Patricia Jung

Einführung in die Shellprogrammierung

Modul IF LIN 04 der Linux-Akademie im Rahmen der Informatica Feminale 2008 in Bremen

Wildcards, Pipes, Ein- und Ausgabeumlenkung sind nicht alles, was das Powertool Shell zu bieten hat – als „Killerfeature“ haben Unixshells (fast) alles eingebaut, was frau von einer Programmiersprache erwartet: Variablen, konditionale Abfragen, Schleifen.

1 Variablen

Eine Variable in der Bash zu setzen ist ganz einfach: Frau denkt sich einen Namen für sie aus und setzt hinter ein Gleichheitszeichen den Wert. Um auf ihren Inhalt zuzugreifen, schreibt frau ein Dollarzeichen vor den Variablennamen. Das Kommando `echo`, ein *Shell-Builtin* (also ein eingebauter Befehl), nutzt frau, um den Wert einer Variablen auszugeben:

```
bash$ variable=wert
bash$ echo $variable
wert
```

Da Leerzeichen von der Shell als Worttrennzeichen benutzt werden, müssen sie geschützt werden, wenn sie Bestandteil einer Zeichenkette sind, die einer Variable zugewiesen wird. Dabei gibt es die „sanfte“ Möglichkeit, den gesamten String in doppelte Anführungszeichen (*Doppelquotes*) zu setzen, und die unnachgiebige Variante mit einfachen Anführungszeichen. Bei ersterer fischt der Dollar weiterhin Variableninhalte heraus, bei letzterer interpretiert ihn die Shell buchstäblich:

```
bash$ variable="wert2 $variable"
bash$ echo $variable
wert2 wert
bash$ variable='wert2 $variable'
bash$ echo $variable
wert2 $variable
```

✎ Warum reagiert die Shell jetzt auf den Befehl `variable=wert3 $variable` mit `bash: wert2: command not found`?

(Musterlösung im Anhang)

✎ Warum schweigt die Shell auf die Befehlsfolge `unset variable; variable=wert3 $variable`, statt die oben genannte Fehlermeldung auszugeben?

(Musterlösung im Anhang)

Außerdem hat frau mit \ die Möglichkeit, einzelne Zeichen ihrer Sonderbedeutung zu berauben.

✎ Setz die Variable `informatica` auf den Wert `feminale` und gib ihren Inhalt so aus, dass auf dem Bildschirm die Worte `"echo $informatica" gibt feminale aus` (einschließlich der Anführungszeichen!) erscheinen!

(Musterlösung im Anhang)

Das Verhalten der Bash und anderer Programme wird nicht nur durch Umgebungsvariablen wie `HOME` oder `PATH`, die frau sich mit dem Kommando `env` anzeigen lassen kann, sondern auch durch *Shellvariablen* beeinflusst. Letztere werden von der Shell selbst genutzt und lassen sich mit dem Kommando `set` anzeigen. Darunter fallen zum Beispiel die Variable `IFS`, in der festgelegt ist, welche Zeichen als Worttrennzeichen fungieren, oder `PS1`, die die Eingabeaufforderung (den *Prompt*) festlegt.

✎ Wie ist der Prompt bei Dir definiert? Sieh in der Manpage zu `bash` nach, welche Platzhalter sich darin verwenden lassen. Ändere Deinen Prompt temporär in `aktuelles_verzeichnis(rechnername)$`!

(Musterlösung im Anhang)

Kindprozesse der Shell erben ihr *Environment*, ihre Umgebung. Variablen, die einfach nur so auf der Kommandozeile gesetzt wurden, gehören nicht da hinein und werden daher mit ihren Inhalten auch nicht an Kindprozesse weitergegeben.

Will frau dafür sorgen, dass eine Variable mit dem aktuellen Inhalt auch auf Kindprozesse übergeht, muss sie sie mit dem Befehl `export` *exportieren*:

```
bash$ variable=wert   # Setzen der Variablen in der aktuellen Shell
bash$ echo $variable
wert
bash$ bash            # Starten einer Kindshell
bash$ echo $variable  # variable wurde nicht an Kindshell exportiert
bash$ exit            # Ausloggen aus der Kindshell
exit
bash$ echo $variable  # zurueck in der Ausgangsshell
wert
bash$ export variable # Exportieren von variable
bash$ bash            # Neue Kindshell
bash$ echo $variable  # Kindshell kennt exportierte variable
wert
bash$ exit
exit
```

2 for-Schleife

Es kommt recht häufig vor, dass frau mehrere Dateien gleichartig bearbeiten will. Einfache Kommandos nehmen oft mehrere Dateien als Argumente an, doch wenn die Aufgabe sich nicht mehr mit einem einzigen Kommando lösen lässt, wird es schwierig. Es sei denn, frau besinnt sich auf die `for`-Schleife:

```
for i in *.HTM; do mv $i `basename $i HTM`html; done
```

Die Shell schaut nach, auf welche Dateien im aktuellen Verzeichnis das Wildcard-Muster `*.HTM` passt und legt deren Dateinamen einen nach dem anderen (d. h. bei jedem Schleifendurchlauf einen) in der Laufvariablen `i` ab.

Das war der Schleifenkopf und damit das erste Kommando. Will frau mehrere Kommandos hintereinanderweg auf der Kommandozeile schreiben, muss sie diese mit Semikola trennen. Das zweite Kommando wird eingeleitet vom Schleifen-Schlüsselwort `do`, das den Beginn des Schleifenrumpfes markiert. In jedem Schleifendurchlauf soll also der Befehl `mv $i `basename $i HTM`html` ausgeführt werden.

Was soll da umbenannt werden? Offensichtlich die Datei, deren Name gerade in der Variablen `i` steht. Der neue Name endet auf `html`, doch was ist das, was zwischen den *Backticks* ` ` steht?

Wenn wir es einzeln anschauen, sieht es gar nicht mehr so fürchterlich aus: `basename $i HTM`. Das Kommando `basename` findet den einfachen Dateinamen seiner Argumentdatei heraus, wobei es sämtliche Pfadangaben wegstreicht. Der Basename von `/etc/passwd` ist zum Beispiel `passwd`.

Gibt frau `basename` ein weiteres Argument mit auf den Weg, interpretiert das Kommando dies als Dateinamenendung, die es ebenfalls wegzustreichen gilt. Im Beispiel hackt `basename` also die Endung `HTM` von der Datei in `i` ab, um anschließend eine neue Endung `html` anhängen zu können. Die Backticks sorgen lediglich dafür, dass `basename` zur Sache kommt, bevor das `mv`-Kommando ausgeführt wird.

Zu guter Letzt gilt es, den Schleifenrumpf mit `done` abzuschließen.

Natürlich muss niemand all diese Kommandos auf eine Zeile schreiben. Tippt frau in der Shell

```
for i in *.HTM
```

ein, erscheint ein sogenannter *Second-Level-Prompt*, den die Umgebungsvariable `PS2` bestimmt:

```
$  echo $PS2
>
```

Dieser erinnert daran, dass das Kommando noch unvollständig ist. Hinter diesem Prompt können wir weiterschreiben

```
> do mv $i $(basename $i HTM)html
> done
$
```

und sparen uns so die Semikola. Außerdem sehen wir hier eine alternative – und wie ich finde, intuitivere und weniger fehleranfällige – Schreibweise für die Backticks: Das zuvor auszuführende Kommando schreiben wir in eine runde Klammer und lesen sozusagen den Inhalt dieses Befehls wie bei einer Variablen mit `$` aus. Schreiben wir

```
#!/bin/bash

for i in *.HTM
do
mv $i $(basename $i HTM)html
done
```

in eine Datei, der wir Ausführbarkeitsrechte verleihen, haben wir ein kleines Programm, ein *Shellskript*, das sich wiederholt ausführen lässt. Fehlt das x-Bit, lassen sich Shellskripte auch ausführen, indem man sie in der Form `sh` *`skriptname`* aufruft.

Die erste Zeile des Skripts – und das gilt nicht nur Shellskripten – ist einem speziellen Kommentar (dem sogenannten *She-Bang*) vorbehalten, der besagt, welcher *Interpreter* den Rest ausführen soll – hier also `/bin/bash`. Natürlich kann frau ein Shellskript auch mit „normalen“ Kommentaren versehen, die der Dokumentation dienen. Einfach ein `#` davor, und schon kümmert sich die Shell nicht mehr um den Rest der Zeile.

3 Konditionale Abfrage

Die Shell kennt selbstverständlich auch eine `if-then-else`-Konstruktion:

```
if [ $PWD != $HOME ]; then echo "Nicht zu Hause"; else echo "Zu Hause"; fi
```

Diese Schreibweise, bei der frau die Bedingung in eckigen Klammern schreibt, ist recht fehlerträchtig, denn zwischen Bedingung und Klammer muss zwingend ein Leerzeichen stehen.[1] Deshalb empfiehlt sich die ungewöhnlichere Variante mit `test`:[2]

```
if test $PWD != $HOME; then echo "Nicht zu Hause"; else echo "Zu Hause"; fi
```

Mit den Operatoren `!=`, `=`, `<` und `>` lassen sich übrigens nur Zeichenketten vergleichen. Um Integerwerte (Gleitkommazahlen kennt die Shell nicht) zu vergleichen, kommen die Operatoren `-eq`, `-ne`, `-lt`, `-le`, `-gt` und `-ge` zum Einsatz:

[1] `/usr/bin/[` gibt es übrigens nicht nur als Shell-Builtin, sondern auch als eigenen Befehl, zu dem es meist eine eigene Manpage gibt.

[2] Auf vielen Systemen ist `[` ein Symlink auf `test`.

```
#!/bin/sh

NAME=$1
GREP=/bin/grep

$GREP $1 /etc/passwd

if test $? -eq 0
    then echo "Die Nutzerin $NAME existiert auf diesem System."
    else echo "Die Nutzerin $NAME existiert nicht auf diesem System."
fi
```

Dieses Beispiel zeigt nebenbei noch zweierlei: zum einen, wie frau Argumente aus dem Kommandozeilenaufruf des Skripts ausliest, und zum anderen, wie frau den Rückgabewert des letzten Kommandos (exakt: der zuletzt ausgeführten Pipe) ermittelt. Beides geht über fest definierte Sondervariablen: In der Variablen `0` steht jeweils der Name des Skripts, in `1` der darauf folgende Parameter usw. (bis `9`). Mit Hilfe der Sondervariablen `?` können wir den besagten Returncode auslesen.

✎ Wie rufst Du dieses Skript auf der Kommandozeile auf?

(Musterlösung im Anhang)

Mit `test` lässt sich übrigens auch die Existenz von Dateien abfragen: `test -d` schaut nach, ob ein entsprechendes Verzeichnis existiert, `test -f`, ob es eine entsprechende „normale" Datei gibt, und `test -x`, ob die Datei existiert und ausführbar ist.[3]

✎ Schreib ein kleines Skript, das mit Hilfe von `wc` herausfindet, wieviele Zeilen jede einzelne Datei in Deinem Homeverzeichnis lang ist.

(Musterlösung im Anhang)

---

4 cut

In Shellskripten lassen sich grundsätzlich alle Kommandozeilenbefehle einsetzen, wobei eine Beschränkung auf Standard-Unixtools wie die in diesem Kurs vorgestellten dann angeraten ist, wenn das Skript auf verschiedenen Installationen laufen soll. Soll es auch auf anderen Unixsystemen ausgeführt werden, gilt es zudem, die verwendeten Befehlsoptionen zu überprüfen.

[3] Weitere Möglichkeiten siehe Manpage.

Ein in Shellskripten sehr nützlicher Befehl ist `cut`. Mit ihm kann frau aus dem Text einer Datei oder von der Standardeingabe Spalten extrahieren. Ob dies byte- bzw. zeichengenau oder anhand von Feldern, die von *Delimitern* (Spaltentrennern) begrenzt werden, geschieht, hängt von der angegebenen Option ab. Ohne explizite Angabe eines Spaltentrenners dienen Tabulatoren als Feldbegrenzer.

```
cut -d ":" -f 1,5 /etc/passwd
```

beispielsweise sucht aus der Passwort-Datei alle auf diesem Rechner vorhandenen Accounts und den dazugehörigen Kommentar heraus.

✎ Schreib ein Shellskript, das für alle Benutzerinnen Deines Rechners eine namentliche Begrüßung ausgibt.

(Musterlösung im Anhang)

✎ Was macht das Kommando `cut -b 6-17`? Was könnte der Unterschied zu `cut -c 6-17` sein?

(Musterlösung im Anhang)

Lösungen

✎ Warum reagiert die Shell jetzt auf den Befehl `variable=wert3 $variable` mit `bash: wert2: command not found`?

Weil die Shell Whitespace (hier ein Leerzeichen) als Trennzeichen zwischen Kommandos und Argumenten interpretiert. Im obigen Fall ersetzt sie `$variable` durch `wert2` und führt dann den Befehl `variable=wert3 wert2` aus. Ein Kommando des Namens `wert2` gibt es aber nicht.

Das hier demonstrierte Verhalten kann frau nutzen, um beispielsweise eine Anwendung in einer Sprache aufzurufen, die nicht der voreingestellten *Locale* entspricht:

```
$ echo $LANG
de_DE.UTF-8
$ LANG=en konqueror&  # ein englischer Konqueror öffnet sich
$ konqueror&  # ein deutscher Konqueror öffnet sich
```

✎ Warum schweigt die Shell auf die Befehlsfolge `unset variable; variable=wert3 $variable`, statt die oben genannte Fehlermeldung auszugeben?

`unset variable` tilgt die genannte Variable aus dem „Gedächtnis“ der Shell – sie gibt es nun nicht mehr. `$variable` liefert als Wert demzufolge nichts, die Shell soll also gar keinen Befehl ausführen, sondern setzt einfach nur die neue Variable `variable` auf den String `wert3`. `echo $variable` gäbe nun wert3 aus.

Mit dem Semikolon erlaubt es die Shell, mehrere, hintereinander auszuführende Befehle in eine Zeile zu schreiben.

✎ Setz die Variable `informatica` auf den Wert `feminale` und gib ihren Inhalt so aus, dass auf dem Bildschirm die Worte `¨echo $informatica¨ gibt feminale aus` (einschließlich der Anführungszeichen!) erscheinen!

Vier Lösungsbeispiele:

```
$ informatica=feminale
$ echo '"echo $informatica" gibt '$informatica" aus"
"echo $informatica" gibt feminale aus
$ echo '"echo $informatica" gibt '$informatica' aus'
"echo $informatica" gibt feminale aus
$ echo '"echo $informatica" gibt '"$informatica aus"
"echo $informatica" gibt feminale aus
$ echo "\"echo \$informatica\" gibt $informatica aus"
"echo $informatica" gibt feminale aus
```

✎ Wie ist der Prompt bei Dir definiert? Sieh in der Manpage zu `bash` nach, welche Platzhalter sich darin verwenden lassen. Ändere Deinen Prompt temporär in `aktuelles_verzeichnis(rechnername)$`!

```
trish@lillagroenn:~$ echo $PS1
\u@\h:\w\$
$ export PS1="\w(\h)$"
~(lillagroenn)$
```

Achtet darauf, dass die Gänsefüßchen sehr wichtig sind – ohne sie schnappt sich die Shell die Backslashes und die runden Klammern und interpretiert sie in der eigenen Sonderbedeutung, und zwar *vor* der Zuweisung des Wertes zur Variablen.

✎ Wie rufst Du das Skript

```
#!/bin/sh

NAME=$1
GREP=/bin/grep

$GREP $1 /etc/passwd

if test $? -eq 0
    then echo "Die Nutzerin $NAME existiert auf diesem System."
    else echo "Die Nutzerin $NAME existiert nicht auf diesem System."
fi
```

auf der Kommandozeile auf?

Nehmen wir an, wir haben es in einer Datei abgelegt, die `exists` heißt und sich im aktuellen Arbeitsverzeichnis befindet. Um herauszufinden, ob die Nutzerin `trish` existiert, rufen wir dann `sh exists trish` auf. Haben wir `exists` mit `chmod` Ausführbarkeitsrechte gegeben, geht auch `./exists trish`.

Das Skript funktioniert übrigens nicht, wenn NIS zum Einsatz kommt.

✎ Schreib ein kleines Skript, das mit Hilfe von `wc` herausfindet, wieviele Zeilen jede einzelne Datei in Deinem Homeverzeichnis lang ist.

```
for datei in $(ls -a ~); do wc -l ~/$datei; done
```

Diese Lösung hat den Nachteil, dass `wc` natürlich nicht die Zeilen eines Verzeichnisses zählen kann und deshalb eine Fehlermeldung der Art `wc: xyz: Is a directory` auf der

Standardfehlerausgabe ausgibt, wenn es auf Verzeichnisse stößt. Diese Meldung stört eindeutig. Wir können sie zwar ins Nirvana umleiten …

```
for datei in $(ls -a ~); do wc -l ~/$datei 2> /dev/null; done
```

…, aber auch jetzt werden Verzeichnisse weiterhin mit der Länge Null aufgelistet. Elegant wäre es, `wc` nur dann von der Leine zu lassen, wenn wir sicher sind, es handelt sich um eine reguläre Datei (`-f`):

```
for datei in $(ls -a ~); do if [ -f ~/$datei ]; then wc -l ~/$datei; fi;
done
```

Eine ganz andere Alternative wäre:

```
find ~ -maxdepth 1 -type f -exec wc -l {} \;
```

Die Option `-maxdepth 1` sorgt dafür, dass `find` sich nicht in Unterverzeichnissen des Homeverzeichnisses verliert.

Es gibt übrigens einen Unterschied zwischen beiden Lösungen: Die `for`-Lösung schließt symbolische Links mit ein. Die Hausaufgabe zur Hausaufgabe: Wie verhindere ich das?

Zudem sei darauf hingewiesen, dass frau sich die Wildcardmöglichkeiten der Shell zu Nutze machen kann, wenn sie eine `for`-Schleife über den (Teil-)Inhalt eines Verzeichnisses laufen lassen will:

```
for datei in ~/*; do echo $datei; done
```

Beachte dabei aber, dass `*` nicht auf Dateien passt, die mit einem Punkt beginnen.

✎ Schreib ein Shellskript, das für alle Benutzerinnen Deines Rechners eine namentliche Begrüßung ausgibt.

```
for name in `cut -d ":" -f 1 /etc/passwd`; do echo "Hallo $name"'!'; done
```

Um das Ausrufezeichen auszugeben ist ein Trick nötig, damit die Shell nicht versucht, es gemäß einer Sonderfunktion zu behandeln: Innerhalb einfacher Anführungszeichen interpretiert die Shell es nicht weiter. `$name` können wir hingegen nicht in einfache Gänsefüßchen setzen, denn wir wollen ja gerade, dass statt der Variablen ihr Inhalt erscheint.

✎ Was macht das Kommando `cut -b 6-17`? Was könnte der Unterschied zu `cut -c 6-17` sein?

`cut -b 6-17` schneidet das sechste bis 17. Byte heraus, die Option `-c` hingegen das sechste bis 17. Zeichen. Das macht keinen Unterschied bei Zeichensätzen, in denen ein Character in einem Byte kodiert ist. Wenn allerdings Unicode o. a. ins Spiel kommen, klappt diese Gleichsetzung nicht mehr.

Dieses Skript basiert passagenweise auf der für die Informatica Feminale erarbeiteten Kursunterlage „Linux ist weiblich“ in deren 5. Auflage. Die Autorin bedankt sich bei Sibylle Nägle und Gabriele Pohl für die Mitarbeit an der 1. Auflage (2001) bzw. der 5. Auflage (2005).

Satz: TEX/LATEX unter Linux