Unix-Shell - Kompendium

Dr. Erwin Hoffmann (FH Frankfurt/Main)

1. Was ist die Unix Shell ?

Die Unix Shell ist
a) eine Umgebung, von der Kommandos aufgerufen werden,
b) ein Interpreter, der die einzelnen Statements zeilenweise und Case-sensitive in der bestehenden Umgebung auswertet,
c) eine spezifische Programmier-Plattform ("Unix Skripte"), bei der man a) und b) berücksichtigen muss.

Es gibt mehrere Shells mit unterschiedlichem Funktionsumfang und Syntax. Wir betrachten hier die folgenden
- **sh** oder **bash** (*Bourne Again Shell*; login, interaktives Arbeiten)
- **ksh** bzw. Korn-Shell (Shell-Skripte)
- **csh** bzw. C-Shell (Shell-Skripte)

2. Unix-Kommandos

a) Jedes unter Unix ausführbare Programm ist ein Kommando.
b) Unix (aber z.T. auch Windows) verfügt über einen Satz von Default-Kommandos (POSIX-Standard) zum Bewegen im Verzeichnisbaum, zur Erzeugung/Löschen/Verschieben/Umbenennen von Dateien und Verzeichnis und viele andere.
c) Kommandos besitzen in der Regel Parameter und Optionen, die als Argumente an das Kommando übergeben werden.
d) Parameter sind Kommando-spezifisch; z.B. **cp** *datei1 datei2* (lautet das Kommando '**cp**' mindestes zwei Argumente, nämlich die Angabe der Quelldatei, die es zu kopieren gilt und den Namen der resultierenden Datei.
e) Optionen sind spezielle Argumente; diese werden in der Regel mit einem '-' (Strich) Zeichen eingeleitet (teilweise ist auch ein führendes '+' möglich). Optionen können einzelnen bezeichnet der (z.B. '-v -p') oder zusammengefasst werden ('-vp').

Beispiel:

> **cp** -v -p *datei1 datei2* und
> **cp** -vp *datei1 datei2* bzw.
> **cp** -pv *datei1 datei2*
> In allen Fällen wird das Kommando **cp** gleich interpretiert.

f) Diese Art der Auswertung von Argumenten wird 'getopt' Stil genannt. Es besteht die Tendenz, die Benennung der Optionen möglichst für unterschiedliche Kommandos immer gleich zu halten (Beispiel: -v für *Verbose*, -h für *Hilfe*, -r für *Rekursiv*).
g) Fehlerfrei abgeschlossene Kommandos (was aber nicht unbedingt inhaltlich erfolgreich bedeutet) werden dem Aufrufer mit dem Exit- bzw. Return-Code 0 mitgeteilt.

Teil 1: Die Shell als Parser

3. Interpretation der Statements

a) Ein Shell-Statement endet mit dem Zeilenende oder einem Semikolon ';'.
b) Die Shell verarbeitet die Statements 'stapelweise' also Zeile für Zeile (was nicht heisst, dass keine Verzweigungen auftreten dürfen).
c) Eine Zeile, die mit einem '#' beginnt, wird von der Ausführung ignoriert (Kommentarzeile).
d) Mehrere Statements können in einen Zeile gepackt werden, indem sie durch ein Semikolon in eine Zeile geschrieben werden.

Beispiel:

```
cp -v datei1 datei2; ls -la datei1 datei2
```

e) Lange Statements lassen sich auf mehrere Zeilen umbrechen, indem der *Backslash* '\' als letztes Zeichen in der 'physikalischen' Zeile geschrieben wird.

Beispiel:

```
cp -vp SEHRLANGERDATEINAME \<LF>
    NOCHVIELLÄNGERERDATEINAME<LF>
```

Hierbei soll <LF> (Linefeed) das Ende der Zeile darstellen.
f) In der Shell ist notwendige Zeichenketten-Separator für die einzelnen Statements (als zwischen Kommando und seinen Argumenten) in der Regel der 'White Space'. White Spaces sind Leerzeichen (' ') oder Tabulatoren; dies wird 'Input Field Separator' (IFS) genannt, der aber (gezielt) verändert werden kann.
g) Mehrere Kommandos können mittels des Klammer-Operators '( )' logisch zusammengefasst werden. Der Return-Wert entspricht dem Gesamtausdruck oder aber dem letzten unabhängig ausgeführten Kommandos.

4. Meta- und Sonderzeichen der Shell

a) Die Shell wertet einige Zeichen 'besonders' aus: Sonderzeichen.

! ? " ' $ & / \ ( ) { } [ ] ` | < > = ~ + # , ; : . + - % @ ^

b) Sonderzeichen sind entweder *Operatoren* (mit fixer Bedeutung) oder aber *Metazeichen* (mit Kontext-abhängiger Bedeutung; *Substitutionszeichen*).
c) Das wichtigste Metazeichen haben wir bereits kennengelernt: Das 'Leerzeichen' (" ").
d) Statements, die Leerzeichen beinhalten (z.B. Dateinamen) und ALS SOLCHE verarbeitet werden sollen, müssen daher besonders gekennzeichnet werden, was mittels Flucht-Symbols (*Escape-Zeichen*) erfolgt: Dem vorausgestellten *Backslash* '\'.

Beispiel:

```
touch 123 456     => erzeugt die Dateien '123' und '456'
touch 123\ 456    => erzeugt Datei '123 456' (einschliesslich Leerzeichen)
```

Mehrere einzufügende Leerzeichen können mit '\ \ \ ' kenntlich gemacht werden.
e) Die Shell versteht und interpretiert (zumindest) einen Subset sogenannter Regulärer Ausdrücke (*Regular Expressions*; RegEx).
f) Die bekanntesten Metazeichen der Shell sind:

Zeichen	Bedeutung	Beispiel
\	Fluchtzeichen; Metazeichen verlieren ihre Bedeutung	\?
?	Platzhalter für genau ein Zeichen	123?456
*	Platzhalter für *beliebig* viele Zeichen; einschliesslich *kein* Zeichen	12*56
[und]	Deklaration einer Zeichenklasse; das '-' Zeichen gibt einen *Gültigkeitsbereich* an; das '!' kehrt die Bedeutung um	[0-9]*, *[!0-9]

5. Quotierung

a) Single Quotes: Will man die 'literarische' Bedeutung eines Statements erzwingen, ist der Ausdruck in einfache Hochkomma zu stellen: 'ausdruck'. Hierdurch werden keine Metazeichen ausgewertet.
b) Double Quotes: Wird hingegen ein Ausdruck in doppelte Hochkomma ("Gänsefüsschen") gestellt, werden die Metazeichen innerhalb dieses Statements VOR der weiteren Verarbeitung ausgewertet.
c) Backslash: Will man innerhalb eines Ausdrucks in "Double Quotes" einige Metazeichen nicht expandieren, ist diesen Zeichen der Backslash vorzustellen: "\? Dies ist ein Ausrufezeichen".

Achtung! Auf der deutschen Standard-Tastatur finden sich noch die *Apostrophen- Zeichen*, die für Quotierung nicht geeignet sind:

- Accent aigu: ´ -- neben der Backspace-Taste (<=)
- Accent grave: ` -- nebem der Backspace-Taste (<=) + Umschalt-Taste (^)

Teil 2: Spezielle Operatoren der Shell

6. Kommando Operatoren

Die Shell kennt drei Formen der Kommando-Ausführung:

Kommando	Bedeutung	Beispiel
cmd *args*	Das ausführbare Programm wird aufgerufen und in der aktuellen Shell ausgeführt; die Argumente werden hier expandiert	**cat** *datei*
`cmd args`	Das ausführbare Programm (eingeschlossen in 'accent grave') wird in der aktuellen Shell ausgeführt; die Argumente explizit übergeben	**echo** "Gefundene Dateien: `**ls** -l`"
$(cmd args)	Das ausführbare Prgramm (eingeschlossen in Klammern, wobei die linke Klammer mit einem $ vorgestellt ist) wird (einschliesslich der angegebenen Argumente) in einer Subshell ausgeführt	$(**wc** -w *datei1*)

7. Datenstrom Operatoren

Nach Aufruf hat die Shell immer mindestens drei Datenströme geöffnet:

a) STDIN: ***&0*** (Standard-Eingabe: Tastatur)
b) STDOUT: ***&1*** (Standard-Ausgabe: Bildschirm)

c) STDERR: ***&2*** (Fehler-Ausgabe: Bildschirm)

&n stellt hierbei einen Datei-Deskriptor dar. Datei-Deskriptoren lassen sich duplizieren, schliessen, in-einander sowie in Dateien umleiten.

Beispiele:

Operation	**Bedeutung**	**Beispiel**
n > DATEI	Ausgaben des Datenstroms *n* wird in DATEI geschrieben	2>/dev/null
n >> DATEI	Ausgabe des Datenstroms *n* wird an DATEI angehangen	1>>DATEI
n < DATEI	Lese von DATEI und überführe in Datenstrom *n*	for ... ; do; ...; done <DATEI
n < &*m*	Dupliziere Eingangs-Datenstrom *n* von *m*	1<&3
n > &*m*	Dupliziere Ausgangs-Datenstrom *n* nach *m*	1>&2
n <> DATEI	Lese und Schreibe von DATEI	1<>DATEI
n < &-	Schliesse Eingangs-Datenstrom *n*	3<&-
n > &-	Schliesse Ausgangs-Datenstrom *n*	2>&-
1 << DEL; ... ; DEL	'Here' Dokument; DEL = Delimitor (eindeutiges Zeichen)	Einlesen von Statements in der Verarbeitung

Im Gegensatz zu sonstigen Shell-Verhalten brauchen Datenstrom-Operatoren auf der Kommandozeile KEIN zuätzliches Leerzeichen.
Die Datenströme lassen sich per Kommando, oder aber für alle nachfolgenden Kommandos in der Shell mittels des **exec** Befehls umlenken:

i) Per Kommando (Beispiel): **cmd** 2>&1 bzw. **cmd1** 2>&1 | **cmd2** 2>&1 (in einer Pipe)
ii) Für alle folgenden: **exec** 2>&1

Im praktischen Einsatz findet besonders die Umleitung der Fehlermeldungen von Kommandos (ausgegeben auf STDERR) häufigen Einsatz.

Beispiel:

```
find /usr -name "*.h" 2>/dev/null
```

Es ist ferner zu beachten, dass beim Umlenken des Datenstroms in eine Datei für den angegebenen Name *keine Expandierung* vorgenommen wird:

```
echo "Kein gute Dateiname" > *.txt
```

8. Unix Pipelines

Das Zeichen '|' stellt den Unix *Pipe-Operator* dar. Hierdurch lassen sich in einer Pipeline der STDOUT des einen Kommandos mit dem STDIN des folgenden Kommandos verketten. Das Pipe-Zeichen wird von Shell-Skripten und Kommandos i.d.R. als Operator interpretiert und benötigt [vgl. 6.] keine zusätzlichen Leerzeichen.

Beispiel:

```
cat datei1.txt datei2.txt | grep 'abc' | tr ':' ';' > ersetzt.txt
```

9. Kommando-Verkettung

Die Shell kennt die folgenden Formen der Kommando-Verkettungen:

a) sequentielle Abarbeitung
b) Pipelining
c) bedingte oder logische Abarbeitung
d) Kommando-Substitution
e) Verschachtelte Ausführung (KSH)

sowie ergänzend

f) Ausführung im Hintergrund

Wir beachten, dass diese Formen mit den Anweisungen für die Datenstrom-Operatoren gekoppelt werden können:

Form	Befehlsfolge	Bedeutung	Beispiel
a)	cmd1 ; cmd2	Führe erst Befehl **cmd1** und dann **cmd2** aus	**cd** VERZEICHNIS; **ls** -la
a)	{ cmd1; cmd2 }	Führe Kommandos **cmd1** und **cmd2** in aktueller Shell aus	{ ls *.c; ls *.h }
a)	(cmd1; cmd2)	Führe Kommandos **cmd1** und **cmd2** gemeinsam in einer Sub-Shell aus	**(date; who) >** loginfo
b)	cmd1 | cmd2	Führe Befehl **cmd1** und überführe STDOUT als STDIN für Befehl **cmd2**	**cat** DATEI **|** **wc** -w
c)	cmd1 && cmd2	Führe zunächst Befehl **cmd1** aus; Befehl **cmd2** aber nur, falls **cmd1** mit Return-Code 0 abschliesst	**grep** 'fehler' DATEI **&& rm** DATEI
c)	cmd1 || cmd2	Führe zunächst **cmd1** aus und **cmd2** nur dann, falls **cmd1** sich mit einem Fehler beendet (Return-Code ungleich 0)	**ls** *.c 2>/dev/null **||** **echo** "Keine C-Dateien gefunden"
d)	cmd1 `cmd2`	Nutze das Resultat von **cmd2** als Argument für **cmd1**	**vi** `**grep** -l ifdef *.c`
e)	cmd1 $(cmd2)	Führe zunächst **cmd2** aus und Übergebe das Resultat an **cmd1**	**tar** -cv $(ls -1)
f)	cmd &	Führe Kommando **cmd** im Hintergrund aus	**grep** -i 'fehler' *.c &

Bemerkungen:

- Bei den Formen d) und e) beachten wir, dass die Shell Variablen in der Bearbeitungsreihenfolge auflöst bzw. substituiert, was nicht immer gewünscht ist. Der umgekehrte Fall kann mittels des speziellen Shell-Komandos **eval** erzwungen werden: **eval cmd1 ${**variable**}**
- Bei der Form f) wird der Prozess im Hintergrund von der laufenden Konsole 'detached'; die Shell bietet aber Möglichkeiten, diesen Prozess wieder in den Vordergrund zu holen bzw. zu steuern (siehe Teil 8).

Teil 3: Variablen der Shell

10. Variablen und ihre Zuweisung

In Shell-Skripten können in der Regel beliebige Variablennamen genutzt werden. Sonderzeichen sind aber in Variablennamen nicht gestattet. Die Shell unterscheidet hinsichtlich Gross- und Kleinschreibung der Variablen.

Die Besetzung einer Variablen erfolgt über das Gleichzeitszeichen (-operator). Im Gegensatz zu den meisten Programmiersprachen MUSS die Zuweisung der Variablen ohne Leezeichen zwischen Variablenname und Gleichheitsoperator bzw. Gleichheitsoperator und Wert erfolgen:

- Richtig: *var*=123; *name*='heinzpeter'
- Falsch: *var*_=_123; *name*_=_heinzpeter (Leerzeichen durch '_' verdeutlicht)

Die Shell kennt vom *Typ* oder *Cast* einer Variablen nur Integer und String. Bei der Zuweisung einer Stringvariablen mit einer Zeichenkette kann diese in einfachen oder doppelten Hochkomma eingeschlossen werden; diese werden ignoriert. Dies ist eine Voraussetzung, einer Variablen Zeichenketten mit Leerzeichen zuzuweisen.

- Identisch: *name*=heinzpeter
  name='heinzpeter'
  name="heinzpeter"
- Notwendig: *name*='heinz peter'
  name="heinz peter"
- Quotierung: *name*="heinz peter (unser 'heinz')"
  name='heinz peter (unser "heinz")'

Die Zuweisung einer Variable mit dem Wert einer anderen sollte vom *Cast* abhängig gemacht werden:

- Typ Integer: *var2*=$*var1*
- Typ String: *neuername*="$*alternname*"
  neuername='$*alternname*'

Die Besetzung einer Variable mit einem Wert kann mittels des **echo** Kommandos ausgegeben werden:

```
echo $name => heinz peter (unser "heinz")
```

11. Gültigkeitsbereiche von Variablen

Die Shell kennt drei Gültigkeitsbereiche von Variablen:

a) 'Normale' Variablen: Diese behalten Gültigkeit in der laufenden Shell und können jederzeit bezüglich ihres Wert geändert (überschrieben) werden.
b) 'Exportierte' Variablen: Diese werden zusätzlich in das Environment der aus der Shell gestarteten Programme/Shells *exportiert*; stehen dort zur Verfügung bzw. können für nachfolgende Programme geändert werden.

- **export** *VARIABLE*=wert (exportiere Variable *VARIABLE* mit Wert *wert*)
- **export** *NEW* (exportiere Variable *NEW* ohne zugewiesenen Wert)

Die ins Environment der Shell exportierten Variablen können mit dem Shell-Befehl **env** ausgegeben werden. Exportierte Variablen werden häufig in Grossbuchstaben geschrieben. Es macht keinen Unterschied, ob die zu exportierende Variable vor, beim oder unmittelbar anschliessend an das **export** Kommando hinsichtlich des

Wertes befüllt wird.
c) 'Lokale' Variablen besitzen nur Gültigkeit innerhalb einer *Function* (einem Unterprogramm) der Shell und können bei der Deklaration vom Typ (Cast) und Inhalt vorgegeben werden. Hierzu dienen die Kommandos **typeset**, **integer** und **set**.

12. Deklaration von Variablen

Die Shell kennt zwei Möglichkeiten Variablen zu deklarieren:

a) Das **typeset** Kommando: Hierdurch lässt sich festlegen, ob eine Variable vom Typ *Integer*, *String*, oder ob es sich um eine *Function* handelt.
Bei String-Variablen besteht die Möglichkeit, diese rechts- oder linksbündig mit definierter Länge abzulegen (-L[n] bzw. -R[n]); oder aber die Zeichenkette als *Upperstring* (-u) bzw. *Lowerstring* (-l) vorzugeben.
Bei Integer-Variablen kann zusätzlich angegeben werden, zu welcher Basis *n* die Variable deklariert ist (-i[*n*]). Ferner kann mittels -U der Integer-Wert als *unsigned* vorgegeben werden. Statt '**typeset** -i' kann auch **integer** verwandt werden.
Zusätzlich können die deklarierten Variablen als *Readonly* (-r) oder als automatisch zu *exportieren* (-x) gekennzeichnet werden.

Beispiele:

```
typeset variable=wert
typeset -l variable=wert      automatische Konvertierung nach lowercase
typeset -u variable=wert      automatische Konvertierung nach UPPERCASE
typeset -i number=123         oder  integer number=123
```

b) Das **set** Kommando (*nur Korn-Shell*): Hierüber können Variablen als ***Arrays*** deklariert (und vorbesetzt) werden.

Beispiele:

```
set -A array
set -A numarray 1 2 3 4
set +A numarray 0 0 0 0
```

Im ersten Beispiel wird die Variable *array* als Array mit unbestimmter Anzahl von Array-Elementen neu deklariert (-A). Der Array-Index (= Anzahl der Array-Elemente) reicht in der Regel von 0 bis 1023 (4095); wobei das erste Array-Elemente den Index 0 erhält.
Im zweiten Beispiel wird die Variable *numarray* als Array vorgeben und die ersten vier Array-Elemente vorbesetzt (-A).
Im dritten Beispiel schliesslich wird die bereits definierte Variable *numarray* um die Array-Elemente 5 bis 8 erweitert (+A), die hier mit '0' vorgegeben werden.

Zugriff auf die einzelnen Array-Elemente erhält man mittels:

- ${*array*[*num*]}

Hierbei steht *num* für den Index, bis das nummerierte Array-Feld. Die geschweifte sowie rechteckige Klammer ist mandatorisch. Wird kein Array-Element referenziert (also keine Angabe von [*num*]), wird automatisch das erste (bzw. 0-te) Elemente angenommen.

Die Anzahl der Array-Elemente ergibt sich über:

- ${#array[@]} bzw.
- ${#array[*]}

Das gesamte Array kann ausgeben werden m:

- **echo** ${array[@]} oder
- **echo** ${array[*]}

Arrays können auf Variablen abgelegt werden:

- variable=${array[@]}

Hierbei sind die einzelnen Felder mittels des $IFS getrennt. Umgekehrt gilt dasselbe:

- **set** -A *newarray* $*variable*

13. Interne Variablen und Parameter

Bei Aufruf der Skripte können Parameter als Argumente übergeben werden:

cmd.script *datei1 datei1*

Die aufrufende Shell interpretiert den Ausdruck `**cmd.script** *datei1 datei2*` entsprechend der Zuweisung der Variablen $IFS (*Input Field Separator*), der im allgemeinen das Leerzeichen, den Tabulator und das Zeilen-Endezeichen beinhaltet.

Dem aufgerufenen Programm (Skript) stehen die so separierten Ausdrücke als interne Variablen $0 ... $n (*Positionsparameter*) zur Verfügung.
Hierbei steht:

- $0 Name (bzw. Pfad) des aufgerufenen Programms
  Beispiel: **cmd.script**)
- $1 ... $n Übergebene Argumente in der angegebenen Reihenfolge
  Beispiel *datei1* => $1 ; *datei2* => $2).
- $@ Alle übergebenen Argumente
  Beispiel: "*datei* 1 *datei2*" => $@; ohne Hochkomma).
- $# Anzahl der übergebenen Argumente
  Beispiel: 2 => $#).

Folgende weitere interne Variablen sind von Bedeutung:

- $$ Prozess-Id (PID) des laufenden Prozesses.
- $? Return- bzw. Exit-Codes des letzten verarbeitenden Kommandos.
- $_ Temporäre Variable zur Ablage von Kommando-Namen; bzw. letztem Argument.

Reservierte Variablen, die von der Shell automatisch genutzt und besetzt werden, sind die folgenden:

- ERRNO Fehlernummer des letzten Aufrufs
- LINENO Zeilennummer des aktuellen Bearbeitungsschritts im Skript
- OLDPWD Pfadangabe vor dem letzten Verzeichniswechsel
- OPTARG Name des letzten von ***getopts*** bearbeiteten Arguments
- OPTIND Numerischer Wert von OPTARG
- PPID Prozess-Id des Vaterprozesses
- PWD Angabe des aktuellen Verzeichnisses
- RANDOM[=n] Berechnete (Pseudo-)Zufallszahl ausgehend vom Wert *n*
- REPLY Rückgabevariable beim Aufruf der internen Kommandos ***select*** und ***read***
- SECONDS[=n] Anzahl von Sekunden seit die Shell gestartet wurde (plus dem Wert *n*)

14. Variablen Substitution

Die Shell nicht nur die *Zuweisung* von *Variablen* mit *Werten*, sondern auch die Zuweisung von *Variablen* mit anderen *Variablen* (*Variablen Substitution*). Deren Zuweisung kann im Vorfeld *syntaktisch* und *inhaltlich* getestet werden.

Zweisung	**Bedeutung**	**Ergebnis/Beispiel**
var=wert	Zuweisung eines Wertes zu einer Variablen	x=123
var=${var1}	Zuweisung der Variablen *var* zu der Variablen *var1*	x=${y}
var=${var1-wert} var=${var1:-wert}	Falls Variable *var1* nicht gesetzt (oder leer) ist, setze *var=wert*; anderfalls setze *var=var1*	x=${y:-123} DIR=${HOME-/tmp}
var=${var1=wert} var=${var1:=wert}	Falls Variable *var1* nicht gesetzt ist, setze *var=wert* und zusätzlich *var1=wert; anderfalls setze var=var1*	x=${y:=123} DIR=${HOME:=/home/user}
var=${var1+wert} var=${var1:+wert}	Ist *var1* gesetzt (bzw. nicht leer) weise *var* den Wert *wert* zu; ansonsten den Leerstring	flag=${optimieren:+-O} cc $flags $* ${1+"$@"}
${var1?wort} ${var1:?wort}	Ist *var1* nicht gesetzt (und/oder leer), gib *wort* auf STDOUT aus	${TMP?"Variable nicht gesetzt"}

Die Korn-Shell (KSH) kennt zuästzlich die folgenden mächtigen (aber nicht immer sauber implementierte) Methoden der Variablen-Substitution:

Zweisung	**Bedeutung**	**Ergebnis/Beispiel**
${#var}	Länge von var	len=${#wort}
${#*} ${#@}	Anzahl der Positionsparameter	$1, $2; ...., ${#*}
${var#muster}	Entfernte Zeichenkette *muster* aus *var* einmalig von links aus	var=${var#'}
${var##muster}	Entferne Zeichenkette *muster* aus *var* von links aus; sooft als möglich	var=${var1## }
${var%muster}	Entfernte Zeichenkette *muster* aus *var* einmalig von rechts aus	var=${var%'}
${var%%muster}	Entferne Zeichenkette *muster* aus *var* von rechts aus; sooft als möglich	var=${var1%% }

Ergänzende Praxisbeispiele ('datei' ist der Name einer Datei per **ls**):

```
name=${datei##*/}
```

Wie auch beim Kommando **basename**, wird hier der Name einer Datei 'datei' ohne Verzeichnisangabe geliefert.

```
dir=${datei%/${name}}
```

Ausgehend vom vollständigen Pfad der Datei 'datei' und mittels 'name' wird hier der Verzeichnisname ermittelt; vergleichbar **dirname**.

Teil 4: Arithmetische und logische Operationen

15. Arithmetrische Ausdrücke

Die meisten Shells besitzen nur Arithmetik für Integer-Variablen und den Grundrechenarten und den Modulus (+, -, *, /, %) und in sehr beschränkten Umfang Verständnis für Nicht-Dezimalzahlen.
Die Variablen brauchen nicht explizit vom Typ Integer sein; String-Variablen führen aber zu einer Fehlerausgabe.

Zwei Varianten der arithmetischen Operationen sind gebräuchlich:

a) Der $(( )) Operator: `resultat=$(($1+$3)); echo $resultat`
b) Das ***let*** Kommando: `let "i = i + 1"; echo $i`

- Inkrement- bzw. Dekrement-Operatoren (++, --) werden nicht unterstützt.
- Innerhalb der arithmetischen Operation werden Leerzeichen zwischen den Variablen bzw. Operatoren ignoriert.
- Mittels des Klammer-Operators '( )' kann die Reihenfolge der Ausdrücke bestimmt werden.

16. Bit-Operatoren

In der Korn-Shell können einige Bit-Operatoren genutzt werden, was aber nur selten der Fall ist. Folgende Operatoren stehen zur Verfügung:

Operator	**Bedeutung**	**Ergebnis/Beispiel**
typeset -i*n var*	Variable *var* wird zur Basis *n* deklariert	typeset -i16 y=11 echo $y => 16#b
B#n	Integer *n* zur Basis *B* in Verbindung mit ***typeset***	typeset -i10 x=8#1000; echo $x => 512
&	Bitwise AND	
!	Bitweise OR	
^	Bitwise eXclusive OR	
<<	Bitwise shift left	
>>	Bitwise shift right	
~	Binäre Inversion	

17. Logische bzw. Verknüpfungs-Operatoren

Die Reihenfolge der Kommando-Verarbeitung kann nicht nur *sequentiell*, sondern auch *konditionell* erfolgen. Hierzu dienen die logischen Operatoren

1. **!** Negation (NICHT)
2. **&&** logisches UND
3. **||** logisches ODER

mit der angegebenen Priorität, wobei auch hier der Klammer-Operator genutzt werden kann.
Im Gegensatz zu den speziellen Shell-Operatoren, ist bei den logischen Operatoren zum folgenden oder vorhergehenden Ausdruck zwingend ein Leerzeichen erforderlich!

- Sequentielle Ausführung: **cp** *datei1 datei2*; **ls** -l *datei2*; **rm** *datei1*
- Konditionelle Ausführung: **cp** *datei1 datei2* && **ls** -l *datei2* && **rm** *datei1*

Im letzten Fall wird das nachfolgende Kommando nur dann ausgeführt, falls das vorige fehlerfrei (mit RC=0) beendet wurde.

Teil 5: Kontrollstrukturen

18. Vergleichs-Operatoren

Vergleichs-Operatoren - nicht zu verwechseln mit (komplexen) Testanweisungen - unterteilen sich in folgende Kategorien:

a) Arithmetische Vergleiche (=> 'verbale' Operatoren)
b) Zeichenketten Vergleiche (=> 'arithmetische' Operatoren)
c) Vergleiche auf Eigenschaften einer Datei.

- Im Vergleichsausdruck müssen die Vergleichs-Operatoren von den Vergleichs-Werten durch Leerzeichen getrennt werden.
- Die Vergleichs-Operatoren lassen sich mit den Verknüpfungs-Operatoren konkatinieren bzw. mit dem Negations-Operator negieren.

Arithmetische Vergleichsoperatoren:

x **-lt** *y*	*x* ist kleiner als *y*	(*less then*)
x **-le** *y*	*x* ist kleiner/gleich *y*	(*less equal*)
x **-eq** *y*	*x* ist gleich *y*	(*equal*)
x **-ne** *y*	*x* ist ungleich *y*	(*not equal*)
x **-ge** *y*	*x* ist grösser/gleich *y*	(*greater equal*)
x **-gt** *y*	*x* ist grösser als *y*	(*greater then*)

Zeichenketten Vergleichsoperatoren:

Identität:	*string1* **=** *string2*	
Ungleichheit:	*string1* **!=** *string2*	
Existenz:	*string*	(Zeichenkette existiert)
Länge:	**-n** *string*	(Zeichenkette hat Länge > 0)
Null-Länge:	**-z** *string*	(Zeichenkette hat Länge 0)
'Inhalt':	abc < abs	(wahr)
	abc > abcd	(falsch)

Zeichenketten 'Stings' sollen (müssen) aber nicht in Hochkommas gesetzt werden. Für den Vergleich leerer Zeichenketten wird auch häufig das Hilfskonstrukt "x$string" != "x" eingesetzt.

Operatoren für Dateiattribute:

Test	**Bedeutung**	**Test**	**Bedeutung**
-a file	*file* existiert	*-g file*	*file* exisiert und ist gelockt; Permission 2644
-s file	*file* ist Grösser als 0 Byte	*-k file*	*file* existiert mit Sticky Bit
-r file	*file* ist lesbar (*read*)	*-u file*	*file* existiert; Permissions 4644
-w file	*file* ist beschreibar (*write*)	*f1 -nt f2*	Datei *f1* ist neuer als *f2*
-x file	*file* ist ausführbar (*execute*)	*ft -ot f2*	Datei *f1* ist älter als *f2*
-f file	*file* ist eine *normale Datei*	*-O file*	*file* existiert und Eigner ist Abfrager
-L file	*file* ist ein *symbolischer Link*	*-G file*	*file* existiert und Gruppe ist Abfragegruppe
-c file	*file* ist ein *Character Device*	*-p file*	*file* ist eine *Named Pipe*
-b file	*file* ist ein *Block Device*	*-t file*	

19. Kontroll-Strukturen

Die meisten Shells kennen folgende Varianten bei den Kontroll-Strukturen für *Vergleich* [siehe 18.]:

1. Das **test** Kommando:
   Es wird ein Ausdruck hinsichtlich seiner 'Wahrheit' überprüft und der Return-Code entsprechend gesetzt.
2. Das [[*Vergleich*]] (Dopppklammer) Konstrukt:
   Die Doppelklammer dient in der Korn-Shell aus Testkommando (vgl. ***let*** Kommando und $((...)) bei arithmetischen Ausdrücken).
3. Die Vergleichsoperator 'einfache Klammer' [*Vergleich*]:
   Kann nur in Verbindung if-then-else, for, while und until Ausdrücken eingesetzt werden; im Gegensatz zu den beiden ersten Konstrukten, die auch 'Standalone' nutzbar sind.

Wahr/falsch

Resultat einer Vergleichs-Operation ist immer, ob der Vergleichs-Abfrage 'wahr' oder 'falsch' ist. 'Wahr' wird mit dem Rückgabewert (Return-Code der Vergleichs-Operation) durch '0'; 'falsch' mit '1' gekennzeichnet. Der Return-Code kann in der Shell über die Variable $? ermittelt werden.

20. Der test Befehl

Der Befehl **test** liefert für den Vergleichausdruck den logischen Wert 0 (wahr) und 1 (falsch). Er kann auch 'standalone' ausgeführt werden.

Beispiel: Sei a=1 und b=2

```
test $a = $b; echo $?         =>    1
test $a != $b; echo $?  =>    0
! test $a != $b; echo $?      =>    1
! test $a = $b; echo $?       =>    0
```

21. Die [[...]] Testanweisung

Der Ausdruck [[*Vergleich*]] ermittelt den Wahrheitswert von *Vergleich*.

Beispiel:

```
[[ -f /etc/passwd ]]; echo $? => 0
```

22. Das [...] konstrukt

Im Unterschied zum [[*Vergleich*]] Test müssen hierbei statt der *Verknüpfungs-Operatoren* (mit Ausnahme der nicht unterstützten Negation) die folgenden eingesetzt werden:

- **&&** => **-a** (and)
- **||** => **-o** (or)

23. if-then-else Anweisungen

In der Shell wird die if-then-else Anweisung durch die komplexe if/elif/else/fi Struktur umgesetzt, wobei mehrere ***elif*** Zweige möglich sind. Jeder Zweig in der Verarbeitungskette muss weitere (ausführbare) Statements beinhalten; ansonsten wird ein Syntax-Fehler ausgegeben.

if-then-else Anweisungen können verschachtelt werden, was ein konditionelles Befehlskonstrukt ermöglicht.

Aufbau:

```
if [ Vergleich1 ]
then
       ... tue was ...(Befehl)                  Evaluationszweig 1
elif [ Vergleich2 ]
then
       .. tue was anders ... (Befehl)           Evaluationszweig n
else
       tue was ganz anderes .... (Befehl)       letzter Evaluationszweig
fi
```

- Statt des Einfach-Klammer Operators können auch die Doppelklammern bzw. der **test** Operator eingesetzt werden.
- Einrücken der Statements erleichtert das Erkennen der Evaluationszweiges; wobei verschachtelte if-then-else Konstrukte durch Mehrfach-Einrückung gekennzeichnet werden können.
- Die ***then*** Anweisung kann auch durch das Semikolon ';' in die gleiche Zeile wie das ***if*** bzw. ***elif*** Statement 'gezogen' werden, was die Übersichtlichkeit erhöht.
- Das ***else*** Statement kommt ohne ***then*** aus.
- Das if-then-else Konstrukt wird immer mit dem ***fi*** beendet.
- Zwingend ist nur das Minimal-Konstrukt ***if/then/fi***.

Ablauf:

- Ein Evaluationszweig im if-then-else Konstrukt wird genau dann (in der angegebenen Folge abgearbeitet, falls die zugehörige Vergleichsoperation 'wahr' ergibt.

- Es wird anschliessend keine weitere Vergleichsoperation durchgeführt.
- Wird keine Vergleichsopertion mit 'wahr' abgeschlossen, wird der ***else*** Zweig ausgeführt (falls angegeben).
- Es empfiehlt sich daher aus Laufzeitgründen. die wahrscheinlichsten 'wahren' Vergleiche an den Anfang zu stellen.

Komplexe Abfragen:

Innerhalb der Vergleichszweige ***if*** und ***elif*** könnn komplexe Vergleiche durchgefüht werden, abhängig davon ob eine Testanweisung [[...]] oder eine einfache Klammerung genutzt wird.

Beispiele:

```
if [[ $a -eq $b || $c -gt $ d ]]
if [ $a -eq $b ] && [ $c -gt $d ]]
```

24. case/esac Anweisung

Die case/esac Anweisung: Flexibler Vergleich eines Testausdrucks und bedingte Ausführung (einfacher)Folge-Statements.

Aufbau und Beispiel:

```
case $variable in
      (test)            Befehl_1 && Befehl_N;;  ($variable = test ?)
      (test1|test2)     Befehl_2; Befehl_X;;    ($variable = test1 |
test2 ?)
      (t*)              Befehl_Y;;              ($variable fängt mit t an ?)
      (*)               Befehl_Z;;              (Default-Anweisung)
esac
```

Der ***case*** Operator vergleicht die Variable $variable der Reihe nach mit der einer Liste von Testausdrücken '(...)'. Wird eine Bedingung erfüllt werden die nachfolgenden Befehle ausgeführt.

- Der Testvergleich kann entweder 'genau' ODER mittels des '|' Operators alternativ ODER mittels Wildcards [vgl. 4.] erfolgen (ODER = exklusives oder).
- Das Ende der Befehls-Statements [vgl. 3] wird durch ein doppeltes Semikolon ';;' mitgeteilt.

Hinweis: Bei manchen Shells ist es erlaubt, den Testausdruck mit einer einfachen runden rechten Klammer ')' statt des geschlossenen Klammerausdrucks zu realisieren; dies empfiehlt sich aber nicht.

25. Ablaufkontrolle und Schleifen

Die Shell kennt zwei Typen von Schleifenkommandos:

- Die *listengesteuerte* ***for*** Schleife.
- Die *vergleichsgesteuerten* Schleifen mit ***while*** und ***until***.

In allen Fällen werden die Befehlssequenzen innerhalb eines Schleifenzweigs durch ein ***do***/***done*** Konstrukt begrenzt. Alternativ kann wie in anderen Programmiersprachen die geschweifte Klammer eingesetzt werden '{ ... }'.

Schleifen können verschachtelt werden, wobei sich die Shell die 'Tiefe' der Verschachtelung merkt. Mittels der Kommandos

- ***break*** [*n*] kann *n* Stufen (Default: 1) in der Verschachtelungstiefe zurückgesprungen werden, bzw. mittels
- ***continue*** [*n*] kann aus der bestehenden Schleife unmittelbar zur Bearbeitung der nächsten (*n*-ten) Schleife gesprungen werden.

Hinweis: Von dieser Möglichkeit nimmt man aber nur im Notfall Gebrauch.

26. Die for Schleife

Die ***for*** Schleife
1. parst die nach ***in*** angegebene *Variablen-Liste* [vgl. 4.] und interpretiert diese,
2. weist sukzessive jede interpretierte Variable der nach ***for*** deklarierten *Testvariablen* zu, und
3. führt ANSCHLIESSEND die Befehle im ***do***/***done***-Zweig aus, solange *Testvariable* existiert (String-Test!).

Beispiel:

```
for datei in *
do
      echo "Überprüfe Datei: $datei"
      ls -l $datei
      file $datei
done
```

- Die zu überprüfenden Variablen-Liste kann explizit oder (wie im Beispiel) implizit angegeben werden.
- Wird im for-Statement auf die in-Anweisung verzichtet, wird als Variablen-Liste automatisch die Shell-Stellungsparameter '$@' angenommen.
- Die Liste der zu überprüfenden Variablen kann auch über ein Kommando erzeugt werden.

=> Im obigen Beispiel liesse sich auch formulieren:
'for datei in `ls`', oder 'for datei in `ls *`' bzw. 'for datei in $(ls *)'

Je nach Inhalt des Verzeichnisses kann dies allerdings zu unterschiedlichen Resultaten führen!

27. Die while Scheife

Die ***while*** Schleife
- führt die Anweisungen zwischen ***do*** und ***done*** solange aus, wie *Kriterium* wahr (d.h. Return-Code gleich 0) ist,
- wobei *Kriterium* immer VOR dem ***do*** Statement, d.h. bei jedem Schleifendurchlauf getestet wird.
- *Kriterium* kann jedes Kommando sein.

Aufbau:

```
while [ Kriterium ]
do
      Befehle
done
```

Hinweis: Ist *Kriterium* immer wahr, wird die Schleife endlos lange fortgesetzt.

28. Die until Schleife

Die ***until*** Schleife
- führt die Anweisungen zwischen ***do*** und ***done*** solange aus, bis *Abbruchkriterium* erfüllt (d.h. Return-Code gleich 0) wird,
- wobei *Abbruchkriterium* immer NACH dem *done* Statement, d.h. bei jedem Schleifendurchlauf getestet wird.
- *Abbruchkriterium* kann auch hier jedes Kommando sein.

Aufbau:

```
until [ Abbruchkriterium ]
do
      Befehle
done
```

Hinweis: Schleifen mit ***for*** und ***until*** stellen äquivalente Formen dar, die sich nur in der Formulierung der Testbedingung unterscheiden.

Teil 6: Shell-Skripte

29. Shell-Skripte

Werden in einer Datei mehrere Kommandos zeilenweise (vgl. 3.) hinterlegt, können diese von der Shell in einem Stapel ('Batch') verarbeitet werden. Hierzu gibt es drei Varianten (Beispiel: *cmd.script*; der Suffix spielt keine Rolle):

- Variante 1: Das Skript wird über den Aufrufen: **sh** *cmd.script* explizit aufgerufen:
- Variante 2: Nachdem das Skript mittels **chmod** +x *cmd.script* ausführbar gemacht wurde, wird es als Kommando *in der laufenden Shell* ausgeführt: ***./cmd.script***.
- Variante 3: In der ersten Zeile von cmd.script die Anweisung '#!/bin/sh' als Kommando-Interpreter hinterlegt Nach **chmod** +x *cmd.script* wird das Skript in der dort eingetragenen Shell gestartet.

30. Übergabe und Verarbeitung von Argumenten

Der Aufruf eines Shell-Skriptes wird zwei-stufig vorgenommen:

1. Die laufende Shell expandiert eventuell vorliegende Argumente bzw. Parameter.
2. Die expandierten Parameter werden entsprechend dem $IFS in Token zerlegt und dem Skript als Aufruf-Argumente (Stellungsparamater) zur Verfügung gestellt: $1 $2 ...

Das eigentliche Skript wird hierbei als Parameter '0' aufgefasst und steht im

aufgerufenen Skript konsequenterweise als $0 zur Verfügung.
Alle Argumente zusammen werden von der aufrufenden Shell in Form der Variablen $@ angeboten; also $@ = $1 $2 .....
Neben der Variablen $@ kann die Ausgabe der Argumente noch wie folgt vorgenommen werden:

- $* Alle Argumente als gemeinsam Zeichenkette: "$1 $2 .... "
- "$@" Alle Argumente einzeln quotiert: "$1" "$2" ....

Häufig wird auch die Form ${1+"$@"} zum Skript-Aufruf genutzt. Hier wird zunächst überprüft, ob ein Argument existiert ($1). Falls dies der Fall ist, wird in Ergänzung die gesamte Argumenten-Kette genutzt.

Im aufgerufenen Programm können die Stellungsparameter mittels des ***shift*** Kommandos manipuliert werden:

- ***shift*** entfernt den ersten Parameter, sodass nun der Wert von $2 an die Stelle von $1 nachrückt.
- ***shift*** *n* entfernt die ersten *n* Parameter und $1 wird nun von $n+1 belegt.

31. GETOPTS

Viele Unix-Befehle und Shell-Skripte nehmen eine Unterscheidung vor, ob ein Argument

- ein *Parameter* oder
- eine *Option* ist.

Optionen werden unter Unix traditionell mit einem "-" Minus-Zeichen vorangestellt, z.B. "-v" für die Option 'verbose'. Häufig wir auch eine zusätzliche Form mit doppelten Minus-Zeichen "--" eingesetzt, also z.B. "--verbose". Beide Formen schalten eine Option 'ein'. Die umgekehrte Variante, eine Option auszuschalten wird mit einem "+" Plus-Zeichen eingeläutet; ist aber so gut wie nicht mehr gebräuchlich.
Optionen können mit ergänzenden *Parametern* versehen werden, wo der Parameter zur Option aber nicht zum aufgerufenen Programm selbst gehört.

Zur allgemeinen Auswertung von Programm-Argumenten mit Parametern, Optionen und Options-Parametern bietet das Shell das ***getopts*** Kommando. Dieses prüft üblicherweise in einer while-Schleife die Stellungsparameter $1 ... $n und wertet per Default ein einleitendes Minus-Zeichen bzw. ein explizit angegebenes Plus-Zeichen als Option aus.

Optionen werden case-sensitive ausgewertet und müssen dem ***getopts*** Kommando explizit (aber in beliebiger Reihenfolge) mitgeteilt werden. Ist der Vorrat an bekannten Optionen erschöpft, übergibt ***getopts*** die weiteren Argumente als Parameter. Damit ***getopts*** Programm-Parameter von Options-Parameter unterscheiden kann, sind Optionen mit zusätzlichen Parameter durch einen nachgestellten Doppelpunkt (:) zu kennzeichnen. Wird eine solche Option erkannt, speichert ***getopts*** im Verarbeitungszyklus das nachfolgende Argument in der Kette als interne Variable $OPTARG ab und die ergänzende interne Variable $OPTIND liefert den Index (Zähler) des nächsten, noch nicht verarbeitenden Arguments.

Beispiel:

```
options="vhi:o:"

while getopts ${options} arg
do case ${arg} in
     (v)   verbose=1;;
     (+v)  verbose=0;;
     (h)   help=1;;
     (i)   inputfile=${OPTARG};;
     (o)   outputfile=${OPTARG};;
     (*)   print -u2 "Bitte Inputdatei (-i) und Outputdatei (-o) angeben!";;
     esac
done

shift OPTIND-1 # Nicht notwendig, falls keine weitere Parameter-Verarbeitung
```

32. Einlesen und Verarbeiten von Dateien

Häufig besteht in Shell-Skripten die Notwendigkeit, (ASCII-) Dateien auszulesen bzw. weiter zu verarbeiten.
Grundsätzlich zerlegt die Shell eine Datei in

- Zeilen, die durch ein End-of-Record (EOR) Zeichen gekennzeichnet sind (unter Unix typischerweise das Line-Feed-Zeichen LF bzw. x'0A' und unter Windows CR+LF, also x'0D0A', sowie eine einzelne Zeile in
- Wörter, die durch 'White Spaces' (Leerzeichen bzw. Space x'20', Tabulator x'09') getrennt sind.

Im Verarbeitungszyklus wird die Datei zeilenweise abgearbeitet; die Zerlegung einer Zeile in (beliebige) Variablen kann hingegen durch die Variable $IFS (*Internal Field Separator*) gesteuert werden; was insbesonders beim Verarbeiten von CSV-Dateien (Comma-Separated-Values) effizient ist (IFS=,;).

Das Einlesen einer Datei kann über zwei Mechanismen erfolgen:

- Sequentielle Verabeitung (schont den Speicher, ist aber langsam), bei der die Dateien zeilenweise gelesen und verarbeitet wird.
- Bulk-Verarbeitung (Speicher-intensiv, aber schneller), die Datei wird auf eine Variable im Speicher geschrieben und diese anschliessend abgearbeitet.

Sequentielle Verarbeitung:

```
while read wort1 wort2 wort3 wortN
do
     echo "wort1" # Erstes Wort
     echo "wortN" # Alle Wörter, die nach dem dritten folgen

done < /pfad/datei
```

Bulk-Verarbeitung (Wort-weise):

```
#datei=$(cat /pfad/datei)

datei=$(< /pfad/datei) # schneller !

for wort in ${datei}
do
     echo "$wort" # jedes einzelne Wort
done
```

Bulk-Verarbeitung (Zeilen-weise):

```
datei=$(< /pfad/datei)

IFSORG=$IFS # Behalte den $IFS für später
IFS=\0x0A   # "\0x" => Variable wird als hexadezimaler Werte betrachtet

for zeile in ${datei}
do
     echo "$zeile" # jedes einzelne Zeile
done

IFS=$IFSORG
```

33. Ein- und Ausgabe

Ein- und Ausgabe von Zeichenstrings in bzw. aus der Shell kann über die folgenden (internen) Kommandos erzielt werden:

Ausgabe:

echo	Ausgabe auf STDOUT (sofern nicht umgelenkt)
print [-n] -u*N*	Ausgabe auf Filedeskriptor *N* (KSH)

Die Argumente von ***echo*** bzw. ***print*** können gemeinsam (=> ein Argument) oder einzeln (=> mehrere Argumente) gequotet sein. Werden die Argumente ohne bzw. mit Doppelquotes angegeben, unterliegen diese der Shell-Expansion. Will man das verhindern, können die Zeichenketten in Einfach-Quotes gesetzt, bzw. es kann vom Escape-Zeichen (\) Gebrauch gemacht werden.

Beide Kommandos lesen alle Zeichen (also auch White-Spaces) bis zum Ende der Kommandozeile ein (Zeilenende-Zeichen). Letzeres wird auch immer mit ausgegeben; ausser bei ***print*** -n (no Newline).

Eingabe:

read *var1 var2 ..*	Einlesen der Variablen *var1 ...* von STDIN
read -u*N* var?prompt var1 ...	Einlesen von Variablen *var* unter Ausgabe von *prompt*

Massgeblich für die Interpretation der Eingabe- und Ausgabezeile in einzelne Zeichenketten-Tokens ist die Deklaration des *Internal Field Separators* $IFS.

34. Funktionen

Funktionen der Shell stellen für die hierin ausgeführten Kommandos eine spezifische Umgebung bereit, die Teile der übergeordneten Shell-Umgebung beinhalten:

- Normale Variablen behalten Gültigkeit,
- das Arbeitsverzeichnis sowie
- Dateizuordnungen, die mit dem Kommando **exec** erzeugt wurden.

Folgende Eigenschaften sind jeder Funktion eigen:

- Die Stellungsparameter ($0, $@, $*); $0 fungiert als Funktionsname; $n sind die Argumente der Funktion,
- lokale Variablen, die mittels typeset (bzw. integer) in der Funktion deklariert sind; mit Ausnahme von Array, die immer 'globale' Gültigkeit

besitzen,
- explizite Dateizuordnungen,
- Signalbehandlungen (Traps) sowie
- Shelloptionen.

Da die Shell im Gegensatz zur Programmiersprache C keine Vorab-Deklaration von Funktionen kennt, müssen diese im Hinblick auf die Interpretation der Statements, *vor* dem ersten Aufruf bekannt sein.

Zwei Aufrufe-Formen von Funktionen sind gebräuchlich:

- function name { kommandos; }
- name() { kommandos; }

Eine Funktion besitzt nach ihrem Aufruf immer einen Rückgabewert (Return-Code). Dieser kann entweder explizit mit dem internen Shell-Kommando ***return*** *n* gesetzt werden; oder aber ist implizit der Return-Code der letzten Operation bzw. des letzten ausgeführten Kommandos. Im aufrufenden Programm steht dieser Return-Code über die Shell-Variable $? zu Verfügung.

Achtung! Die Variable $? darf nicht negativ sein und sollte einen Wert von 1000 nicht überschreiten (in C deklariert als kleiner, positiver Integer-Wert).

Beispiel:

```
#!usr/local/bin/ksh
#
## Beispiel eines Shell Skriptes mit Funktionen
#
#  Angabe von Argumenten + Aufruf-Optionen (Synopsis)
#  Angabe des Autors (Copyright)
#  Versions-Informationen + Datum (Versionskontrolle)
#
## Initialisierung von Variablen ---------------------------- Abschnitt 1
#
## set -o xtrace # Debugging!
#
echo "$0: Das ist mein Name!"
echo "Aufruf erfolgt an: `date`"
#
## Bereich der Funktionen ----------------------------------- Abschnitt 2
#
function subroutine1
{
 integer rc=0

 echo "Aufruf der Funktion $0 mit den Argumenten \"$*\""

 return ${rc}
}
#
subroutine2()
{
 integer rc=0

 echo "Aufruf der Funktion $0 mit den Argumenten \"$*\""

 retrun ${rc}
}
#
## Hauptprogramm (Main) ------------------------------------- Abschnitt 3
#
```

```
subroutine1 "Hello" "World"
rc=$0
echo "Return-Code von Funktion 'subroutine1': ${rc}"

subroutine2 "Hello World"
rc=$0
echo "Return-Code von Funktion 'subroutine2': ${rc}"

exit 0
```

Der Aufbau eine Shell-Skriptes kann also wie folgt gewählt werden:

- Abschnitt 0 (*Präambel*) Angabe Interpreter; Beschreibung des Programms, Synopsis, Copyright + Version
- Abschnitt 1 (*Deklarationen*) beinhaltet das Setzen von gloablen Optionen und Variablen einschliesslich der Auswertung von Argumenten (mit **getopts**).
- Abschnitt 2 (*Funktionen*) weist die Funktionen auf, die in der Reihenfolge ihres Aufrufs eingebunden sein müssen.
- Abschnitt 3 (*Main*) mit Aufgruf der Funktionen und Abschluss mittels des ***exit*** Kommandos.

35. Aliase

Die Shell ermöglicht die Deklaration von Aliasen. Ein Alias ist besonders dann hilfreich, wenn ein externes Kommando mit speziellen Optionen öfter im Shell-Skript genutzt werden soll. Drei Einsatzformen des internen Kommandos ***alias*** sind nutzbar:

- *Kommando-Substitution:* ***alias*** dir='basename `pwd`'
  Das externe Kommando **pwd** (unter Abzug der Verzeichnishierarchie) wird auf den internen Alias ***dir*** abgebildet.
- *Exportierter Alias:* ***alias*** -x dir='basename `pwd`'
  Der Alias ***dir*** wird nun auch in Subshells exportiert und kann hier genutzt werden.
- *Tracked Alias:* ***alias*** -t dir='basename `pwd`'
  Der Alias ***dir*** behinhaltet jetzt nicht nur Referenzen auf die Unix-Kommandos **pwd** und **basename**, die zur Laufzeit des Kommandos (über den Default-Pfad) aufgelöst werden müssen sondern Referenzen auf den kompletten *Pfad* der Kommandos.

Hinweis: Das externe Kommando muss zur Deklaration als Alias in einfachen Hochkommas eingeschlossen sein.

36. Debuggen von Shell-Skripten

Treten Fehler in Shell-Skripten auf, sind diese nicht einfach zu diagnostizieren. Häufig liegen aber einfache Syntax-Fehler vor (unvollständige Klammerpaare bzw. Hochkommas), die am besten mit einem syntax-fähigen Editor (emacs) erkannt werden können.

In Korn-Shell Skripten können die Start-Optionen:

- ***set*** -o xtrace
- ***set*** -o trackall

genutzt werden. Im ersten Fall werden nicht nur die Resultate, sondern auch die ausgeführten Kommandos und Zwischenergebnisse dargestellt. Der zweite Fall ist nützlich für nicht-definiert lokale Variablen. Keine der Optionen ist aber wirklich hilfreich bei Fehlern in internen Funktionen.

Teil 7: Aufruf der Shell und Shell-Umgebung

37. Aufruf der Shell und Beendigung der Shell

Unter Unix findet die Shell typischerweise mit fünf unterschiedlichen Aufrufmerkmalen Anwendung:

- Login-Shell:
  Diese ist standardmässig im Login-Prozess eingebunden und wird nach erfolreich Authentisierung des Nutzers ausgeführt. Schaltstelle hierfür ist die Konfigurationsdatei /etc/passwd, in der für jeden Benutzer explizit eine Login-Shell angegeben werden muss, die mit der Option '-l' bzw. '--login' gestartet wird.

- Interaktive Shell:
  Wird die Shell mit der Option '-i' aufgerufen, wird diese zur interakativen Shell, bei der STDIN, sowie STDOUT und STDERR mit der laufenden Konsole verbunden ist. Zusätzlich wird der Default-Prompt $PS1 gesetzt sowie die T*ilde-Expansion* für die $HOME Variable (~ = $HOME) vorgenommen und häufig auch das sog. F*ilename-Completion* (über die Tabulatur-Taste) eingeschaltet. Im allgemeinen werden auch die eingegebenen Kommandos in einem sog. *History-File* abgelegt und können durch Vorstellen mit einem Ausrufezeichen (!cmd) bzw. über das **history** Kommando und mittels (!n [Kommandosequenz]) wieder geholt werden.

- Standard Shell:
  Der Benutzer kann mehrer Instanzen der Shell einfach durch Angabe des Shell-Namens erzeugen. Die aktive Shell wird in der Environment-Varablen $SHELL gelistet; die aktuelle Instanz dieser Shell über die ergänzende Variablen $SHLVL.

- Privilegierte Shell:
  Ist die effektive Unix Benutzer-ID gleich der realen, wird die Shell automatisch mit einer Konfigurationsdatei ~/.profile gestartet und über die Envrionment-Variablen $ENV eine zusätzliche Konfiguarationsdatei (üblicherweise ~/.shrc) vorgegeben. In Resultat führt dies zu gewissen Voreinstellungen. Ist ersteres nicht der Fall, bzw. wird die Shell mit der Option '-p' aufgerufen, spricht man von einer 'privileged shell', die ohne Voreinstellungen gestartet wird.

- Restricted Shell:
  Die restricted Shell (aufgerufen über die Option '-r' bwz. explizit als **rsh**/**rksh**) wird gelegentlich in sicherheits-relevanten Bereichen eingesetzt, da einige Standard-Operationen verboten sind:
  - Wechseln des Verzeichnis mittels **cd**.
  - Überschreiben bzw. Setzen der Pfad-Variablen $PATH sowie $ENV und $SHELL.
  - Absolute Adressierung eines Kommandos über ein einleitendes '/'.
  - Unterbindung der Umleite-Operatoren '>' und '>>'.

Die Shell kann unter Angabe des Kommandos **exit** (bzw. **logout**) beendet werden. Alternativ hierzu lässt sich auch das Signal TERM (auf der Konslole unter Eingabe von ^d [STRG-d]) nutzen.

38. Shell Konfigurations-Dateien

Beim Aufruf der Shell werden (abhängig von Aufrufe-Parametern) bestimmte Konfigurationsdateien gelesen und hierüber Konfigurations-Parameter gesetzt.

Allgemein (sh, bash, ksh):

- .profile ist die Standard-Konfigurationsdatei für Login-Shells, setzt $ENV
- .shrc wird über die $ENV Variable gesetzt

Nur c-Shell:

- .cshrc Konfigurationsdatei für die c-Shell
- .login csh login Konfigurationsdatei

Diese 'Punkt-Dateien' stellen ein Mittelding zwischen einer Konfigurationsdatei einerseites und einem Skript andererseits dar. Bei der KSH (Korn-Shell) steht zum Einlesen von Punkt-Dateien das interne Kommando '.' bereit. Wird dieses ausgeführt (". .conf") werden automatisch alle in .conf deklarierten Variablen exportiert.

39. Umgebungsvariablen und Shortcuts

Ohne das Setzen von Umgebungsvariablen, wäre das Arbeiten in der Shell viel mühsamer; besonders beim Bildschirmsitzungen.

- Allgemeine Umgebungsvariablen:
  ENV; HOME; IFS; PATH; LANG; SHELL

- Umgebungsvariablen für die Console:
  COLUMNS; LINES; EDITOR; PS*n* (Prompt-Zeichenkette); TERM

- Umgebungsvariablen für den Editor:
  EDITOR; VISUAL

- Umgebungsvariablen für den Mailclient (obsolet):
  MAIL; MAILCHECK; MAILPATH

Üblicherweise ist die Variable $PATH von besonderem Intersse. Eine gängige, aber unsichere Praxis ist es, das aktuelle Verzeichnis automatisch aufzunehmen:

- PATH=$PATH:.

Sicherer ist es, speziell Skripte nicht beliebig im Dateibaum unterzubringen, sondern in einem dedizierten Verzeichnis, was sich Unix-typisch *bin* nennt:

- PATH=$PATH:$HOME/bin

Die Shell kennt auch verschiedene Shortcuts, die auf Verzeichnisse bzw. Befehle verwiesen:

- Das aktuelle Verzeichnis: .
- Das übergeordnete Verzeichnis: ..
- Das $HOME Directory: ~

- Filename Completion: Anfangsbuchstaben + Tabulator (Suche $PATH)
- Kommando Wiederholung: Cursor Taste hoch/runter
- History Kommando: !cmd; bzw. !Kommandonummer (aus **history**)

Teil 8: Prozesse und Prozess-Steuerung

40. Jobs und Job-Steuerung

Der Aufruf eines Skripts oder eines Kommandos aus der aktuellen Shell kann gesteuert werden:

- Standard-Aufruf: cmd -- ./skript.sh
  Kommando/Skript wird im Vordergrund ausgeführt; es ist mit der laufenden Konsole verbunden, und die aktuelle Shell muss warten bis sich das Kommando/Skript beendet hat, bzw. kann mittels STRG-C abgebrochen werden.

- Dämonisierter Aufruf: cmd **&** -- ./script.sh **&**
  Das Kommando/Skript wird im Hintergrund ausgeführt (Daemon-Prozess); es ist von der Kondole 'detached'. Während es läuft kann in der aufrufenden Shell weitergearbeit werden. Das Kammando/Skript wird als *Job* betrachtet und kann mittels Shell-Kommandos angehalten, fortgesetzt und abgebrochen werden. Ferner kann der Job zwischen Foreground- und Background-Verarbeitung umgeschaltet werden. Ebenfalls kann der Prozess über die gängigen Signalgeber (**kill** Kommando) beeinflusst werden. Wir die aktuelle Shell beendet, gilt dies auch für den gestarteten Job.
- Geschützter Aufruf: **nohup** cmd (**&**) -- **nohup** ./skript.sh (**&**)
  Das Kommando **nohup** schützt das aufgerufene Kommando/Skript for externen Signalen. Das Kommando/Skript läuft so lange weiter, bis intern die Verarbeitung abgeschlossen ist bzw. ein Abbruch erfolgte. Es kann nicht durch Signale beeinflusst werden, ausser durch das KILL Signal (über den Dispatcher). Ebenfalls läuft es auch dann noch weiter, wenn die aufrufende Shell beendet wird (z.B. über **logout**).

Die Job-Kontrolle der Shell ermöglicht es, mehrere Prozesse gleichzeitig zu beeinflussen. Hierzu erhält jeder dämonisierte Aufruf eine Job-Id, über den dieser referenziert werden kann:

- ***jobs*** Gibt die Liste der aktuellen Jobs mit Job-Id aus.
- ***fg*** (%*n*) Überführe aktuellen Job, bzw. Job Nummer *n* in den Vordergrund
- ***bg*** Stellt den aktuellen Job in den Hintergrund
- ***stop*** (%*n*) Suspendiert einen Hintergrund-Job
- ***wait*** (%*n*) Warte auf die Beendigung von Hintergrund-Job *n*
- **suspend** (%*n*) Suspendiere Vordergrund-Job *n*

- %% Laufender Job
- %*s* Job mit Anfangsbuchstabe der Kommandozeile *s*
- %?*s* Job, dess Kommandozeile den Buchstaben *s* enthält
- %*n* Job Nummer *n*
- %+ Laufender Job
- %- Vorheriger Job

41. Co-Prozesse

Als Herausstellungsmerkmal besitzt die Korn-Shell die Möglichkeit, Prozesse (*Jobs*) nicht nur in den Hintergrund zu stellen, sondern diese auch mit der aktuellen (interaktiven) Shell kommunizieren zu lassen, also quasi eine Pipe zwischen beiden Prozessen aufzubauen:

- cmd |&

Die Kommunikation zwischen Vater- und Kindprozess erfolgt mittels der Option '-p' bei den Kommandos **read** und **write**.

Beispiel:

```
## Start des Unix Taschenrechners 'bc'

bc -l |&
#
## Überprüfe
#
jobs
[1] + Running              bc -l
#
## Berechne
#
print -p "sqrt(13)"
read -p ergebnis
print "Ergebnis: $ergebnis"
Ergebnis: 3.6055512754
#
## Stop bc
#
stop %1
[1] + Stopped (signal)     bc -l
```

Teil 9: Literatur und Hinweise

LITERATUR:

Bernd Eggink: Kornshell-Programmierung; Hanser-Verlag, 1994
Daniell Gilly: Unix in a Nutshell, O'Reilly & Associates, 1992

HINWEIS:

Das Kompendium hat keinen Anspruch auf Vollständigkeit.
Bei der Umwandlung des Manuskripts ins PDF Format sind Formatierungsfehler aufgetreten.
Inhaltliche Fehler können nicht ausgeschlossen werden; über eine Korrektur/Hinweis per Mail würde ich mich freuen.