date)
Beide Substitutionen werden jeweils nur einmal durchgeführt. Daher liefert das Kommando

```
echo $a
```

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

3.) Interpretation der Freiräume

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

```
echo ''
```

erzeugt eine leere Zeichenkette als erstes Argument für echo, während durch

```
echo $empty
```

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

4.) File-Namen-Generierung

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

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

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

vorgegebene Liste von Worten Wort1 , Wort2 usw. ausgeführt. Für das im case-Konstrukt durch

```
case Wort in
```

angegebene Wort ist keine File-Namen-Generierung möglich. Hier können lediglich Parameter- und Kommandosubstitutionen durchgeführt werden.
Neben den bereits beschriebenen Möglichkeiten 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 Anführungsstriche eingeschlossen. Der Unterschied zu den vorher beschriebenen Apostrophierungsmechanismen besteht darin, daß in einer durch Anführungsstriche eingeschlossenen Zeichenkette Parameter- und Kommandosubstitutionen ausgeführt werden. Eine Interpretation von Freiräumen und eine File-Namen-Generierung erfolgt nicht. Durch

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

werden auf dem Bildschirm die Ausgaben

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

erzeugt. Eine besondere Bedeutung innerhalb von Anführungsstrichen haben demnach die Zeichen:

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

Die folgende Tabelle gibt eine Gesamtübersicht darüber, welchen Sonderzeichen bei den einzelnen Apostrophierungsarten sowie bei der Kommandosubstitution eine besondere Bedeutung zukommt.

Sonderzeichen

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

Wobei 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 genügt, daß bei Substitutionen jeweils nur eine einfache Ersetzung durchgeführt 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 zur Ausführung 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 (für die durch Shell bereits eine Bearbeitung, wie zu Beginn dieses Abschnittes beschrieben, durchgeführt wurde) als Kommando an Shell übergeben werden, wodurch eine zweite Bearbeitung des eigentlichen Kommandos ausgelöst wird.

Dadurch ist nach dem Setzen von

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

auch ein Kommando der Art

```
$spr
```

möglich, welches das gleiche bewirkt, wie

```
anycommand | sort | pr
```

nämlich das Sortieren und Formatieren der Ausgaben von anycommand. Die Benutzung von eval in diesem Beispiel ist deshalb nötig, da nach einer Substitution ($spr) keine Interpretation von Sonderzeichen (wie "|") mehr erfolgt.

4.6. Kommandoausführung

Wenn in einem Kommando, wie in Abschnitt 4.5. beschrieben, alle notwendigen Ersetzungen vorgenommen sind, 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 ursprünglich arbeitenden Shell-Prozesses und enthält in seiner Ausführungsumgebung die gleichen geöffneten Files, wie der Parent-Prozeß (also mindestens stdin, stdout, stderr) sowie den gleichen Status für die einzelnen MUTOS-Signale (siehe Abschnitt 4.7.).
Sind im abzuarbeitenden Kommando E/A-Richtungsänderungen angegeben, so werden diese erst im Child-Prozeß wirksam, d.h. nach der Ausführung von fork. Somit haben E/A-Richtungsänderungen auch keinen Einfluß auf den ursprünglichen Shell-Prozeß und gelten immer nur für die Dauer der Abarbeitung des zugehörigen Kommandos, d. h. in dem gestarteten Child-Prozeß.
Als Ergänzung zu den im Abschnitt 2.3. vorgestellten elementaren Möglichkeiten von E/A-Richtungsänderungen, sollen nun einige weiterführende Bemerkungen zu diesem Thema gemacht werden.
Für die hinter einem Zeichen zur E/A-Richtungsänderung angegebene Zeichenkette, die die neue Ein- bzw. Ausgaberichtung angibt, können Parameter- und Kommandosubstitutionen ausgeführt werden. Eine Filenamen-Generierung und eine Interpretation von Freiräumen erfolgt für diesen Teil eines Kommandos nicht. Somit wird durch

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

der Inhalt des Files text nicht etwa an alle Files in der aktuellen Directory angefügt, die auf .t enden, sondern nur einmal in ein File mit dem Namen *.t geschrieben.

Kommandosubstitutionen können, analog zu den im Abschnit 3.3. bereits erwähnten Parametersubstitutionen, ebenfalls in Eingabepaketen benutzt werden.

Ein Illustrationsbeispiel dafür sei die folgende Shell-Prozedur edn:::

```
        str1=$1
        str2=$2
        while   true
        do      case $3 in
                        "")   exit    ;;
                        * )   ed $3 <<@
                              1,\$s/$str1/$str2/g
                              w `basename $3 .old`.new
        @
                                      ;;
                esac
                shift
        done
```

Der Aufruf von edn hat die Gestalt

```
        edn string1 string2 filename ...
```

und bewirkt, daß in allen angegebenen Files filenname . die Zeichenkette string1 durch string2 ersetzt wird. Endet der Filename auf .old, so wird diese Endung im Namen des neuen Files gestrichen. An den durch basename erzeugten "Stammnamen" des Files wird die Endung .new angefügt. Die alten Files bleiben erhalten. Das Bearbeiten aller als Argumente angegebenen Files wird durch das spezielle Shell-Kommando shift erreicht, durch welches vor dem erneuten Durchlaufen der Schleife die Stellungsparameter ab "$2", "$3", ... umbenannt werden in "$1", "$2", ... Dadurch hat $3 als Wert immer den Namen des nächsten zu editierenden Files.

Weitere Möglichkeiten der E/A-Richtungsänderung werden durch die folgenden Formen realisiert:

>&digit Der File-Deskriptor digit (eine Ziffer) wird unter Benutzung des Systemrufes dup(2) dupliziert und das entstandene Ergebnis dem Standard-Output (File-Deskriptor 1) zugeordnet, d.h. Ausgaben über 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 File-Deskriptor digit zugeordnet ist.

>&- Der Standard-Output wird geschlossen.

<&- Der Standard-Input wird geschlossen.

Für alle Formen von E/A-Richtungsänderungen ist es möglich, dem Zeichen zur Richtungsänderung eine Ziffer voranzustellen. Dies bewirkt, daß sich die auszuführende Richtungsänderung nicht auf den Standard-Input bzw. -Output bezieht, sondern auf den File-Deskriptor, der durch die angegebene Ziffer dargestellt ist. So wird zum Beispiel durch

```
        command 2>errfile
```

erreicht, daß bei Abarbeitung des Kommandos command die Meldungen, die über stderr (File-Deskriptor 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 über stderr und stdout zum gleichen File gerichtet sind, wie es im Standardfall ist, wo beide File-Deskriptoren dem Bildschirmterminal des Nutzers zugeordnet sind (Spezial-File (dev/tty*). Das Zusammenlegen der beiden Ausgabeströme wird dadurch erreicht, daß File-Deskriptor 2 ein Duplikat des File-Deskriptors 1 wird. Ein Anwendungsbeispiel für eine solche E/A-Richtungsänderung ist das Kommando nohup(1), das als Shell-Prozedur realisiert ist (siehe Abschnitt 4.7.).
Für die Umgebung von Hintergrundkommandos wird durch die Shell bezüglich 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 Hintergrundkommando und die Shell, versuchen dieselben Eingaben zu lesen. Ein solches Teilen der Eingaben könnte zu einem Durcheinander führen, wenn im Hintergrund ein Programm versehentlich ohne E/A-Richtungsänderung gestartet ist, das während seiner Arbeit Eingaben über Standard-Input verlangt.

4.7. Signalbehandlung

Im Betriebssystem MUTOS werden 15 verschiedene Signale behandelt (siehe signal(2) für ausführlichere Erläuterungen). Der folgende kurze Überblick gibt die Signale und ihre symbolischen Namen an:

1	SIGHUP
2	SIGINT
3*	SIGQUIT
4*	SIGILL
5*	SIGTRAP
6*	SIGIOT
7*	SIGEMT
8*	SIGFPE
9	SIGKILL
10*	SIGBUS
11*	SIGSEGV
12*	SIGSYS
13	SIGPIPE
14	SIGALRM
15	SIGTERM

Die mit einem "*" gekennzeichneten Signale führen zum Entstehen eines 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 Signals abgebrochen wird, ohne daß er zuvor noch irgendwelche Aktionen ausführen kann. Eine zweite Möglichkeit besteht darin, Signale zu ignorieren. In diesem Falle wird der Prozeß fortgesetzt, als hätte es das Ereignis, welches zum Senden des Signals geführt hat, nicht gegeben. Die dritte Möglichkeit ist das Auffangen von Signalen. Hierzu ist es notwendig, daß durch den Prozeß festgelegt wird, welche Aktionen auszuführen sind, wenn ein bestimmtes Ereignis eingetreten ist, d. h. das zugehörige Signal an den Prozeß gesendet wurde.
Diese Möglichkeiten der Signalbehandlung werden dem Shell-Benutzer auch auf der Ebene der Kommandosprache geboten. Die für eine Behandlung durch Shell relevanten Signale sind vor allem 1, 2, 3, 14 und 15. Die Shell selbst ignoriert das Signal 3, das einzige externe Signal, das zum Entstehen eines Core-Files führen könnte. Zur Signalbehandlung existiert das spezielle Shell-Kommando trap.

Normalerweise wird die Abarbeitung einer Shell-Prozedur abgebrochen, wenn an den zugehörigen Prozeß ein Signal gesendet wird (z.B. ein Interrupt-Signal vom Bedienterminal des Nutzers). Durch

```
trap action 2
```

wird das Signal 2 (Terminal-Interrupt) jedoch abgefangen. Empfängt der Prozeß dieses Signal, so wird die Abarbeitung der Shell-Prozedur nicht abgebrochen, sondern nur unterbrochen, und das Kommando action ausgeführt. Anschließend wird die Shell-Prozedur an der Stelle fortgesetzt, wo sie unterbrochen wurde. Wurden durch eine Shell-Prozedur beispielsweise Files angelegt, die temporären Charakter haben sollen, so muß dafür gesorgt werden, daß diese Files nicht nur bei normaler Beendigung der Prozedur, sondern auch bei einem Abbruch nach Empfang eines Signals gelöscht werden. Dazu könnte folgendes Kommando dienen:

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

Beim Empfang eines der Signale 1, 2, 3 oder 15 wird das temporäre File /tmp/ddir$$ gelöscht. Durch das spezielle Shell-Kommando exit wird die Shell-Prozedur danach beendet.
Ein Signal 0 gibt es in MUTOS nicht. Für das trap-Kommando wird diese Signalnummer deshalb dazu benutzt, um Aktionen zu definieren, die bei normaler Beendigung einer Shell-Prozedur auszuführen sind.

```
trap endfile 0
```

sorgt dafür, daß nach Beendigung der Prozedur, in der dieses Kommando erscheint, eeennndddfile 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 angegeben Signale abzuarbeitenden Kommandos, angegeben wird. Ein Beispiel dafür ist die Benutzung von trap im Kommando nohup(1):

```
trap "" 1 15
if test -t 2>&1  ; then
     echo "Sending output to 'nohup.out'"
     exec nice -5 $* >>nohup.out 2>&1
else
     exec nice -5 $* 2>&1
fi
```

Durch nohup werden alle diesem MUTOS-Kommando als Argumente übergebenen Kommandos mit einer niedrigeren Priorität (nice-5) abgearbeitet und deren Ausgaben nach nohup.out geschrieben, falls die File-Descriptoren 1 und 2 einem Terminal zugeornet waren. Die abzuarbeitenden Kommandos können durch die Signale 1 und 15 nicht unterbrochen werden.
Wird trap für bestimmte Signale ohne Angabe einer auszuführenden Aktion aufgerufen, so wird beim Empfang dieser Signale wieder die Standardaktion, d.h. Abbruch des Prozesses, ausgeführt. Ein solcher Aufruf wie

```
trap 2
```

dient daher dem Rücksetzen früher getroffener Festlegungen bzgl. des Signals 2. Durch

```
trap
```

wird eine Liste der den Signalen aktuell zugeordneten Aktionen ausgegeben.

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

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

Ausgehend von der aktuellen Directory wird nacheinander in alle Subdirectories verzweigt. Nach der Verzweigung wird jeweils der Name der Subdirectory ausgegeben. An dieser Stelle kann der Nutzer durch einen Interrupt die Abarbeitung von scan beenden. Ansonsten wartet scan durch das spezielle Shell-Kommando read auf eine Eingabe über Standard-Input. Wurde eine Eingabezeile gelesen, so wird deren Wortlaut der Variablen com zugewiesen, als Kommando aufgefaßt und abgearbeitet. Während der Abarbeitung dieses Kommandos ist scan nicht durch einen Terminal-Interrupt unterbrechbar. Wird End-of-File eingegeben, so erfolgt die Verzweigung in die nächste Subdirectory, da read in diesem Falle einen Rückkehrwert liefert, der ungleich Null ist. Und dies ist für while das Kriterium zum Beenden der Schleife.
Für Hintergrundkommandos wird durch die Shell eine spezielle Signalbehandlung organisiert. Solche Kommandos ignorieren die Signale 2 und 3, so daß diese Signale vom Terminal aus gesendet werden können, ohne die Arbeit der im Hintergrund laufenden Prozesse zu beeinflussen. Falls eine Shell-Prozedur im Hintergrund abgearbeitet wird, so kann ein in dieser Prozedur auftretendes trap-Kommando die Behandlung der Signale 2 und 3 nicht verändern.

4.8. Fehlerbehandlung

Die Behandlung von Fehlern, die während der Ausführung von Kommandos festgestellt werden, hängt 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 ein Terminal gebunden sind. Auch alle durch

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

aufgerufenen Shell-Prozesse gelten als interaktiv in diesem Sinne.
Ursachen für Fehler bei der Ausführung eines Kommandos können sein:

- Das Kommando (identifiziert durch seinen Namen) existiert nicht oder kann nicht ausgeführt werden (Zugriffsrechte).
- änderungen der E/A-Richtungen können nicht durchgeführt werden, z. B. wenn ein angegebenes File nicht existiert bzw. nicht geöffnet oder erstellt werden kann.
- Die Kommandoausführung wird durch den Empfang eines Signals (z. B. Signal 10 - Ausschrift "bus error") vorzeitig beendet.
- Die Kommandoausführung wird normal beendet, aber der Rückkehrwert des Kommandos ist ungleich Null.

Fehler, die aus den bisher aufgeführten Ursachen resultieren, bewirken (bis auf den letzten Fall) die Ausgabe einer Fehlernachricht und den übergang zur Abarbeitung des nächsten Kommandos in Shell-Prozeduren bzw. das Warten auf weitere Eingaben im interaktiven Fall.
Weitere Fehler können sein:

- Syntaxfehler, z.B. das Vergessen von done zum Beenden einer Schleife oder von ";;" zum Abgrenzen der einzelnen Fälle eines case-Konstruktes.
- Empfang eines Signals wie 2 (Terminal-Interrupt).
- Fehlerhafte Ausführung eines der speziellen Shell-Kommandos cd, eval, exec, ...

Fehler dieser Art bewirken den Abbruch der Abarbeitung einer Shell-Prozedur (nach Ende des gerade ausgeführten 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 irgendein Fehler festgestellt wurde.

4.9. Aufruf der Shell

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

-c string Die Shell liest Kommandos aus string ein und führt sie aus.

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

-i Eine so aufgerufene 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
```

eine interaktive Shell nicht mit beendet wird. Das Signal 2 wird abgefangen, aber keine zugehörige Aktion ausgeführt. Dadurch kann ein Wartezustand, ausgelöst durch das spezielle Shell-Kommando wait, mit dem Signal 2 unterbrochen werden.

5. Schlußbemerkungen

Nach der Lektüre dieser vorliegenden Schrift kann vom Leser nicht erwartet werden, daß er alles hier aufgeführte sofort in seine praktische Arbeit umsetzen kann. Die Ausnutzung der vielfältigen Möglichkeiten der Shell wird sich vielmehr erst im Laufe einer längeren Praxis unter MUTOS oder bei der Arbeit unter einer ähnlichen Nutzerschnittstelle (z.B. PSU an ESER-Rechnern) ergeben.
Als schnelle Nachschlaghilfe wird in dem folgenden Anhang eine übersicht zur Grammatik sowie zu den Sonderzeichen und reservierten Worten der Shell gegeben. Als Ergänzung und zum Nachschlagen ist es darüber hinaus empfehlenswert, die Shell-Beschreibung in der Programmtechnischen Beschreibung Teil1 (sh(1)) zu benutzen.

Anlage A 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, Zeilenendezeichen ; & | < > ( )
Name:	Zeichenfolge, die aus Buchstaben, Ziffern oder Unterstrichen bestehen kann
Ziffer:	0 1 2 3 4 5 6 7 8 9

Anlage B Sonderzeichen und reservierte Worte

a) Syntax

;	Trennung von Kommandos
&	Hintergrundkommandos
|	Pipe-Symbol
( )	Kommandogruppe
;;	Begrenzung eines Musterteils
&&	UND-Verknüpfung
||	ODER-Verknüpfung
<	Eingabe-Richtungsänderung
>	Ausgabe-Richtungsänderung
<<	Eingabepaket
>>	Anfügen von Ausgaben

b) Substitutionen

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

c) Muster

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

d) Apostrophieren

\	apostrophiert das nächstfolgende 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 müssen jeweils als erstes Wort eines Kommandos erschei-
nen. Ausnahmen sind in und }.