



FH Schmalkalden  
Fak. Elektrotechnik

## **Einführung in sh und csh**



# Inhaltsverzeichnis

1	Einleitung	5
1.1	Begriff Shell . . . . .	5
1.2	Verschiedene Shells . . . . .	5
2	Kommandos	6
2.1	Aufbau von Kommandos . . . . .	6
2.2	Abarbeitung von Kommandos . . . . .	6
2.3	Ende eines Kommandos . . . . .	7
3	Die C-Shell	7
3.1	Verknüpfung von Kommandos . . . . .	7
3.2	Wildcards, Aliase, Variablen und Ersetzungen . . . . .	7
3.2.1	Übersicht . . . . .	7
3.2.2	Wildcards . . . . .	8
3.2.3	Lokale Variablen . . . . .	8
3.2.4	Aliase . . . . .	11
3.2.5	Ersetzung . . . . .	12
3.2.6	Der History-Mechanismus . . . . .	13
3.3	Vordefinierte Variablen . . . . .	16
3.4	Aufruf-Parameter und Variablen in Scripten . . . . .	16
3.5	Umgebungsvariablen . . . . .	16
3.6	Wichtige Variablen bei der Arbeit mit csh . . . . .	17
3.7	source, exec und exit . . . . .	19
3.8	Nutzer-Einstellungen . . . . .	20
3.8.1	Start eines Shell-Prozesses . . . . .	20
3.8.2	Standard-Zugriffsrechte auf Dateien . . . . .	20
3.8.3	Prompt-Zeichen . . . . .	20
3.8.4	Einstellungen für Vorsichtige . . . . .	20
3.8.5	Standard-Editor festlegen . . . . .	21
3.8.6	Standard-Drucker festlegen . . . . .	21
3.9	Ein- und Ausgabeumlenkung . . . . .	22
3.10	Programmieren mit der C-Shell . . . . .	24
3.10.1	Ausdrücke und Operatoren . . . . .	24
3.10.2	Binäre Verzweigung . . . . .	25
3.10.3	Schleifen . . . . .	26
3.10.4	Mehrfachverzweigung . . . . .	27
3.10.5	Sprungmarken . . . . .	28
3.10.6	Beispiele . . . . .	29
4	Die Bourne-Shell	32
4.1	Verknüpfung von Kommandos . . . . .	32

4.2	Wildcards, Variablen und Ersetzungen . . . . .	33
4.2.1	Übersicht . . . . .	33
4.2.2	Wildcards . . . . .	34
4.2.3	Lokale Variablen . . . . .	34
4.2.4	Ersetzungen . . . . .	35
4.3	Vordefinierte Variablen . . . . .	36
4.4	Aufruf-Parameter und Variablen in Scripten . . . . .	36
4.5	Umgebungsvariablen . . . . .	36
4.6	Wichtige Variablen bei der Arbeit mit sh . . . . .	37
4.7	., exec und exit . . . . .	39
4.8	Nutzer-Einstellungen . . . . .	40
4.9	Ein- und Ausgabeumlenkung . . . . .	41
4.10	Programmieren mit der Bourne-Shell . . . . .	42
4.10.1	Ausdrücke und Operatoren . . . . .	42
4.10.2	Binäre Verzweigung . . . . .	44
4.10.3	Schleifen . . . . .	45
4.10.4	Mehrfachverzweigung . . . . .	46
4.10.5	Beispiele . . . . .	47

# 1 Einleitung

## 1.1 Begriff Shell

Der Begriff „Shell“ bzw. Kommandointerpreter bezeichnet ein Programm, das Eingaben zeilenweise entgegennimmt und verarbeitet. Bei der Verarbeitung der Eingabe werden externe Programme gestartet, falls erforderlich.

Dabei kann die Eingabe entweder von einem Terminal oder aus einer Datei stammen. Erfolgt die Eingabe über ein Terminal, wird der Nutzer durch Ausgabe eines Promptzeichens dazu aufgefordert, eine Eingabe vorzunehmen.

## 1.2 Verschiedene Shells

Für UNIX-Systeme gibt es verschiedene Shells.

Der X/OPEN-Standard schreibt für UNIX-Systeme lediglich die Bourne-Shell (sh) vor.

Daneben werden von den verschiedenen Herstellern meist zusätzliche weitere Shells mit ausgeliefert. Als Quasi-Standard enthalten die meisten UNIX-Systeme die C-Shell (csh).

Open-Source-Betriebssysteme (z.B. Linux) enthalten nicht die Originalversionen von sh und csh sondern Clones, z.B. die Korn-Shell (ksh) und die Turbo-C-Shell (tcsh). Diese Clones stellen meist Funktionalitäten bereit, die über die der Originale hinausgehen.

Die C-Shell und ihre Clones bieten Vorteile im interaktiven Betrieb mit dem Benutzer, sie sind benutzerfreundlicher<sup>1</sup> als die Bourne-Shell und ihre Clones.

Da die Bourne-Shell auf allen UNIX-Systemen vorzufinden ist, wird sie hauptsächlich für Administrationszwecke (z.B. bei der Installation von Software, zum Starten von Diensten beim Systemstart. . .) eingesetzt, während durch normale Nutzer meist die C-Shell (bzw. deren Clones) genutzt werden. Diese Aufteilung ist allerdings nicht zwingend.

---

<sup>1</sup>meiner Meinung nach

## 2 Kommandos

### 2.1 Aufbau von Kommandos

Nach dem Prompt kann jeweils ein Kommando eingegeben werden. Jedes Kommando besteht dabei aus

- Kommandoname (Name des auszuführenden Programmes) im ersten Wort und
- Parameter, die dem Programm übergeben werden in allen folgenden Worten.

Es gibt zwei verschiedene Arten von Parametern:

- Optionen, die die Arbeitsweise des Programmes beeinflussen und
- Argumente wie z.B. Dateinamen.

Im Beispiel

```
cp -i /tmp/* .
```

ist „cp“ der Kommandoname, d.h. es wird nach einer ausführbaren Datei mit dem Namen „cp“ gesucht.

Die Option „-i“ legt fest, dass interaktiv kopiert wird, d.h. für jede einzelne Datei eine Abfrage durchgeführt wird, ob die Datei kopiert werden soll.

Die Argumente „/tmp/\*“ und „.“ sind die benötigten Argumente für die Quelldateien und das Zielverzeichnis.

### 2.2 Abarbeitung von Kommandos

Jedes Kommando wird in einem sogenannten Prozess bearbeitet. Als Prozess wird eine Instanz eines laufenden Programmes bezeichnet.

Wird beispielsweise in mehreren Terminalfenstern jeweils das Kommando

```
ls
```

eingegeben, so wird zwar (normalerweise) immer dasselbe Programm ls aus derselben Datei (meist /usr/bin/ls) gestartet, es sind jedoch mehrere Prozesse aktiv, die jeweils ihre eigene Standardeingabe, Standardausgabe und Standardfehlerausgabe besitzen. Weiterhin verfügt jeder Prozess über Informationen über sein aktuelles Verzeichnis, Unterbrechungsbehandlung und Umgebungsvariablen.

## 2.3 Ende eines Kommandos

Die Abarbeitung eines Kommandos liefert einen Wert, den sogenannten Status. Dieser wird vom Programm meist durch Aufruf der Funktion „exit()“ erzeugt. Dieser Status ist bei erfolgreicher Ausführung des Kommandos meist „0“, ansonsten wird ein Fehlercode zurückgegeben.

# 3 Die C-Shell

## 3.1 Verknüpfung von Kommandos

Kommandos können auf folgende Weise verknüpft werden:

- *cmd1 ; cmd2*  
Sequentielle Ausführung, d.h. Programm *cmd2* wird gestartet, nachdem *cmd1* beendet wurde.
- *cmd &*  
Ausführung „im Hintergrund“ (asynchrone Ausführung).  
Das Kommando wird in einem Hintergrundprozess ausgeführt. Die Prozessnummer des Hintergrundes wird ausgegeben, die Arbeit am Terminal kann fortgesetzt werden, während der Prozess noch läuft.
- *cmd1 && cmd2*  
Nur wenn *cmd1* erfolgreich beendet wurde, wird *cmd2* ausgeführt.
- *cmd1 || cmd2*  
Nur wenn *cmd1* nicht erfolgreich beendet wurde, wird *cmd2* ausgeführt.
- *cmd1 | cmd2*  
Die Kommandos bilden eine Pipe, d.h. die Standardausgabe von *cmd1* wird vom Kommando *cmd2* als Standardeingabe verarbeitet.  
Der exit-Wert von *cmd2* (bzw. dem letzten Kommando der Pipe) ergibt den exit-Wert der Verknüpfung.

## 3.2 Wildcards, Aliase, Variablen und Ersetzungen

### 3.2.1 Übersicht

Wildcards, Aliase, Variablen und Ersetzungen sind Mechanismen, bei denen Text aus der Eingabezeile durch anderen Text ersetzt wird.

Der Ersetzungstext kann dabei aus vorher definierten Variablen, aus der Inspektion von Verzeichnisinhalten oder aber aus anderen Kommandos stammen.

Diese Mechanismen werden meist genutzt, um möglichst wenig Text eingeben zu müssen bzw. um Tippfehlern vorzubeugen.

Die Ersetzungen finden vor der eigentlichen Verarbeitung der Zeilen statt.

### 3.2.2 Wildcards

Der Kopierbefehl in Kapitel 2.1 auf Seite 6 benutzte bereits Wildcards, d.h. Sonderzeichen, die für beliebige Zeichenketten in Dateinamen stehen können.

Folgende Wildcard-Konstruktionen können für Dateinamen verwendet werden:

Wildcard	Bedeutung
*	beliebige 0..14 Zeichen lange Zeichenkette, auch leere
?	ein beliebiges Zeichen
[abcdefg123]	eines der Zeichen a, b, c, d, e, f, g, 1, 2 oder 3
[a-c]	eines der Zeichen a, b oder c
{a,b}	erzeugt zwei Texte, wobei in dem einen der Text <i>a</i> steht, im anderen <i>b</i>
~	das HOME-Verzeichnis des aktuellen Nutzers
~ <i>uname</i>	das HOME-Verzeichnis des Nutzers <i>uname</i>

### 3.2.3 Lokale Variablen

In einer Shell können Variablen mit Text belegt werden, der dann später durch Bezug auf die Variable wieder abgerufen werden kann.

Lokale Variablen gelten nur innerhalb des aktuellen Prozesses und werden nicht an Subprozesse weitergegeben, die von der Shell gestartet wurden.

Zum Setzen von Variablen dient der „set“-Befehl

```
set <name> = <value>
```

Hierbei muss anstelle von *<name>* der Name der zu setzenden Variable eingesetzt werden und anstelle von *<value>* der Text. Im Beispiel

```
set dir = "/etc"  
ls $dir
```

wird der Variablen *dir* der Wert „/etc“ zugewiesen.

Im anschließenden ls-Befehl wird die Variable benutzt, dabei wird durch die Shell automatisch

```
$name  
${name}
```

durch den Wert der Variablen *name* ersetzt.

Folgende weitere Formen zum Setzen von Variablen sind möglich:

- `set name`  
definiert die Variable *name* mit der leeren Zeichenkette als Wert.

- `set name = (wordlist)`  
verwendet eine Liste von Worten (getrennt durch Whitespaces), um die Variable „name“ zu definieren.  
Die Klammern sind im Wert der Variablen nicht enthalten.

Zur Verwendung/Ersetzung von Variablen gibt es folgende Sonderformen:

- `$#name`  
 `${name}`  
liefert die Anzahl der Wörter im Wert der Variablen *name*.
- `$?name`  
 `${name}`  
liefert 1, wenn *name* definiert ist, ansonsten 0.
- `$name:r`  
 `${name:r}`  
entfernt den letzten Suffix (die letzte Dateinamens-Endung).
- `$name:t`  
 `${name:t}`  
entfernt führende Pfadnamen-Komponenten (nur der Dateiname bleibt übrig).
- `$name:h`  
 `${name:h}`  
entfernt die letzte Pfadnamen-Komponente (nur der Verzeichnisname bleibt übrig).

Abb. 1 auf der nächsten Seite zeigt den Einsatz der Variablen-Modifizierer.

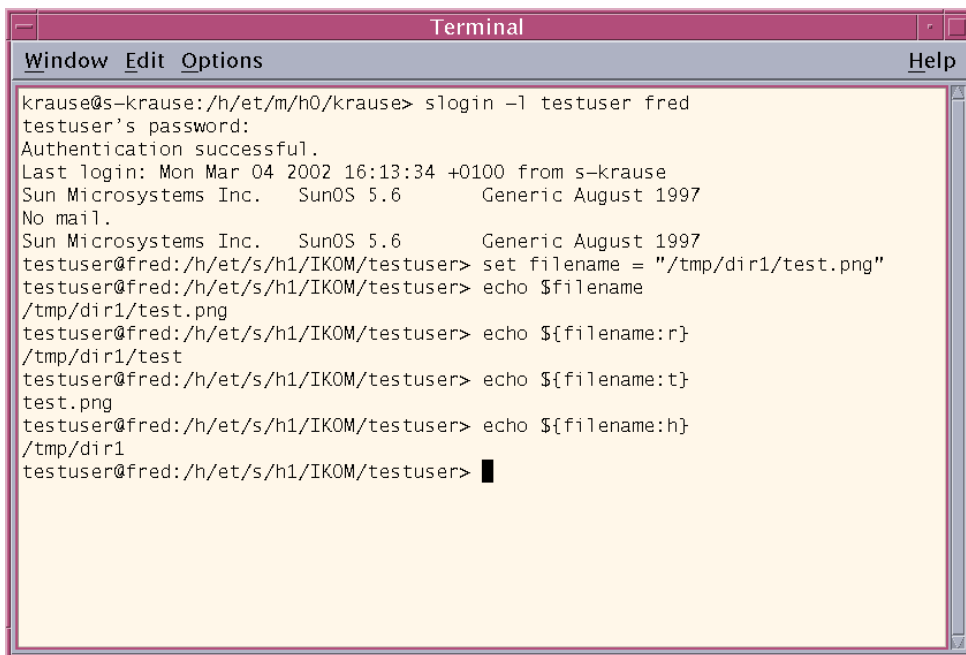
Mit

```
unset <name>
```

wird die Variable *<name>* wieder entfernt, z.B. entfernt

```
unset dir
```

die Variable *dir*.



The image shows a terminal window titled "Terminal" with a menu bar containing "Window", "Edit", "Options", and "Help". The terminal output is as follows:

```
krause@s-krause:/h/et/m/h0/krause> slogin -l testuser fred
testuser's password:
Authentication successful.
Last login: Mon Mar 04 2002 16:13:34 +0100 from s-krause
Sun Microsystems Inc. SunOS 5.6 Generic August 1997
No mail.
Sun Microsystems Inc. SunOS 5.6 Generic August 1997
testuser@fred:/h/et/s/h1/IKOM/testuser> set filename = "/tmp/dir1/test.png"
testuser@fred:/h/et/s/h1/IKOM/testuser> echo $filename
/tmp/dir1/test.png
testuser@fred:/h/et/s/h1/IKOM/testuser> echo ${filename:r}
/tmp/dir1/test
testuser@fred:/h/et/s/h1/IKOM/testuser> echo ${filename:t}
test.png
testuser@fred:/h/et/s/h1/IKOM/testuser> echo ${filename:h}
/tmp/dir1
testuser@fred:/h/et/s/h1/IKOM/testuser> █
```

Abbildung 1: Variablen-Modifizierer im Einsatz

### 3.2.4 Aliase

Mit einem Alias kann eine Abkürzung – beispielsweise für häufig genutzte Kommandos – definiert werden. Mit

```
alias <name> <value>
```

wird der Alias mit dem Namen *<name>* definiert, der den Wert *<value>* erhält. Beispielsweise wird mit

```
alias ll "ls -al"
```

erreicht, dass die Shell bei der Eingabe von „ll“ vor der Befehlsausführung dieses automatisch durch „ls -al“ ersetzt.

Wird

```
alias
```

ohne Argumente aufgerufen, wird eine Liste aller derzeit definierten Aliase ausgegeben.

Mit

```
unalias <name>
```

wird die Definition des Aliases *<name>* entfernt,

```
unalias ll
```

entfernt also den Alias für „ll“.

Mitunter soll ein Alias so gesetzt werden, dass zwei (bzw. mehrere) Kommandos ausgeführt werden sollen, wobei das erste Kommando die Parameter erhalten soll. In diesem Fall wird die Zeichenkette „\!\*“ an der Stelle in den Alias eingefügt, an der die Parameter genutzt werden sollen. Der gesamte Alias-Wert muss dann in einfache Anführungszeichen gesetzt werden, um die Interpretation der Sonderzeichen zu verhindern.

Mit

```
alias cd 'cd \!* ; set prompt = (/usr/bin/pwd'\>\ )'  
alias chdir 'chdir \!* ; set prompt = (/usr/bin/pwd'\>\ )'
```

wird veranlasst, dass bei jedem Aufruf der Befehle „cd“ bzw. „chdir“ nicht nur in das entsprechende Verzeichnis gewechselt wird sondern auch das Promptzeichen so angepasst wird, dass es das aktuelle Verzeichnis anzeigt.

### 3.2.5 Ersetzung

Kommt in der Eingabe

- *'cmd'*

vor, wird es durch die Standardausgabe des Kommandos *cmd* ersetzt.

### 3.2.6 Der History-Mechanismus

Der History-Mechanismus erlaubt es, sich auf vorangegangene Kommandos zu beziehen.

Um den History-Mechanismus zu nutzen, muss die lokale Variable „history“ (siehe Abschnitt 3.6 auf Seite 17) gesetzt werden, diese muss die Anzahl der Kommandos enthalten, die gespeichert werden soll. Das Setzen der Variable erfolgt zweckmäßigerweise in der Datei „\$HOME/.cshrc“ (siehe Abschnitt 3.8 auf Seite 20). Mit dem Befehl

```
history
```

werden die noch im History-Mechanismus gespeicherten Kommandos angezeigt.

Mit folgenden Konstruktionen kann auf vorangegangene Kommandos zurückgegriffen werden:

- `!!`  
das vorherige Kommando
- `!#`  
das laufende Kommando
- `!n`  
Kommando Nummer *n*
- `!-n`  
vom laufenden Kommando *n* Kommandos zurückgezählt
- `!string`  
das letzte mit dem *string* beginnende Kommando
- `!?string?`  
das letzte Kommando, das das Muster *string* enthielt

Mit Hilfe der vorangegangenen Konstrukte können Kommandozeiger *c* gebildet werden, aus denen mit den nachfolgenden Konstrukten einzelne Worte extrahiert werden können.

- `!c^`  
erstes Argument nach dem Kommandonamen
- `!c$`  
letztes Argument
- `!c*`  
alle Argumente
- `!c-n`  
Wörter 0 bis *n*

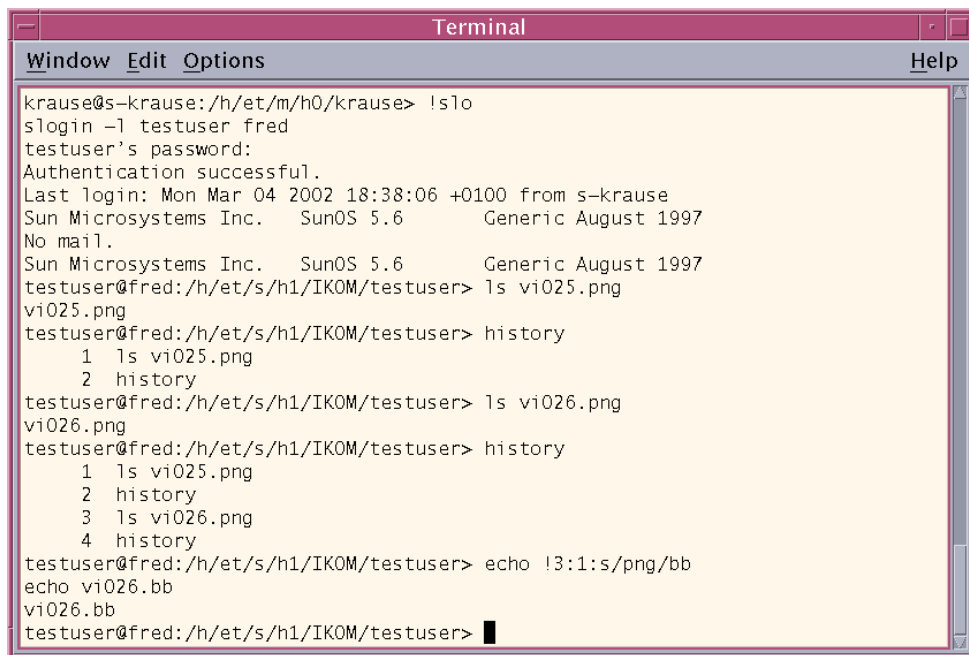
- `!c:0`  
Wort 0 (Kommandoname)
- `!c:n`  
Wort Nummer  $n$
- `!c:m-n`  
Worte  $m \dots n$
- `!c:m*`  
alle verbleibenden Wörter ab einschließlich Wort Nummer  $m$ .
- `!c:m-`  
alle verbleibenden Wörter ab einschließlich Wort Nummer  $m$  ausschließlich des letzten Wortes.

Wurde mit den vorangegangenen Konstrukten ein Kommando  $c$  oder ein Wort  $w$  ausgewählt, können die folgenden Modifikatoren angehängt werden:

- `:s/alt/neu`  
ersetzt die Zeichenfolge *alt* durch die Zeichenfolge *neu*. Steht in *neu* ein „&“, so wird der Text *alt* an dieser Stelle eingesetzt.
- `:gs/alt/neu`  
führt die Ersetzung in allen Wörtern des Kommandos aus.
- `:p`  
druckt das modifizierte Kommando, aber startet es nicht.
- `:t`  
entfernt führende Pfadnamen, lässt nur den Dateinamen übrig.
- `:h`  
entfernt letzte Komponente im Pfadnamen, lässt Verzeichnisnamen übrig.
- `:r`  
entfernt die letzte Dateieindung.

In Abb. 2 auf der nächsten Seite wählt der Historybezug im „echo“-Befehl aus Kommando 3 das erste Argument und substituiert die Zeichenkette „png“ durch „bb“.

Ich persönlich nutze den History-Mechanismus fast ausschließlich, um durch Eingabe von „! $n$ “, „! $n$ “ bzw. „!*string*“ bereits eingegebene Kommandos nochmals neu zu starten, ohne sie komplett neu eintippen zu müssen.

A terminal window titled "Terminal" with a menu bar containing "Window", "Edit", "Options", and "Help". The terminal shows a user logging in as "testuser" and performing several commands. The "history" command is used twice to show the list of previously entered commands. The first history shows "1 ls vi025.png" and "2 history". The second history shows "1 ls vi025.png", "2 history", "3 ls vi026.png", and "4 history". The terminal ends with the command "echo !3:1:s/png/bb" which outputs "vi026.bb".

```
krause@s-krause:/h/et/m/h0/krause> !slo
slogin -l testuser fred
testuser's password:
Authentication successful.
Last login: Mon Mar 04 2002 18:38:06 +0100 from s-krause
Sun Microsystems Inc. SunOS 5.6 Generic August 1997
No mail.
Sun Microsystems Inc. SunOS 5.6 Generic August 1997
testuser@fred:/h/et/s/h1/IKOM/testuser> ls vi025.png
vi025.png
testuser@fred:/h/et/s/h1/IKOM/testuser> history
 1  ls vi025.png
 2  history
testuser@fred:/h/et/s/h1/IKOM/testuser> ls vi026.png
vi026.png
testuser@fred:/h/et/s/h1/IKOM/testuser> history
 1  ls vi025.png
 2  history
 3  ls vi026.png
 4  history
testuser@fred:/h/et/s/h1/IKOM/testuser> echo !3:1:s/png/bb
echo vi026.bb
vi026.bb
testuser@fred:/h/et/s/h1/IKOM/testuser> █
```

Abbildung 2: History-Mechanismus in Aktion

### 3.3 Vordefinierte Variablen

Wenn eine C-Shell läuft, sind eine Reihe von Variablen von Beginn an definiert, u.a.:

- \$\$  
Prozessnummer der aktuellen C-Shell

### 3.4 Aufruf-Parameter und Variablen in Scripten

Shell-Scripte können wie ausführbare Programme Parameter erhalten. Diese sind im Script als „\$1“, „\$2“... ansprechbar.

Mit „\$\*“ sind alle Parameter gesammelt abrufbar.

Alternativ dazu kann auch die Schreibweise „\$argv[1]“, „\$argv[2]“... bzw „\$argv[\*]“ genutzt werden.

Bei der Ausführung von Scripten sind folgende Variablen definiert:

- \$0  
Name der Datei, von der die Kommandos gelesen werden.
- \$?0  
liefert 1, wenn die Kommando-Datei bekannt ist, ansonsten 0.

Soll ein C-Shell-Script eine Zeile aus der Standardeingabe einlesen, wird hierfür die Pseudo-Variable

- \$<

genutzt.

### 3.5 Umgebungsvariablen

Die bereits besprochenen lokalen Variablen sind nur innerhalb des Prozesses gültig, der die Shell ausführt.

Wird durch die Shell ein neuer Prozess erzeugt – beispielsweise beim Start eines Programmes – kann dieser neue Prozess nicht auf die lokalen Variablen der Shell zugreifen.

Im Unterschied dazu gibt es die sogenannte Umgebung. Dabei handelt es sich um eine Sammlung von Variablen, die Unterprozessen (und dann wiederum auch deren Unterprozessen) mitgegeben wird.

Mit

```
setenv <name> <value>
```

wird die Umgebungsvariable <name> auf den Wert <value> gesetzt. Zu beachten ist, dass kein Gleichheitszeichen genutzt wird.

Beispielsweise weist

```
setenv FULLNAME "Test-Benutzer"
```

der Umgebungsvariable `FULLNAME` die Zeichenkette *Test-Benutzer* zu.  
Zum Abfragen von Umgebungsvariablen wird dieselbe Syntax genutzt wie für die Abfrage lokaler Variablen.

```
echo $FULLNAME
```

gibt demzufolge die Zeichenkette „Test-Benutzer“ aus, ebenso

```
echo ${FULLNAME}
```

Mit

```
unsetenv <name>
```

wird die Umgebungsvariable `<name>` gelöscht,

```
unsetenv FULLNAME
```

löscht die Umgebungsvariable `FULLNAME`.

### 3.6 Wichtige Variablen bei der Arbeit mit `csh`

Eine Reihe von lokalen Variablen und Umgebungsvariablen hat eine Sonderbedeutung, die die Arbeitsweise der Shell steuert bzw. beim Starten von Programmen zum Tragen kommt.

Insbesondere sind dies:

- *path*

Die lokale Variable *path* legt fest, in welchen Verzeichnissen in welcher Reihenfolge nach ausführbaren Programmen gesucht wird.

Die Verzeichnisse sind dabei durch Leerzeichen voneinander getrennt.

Die C-Shell berechnet bei ihrem Start eine Hash-Tabelle, in der alle erreichbaren Kommandonamen enthalten sind.

Diese Hash-Tabelle muss mit dem Kommando

```
rehash
```

neu angelegt werden, wenn Veränderungen am Pfad vorgenommen wurden, z.B. durch Installation neuer Software.

- *LD\_LIBRARY\_PATH*

Die Umgebungsvariable *LD\_LIBRARY\_PATH* wird nicht durch die Shell selbst genutzt sondern vom Programm „ld“, das ausführbare Programme startet.

Moderne UNIX-Systeme ermöglichen es, Programme dynamisch zu linken, diese Programme nutzen dann shared libraries.

Die Umgebungsvariable *LD\_LIBRARY\_PATH* legt fest, welche Verzeichnisse in welcher Reihenfolge nach shared libraries durchsucht werden.

Die Verzeichnisse sind dabei durch Doppelpunkte voneinander getrennt.

- *MANPATH*  
Die Umgebungsvariable *MANPATH* wird vom Programm „man“ genutzt. Sie enthält – durch Doppelpunkte getrennt – eine Liste von Verzeichnissen, in denen die Dateien der Online-Hilfe liegen.
- *prompt*  
Das Promptzeichen, das angezeigt wird. Voreingestellt ist hier meist die Raute „#“.
- *history*  
Diese Variable enthält die Anzahl der Kommandos, die für den History-Mechanismus (siehe Abschnitt 3.2.6 auf Seite 13) gespeichert werden sollen.
- *noclobber*  
Ist diese lokale Variable gesetzt, werden durch Ausgabeumlenkung keine bereits bestehenden Dateien überschrieben.  
Beispielsweise führt  

```
ls > ls.out
```

zu einer Fehlermeldung, wenn die Datei „ls.out“ bereits existiert.
- *ignoreeof*  
Unter UNIX wird das Zeichen „CTRL-d“ üblicherweise als Dateiende aufgefasst.  
Liest ein Shell-Prozess dieses Zeichen, geht er davon aus, dass keine weitere Eingabe erfolgt und beendet sich.  
Ist die Variable *ignoreeof* gesetzt, führt eine Eingabe von „CTRL-d“ am Terminal (wenn es beispielsweise versehentlich eingegeben wurde) nicht zum Beenden der Shell.  
Stattdessen ist eine explizite Eingabe der Kommandos „logout“ bzw. „exit“ erforderlich.
- *verbose*  
Ist die Variable *verbose* gesetzt, werden alle Eingaben protokolliert (d.h. auf die Standardeingabe ausgegeben), wie sie vorgefunden wurden.
- *echo*  
Ist die Variable *echo* gesetzt, werden alle Eingaben so protokolliert, wie sie abgearbeitet werden, d.h. nachdem alle Ersetzungen durchgeführt wurden.

## 3.7 source, exec und exit

Mit

```
source <Dateiname>
```

wird eine Datei mit Kommandos eingelesen und im aktuellen Prozess verarbeitet.

Mit

```
exec <Kommando>
```

wird das angegebene Kommando im Prozess des Kommandointerpreters gestartet.

Mit

```
exit [<Wert>]
```

wird der Kommandointerpreter verlassen, die Shell gibt den angegebenen Wert als Statuswert zurück.

## 3.8 Nutzer-Einstellungen

### 3.8.1 Start eines Shell-Prozesses

Wird eine Instanz von `csh` gestartet, wird zunächst im Home-Verzeichnis des Benutzers eine ausführbare Datei „`cshrc`“ gesucht.

Falls diese vorhanden ist, wird sie im aktuellen Prozess abgearbeitet, bevor mit der Verarbeitung der Eingabe begonnen wird.

Ist die C-Shell eine Login-Shell, wird zusätzlich nach einer Datei „`login`“ gesucht, falls diese vorhanden und ausführbar ist, wird sie ebenfalls abgearbeitet.

Beide Dateien werden gewöhnlich genutzt, um Variablen und Aliases zu setzen.

Die nachfolgenden Abschnitte enthalten Vorschläge für den Inhalt dieser beiden Dateien.

### 3.8.2 Standard-Zugriffsrechte auf Dateien

In „`login`“ sollte eine Zeile

```
umask 027
```

o.ä. enthalten sein, die festlegt, welche Zugriffsrechte auf neu angelegte Dateien gewährt werden.

Die Oktalzahl gibt an, welche Rechte *gesperrt* werden sollen.

Im Beispiel wird den Angehörigen der gleichen Gruppe das Schreiben verboten, allen anderen Nutzern wird jeglicher Zugriff verboten.

### 3.8.3 Prompt-Zeichen

Es erleichtert die Arbeit am Rechner, wenn man am Prompt-Zeichen sehen kann, in welchem Verzeichnis man sich befindet.

Arbeitet man in einem Workstation-Netz, ist es weiterhin nützlich, unter welchem Benutzernamen und auf welchem Rechner die aktuelle Shell arbeitet.

Da das Prompt-Zeichen bei jedem Verzeichniswechsel geändert werden muss, sind Aliase für die Befehle „`cd`“ und „`chdir`“ erforderlich. Die Definition muss also in „`cshrc`“ erfolgen (Aliase werden nicht an Subshells vererbt).

```
set hname = `hostname`
alias cd 'cd \!* ; set prompt = ($LOGNAME@$hname\:'pwd'\>\ )'
alias chdir 'chdir \!* ; set prompt = ($LOGNAME@$hname\:'pwd'\>\ )'
set prompt = ($LOGNAME@$hname\:'pwd'\>\ )'
```

### 3.8.4 Einstellungen für Vorsichtige

Für Anfänger ist es mitunter nützlich, das versehentliche Überschreiben von Dateien auszuschließen, ebenso wie das versehentliche Löschen bzw. Kopieren.

Entsprechende Aliases und lokale Variablen können in „`cshrc`“ definiert werden:

```
set noclobber
alias rm 'rm -i'
alias cp 'cp -i'
alias mv 'mv -i'
```

### 3.8.5 Standard-Editor festlegen

Einige Programme suchen in der Umgebung nach den Variablen EDITOR bzw. VISUAL, wenn Textdateien bearbeitet werden müssen.

Sind diese Variablen nicht vorhanden, wird ein Standard-Editor gestartet. Wenn man Pech hat, handelt es sich dabei um die nicht sonderlich nutzerfreundlichen <sup>2</sup> Editoren „ed“ oder „ex“.

Um den Standard-Editor festzulegen, setzt man die entsprechenden Umgebungsvariablen in „.login“.

```
setenv EDITOR vi
setenv VISUAL vi
```

Arbeitet man ausschließlich mit der graphischen Nutzeroberfläche, kann man auch den mit dieser Oberfläche mitgelieferten Editor benutzen, z.B. mit

```
setenv EDITOR dtpad
setenv VISUAL dtpad
```

### 3.8.6 Standard-Drucker festlegen

Wird beim Versenden von Druckaufträgen mit „lpr“ kein Drucker angegeben, wählt das Drucksystem einen Standard-Drucker aus. Die Vorgehensweise ist bei verschiedenen Drucksystemen unterschiedlich, meist werden aber Umgebungsvariablen wie z.B. LPDEST (System-V-Drucksystem), NGPRINTER (LPRng) oder PRINTER (BSD-Drucksystem) mit ausgewertet.

Wenn ein bestimmter Drucker bevorzugt verwendet werden soll und dieser Drucker nicht bereits im Drucksystem als Standard vereinbart ist, sollten die drei Umgebungsvariablen in „.login“ gesetzt werden.

```
setenv LPDEST    laserjet
setenv NGPRINTER laserjet
setenv PRINTER   laserjet
```

---

<sup>2</sup>meiner Meinung nach

## 3.9 Ein- und Ausgabeumlenkung

Die C-Shell stellt folgende Operatoren zur Umlenkung von Ein- und Ausgabe bereitgestellt:

- *cmd < Dateiname*

Das Kommando *cmd* erhält seine Standardeingabe aus der angegebenen Datei, z.B. zählt

```
wc < ls.out
```

die Zeichen, Wörter und Zeilen der Datei „ls.out“.

- *cmd << Endwort*

Das Programme *cmd* erhält seine Standardeingabe aus dem nachfolgenden Text, dabei wird die Eingabe durch eine Zeile beendet, die genau aus dem angegebenen Endwort besteht.

Im Beispiel

```
wc << ENDE
Dies ist ein Test.
Blah blah blah.
ENDE
```

erhält „wc“ den Text

```
Dies ist ein Test.
Blah blah blah.
```

als Eingabe. Im Text werden Variable ersetzt und Kommandoaufrufe der Form ‘cmd’ werden ausgeführt, falls die Endzeile keines der Zeichen Backslash, einfache oder doppelte Anführungszeichen oder Rückwärtsapostroph enthält.

- *cmd > Dateiname*

Die Standardausgabe des Kommandos *cmd* wird in die angegebene Datei umgeleitet. Falls die Variable *noclobber* gesetzt ist und die Datei existiert, wird eine Fehlermeldung ausgegeben und der Befehl nicht ausgeführt.

- *cmd >! Dateiname*

Wie „>“, jedoch entfällt der *noclobber*-Test.

- *cmd >& Dateiname*

*cmd >&! Dateiname*

Wie „>“ und „>!“, allerdings werden sowohl Standardausgabe als auch Standardfehlerausgabe in die Datei geschrieben.

- *cmd >> Dateiname*  
Die Standardausgabe des Kommandos wird an die Datei angehängen. Falls die Variable *noclobber* definiert ist und die Datei nicht existiert, wird mit einer Fehlermeldung abgebrochen.
- *cmd >>! Dateiname*  
Wie „>>“ jedoch ohne *noclobber*-Test.
- *cmd >>& Dateiname*  
Standardausgabe und Standardfehlerausgabe werden an die angegebene Datei angehängen, ein *noclobber*-Test erfolgt.
- *cmd >>&! Dateiname*  
Wie „>>&“, jedoch ohne *noclobber*-Test.
- *cmd1 | cmd2*  
Die Standardausgabe von *cmd1* wird in die Standardeingabe des Kommandos *cmd2* geleitet.
- *cmd1 |& cmd2*  
Standardausgabe und Standardfehlerausgabe des Kommandos *cmd1* bilden die Standardeingabe des Kommandos *cmd2*.

## 3.10 Programmieren mit der C-Shell

### 3.10.1 Ausdrücke und Operatoren

Einige der C-Shell-internen Kommandos verarbeiten Ausdrücke, in denen Operatoren ähnlich wie in der Programmiersprache C und mit denselben Vorrangregeln verwendet werden (typischerweise in *if*-, *while*- u.ä. Anweisungen)

Die Teile eines Ausdruckes werden durch Whitespaces (Leerzeichen und Tabulatoren) voneinander getrennt.

Der Wert eines jeden Ausdruckes ist ein String (eine Zeichenkette), der eine Dezimalzahl repräsentieren kann.

Hier die Operatoren, gruppiert nach Vorrang:

Operator	Bedeutung
( <i>Ausdruck</i> )	Gruppierung
~	Einerkomplement
!	Logische Negation
* / %	Multiplikation, Division, Divisionsrest - alle rechtsassoziativ
+ -	Addition, Subtraktion (rechtsassoziativ)
<< >>	Bitweise Links- bzw. Rechtsverschiebung
< > <= >=	kleiner als, größer als, kleiner-gleich, größer-gleich
== != =~ !~	gleich, ungleich, Pattern-Match, kein Pattern-Match (die Operatoren == und != führen String-Vergleiche aus, =~ und !~ prüfen, ob ein String auf der linken Seite einem Filename-Substitution-Pattern auf der rechten Seite entspricht bzw. nicht entspricht.)
-r <i>Dateiname</i>	Leserecht auf Datei vorhanden
-w <i>Dateiname</i>	Schreibrecht auf Datei vorhanden
-x <i>Dateiname</i>	Ausführungsrecht auf Datei vorhanden
-e <i>Dateiname</i>	Datei existiert
-o <i>Dateiname</i>	die Datei gehört dem Nutzer, unter dessen Account das Script läuft
-z <i>Dateiname</i>	die Datei ist leer
-f <i>Dateiname</i>	reguläre Datei
-d <i>Dateiname</i>	Verzeichnis
{ <i>Kommando</i> }	true, wenn Kommando erfolgreich war, ansonsten false

### 3.10.2 Binäre Verzweigung

```
if ( Ausdruck ) then
    Kommandos
endif
```

Der *Ausdruck* wird ausgewertet, ist er wahr, werden die *Kommandos* ausgeführt.

```
if ( Ausdruck ) then
    Kommandos1
else
    Kommandos2
endif
```

Der *Ausdruck* wird ausgewertet, ist er wahr, werden die *Kommandos1* ausgeführt, ansonsten die *Kommandos2*.

```
if ( Ausdruck1 ) then
    Kommandos1
else if ( Ausdruck2 ) then
    Kommandos2
else if ( Ausdruck3 ) then
    Kommandos3
...
else
    Kommandos_n
endif
```

Mehrfachverzweigung. Zu beachten ist, dass das „else“ sich auf das letzte vorangegangene „else if“ bezieht.

### 3.10.3 Schleifen

```
foreach name ( liste )  
  Kommandos  
end
```

Die Variable *name* erhält nacheinander alle Wörter aus der Liste *liste* zugewiesen. Für jedes Wort werden die angegebenen Kommandos ausgeführt.

```
while ( Ausdruck )  
  Kommandos  
end
```

Solange der *Ausdruck* wahr ist, wird ein neuer Schleifendurchlauf gestartet, in dem die *Kommandos* ausgeführt werden.

Zyklen können mit dem Kommando

```
break
```

abgebrochen werden, mit dem Kommando

```
continue
```

wird zum nächsten Test der Abbruchbedingung gesprungen.

### 3.10.4 Mehrfachverzweigung

```
switch ( Wort )
  case String1:
    Kommandos1
  breaksw
  ...
  case Stringn:
    Kommandosn
  breaksw
  default:
    Kommandosd
  breaksw
endsw
```

Je nachdem, ob das angegebene Wort (typischerweise meist in Form einer Variable angegeben) *String1...Stringn* ist, wird eine der Kommando-Folgen *Kommandos1...Kommandosn* ausgeführt.

Passt keiner der vorgegebenen Strings, wird die Kommando-Folge *Kommandosd* ausgeführt.

### **3.10.5 Sprungmarken**

*label:*

definiert eine Sprungmarke.

*goto label*

springt zur angegebenen Marke.

### 3.10.6 Beispiele

Die Datei „exists.csh“ schaut nach, ob die Datei „prog1.c“ existiert und gibt eine Meldung aus, falls sie existiert.

```
#!/bin/csh
if ( -f prog1.c ) then
    echo Die Datei prog1.c existiert.
endif
```

Die Datei „exists2.csh“ schaut nach, ob die Datei „prog1.c“ existiert und gibt eine Erfolgs- oder eine Fehlermeldung aus.

```
#!/bin/csh
if ( -f prog1.c ) then
    echo Die Datei prog1.c existiert.
else
    echo Die Datei prog1.c existiert nicht.
endif
```

Die Datei „whatday.csh“ gibt den Wochentag aus. Dazu wird der Befehl „date“ aufgerufen, aus dessen Ausgabe wird der Text bis zum ersten Leerzeichen entnommen („cut“) und der Variablen „i“ zugewiesen.

In Abhängigkeit von der Variablen „i“, die eine englische Abkürzung für den Wochentag enthält, wird die Variable „wochentag“ gesetzt und am Ende des Scripts ausgegeben.

```
#!/bin/csh
set i = `date | cut -f 1 -d ' '`
switch ( $i )
  case Mon:
    set wochentag = "Montag"
  breaksw
  case Tue:
    set wochentag = "Dienstag"
  breaksw
  case Wed:
    set wochentag = "Mittwoch"
  breaksw
  case Thu:
    set wochentag = "Donnerstag"
  breaksw
  case Fri:
    set wochentag = "Freitag"
  breaksw
  case Sat:
    set wochentag = "Samstag"
  breaksw
  case Sun:
    set wochentag = "Sonntag"
  breaksw
  default:
    set wochentag = "Unbekannt"
  breaksw
endsw
echo $wochentag
```

Das Script „bb.csh“ sucht in allen Dateien mit der Endung „.eps“ nach den Zeilen die mit „%%BoundingBox:“ beginnen. Diese Zeilen werden in einer gleichnamigen Datei mit der Endung „.bb“ abgespeichert.

Hierzu wird der Variablen „i“ nacheinander jeder Dateiname aus der Liste zugewiesen, die sich aus „\*.eps“ ergibt.

Für jede Zuweisung wird dann der Inhalt der Variablen „i“ der Variablen „j“ zugewiesen, dabei wird mit dem Modifizierer „:r“ die letzte Dateiendung entfernt. Der dabei entstehende Dateiname wird dann der Variablen „b“ zugewiesen, die Endung „.bb“ wird angehängt.

Nachdem die Namen von Quell- und Zielfeile bekannt sind, wird mit dem Programm „grep“ nach den Zeilen gesucht, die den Text „%%BoundingBox:“ am Zeilenanfang enthalten. Der zu durchsuchende Text gelangt durch Eingabeumleitung in die Standardeingabe des grep-Befehles, die Ausgabe wird in die gewünschte Zielfeile umgeleitet.

```
#!/bin/csh
foreach i (*.eps)
  set j = $i:r
  set b = ${j}.bb
  grep '^%%BoundingBox:' < $i > $b
end
```

# 4 Die Bourne-Shell

## 4.1 Verknüpfung von Kommandos

Kommandos können auf folgende Weise verknüpft werden:

- *cmd1 ; cmd2*  
Sequentielle Ausführung, d.h. Programm *cmd2* wird gestartet, nachdem *cmd1* beendet wurde.
- *cmd &*  
Ausführung „im Hintergrund“ (asynchrone Ausführung).  
Das Kommando wird in einem Hintergrundprozess ausgeführt. Die Prozessnummer des Hintergrundes wird ausgegeben, die Arbeit am Terminal kann fortgesetzt werden, während der Prozess noch läuft.
- *cmd1 && cmd2*  
Nur wenn *cmd1* erfolgreich beendet wurde, wird *cmd2* ausgeführt.
- *cmd1 || cmd2*  
Nur wenn *cmd1* nicht erfolgreich beendet wurde, wird *cmd2* ausgeführt.
- *cmd1 | cmd2*  
Die Kommandos bilden eine Pipe, d.h. die Standardausgabe von *cmd1* wird vom Kommando *cmd2* als Standardeingabe verarbeitet.  
Der exit-Wert von *cmd2* (bzw. dem letzten Kommando der Pipe) ergibt den exit-Wert der Verknüpfung.
- *( cmd )*  
führt das Kommando *cmd* in einer eigenen Instanz des Kommandointerpreters *sh* aus.
- *{ cmd; }*  
führt das Kommando *cmd* als Unterprogramm des aktuellen Kommandointerpreters aus ohne einen neuen Prozess zu erzeugen. Die geschweiften Klammern sind separate Worte, daher müssen Leerzeichen zwischen Klammern und Kommando stehen.
- *name() { cmd; }*  
definiert eine Shell-Funktion. Diese kann über die angegebene Bezeichnung gestartet werden.  
Folgen dem Aufruf Argumente, werden diese als \$1, \$2... bereitgestellt.

## **4.2 Wildcards, Variablen und Ersetzungen**

### **4.2.1 Übersicht**

Wildcards, Variablen und Ersetzungen sind Mechanismen, bei denen Text aus der Eingabezeile durch anderen Text ersetzt wird.

Der Ersetzungstext kann dabei aus vorher definierten Variablen, aus der Inspektion von Verzeichnisinhalten oder aber aus anderen Kommandos stammen.

Diese Mechanismen werden meist genutzt, um möglichst wenig Text eingeben zu müssen bzw. um Tippfehlern vorzubeugen.

Die Ersetzungen finden vor der eigentlichen Verarbeitung der Zeilen statt.

## 4.2.2 Wildcards

Wildcards sind Sonderzeichen in Kommandos, die für beliebige Zeichenketten in Dateinamen stehen können.

Folgende Wildcard-Konstruktionen können für Dateinamen verwendet werden:

Wildcard	Bedeutung
*	beliebige 0..14 Zeichen lange Zeichenkette
?	ein beliebiges Zeichen
[abcdefg123]	eines der Zeichen a, b, c, d, e, f, g, 1, 2 oder 3
[a-c]	eines der Zeichen a, b oder c
[!c]	ein beliebiges Zeichen ungleich c

## 4.2.3 Lokale Variablen

In einer Shell können Variable mit Text belegt werden, der dann später durch Bezug auf die Variable wieder abgerufen werden kann.

Lokale Variable gelten nur innerhalb des aktuellen Prozesses und werden nicht an Subprozesse weitergegeben, die von der Shell gestartet wurden.

Zum Setzen von Variablen wird der Konstrukt

```
<name>=<value>
```

genutzt. Hierbei dürfen keine Leerzeichen vor und nach dem Gleichheitszeichen stehen. Anstelle von <name> muss der Name der zu setzenden Variable eingesetzt werden und anstelle von <value> der Text. Im Beispiel

```
dir="/etc"  
ls $dir
```

wird der Variablen *dir* der Wert „/etc“ zugewiesen.

Im anschließenden ls-Befehl wird die Variable benutzt, dabei wird durch die Shell automatisch

```
$name  
${name}
```

durch den Wert der Variablen *name* ersetzt.

Beim Abfragen/Ersetzen von Variablen sind folgende Sonderformen möglich:

- `${name:-value}`  
liefert den Wert der Variablen, wenn diese definiert ist und nicht leer ist. Andernfalls wird der Text *value* eingesetzt.
- `${name:=value}`  
wie oben, jedoch wird *name* gleich mit zu *value* definiert, falls die Variable undefiniert oder leer war.

- $\${name:?text}$   
 $\${name:?}$   
liefert den Wert der Variablen, wenn diese definiert und nicht leer ist. Andernfalls wird die Bourne-Shell mit der Ausschrift *text* bzw. mit einer Fehlermeldung verlassen.
- $\${name:+value}$   
substituiert den Text *value*, falls *name* definiert und nicht leer ist. Andernfalls wird nichts substituiert.

Die o.g. Sonderformen existieren in einer weiteren Variante ohne den Doppelpunkt. Bei diesen Varianten entfällt jeweils der Test, ob die Variable leer ist.

#### **4.2.4 Ersetzungen**

Kommt in der Eingabe

- *'cmd'*

vor, wird es durch die Standardausgabe des Kommandos *cmd* ersetzt.

## 4.3 Vordefinierte Variablen

Wenn eine Shell läuft, sind eine Reihe von Variablen von Beginn an definiert, u.a.:

- \$\$  
Prozessnummer der aktuellen Shell
- \$?  
Ergebniswert des letzten Kommandos
- \$!  
Prozessnummer des zuletzt gestarteten Hintergrundprozesses

## 4.4 Aufruf-Parameter und Variablen in Scripten

Shell-Scripte können wie ausführbare Programme Parameter erhalten. Diese sind im Script als „\$1“, „\$2“... ansprechbar.

Mit „\$\*“ sind alle Parameter gesammelt abrufbar.

„\$#“ gibt die Anzahl der Parameter als Dezimalzahl an.

## 4.5 Umgebungsvariablen

Die bereits besprochenen lokalen Variablen sind nur innerhalb des Prozesses gültig, der die Shell ausführt.

Wird durch die Shell ein neuer Prozess erzeugt – beispielsweise beim Start eines Programmes – kann dieser neue Prozess nicht auf die lokalen Variablen der Shell zugreifen.

Im Unterschied dazu gibt es die sogenannte Umgebung. Dabei handelt es sich um eine Sammlung von Variablen, die Unterprozessen (und dann wiederum auch deren Unterprozessen) mitgegeben wird.

Mit

```
export <name>
```

wird die lokale Variable <name> in die Umgebung eingefügt. Beispielsweise weist

```
FULLNAME="Test-Benutzer"  
export FULLNAME
```

oder

```
export FULLNAME="Text-Benutzer"
```

der Umgebungsvariable FULLNAME die Zeichenkette *Test-Benutzer* zu.

Zum Abfragen von Umgebungsvariablen wird dieselbe Syntax genutzt wie für die Abfrage lokaler Variablen.

```
echo $FULLNAME
```

gibt demzufolge die Zeichenkette „Test-Benutzer“ aus, ebenso

```
echo ${FULLNAME}
```

## 4.6 Wichtige Variablen bei der Arbeit mit sh

Eine Reihe von lokalen Variablen und Umgebungsvariablen hat eine Sonderbedeutung, die die Arbeitsweise der Shell steuert bzw. beim Starten von Programmen zum Tragen kommt.

Insbesondere sind dies:

- *PATH*  
Die Umgebungsvariable *PATH* legt fest, in welchen Verzeichnissen in welcher Reihenfolge nach ausführbaren Programmen gesucht wird.  
Die Verzeichnisse sind dabei durch Doppelpunkte voneinander getrennt.
- *LD\_LIBRARY\_PATH*  
Die Umgebungsvariable *LD\_LIBRARY\_PATH* wird nicht durch die Shell selbst genutzt sondern vom Programm „ld“, das ausführbare Programme startet.  
Moderne UNIX-Systeme ermöglichen es, Programme dynamisch zu linken, diese Programme nutzen dann shared libraries.  
Die Umgebungsvariable *LD\_LIBRARY\_PATH* legt fest, welche Verzeichnisse in welcher Reihenfolge nach shared libraries durchsucht werden.  
Die Verzeichnisse sind dabei durch Doppelpunkte voneinander getrennt.
- *MANPATH*  
Die Umgebungsvariable *MANPATH* wird vom Programm „man“ genutzt. Sie enthält – durch Doppelpunkte getrennt – eine Liste von Verzeichnissen, in denen die Dateien der Online-Hilfe liegen.
- *PS1*  
Das primäre Promptzeichen. Voreingestellt ist hier meist das Dollarzeichen.
- *PS2*  
Das sekundäre Promptzeichen (wird angezeigt, wenn zu einer Eingabezeile weitere Eingabe erwartet wird). Voreingestellt ist meist „>“.
- *IFS*  
Der Feldseparator. Diese Umgebungsvariable legt fest, durch welche Zeichen der Kommandoname und die einzelnen Parameter voneinander getrennt werden. Voreingestellt sind hier Leerzeichen und Tabulator.
- -v  
Mit

```
set -v
```

wird erreicht, dass die Shell alle Eingaben protokolliert, wie sie vorgefunden wurden.

- `-x`  
Mit

```
set -x
```

werden alle Eingaben so protokolliert, wie sie verarbeitet werden, d.h. nachdem alle Ersetzungen... vorgenommen wurden.

## 4.7 ., exec und exit

Mit

```
. <Dateiname>
```

wird eine Datei mit Kommandos eingelesen und im aktuellen Prozess verarbeitet.

Mit

```
exec <Kommando>
```

wird das angegebene Kommando im Prozess des Kommandointerpreters gestartet.

Mit

```
exit [<Wert>]
```

wird der Kommandointerpreter verlassen, die Shell gibt den angegebenen Wert als Statuswert zurück.

## 4.8 Nutzer-Einstellungen

Wird sh als Login-Shell gestartet, wird im Home-Verzeichnis nach einer ausführbaren Datei „.profile“ gesucht.

Ist diese vorhanden, wird sie im aktuellen Prozess ausgeführt, bevor mit der Verarbeitung der Eingabe begonnen wird. Weiterhin sollte in „.profile“ eine Zeile

```
umask 027
```

o.ä. enthalten sein, die festlegt, welche Zugriffsrechte auf neu angelegte Dateien gewährt werden.

Die Oktalzahl gibt an, welche Rechte *gesperrt* werden sollen.

## 4.9 Ein- und Ausgabeumlenkung

Die Bourne-Shell stellt folgende Ein- und Ausgabeumlenkungen zur Verfügung:

- *cmd <Dateiname*  
Das Kommando erhält seine Standardeingabe aus der angegebenen Datei.
- *cmd >Dateiname*  
Die Standardausgabe des Kommandos erfolgt in die angegebene Datei.
- *cmd >>Dateiname*  
Die Standardausgabe des Kommandos wird an die Datei angehängt. Falls erforderlich wird die Datei neu angelegt.
- *cmd <<Endwort*  
Der nachfolgende Text bis zu (ausschließlich) einer Zeile, die das Wort *Endwort* enthält, wird dem Kommando als Standardeingabe übergeben. Im Text finden Ersetzungen und Variablensubstitution statt.
- *cmd <<-Endwort*  
Wie „<<“, im Text und der Endzeile werden führende Tabulator-Zeichen ignoriert.
- *cmd1 | cmd2*  
Die Standardausgabe von *cmd1* wird in die Standardeingabe des Kommandos *cmd2* geleitet.
- *cmd <&m*  
*cmd >&m*  
Der Filedeskriptor *m* wird dupliziert und für die Umlenkung genutzt.

Zusätzlich sind die Konstruktionen

*n*Umleitungsoperator*Dateiname*

*n*Umleitungsoperator*&m*

vorhanden. Dabei wird der Deskriptor *n* auf die angegebene Datei bzw. den angegebenen Deskriptor *m* umgelenkt.

Mit

*cmd 2>&1*

wird die Standardfehlerausgabe auf die Standardausgabe umgelenkt, hierbei treten aber möglicherweise durch die Pufferung der Daten vor der Ausgabe Probleme auf.

## 4.10 Programmieren mit der Bourne-Shell

### 4.10.1 Ausdrücke und Operatoren

Wird im folgenden Text von „Bedingungen“ (*condition*) gesprochen, stehen folgende Varianten zur Verfügung:

- Ein Kommando wird ausgeführt. Die Bedingung ist erfüllt, wenn das Kommando mit dem exit-Wert „0“ terminiert.
- Das Programm

`test expression`

wird verwendet, um einen booleschen Ausdruck zu erzeugen.

Eine andere Schreibweise hierfür ist

`[ expression ]`

Als Test-Ausdrücke (*expression*) sind u.a. folgende Konstrukte zulässig:

- `-s Dateiname`  
Die Datei existiert und hat eine Länge größer als 0.
- `-r Dateiname`  
Die Datei ist lesbar.
- `-w Dateiname`  
Die Datei ist schreibbar.
- `-x Dateiname`  
Die Datei kann ausgeführt werden.
- `-f Dateiname`  
Es handelt sich um eine reguläre Datei.
- `-d Dateiname`  
Es handelt sich um ein Verzeichnis.
- `-z String`  
Die Länge des Strings ist 0.
- `-n String`  
Die Länge des Strings ist ungleich 0.
- `String1 = String2`  
Die Strings sind gleich.
- `String1 != String2`  
Die Strings sind ungleich.
- `String`  
Die Zeichenkette ist nicht leer.

- *Zahl1 -op Zahl2*  
Die ganzen Zahlen werden algebraisch miteinander verglichen, zulässige Operatoren *op* sind eq (Gleichheit), ne (Ungleichheit), gt (größer als), ge (größer als oder gleich), lt (kleiner als) und le (kleiner als oder gleich).
- *! Bool*  
Negation.
- *Bool1 -a Bool2*  
Boolesche und-Verknüpfung der beiden Ausdrücke.
- *Bool1 -o Bool2*  
Boolesche oder-Verknüpfung der beiden Ausdrücke.
- *( Bool )*  
Klammerung.
- *-c Dateiname*  
Es handelt sich um eine zeichenorientierte Gerätedatei.
- *-b Dateiname*  
Es handelt sich um eine blockorientierte Gerätedatei.
- *-p Dateiname*  
Es handelt sich um eine Pipe.
- *-u Dateiname*  
Das SUID-Bit der Datei ist gesetzt.
- *-g Dateiname*  
Das SGID-Bit der Datei ist gesetzt.
- *-t Filedeskriptor*  
Die Datei mit der angegebenen Filedeskriptor-Nummer ist einem Terminal zugeordnet.

## 4.10.2 Binäre Verzweigung

```
if condition
then
    commands
fi
```

führt die Kommandos der Liste *commands* aus, wenn die Bedingung *condition* erfüllt ist.

```
if condition
then
    commands1
else
    commands2
fi
```

führt die Kommandos der Liste *commands1* aus, wenn die Bedingung *condition* erfüllt ist, ansonsten die Kommandos der Liste *commands2*.

```
if condition1
then
    commands1
elif condition2
then
    commands2
elif condition3
then
    commands3
...
else
    othercommands
fi
```

testet, ob die Bedingung *condition1* erfüllt ist und führt in diesem Fall die Kommandos aus Liste *commands1* aus.

Andernfalls wird getestet, ob die Bedingung *condition2* erfüllt ist, in diesem Fall werden die Kommandos aus Liste *commands2* ausgeführt.

Andernfalls wird ggf. mit dem Testen weiterer Bedingungen fortgefahren.

Ist keine der Bedingungen erfüllt, werden die Kommandos aus der Liste *othercommands* im „else“-Zweig ausgeführt.

### 4.10.3 Schleifen

```
for var in wordlist
do
    commands
done
```

weist der Variable *var* nacheinander alle Worte der Liste *wordlist* zu und führt mit jeder Zuweisung die Kommandos aus der Liste *commands* aus.

Wird der Teil

```
in wordlist
```

weggelassen, werden die Positionsparameter nacheinander in die Variable eingesetzt.

```
while condition
do
    commands
done
```

testet, ob die Bedingung *condition* erfüllt ist.

Falls ja, werden die Kommandos der Liste *commands* ausgeführt, anschließend wird erneut zum Testen der Bedingung zurückgegangen. . .

Die Anweisungsliste wird ausgeführt, *solange* die Bedingung erfüllt ist.

```
until condition
do
    commands
done
```

testet, ob die Bedingung *condition* erfüllt ist.

Ist die Bedingung *nicht* erfüllt, werden die Kommandos der Liste *commands* ausgeführt, anschließend wird erneut zum Testen der Bedingung zurückgegangen. . .

Die Anweisungsliste wird ausgeführt, *bis* die Bedingung erfüllt ist.

#### 4.10.4 Mehrfachverzweigung

```
case string in
  pattern1 )
    commands1
  ;;
  pattern2 )
    commands2
  ;;
  ...
esac
```

vergleicht die Zeichenkette *string* mit den Mustern *pattern1*, *pattern2*... und führt die Liste mit Kommandos aus, die dem passenden Testmuster folgt. Üblicherweise stammt der Text *string* aus einer Variablen.

Für die Muster gelten die gleichen Wildcards (siehe Abschnitt 4.2.2 auf Seite 34) wie beim Erzeugen von Dateinamen.

## 4.10.5 Beispiele

Die Datei „exists.sh“ überprüft, ob die Datei „prog1.c“ existiert, falls ja wird eine Erfolgsmeldung ausgegeben.

```
#!/bin/sh
if [ -f prog1.c ]
then
    echo Die Datei prog1.c existiert.
fi
```

Die Datei „exists2.sh“ überprüft, ob die Datei „prog1.c“ existiert, es wird eine Erfolgs- oder eine Fehlermeldung ausgegeben.

```
#!/bin/sh
if [ -f prog1.c ]
then
    echo Die Datei prog1.c existiert.
else
    echo Die Datei prog1.c existiert nicht.
fi
```

Die Datei „whatday.sh“ gibt den Wochentag aus. Dazu wird der Befehl „date“ aufgerufen, aus dessen Ausgabe wird der Text bis zum ersten Leerzeichen entnommen („cut“) und der Variablen „i“ zugewiesen.

In Abhängigkeit von der Variablen „i“, die eine englische Abkürzung für den Wochentag enthält, wird die Variable „wochentag“ gesetzt und am Ende des Scripts ausgegeben.

```
#!/bin/sh
i='date | cut -f 1 -d ' ' '
case $i in
  Mon)
    wochentag="Montag"
    ;;
  Tue)
    wochentag="Dienstag"
    ;;
  Wed)
    wochentag="Mittwoch"
    ;;
  Thu)
    wochentag="Donnerstag"
    ;;
  Fri)
    wochentag="Freitag"
    ;;
  Sat)
    wochentag="Samstag"
    ;;
  Sun)
    wochentag="Sonntag"
    ;;
  *)
    wochentag="Unbekannt"
    ;;
esac
echo $wochentag
```

Weitere Beispiele für die Verwendung der Mehrfachverzweigung finden Sie in den System-Start-Scripten (in /etc/rc.d/rc2.d bzw /etc/rc2.d).

In diesen Scripten werden jeweils die Aufrufparameter geprüft, „start“ bewirkt den Start bestimmter Services, „stop“ das Herunterfahren von Services.

Das Script „conv2pdf.sh“ wandelt die Dateien mit der Endung „.eps“ in PDF-Dateien um.

Hierzu wird der Variablen „i“ nacheinander jeder Dateiname zugewiesen, der sich aus der Wildcard-Ersetzung von „\*.eps“ ergibt.

Für jede dieser Zuweisungen wird dann das Programm „epstopdf“ aufgerufen, das eine Datei mit der Endung „.eps“ in eine Datei gleichen Namens mit der Endung „.pdf“ umwandelt (Voraussetzung ist natürlich, dass das Programm „epstopdf“ installiert ist).

```
#!/bin/sh
for i in *.eps
do
    epstopdf $i
done
```

# Index

Addition, 23  
Alias, 10  
Argumente, 5  
asynchron, 6  
Aufruf-Parameter, 15, 35  
  
break, 25  
  
case, 26, 45  
Clones, 4  
continue, 25  
  
Division, 23  
Divisionsrest, 23  
  
Einerkomplement, 23  
elif, 24, 43  
else, 24, 43  
Ersetzung, 11, 34  
  
Fehlercode, 6  
for, 44  
foreach, 25  
  
goto, 27  
  
History, 12  
  
if, 24, 43  
  
Kommando, 5  
Kommandointerpreter, 4  
Kommandoname, 5  
Korn-Shell, 4  
ksh, 4  
  
Modifikator, 13  
Multiplikation, 23  
  
Negation  
    logisch, 23  
Nutzer-Einstellungen, 19, 39  
  
Optionen, 5

Parameter, 5  
Prompt-Zeichen, 19  
Prozeß, 5  
  
Shell, 4  
Status, 6  
Subtraktion, 23  
switch, 26  
  
tosh, 4  
Test  
    Ausführungsrecht, 23, 41  
Datei  
    Eigentümer, 23  
    Gerät, 42  
    Größe, 41  
    leer, 23  
    Pipe, 42  
    regulär, 23, 41  
    SGID, 42  
    SUID, 42  
    Terminal, 42  
    Verzeichnis, 41  
Existenz, 23  
Leserecht, 23, 41  
Schreibrecht, 23, 41  
String  
    Gleichheit, 41  
    Länge, 41  
Vergleich  
    algebraisch, 42  
Verknüpfung, 42  
Verzeichnis, 23  
test, 41  
Turbo-C-Shell, 4  
  
umask, 19, 39  
Umgebung, 15, 35  
Umlenkung, 21, 40  
until, 44  
  
Variable, 7, 33

vordefiniert, 15, 35  
Vergleich, 23  
Verschiebung  
    bitweise, 23  
  
while, 25, 44  
Wildcard, 7, 33