

Einführung in die Shell-Programmierung

Contents

1 Einführung in die Shell-Programmierung

Dieses Kapitel widmet sich der Einführung in die Shell-Programmierung. Hierbei handelt es sich keinesfalls um eine vollständige Dokumentation. Für weiterführende Details empfehle ich dringend einen Blick in die `man` Pages der jeweiligen Programme sowie der Bash (`man bash`) zu werfen.

Allgemein =====

Was ist die Shell? Die Shell ist ein Interpreter, der Kommandos entsprechend seiner eigenen Syntax interaktiv oder selbstständig ausführt. Was im Folgenden als Shell bezeichnet wird, ist ein Metaprogramm dessen Hauptaufgabe es ist, weitere Programme zu laden. Die Shell stellt die Schnittstelle zwischen Benutzer und Betriebssystem dar. In der UNIX-Welt hat sie den Status eines Benutzerprogramms und kann deshalb nach Belieben ausgetauscht werden.

Im folgenden die wichtigsten Vertreter:

sh:	Bourne Shell	Die Mutter aller Shells
csh:	C-Shell	Shell mit C-ähnlicher Syntax
ksh:	Korn Shell	Mächtige, C-orientierte Shell (Solaris)
zsh:	Z-Shell	Erweiterte, komfortable Shell, Bash kompatibel

```
bash:    Bourne Again      Erweiterte, komfortable Bourne Shell
        SH
```

In diesem Kapitel wollen wir uns mit der Bash beschäftigen. Sie ist der Standard Kommandointerpreter unter LINUX. Die Bash ist kompatibel zur Standard-Bourne-Shell welche von Steven R. Bourne fuer AT&T Unix entwickelt wurde.

2 Erste Schritte

Im Grunde ist ein Shellsript nichts anderes als eine Textdatei, in der Befehlsfolgen enthalten sind. Diese Befehlsfolgen können mit Hilfe von Schleifen und Variablen gesteuert werden. Man kann solche Befehlsfolgen auch direkt in der Shell eingeben. Denkbar waeren zum Beispiel Folgendes:

```
asterix% echo "Hallo Europa";echo "Hallo Osterhase..."
Hallo Europa
Hallo Osterhase...
asterix%
```

Oder aber man schreibt ein kleines Script, daß man dann jederzeit wieder aufrufen kann. Zu diesem Zwecke öffne man einen Editor seiner Wahl (vi, emacs, nedit, ...) und gebe folgendes ein:

```
#!/bin/sh
# Das ist ein Kommentar
echo "Hallo Europa"
echo "Hallo Osterhase..."
```

Danach die Datei unter einem selbstgewaehlten Namen abspeichern und ausfuehrbar machen. Danach kann das Script gestartet werden.

```
asterix% chmod 744 meine_datei.sh
asterix% ./meine_datei.sh
Hallo
Hallo Osterhase...
asterix%
```

Eine Shell-Skript beginnt mit der Angabe des Kommandointerpreters. Zeile 1 ist also der Pfad zu dem Programm, das die folgenden Zeilen interpretieren kann. Hier wird auf die Bourne Shell (sh) verwiesen. Die Datei `/bin/sh` ist in unserem Fall ein Symlink auf die Bash.

Kommentare werden zeilenweise mit `#` gekennzeichnet. Das Shellkommando `echo` existiert in zwei Formen. Zum einen ist es ein in der Shell enthaltendes Kommando, zum anderen ist es aber auch eine Datei, welche sich im Verzeichnis `/bin` befindet. Linux bietet eine Vielzahl von Tools und Kommandos, die das Arbeiten auf der Textkonsole ermöglichen und erleichtern. Doch dazu später mehr.

3 Variablen und Quoting

Wie in jeder Programmiersprache gibt es auch in der Shell Variablen. Die Bash unterscheidet bei Variablen nicht nach Typen. Grundsätzlich wird jede Variable erst einmal als String aufgefaßt. Je nach Kontext kann sie dann auch als Integer interpretiert werden.

Hier nun einige Beispiele, die den Umgang erläutern sollen:

```
#!/bin/sh
# Wertzuweisung
Variable_1=10
Variable_2="Der Mond ist ein grüner Käse."
Variable_3="A B C D"
Variable_4=$(hostname) # alte Schreibweise: Variable_4=`hostname`
# Wertabfrage
echo \ $Variable_1 = $Variable_1
echo "\ $Variable_2 = $Variable_2"
echo '$Variable_1 + $Variable_2' = ${Variable_1}${Variable_2}
echo $Variable_3
echo "$Variable_3"
echo $Variable_4
```

Ergibt:

```

$Variable_1 = 10
$Variable_2 = Der Mond ist ein gruener Kaese.
$Variable_1 + $Variable_2 = 10Der Mond ist ein gruener Kaese.
A B C D
A B  C   D
asterix

```

Um \$ \ " auf dem Bildschirm darzustellen zu können, muessen sie wie folgt maskiert werden: \\$ \\ \".

In Zeile 6 wird die Shell angewiesen, erst den rechten Teil der Variablenzuweisung auszufuehren, naemlich den Befehl `hostname`. Dies wird durch `$(...)` oder "Backquotes" erreicht . Die Variable wird also mit dem Hostnamen des jeweiligen Rechners belegt.

Zu lange Befehlszeilen können mit einem \ getrennt werden.

```

#!/bin/sh
echo "Dies ist ein wahnsinnig langer, sinnfreier und \
    unglaublich ueberfluessiger Satz."

```

3.1 Spezielle Typen von Variablen

local variables Diese sind nur innerhalb eines Code Blocks (`{...}`) oder einer Funktion gueltig und werden mit `local Variable` definiert.

environmental variables Sie bestimmen das Erscheinungsbild der Shell und dienen der Anpassung ihres Verhaltens in der Systemumgebung. Die Prozessumgebung wird vom Elternprozeß an seine Kinder vererbt. D.h. alle Umgebungsvariablen der Bash werden an die von ihr gestarteten Prozesse weitergegeben. Die Kommandos `printenv` und `export` geben alle Umgebungsvariablen aus. Mit dem `set` Kommando könne Umgebungsvariablen gesetzt werden.

3.2 Quotierung

Quotierung wird benutzt, um die spezielle Bedeutung von Kontrollzeichen, reservierten Wörtern und Namen auszuschalten. Es gibt 3 Formen:

Fluchtsymbol `\` Es entwertet das unmittelbar folgende Sonderzeichen. Ein durch `\` entwertetes Zeilenende wird ignoriert.

Hochkomma ' (Quote) Von Hochkomma eingeschlossene Worte werden von der Shell nicht weiter bearbeitet. Allerdings darf ein Hochkomma nicht in Hochkommas eingeschlossen sein. Auch nicht, wenn es durch `\` maskiert ist.

Anfuhrungszeichen " (Doublequotes) Von in Anfuhrungszeichen eingeschlossenen Wörtern erkennt die Shell nur die Sonderzeichen `$` `'` `\` als solche. Das Fluchtsymbol behält seine Bedeutung fuer die Zeichen `$` `'` `"` `\` oder dem Zeilenende.

4 Arrays

Die Bash unterstuetzen auch eindimensionale Arrays. Diese können mit `declare -a Variable` initialisiert werden, muessen aber nicht. Einzelnen Elemente des Arrays werden mit `${variable[xx]}` angesprochen.

```
#!/bin/bash
array=( zero one two three four five )
array[6]="Dieser Text ist ein Element des Arrays"

echo ${array[0]}      # zero
echo ${array[1]}      # one
echo ${array:0}       # zero
                      # Parametererweiterung, erstes Element.
echo ${array:1}       # ero
                      # Parametererweiterung, erstes Element,
                      # Start an Position #1 (2. Buchstabe).
echo ${array[1]:1}    # ne
                      # Parametererweiterung, zweites Element,
                      # Start an Position #1 (2. Buchstabe).
echo ${#array[2]}     # 3
                      # Laenge des dritten Elements.
element_count=${#array[@]} # oder
element_count=${#array[*]} # Anzahl der Elemente: 7
```

5 Kommandosubstitution

Die Kommandosubstitution erlaubt es, die Ausgabe eines Kommandos direkt an eine Variable zu uebergeben. Zwei Formen sind möglich:

`$(Kommando)` oder `'Kommando'`

Bei der Substitution mit Backquotes (nicht Hochkomma!) muessen Sonderzeichen maskiert werden. Bei der Klammervariante bleiben alle Zeichen unveraendert.

Damit laeßt sich z.B. der Inhalt eines Textfiles in einen Array laden.

```
#!/bin/bash
filename=sample_file
# cat sample_file
#
# 1 a b c
# 2 d e fg
declare -a array1
array1=($(cat "$filename" | tr '\n' ' '))
# Loads contents
# of $filename into array1.
# list file to stdout.
# change linefeeds in file to spaces.
echo ${array1[@]}
# List the array:
# 1 a b c 2 d e fg
#
# Each whitespace-separated "word" in the file
#+ has been assigned to an element of the array.
element_count=${#array1[*]}
echo $element_count          # 8
```

6 1.6 Klammererweiterung

Die Klammererweiterung erzeugt aus einer, in geschweiften Klammern eingeschlossenen, Liste von Bausteinen mehrere Zeichenketten. Zum Beispiel

erzeugt der Befehl:

```
mkdir ~/ {dir1,dir2}
```

die Verzeichnisse dir1 und dir2 im Homeverzeichnis. Der Befehl

```
mkdir ~/ {dir1,dir2} {1,2,3}
```

die Verzeichnisse dir11, dir12, dir13, dir21, dir22 und dir23.

7 Parameter

7.1 Positionsparameter

Einem Shellskript können beim Aufruf auch Parameter mitgegeben werden.

```
COMMAND Parameter1 Parameter2 ...
```

Diese Parameter lassen sich im Script mit `$1`, `$2`, ... abfragen. Ab dem 10. uebergebenen Parameter muessen geschweifte Klammern gesetzt werden (`{10}`).

Mit dem `shift`-Kommando ist es möglich, die Positionsparameter nach links zu verschieben. Das bedeutet, daß der zweite Parameter der Erste wird, der Dritte der Zweite usw. . `$0` (der Scriptname) bleibt unberuehrt. Dieses Kommando macht zum Beispiel bei Funktionen Sinn:

```
#!/bin/bash
multiply ()          # multipliziert die uebergebenen Parameter
{                   # Anzahl der Parameter ist variabel
    local product=1
    until [-z "$1" ] # Until nutzt den ersten uebergebenen Parameter ...
    do
        let "product *= $1"
        shift
    done
    echo $product      # wird nicht auf STDOUT ausgegeben,
}                     # wenn es an eine Variable uebergeben wird
mult1=15383; mult2=25211
val1=`multiply $mult1 $mult2`
echo "$mult1 X $mult2 = $val1"
                        # 387820813
```

7.2 Spezialparameter

\$*	bezeichnet alle Positionsparameter von 1 an. In Anführungszeichen gesetzt, steht “\$*” fuer
\$@	bezeichnet alle Positionsparameter von 1 an. In Anführungszeichen gesetzt, wird es durch d
\$#	Anzahl der Positionsparameter
\$?	Rueckgabewert (Status) des zuletzt ausgefuehrten Kommandos
\$-	steht fuer die Optionsflags (von set oder aus der Kommandozeile).
\$\$	Prozessnummer der Shell
\$_	Prozessnummer des zuletzt im Hintergrund aufgerufenen Kommandos
\$0	Name des Shellskripts
_	letztes Argument des zuletzt ausgefuehrten Kommandos

8 Parametererweiterung

`${Parameter}` Laßt sich ein Variablenname nicht eindeutig von den darauffolgenden Zeichen trennen, oder besteht ein Positionsparameter aus mehr als einer Ziffer, muß dieser Parameter in geschweifte Klammern gesetzt werden.

Die folgenden Konstruktionen stellen verschiedene Arten bedingter Parametererweiterung dar. Enthält die Konstruktion einen Doppelpunkt, so wird der Parameter darauf hin getestet, ob er leer oder ungesetzt ist. Wird der Doppelpunkt in diesen Konstruktionen weggelassen, wird nur darauf getestet, ob er gesetzt (auch leer!) oder ungesetzt ist.

Parametererweiterungen eignen sich z.B. fuer die Defaultwertzuweisung bei Variablen. Sie dürfen nur als Bestandteil eines Kommandos oder einer Zuweisung durchgeführt werden. Soll eine Parametererweiterung als einzelnes Kommando stehen, beispielsweise bei einer Fehlermeldung, dann muß die Zeile mit einem Doppelpunkt begonnen werden.

`${Parameter:-default}**`** Wenn der *Parameter* ungesetzt oder leer ist, wird *default* anstelle des gesamten Ausdrucks eingesetzt.

`${Parameter:=default}**`** Wenn der *Parameter* ungesetzt oder leer ist, wird der Inhalt von *default* dem Parameter zugewiesen und der

neue *Parameter* eingesetzt. Positionsparametern und Spezialparametern kann allerdings auch auf diese Weise kein Wert zugewiesen werden.

`${Parameter:?err_msg}**`** gibt eine Fehlermeldung wenn der *Parameter* leer oder ungesetzt ist. *err_msg* wird als Fehlermeldung auf STDERR ausgegeben. Ist der *Parameter* gueltig gesetzt, wird sein Inhalt eingesetzt.

`${Parameter:+alt_value}**`** erzwingt die Benutzung eines anderen Wertes. Wenn der *Parameter* weder leer, noch ungesetzt ist, wird der Inhalt von *alt_value* eingesetzt. Sonst wird nichts eingesetzt.

`${Parameter:Offset:Laenge}**`** Hier wird *Parameter*, von *Offset* an, mit der *Laenge* *Laenge* neu gesetzt.

`${#Parameter}**`** gibt die Anzahl der Zeichen im Parameter wieder.

`${var#Pattern}` und **`${var##Pattern}**`** entfernt den uebereinstimmenden Teil von *Pattern* in *var* beginnend von links. Bei *#* wird das kuerzeste treffende Stueck entfernt, bei *##* das Laengste.**

`${var%Pattern}` und **`${var%%Pattern}**`** entfernt den uebereinstimmenden Teil von *Pattern* in *var* beginnend von rechts. Bei *%* wird das kuerzeste treffende Stueck entfernt, bei *%%* das Laengste.**

`${var/Pattern/Replacement}` und **`${var/Pattern//Replacement}**`**
Das größte auf *Pattern* passende Stueck in *var* wird durch *Replacement* ersetzt. Bei */* wird einmal ersetzt, bei *//* wird jede auftretende uebereinstimmung ersetzt. Werden Positionsparameter oder Arrays uebergeben, wird das Kommando auf jeder einzelnen Parameter bzw. jedes Feld angewandt.**

9 `!/bin/bash`

```
leer= default="voll" string="1234567890" array=( zero one two three
four five )
```

```
echo leer-default} # gibt nichts aus, denn $leer ist definiert
echo undef-default} # gibt "voll" aus, denn # $undef ist nicht definiert
echo leer : -default} # gibt "voll" aus (:)
```

```

default_filename=generic.data ${1:j'Dateiname wird auf
generic.data gesetzt."} # Fehlermeldung, wenn $1 fehlt

filename=1 :=default_filename} # setzen des Parameters

leer=leer : +default} # sollte leer nicht NULL sein, # wird er mit "voll"
belegt

echo ${string:0:1} # von links beginnend mit 0 und einem Zeichen: 1
echo ${string:(-3):2} # von rechts und 2 Zeichen: 89

laeng_string=${#string} # ergibt 10 echo arrayLaengedeserstenElements :
4elementcount =#{#array[@]} # oder element_count=${#array[*]} #
Anzahl der Elemente: 6

var1=abcd12345abc6789 pattern1=ac # (wildcard) trifft alles zwischen
'a' und 'c' pattern2=b*9 # alles zwischen 'b' und '9' echo var1pattern1}
# d12345abc6789 echo var1pattern1} # 6789 echo var1

echo ${pattern1/abc/ABC} # "abcd12345abc6789" -> "ABCd12345abc6789"
echo ${pattern1//abc/ABC} # "abcd12345abc6789" -> "ABCd12345ABC6789"

```

10 Bedingte Ausfuehrung

Wie in jeder Programmiersprache, können Kommandos auch miteinander verknuepft werden.

COMMAND1 && COMMAND2 stellen eine logische UND-Verknuepfung dar. Wurde Kommando1 fehlerfrei ausgefuehrt (exit status 0 heißt Abarbeitung ohne Fehler), wird auch Kommando2 ausgefuehrt.

COMMAND1 || COMMAND2 Stellen eine logische ODER-Verknuepfung dar. Kommando2 wird nur ausgefuehrt, wenn bei Kommando1 ein Fehler aufgetreten ist.

11 Tests, Verzweigungen und Schleifen

11.1 Statement `if ... then`

Syntax: `if Liste then Liste [elif Liste then Liste...][else
Liste] fi`

If `... then` Konstruktionen ueberpruefen, ob der Exit-Status einer Liste von Kommandos 0 ist. Ist dies der Fall, werden weitere, entsprechend definierte, Kommandos ausgefuehrt.

Im folgenden Beispiel wird mit dem Kommando **grep** in einer Textdatei nach Zeilen, die das Wort "Bash" enthalten, gesucht. Existieren solche Zeilen, gibt **grep** als Exit-Status 0 (= true) aus. Das bedeutet, die folgenden Kommandos werden ausgefuehrt.

```
#!/bin/sh
if grep Bash file.txt
then echo "File contains at least one occurrence of Bash."
fi
```

Es existiert auch ein verwandtes Kommando: `[...]`. Dieser Ausdruck ist ein Synonym fuer das Bash-Kommando **test**. Es existiert außerdem ein externes Kommando `/usr/bin/test`.

```
if [condition1 ]
then
    command1
    command2
    command3
elif [condition2 ]
# Same as else if
then
    command4
    command5
else
    default-command
fi
```

Weitere Informationen zum Kommando `test` bzw. [...] entnehmen Sie bitte der jeweiligen `man`-Page.

11.2 Statement for ... do

Syntax: `for Name [in Wort] do Liste done`

Mit `Name` wird eine Shellvariable definiert, die in jedem Schleifendurchlauf einen neuen Wert erhält. Die Werte werden normalerweise mit dem Schlüsselwort `in` uebergeben. Wird der Teil `in Wort` weggelassen, wird die Liste fuer jeden gesetzten Parameter (img1.png) einmal ausgefuehrt.

```
#!/bin/sh
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus
do
    echo $planet
done
# oder aber auch
NUMBERS="9 7 3 8 37.53"
for number in `echo $NUMBERS` # for number in 9 7 3 8 37.53
do
    echo "$number "
done
```

11.3 Statement while und until

Syntax: `while Liste do Liste done`

Syntax: `until Liste do Liste done`

Der Schleifenkörper `do Liste done` wird so lange wiederholt , bis die in `while Liste` formulierte Bedingung falsch ist.

Die `until`-Schleife entspricht der `while`-Schleife mit dem Unterschied, daß der `do`-Teil so lange ausgefuehrt wird, wie das letzte Kommando der `until Liste` einen Status ungleich 0 liefert.

```
#!/bin/sh
var0=0
```

```
LIMIT=10

while ["$var0" -lt "$LIMIT" ]
do
    echo -n "$var0 "          # -n suppresses newline.
    var0=`expr $var0 + 1`    # var0=$(( $var0+1 )) also works.
done
```

11.4 case Statement

Syntax: `case Wort in [Muster [| Muster]) Liste ;; ...] esac`

Mit der `case`-Anweisung können Verzweigungen programmiert werden. `Wort` wird mit den angegebenen Mustern verglichen. Bei Übereinstimmung wird die `Liste` von Kommandos ausgeführt. In den Suchmustern können auch Wildcards und reguläre Ausdrücke verwendet werden.

```
#!/bin/sh
arch=$1
case $arch in
    i386 ) echo "80386-based machine";;
    i486 ) echo "80486-based machine";;
    i586 ) echo "Pentium-based machine";;
    i686 ) echo "Pentium2+-based machine";;
    *    ) echo "Other type of machine";;
esac
```

12 Arithmetik

Arithmetische Operationen werden über die Shellkommandos `expr` und `let` realisiert. Dabei ist `let` ein internes Kommando der Bash, und `expr` ein Externes. Es ist sowohl möglich Berechnungen mit Hilfe des Kommandoaufrufes zu machen, als auch durch eine verkürzte Schreibweise:

```
#!/bin/sh
z=`expr $z + 3` # Aufruf des externen Kommandos expr
```

```

let z=z+3    # Aufruf des internen Kommandos
let "z += 3" # Mit Quotes sind Leerzeichen und special operators erlaubt.
z=$((z+3))   # neue verkuerzte Schreibweise (ab Version 2.0)
z=${z+3}     # alte Schreibweise

```

Berechnungen finden, wie in C, mit “lange Ganzzahlwerten” statt. Eine ueberlaufkontrolle gibt es nicht. Division durch Null fuehrt zu einem Fehler, der aber mit Hilfe der `trap`-Shellfunktion abgefangen werden kann. Folgende Operatoren sind erlaubt (Prioritaetshirarchie):

<code>+ -</code>	Vorzeichen
<code>! ~</code>	logische und bitweise Negation
<code>* / %</code>	Multiplikation, Division, Modulo
<code>+ -</code>	Addition und Subtraktion
<code><< >></code>	bitweise links und rechts-Shift-Operation
<code><= >= <></code>	Vergleiche
<code>== !=</code>	gleich und ungleich
<code>&</code>	bitweise Addition
<code>~</code>	bitweise XOR
<code> </code>	bitweise ODER
<code>&&</code>	bitweise UND
<code> </code>	logisches ODER

13 Funktionen

Wie auch in C, kann man in der Bash einzelne Programmteile zu Funktionen zusammenfassen. Mit dem `local`-Shellkommando ist es möglich, lokale Variablen fuer Scriptfunktionen zu erzeugen. Mit `return` können Werte aus einer Funktion zurueckgegeben werden.

```

#!/bin/bash
myadd() {
    # $1 erstes Argument
    tmp=0
    args=$@
    for i in $args

```

```

        do
        tmp=`expr $tmp + $i`
        done
        return $tmp
}
# main
myadd 1 2 3 $VAR
RES=$?
myadd $RES 5 6 $VAR2
RES=$?

```

14 Ein-/Ausgabe-Umleitungen

Jedes Programm erhält beim Start drei offene "Datenkanäle":

Standard Input:	STDIN (0)
Standard Output:	STDOUT (1)
Standard Error Output:	STDERR (2)

Durch das Umlenken der Ein-/Ausgabekanäle können Dateien oder Dateisysteme zum Lesen bzw. Schreiben fuer Kommandos geöffnet werden. Es gibt viele Möglichkeiten Daten umzuleiten. Hier ist eine kleine Auswahl:

COMMAND<infile	Eingabeumlenkung
COMMAND>outfile	Ausgabeumlenkung
COMMAND»outfile	Ausgabeumlenkung, anhaengen
COMMAND 2>&1	STDERR mit auf STDOUT legen
COMMAND »EofListe Liste EofListe	Zeilen in <i>Liste</i> werden in COMMAND umgeleitet

15 Pipes

Bei einer Pipe wird der Standardausgabekanal eines Kommandos mit dem Standardeingabekanal eines anderen Kommandos zusammengelegt. Dabei werden beide Kommandos als separate Prozesse gleichzeitig gestartet.

Beim folgenden Beispiel wird der Inhalt der Datei `.zshrc` durch `cat` auf den Standardausabekanal geschrieben und an `grep` ueber den Standard-eingabekanal uebergeben. `grep` sucht nach allen Zeilen die das Wort “HISTSIZE” enthalten, und gibt diese aus.

```
asterix% cat .zshrc|grep -i HISTSIZE
export HISTSIZE=1000
```

16 Textmanipulationen

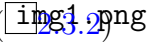
In diesem Kapitel soll es um das komplexe Thema “Suchen und Ersetzen” gehen. Im Weiteren wird auf die Grundlagen eingegangen und zwei der mächtigsten Tools auf diesem Gebiet kurz beleuchtet. Wer es wirklich genau wissen will, sollte sich im Internet nach weiterführenden Dokumentationen umsehen.

17 reguläre Ausdrücke

Es gibt in der UNIX-Welt einige sehr mächtige Tools (`sed`, `awk`, `grep`), die das Durchsuchen von Texten nach bestimmten Mustern ermöglichen. Um diese Tools effektiv nutzen zu können, ist es unbedingt notwendig, sich mit regulären Ausdrücken zu beschäftigen.

Reguläre Ausdrücke beschreiben eine nicht leere Menge von Zeichenfolgen, die aus Textzeichen (Buchstaben, Ziffern, Sonderzeichen) und/oder Metazeichen mit erweiterter Bedeutung bestehen. Textzeichen stehen für sich selbst, Metazeichen (Spezialzeichen) stellen Operatoren dar, mit deren Hilfe komplexe Textmuster beschrieben werden können. Als Begrenzung der Mustersuche gilt in den meisten Fällen (Tools) das Zeilenende. D.h. es ist nicht möglich reguläre Ausdrücke zu definieren, die über das Zeilenende hinaus prüfen.

17.1 Zeichenklassen

.	ist ein Platzhalter und bezeichnet jedes einzelne Zeichen außer das Zeilenende.
[abc\$]	trifft alle aufgeführten Zeichen.
a-c	bezeichnet alle Zeichen im angegebenen Limit.
[^exp]	trifft alle Zeichen außer den angegebenen.
^abc	trifft das angegebene Muster, wenn es am Zeilenanfang steht.
abc\$	trifft das angegebene Muster, wenn es am Zeilenende steht.
\	Maskierung des folgenden Zeichens ()

17.2 Wiederholungsoperatoren

*	trifft den vorangegangenen Ausdruck 0 oder mehrmals.
+	trifft den vorangegangenen Ausdruck ein oder mehrmals.
?	trifft den vorangegangenen Ausdruck 0 oder einmal.
	ist ein Trennzeichen. Trifft entweder den folgenden oder vorangegangenen Ausdruck.
(...)	bildet eine Gruppe von regulären Ausdrücken.

Die Syntax von **grep** und **egrep** variiert in manchen Punkten. Für +, ?, |, (...) ergibt sich für **grep** eine andere Schreibweise: \+, ?, |, \(...\).

Reguläre Ausdrücke werden von links nach rechts aufgelöst. Operatoren werden in der folgenden Reihenfolge abgearbeitet:

[...]   &Verknüpfung  &Verknüpfungen |

Abweichungen davon können mit Klammerung einzelner Ausdrücke erreicht werden.

Die Operatoren ?, +, *, ^, \$ und | können wiederum auch auf gruppierte (geklammerte) Ausdrücke angewendet werden. Beispielsweise trifft

(AB|CD+)?(EF)+

die Zeichenketten ABEF, CDEF, CDDEF, EF EF, EF EF EF usw.

Das hinter dem ersten Klammernpaar stehende Fragezeichen bedeutet, daß das Vorkommen, der darin enthaltenden Zeichenketten AB und CD, optional ist. Das Pluszeichen hinter D sagt aus, daß nach einem oder mehreren D's gesucht wird (CD, CDD, CDDDD, ...). Die so gefundenen Zeichenketten der ersten Klammer, müssen unmittelbar gefolgt sein von mindestens einem

Vorkommen der Zeichenkette EF. Das Plus bezieht sich hier auf den ganzen Klammersausdruck.

Wie aus obigen Regeln zu ersehen ist, handelt es sich bei regulären Ausdrücken um eine Wissenschaft. Die hier aufgezeigten Muster stellen nur einen Auszug dar. Noch ein paar Beispiele hinterher:

<code>.aus</code>	trifft Haus, raus, Maus, Laus,...
<code>xy*z</code>	trifft auf xy...was auch immer...z
<code>^abc</code>	jede Zeile, die mit abc beginnt
<code>abc\$</code>	jede Zeile, die mit abc endet
<code>*</code>	trifft jeden Stern
<code>[Mr]aus</code>	trifft Maus und raus
<code>[[abc]</code>	trifft [(muß am Anfang stehen), a, b, c
<code>[KT]?ELLER</code>	trifft ELLER, TELLER, KELLER
<code>[^a-zA-Z]</code>	schließt alle Buchstaben aus
<code>[0-9]\$</code>	trifft jede Zeile, die mit einer Zahl endet
<code>[0-9][0-9]</code>	trifft jede zweistellige Nummer
<code>H(e a)llo</code>	trifft Hallo und Hello
<code>(ab)?</code>	trifft entweder "ab" oder nichts ("ab" ist optional)
<code>^\$</code>	trifft alle Leerzeilen

Aber es ist noch mehr möglich.

<code>\{n,m\}</code>	trifft ein Muster mindestens n-mal und höchstens m-mal
<code>\<abc\></code>	trifft das eingeschlossene Muster nur, wenn es sich um ein separates Wort handelt
<code>\(abc\)</code>	Die Klammern fassen Ausdrücke zusammen. Jede Zeile wird nach angegebenen Mustern
<code>\n</code>	referenziert obige Muster

Es lassen sich des weiteren syntaktische Gruppen bilden. Hierbei handelt es sich nur um eine andere Schreibweise bereits besprochener Ausdrücke. Diese Schreibweise kann die Lesbarkeit regulärer Ausdrücke deutlich verbessern.

<code>[:alnum:]</code>	alle alphanumerischen Zeichen [A-Za-z0-9]
<code>[:alpha:]</code>	alle Buchstaben [A-Za-z]
<code>[:blank:]</code>	ein oder mehrere Leerzeichen und Tab
<code>[:cntrl:]</code>	alle Kontrollzeichen wie z.B. <newline>

[digit:]	alle dezimalen Zahlen [0-9]
[graph:]	alle druckbaren Zeichen (ASCII 33 - 126) ohne das Leerzeichen
[print:]	alle druckbaren Zeichen
[lower:]	alle Kleinbuchstaben [a-z]
[upper:]	alle Großbuchstaben [A-Z]
[space:]	Leerzeichen und horizontales Tab
[xdigit:]	alle hexadezimalen Zahlen [0-9A-Fa-f]

Reguläre Ausdrücke sind zwar allgemein gültig, jedoch ist der Funktionsumfang der einzelnen Tools nicht einheitlich.

	[...]	.	*	?	+	^	\$		()
grep	x	x	x			x	x		
egrep	x	x	x	x	x	x	x	x	x
sed	x	x	x			x	x	x	x
awk	x	x	x	x	x	x	x	x	x

18 [rep

Syntax: `grep [-CVbchilnsvwx] [-Anzahl] [-AB Anzahl] [[-e] Ausdruck | -f Datei] [Datei]`

grep durchsucht die angegebenen Dateien (oder die Daten aus der Standard-eingabe) nach einem Ausdruck und gibt die entsprechenden Zeilen aus. Der Status von `grep` ist 0, wenn der Ausdruck gefunden wurde und sonst 1.

Wieder ein paar einfache Beispiele:

Befehl	<code>cat file</code>	<code>grep b.*g file</code>	<code>grep b.*g. file</code>	<code>grep ggg* file</code>
Resultat	big	big	bigger	bigger
	bad bug	bad bug	boogy	
	bag	bag		
	bigger	bigger		
	boogy	boogy		

Stern und Punkt sind Sonderzeichen. Will man nach Mustern suchen, die den Punkt als literarisches Zeichen auffassen, so muß dieser maskiert werden.

```
ls | grep Name.ext
```

trifft auch Name0ext , NameBext , usw. .

```
ls | grep Name\.ext
```

trifft nur die Datei File mit dem Namen **Name.ext**. .

Wir wollen in einem Textfile alle Zeilen, die den Namen Fred Feuerstein und Fredericke Feuerstein enthalten. Das bedeutet der Teil “ericke” ist optional.

```
grep "Fred\((ericke\)\)? Feuerstein" textfile
```

Die Klammern bilden eine Gruppe. Das Fragezeichen bedeutet ein oder kein Vorkommen des vorherigen Musters.

Hier werden Klammern innerhalb anderer Klammern ausgeschlossen:

```
grep "([^(]*)")"
```

Trifft (hello) und (aksjdhaksj d ka) aber nicht x=(y+2(x+1)) .

Jetzt wollen wir nach sich wiederholenden Mustern suchen. Eine gutes Beispiel sind Telefonnummern. Wir suchen nach einer Vorwahl (3 Ziffern) und der Nummer (7 Ziffern), getrennt durch einen - , einem Leerzeichen oder garnicht.

```
grep "[0-9]\{3\}[-]\?[0-9]\{7\}" file
```

[0-9] steht für alle Zahlen, \{3\} besagt, daß sich das vorherige Muster 3 mal wiederholen soll. [-]\? repräsentiert die Auswahl des Trennzeichens (Leerzeichen, - oder garnichts).

Angenommen, wir suchen eine Zeile in der nur das Wort “Hallo” steht. Es ist zudem noch möglich, daß sich vor und/oder hinter “Hallo” Leerzeichen befinden. Eine Möglichkeit wäre folgendes

```
grep "^[:space:]*Hallo[:space:]*$" file
```

^ steht für den Zeilenanfang, \$ für das Zeilenende.

Manchmal ist es nötig, Zeilen zu suchen, in denen entweder das Eine oder das Andere steht.

```
grep "Ich habe \(Schröder\|Stoiber\) gewählt" file
```

\| entspricht einem logischen ODER.

Hat man einmal ein Muster in `\(...\)` definiert, kann man es mit `\Zahl` erneut einsetzen.

```
echo bla blub bla | grep '\(bla\).*\1'
```

19 sed Stream Editor

Syntax: `sed [-n] [-e Editorkommando] [-f Sriptdatei] [inputfile]`

sed ist ein Editor zur automatischen Textbearbeitung. Die Bearbeitung erfolgt mit Editorkommandos, die dem **sed** in einer separaten Scriptdatei oder direkt in der Kommandozeile übergeben werden. Um in der Kommandozeile mehrere Editorkommandos zu übergeben, kann die `-e` Option mehrfach verwendet werden. Die Editorkommandos können auch durch ein Semikolon getrennt werden. Wird nur ein einziges Editorkommando in der Kommandozeile übergeben, kann die Kennzeichnung mit der `-e` Option auch weggelassen werden. Damit die Shell keine Veränderungen an der Zeichenkette mit dem Editorkommando vornimmt, muß sie in Hochkommata eingeschlossen werden.

Eine Scriptdatei kann beliebig viele Editorkommandos enthalten, die durch Zeilenende oder Semikolon von einander getrennt werden müssen.

Jedes Kommando besteht aus einem Adressteil und einem Funktionsteil. Der Adressteil gibt an, welche Zeilen einer Textdatei mit diesem Kommando bearbeitet werden sollen, und der Funktionsteil beschreibt die Veränderung, die an den im Adressteil bestimmten Zeilen vorgenommen werden soll. Wenn kein Adressteil angegeben ist, wird die Funktion mit jeder Zeile ausgeführt.

Die Bearbeitung eines Textes erfolgt, indem die Eingabe zeilenweise in einen Arbeitsspeicher gelesen wird, und dann die Adressteil aller Editorkommandos der Reihe nach mit dem Text im Arbeitsspeicher verglichen werden. Die Funktionen der passenden Kommandos werden in der Reihenfolge ihres Auftretens ausgeführt. Normalerweise wird nach der Bearbeitung aller Kommandos der Inhalt des Arbeitsspeichers auf die Standardausgabe ausgegeben und danach durch die nächste Eingabezeile ersetzt. Die automatische Ausgabe des Arbeitsspeichers nach jeder Zeile kann mit der Option `-n` unterdrückt werden.

Zusätzlich zu dem Arbeitsspeicher gibt es noch einen Zwischenspeicher

(Puffer), der von verschiedenen Funktionen benutzt werden kann.

Der Arbeitsspeicher kann auch mehrere Zeilen auf einmal enthalten.

Im Adressteil können die Zeilen entweder durch ihre Zeilennummern, oder durch reguläre Ausdrücke ausgewählt werden. Alle Funktionen außer den `a`, `i`, `q` und `=` akzeptieren einen Adressbereich, bei dem eine Start- und eine Endadresse durch ein Komma getrennt angegeben werden. Ein Dollarzeichen steht für die letzte Zeile. Wenn eine Endadresse mit einem regulären Ausdruck bezeichnet ist, wird die erste passende Zeile als Bereichsende eingesetzt.

Reguläre Ausdrücke müssen in einfachen Schrägstrichen (Slashes `/`) eingeschlossen werden. `sed` benutzt die gleichen Routinen zur Auswertung regulärer Ausdrücke wie `emacs` oder `grep`. Darüberhinaus kann auch die an die `sed` Syntax angelehnte Konstruktion `\#Muster\#` (mit jedem beliebigen Zeichen für `\#`) benutzt werden, die wie `/Muster/` interpretiert wird.

Im Muster kann auch ein `\n` vorkommen, das auf das Zeilenende paßt.

19.1 Optionen von sed

Mit `sed` gibt es schier unendliche Möglichkeiten der Textmanipulation. `sed` bietet unter anderem folgende Funktionalität:

Operator	Name	Effekt
<code>[Muster/Adressraum]/p</code>	<code>print</code>	gibt den mit <code>[Muster/Adressraum]</code> gekennzeichneten Text aus
<code>[Adressraum]/d</code>	<code>delete</code>	Löschen des mit <code>[angegebener Adressraum]</code> gekennzeichneten Textes
<code>s/Muster1/Muster2/</code>	<code>substitute</code>	Ersetze das erste in einer Zeile auftretende Muster <code>Muster1</code> durch <code>Muster2</code>
<code>[Adressraum]/s/Muster1/Muster2/</code>	<code>substitute</code>	Ersetze über einen angegebenen Adressraum das erste Auftreten von <code>Muster1</code> durch <code>Muster2</code>
<code>[Adressraum]/y/Muster1/Muster2/</code>	<code>transform</code>	Transformiere über einen angegebenen Adressraum <code>Muster1</code> in <code>Muster2</code> um
<code>g</code>	<code>global</code>	Wendet das vorherstehende Kommando auf jede Zeile an, die das Muster enthält
<code>[Anzahl]q</code>	<code>quit</code>	beendet <code>sed</code> nach "Anzahl" Zeilen.
<code>[Muster/Adressraum]/w file</code>	<code>write</code>	Schreibt gefundene Zeilen in <code>file</code> .
<code>iText</code>	<code>insert</code>	fügt Text vorher ein
<code>aText</code>	<code>append</code>	fügt Text danach ein
<code>cText</code>	<code>change</code>	ersetzt durch Text
<code>=</code>		Gibt die aktuelle Eingabezeilennummer aus.
<code>{...}</code>		Die von den Klammern eingeschlossenen und davor stehenden Befehle werden für jede Zeile, die das Muster enthält, ausgeführt.

19.2 Beispiele

Im ersten Beispiel soll das Wort “UNIX” durch das Wort “Linux” ersetzt werden. Das abschließende `g` sorgt dafür, daß **jedes** in einer Zeile befindliche “UNIX” durch “Linux” ersetzt wird. Ohne `g` würde nur das erste auftretende “UNIX” ersetzt. Der Befehl ist in Hochkommata eingefaßt, damit Sonderzeichen nicht von der Shell interpretiert werden (in diesem Fall die Klammern). Ohne die Hochkommata sieht der Befehl etwas anders aus. Klammern und Leerzeichen müssen maskiert werden.

```
sed 's/UNIX(TM)/Linux/g' file
sed -e s/UNIX\ \(TM\) /Linux/g file
```

Es ist möglich die Befehlsfolgen, die `sed` abarbeiten soll, in einer Scriptdatei zu speichern. Die Ausgabe läßt sich auf dem üblichen Wege in eine Datei umleiten. Der Befehl sieht dann so aus:

```
sed -n -f muster_file inputfile >outputfile
```

Normalerweise gelten die vergegebenen Muster immer für den gesamten Text. Aber es ist auch möglich bestimmte Zeilen und Bereiche zu adressieren. Dies kann mit Zahlen, Mustern und mit dem `$`-Zeichen, als Kennung für die letzte Zeile, geschehen. Mit dem `!` können Zeilen ausgeschlossen werden. Die Syntax sieht dann folgendermaßen aus:

```
[Adresse1],[Adresse2] Kommando [Parameter]
```

Beispiele sagen mehr als tausend Worte:

```
sed '/10,$/ s/WWJD/TWJD/g' file
sed '/Josef/ s/WWJD/TWJD/g' file
sed '1!s/WWJD/TWJD/g' file
```

Im erste Beispiel wird von Zeile 10 bis zum Dateiende der String “WWJD” in “TWJD” geändert. Im zweiten Beispiel wird nur in Zeilen ersetzt, in denen das Wort “Josef” vorkommt. Das letzte Beispiel verändert “WWJD” in “TWJD” überall, außer in der ersten Zeile.

Noch ein paar nützliche Muster:

<code>8d</code>	löscht die achte Zeile
<code>/^\$/d</code>	löscht alle leeren Zeilen
<code>1,/^\$/d</code>	löscht alles von Zeile 1 bis einschließlich der ersten leeren Zeile

/GUI/d	löscht alle Zeilen in denen "GUI" vorkommt
/Jones/p	gibt nur Zeilen aus in denen der Name "Jones" vorkommt (mit -n)
1,10 p	gibt Zeilen 1 bis 10 aus (mit -n)
/^begin/,/^end/p	gibt jede Zeile aus, die sich zwischen Zeilen die mit "begin" und "end" anfangen
s/Windows/Linux/	ersetzt das erste in einer Zeile vorkommende, "Windows" mit "Linux"
s/BSOD/stability/g	setzt "stability" für jedes "BSOD" ein
s/00*/0/g	ersetzt "00", "000", ... mit "0"
s/GUI//g	löscht "GUI" in jeder Zeile
/^[0-9]/s/^/ /	alle Zeilen, die mit einer Zahl beginnen, um 3 Leerzeichen einrücken
/^\$/s/^/XXX/	alle leeren Zeilen mit "XXX" auffüllen
10q	zeigt die ersten 10 Zeilen an
/^X/w file	schreibt alle Zeilen, die mit "X" beginnen, in file

Kniffliger wird die Angelegenheit, wenn es sich um die Optionen i, a und c handelt:

```
#!/usr/bin/sed -nf

/ganz bestimmter Text/{
    i\
    Text gehört davor
    a\
    Text der danach stehen soll
}
```

Kommando und Ergebnis sehen dann so aus:

```
asterix% echo 'ganz bestimmter Text' | sed -f scriptdatei
Text gehört davor
ganz bestimmter Text
Text der danach stehen soll
asterix%
```

Aber es geht natürlich auch ohne Scriptdatei:

```
asterix% echo 'ganz bestimmter Text' | \
sed -e '/ganz bestimmter Text/{;i\ ' \
-e 'Text gehört davor' -e 'a\ ' -e 'Text der danach stehen soll' -e ' }'
```


An diesem Beispiel kann man des weiteren erkennen, daß es möglich ist, Befehlsgruppen zu bilden. Eine Gruppe wird durch die geschweiften Klammern zusammengefaßt. Das angegebene Suchmuster ‘/ganz bestimmter Text/’ wird dadurch von a und i gemeinsam genutzt.

Nichts verstanden? Macht nichts, es ist auch nicht einfach! Im Internet gibt es eine Vielzahl von Tutorien zu regulaeren Ausdruecken, sowie zu **sed** und **grep**. Und wie immer kann auch ein Blick in die **man**-Page nicht schaden. [LE,RO] [LO,RE]

Pawel Slabiak 2004-08-02