Shell Programmierung

Bourne-Shell: Oft zur Programmierung benutzt (Shell-Scripte)

Vorteile:

- Überall verfügbar
- Ziemlich portabel
- Reichhaltige Sprache

Nachteile:

- Langsam
- Wenige eingebaute APIs

1. Lexikalische Struktur

Die Shell liest immer zeilenweise. Zeilen werden dann an Whitespace und Operatoren in Wörter zerlegt.

Operatoren sind `&`, `&&`, `()`, `;`, `;;`, `|`, `||` und Newlines, sowie Redirektionen: `<`, `>`, `>|`, `<<`, `>>`, `<&`, `>&`, `<<-` und `<>`

Die spezielle Bedeutung eines Zeichens als Operator kann durch Quoting aufgehoben werden:

```
echo 'kein Operator: >'
echo "Kein Operator: <, aber Variable PATH: $PATH"
echo kein Operator: \>
```

Spezialfall: Backquotes zur Kommandosubstitution:

```
echo `ls|wc -l` Files vorhanden
```

2. Variablen

Bekannt: Lokale Variablen, globale Variablen, Argumentvariablen.

In der Bourne Shell müssen globale Variablen explizit exportiert werden:

```
export PATH=/bin:/usr/bin
```

Vor jedem Kommando kann man Variablen setzen:

```
PATH=/bin LOKVAR=nix  ls -l /tmp
```

Solche Änderungen gelten aber nur für das Kommando.

Vorsicht auch bei der Reihenfolge der *Expansion*:

```
PATH=/bin echo $PATH
```

Gibt den *alten* Wert von `PATH` aus, da die Expansion vor der Zuweisung von `PATH` komplett ausgeführt wird.

Expansion von Variablen/Parametern:

- `${param}` Expandiert zu Wert der Variablen `param`, oder dem leeren Wort, wenn die Variable nicht gesetzt ist.
- `${param:-wort}` *Default Wert*: Wenn Variable `param` ungesetzt oder leer, expandiert dies zu `wort`, sonst dem Inhalt der Variablen.
- `${param:=wort}` *Default Assignment*: Wie oben, nur wird auch Variable selbst gesetzt.
- `${param:?[wort]}` *Failure test*: Wenn Variable ungesetzt oder leer, bricht die Shell mit Fehler `wort` ab.
- `${param:+wort}` *Alternative Value*: Wenn Variable nicht gesetzt oder leer ist, dann substituiere leeres Wort, sonst den Wert der Variablen.

In den Versionen ohne das : wird statt 'leer oder ungesetzt' nur auf 'ungesetzt' getestet.

Kompliziertere Ersetzungen:

- `${param%wort}`: *Shortest Suffix*. `wort` wird als Pattern aufgefasst und der kürzeste Suffix des Wertes von `param`, der auf das Pattern passt, wird gelöscht.
- `${param%%wort}`: *Longest Suffix*. Wie oben, nur wird der längste Suffix gelöscht.
- `${param#wort}` *Shortest Prefix*
- `${param##wort}` *Longest Prefix* ähnlich, nur mit dem Präfix.

Die Länge des Inhaltes einer Variablen liefert `${#param}`

Arithmetische Ausdrücke: `$((ausdruck))` rechnet `ausdruck` aus.

3. Umlenkungen

In der Bourne Shell kann man Filedeskriptoren umlenken:

```
find / -name nix 2>/dev/null
make >log 2>&1
```

Reihenfolge der Umlenkungen ist wichtig :-)

- `[n]>file` Lenkt Deskriptor n in `file` um (Default: $n = 1$).
- `[n]>|file` Wie oben, nur 'clobbering' ingorieren.
- `[n]>>file` Anhängen an File.
- `[n]<file` Eingabe von Deskriptor n aus `file` (Default $n = 0$).
- `[n]<&m` Eingabe für n aus Filedeskriptor m.
- `[n]<&-` Deskriptor n schliessen.

- `[n]>&m` Ausgabe von Deskriptor n in m.
- `[n]>&-` Schliessen von n.
- `[n]<>file` Ein- und Ausgabe aus `file`.
- `[n]<<delim` Inline Eingabe für Deskriptor n.

So lenkt etwa `make 2>&1 >log` die Fehler auf das Terminal, den Standardausgabetext in ein File, während `make >log 2>&1` beides in das File schreibt.

Nützlich ist die Inline Eingabe, etwa

```
cat <<EOT
Dies ist ein Inline Text, der statt der stdin an 'cat' geht.
Variablen werden expandiert: $PATH
Backquotes ebenfalls: `ls`
EOT
```

4. Kommandoketten

Kommandos können verkettet werden:

- Hintereinanderschreiben (durch NL oder ; getrennt)
- Durch Shortcuts getrennt: `||` und `&&`
- Durch Pipes verbunden: `|`
- In Subshell eingeschlossen: `(` und `)`
- Gruppiert: `{` und `}`

Shortcuts: Werten den *Exit-Status* des ersten Kommandos aus und führen dann ggf. das zweite Kommando aus (vgl. C):

```
mkdir foobar && cd foobar
mkdir foobar || echo "Kann foobar nicht anlegen"
```

Man kann Kommandos in einer Subshell starten:

```
(cd /tmp; mkdir foobar && cd foobar && tar xvf ~/test.tar) >log
```

Subshell hat keine Auswirkungen auf aufrufende Shell.

Gruppierung: Startet Kommandos nicht in Subshell. Auswirkungen der Kommandos direkt auf diese Shell.

```
{ cd /tmp; mkdir foobar && cd foobar && tar xvf ~/test.tar} >log
```

Ergebnis einer einfachen Kette (etwa `cd /; mkdir foobar`) ist Ergebnis des letzten Kommandos.

Die Shell hat nur den Exit Status eines Kommandos zur Verfügung, um Kontrollfluss zu entscheiden.

Es gibt keine eingebauten Tests auf z.B. Vorhandensein von Files, etc.

Hierzu gibt es aber das externe `test` Programm, das solche Tests machen kann und dann mit `0` oder `1` (Fehler) sich beendet.

Also etwa:

```
if test -f /etc/hosts; then
 echo /etc/hosts vorhanden
fi


if [ -d /var/spool/mail ]; then
 echo Directory /var/spool/mail vorhanden
fi
```

`test` (oder `[`) enthält eine ganze Reihe von Tests, siehe *test(1)*.

5. Conditionale

Die Bourne Shell kennt zwei Conditionale: `if` und `case`

Die `if` Abfrage erlaubt auch kurze Alternativ-Zweige (`elif`):

```
if [ -f /etc/hosts ]; then
 HP=/etc/hosts
elif [ -f /usr/etc/hosts ]; then
 HP=/usr/etc/hosts
else
 HP=/dev/null
fi
```

Die **`case`** Anweisung dient zum Matchen eines Argumentes gegen Alternativen:

```
case `uname -m` in
 i386) ARCH=Intel
       ;;
 sparc) ARCH=SPARC
       ;;
 mips*) ARCH=Mips
       ;;

 alpha|arm) ARCH=UNSUPPORTED
         ;;
esac
```

6. Schleifen

Die Bourne-Shell kennt drei Arten von Schleifen: `while`, `until` und `for`

Bei den beiden ersten wird eine Schleife solange abgearbeitet, wie eine Bedingung wahr (falsch) ist. Die Bedingung ist dabei der Exit-Status eines Kommandos.

```
SEPPATH=`echo $PATH|tr ':' ' ' `
( set - $SEPPATH
  while [ -n "$1" ]; do
    echo We also search in Directory $1
    shift
  done
)
```

Bei der `for` Schleife wird der Schleifenrumpf wiederholt durchlaufen, wobei eine Variable nacheinander an Werte aus einer vorgegebenen Liste gebunden wird.

```
SEPPATH=‘echo $PATH|tr ’:’ ’ ’ ‘
for d in $SEPPATH; do
    echo Searching in $d
done
```

Wenn die Liste der Werte statisch ist, dann ist `for` effizienter, als das obige `while` Konstrukt.

Nachteil von `for`: Liste der Werte wird vor dem ersten Schleifendurchlauf berechnet.

`while` kann dynamisch entscheiden, da die Bedingung vor jedem Durchlauf getestet wird.

```
answer=
while [ -z "$answer" ]; do
    echo -n Yes or No:
    read answer

    case $answer in
      [yY]es) answer=Y
              ;;
      [nN]o) answer=N
             ;;
      *) answer=
         ;;
    esac
done
```

7. Funktionen

Man kann in der Bourne Shell auch Funktionen definieren. Diese können dann wie normale Kommandos aufgerufen werden.

```
errexit ()
{
 echo "An error occurred: $1" >&2
 echo "The program terminates " >&2
 exit 1
}
```

Diese Funktion kann später im Script aufgerufen werden:

```
if [ ! -f /etc/hosts ]; then
    errexit "No /etc/hosts file"
fi
```

Natürlich kann eine solche Funktion aber auch einen 'Exit-Status' zurückliefern:

```
tes ()
{
 local l
 l=`ls $1|wc -l`
 return $l
}

tes /tmp
numfiles=$?

if tes; then
 echo no files in actual directory
fi
```