Study-Guide: Shells, Scripting, Programmierung und Compilieren

Die folgenden zwei Kapitel behandeln die grundlegende Fähigkeit der Shell, programmiert zu werden. Die wirklich tiefergehenden Fähigkeiten zu diesem Thema werden erst in Level 2 (lpi201 und lpi202) besprochen und notwendig. Anpassung und Verwendung der Shell-Umgebung Anpassen und Schreiben einfacher Scripts

Seite:	-= LinuxLernSystem =- (http://www.lpi-test.de)
Kurs:	LPIC-1 [102]
Buch:	Study-Guide: Shells, Scripting, Programmierung und Compilieren
Gedruckt von:	André Scholz
Datum:	Dienstag, 1 November 2005, 10:42 Uhr

Inhaltsverzeichnis

1.109 - Shells, Scripting, Programmierung und Compilieren

Die folgenden zwei Kapitel behandeln die grundlegende Fähigkeit der Shell, programmiert zu werden. Die wirklich tiefergehenden Fähigkeiten zu diesem Thema werden erst in Level 2 (lpi201 und lpi202) besprochen und notwendig.

- Anpassung und Verwendung der Shell-Umgebung
- Anpassen und Schreiben einfacher Scripts

1.109.1 - Anpassung und Verwendung der Shell-Umgebung

Beschreibung: Prüfungskandidaten sollten in der Lage sein, Shell-Umgebungen auf die Bedürfnisse der Benutzer hin anzupassen. Dieses Lernziel beinhaltet das Setzen von Umgebungsvariablen (z.B. PATH) beim Login oder beim Aufruf einer neuen Shell. Ebenfalls enthalten ist das Schreiben von Bash-Funktionen für oft benutzte Kommandoabfolgen.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- `~/.bash_profile`
- `~/.bash_login`
- `~/.profile`
- `~/.bashrc`
- `~/.bash_logout`
- `~/.inputrc`
- **function** (eingebautes Bash-Kommando)
- **export**
- **env**
- **set** (eingebautes Bash-Kommando)
- **unset** (eingebautes Bash-Kommando)

Login-Shell versus NoLogin-Shell

Unix unterscheidet beim Aufruf einer Shell, ob es sich um eine Login-Shell handelt, oder um eine No-Login-Shell. Der Unterschied ist einfach: Die Login-Shell ist die Shell, die beim Login vom login-Programm gestartet wurde. Jede weitere Shell, die von der Login-Shell aufgerufen wird ist eine No-Login-Shell.

Der Grund für diese Unterscheidung liegt in der Frage der Konfigurationsdateien, die später noch genauer dargestellt werden. Zunächst nur so viel: Die Login-Shell muß konfiguriert werden, sie erhält beim Start viele Variablen, die sie an spätere Shells weitervererbt.

Eine Shell, die wiederum von der Loginshell aufgerufen wird hat diese Konfiguration nicht nötig. Durch die Tatsache, daß Variablen an aufgerufene Shells weitergegeben werden können, erledigt sich das von selbst. Außerdem wäre es eine sehr zeitaufwendige Methode, wenn jede Subshell jedesmal alle Konfigurationsdateien abarbeiten müsste.

Die Frage, die sich grundsätzlich stellt ist ja auch die, wozu eine Shell denn überhaupt eine Subshell aufruft. Das kommt sehr viel häufiger vor, als auf den ersten Blick ersichtlich. Jedesmal, wenn die Shell ein Script abarbeitet, wird dazu von ihr eine Subshell aufgerufen. Wenn bei jedem Scriptaufruf dann alle Konfigurationsdateien abgearbeitet werden würden, wäre das eine echte Zeitfrage.

Auch in der graphischen Benutzeroberflache, die ja oft aus einer Shell heraus gestartet wird (startx) erscheinen wieder mehrere xterm-Fenster, in denen jeweils eine Shell läuft. All diese Shells sind keine Login-Shells.

Eine weitere Unterteilung der No-Login-Shells ist die Frage, ob es sich om eine interaktive Shell handelt, oder nicht. Diese Frage ist aber unerheblich, wenn es um die Konfiguration geht.

Die Geltungsbereiche von Shell-Variablen

Die Shellvariablen und Befehle wie **set, unset** und env wurden schon im Abschnitt 1.103.1 der Vorbereitung auf die LPI101 Prüfung besprochen. Hier geht es jetzt um die Frage des Geltungsbereichs von Variablen.

Wenn innerhalb einer Shell eine Variable definiert wird, und danach eine zweite Shell aufgerufen wird, so ist die Variable innerhalb der zweiten Shell nicht automatisch gültig. Das läßt sich einfach ausprobieren, indem die

folgenden Befehle eingegeben werden:

```
$ Testvariable=Hallo
$ echo $Testvariable
Hallo
$ bash
$ echo $Testvariable

$ exit
$ echo $Testvariable
Hallo
```

Wir haben also zunächst eine Variable `Testvariable` mit dem Wert `Hallo` angelegt. Im zweiten Schritt geben wir den Inhalt der Variable aus, das funktioniert erwartungsgemäß. Jetzt starten wir eine zweite Shell mit dem Aufruf von **bash**. Der erneute Versuch, sich den Inhalt der Variablen anzusehen schlägt fehl. Erst wenn wir die zweite Shell mit `exit` wieder beenden, funktioniert die Variablenausgabe wieder.

Damit eine Shell eine Variable an ihre Subshells weitergibt, muß die Variable exportiert werden. Das geschieht mit dem Shellbefehl **export**. Es gibt zwei Formen der Anwendungen, entweder wird eine Variable zuerst definiert und dann exportiert, oder die Variablendefinition geschieht gleich zusammen mit `export`:

```
Variable1=Versuch
export Variable1
export Variable2="Noch ein Versuch"
```

Eine exportierte Variable wird grundsätzlich an alle folgenden Subshells weitergegeben und ist dort verfügbar. Selbst wenn die Subshell wiederum eine Shell aufruft, ist die Variable auch innerhalb dieser dritten Shell gültig.

Wird allerdings die Variable dann in einer Subshell verändert, so bezieht sich diese Veränderung nur auf die Subshell. Innerhalb der aufrufenden Shell behält die Variable ihren alten Wert. Das beweist, daß es sich eben nicht um die selbe Variable handelt, sondern daß eine exportierte Variable in einer Subshell nochmal extra angelegt wird.

```
$ export name=hans
$ echo $name
hans
$ bash
$ echo $name
hans
$ name=otto
$ echo $name
otto
$ exit
$ echo $name
hans
```

Wie schon bei der Darstellung des Unterschiedes zwischen LoginShell und NoLoginShell gezeigt, wird jedesmal eine Subshell aufgerufen, wenn ein Shellscript abgearbeitet werden soll. Wenn in diesem Script Variablen definiert werden, dann gelten diese Variablen natürlich nur innerhalb der Subshell. Die aufrufende Shell bekommt von diesen Variablen überhaupt nichts mit.

Das ist aber manchmal sehr unpraktisch. Wenn wir etwa ein Script schreiben, um verschiedene benötigte Shellvariablen anzulegen, dann hilft uns das herzlich wenig, denn jedesmal wenn wir das Script aufrufen werden diese Variablen zwar in der Subshell, die das Script ausführt erzeugt - aber jedesmal, wenn das Script beendet wurde (und damit auch die Subshell) sind die Variablen nicht mehr gültig.

Damit es trotzdem möglich ist, ein Script zu verwenden um Variablen zu definieren, gibt es die Möglichkeit die Shell dazu zu zwingen, keine Subshell zur Abarbeitung des Scripts aufzurufen. Das geschieht einfach durch das Voranstellen eines Punktes vor dem Scriptaufruf. Selbstverständlich muß zwischen dem Punkt und dem

Scriptnamen dabei ein Leerzeichen stehen.

Dadurch wird die Shell gezwungen, das Script selbst abzuarbeiten. Die Variablen, die innerhalb dieses Scripts definiert wurden, sind jetzt auch nach der Abarbeitung des Scriptes noch gültig.

Das klingt schön, hat aber einen bedeutenden Haken. Wenn innerhalb des Scrips ein exit vorkommt, normalerweise dazu benutzt, um das Script zu beenden, wird ja die Shell, die das Script ausführt, beendet. Wurde das Script jetzt mit vorgestelltem Punkt aufgerufen, also von unserer Loginshell selbst, dann wird das exit die LoginShell beenden! Wir müßten uns also erneut einloggen.

Scripts, die mit führendem Punkt aufgerufen werden sollen, sollten daher auf keinen Fall - auch nicht zur Fehlerbehandlung - ein exit benutzen.

Alias und Funktion in der interaktiven Shell

Neben den Variablen gibt es noch zwei weitere Formen, die innerhalb einer Shell definiert werden können. Das Alias und die Funktion.

Alias

Ein Alias ist sozusagen ein Mechanismus, der der Shell klar macht, daß ein bestimmter Name eine bestimmte Bedeutung hat. Jedesmal, wenn die Shell auf einen Befehl trifft, überprüft sie zuerst, ob es sich dabei um einen Alias handelt, erst wenn es klar ist, daß es kein Alias ist, wird der Unix-Befehl gesucht. Das heißt, daß damit die Bedeutung von Unix-Befehlen überlagert werden kann. Ein Alias wird nur ein einziges Mal aufgelöst, so daß es möglich ist, bestehende Unix-Befehle umzudefinieren.

Beispiele:

```
alias cp=cp -i
```

Damit wird der Befehl cp grundsätzlich als cp -i ausgeführt, das heißt, er frägt vor dem Überschreiben einer Zieldatei nach, ob das in Ordnung ist.

```
alias ...=cd ..;cd..
```

Der Befehl `...` ist ein Alias auf den Befehl `cd ..;cd ..`, also zweimal ins nächsthöhere Verzeichnis zu wechseln.

```
alias werbinich="echo $LOGNAME \($UID\)"
```

Der Befehl `werbinich` gibt in einer Zeile den Usernamen und die UserID des Users aus, der ihn ausführt.

Um Aliase wieder loszuwerden gibt es den Befehl **unalias** mit dem ein Alias wieder gelöscht werden kann.

Zur Gültigkeit von Aliasen gilt genau das selbe, wie zur Gültigkeit von Shellvariablen. Sie haben nur Gültigkeit in der Shell, in der sie definiert wurden. Im Gegensatz zu Variablen lassen sich Aliase aber nicht exportieren - zumindestens nicht bei modernen Shells. Wenn ein Alias auch innerhalb einer Subshell gelten soll, so muß er in einer Konfigurationsdatei definiert werden, die auch von einer No-Login-Shell abgearbeitet wird, also etwa `~/.bashrc`.

Funktionen

Der Alias-Mechanismus erlaubt eine vielseitige Anwendung auch kombinierter Befehle, die unter einem neuen Namen zusammengefasst werden. Er hat aber zwei wesentliche Einschränkungen. Innerhalb des Alias ist es nicht möglich, auf einzelne Parameter des Aliases zuzugreifen, die ihm mitgegeben wurden. Es ist ja tatsächlich ein Textersatz, der hier vorgenommen wird.

Der zweite Nachteil des Aliases ist, daß seine Interpretation nur einmal stattfindet, dadurch kann er nicht rekursiv

aufgerufen werden.

Beide Nachteile (die in Wahrheit keine Nachteile sind, sondern nur Unterschiede zur Funktion) werden von den Shellfunktionen abgeschafft. Es handelt sich hier um einen echten Funktionsaufruf, der die mitgegebenen Parameter bearbeiten kann und der rekursiv laufen kann, das heißt, die Funktion kann sich selber aufrufen.

Shellfunktionen sind kleine Unterprogramme, die einen eigenen Namen haben. Sie können sowohl im Script, als auch in der interaktiven Shell verwendet werden.

Die prinzipielle Aufgabe von Funktionen in Programmiersprachen ist, immer wiederkehrende Aufgaben oder logisch zusammenhängende Teile eines Programms zu separieren und als extra Funktion zu lösen. Dabei können einer Funktion, genauso wie dem Script selbst, Parameter übergeben werden; dazu kann eine Funktion einen eigenen (numerischen) Rückgabewert liefern.

Der prinzipielle Aufbau einer Shellfunktion ist:

```
function Funktionsname()
{
  Kommando1
  ...
}
```

Das Schlüsselwort `function` kann auch weggelassen werden, weil die Shell eine Funktion an den Klammern () am Ende des Funktionsnamens erkennt.

Parameter, die einer Funktion mitgegeben werden, werden nicht in den Klammern im Funktionskopf vordefiniert, wie in anderen Programmiersprachen. Innerhalb der Funktion gelten für die übergebenen Parameter die gleichen Regeln, wie beim Script selbst. `$1` bezeichnet den ersten übergebenen Parameter, `$2` den zweiten usw. Auch `$*, $@` und `$#` beziehen sich auf die Parameter der Funktion und nicht mehr auf die des Scripts. Allein der Parameter `$0` behält den Namen des Scripts und nicht den der Funktion.

Genauer gesagt gelten all die genannten Variablen (`$1 - $9, $#, $*, $@`) innerhalb einer Funktion als lokale Parameter.

Funktionen können direkt eingegeben werden, das ist aber eher selten. Meist werden sie in den Konfigurationsdateien erstellt und exportiert. Sie müssen wie Variablen und Aliase exportiert werden, damit sie auch in späteren Shells gültig sind.

Wollen wir z.B. eine Funktion schreiben, die beim Verzeichniswechsel gleich den Inhalt des Verzeichnisses anzeigt, in das gewechselt wurde, so können wir das folgendermaßen erledigen:

```
function lscd()
{
  cd $*
  ls
}
```

In einem Alias wäre das nicht möglich gewesen, weil wir ja den Aufrufparameter ($*) nicht gehabt hätten. Was aber, wenn wir diese Funktion nicht `lscd` sondern einfach nur `cd` genannt hätten?

Das hätte ziemlich schnell dazu geführt, daß der Computer keinen Arbeitsspeicher mehr übrig gehabt hätte. Die Funktion hätte sich nämlich permanent selbst aufgerufen. Ohne eine vernünftige Abbruchbedingung hätte das dazu geführt, daß es zu einer Endlosschleife kommt, die in jedem Durchlauf Speicher anfordert. Es wäre also ein sogenannter rekursiver Aufruf gewesen.

Damit zumindestens interne Befehle der Shell wie `cd` überlagert werden können, bietet die Shell das reservierte Wort `builtin` an, das anzeigt, daß in jedem Fall der interne Befehl und nicht die Funktion aufgerufen werden soll. Um unser Beispiel also arbeitsfähig zu machen müßten wir schreiben:

```
function cd()
{
  builtin cd $*
  ls
}
```

Die Konfigurationsdateien der bash

Bei den Konfigurationsdateien für die Shell handelt es sich um einfache Shellscripts, deren Ausführungsrecht nicht gesetzt sein muß. Die Shell führt diese Scripts selbst aus, ohne eine Subshell aufzurufen. Die Variablen, Aliase und Funktionen, die in diesen Scripts definiert wurden, gelten also innerhalb der aufrufenden Shell. In der Regel werden diese Variablen, Aliase und Funktionen exportiert, damit sie auch in kommenden Subshells gültig sind.

Natürlich können auch andere Programme aus den Konfigurationsscripts heraus gestartet werden, typisch ist z.B. der Aufruf von umask, um festzulegen, mit welchen Zugriffsrechten Dateien versehen werden sollen, die neu angelegt werden. Auch Tastaturdefinitionen o.ä. können hier eingestellt werden.

Grundsätzlich unterscheidet die Shell bei Konfigurationsdateien, zwischen Login-Shells und Nicht-Login-Shells. Nur bei Login-Shells werden alle Konfigurationsdateien abgearbeitet. Bei allen anderen Shells wird nur optional eine Datei abgearbeitet, die nicht von der Login-Shell benutzt wird. Konkret werden die folgenden Dateien in der angegebenen Reihenfolge und Abhängigkeit von der Login-Shell abgearbeitet:

Wenn die Datei `/etc/profile`existiert, wird sie abgearbeitet

Wenn im Heimatverzeichnis des Users die Datei `.bash_profile` existiert wird sie abgearbeitet.

Wenn diese Datei nicht existiert wird die Datei `.bash_login`im Heimatverzeichnis des Users gesucht und falls gefunden, abgearbeitet.

Wenn auch diese Datei nicht existiert, so wird im Heimatverzeichnis des Users die Datei `.profile`gesucht und falls gefunden abgearbeitet.

Diese Konstruktion ermöglicht es, sowohl die systemweiten Einstellungen vorzunehmen, als auch jedem User die Freiheit zu lassen, selbst Einstellungen vorzunehmen. Die `/etc/profile`-Datei kann nur vom Systemverwalter verändert werden, alle anderen Dateien befinden sich ja in den Heimatverzeichnissen der einzelnen User. Sie sind meist frei editierbar.

Eine einzige Datei wird von der Nicht-Login-Shell abgearbeitet, falls sie existiert - `.bashrc` im Heimatverzeichnis jedes Users.

Es gibt auch noch eine Datei, die beim Logout - also dem Beenden der LoginShell - abgearbeitet wird, sofern sie existiert. Sie heißt `.bash_logout` und befindet sich auch im Heimatverzeichnis der einzelnen User. Damit ist es z. B. möglich, daß temporäre Dateien gelöscht werden oder ähnliche Aufräumarbeiten erledigt werden.

Die Bourne Again Shell benutzt eine Library, die für den Umgang mit Eingabezeilen gemacht ist. Diese Library heiß Readline und ist ihrerseits konfigurierbar. Ihre Einstellungen nimmt sie in der Datei `.inputrc` im Homeverzeichnis jedes Users vor. In dieser Datei können Tastenbindungen vorgenommen werden, um die Standard-Tastenbelegung für die Eingabezeile zu verändern.

1.109.2 - Anpassen und Schreiben einfacher Scripts

Beschreibung: Prüfungskandidaten sollten in der Lage sein, existierende Scripts anzupassen und einfache neue (ba)sh Scripts zu schreiben. Dieses Lernziel beinhaltet die Verwendung der Standard sh Syntax (Schleifen, Tests), das Verwenden von Kommandosubstitution, das Prüfen von Kommando-Rückgabewerten, Testen des Status einer Datei und Schicken von Mails an den Superuser unter bestimmten Bedingungen. Ebenfalls enthalten ist die Vorsorge, daß der korrekte Interpreter auf der ersten Zeile (#!) von Scripts aufgerufen wird. Weiters enthalten ist die Verwaltung von Speicherort, Eigentum und Ausführungs- und suid-Rechten von Scripts.

Die wichtigsten Dateien, Bezeichnungen und Anwendungen:

- **while**
- **for**
- **test**
- **chmod**

Das Programmieren der Shell

Der Shell ist es prinzipiell egal, ob sie ihre Anweisungen interaktiv über die Tastatur bekommt, oder ob sie diese Angaben aus einer Datei herauslesen soll. Es ist also möglich, mehrere Befehlszeilen in eine Datei zu schreiben, die dann von der Shell ausgeführt werden, als ob sie nacheinander eingegeben wurden.

Dabei gehen die Möglichkeiten der Shell aber weit über die Batch-Programmierung von DOS hinaus. Sie bietet echte Schleifen, vernünftige Ein/Ausgabe, flexible Variablen, Bedingungsüberprüfung und vieles mehr. Es ist sogar möglich, Funktionen zu nutzen und so genauso strukturiert zu arbeiten, wie es in modernen Hochsprachen üblich ist.

Shellscripts werden in der Systemverwaltung sehr gerne und häufig eingesetzt, weil sie sehr schnell und effizient Lösungen für häufig vorkommende Probleme ermöglichen. Das Programmieren der Shell ist aber ebn eine richtige kleine Programmiersprache, es würde den Rahmen des Kurses sicher sprengen, wenn das hier ein kompletter Programmierkurs wäre. Im Folgenden werden alle wichtigen Elemente der Shellscripts beschrieben, die sichere Anwendung kommt aber sicher nur mit viel Praxis...

Das grundlegende Prinzip der Shellscripts

Prinzipiell ist ein Shellscript nichts anderes als eine Textdatei, die einzelne Befehlszeilen enthält, die von der Shell nacheinander abgearbeitet werden. Jede Zeile, die nicht mit einem Doppelkreuz (#) beginnt, wird von der Shell als Befehlszeile interpretiert und ausgeführt. Zeilen, die mit dem Doppelkreuz beginnen werden als Kommentarzeilen bewertet und nicht ausgeführt. Die einfachste Form wäre also, eine Textdatei zu erstellen, die ein paar Unix-Befehle enthält, diese Datei unter einem bestimmten Namen zu speichern z.B. script1 und dann eine Shell aufzurufen und ihr diesen Namen als Parameter mitzugeben also etwa

```
bash script1
```

Das ist aber natürlich nicht die eleganteste Lösung, schöner wäre es ja, das Script direkt wie ein Programm aufrufen zu können. Dazu müssen wir dem Script einfach nur das Ausführungsrecht geben.

```
chmod +x script1
```

Wenn wir uns jetzt das Script mit **ls -l** ansehen, so hat es jetzt neben den bisherigen Zugriffsrechten auch das x-Recht. Es ist jetzt durch die Nennung seines Namens aufrufbar. (Näheres zu **chmod** auf der entsprechenden Handbuchseite und im Abschnitt 1.104.5 der Vorbereitung auf die LPI101 Prüfung.)

Das geht solange gut, solange wir immer mit der selben Shell arbeiten oder das Script keine Elemente enthält, die

eine andere Shell nicht verstehen würde. Aber sobald wir in unserem Script ein paar bash-Spezialitäten einbauen würden und dann plötzlich mit der C-Shell arbeiten würden fielen wir auf die Nase.

Das Problem ist also, daß unser Script selbst noch keine Angaben enthält, von welcher Shell es ausgeführt werden soll. Aber keine Angst, auch hierfür gibt es eine einfache Lösung. Jede Shell unter jedem Unix versteht eine ganz spezielle Kommentarzeile.

Wenn ein Script in der ersten Zeile die Anweisung

```
#!Programm
```

aufweist, so startet jede Shell das *Progamm* und gibt ihm als Parameter den Namen des Scripts mit. Wenn wir also in Zukunft grundsätzlich als erste Zeile unserer Scripts schreiben:

```
#!/bin/bash
```

sind wir auf der sicheren Seite. Selbst wenn wir mit der C-Shell oder einem ganz anderen Kommandointerpreter arbeiten wird für die Abarbeitung unseres Scripts jetzt grundsätzlich die bash benutzt.

Ein häufiger Fehler ist es, daß ein Schreibfehler in diese erste Zeile gemacht wird. Wenn dann das Script ausgeführt werden soll, dann findet die Shell den Interpreter nicht, da der ja falsch geschrieben ist. Also bringt sie eine etwas mißverständliche Fehlermeldung:

```
bash: Scriptname: No such file or directory
```

Das könnte man jetzt mißverstehen und denken, die Shell hätte das Script nicht gefunden. In Wahrheit hat sie den Interpreter nicht gefunden...

Kommandozeilenparameter

Was wäre ein Shellscript, wenn wir ihm keine Parameter übergeben könnten? Natürlich bietet die Shell diese Möglichkeit, noch dazu auf eine sehr einfache Art und Weise.

Zunächst noch einmal die Erinnerung, was sind Kommandozeilenparameter? Wenn wir ein Script schreiben, das z. B. `addiere` heißt und das zwei Zahlen addieren soll, dann werden wir vom Anwender des Scripts erwarten, daß er die beiden zu addierenden Zahlen als Parameter übergibt, daß er also z.B. schreibt:

```
addiere 10 20
```

In diesem Fall wäre der Kommandozeilenparameter Nummer 1 also die 10, der Parameter2 die 20. Innerhalb der Shell können diese Parameter angesprochen werden mit `$1` und `$2`. Unser Script könnte also folgendermaßen aussehen:

```
#!/bin/bash
let ergebnis=$1+$2
echo $1 plus $2 ergibt $ergebnis
```

Spezielle Variablen für die Kommandozeilenparameter

Grundsätzlich unterstützt die Bourne-Shell bis zu neun Parameter (`$1 - $9`), die direkt angesprochen werden können. Die Bourne-Again-Shell (bash) kann bis zu 99 Parameter direkt ansprechen, indem die Nummern der Parametr oberhalb des neunten in geschweifte Klammern gesetzt werden (`${10} - ${99}`). Der Parameter `$0` enthält, wie in allen anderen Hochsprachen auch, den Namen des Scripts, wie es aufgerufen wurde:

Script	Parameter1	Parameter2	Parameter3	Parameter4	...
$0	$1	$2	$3	$4	...

Oft ist es gar nicht notwendig, die Parameter einzeln anzusprechen. Wenn wir z.B. einen Unix-Befehl umbenennen

wollten, z.B. cp in kopiere, dann wäre es ja lästig, wenn wir innerhalb des Scripts die ganzen denkbaren Parameter einzeln angeben müssten. Dazu gibt es die Sonderform $@ und $*, die beide alle angegebenen Parameter bezeichnen. In unserem Script müßten also nur die Zeilen stehen:

```
#!/bin/bash
cp $*
```

Egal, wieviele Parameter jetzt angegeben wurden, alle werden einfach durch das $* übermittelt.

Sehr häufig ist es notwendig zu erfahren, wieviele Parameter überhaupt angegeben wurden. Dazu dient die spezielle Variable $#.

Zusammenfassend existieren also folgende spezielle Variablen für die Kommandozeilenparameter:

${n}	Der nte Parameter bei mehr als 9 Parametern (nur bash)
$@	Steht für alle angegebenen Parameter
$*	Steht für alle angegebenen Parameter
$#	Steht für die Anzahl aller angegebenen Parameter

Der Unterschied zwischen $* und $@ ist marginal und wird sehr selten gebraucht. Er bezieht sich nur auf die Frage, wie die Shell reagiert, wenn $* oder $@ in doppelten Anführungszeichen stehen.

Das	wird ersetzt durch
"$*"	"$1 $2 $3 ..."
"$@"	"$1" "$2" "$3" "..."

Der Befehl shift

Der Befehl shift verschiebt die ganze Kette der Kommandozeilenparameter um eines nach links. Das bedeutet, der Parameter 2 wird zum Parameter1, der Parameter3 zum Parameter2 usw. Der erste Parameter fällt weg. Damit kann eine unbestimmte Anzahl von Parametern bearbeitet werden, indem in einer Schleife immer nur der erste Parameter verarbeitet wird und anschließend der Befehl shift aufgerufen wird. Die Schleife wird solange wiederholt, bis keine Parameter mehr übrig sind.

Auch wenn die Schleifenkonstruktion noch unbekannt ist folgt hier ein Beispiel. Das folgende Script gibt alle eingegebenen Parameter untereinander aus:

```
#!/bin/bash
while [ $# -gt 0 ]   #Solange die Anzahl der Parameter ($#) größer 0
do
  echo $1            #Ausgabe des ersten Parameters
  shift              #Parameter verschieben $2->$1, $3->$2, $4->$3,...
done
```

Der Befehl set

Normale Variablen bekommen ihre Werte durch das Gleichheitszeichen. Die Konstruktion

Variablenname=Wert

ist aber für Kommandozeilenparameter nicht möglich. Es ist also verboten zu schreiben

```
1=Hans
```

um damit $1 den Wert Hans zu geben. Falls in seltenen Fällen es doch notwendig sein sollte, die Kommandozeilenparameter zu ändern, so kann das mit dem Befehl **set** erledigt werden. Allerdings können nur alle

Parameter gleichzeitig verändert werden. **set** schreibt einfach die gesamte Parameterkette neu.

Das heißt, alle Parameter, die dem Befehl **set** übergeben werden, werden so behandelt, als wären sie dem Script übergeben werden. Die echten Parameter, die dem Script wirklich übergeben wurden fallen dadurch alle weg, auch wenn **set** weniger Parameter erhält, als dem Script mitgegeben wurden.

Diese Anwendung von **set** macht zum Beispiel Sinn, wenn wir ein Script schreiben wollen, das zwingend zwei Kommandoparameter braucht. Wenn wir am Anfang die Anzahl der Parameter überprüfen und merken, daß das Script keine Parameter bekommen hat, so können wir mit **set** voreingestellte Parameter definieren.

Bedingungsüberprüfungen

Eine der grundlegendsten Dinge beim Programmieren ist die Überprüfung von verschiedenen Bedingungen, um damit auf bestimmte Gegebenheiten zu reagieren. Die Überprüfung von Bedingungen hat in der Regel bei Programmiersprachen zwei verschiedene Formen, die if-Anweisung und eine Mehrfachauswahl. Beide werden von der Shell unterstützt.

Die if-Anweisung

Die einfache if-Anweisung

Grundsätzlich hat die if-Anweisung der Bourne-Shell eine sehr einfache Form. Nach dem if steht ein Befehl, der ausgeführt wird. Gibt dieses Kommando eine 0 als Rückgabewert zurück, so gilt die Bedingung als erfüllt und die Aktionen, die zwischen dem folgenden then und fi stehen werden ausgeführt.

```
if Kommando
then
  Aktion
  Aktion
  ...
fi
```

Natürlich sind die Aktionen auch wieder normale Unix-Befehle. Das "fi", das den Block beendet, der durch "if ... then" begonnen wurde, ist einfach nur das "if" rückwärts geschrieben.

Das Programm test

Damit es jetzt sinnvolle Möglichkeiten gibt, Bedingungen zu überprüfen brauchen wir ein Programm, das verschiedene Tests durchführt und jeweils bei gelungenem Test eine 0 als Rückgabewert zurückgibt, bei mislingenem Test eine 1. Dieses Programm heißt **test** und ermöglicht alle wesentlichen Bedingungsüberprüfungen, die für das Shell-Programmieren notwendig sind.

Damit wir nicht jedesmal schreiben müssen

```
if test ...
```

gibt es einen symbolischen Link auf das Programm **test**, der einfach **[** heißt. Allerdings verlangt das Programm **test**, wenn es merkt, daß es als **[** aufgerufen wurde, auch als letzten Parameter eine eckige Klammer zu. Damit ist es also möglich zu schreiben:

```
if [ ... ]
```

Wichtig ist hierbei, daß unbedingt ein Leerzeichen zwischen if und der Klammer und zwischen der Klammer und den eigentlichen Tests stehen muß. Es handelt sich bei der Klammer ja tatsächlich um einen Programmaufruf!

Die verschiedenen Bedingungsüberprüfungen mit test bzw. [

-r *Dateiname*

Die Datei *Dateiname* existiert und ist lesbar

-w *Dateiname*
Die Datei *Dateiname* existiert und ist beschreibbar

-x *Dateiname*
Die Datei *Dateiname* existiert und ist ausführbar

-d *Dateiname*
Die Datei *Dateiname* existiert und ist ein Verzeichnis

-s *Dateiname*
Die Datei *Dateiname* existiert und ist nicht leer

-b *Dateiname*
Die Datei *Dateiname* existiert und ist ein blockorientiertes Gerät

-c *Dateiname*
Die Datei *Dateiname* existiert und ist ein zeichenorientiertes Gerät

-g *Dateiname*
Die Datei *Dateiname* existiert und das SGID-Bit ist gesetzt

-k *Dateiname*
Die Datei *Dateiname* existiert und das Sticky-Bit ist gesetzt

-u *Dateiname*
Die Datei *Dateiname* existiert und das SUID-Bit ist gesetzt

-p *Dateiname*
Die Datei *Dateiname* existiert und ist ein Named Pipe

-e *Dateiname*
Die Datei *Dateiname* existiert

-f *Dateiname*
Die Datei *Dateiname* existiert und ist eine reguläre Datei

-L *Dateiname*
Die Datei *Dateiname* existiert und ist ein symbolischer Link

-S *Dateiname*
Die Datei *Dateiname* existiert und ist ein Socket

-O *Dateiname*
Die Datei *Dateiname* existiert und ist Eigentum des Anwenders, unter dessen UID das test-Programm gerade läuft

-G *Dateiname*
Die Datei *Dateiname* existiert und gehört zu der Gruppe, zu der der User gehört, unter dessen UID das test-Programm gerade läuft

Datei1* -nt *Datei2
Datei1 ist neuer als *Datei2* (newer than)

Datei1* -ot *Datei2
Datei1 ist älter als *Datei2* (older than)

Datei1* -ef *Datei2
Datei1 und *Datei2* benutzen die gleiche I-Node (equal file)

-z *Zeichenkette*
Wahr wenn *Zeichenkette* eine Länge von Null hat.

-n *Zeichenkette*
Wahr wenn *Zeichenkette* eine Länge von größer als Null hat.

Zeichenkette1* = *Zeichenkette2
Wahr wenn *Zeichenkette1* gleich *Zeichenkette2*

Zeichenkette1* != *Zeichenkette2
Wahr wenn *Zeichenkette1* ungleich *Zeichenkette2*

Wert1* -eq *Wert2
Wahr, wenn *Wert1* gleich *Wert2* (equal)

Wert1* -ne *Wert2
Wahr, wenn *Wert1* ungleich *Wert2* (not equal)

Wert1* -gt *Wert2
Wahr, wenn *Wert1* größer *Wert2* (greater than)

Wert1* -ge *Wert2
Wahr, wenn *Wert1* größer oder gleich *Wert2* (greater or equal)

Wert1* -lt *Wert2
Wahr, wenn *Wert1* kleiner *Wert2* (less than)

Wert1* -le *Wert2

Wahr, wenn *Wert1* kleiner oder gleich *Wert2* (less or equal)

!*Ausdruck*

Logische Verneinung von *Ausdruck*

Ausdruck* -a *Ausdruck

Logisches UND. Wahr, wenn beide Ausdrücke wahr sind

Ausdruck* -o *Ausdruck

Logisches ODER. Wahr wenn mindestens einer der beiden Ausdrücke wahr ist

Mit diesen Tests sind so ziemlich alle denkbaren Bedingungsüberprüfungen möglich, die in einem Shellscript notwendig sind.

Die erweiterte if-else Anweisung

Natürlich bietet die if-Anweisung auch eine Erweiterung zur normalen Form, die sogenannte if-else Anweisung. Es ist also möglich zu schreiben:

```
if [ Ausdruck ]
then
  Kommandos
else
  Kommandos
fi
```

Die if-elif-else Anweisung

Um noch einen Schritt weiterzugehen bietet die if-Anweisung sogar ein weiteres if im else, das sogenannte elif, das wieder eine Bedingung überprüft:

```
if [ Ausdruck ]
then
  Kommandos
elif [ Ausdruck ]
then
  Kommandos
else
  Kommandos
fi
```

Mehrfachauswahl mit case

Oft kommt es vor, daß eine Variable ausgewertet werden muß und es dabei viele verschiedenen Möglichkeiten gibt, welche Werte diese Variable annehmen kann. Natürlich wäre es mit einer langen if-elif-elif-elif... Anweisung möglich, so etwas zu realisieren, das wäre aber sowohl umständlich, als auch schwer zu lesen. Damit solche Fälle einfacher realisiert werden können, gibt es die Mehrfachauswahl mit case. Der prinzipielle Aufbau einer case-Entscheidung sieht folgendermaßen aus:

```
case Variable in
  Muster1) Kommando1 ;;
  Muster2) Kommando2 ;;
  Muster3) Kommando3 ;;
  ...
esac
```

Zu beachten sind zunächst die Klammern, die das Muster abschließen. Das Kommando, das zum jeweiligen Muster passt muß mit zwei Strichpunkten abgeschlossen werden. Statt einem Kommando können nämlich auch mehrere Kommandos, durch Strichpunkt getrennt stehen. Nur die doppelten Strichpunkte machen der Shell klar, daß das Kommando für den bestimmten Fall jetzt fertig ist.

Der Abschluß mit esac ist wieder einfach das Wort case rückwärts geschrieben.

Programmschleifen im Script

Programmierung, insbesondere die heute übliche Form der strukturierten Programmierung ist ohne Schleifen nicht möglich. Unter Programmschleifen versteht man eine Wiederholung eines Teils des Programms, bis eine bestimmte Bedingung erfüllt ist.

Die Shell bietet zwei Formen der Schleifen. die Kopfgesteuerte Schleife und die for-Schleife, die eine Liste abarbeitet.

Die kopfgesteuerte Schleife mit while

Die kopfgesteuerte Schleife überprüft vor jedem Schleifendurchgang die Bedingung, die festlegt, ob die Schleife tatsächlich nochmal durchlaufen werden soll. Im Extremfall wird diese Schleife also kein einziges Mal durchlaufen, wenn nämlich die Bedingung schon von vorneherein nicht wahr ist.

Als Bedingung wird wieder jedes Programm akzeptiert, das einen Rückgabewert liefert. Ein Rückgabewert von 0 bedeutet, die Bedingung ist wahr, jeder andere bedeutet unwahr. Wie bei der if-Anweisung wird hier meistens wieder das Programm **test** oder eben dessen abgewandelte Form **[** benutzt. Die verschiedenen Überprüfungen wurden bei der if-Anweisung detailiert dargestellt.

Die prinzipielle Form der while-Schleife mit dem [-Programm als Bedingungsüberprüfung sieht dann so aus:

```
  while [ Ausdruck ]
do
  Kommandos...
done
```

Die Listenschleife mit for

Eine Listenschleife durchläuft die Schleife so oft, wie die Liste Elemente hat. Bei jedem Schleifendurchlauf bekommt die Variable den Wert des jeweiligen Listenelements.

```
  for Variable in Liste
  do
    Kommandos...
  done
```

Als Liste gilt dabei jede mit Leerzeichen, Tabs oder Zeilentrennern getrennte Liste von Worten. Damit diese Funktionsweise etwas klarer wird ein einfaches Beispiel:

```
  #!/bin/bash
  for i in Hans Peter Otto
  do
    echo $i
  done
```

Diese Schleife würde also dreimal durchlaufen. Im ersten Durchgang bekommt die Variable i den Wert "Hans", im zweiten "Peter" und im dritten "Otto". Die Ausgabe der Schleife wäre also einfach

```
  Hans
  Peter
  Otto
```

Richtig interessant wird diese Schleife jedoch, wenn als Liste ein Jokerzeichenkonstrukt steht wie etwa `*.txt` - Die Shell löst dieses Konstrukt ja in eine Liste aller Dateien im aktuellen Verzeichnis auf, die auf das Muster passen. Die Schleife wird also sooft durchlaufen, wie es Dateien gibt, die die Endung `.txt` vorweisen...

Eine andere häufige Form der Anwendung ist die Abarbeitung aller Kommandozeilenparameter eines Shellscripts.

Die Variable $@ bietet ja diese Parameter alle zusammen. Diese Schleife wird so oft benutzt, daß es dafür eine Sonderform gibt, statt zu schreiben:

```
for name in $@
do
  ...
done
```

reicht es zu schreiben

```
for name
do
  ...
done
```

Eine große Stärke der for-Schleife ist es, als Liste die Ergebnisse eines Unix-Befehls einzugeben. Mit Hilfe der Kommandosubstitution ist es ohne weiteres möglich, die komplexesten Unix-Befehle einzugeben, die als Ergebnis eine Liste ausgeben und diese Liste dann in der Schleife zu verarbeiten.

So kann man beispielsweise eine Liste aller User, die dem System bekannt sind dadurch erhalten, daß man mit dem Befehl **cut** die erste Spalte der Datei `/etc/passwd` ausschneidet. Der Befehl würde folgendermaßen aussehen:

```
cut -d: -f1 /etc/passwd
```

Um die Liste, die dieser Befehl ausgibt, als Liste für die for-Schleife zu nutzen muß der Befehl ja nur in Grave-Zeichen gesetzt werden, also

```
#!/bin/bash
for i in `cut -d: -f1 /etc/passwd`
do
  echo $i
done
```

Shellfunktionen

Shellfunktionen sind kleine Unterprogramme, die einen eigenen Namen haben. Sie können sowohl im Script, als auch in der interaktiven Shell verwendet werden. Hier ist natürlich die Funktionsweise innerhalb eines Scripts Objekt der Darstellung.

Die Syntax von Funktionen wurden bereits im letzten Kapitel beschrieben, hir nur noch ein paar Anwendungsbeispiele im Script.

Als (zweifellos sinnloses) Beispiel folgt hier ein kleines Script, das eine Funktion enthält, die von einem bestimmten Startwert zu einem bestimmten Endwert zählt. Start- und Endwert werden im Hauptprogramm erfragt, die eigentliche Aufgabe des Zählens erledigt die Funktion:

```
#!/bin/bash

function zaehle_von_bis()
{
  i=$1                        # i bekommt den Wert des ersten Parameters
  while [ $i -le $2 ]         # Solange i kleiner/gleich Parameter2
  do
    echo $i                   # Ausgabe der Zahl i
    let i=$i+1                # i wird um 1 erhöht
  done
}
```

```
# Das Hauptprogramm

read -p "Startwert: " zahl1   # Startwert einlesen
read -p "Endwert: " zahl2     # Endwert einlesen
zaehle_von_bis $zahl1 $zahl2  # Aufruf der Funktion mit den gelesenen Werten
```

Rückgabewerte überprüfen

Wenn innerhalb eines Scripts ein Befehl aufgerufen wurde, so kann es für den weiteren Ablauf sehr wichtig sein, ob der Befehl erfolgreich war oder nicht. Diese Frage beantworten uns die Rückgabewerte (exit-codes). Jedes Programm gibt dem Betriebssystem einen ganzzahligen Rückgabewert zurück, wenn es abgeschlossen ist. Ist dieser Wert 0, dann bedeutet es eine fehlerfreie Beendigung des Programms, in allen anderen Fällen bedeutet es einen Fehler.

Das Programm **test** beispielsweise nutzt diese Fähigkeit, um bestimmte Tests durchzuführen und die Ergebnisse als Rückgabewert (0 bedeutet Test war erfolgreich also wahr) zurückzuliefern.

Die Shell kennt drei Methoden, diesen Rückgabewert zu analysieren:

Die Variable $?
Die Variable $? enthält immer den Rückgabewert des letzten ausgeführten Programms. Sie kann nach dem Aufruf eines Programms z.B. mit der if-Bedingungsüberprüfung abgefragt werden:

```
...
Programm
if [ $? -eq 0 ]
then
  ...       # Programm war erfolgreich
fi
```

Sobald ein anderes Programm abgelaufen ist, hat die Variable $? aber bereits einen anderen Wert, den des neuen Programms eben.

Verknüpfung zweier Programme mit &&
Wenn zwei Programme mit zwei direkt aufeinanderfolgenden Ampersands (&)verknüpft werden, so wird das zweite Programm nur ausgeführt, wenn das erste mit Rückgabewert 0 abgeschlossen wurde.

Verknüpfung zweier Programme mit ||
Wenn zwei Programme mit zwei direkt aufeinanderfolgenden Pipe-Symbolen (|)verknüpft werden, so wird das zweite Programm nur ausgeführt, wenn das erste mit Rückgabewert ungleich 0 abgeschlossen wurde.

Eine häufig benutzte Zeile, die die dritte Möglichkeit ausnutzt ist

```
test -x /usr/bin/foo || exit
```

Wenn das Programm `/usr/bin/foo`nicht existiert und ausführbar ist, wird das Script mit `exit`beendet.

Mail an root unter bestimmten Bedingungen

Ein Script wird häufig automatisch abgearbeitet auch wenn kein User am Terminal anwesend ist. Wenn es jetzt eine Meldung an den Benutzer oder den Systemverwalter ausgeben will, bleibt nur das Mail-System. Oder es soll eine Nachricht an den Systemverwalter abschicken, wenn bestimmte Bedingungen erfüllt sind, auch hier ist die E-Mail die beste Methode. Dazu existiert der Befehl **mail**.

Dieser Befehl schickt eine Mail mit bestimmter Titelzeile an eine bestimmte Adresse. Die Mail selbst wird entweder

aus einer Datei oder aus einem Here-Dokument von der Standard-Eingabe gelesen. Im Script ist die Form des Here-Documents am verbreitetsten:

```
mail -s Titel Adresse <<EOM
Beliebige Textzeilen
EOM
```

Alles, was zwischen den beiden EOMs steht, wird an die angegebene Adresse per Mail verschickt. Beinhaltet der Titel Leerzeichen, so muß er in Anführungszeichen gesetzt werden. Als Adresse für den Systemverwalter kann einfach `root`eingegeben werden:

```
#!/bin/bash
Auslastung=`df /dev/hda1| grep ^/ |cut -c52-54`
if [ $Auslastung -gt 90 ]
then
  mail -s "Alarm: Platte bald voll" root <<EOM
    Hallo Systemverwalter. Die Platte /dev/hda1 ist bald voll.
    Sie ist zu $Auslastung belegt.
    Mach was!!!
  EOM
fi
```

Es können aber auch Programme direkt ihre Ausgaben an **mail**weiterpipen, also etwa

```
df | mail -s "Ausgabe von df" root
```

In beiden Fällen wird eine Mail an root verschickt, im ersten Beispiel, wird eine Warnung an root weitergegeben, wenn die Platte `/dev/hda1`voller als 90% ist, im zweiten wird einfach die Ausgabe von **df**gemailt.

Speicherort von Scripts

Scripte sind nur dann ausführbar, wenn sie innerhalb des Suchpfades liegen. Bei der Wahl einer geeigneten Stelle für eigene Scripts sollte unterschieden werden, wofür sie gedacht sind.

Soll nur der Systemverwalter diese Scripts ausführen dürfen, dann empfiehlt es sich,

1. die Scripts entweder in `/usr/local/sbin` oder `/root/bin` abzuspeichern. Beide Verzeichnisse liegen nur im Suchpfad des Systemverwalters.
2. die Zugriffsmodi auf 700 zu setzen (`rwx------`), und dafür zu sorgen, daß sie root gehören. Somit kann sie nur root ausführen.

Sollen sie aber von allen Usern ausführbar sein, so sollten sie nach `/usr/local/bin`gelegt werden und den Zugriffsmodus 755 (`rwxr-xr-x`) besitzen. `/usr/local/bin`ist im Suchpfad aller User.

Grundsätzliche Vorsicht ist mit der Verwendung des SUID-Bits angeraten. Die Shell reagiert aber seit einigen Jahren sehr vorsichtig darauf und bezieht sich ihre Informationen über die Identität nicht aus der Effektiven UID sondern aus der echten UID. Ältere Versionen können hier aber erhebliche Schwierigkeiten machen...