Shellprogrammierung

Autor: Ronald Schaten (*kahless@bigfoot.de*)
Layout: Johannes Kolb (*johannes.kolb@web.de*)
Layout: Matthias Kleine (*kleine_matthias@gmx.de*)
Layout: Matthias Hagedorn (*matthias.hagedorn@selflinux.org*)

Dieses Kapitel behandelt den fortgeschrittenen Umgang mit Bourne-Shell kompatiblen Shells, der insbesondere im Erstellen und Pflegen von Shell-Skripten Anwendung findet.

Inhaltsverzeichnis

1 Was ist die Shell?

1.1 Sinn und Zweck
1.2 Die Qual der Wahl

2 Wofür Shell-Programmierung?

3 Wie sieht ein Shell-Skript aus?

3.1 Grundsätzliches
3.1.1 HowTo
3.1.2 Rückgabewerte
3.2 Variablen
3.3 Vordefinierte Variablen
3.4 Variablen-Substitution
3.5 Quoting
3.6 Meta-Zeichen
3.7 Mustererkennung
3.8 Programmablaufkontrolle
3.9 Kommentare (#)
3.10 Auswahl der Shell (#!)
3.11 Null-Befehl (:)
3.12 Source (.)
3.13 Funktionen
3.14 Bedingungen ([ ])
3.15 if. . .
3.16 case. . .
3.17 for. . .
3.18 while. . .
3.19 until. . .
3.20 continue
3.21 break
3.22 exit
3.23 Befehlsformen
3.24 Datenströme

4 Wo sind Unterschiede zu DOS-Batchdateien?

5 Anhang A: Beispiele

5.1 Schleifen und Rückgabewerte
5.1.1 Schleife, bis ein Kommando erfolgreich war
5.1.2 Schleife, bis ein Kommando nicht erfolgreich war
5.2 Ein typisches Init-Skript
5.3 Parameterübergabe in der Praxis
5.4 Fallensteller: Auf Traps reagieren

6 Anhang B

6.1 Quellen

1 Was ist die Shell?

Die Shell ist ein Programm, mit dessen Hilfe das System die Benutzerbefehle verstehen kann. Aus diesem Grund wird die Shell auch oft als Befehls- oder Kommandointerpreter bezeichnet.

1.1 Sinn und Zweck

In einem klassischen Unix-System (ohne die grafische Oberfläche X) greifen die Benutzer über Terminals auf das System zu. Auf diesen Terminals können nur Textzeichen dargestellt werden. Um dem Benutzer die Arbeit mit dem System effektiv möglich zu machen, gibt es die Shell. Die Shell wird dabei für drei Hauptaufgaben benutzt:

* Interaktive Anwendung (Dialog)
* Anwendungsspezifische Anpassung des Unix-Systemverhaltens (Belegen von Umgebungsvariablen)
* Programmierung (Shell-Skripting). Zu diesem Zweck stehen einige Mechanismen zur Verfügung, die aus Hochsprachen bekannt sind (Variablen, Datenströme, Funktionen usw.).

Ursprünglich handelte es sich dabei um ein relativ einfaches Programm, der Bourne Shell (wird oft auch Standard-Shell genannt). Dies ist praktisch die **Mutter aller Shells**. Aus dieser entwickelten sich im Laufe der Zeit mehrere Varianten, die alle ihre eigenen Vor- und Nachteile mit sich bringen. Da es unter Unix kein Problem darstellt den Kommandointerpreter auszutauschen, stehen auf den meisten Systemen mehrere dieser Shells zur Verfügung. Welche Variante ein Benutzer verwenden möchte ist reine Geschmackssache.

1.2 Die Qual der Wahl

Um die Auswahl einer Shell zu erleichtern, werden hier die wichtigsten Varianten kurz vorgestellt. Sie sind aufgeteilt in Einfach- und Komfort-Shells. Die Komfort-Shells zeichnen sich durch komfortablere Funktionen zur interaktiven Bedienung aus, während die Einfach-Versionen üblicherweise für die Programmierung benutzt werden.

Einfach-Shells:

* Die ***Bourne***- oder ***Standard-Shell*** (`sh`) ist die kompakteste und einfachste Form. Sie bietet schon Mechanismen wie die Umlenkung der Ein- oder Ausgaben, Wildcards zur Abkürzung von Dateinamen, Shell-Variablen und einen Satz interner Befehle zum Schreiben von Shell-Prozeduren. Neuere Versionen beherrschen auch das Job-Controlling. Für die Entwicklung von Shell-Skripten sollte man sich auf diese Shell beschränken, da sie auf praktisch allen Systemen zur Verfügung steht. So bleiben die Skripte portabel.
* Die ***Korn-Shell*** (`ksh`), eine Weiterentwicklung der ***Bourne-Shell***, erlaubt das Editieren in der Befehlszeile. Außerdem gibt es hier History-Funktionen um auf zurückliegende Befehle zurückgreifen zu können, eine Ganzzahl-Arithmetik, verbesserte Möglichkeiten zur Mustererkennung, Aliase und das Job-Controlling. Ein Alias ist dabei eine Abkürzung für einen Befehl. Beispielsweise kann man das häufig benutzte `ls -la` einfach durch `la` ersetzen. Unter Job-Controlling versteht man einen Mechanismus, mit dessen Hilfe der Benutzer die Ausführung von Prozessen selektiv stoppen oder fortsetzen kann.
* Die ***C-Shell*** (`csh`) bietet ähnliche Annehmlichkeiten wie die *`Korn-Shell`*, lehnt sich aber in der Syntax sehr stark an die Programmiersprache C an. Sie sollte nach Möglichkeit nicht zur Shell-Programmierung benutzt werden, da sie an vielen Stellen nicht so reagiert, wie man es erwarten sollte.

Komfort-Shells:

* Die ***Bourne-Again-Shell*** (`bash`) ist voll abwärtskompatibel zur `sh`, bietet aber von allen Shells die komfortabelsten Funktionen für das interaktive Arbeiten. Da die Bash ein GNU-Produkt ist, ist sie die

Standard-Shell auf allen Linux-Systemen. Sie steht aber auch auf den meisten anderen Unixen zur Verfügung.

* Die ***T-C-Shell***(`tcsh`) verhält sich zur ***C-Shell*** wie die ***Bourne-Again-Shell*** zur Standard-Shell. Sie ist voll kompatibel, bietet aber zusätzliche Komfort-Funktionen.
* Die ***Stand-Alone-Shell*** (`sash`) ist vor allem nützlich für die **System-Recovery**. Sie kann gegen statische Bibliotheken gelinkt werden und beinhaltet bereits (teilweise vereinfachte) Formen von Standard-Systemkommandos. Kann man also, nach einem System-Crash, eine statisch gelinkte sash erreichen, ist es oft möglich, mit ihrer Hilfe das System wiederherzustellen. Nähere Informationen finden sich auf http://www.canb.auug.org.au/~dbell/ und http://www.baiti.net/sash/.

2 Wofür Shell-Programmierung?

Shell-Skripte werden im Wesentlichen aus zwei Gründen geschrieben: Erstens, weil man so ständig wiederkehrende Kommandos zusammenfassen kann, die dann mit einem einfachen Aufruf starten, und zweitens, weil man so einfache Programme schreiben kann, die relativ intelligent Aufgaben erledigen können.

Der erste Aspekt ist wichtig, wenn man beispielsweise regelmäßig auftretende Aufgaben erledigen möchte, wie z. B. das Backup von Log-Dateien. In dem Fall schreibt man sich ein Skript, das die Dateien archiviert, und sorgt dafür, dass dieses Skript in regelmäßigen Abständen aufgerufen wird (per Cron-Job).

Der zweite Fall tritt ein, wenn man eine mehr oder weniger komplexe Abfolge von Befehlen ausführen möchte, die voneinander abhängen. Ein Skript, das zum Beispiel eine Audio-CD kopieren soll, muss das Brennprogramm nur dann aufrufen, wenn der Einlesevorgang erfolgreich abgeschlossen wurde.

3 Wie sieht ein Shell-Skript aus?

Wie schon erwähnt, kann ein Shell-Skript beinahe alles, was eine richtige Programmiersprache kann. Dazu stehen mehrere Mechanismen zur Verfügung. Um den Umfang dieses Dokuments nicht zu sprengen, werden an dieser Stelle nur die wichtigsten vorgestellt.

3.1 Grundsätzliches

Zunächst soll die Frage geklärt werden, wie man überhaupt ein ausführbares Shell-Skript schreibt. Dabei wird vorausgesetzt, dass dem Benutzer der Umgang mit mindestens einem Texteditor (*vi*, *emacs* etc.) bekannt ist.

3.1.1 HowTo

Zunächst muss mit Hilfe des Editors eine Textdatei angelegt werden, in die der **Quelltext** geschrieben wird. Wie der aussieht, kann man anhand der folgenden Abschnitte und der Beispiele im Anhang erkennen. Beim Schreiben sollte man nicht mit Kommentaren geizen, da ein Shell-Skript auch schon mal sehr unleserlich werden kann.

Die Datei ist unter geeignetem Namen zu speichern. Bitte hierfür nicht den Namen **test** verwenden. Es existiert ein Unix-Systemkommando mit diesem Namen. Dieses steht fast immer eher im Pfad, d. h. beim Kommando `test` würde nicht das eigene Skript ausgeführt, sondern das Systemkommando. Dies ist einer der häufigsten und zugleich einer der verwirrendsten Anfängerfehler. Mehr zu dem `test`-Kommando unter 3.14

Danach muss sie ausführbar gemacht werden. Das geht mit dem Unix-Kommando `chmod`.

Rechte werden unter Unix getrennt für den Benutzer (`user`, `u`), die Gruppe (`group`, `g`) oder Andere (`others`, `o`) vergeben. Außerdem kann man die Rechte für Gruppen zusammen (`all`, `a`) setzen. Man kann getrennt die Rechte für das Lesen (`read`, `r`), das Schreiben (`write, w`) und die Ausführung (`execution, x`) einstellen. Um die Rechte zu setzen, muss man `chmod` in Parametern mitgeben, auf wen sich das Kommando bezieht, ob das Recht gesetzt (`+`) oder weggenommen (`-`) werden soll, und welche Rechte gemeint sind. Damit alle Benutzer das Skript ausführen dürfen, benutzt man das Kommando `chmod ugo+x name` oder einfach `chmod +x name`. Mit `chmod u+x name` hat nur der Besitzer der Datei Ausführungsrechte.

Dann kann das Skript gestartet werden. Da sich aus Sicherheitsgründen auf den meisten Systemen das aktuelle Verzeichnis nicht im Pfad des Benutzers befindet, muss man der Shell mitteilen, wo sie zu suchen hat: Mit `./name` wird versucht, im aktuellen Verzeichnis (`./`) ein Programm namens name auszuführen.

Auf den meisten Systemen befindet sich im Pfad der Eintrag `~/bin` bzw. Bedingungen ([])`/home/benutzername/bin`, das bedeutet, dass man Skripte, die immer wieder benutzt werden sollen, dort ablegen kann, so dass sie auch ohne eine Pfadangabe gefunden werden. Wie der Pfad genau aussieht kann man an der Shell durch Eingabe von `echo $PATH` herausfinden.

3.1.2 Rückgabewerte

Wenn unter Unix ein Prozeß beendet wird, gibt er einen Rückgabewert (auch Exit-Code oder Exit-Status genannt) an seinen aufrufenden Prozeß zurück. So kann der Mutterprozeß kontrollieren, ob die Ausführung des Tochterprozesses ohne Fehler beendet wurde. In einigen Fällen (z. B. `grep`) werden unterschiedliche Exit-Codes für unterschiedliche Ereignisse benutzt.

Dieser Rückgabewert wird bei der interaktiven Benutzung der Shell nur selten benutzt. Aber in der

Programmierung von Shell-Skripten ist er von unschätzbarem Wert. So kann das Skript automatisch entscheiden, ob bestimmte Aktionen ausgeführt werden sollen, die von anderen Aktionen abhängen. Beispiele dazu sieht man bei der Beschreibung der Kommandos `if` (3.15), `case` (3.16), `while` (3.18) und `until` (3.19), sowie in dem Abschnitt über Befehlsformen (3.23).

In der ***Bourne-Shell*** wird der Exit-Code des letzten aufgerufenen Programms in der Variable `$?` abgelegt. Üblicherweise geben Programme den Wert `0` zurück, bei irgendwelchen Problemen einen von `0` verschiedenen Wert. Das wird im folgenden Beispiel deutlich:

```
user@linux / # cp datei /tmp
user@linux / # echo $?

0

user@linux / # cp datie /tmp

cp: datie: Datei oder Verzeichnis nicht gefunden

user@linux / # echo $?

1
```

Normalerweise wird man den Exit-Code nicht in dieser Form abfragen. Sinnvoller ist folgendes Beispiel, in dem eine Datei erst gedruckt, und dann - falls der Ausdruck erfolgreich war - gelöscht wird:

```
user@linux / # lpr datei && rm datei
```

Näheres zur Verknüpfung von Aufrufen steht im Kapitel über Befehlsformen (3.23). Beispiele zur Benutzung von Rückgabewerten in Schleifen finden sich im Anhang unter A.1.

Auch Shell-Skripte können einen Rückgabewert an aufrufende Prozesse zurückgeben. Wie das geht, steht in dem Abschnitt zu exit (3.22).

3.2 Variablen

In einem Shell-Skript hat man - genau wie bei der interaktiven Nutzung der Shell - Möglichkeiten, über Variablen zu verfügen. Anders als in den meisten modernen Programmiersprachen gibt es aber keine Datentypen wie Ganzzahlen, Fließkommazahlen oder Strings. Alle Variablen werden als String gespeichert. Wenn die Variable die Funktion einer Zahl übernehmen soll, dann muss das verarbeitende Programm die Variable entsprechend interpretieren. (Für arithmetische Operationen steht das Programm expr zur Verfügung, siehe Zählschleifen-Beispiel unter 3.18)

Man muss bei der Benutzung von Variablen sehr aufpassen, wann die Variable expandiert wird und wann nicht. (Mit Expansion ist das Ersetzen des Variablennamens durch den Inhalt gemeint). Grundsätzlich werden Variablen während der Ausführung des Skriptes immer an den Stellen ersetzt, an denen sie stehen. Das passiert in jeder Zeile, unmittelbar bevor sie ausgeführt wird. Es ist also auch möglich, in einer Variable einen Shell-Befehl abzulegen. Im Folgenden kann dann der Variablenname an der Stelle des Befehls stehen. Um die Expansion einer Variable zu verhindern, benutzt man das Quoting (siehe unter 3.5).

Wie aus diversen Beispielen hervorgeht, belegt man eine Variable, indem man dem Namen mit dem Gleichheitszeichen einen Wert zuweist. Dabei darf zwischen dem Namen und dem Gleichheitszeichen **keine**

Leerstelle stehen, ansonsten erkennt die Shell den Variablennamen nicht als solchen und versucht, ein gleichnamiges Kommando auszuführen - was meistens durch eine Fehlermeldung quittiert wird.

Wenn man auf den Inhalt einer Variablen zugreifen möchte, leitet man den Variablennamen durch ein `$`-Zeichen ein. Alles was mit einem `$` anfängt wird von der Shell als Variable angesehen und entsprechend behandelt (expandiert).

3.3 Vordefinierte Variablen

Es gibt eine Reihe von vordefinierten Variablen, deren Benutzung ein wesentlicher Bestandteil des Shell-Programmierens ist. Die wichtigsten eingebauten Shell-Variablen sind:

`$n`	Aufrufparameter mit der Nummer n, n <= 9
`$*`	Alle Aufrufparameter
`$@`	Alle Aufrufparameter
`$#`	Anzahl der Aufrufparameter
`$?`	Rückgabewert des letzten Kommandos
`$$`	Prozeßnummer der aktiven Shell
`$!`	Prozeßnummer des letzten Hintergrundprozesses
`ERRNO`	Fehlernummer des letzten fehlgeschlagenen Systemaufrufs
`PWD`	Aktuelles Verzeichnis (wird durch `cd` gesetzt)
`OLDPWD`	Vorheriges Verzeichnis (wird durch `cd` gesetzt)

3.4 Variablen-Substitution

Unter Variablen-Substitution versteht man verschiedene Methoden um die Inhalte von Variablen zu benutzen. Das umfaßt sowohl die einfache Zuweisung eines Wertes an eine Variable als auch einfache Möglichkeiten zur Fallunterscheidung. In den fortgeschritteneren Shell-Versionen (`bash`, `ksh`)existieren sogar Möglichkeiten, auf Substrings von Variableninhalten zuzugreifen. In der Standard-Shell benutzt man für solche Zwecke üblicherweise den Stream-Editor `sed`. Einleitende Informationen dazu finden sich im Kapitel über die Mustererkennung (3.7).

Die folgenden Mechanismen stehen in der Standard-Shell bereit, um mit Variablen zu hantieren. Bei allen Angaben ist der Doppelpunkt optional. Wenn er aber angegeben wird, muss die Variable einen Wert enthalten.

`Variable = Wert`	Setzt die Variable auf den Wert.
`${Variable}`	Nutzt den Wert von Variable. Die Klammern müssen nicht mit angegeben werden, wenn die Variable von Trennzeichen umgeben ist.
`${Variable:-Wert}`	Nutzt den Wert von Variable. Falls die Variable nicht gesetzt ist, wird der Wert benutzt.
`${Variable:=Wert}`	Nutzt den Wert von Variable. Falls die Variable nicht gesetzt ist, wird der Wert benutzt, und Variable erhält den Wert.
`${Variable:?Wert}`	Nutzt den Wert von Variable. Falls die Variable nicht gesetzt ist, wird der Wert ausgegeben und die Shell beendet. Wenn kein Wert angegeben wurde, wird der Text **parameter null or not set** ausgegeben.
`${Variable:+Wert}`	Nutzt den Wert, falls die Variable gesetzt ist, andernfalls nichts.

Beispiele:

`$ h=hoch r=runter l=`	Weist den drei Variablen Werte zu, wobei l einen leeren Wert erhält.

`$ echo ${h}sprung`	Gibt hochsprung aus. Die Klammern müssen gesetzt werden, damit h als Variablenname erkannt werden kann.
`$ echo ${h-$r}`	Gibt hoch aus, da die Variable h belegt ist. Ansonsten würde der Wert von r ausgegeben.
`$ echo ${tmp-`date`}`	Gibt das aktuelle Datum aus, wenn die Variable tmp nicht gesetzt ist. (Der Befehl `date` gibt das Datum zurück)
`$ echo ${l=$r}`	Gibt runter aus, da die Variable l keinen Wert enthält. Gleichzeitig wird l der Wert von r zugewiesen.
`$ echo $l`	Gibt runter aus, da l jetzt den gleichen Inhalt hat wie r.

3.5 Quoting

Dies ist ein sehr schwieriges Thema, da hier mehrere ähnlich aussehende Zeichen völlig verschiedene Effekte bewirken. Unix unterscheidet allein zwischen drei verschiedenen Anführungszeichen. Das Quoten dient dazu, bestimmte Zeichen mit einer Sonderbedeutung vor der Shell zu 'verstecken' um zu verhindern, dass diese expandiert (ersetzt) werden.

Die folgenden Zeichen haben eine spezielle Bedeutung innerhalb der Shell:

`;`	Befehls-Trennzeichen
`&`	Hintergrund-Verarbeitung
`( )`	Befehls-Gruppierung
`|`	Pipe
`< > &`	Umlenkungssymbole
`* ? [ ] ~ + - @ !`	Meta-Zeichen für Dateinamen
`` ` ` (Backticks) ``	Befehls-Substitution (Die Backticks erhält man durch [shift] und die Taste neben dem Backspace.
`$`	Variablen-Substitution
`[newline] [space] [tab]`	Wort-Trennzeichen

Die folgenden Zeichen können zum Quoten verwendet werden:

`" " (Anführungszeichen)`	Alles zwischen diesen Zeichen ist buchstabengetreu zu interpretieren. Ausnahmen sind folgende Zeichen, die ihre spezielle Bedeutung beibehalten: `$ ` ` "`
`' ' (Ticks)`	Alles zwischen diesen Zeichen wird wörtlich genommen, mit Ausnahme eines weiteren `'` und `\`. (Die Ticks erhält man bei deutschen Tastaturen durch die Taste neben dem Backspace -- ohne [shift].)
`\ (Backslash)`	Das Zeichen nach einem `\` wird wörtlich genommen. Anwendung z. B. innerhalb von `" "`, um `"`, `$` und `` ` `` zu entwerten. Häufig verwendet zur Angabe von Leerzeichen (`space`) und Zeilenendezeichen, oder um ein `\`-Zeichen selbst anzugeben.

Beispiele:

```
user@linux / # echo 'Ticks "schützen" Anführungszeichen'
Ticks "schützen" Anführungszeichen
user@linux / # echo "Ist dies ein \"Sonderfall\"?"
```

```
Ist dies ein "Sonderfall"?
user@linux / # echo "Sie haben `ls | wc -l` Dateien in `pwd`"
Sie haben 43 Dateien in /home/rschaten
user@linux / # echo "Der Wert von \$x ist $x"
Der Wert von $x ist 100
```

3.6 Meta-Zeichen

Bei der Angabe von Dateinamen können eine Reihe von Meta-Zeichen verwendet werden, um mehrere Dateien gleichzeitig anzusprechen oder um nicht den vollen Dateinamen ausschreiben zu müssen. (Meta-Zeichen werden auch Wildcards, Joker-Zeichen oder Platzhalter genannt.)

Die wichtigsten Meta-Zeichen sind:

`*`	Eine Folge von keinem, einem oder mehreren Zeichen
`?`	Ein einzelnes Zeichen
`[abc]`	Übereinstimmung mit einem beliebigen Zeichen in der Klammer
`[a-q]`	Übereinstimmung mit einem beliebigen Zeichen aus dem angegebenen Bereich
`[!abc]`	Übereinstimmung mit einem beliebigen Zeichen, das nicht in der Klammer ist
`~`	Home-Verzeichnis des aktuellen Benutzers
`~name`	Home-Verzeichnis des Benutzers name
`~+`	Aktuelles Verzeichnis
`~-`	Vorheriges Verzeichnis

Beispiele:

`ls neu*`	Listet alle Dateien, die mit 'neu' anfangen
`ls neu?`	Listet 'neuX', 'neu4', aber nicht 'neu10'
`ls [D-R]*`	Listet alle Dateien, die mit einem Großbuchstaben zwischen D und R anfangen (Natürlich wird in Shell-Skripten -- wie überall in der Unix-Welt -- zwischen Groß- und Kleinschreibung unterschieden.)

3.7 Mustererkennung

Man unterscheidet in der Shell-Programmierung zwischen den Meta-Zeichen, die bei der Bezeichnung von Dateinamen eingesetzt werden und den Meta-Zeichen, die in mehreren Programmen Verwendung finden, um z. B. Suchmuster zu definieren. Diese Muster werden auch reguläre Ausdrücke (**regular expression**) genannt. Sie bieten wesentlich mehr Möglichkeiten als die relativ einfachen Wildcards für Dateinamen.

In der folgenden Tabelle wird gezeigt, in welchen Unix-Tools welche Zei- chen zur Verfügung stehen. Eine ausführlichere Beschreibung der Einträge findet sich danach.

	ed	ex	vi	sed	awk	grep	egrep	
`.`	X	X	X	X	X	X	X	Ein beliebiges Zeichen
`*`	X	X	X	X	X	X	X	Kein, ein oder mehrere Vorkommen des vorhergehenden Ausdrucks.
`^`	X	X	X	X	X	X	X	Zeilenanfang
`$`	X	X	X	X	X	X	X	Zeilenende
`\`	X	X	X	X	X	X	X	Hebt die Sonderbedeutung des folgenden Zeichens auf.
`[ ]`	X	X	X	X	X	X	X	Ein Zeichen aus einer Gruppe
`\( \)`	X	X		X				Speichert das Muster zur späteren Wiederholung.
`\{ \}`	X			X		X		Vorkommensbereich
`\< \>`	X	X	X					Wortanfang oder -ende
`+`					X		X	Ein oder mehrere Vorkommen des vorhergehenden Ausdrucks.
`?`					X		X	Kein oder ein Vorkommen des vorhergehenden Ausdrucks.
`|`					X		X	Trennt die für die Übereinstimmung verfügbaren Alternativen.
`( )`					X		X	Gruppiert Ausdrücke für den Test.

Bei einigen Tools (`ex`, `sed` und `ed`) werden zwei Muster angegeben: Ein Suchmuster (links) und ein Ersatzmuster (rechts). Nur die folgenden Zeichen sind in einem Ersatzmuster gültig:

	ex	sed	ed	
`\`	X	X	X	Sonderbedeutung des nächsten Zeichens aufheben.
`\n`	X	X	X	Verwendet das in `\( \)` gespeicherte Muster erneut.
`&`	X	X		Verwendet das vorherige Suchmuster erneut.
`~`	X			Verwendet das vorherige Ersatzmuster erneut.
`\u \U`	X			Ändert das (die) Zeichen auf Großschreibung.
`\l \L`	X			Ändert das (die) Zeichen auf Kleinschreibung.
`\E`	X			Hebt das vorangegangene `\U` oder `\L` auf.
`\e`	X			Hebt das vorangegangene `\u` oder `\l` auf.

Sonderzeichen in Suchmustern:

`.`	Steht für ein beliebiges *einzelnes* Zeichen, mit Ausnahme des Zeilenendezeichens.
`*`	Steht für eine beliebige (auch leere) Menge des einzelnen Zeichens vor dem Sternchen. Das vorangehende Zeichen kann auch ein regulärer Ausdruck sein. Beispielsweise steht `.*` für eine beliebige Anzahl eines beliebigen Zeichens
`^`	Übereinstimmung, wenn der folgende Ausdruck am Zeilenanfang steht.
`$`	Übereinstimmung, wenn der vorhergehende Azusdruck am Zeilenende steht.
`\`	Schaltet die Sonderbedeutung des nachfolgenden Zeichens ab.
`[ ]`	Steht für *ein* beliebiges Zeichen aus der eingeklammerten Gruppe. Mit dem Bindestrich kann man einen Bereich aufeinanderfolgender Zeichen auswählen (`[a-e]`). Ein Zirkumflex (`~`) wirkt als Umkehrung: `[^a-z]` erfaßt alle Zeichen, die keine Kleinbuchstaben sind. Ein Bindestrich oder eine schließende eckige Klammer am Listenanfang werden als Teil der Liste angesehen, alle anderen Sonderzeichen verlieren in der Liste ihre Bedeutung.

`\( \)`	Speichert das Muster zwischen `\(` und `\)` in einem speziellen Puffer. In einer Zeile können bis zu neun solcher Puffer belegt werden. In Substitutionen können sie über die Zeichenfolgen `\1` bis `\9` wieder benutzt werden.
`\{ \}`	Steht für den Vorkommensbereich des unmittelbar vorhergehenden Zeichens. `\{n\}` bezieht sich auf genau n Vorkommen, `\{n,\}` auf mindestens n Vorkommen und `\{n,m\}` auf eine beliebige Anzahl von Vorkommen zwischen n und m. Dabei müssen n und m im Bereich zwischen 0 und 256 liegen.
`\< \>`	Steht für ein Zeichen am Anfang (`\<`) oder am Ende (`\>`) eines Wortes.
`+`	Steht für ein oder mehrere Vorkommen des vorhergehenden regulären Ausdrucks = `\{1,\}`
`?`	Steht für kein oder ein Vorkommen des vorhergehenden Ausdrucks. = `\{0,1\}`
`|`	Übereinstimmung, wenn entweder der vorhergehende oder der nachfolgende reguläre Ausdruck übereinstimmen.
`( )`	Steht für die eingeschlossene Gruppe von regulären Ausdrücken.

Sonderzeichen in Ersatzmustern:

`\`	Hebt die spezielle Bedeutung des nächsten Zeichens auf.
`\n`	Ruft das n-te Muster aus dem Puffer ab (siehe oben, unter `\( \)`.) Dabei ist n eine Zahl zwischen 1 und 9.
`&`	Verwendet das vorherige Suchmuster erneut als Teil eines Ersatzmusters.
`~`	Verwendet das vorherige Ersatzmuster erneut im momentanen Ersatzmuster.
`\u`	Ändert das erste Zeichen des Ersatzmusters auf Großschreibung.
`\U`	Ändert alle Zeichen des Ersatzmusters auf Großschreibung.
`\l`	Ändert das erste Zeichen des Ersatzmusters auf Kleinschreibung.
`\L`	Ändert alle Zeichen des Ersatzmusters auf Kleinschreibung.
`\e`	Hebt das vorangegangene `\u` oder `\l` auf.
`\E`	Hebt das vorangegangene `\U` oder `\L` auf.

Beispiele: Muster

`Haus`	Die Zeichenfolge "Haus".
`^Haus`	"Haus" am Zeilenanfang.
`Haus$`	"Haus" am Zeilenende.
`^Haus$`	"Haus" als einziges Wort in einer Zeile.
`[Hh]aus`	"Haus" oder "haus"
`Ha[unl]s`	"Haus", "Hals" oder "Hans"
`[^HML]aus`	Weder "Haus", noch "Maus", noch "Laus", dafür aber andere Zeichenfolgen, welche "aus" enthalten.
`Ha.s`	Der dritte Buchstabe ist ein beliebiges Zeichen.
`^...$`	Jede Zeile mit genau drei Zeichen.
`^\.`	Jede Zeile, die mit einem Punkt beginnt.
`^\.[a-z][a-z]`	Jede Zeile, die mit einem Punkt und zwei Kleinbuchstaben beginnt.
`^\.[a-z]\{2\}`	Wie oben, jedoch nur in `grep` und `sed` zulässig.
`^[^.]`	Jede Zeile, die nicht mit einem Punkt beginnt.
`Fehler*`	"Fehle"(!), "Fehler", "Fehlers", etc.
`"Wort"`	Ein Wort in Anführunszeichen.
`"*Wort"*`	Ein Wort mit beliebig vielen (auch keinen) Anführungszeichen.
`[A-Z][A-Z]*`	Ein oder mehrere Großbuchstaben.
`[A-Z]+`	Wie oben, jedoch nur in egrep und awk zulässig.

`[A-Z].*`	Ein Großbuchstabe, gefolgt von keinem oder beliebig vielen Zeichen.
`[A-Z]*`	Kein, ein oder mehrere Großbuchstaben.
`[a-zA-Z]`	Ein Buchstabe.
`[^0-9a-zA-Z]`	Symbole (weder Buchstaben noch Zahlen).
`[0-9a-zA-Z]`	Jedes alphanumerische Zeichen.

Beispiele: `egrep`- oder `awk`-Muster

`[567]`	Eine der /Zahlen 5, 6 oder 7.
`fuenf|sechs|sieben`	Eines der Worte fuenf, sechs oder sieben.
`80[234]?86>`	"8086", "80286", "80386", "80486".
`F(ahr|lug)zeug`	"Fahrzeug" oder "Flugzeug"

Beispiele: `ex`- oder `vi`-Muster

`\<The`	Wörter wie "Theater" oder "Thema".
`ung\>`	Wörter wie "Teilung" oder "Endung".
`\<Wort\>`	Das Wort "Wort".

Beispiele: `sed`- oder `grep`-Muster

`0\{5,\}`	Fünf oder mehr Nullen in Folge
`[0-9]-[0-9]\{3\}-[0-9]\{5\}-[0-9X]`	*ISBN-Nummern* in der Form n-nnn-nnnnn-n, das letzte Zeichen kann auch ein X sein.

Beispiele: Suchen und Ersetzen mit `sed` und `ex`. Im Folgenden werden Leerzeichen durch _ und Tabulatoren durch TAB gekennzeichnet. Befehle für `ex` werden mit einem Doppelpunkt eingeleitet.

`s/.*/( & )/`	Wiederholt die ganze Zeile, fügt aber Klammern hinzu.
`s/.*/mv & &.old/`	Formt eine Wortliste (ein Wort pro Zeile) zu mv-Befehlen um.
`/^$/d`	Löscht Leerzeilen.
`:g/^$/d`	Wie oben, im `ex`-Editor.
`/^[_TAB]*$/d`	Löscht Leerzeilen und Zeilen, die nur aus Leerzeichen oder Tabulatoren bestehen.
`:g/^[_TAB]*$/d`	Wie oben, im `ex`-Editor.
`/ */ /g`	Wandelt ein oder mehrere Leerzeichen in ein Leerzeichen um.
`:%s/ */ /g`	Wie oben, im `ex`-Editor.
`:s/[0-9]/Element &:/`	Wandelt (in der aktuellen Zeile) eine Zahl in ein Label für ein Element um.
`:s`	Wiederholt die Substitution beim ersten Vorkommen.
`:&`	Wie oben.
`:sg`	Wie oben, aber für alle Vorkommen in einer Zeile.
`:&g`	Wie oben.
`:%&g`	Wiederholt die Substitution im ganzen Puffer.
`:.,$s/Wort/\U&/g`	Wandelt von der aktuellen bis zur letzten Zeile das Wort Wort in Großschreibung um.
`:%s/.*/\L&/`	Wandelt die gesamte Datei in Kleinschreibung um.
`:s/\<./\u&/g`	Wandelt den ersten Buchstaben jedes Wortes in der aktuellen Zeile in Großschreibung um.
`:%s/ja/nein/g`	Ersetzt das Wort ja durch nein.

`:%s/Ja/~/g`	Ersetzt global ein anderes Wort (Ja) durch nein (Wiederverwendung des vorherigen Ersatzmusters).

3.8 Programmablaufkontrolle

Bei der Shell-Programmierung verfügt man über ähnliche Konstrukte wie bei anderen Programmiersprachen, um den Ablauf des Programms zu steuern. Dazu gehören Funktionsaufrufe, Schleifen, Fallunterscheidungen und dergleichen.

3.9 Kommentare (#)

Kommentare in der Shell beginnen immer mit dem Nummern-Zeichen (#). Dabei spielt es keine Rolle, ob das Zeichen am Anfang der Zeile steht, oder hinter irgendwelchen Befehlen. Alles von diesem Zeichen bis zum Zeilenende (bis auf eine Ausnahme - siehe unter 3.10).

3.10 Auswahl der Shell (#!)

In der ersten Zeile eines Shell-Skriptes sollte definiert werden, mit welcher Shell das Skript ausgeführt werden soll. Das System öffnet dann eine Subshell und führt das restliche Skript in dieser aus.

Die Angabe erfolgt über eine Zeile in der Form `#!/bin/sh`, wobei unter `/bin/sh` die entsprechende Shell (in diesem Fall die ***Bourne-Shell***) liegt. Dieser Eintrag wirkt nur dann, wenn er in der ersten Zeile des Skripts steht.

3.11 Null-Befehl (:)

Dieser Befehl tut nichts, außer den Status `0` zurückzugeben. Er wird benutzt, um Endlosschleifen zu schreiben (siehe unter 3.18), oder um leere Blöcke in `if`- oder `case`-Konstrukten möglich zu machen.

Beispiel: Prüfen, ob jemand angemeldet ist:

checkuser.sh

```
if who | grep $1 > /dev/null # who: Liste der Benutzer
                             # grep: Suche nach Muster
then :                      # tut nichts
  else echo "Benutzer $1 ist nicht angemeldet"
fi
```

3.12 Source (.)

Der Source-Befehl wird in der Form `. skriptname` angegeben. Er bewirkt ähnliches wie ein `#include` in der Programmiersprache *C*.

Die Datei (auf die das Source ausgeführt wurde) wird eingelesen und ausgeführt, als ob ihr Inhalt an der Stelle des Befehls stehen würde. Diese Methode wird zum Beispiel während des *Bootvorgangs* in den *Init-Skripten* benutzt, um immer wieder benötigte Funktionen (Starten eines Dienstes, Statusmeldungen auf dem Bildschirm etc.) in einer zentralen Datei pflegen zu können (siehe Beispiel unter A.2).

3.13 Funktionen

Es ist in der Shell auch möglich, ähnlich wie in einer 'richtigen' Programmiersprache, Funktionen zu deklarieren und zu benutzen. Da die ***Bourne-Shell*** (`sh`) nicht über Aliase verfügt, können einfache Funktionen als Ersatz dienen. Mit dem Kommando `exit` (siehe unter 3.22) hat man die Möglichkeit, aus einer Funktion einen Wert zurückzugeben.

Beispiel: Die Funktion gibt die Anzahl der Dateien im aktuellen Verzeichnis zurück. Aufgerufen wird diese Funktion wie ein Befehl, also einfach durch die Eingabe von `count`.

countfunction.sh

```
count () {
      ls | wc -l # ls: Liste aller Dateien im Verzeichnis
      # wc: Word-Count, zählt Wörter
}
```

3.14 Bedingungen ([ ])

Da die Standard-Shell keine arithmetischen oder logischen Ausdrücke auswerten kann, muss dazu ein externes Programm benutzt werden. (`if` und Konsorten prüfen nur den Rückgabewert eines aufgerufenen Programmes -- `0` bedeutet **true**, alles andere bedeutet **false**, siehe auch 3.1.2.) Dieses Programm heißt test. Üblicherweise besteht auf allen Systemen auch noch ein Link namens `[` auf dieses Programm. Dieser Link ist absolut gleichwertig zu benutzen. Dementsprechend ist es auch zwingend erforderlich, nach der Klammer ein Leerzeichen zu schreiben. Das dient dazu, Bedingungen in `if`-Abfragen u. ä. lesbarer zu machen. Um dieses Konzept der Lesbarkeit zu unterstützen, sollte man diese öffnende Klammer auch wieder schließen (obwohl das nicht zwingend nötig ist).

Das `test`-Programm bietet sehr umfangreiche Optionen an. Dazu gehören Dateitests und Vergleiche von Zeichenfolgen oder ganzen Zahlen. Diese Bedingungen können auch durch Verknüpfungen kombiniert werden. Dateitests:

`-b Datei`	Die Datei existiert und ist ein blockorientiertes Gerät
`-c Datei`	Die Datei existiert und ist ein zeichenorientiertes Gerät
`-d Datei`	Die Datei existiert und ist ein Verzeichnis
`-f Datei`	Die Datei existiert und ist eine reguläre Datei
`-g Datei`	Die Datei existiert und das Gruppen-ID-Bit ist gesetzt
`-h Datei`	Die Datei existiert und ist ein symbolischer Link
`-k Datei`	Die Datei existiert und das Sticky-Bit ist gesetzt
`-p Datei`	Die Datei existiert und ist eine Named Pipe
`-r Datei`	Die Datei existiert und ist lesbar
`-s Datei`	Die Datei existiert und ist nicht leer
`-t [n]`	Der offene Dateideskriptor n gehört zu einem Terminal; Vorgabe für n ist 1.
`-u Datei`	Die Datei existiert und das Setuid-Bis ist gesetzt
`-w Datei`	Die Datei existiert und ist beschreibbar
`-x Datei`	Die Datei existiert und ist ausführbar

Bedingungen für Zeichenfolgen:

`-n s1`	Die Länge der Zeichenfolge s1 ist ungleich Null
`-z s1`	Die Länge der Zeichenfolge s1 ist gleich Null
`s1 = s2`	Die Zeichenfolgen s1 und s2 sind identisch
`s1 != s2`	Die Zeichenfolgen s1 und s2 sind nicht identisch
`Zeichenfolge`	Die **Zeichenfolge** ist nicht Null

Ganzzahlvergleiche:

`n1 -eq n2`	n1 ist gleich n2
`n1 -ge n2`	n1 ist größer oder gleich n2
`n1 -gt n2`	n1 ist größer als n2
`n1 -le n2`	n1 ist kleiner oder gleich n2
`n1 -lt n2`	n1 ist kleiner n2
`n1 -ne n2`	n1 ist ungleich n2

Kombinierte Formen:

`(Bedingung)`	Wahr, wenn die Bedingung zutrifft (wird für die Gruppierung verwendet). Den Klammern muss ein \ vorangestellt werden.
`! Bedingung i`	Wahr, wenn die Bedingung nicht zutrifft (`NOT`).
`Bedingung1 -a Bedingung2`	Wahr, wenn beide Bedingungen zutreffen (`AND`).
`Bedingung1 -o Bedingung2`	Wahr, wenn eine der beiden Bedingungen zutrifft (`OR`).

Beispiele:

`while test $# -gt 0`	Solange Argumente vorliegen. . .
`while [ -n "$1" ]`	Solange das erste Argument nicht leer ist. . .
`if [ $count -lt 10 ]`	Wenn $count kleiner 10. . .
`if [ -d RCS ]`	Wenn ein Verzeichnis RCS existiert. . .
`if [ "$Antwort" != "j" ]`	Wenn die Antwort nicht "j" ist. . .
`if [ ! -r "$1" -o ! -f "$1" ]`	Wenn das erste Argument keine lesbare oder reguläre Datei ist. . .

3.15 if. . .

Die `if`-Anweisung in der Shell-Programmierung macht das gleiche wie in allen anderen Programmiersprachen, sie testet eine Bedingung auf Wahrheit und macht davon den weiteren Ablauf des Programms abhängig.

Die Syntax der `if`-Anweisung lautet wie folgt:

if-beispiel.sh

```
if Bedingung1
   then Befehle1
   [ elif Bedingung2
   then Befehle2 ]
   ...
   [ else Befehle3 ]
fi
```

Wenn die Bedingung1 erfüllt ist, werden die Befehle1 ausgeführt; andernfalls, wenn die Bedingung2 erfüllt ist, werden die Befehle2 ausgeführt. Trifft keine Bedingung zu, sollen die Befehle3 ausgeführt werden.

Bedingungen werden normalerweise mit dem Befehl `test` (siehe unter 3.14) formuliert. Es kann aber auch der Rückgabewert (siehe unter 3.1.2) jedes anderen Kommandos ausgewertet werden. Für Bedingungen, die auf jeden Fall zutreffen sollen steht der `Null`-Befehl (:, siehe unter 3.11) zur Verfügung.

Beispiele: Man achte auf die Positionierung der Semikoli.

test-beispiele.sh

```
#!/bin/sh
# Füge eine 0 vor Zahlen kleiner 10 ein:
if [ $counter -lt 10 ]; then
   number=0$counter; else number=$counter; fi

# Erstelle ein Verzeichnis, wenn es noch nicht existiert:
if [ ! -e $dir ]; then
   mkdir $dir; fi # mkdir: Verzeichnis erstellen
```

3.16 case. . .

Auch die `case`-Anweisung ist vergleichbar in vielen anderen Sprachen vorhanden. Sie dient, ähnlich wie die `if`-Anweisung, zur Fallunterscheidung. Allerdings wird hier nicht nur zwischen zwei Fällen unterschieden (Entweder / Oder), sondern es sind mehrere Fälle möglich. Man kann die `case`-Anweisung auch durch eine geschachtelte `if`-Anweisung völlig umgehen, allerdings ist sie ein elegantes Mittel um den Code lesbar zu halten.

Die Syntax der `case`-Anweisung lautet wie folgt:

case-beispiel-simpel.sh

```
#!/bin/sh
case Wert in
     Muster1) Befehle1;;
     Muster2) Befehle2;;
     ...
esac
```

Wenn der Wert mit dem Muster1 übereinstimmt, wird die entsprechende Befehlsgruppe (Befehle1) ausgeführt, bei Übereinstimmung mit Muster2 werden die Kommandos der zweiten Befehlsgruppe (Befehle2) ausgeführt, usw. Der letzte Befehl in jeder Gruppe muss mit ;; gekennzeichnet werden. Das bedeutet für die Shell soviel wie **springe zum nächsten esac**, so dass die anderen Bedingungen nicht mehr überprüft werden.

In den Mustern sind die gleichen Meta-Zeichen erlaubt wie bei der Auswahl von Dateinamen. Wenn in einer Zeile mehrere Muster angegeben werden sollen, müssen sie durch ein Pipezeichen (`|`, logisches ODER) getrennt werden.

Beispiele:

case-beispiel-fortgeschritten.sh

```
#!/bin/sh
# Mit dem ersten Argument in der Befehlszeile
# wird die entsprechende Aktion festgelegt:

case $1 in # nimmt das erste Argument
     Ja|Nein) response=1;;
              -[tT]) table=TRUE;;
           *) echo "Unbekannte Option"; exit 1;;
esac

# Lies die Zeilen von der Standardeingabe, bis eine
# Zeile mit einem einzelnen Punkt eingegeben wird:

while : # Null-Befehl (immer wahr, siehe unter 3.11)

do
        echo "Zum Beenden . eingeben ==> \c"
        read line # read: Zeile von StdIn einlesen
        case "$line" in
             .) echo "Ausgefuehrt"
                break;;
             *) echo "$line" >> $message ;;
        esac
done
```

3.17 for. . .

Dieses Konstrukt ähnelt nur auf den ersten Blick seinen Pendants aus anderen Programmiersprachen. In anderen Sprachen wird die `for`-Schleife meistens dazu benutzt, eine Zählvariable über einen bestimmten Wertebereich iterieren zu lassen (`for i = 1 to 100...next`). In der Shell dagegen wird die Laufvariable nicht mit aufeinanderfolgenden Zahlen belegt, sondern mit einzelnen Werten aus einer anzugebenden Liste. (Wenn man trotzdem eine Laufvariable braucht, muss man dazu die `while`-Schleife **mißbrauchen**, siehe unter 3.18)

Die Syntax der `for`-Schleife lautet wie folgt:

for-syntax.sh

```
#!/bin/sh
for x [ in Liste ]
do
        Befehle
done
```

Die Befehle werden ausgeführt, wobei der Variablen x nacheinander die Werte aus der Liste zugewiesen werden. Wie man sieht ist die Angabe der Liste optional, wenn sie nicht angegeben wird, nimmt x der Reihe nach alle Werte aus `$@` (in dieser vordefinierten Variablen liegen die Aufrufparameter - siehe unter 3.3) an. Wenn die Ausführung eines Schleifendurchlaufs bzw. der ganzen Schleife abgebrochen werden soll, müssen die Kommandos `continue` (3.20) bzw. `break` (3.21) benutzt werden.
Beispiele:

for-beispiele.sh

```
#!/bin/sh
# Seitenweises Formatieren der Dateien, die auf der
# Befehlszeile angegeben wurden, und speichern des
# jeweiligen Ergebnisses:

for file do
    pr $file > $file.tmp # pr: Formatiert Textdateien
done

# Durchsuche Kapitel zur Erstellung einer Wortliste (wie fgrep -f):

for item in `cat program_list` # cat: Datei ausgeben
do
        echo "Pruefung der Kapitel auf"
        echo "Referenzen zum Programm $item ..."
        grep -c "$item.[co]" chap* # grep: nach Muster suchen
done

# Ermittle einen Ein-Wort-Titel aus jeder Datei und
# verwende ihn als neuen Dateinamen:

for file do
    name=`sed -n 's/NAME: //p' $file`
    # sed: Skriptsprache zur
    # Textformatierung
    mv $file $name
    # mv: Datei verschieben
    # bzw. umbenennen
done
```

3.18 while. . .

Die `while`-Schleife ist wieder ein Konstrukt, das einem aus vielen anderen Sprachen bekannt ist: Die **kopfgesteuerte** Schleife.

Die Syntax der while-Schleife lautet wie folgt:

while-syntax.sh

```
#!/bin/sh
while Bedingung
do
        Befehle
done
```

Die Befehle werden so lange ausgeführt, wie die Bedingung erfüllt ist. Dabei wird die Bedingung vor der Ausführung der Befehle überprüft. Die Bedingung wird dabei üblicherweise, genau wie bei der `if`-Anweisung, mit dem Befehl `test` (siehe unter 3.14) formuliert. Wenn die Ausführung eines Schleifendurchlaufs bzw. der ganzen Schleife abgebrochen werden soll, müssen die Kommandos `continue` (3.20) bzw. `break` (3.21) benutzt werden.
Beispiel:

while-beispiel01.sh

```
#!/bin/sh
# Zeilenweise Ausgabe aller Aufrufparameter:

while [ -n "$1"]; do
      echo $1
      shift # mit shift werden die Parameter nach
                # Links geshiftet (aus $2 wird $1)
done
```

Eine Standard-Anwendung der `while`-Schleife ist der Ersatz für die Zählschleife. In anderen Sprachen kann man mit der `for`-Schleife eine Zählvariable über einen bestimmten Wertebereich iterieren lassen (`for i = 1 to 100...next`). Da das mit der `for`-Schleife der Shell nicht geht, ersetzt man die Funktion durch geschickte Anwendung der `while`-Schleife:

while-beispiel02.sh

```
#!/bin/sh
# Ausgabe der Zahlen von 1 bis 100:

i=1
while [ $i -le 100 ]
do
        echo $i
        i=`expr $i + 1`
done
```

3.19 until. . .

Die `until`-Schleife ist das Gegenstück zur `while`-Schleife: Die ebenfalls aus vielen anderen Sprachen bekannte **fußgesteuerte** Schleife.

Die Syntax der `until`-Schleife lautet wie folgt:

until-syntax.sh

```
#!/bin/sh
until Bedingung
do
        Befehle
done
```

Die Befehle werden ausgeführt, bis die Bedingung erfüllt ist. Dabei wird die Bedingung nach der Ausführung der Befehle überprüft. Die Bedingung wird dabei üblicherweise, genau wie bei der `if`-Anweisung, mit dem Befehl `test` (siehe unter 3.14) formuliert. Wenn die Ausführung eines Schleifendurchlaufs bzw der ganzen Schleife abgebrochen werden soll, müssen die Kommandos `continue` (3.20) bzw. `break` (3.21) benutzt werden.

Beispiel: Hier wird die Bedingung nicht per `test` sondern mit dem Rückgabewert des Programms `grep` formuliert.

until-beispiel.sh

```
#!/bin/sh
# Warten, bis sich der Administrator einloggt:

until who | grep "root"; do
      # who: Liste der Benutzer
      # grep: Suchen nach Muster
      sleep 2 # sleep: warten
done
echo "Der Meister ist anwesend"
```

3.20 continue

Die Syntax der `continue`-Anweisung lautet wie folgt:

continue-syntax.sh

```
#!/bin/sh
continue [ n ]
```

Man benutzt `continue` um die restlichen Befehle in einer Schleife zu überspringen und mit dem nächsten Schleifendurchlauf anzufangen. Wenn der Parameter n angegeben wird, werden n Schleifenebenen übersprungen.

3.21 break

Die Syntax der `break`-Anweisung lautet wie folgt:

break-syntax.sh

```
break [ n ]
```

Mit `break` kann man die innerste Ebene (bzw. n Schleifenebenen) verlassen ohne den Rest der Schleife auszuführen.

3.22 exit

Die Syntax der `exit`-Anweisung lautet wie folgt:

exit-syntax.sh

```
exit [ n ]
```

Die `exit`-Anweisung wird benutzt, um ein Skript zu beenden. Wenn der Parameter n angegeben wird, wird er

von dem Skript als Exit-Code zurückgegeben.

3.23 Befehlsformen

Es gibt eine Reihe verschiedener Möglichkeiten, Kommandos auszuführen:

`Befehl &`	Ausführung von Befehl im Hintergrund
`Befehl1 ; Befehl2`	Befehlsfolge, führt mehrere Befehle in einer Zeile aus
`(Befehl1 ; Befehl2)`	Subshell, behandelt Befehl1 und Befehl2 als Befehlsfolge
`Befehl1 | Befehl2`	Pipe, verwendet die Ausgabe von Befehl1 als Eingabe für Befehl2
`Befehl1 `Befehl2``	Befehls-Substitution, verwendet die Ausgabe von Befehl2 als Argumente für Befehl1
`Befehl1 && Befehl2`	`AND`, führt zuerst Befehl1 und dann (wenn Befehl1 erfolgreich war) Befehl2 aus
`Befehl1 || Befehl2`	`OR`, entweder Befehl1 ausführen oder Befehl2 (Wenn Befehl1 nicht erfolgreich war)
`{ Befehl1 ; Befehl2 }`	Ausführung der Befehle in der momentanen Shell

Beispiele:

`nroff Datei &`	Formatiert die Datei im Hintergrund
`cd; ls`	Sequentieller Ablauf
`(date; who; pwd) > logfile`	Lenkt alle Ausgaben um
`sort Datei | lp`	Sortiert die Datei und druckt sie
`vi `grep -l ifdef *.c``	Editiert die mittels grep gefundenen Dateien
`grep XX Datei && lp Datei`	Druckt die Datei, wenn sie XX enthält
`grep XX Datei || lp Datei`	Druckt die Datei, wenn sie XX nicht enthält

3.24 Datenströme

Eines der markantesten Konzepte, das in Shell-Skripten benutzt wird, ist das der Datenströme. Die meisten der vielen Unix-Tools bieten die Möglichkeit, Eingaben aus der sogenannten **Standard-Eingabe** entgegenzunehmen und Ausgaben dementsprechend auf der **Standard-Ausgabe** zu machen. Es gibt noch einen dritten Kanal für Fehlermeldungen, so dass man eine einfache Möglichkeit hat, fehlerhafte Programmdurchläufe zu behandeln indem man die Fehlermeldungen von den restlichen Ausgaben trennt.

Es folgt eine Aufstellung der drei Standardkanäle:

Datei-Deskriptor	Name	Gebräuchliche Abkürzung	Typischer Standard
`0`	Standardeingabe	stdin	Tastatur
`1`	Standardausgabe	stdout	Terminal
`2`	Fehlerausgabe	stderr	Terminal

Die standardmäßige Eingabequelle oder das Ausgabeziel können wie folgt geändert werden:

Einfache Umlenkung:	
`Befehl > Datei`	Standardausgabe von Befehl in Datei schreiben. Die Datei wird

	überschrieben, wenn sie schon bestand.
`Befehl >> Datei`	Standardausgabe von Befehl an Datei anhängen. Die Datei wird erstellt, wenn sie noch nicht bestand.
`Befehl < Datei`	Standardeingabe für Befehl aus Datei lesen.
`Befehl1 | Befehl2`	Die Standardausgabe von Befehl1 wird an die Standardeingabe von Befehl2 übergeben. Mit diesem Mechanismus können Programme als **Filter** für den Datenstrom eingesetzt werden. Das verwendete Zeichen heißt **Pipe**.

Umlenkung mit Hilfe von Datei-Deskriptoren:	
`Befehl >&n`	Standard-Ausgabe von Befehl an den Datei-Deskriptor n übergeben.
`Befehl m>&n`	Der gleiche Vorgang, nur wird die Ausgabe, die normalerweise an den Datei-Deskriptor m geht, an den Datei-Deskriptor n übergeben.
`Befehl >&-`	Schließt die Standard-Ausgabe.
`Befehl <&n`	Standard-Eingabe für Befehl wird vom Datei-Deskriptor n übernommen.
`Befehl m<&n`	Der gleiche Vorgang, nur wird die Eingabe, die normalerweise vom Datei-Deskriptor m stammt, aus dem Datei- Deskriptor n übernommen.
`Befehl <&-`	Schließt die Standard-Eingabe.

Mehrfach-Umlenkung:	
`Befehl 2> Datei`	Fehler-Ausgabe von Befehl in Datei schreiben. Die Standard-Ausgabe bleibt unverändert (z. B. auf dem Terminal).
`Befehl > Datei 2>&1`	Fehler-Ausgabe und Standard-Ausgabe von Befehl werden in die Datei geschrieben.
`(Befehl > D1) 2>D2`	Standard-Ausgabe erfolgt in die Datei D1; Fehler-Ausgabe in die Datei D2.
`Befehl | tee Dateien`	Die Ausgaben von Befehl erfolgen an der Standard-Ausgabe (in der Regel: Terminal), zusätzlich wird sie vom Kommando `tee` in die Dateien geschrieben.

Zwischen den Datei-Deskriptoren und einem Umlenkungssymbol darf kein Leerzeichen sein; in anderen Fällen sind Leerzeichen erlaubt.
Beispiele:

`cat Datei1 > Neu`	Schreibt den Inhalt der Datei1 in die Datei Neu.
`cat Datei2 Datei3 >> Neu`	Hängt den Inhalt der Datei2 und der Datei3 an die Datei Neu an.
`mail name < Neu`	Das Programm mail liest den Inhalt der Datei Neu.
`ls -l | grep "txt" | sort`	Die Ausgabe des Befehls `ls -l` (Verzeichnisinhalt) wird an das Kommando grep weitergegeben, das darin nach **txt** sucht. Alle Zeilen, die das Muster enthalten, werden anschließend an sort übergeben und landen dann sortiert auf der Standardausgabe.

Gerade der Mechanismus mit dem Piping sollte nicht unterschätzt werden. Er dient nicht nur dazu, relativ kleine Texte zwischen Tools hin- und herzureichen. An dem folgenden Beispiel soll die Mächtigkeit dieses kleinen Zeichens gezeigt werden:

Es ist mit den passenden Tools unter Unix möglich, eine ganze Audio-CD mit zwei Befehlen an der

Kommandozeile zu duplizieren. Das erste Kommando veranlaßt, dass die TOC (Table Of Contents) der CD in die Datei `cd.toc` geschrieben wird. Das dauert nur wenige Sekunden. Die Pipe steckt im zweiten Befehl. Hier wird der eigentliche Inhalt der CD mit dem Tool `cdparanoia` ausgelesen. Da kein Dateiname angegeben schreibt `cdparanoia` die Daten auf seine Standardausgabe. Diese wird von dem Brennprogramm `cdrdao` übernommen und in Verbindung mit der TOC **on the fly** auf die CD geschrieben.

cd-kopieren.sh

```
#!/bin/sh
cdrdao read-toc --datafile - cd.toc
cdparanoia -q -R 1- - | cdrdao write --buffers 64 cd.toc
```

4 Wo sind Unterschiede zu DOS-Batchdateien?

Unter *DOS* werden Batch-Dateien oft dazu benutzt, lange Kommandos abzukürzen um die Tipparbeit zu vermindern, oder um sich das Merken von vielen Parametern zu ersparen. Diese Aufgabe überläßt man unter Unix am besten den **Shell-Aliasen**.

Shell-Skripte können viel mehr als Batch-Dateien.

Wie der Name schon sagt, sind Batch-Dateien im Wesentlichen nur ein **Stapel** von Anweisungen, die nacheinander ausgeführt werden. Mit neueren *DOS*-Versionen sind zwar auch einige einfache Mechanismen zur Verzweigung hinzugekommen, aber das entspricht bei weitem nicht den Möglichkeiten, die man an der Shell hat.

Shell-Skripte kann man dagegen eher mit einer **richtigen** Programmiersprache vergleichen. Es stehen alle Konstrukte zur Verfügung, die eine Programmiersprache auszeichnen (Funktionen, Schleifen, Fallunterscheidungen, Variablen, etc).

5 Anhang A: Beispiele

5.1 Schleifen und Rückgabewerte

Man kann mit einer `until`- bzw. mit einer `while`-Schleife schnell kleine aber sehr nützliche Tools schreiben, die einem lästige Aufgaben abnehmen.

5.1.1 Schleife, bis ein Kommando erfolgreich war

Angenommen, bei der Benutzung eines Rechners tritt ein Problem auf, bei dem nur der Administrator helfen kann. Dann möchte man informiert werden, sobald dieser an seinem Arbeitsplatz ist. Man kann jetzt in regelmäßigen Abständen das Kommando `who` ausführen, und dann in der Ausgabe nach dem Eintrag **root** suchen. Das ist aber lästig.

Einfacher geht es, wenn wir uns ein kurzes Skript schreiben, das alle 30 Sekunden automatisch überprüft, ob der Admin angemeldet ist. Wir errei- chen das mit dem folgenden Code:

auf-root-warten.sh

```
#!/bin/sh
until who | grep "^root "
      do sleep 30
done
echo Big Brother is watching you!
```

Das Skript führt also so lange das Kommando aus, bis die Ausführung erfolgreich war. Dabei wird die Ausgabe von `who` mit einer Pipe (3.24) in das `grep`-Kommando umgeleitet. Dieses sucht darin nach einem Auftreten von **root** am Zeilenanfang. Der Rückgabewert von `grep` ist `0` wenn das Muster gefunden wird, `1` wenn es nicht gefunden wird und `2` wenn ein Fehler auftrat. Damit der Rechner nicht die ganze Zeit mit dieser Schleife beschäftigt ist, wird im Schleifenkörper ein sleep 30 ausgeführt, um den Prozeß für 30 Sekunden schlafen zu schicken. Sobald der Admin sich eingeloggt hat, wird eine entsprechende Meldung ausgegeben.

5.1.2 Schleife, bis ein Kommando nicht erfolgreich war

Analog zum vorhergehenden Beispiel kann man auch ein Skript schreiben, das meldet, sobald sich ein Benutzer abgemeldet hat. Dazu ersetzen wir nur die `until`-Schleife durch eine entsprechende `while`-Schleife:

warten-bis-root-verschwindet.sh

```
#!/bin/sh
while who | grep "^root "
      do sleep 30
done
echo Die Katze ist aus dem Haus, Zeit, dass die Mäuse tanzen!
```

Die Schleife wird nämlich dann so lange ausgeführt, bis `grep` einen Fehler (bzw. eine erfolglose Suche) zurückmeldet.

5.2 Ein typisches Init-Skript

Dieses Skript dient dazu, den Apache HTTP-Server zu starten. Es wird während des Bootvorgangs gestartet, wenn der dazugehörige Runlevel initialisiert wird.

Das Skript muss mit einem Parameter aufgerufen werden. Möglich sind hier `start`, `stop`, `status`, `restart` und `reload`. Wenn falsche Parameter übergeben wurden, wird eine entsprechende Meldung angezeigt.

Das Ergebnis der Ausführung wird mit Funktionen dargestellt, die aus der Datei `/etc/rc.d/init.d/functions` stammen. Ebenfalls in dieser Datei sind Funktionen, die einen Dienst `starten` oder `stoppen`.

Zunächst wird festgelegt, dass dieses Skript in der ***Bourne-Shell*** ausgeführt werden soll (3.24).

beispiel.sh

```
#!/bin/sh
```

Dann folgen Kommentare, die den Sinn des Skriptes erläutern (3a.9).

beispiel.sh (Fortsetzung)

```
## Startup script for the Apache Web Server
#
# chkconfig: 345 85 15
# description: Apache is a World Wide Web server. It is \
#              used to serve HTML files and CGI
#
# processname: httpd
# pidfile: /var/run/httpd.pid
# config: /etc/httpd/conf/access.conf
# config: /etc/httpd/conf/httpd.conf
# config: /etc/httpd/conf/srm.conf
```

Jetzt wird die Datei mit den Funktionen eingebunden (3.13).

beispiel.sh (Fortsetzung)

```
# Source function library.
/etc/rc.d/init.d/functions
```

Hier werden die Aufrufparameter ausgewertet (3.17).

beispiel.sh (Fortsetzung)

```
# See how we were called.
case "$1" in
     start)
        echo -n "Starting httpd: "
```

Nachdem eine Meldung über den auszuführenden Vorgang ausgegeben wurde, wird die Funktion `daemon` aus der Funktionsbibliothek ausgeführt. Diese Funktion startet das Programm, dessen Name hier als Parameter übergeben wird. Dann gibt sie eine Meldung über den Erfolg aus.

beispiel.sh (Fortsetzung)

```
        daemon httpd
        echo
```

Jetzt wird ein Lock-File angelegt. (Ein Lock-File signalisiert anderen Prozessen, dass ein bestimmter Prozeß bereits gestartet ist. So kann ein zweiter Aufruf verhindert werden.)

beispiel.sh (Fortsetzung)

```
        touch /var/lock/subsys/httpd
        ;;
      stop)
        echo -n "Shutting down http: "
```

Hier passiert im Prinzip das gleiche wie oben, nur dass mit der Funktion `killproc` der Daemon angehalten wird.

beispiel.sh (Fortsetzung)

```
        killproc httpd
        echo
```

Danach werden Lock-File und PID-File gelöscht. (In einem sogenannten PID-File hinterlegen einige Prozesse ihre Prozeß-ID, um anderen Programmen den Zugriff zu erleichtern, z.B. um den Prozeß anzuhalten etc.)

beispiel.sh (Fortsetzung)

```
        rm -f /var/lock/subsys/httpd
        rm -f /var/run/httpd.pid
        ;;
    status)
```

Die Funktion `status` stellt fest, ob der entsprechende Daemon bereits läuft, und gibt das Ergebnis aus.

beispiel.sh (Fortsetzung)

```
        status httpd
        ;;
  restart)
```

Bei Aufruf mit dem Parameter `restart` ruft sich das Skript zwei mal selbst auf (in `$0` steht der Aufrufname des laufenden Programms). Einmal, um den Daemon zu `stoppen`, dann, um ihn wieder zu `starten`.

beispiel.sh (Fortsetzung)

```
        $0 stop
        $0 start
        ;;
   reload)
        echo -n "Reloading httpd: "
```

Hier sendet die `killproc`-Funktion dem Daemon ein Signal das ihm sagt, dass er seine Konfiguration neu einlesen soll.

beispiel.sh (Fortsetzung)

```
        killproc httpd -HUP
        echo
        ;;
        *)
        echo "Usage: $0 {start|stop|restart|reload|status}"
```

Bei Aufruf mit einem beliebigen anderen Parameter wird eine Kurzhilfe ausgegeben. Dann wird dafür gesorgt, dass das Skript mit dem Exit-Code 1 beendet wird. So kann festgestellt werden, ob das Skript ordnungsgemäß beendet wurde (3.22).

beispiel.sh (Fortsetzung)

```
        exit 1
esac
exit 0
```

5.3 Parameterübergabe in der Praxis

Es kommt in der Praxis sehr oft vor, dass man ein Skript schreibt, dem der Anwender Parameter übergeben soll. Wenn das nur eine Kleinigkeit ist (zum Beispiel ein Dateiname), dann fragt man einfach die entsprechenden vordefinierten Variablen (3.3) ab. Sollen aber **richtige** Parameter eingesetzt werden, die sich so einsetzen lassen wie man es von vielen Kommandozeilentools gewohnt ist, dann benutzt man das Hilfsprogramm `getopt`. Dieses Programm parst die originalen Parameter und gibt sie in **standardisierter** Form zurück.

Das soll an folgendem Skript verdeutlicht werden. Das Skript kennt die Optionen `-a` und `-b`. Letzterer Option

muss ein zusätzlicher Wert mitgegeben werden. Alle anderen Parameter werden als Dateinamen interpretiert.

getopt.sh

```
#!/bin/sh
set -- `getopt "ab:" "$@"` || {
```

Das `set`-Kommando belegt den Inhalt der vordefinierten Variablen (3.3) neu, so dass es aussieht, als ob dem Skript die Rückgabewerte von `getopt` übergeben wurden. Man muss die beiden Minuszeichen angeben, da sie dafür sorgen, dass die Aufrufparameter an getopt und nicht an die Shell selbst übergeben werden. Die originalen Parameter werden von `getopt` untersucht und modifiziert zurückgegeben: a und b werden als Parameter Markiert, b sogar mit der Möglichkeit einer zusätzlichen Angabe.

Wenn dieses Kommando fehlschlägt ist das ein Zeichen dafür, dass falsche Parameter übergeben wurden. Also wird nach einer entsprechenden Meldung das Programm mit Exit-Code 1 verlassen.

getopt.sh (Fortsetzung)

```
        echo "Anwendung: `basename $0` [-a] [-b Name] Dateien" 1>&2
        exit 1
}
echo "Momentan steht in der Kommandozeile folgendes: $*"
aflag=0 name=NONE
while :
do
```

In einer Endlos-Schleife, die man mit Hilfe des Null-Befehls (:, 3.11) baut, werden die **neuen** Parameter der Reihe nach untersucht. Wenn ein `-a` vorkommt, wird die Variable `aflag` gesetzt. Bei einem `-b` werden per shift alle Parameter nach Links verschoben, dann wird der Inhalt des nächsten Parameters in der Variablen name gesichert.

getopt.sh (Fortsetzung)

```
        case "$1" in
             -a) aflag=1 ;;
             -b) shift; name="$1" ;;
             --) break ;;
```

Wenn ein `--` erscheint, ist das ein Hinweis darauf, dass die Liste der Parameter abgearbeitet ist. Dann wird per `break` (3.21) die Endlosschleife unterbrochen. Die Aufrufparameter enthalten jetzt nur noch die eventuell angegebenen Dateinamen, die von dem restlichen Skript wie gewohnt weiterverarbeitet werden können.

getopt.sh (Fortsetzung)

```
        esac
        shift
done
shift
```

Am Ende werden die Feststellungen ausgegeben.

getopt.sh (Fortsetzung)

```
echo "aflag=$aflag / Name = $name / Die Dateien sind $*"
```

5.4 Fallensteller: Auf Traps reagieren

Ein laufendes Shell-Skript kann durch Druck auf die Interrupt-Taste (normalerweise [CTRL + C]) unterbrochen werden. Durch Druck auf diese Taste wird ein Signal an den entsprechenden Prozeß gesandt, das ihn bittet sich zu beenden. Dieses Signal heißt *SIGINT* (für SIGnal INTerrupt) und trägt die Nummer 2. Das kann ein kleines Problem darstellen, wenn das Skript sich temporäre Dateien angelegt hat, da diese nach der Ausführung nur noch unnötig Platz verbrauchen und eigentlich gelöscht werden sollten. Man kann sich sicher auch noch wichtigere Fälle vorstellen, in denen ein Skript be- stimmte Aufgaben auf jeden Fall erledigen muss, bevor es sich beendet.

Es gibt eine Reihe weiterer Signale, auf die ein Skript reagieren kann. Alle sind in der Man-Page von signal beschrieben. Hier die wichtigsten:

Nummer	Name	Bedeutung
0	Normal Exit	Wird durch das exit-Kommando ausgelöst.
1	SIGHUP	Wenn die Verbindung abbricht (z.B. wenn das Terminal geschlossen wird).
2	SIGINT	Zeigt einen Interrupt an ([CTRL + C]).
15	SIGTERM	Wird vom kill-Kommando gesendet.

Wie löst man jetzt dieses Problem? Glücklicherweise verfügt die Shell über das trap-Kommando, mit dessen Hilfe man auf diese Signale reagieren kann. Die Anwendung soll in folgendem Skript beispielhaft dargestellt werden.

Das Skript soll eine komprimierte Textdatei mittels zcat in ein temporäres File entpacken, dieses mit pg seitenweise anzeigen und nachher wieder löschen.

zeige-komprimierte-datei.sh

```
#!/bin/sh
stat=1
temp=/tmp/zeige$$
```

Zunächst werden zwei Variablen belegt, die im weiteren Verlauf benutzt werden sollen. In stat wird der Wert abgelegt, den das Skript Falle eines Abbruchs als Exit-Status zurückliefern soll. Die Variable temp enthält den Namen für eine temporäre Datei. Dieser setzt sich zusammen aus /tmp/zeige und der Prozeßnummer des laufenden Skripts. So soll sichergestellt werden, dass noch keine Datei mit diesem Namen existiert.

zeige-komprimierte-datei.sh (Fortsetzung)

```
trap 'rm -f $temp; exit $stat' 0
trap 'echo "`basename $0`: Ooops..." 1>&2' 1 2 15
```

Hier werden die `Traps` definiert. Bei Signal `0` wird die temporäre Datei gelöscht und der Wert aus der Variable stat als Exit-Code zurückgegeben. Dabei wird dem `rm`-Kommando der Parameter `-f` mitgegeben, damit keine Fehlermeldung ausgegeben wird, falls die Datei (noch) nicht existiert. Dieser Fall tritt bei jedem Beenden des Skriptes auf, also sowohl bei einem normalen Ende, als auch beim Exit-Kommando, bei einem Interrupt oder bei einem `Kill`. Der zweite `Trap` reagiert auf die Signale `1`, `2` und `15`. Das heißt, er wird bei jedem unnormalen Ende ausgeführt. Er gibt eine entsprechende Meldung auf die Standard-Fehler-Ausgabe (3.10) aus. Danach wird das Skript beendet, und der erste `Trap` wird ausgeführt.

zeige-komprimierte-datei.sh (Fortsetzung)

```
case $# in
    1) zcat "$1" > $temp
       pg $temp
       stat=0
       ;;
```

Jetzt kommt die eigentliche Funktionalität des Skriptes: Das case-Kommando (3.16) testet die Anzahl der übergebenen Parameter. Wenn genau ein Parameter übergeben wurde, entpackt `zcat` die Datei, die im ersten Parameter angegeben wurde, in die temporäre Datei. Dann folgt die seitenweise Ausgabe mittels `pg`. Nach Beendigung der Ausgabe wird der Status in der Variablen auf `0` gesetzt, damit beim Skriptende der korrekte Exit-Code zurückgegeben wird.

zeige-komprimierte-datei.sh (Fortsetzung)

```
    *) echo "Anwendung: `basename $0` Dateiname" 1gt;&2
esac
```

Wenn `case` eine andere Parameterzahl feststellt, wird eine Meldung mit der Aufrufsyntax auf die Standard-Fehlerausgabe geschrieben.

6 Anhang B

6.1 Quellen

* Bash Reference Manual (http://www.gnu.org/manual/bash-2.02/bashref.html)
* Unix In A Nutshell (http://www.oreilly.com/catalog/unixnut3/)
* Unix Power Tools (http://www.oreilly.com/catalog/upt2/)
* Von DOS nach Linux HOWTO (http://www.tu-harburg.de/semb2204/dlhp/HOWTO/DE-DOS-nach-Linux-HOWTO.html)