2.4.F Skripte/Verzweigung – Ergänzungen und Bilder

2.4.F.1 `test`-Programm und Maskierung

Wie immer bei der Shell muss man auch beim Programm `test` darauf achten, dass unerwartete Variablen-Inhalte nichts Schlimmes anrichten. Was passiert zum Beispiel, wenn eine Variable nicht gesetzt ist?

Terminal

```
schueler@debian964:~$ set hallo # $1="hallo", $2 nicht gesetzt
schueler@debian964:~$ set -x
schueler@debian964:~$ if test $1 == "Hallo" ; then echo hallo; fi
+ '[' hallo == Hallo ']'
schueler@debian964:~$ if test $2 == "Hallo" ; then echo hallo; fi
+ test == Hallo
bash: test: ==: Einstelliger (unärer) Operator erwartet.
```

Die Abhilfe besteht darin, den Variablen-Ausdruck zu maskieren.

Terminal

```
schueler@debian964:~$ if test "$2" == Hallo ; then echo hallo; fi
+ test '' == Hallo
```

Jetzt kann es noch sein, dass eine Variable genauso aussieht wie eine Option des Programms `test`.

Terminal

```
schueler@debian964:~$ set -- -f # $1="-f"
+ set -- -f
schueler@debian964:~$ if test "$1" == Hallo ; then echo hallo; fi
+ test -f == Hallo
```

Ältere Versionen von `test` interpretierten die Zeile so, dass nach der Datei mit dem Namen == gesucht werden sollte, wobei dann ein Parameter (`Hallo`) übrigblieb, was zu einer Fehlermeldung führte. Die Abhilfe funktioniert wie folgt:

Terminal

```
schueler@debian964:~$ if test "_$1" == _Hallo ; then echo hallo; fi
+ test _-f == _Hallo
```

Statt des Unterstrichs kann dabei auch jede andere Zeichenkette verwendet werden (solange sie keine Sonderzeichen enthält).

2.4.F.2 Reguläre Ausdrücke in Logischen Ausdrücken benutzen

Manchmal möchte man bei einer Eingabe abfragen, ob es sich um eine IP-Adresse handeln kann. Für die Abfrage auf eine IP-Adresse stehe ein Regulärer Ausdruck zur Verfügung. `grep` und `sed` können mit Regulären Ausdrücken umgehen, aber wie sieht es mit der Shell aus?

Die erste Lösung verwendet ein externes Programm, nämlich `grep`:

Terminal

```
ADRESSE=1.2.3.4
schueler@debian964:~$ if echo "$ADRESSE" \
  | egrep '^([0-9]1,3.)3[0-9]1,3$' &>/dev/null
> then
>    tr echo IP
> else
>    echo keine IP
> fi
IP
```

Die zweite Lösung benutzt die Fähigkeit der Bash-Shell, reguläre Ausdrücke zu benutzen:

Terminal
```
schueler@debian964:~$ if [[ "$ADRESSE" =~ ([0-9]1,3.)3[0-9]1,3 ]]
> then
>    echo IP
> else
>    echo keine IP
> fi
IP
```

Statt der vertrauten Klammer `[` steht hier `[[`. Der Befehl `[[` ist ein eingebauter Befehl der Shell (Hilfe deswegen mit `help [[`). Im Gegensatz dazu ist `[` bzw. `test` ein externes Programm. Der Befehl `[[` muss als letzten Parameter `]]` bekommen. Das sieht dann symmetrisch aus (wie bei `[` und `]`). Ansonsten kann man mit `[[` alles machen, was bei `[` bzw. `test` auch funktioniert[1]. Der Befehl `[[` ist erst vor einigen Jahren zur Bash-Shell hinzugekommen. Da es sich um einen eingebauten Befehl handelt, ist `[[` allgemein etwas schneller als `[` bzw. `test`.

Im obigen Beispiel kommt ein weiterer Pluspunkt hinzu: `[[` versteht auch reguläre Ausdrücke – das waren die Textmuster, die in `grep` , `sed` und vielen Programmiersprachen benutzt werden. Dazu setzt man zwischen das zu untersuchende Wort (links) und das Textmuster (rechts) den Operator =~. Wichtig dabei sind folgende Regeln:

- Das Muster steht auf der rechten Seite.
- Für C++- und Java-Programmierer: Der Operator ist =~, bei C++ und Java liegt bei Operatoren, die ein Gleichheitszeichen enthalten, das Gleichheitszeichen immer hinten. Hier ist das Gleichheitszeichen vorne!
- Das Suchmuster darf *nicht* maskiert werden!
- Es werden *extended posix regular expressions* verwendet wie bei `egrep`.

Zum Ausprobieren kann man den Rückgabewert abfragen:

Terminal
```
schueler@debian964:~$ [[ TEST =~ XY.* ]]; echo $? #regex passt nicht
1
schueler@debian964:~$ [[ TEST =~ T.ST ]]; echo $? #regex passt
0
```

Hilfe erhält man mit `help [[`.

Ein weiterer Unterschied von `[[` zu `[` bzw. `test`: Wenn man bei `[[` einen der Vergleiche == oder != verwendet, wird *pattern matching* verwendet (mit Erweiterungen wie bei `extglob`). Zur Erinnerung: Reguläre Ausdrücke und *pattern matching* sind zwei verschiedene Sprachen! Auch in diesen Fällen muss das Suchmuster auf der rechten Seite stehen:

Terminal
```
schueler@debian964:~$ [[ TEST == T?XT ]]; echo $? #pattern passt nicht
1
schueler@debian964:~$ [[ TEST == T?ST ]]; echo $? #pattern passt
0
```

Durch das (ohnehin übliche) Maskieren kann man das *pattern matching* an dieser Stelle ausschalten:

Terminal
```
schueler@debian964:~$ [[ TEST == "T?ST" ]]; echo $? #kein p.m.
1
```

[1] Ein Unterschied: Die Doku zu `help` sieht als Vergleichsoperator ein Gleichheitszeichen vor, die Doku zu `[[` sieht zwei Gleichheitszeichen vor. Trotzdem funktioniert bei beiden auch die jeweils andere Schreibweise.

2.4.F.3 Shell-Grammatik: Was ist eine Befehlsliste?

Die Shell-Grammatik unterscheidet zwischen einfachen Befehlen (*simple commands*), Befehlsketten (*pipelines*) und Befehlslisten (*lists*).

Ein *einfacher Befehl* ist wie gehabt eine Ansammlung von Worten und Umleitungen, die durch ein Zeilenende oder einen der Kontroll-Operatoren `|| & && ; ;; ( ) |` beendet wird. Das erste Wort, das keine Umleitung ist, wird als Befehlsname ausgewertet.

Eine *Befehlskette* ist entweder ein einfacher Befehl oder eine Folge von Befehlen, die durch `|` getrennt und durch das Zeilenende (oder einen Kontroll-Operator) abgeschlossen ist.

Bei Schleifen und Verzweigungen kommt es oft auf den Rückgabewert von Befehlsketten an:

- Rückgabewert des letzten Befehls ist der Rückgabewert der Kette.
- Setzt man vor die Kette ein Ausrufezeichen `!`, so wird der Rückgabewert der Kette invertiert (0 nach 1, nicht-0 nach 0). Das funktioniert natürlich auch für einfache Befehle.

Eine *Befehlsliste* besteht aus mehreren Befehlsketten, die durch `; & && ||` getrennt sind. Abgeschlossen wird die Befehlsliste durch ein Zeilenende, durch ein `;` oder ein `&` [2] [3].

Der Rückgabewert der Befehlsliste ist der Rückgabewert der letzten ausgeführten Befehlskette.

2.4.F.4 Shell-Grammatik: Verbundbefehle

Die Grammatik der Shell erlaubt es, mehrere Befehle zusammenzufassen:

```
1 ( BL; ) # Befehlsliste BL wird in neuer (Sub-) Shell ausgefuehrt
2 { BL; } # Klammerung ohne neue (Sub-) Shell
3 (( AUSDRUCK )) # Ausdruck wird arithmetisch ausgewertet;
4                # ergibt Wert 1, falls Ergebnis 0 ist, sonst 1
5 [[ AUSDRUCK ]] # Bedingungsausdruck wird logisch ausgewertet,
6                # spart den Aufruf des Programmes test
7                # Operatoren: ==, !=, =~ (regex!), !, &&, ||, ()
```

2.4.F.5 Betriebsarten der Shell

Die Manual-Page der Bash unterscheidet zwischen drei wichtigen Betriebsarten:

a) Interaktive *Login-Shell*: die Shell, die nach dem Login aufgerufen wird

b) *Interaktive Shell*: eine Shell, die ohne Skriptnamen aufgerufen wird, z.B. durch Öffnen eines Konsolen-Fensters unter KDE

c) *Nicht-interaktive Shell*: eine Shell, durch Aufruf eines Skriptes (oder mit einem Skriptnamen als Parameter) aufgerufen wird und dazu dient, ein Skript auszuführen.

Mit dem Befehl `shopt -p login_shell` kann man herausfinden, ob sich die Shell in der entsprechenden Betriebsart befindet. Das Ergebnis kann lauten:

- `s` (=set): eingeschaltet
- `u` (=unset): ausgeschaltet

[2] Die Operatoren `&&` und `||` haben dabei Vorrang vor `;` und `&`.

[3] Mit dem Operator `;` wird dabei die Kontrollstruktur der Sequenz verwirklicht. Mit dem Operator `&`, der in interaktiven Shells die Hintergrundausführung bewirkt, wird die Kontrollstruktur der Konkurrenz (Gleichzeitigkeit) bewirkt.