Unix & Shell-Programmierung SS21

Vorlesungswoche 5

Helga Karafiat

FH Wedel

Arithmetik und Auswertung von Ausdrücken mit `expr`

- Spezialisiertes Tool zur Auswertung von Ausdrücken (veraltet)
  - Ausdrücke (Berechnungen, Vergleiche, ...) werden als Parameter übergeben
  - Ergebnis der Auswertung wird auf `stdout` geschrieben, daher nur innerhalb von Kommandoersetzung sinnvoll einsetzbar
  - Nur Ganzzahlen, nicht gegen Überläufe geschützt, Division durch Null wird abgefangen
  - Verschachtelte Ausdrücke durch Klammerung mit runden Klammern möglich
  - Ersetzung von Variablen durch die Shell
- Schwächen von `expr`
  - Umständliche Benutzung, da alle Angaben Parameter sind (Viel Whitespace notwendig!)
  - Quoting von Zeichen, die eine Bedeutung in der Shell haben, unabdingbar
- Der POSIX-Standard rät von der weiteren Verwendung ab (aufgrund von Abwärtskompatibilität weiterhin vorhanden)

- Operatoren für Berechnungen (Rest: `man expr`)

Ausdruck	Bedeutung
E1 + E2	Summe von E1 und E2
E1 - E2	Differenz von E1 und E2
E1 * E2	Produkt von E1 und E2
E1 / E2	Ganzzahlige Division von E1 durch E2
E1 % E2	Rest der ganzzahligen Division von E1 durch E2

- Beispiele:
  - `echo \`expr 1 + 2\``
  - `echo \`expr \(1 + 2 \) * 3\``
  - `FOO=1 ; $FOO=\`expr $FOO + 2\``

Arithmetik in der Shell (Arithmetic Expansion)

- Schreibweise `$(( ... ))`
- erlaubt es arithmetische Ausdrücke auszuwerten und durch ihr Ergebnis zu ersetzen
- nur Ganzzahlarithmetik
  (nicht gegen Überläufe geschützt, Division durch Null wird abgefangen)
- orientiert sich am C-Standard für Rechenoperationen (keine vollständige Umsetzung)
- Verwendung analog zur Kommandoersetzung, Rückgabewerte:
  - der zuletzt berechnete Wert oder
  - Ergebnis des Vergleichs (0=false, 1=true)
- kein Quoting notwendig und beliebiger Whitespace erlaubt
- Ersetzung von Variablen wird vorgenommen, Angabe als:
  - `VARNAME`: Ersetzung durch die Arithmetic Expansion
    Variableninhalt wird ohne weitere Auswertung übernommen, geeignet für Ganzzahl-Variablen
  - `$VARNAME`: Zunächst Ersetzung der Variablen durch die Shell (Variable Expansion)
    geeignet zur Angabe von Rechenausdrücken als Variable, Auflösung nicht rekursiv

- Unterstützte Operationen:

Operationen	Bedeutung
+, -, *, /, %	Rechenoperatoren
=, +=, -=, *=, /=, %=	Zuweisungsoperatoren
`expression ? expression : expression`	Bedingte Ausführung
<, <=, >, >=, ==, !=	Vergleichsoperatoren
&&, ||, !	Logische Verknüpfungen / Negation
&, |, ^, ~, <<, >>, &=, |=, <<=, >>=	Bitweise Operatoren

- Beispiele:
  - `echo $(((1+2)*3))`
  - `FOO=1 ; FOO=$(( FOO+1 ))`
  - `FOO=1 ; BAR=$(( FOO > 2 ? -5 : 7 ))`
  - `CALC="1+2" ; echo $(( $CALC ))`

Spezielle Arithmetik-Erweiterung der bash

- eigenständige Arithmetic Evaluation `(( ... ))` zusätzlich zur Arithmetic Expansion
- kein Rückgabewert, Exitcode signalisiert das Ergebnis des letzten Ausdrucks:
  - 0: Vergleich wahr, Berechnungsergebnis / Zahl ungleich Null
  - 1: Vergleich unwahr, Berechnungsergebnis / Zahl ist Null
- wird innerhalb der Arithmetic Expansion in der `bash` verwendet
  (d.h. Arithmetic Expansion der `bash` hat erweiterten Funktionsumfang zu POSIX)
- Einige Erweiterungen:
  - Post- und Pre- inkrement und -dekrement mit `--` und `++`, z. B. `(( FOO++ ))`
  - Exponentialrechnung mit `**`, z. B. `(( 2**8 ))`
  - Angabe mehrerer Ausdrücke getrennt durch Kommata, z. B. `(( FOO=BAR+1, BAR=4 ))`
  - Rekursive Ersetzung von Variablen, die von der Arithmetic Evaluation ersetzt werden
    (Angabe mit `$VARNAME` führt zur Ersetzung durch die Shell, nicht rekursiv)
- kann bei arithmetischen Vergleichen anstelle von `test` verwendet werden oder
  zur Ausnutzung von Seiteneffekten
- in der Manpage der `bash` unter `ARITHMETIC EVALUATION`

Schleifen in der Shell (`while`)

- Syntax:

```
while CMD1
do
  CMD2
done
```

- Beispiel (Countdown)

```
CNT=5

while [ $CNT -ne 0 ]
do
  echo $CNT
  CNT=$(( $CNT - 1 ))
done
```

Schleifen in der Shell (`until`)

- Syntax:

```
until CMD1
do
  CMD2
done
```

- Beispiel (Countdown)

```
CNT=5

until [ $CNT -eq 0 ]
do
  echo $CNT
  CNT=$(( $CNT - 1 ))
done
```

Schleifen in der Shell (`for`)

- Syntax:

```
for VAR in LIST
do
  CMD1
done
```

- Beispiel (Werte ausgeben)

```
for ENTRY in a b c
do
  echo $ENTRY
done
```

- Beispiel (Backups von allen html-Dateien im Ordner erstellen)

```
for FILE in *.html
do
  cp $FILE ${FILE}.bak
done
```

Verzweigungen mit case

- Verzweigungen über viele Bedingungen (z.B. in Schleifen) werden mit `if - elif - else - fi` schnell unübersichtlich
- Elegantere Variante: Verzweigung mit `case`
- Syntax:

```
case Eingabe in
  Muster)
    ...
    ;;
  Muster)
    ...
    ;;
esac
```

- `;;` bricht die weitere Auswertung an der Stelle ab
- Mehrere Muster können mit | zu einer Option zusammengefasst werden
- `case` erlaubt die Verwendung der gleichen Wildcards wie die Shell als Muster

- Beispiel:

```
case $1 in
  *.c|*.h)
    echo "C-Datei: $1."
    ;;
  *.sh)
    echo "Shell Skript: $1."
    ;;
  *.*)
    echo "Unbekannter Dateityp: $1."
    ;;
  *)
    echo "$1 hat gar keine Endung."
    ;;
esac
```

Parameter mit `for` sequentiell verarbeiten

- Einfaches Beispiel (Alle Kommandozeilenparameter ausgeben)

```
for PARAM in "$@"
do
  echo $PARAM
done
```

- Gut geeignet, wenn jeder Parameter behandelt und gleich behandelt wird, z.B.

```
for PARAM in "$@"
do
  case $PARAM in
    -h | --help)
      echo "Hilfe!"
      ;;
    -p)
      echo "Eine wichtige Ausgabe."
      ;;
    -q)
      echo "Etwas anderes wichtiges."
      ;;
    *)
      echo "Unbekannte Eingabe!"
      ;;
  esac
done
```

Parameter mit `while` sequentiell verarbeiten

- Internes Shell-Kommando `shift` verschiebt die Parameterliste einen Platz nach links (konsumiert den ersten Parameter und setzt den Wert des zweiten auf den ersten, ...) und dekrementiert #-Variable
- Einfaches Beispiel (Alle Kommandozeilenparameter ausgeben)

```
while [ $# -gt 0 ]
do
  echo "$1"
  shift
done
```

- Aufruf von `shift n` mit Anzahl der zu shiftenden Werte ist möglich, Vorsicht:
  - `dash` gibt Fehlermeldung aus und bricht ab, wenn n größer ist als die Anzahl der vorhandenen Parameter
  - `bash` führt in dem Fall den shift nicht aus, was im Zweifelsfall zu einer Endlosschleife führt

- Gut geeignet, wenn mehrere Parameter in einer Option verarbeitet werden, z.B.:

```
while [ $# -gt 0 ]
do
  case "$1" in
    -h) echo "Help me!"
        shift
        ;;
    -1) echo "Angabe: $2"
        shift ; shift
        ;;
    -2) echo "Angaben: $2 $3"
        shift ; shift ; shift
        ;;
    *)  echo "Unbekannte Eingabe"
        shift
        ;;
  esac
done
```

Inline Dokumente

- Speichern von Texten in Shell-Skripten (z.B. für Usage oder bei automatisierter Generierung von Texten)
- Weiterer Umleitungsoperator: `<<`
  steht für Hier-Dokument (here document)
- Syntax: `CMD << Endmarke`
  - Endmarke kann jede Zeichenkette sein, die nicht allein in einer eigenen Zeile im Text vorkommt
  - Häufige Endmarken: `EOT` oder `EOF`
- Ersetzungen innerhalb der Texte werden standardmäßig vorgenommen, können aber mit `CMD << \Endmarke` oder `CMD << 'Endmarke'` unterbunden werden
- Vorteile gegenüber der Speicherung in einer extra Datei:
  - deutlich bessere Portabilität
  - sinnvollere Kapselung (z.B. Usage gehört zum Programmcode)

- Beispiele:

```
cat << EOT
Ein Beispieltext, der genau so wie er hier steht als
Konstante in einem Skript gespeichert wird.
Er wird beendet durch die Angabe der Endmarke EOT
in einer eigenen Zeile.
EOT
```

```
cat << EOT
Das ist der Wert von "$FOO" in einem längeren Text, der
im Skript gespeichert ist.
EOT
```

```
cat << \EOT
Das ist nicht der Wert von "$FOO" in einem längeren Text
EOT
```

Funktionen

- Eigenschaften:
  - werden im gleichen Prozess ausgeführt wie der Aufrufer
  - werden aufgerufen wie Kommandos
  - sind im Wesentlichen Shell-Skript-Sequenzen, die über ihren Namen aufgerufen werden
  - besitzen (wie ein richtiges Shell-Skript) ihre eigenen Argumente, verfügen über `stdin`, `stdout` und `stderr` und haben einen Exitcode (Returncode)
- Deklaration
  - Syntax

    ```
    function_name () {
      ...
    }
    ```
  - `()` deklariert die Funktion, `{}` fassen die Inhalte zusammen
  - Funktionsparameter werden mit `$1, ..., $n` referenziert
  - `$0` bleibt unverändert
  - Es gibt (wie bei einem Shell-Skript auch) keine explizite Deklaration der Eingabeparameter (daher auf Kommentierung achten)

- Setzen des Exit-Codes der Funktion über `return n`, wenn nicht explizit gesetzt, dann Exitcode des zuletzt aufgerufenen Befehls

- Variablen in Funktionen
  - alle außerhalb deklarierten Variablen (außer Parameter) sind in Funktionen zugreifbar (Verwendung globaler Variablen ist meistens schlechter Stil)
  - Variablen, die in Funktionen deklariert werden, sind global, außer man verwendet das Schlüsselwort `local`
  - `local` ist nicht POSIX-konform, aber selbst in minimalen Shells wie `dash` implementiert
- optionales Schlüsselwort: `function`
  - Nur in moderneren Shells (`bash`, `zsh`, `ksh93*`) für bessere Lesbarkeit enthalten
  - Nicht POSIX-konform
  - Syntax:

```
function function_name () {
  ...
}
```

<!– TODO Wichtiger Hinweis: letztes Mal ist einer gruppe bei Aufgabe 4 was interessantes aufgefallen - hier mal meine Antwort: ihr seid tatsächlich auf ein ganz interessantes Poblem gestoßen.

- Beispiel stdout: Ausgabe der Usage

```
usage () {
        cat << EOT
Usage: Wenn der erste Parameter "ok" lautet, wird
die Usage auf stdout ausgegeben, sonst auf stderr.
EOT
}

if [ "X$1" = "Xok" ]
then
        usage
        exit 0
else
        usage 1>&2
        exit 1
fi
```

- Beispiel Returncode: Zahlen vergleichen

```
compareNumbers () {
  [ $1 -eq $2 ] && return 0
  [ $1 -lt $2 ] && return 1
  [ $1 -gt $2 ] && return 2
  return 3
}

compareNumbers $1 $2
RC=$?

case $RC in
  0) echo "Numbers are equal" ;;
  1) echo "The first one is smaller" ;;
  2) echo "The second one is smaller" ;;
  3) echo "Something went terribly wrong!" ;;
esac
```

- Vorsicht: Returncodes folgen den gleichen Konventionen wie der Exitcode (Werte von 0-126)

- Beispiel Nutzung in einer Pipeline: Rot13-Kodierung

```
encode () {
  tr "A-Z" "a-z" |  tr "a-z" "n-za-m"
}

RESULT=""

case $1 in
  -e) RESULT="Encoded String: $(echo $2 | encode)" ;;
  -d) RESULT="Decoded String: $(echo $2 | encode)" ;;
esac

echo $RESULT
```

Ein vollständiges kleines (sinnloses) Skript

```
#!/bin/sh

usage () {
  cat << EOT
Usage: use -h or -u please.
EOT
}

error () {
  echo "Error: $1" 1>&2
  usage() 1>&2
}

ERROR=0

for PARAM in "$@"
do
  case $PARAM in
    -h) echo "Hilfe!"
        ;;
    -u) echo "Unsinn :)"
        ;;
    -*) error "Unbekannte Option"
        ERROR=1
        ;;
     *) error "Gar keine Option"
        ERROR=2
        ;;
done

exit $ERROR
```

Howto debug

- Debug-Ausgaben mit `echo` machen,
  z.B. `echo "Vor der Schleife"` oder `echo "\$FOO ist $FOO"`
- Nützliche Shell Optionen:
  - `-x`: Aktiviert x-trace Option. Führt dazu, dass jeder Befehl vor der Ausführung mit einem führenden + auf `stderr` ausgegeben wird.
  - `-n`: Aktiviert noexec Option. Befehle werden nur gelesen, aber nicht ausgeführt. Nützlich für einen Syntaxcheck, insbesondere bei Skripten die länger laufen.
  - `-u`: Aktiviert nounset Option. Die Shell warnt beim Zugriff auf nicht gesetzte Variablen und beendet das Skript.
  - `-v`: Aktiviert verbose Option. Die Shell schreibt die Skriptzeilen (ihre Eingabe) in der Verarbeitungsreihenfolge auf `stderr`.
  - Die Shell Optionen werden direkt bei der Shebang mit angegeben, z.B. `#!/bin/sh -x` oder das Skript wird explizit mit einer entsprechend gesetzen Shell gestartet, z.B. `sh -x ./foo.sh`
- Fehlermeldungen der Shell lesen und nachvollziehen

Howto code

- Code konsistent und lesbar formatieren (Einrückung, Leerzeilen, Zeilenumbrüche etc.)
  (erleichtert die Suche nach Fehlern)
- Variablen am Anfang der Hauptverarbeitung deklarieren
  (vermindert Inkonsistenzen)
- Funktionen dann verwenden, wenn sie die Lesbarkeit erhöhen oder mehrfach verwendete Befehlssequenzen kapseln
  - Aufteilung in zu viele Funktionen und/oder zu tiefe Schachtelung schlecht lesbar und wartbar
  - Klassische Stolperfallen:
    Ausgaben innerhalb von Funktionen, Setzen von globalen Variablen innerhalb von Funktionen
- Nicht zu viele `exits` und verschiedene Exitcodes ja nach Fehler verwenden
  (Programmverlauf bleibt nachvollziehbar)
- Interpreterangabe nicht vergessen
  (kann sonst je nach aufrufender Shell zu komischen Effekten führen)
- Testen von Befehlen / Befehlsketten direkt in der Shell
  (kleinteiliges Arbeiten erleichtert Funktionsprüfung und Fehlersuche)