UNIX™ Shell-Programmierung

Erweiterungskurs zu *Nutzung des Betriebssystems UNIX*

17. Überarbeitung | **Autoren:** Mitarbeiter des JSC |

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Inhalt

Einführung

Shell-Programmierung

Literatur / Hilfe

Inhalt

Voraussetzungen

Kenntnisse des Grundkurses “**Nutzung des Betriebssystems UNIX**“, u.a:

- Umgang mit einem Editor (`vi`, `kate`, o.ä.)
- Shell Ersetzungsmechanismen (Bedeutung der unterschiedlichen Hochkommata `` `...` ``, `'...'`, `"..."`)
- Shell Expansion von Dateinamen (Bedeutung der Wildcard-Zeichen `*`, `?`, `[...]`, \, reguläre Ausdrücke)
- I/O Redirection (Bedeutung der Symbole `>`, `>>`, `<`, `<<`)
- Pipelines (Bedeutung des Symbols `|`)
- Shell Variablen allgemein (Definition, Gültigkeitsbereich, `export`-Kommando)

Shell Ersetzungmechanismen I

’text’ Maskierung (*single quotes*)
keine Ersetzung; nur besondere Zeichen (bspw. \n, \t) werden interpretiert

Beispiel:

```
$ TEXT='Directory:'
$ echo '*New $TEXT `pwd`'
*New $TEXT `pwd`
```

`cmd` Kommandoersetzung (*backquotes*)
Ausgabe des Befehls ist das Ergebnis

Beispiel:

```
$ cd /etc; TEXT='Directory:'
$ echo '*New $TEXT' `pwd`
*New $TEXT /etc
```

Shell Ersetzungmechanismen II

"text" Parameterersetzung (*double quotes*)
$-Variablen und Kommandos werden ersetzt; keine Dateinamenauflösung

Beispiel:

```
$ TEXT='Directory:'
$ echo "*New $TEXT `pwd`"
*New Directory: /etc
```

$(cmd) Kommandoersetzung, zweite Möglichkeit
Ausgabe des Befehls ist das Ergebnis

Beispiel:

```
$ cd /etc; TEXT='Directory:'
$ echo '*New $TEXT' $(pwd)
*New $TEXT /etc
```

Shell Expansion I

Metazeichen werden zur Expansion von Verzeichnis- und Dateinamen genutzt (Beispiele unter *$HOME/test_reg_expr*)

*****
jede beliebige Zeichenkette (auch leer)

Beispiel: **`ls abc.*`**

→ alle Dateien beginnend mit *abc.*

?
ein beliebiges Zeichen

Beispiel: **`ls abc.?`**

→ alle Dateien beginnend mit *abc.*, gefolgt von einem beliebigen Zeichen

\
Zeichen zum Verstecken der Wildcardzeichen

Beispiel: **`ls abc.\*`**

→ die Datei *abc.** und nicht alle Dateien, die mit *abc.* beginnen

Shell Expansion II

Reguläre Ausdrücke können ebenso zur Angabe von Verzeichnis- und Dateinamen verwendet werden

[...] eines der in ... angegebenen Zeichen

Beispiel: **`ls abc.[123]`**

→ alle Dateien beginnend mit *abc.*, gefolgt von einer der Ziffern 1, 2 oder 3

[!...] keines der in ... angegebenen Zeichen

Beispiel: **`ls abc.[!89]`**

→ alle Dateien beginnend mit *abc.*, gefolgt von einer Ziffer 0-7, einem Zeichen A-z oder einem Sonderzeichen

Shell Expansion III

?(muster) oder **?(muster1|muster2|...)**
kein- oder einmaliges Vorkommen des/der Muster
Beispiel: `ls abc.?(3|[5-8])`
→ alle Dateien beginnend mit *abc.*, gefolgt von der Ziffer 3, 5, 6, 7, 8 oder dem Leerstring

***(muster)** oder ***(muster1|muster2|...)**
kein-, ein- oder mehrmaliges Vorkommen des/der Muster
Beispiel: `ls abc.*(1|2)`
→ alle Dateien beginnend mit *abc.*, gefolgt von 1 und 2, deren Kombinationen mit Wiederholung oder dem Leerstring

+(muster) oder **+(muster1|muster2|...)**
ein- oder mehrmaliges Vorkommen des/der Muster
Beispiel: `ls abc.+(1|2)`
→ alle Dateien beginnend mit *abc.*, gefolgt von den Ziffern 1 und 2 oder deren Kombinationen mit Wiederholung

Shell Expansion IV

@(muster) oder **@(muster1|muster2|...)**

genau einmaliges Vorkommen des/der Muster

Beispiel: `ls abc.@(dvi|log|aux)`

→ alle Dateien beginnend mit *abc.*, gefolgt von den angegebenen Endungen

!(muster) oder **!(muster1|muster2|...)**

keinmaliges Vorkommen des/der Muster

Beispiel: `ls abc.!(tex)`

→ alle Dateien beginnend mit *abc.* und beliebigen Endungen außer *'tex'*

Reguläre Ausdrücke I

Reguläre Ausdrücke nötig unter anderem bei **`grep`** oder **`egrep`**

. genau **ein** Zeichen

\ versteckt das folgende Zeichen vor der Auswertung

Beispiel: **\.**

^ Muster am Anfang der Zeile

Beispiel: **`grep ^Me.er telefon`**

$ Muster am Ende der Zeile

Beispiel: **`grep 47011$ telefon`**

\< Muster am Anfang des Wortes

Beispiel: **`grep '\<.ei' telefon`**

\> Muster am Ende des Wortes

Beispiel: **`grep 'er\>' telefon`**

[abc],[x-z],[0-9] Zeichenklasse

Reguläre Ausdrücke II

[...] **eins** der aufgeführten Zeichen darf vorkommen

[^...] **keins** der aufgeführten Zeichen darf vorkommen

***** vorangestellter Ausdruck kann beliebig oft vorkommen, auch keinmal

Beispiel: `grep 'data[0-9]*' file.log`

.* beliebige Zeichenkette

+ Ausdruck muss mindestens einmal vorkommen (*nur* `egrep`)

Beispiel: `egrep 'data[0-9]+' file.log`

? Ausdruck kann kein- oder einmal vorkommen (*nur* `egrep`)

Beispiel: `egrep 'data[0-9]?' file.log`

I/O Redirection

> Schreiben auf eine Datei

Beispiel: **`who > output`** → schreibt pro "logged in" Benutzer eine Zeile in die Datei *output*

>> Anhängen an eine Datei

Beispiel: **`id >> output`** → schreibt Informationen zur Benutzernummer an das Ende der Datei *output*

< Lesen einer Datei

Beispiel: **`wc -l < input`** → liest die Datei *input* und bestimmt die Anzahl der Zeilen

<< Erstellen eines Here-Documents

Beispiel:

```
ssh zam4272 << EOF
echo "Rechnername: $(hostname) "
echo "Datum/Uhrzeit: `date`"
EOF
```

→ nimmt die nächsten Zeilen bis zum Delimeter **EOF** als Eingabe für den **ssh**-Befehl

Inhalt

Definition eines Shell-Scripts I

Definition:

Ein **Shell-Script**, auch als Shell-Prozedur bezeichnet, ist eine Datei bestehend aus einer Aneinanderreihung von Kommandos.

Kommentare:

Steht in einer Zeile ein #-Zeichen, wird der Rest der Zeile von der Shell nicht interpretiert.

Beispiel:

```
#*********************************
# Function: Print duplex at ZAM   *
# Format:   mylpr2 [files]        *
#*********************************
lpr -Pzam23 $*  # print all files
lpq -Pzam23     # list queue
```

Definition eines Shell-Scripts II

- Da es syntaktisch unterschiedliche Shells gibt, können Shell-Scripts gekennzeichnet werden, so dass sie zwingend in einer bestimmten Shell ausgeführt werden.
- Dazu in der 1. Zeile ab der 1. Spalte:

#!/bin/sh	→ Bourne-Shell
#!/bin/csh	→ C-Shell
#!/bin/ksh	→ Korn-Shell
#!/bin/bash	→ Bourne-Again-Shell

- Es gibt Systeme, die die #! Notation nicht unterstützen; dann gilt beim 1. Zeichen:

:	→	Bourne-Shell
#	→	C-Shell

- Default:
  - abhängig vom System
  - meist eine Version der Bourne Shell
  - vereinzelt auch die Korn Shell

Syntax

- Leerzeilen zur Strukturierung sind erlaubt.
- Folgezeilen:

first_part_of_command_string \
rest_of_command_string

Ein Befehl kann sich über mehrere Zeilen erstecken. Dazu muss als letztes Zeichen einer Zeile, die fortgesetzt werden soll, ein "\" stehen (verhindert die Interpretation des Newline bzw. Linefeed).

Beispiel:

```
#!/bin/ksh
#*********************************
# Function: Display a message     *
# Format:   msg                   *
#*********************************
echo "Diese Zeile wird \
hier fortgesetzt"
echo "Dies ist die 2. Ausgabezeile"
```

Sequentielle Kommando Listen I

- einfache Liste:

  command1 **;** ***command2*** [; ...]

  - mehrere Befehle können durch “**;**” getrennt in eine Zeile geschrieben werden
  - Befehle werden hintereinander ausgeführt
  - Return Code des vorherigen Befehls spielt keine Rolle
  - I/O Redirection kann für jeden Befehl angegeben werden

  Beispiel: `cd mydir; rm main.o`

- Pipelines:

  command1 | ***command2*** [| ...]

  - Abarbeitung der Befehle sequentiell (li $\rightarrow$ re)
  - Ausgabe des vorherigen Befehls ist Eingabe des nächsten Befehls
  - I/O Redirection möglich (**Achtung**: Open von rechts nach links!)

  Beispiel: `grep zam /etc/hosts | more`

Übung

1. Schauen Sie sich das Shell-Script *syntax.demo* im Verzeichnis *$HOME/bin* an.

2. Führen Sie das Shell-Script aus, indem Sie den Namen eingeben:

   `syntax.demo`

Sequentielle Kommando Listen II

- Verbund-Kommando (Form 1):

  { *command1* ; *command2;* [...] }

  - letztes Semikolon ist zwingend
  - Leerzeichen vor/hinter geschweiften Klammern sind zwingend
  - Abarbeitung der Befehle geschieht sequentiell
  - es wird **keine** neue Shell gestartet
  - Shell-Variablen sind bekannt und können verändert werden *(auch unbeabsichtigt!)*
  - gemeinsame I/O Redirection der Befehle, d.h. lesen/schreiben ebenfalls sequentiell *(kein neues Open!)*

Beispiele:

```
{ date; ~/bin/syntax.demo; date; } > out
A=1; { echo $A; A=2; }; echo $A
```

Sequentielle Kommando Listen III

- Verbund-Kommando (Form 2):

(***command1*** ; ***command2*** [;...])

  - letztes Semikolon und Leerzeichen vor/hinter den Klammern sind nicht zwingend
  - I/O Redirection wirkt wie in Form 1
  - es wird zur Abarbeitung eine neue Shell aufgemacht
  - Shell-Variablen sind bekannt, Änderungen sind aber nur in der neuen Shell gültig

Beispiele:
```
(mypgm1; mypgm2) < inp > out
A=1; (echo $A; A=2); echo $A
```

Bedingte Kommando Listen I

- True Liste:

command1* && *command2 [&& ...]

Die dem && folgenden Befehle werden nur ausgeführt, wenn der vorherige Befehl erfolgreich endet.

Beispiel: `cd mydir && rm main.o`

- False Liste:

command1 || ***command2*** [|| ...]

Die dem || folgenden Befehle werden nur ausgeführt, wenn der vorherige Befehl nicht erfolgreich endet.

Beispiel: `cd mydir || mkdir mydir`

Bedingte Kommando Listen II

- Kombinationen von && und || sind erlaubt

cmd1 && cmd2 || cmd3	(Form 1)
cmd1 || cmd2 && cmd3	(Form 2)

  - die Abarbeitung erfolgt von links nach rechts
  - Fortsetzungszeilen sind durch \ möglich
    Beispiel:
    ```
    cd /bin && ls -l \
    sub*
    ```
  - *cmdx* kann wieder eine Liste von Befehlen in Form eines Verbund-Kommandos sein
    Beispiel:
    ```
    cd /bin && (touch abc;ls -l)
    ```

(1) bedeutet: falls *cmd1* ok, dann *cmd2* ;
falls *cmd1* oder *cmd2* fehlerhaft, dann *cmd3*

Beispiel:
```
cd mydir && rm main.o || pwd
```

(2) bedeutet: falls *cmd1* fehlerhaft, dann *cmd2* ;
falls *cmd1* oder *cmd2* ok, dann *cmd3*

Beispiel:
```
cd dir 2>/dev/null || { mkdir dir; \
cd dir; } && mypgm > my.out
```

Aufruf eines Shell-Scripts I

Externe Shell

Ein Shell-Script kann aufgerufen werden durch:

ksh ***filename [arguments]*** (1)

oder:

filename [arguments] (2)

- Form (2) gilt, falls das Directory mit der Datei *filename* im Suchpfad enthalten ist; Empfehlung: *$HOME/bin*
- die Datei *filename* muss bei Form (2) *read* und *execute* Rechte besitzen: `chmod u+x` *filename*
- es wird eine separate Shell gestartet, die als Eingabe die Datei verarbeitet
- es stehen ***nur exportierte Variablen*** zur Verfügung
- eine Beeinflussung der Umgebung der rufenden Shell ist nicht möglich

Übung

1. Setzen Sie die Shell-Variable `SHOWXVAR` und exportieren Sie diese:

   ```
   export SHOWXVAR=old
   ```

2. Listen Sie die Variable auf:

   ```
   echo $SHOWXVAR
   ```

3. Führen Sie das Shell-Script *showx.demo* in einer separaten Shell aus:

   ```
   showx.demo
   ```

4. Listen Sie nach Ausführung des Scripts wiederum die Variable auf.

5. Welchen Wert hat die Variable und warum?

Aufruf eines Shell-Scripts II

Interne Shell

Soll ein Shell-Script innerhalb der gleichen Shell ausgeführt werden, muss es mit dem *Punkt*-Befehl aufgerufen werden:

```
. filename
```

- zwischen "." und *filename* muss mindestens ein Leerzeichen stehen
- es reicht dann *read* Zugriffsrecht auf die Datei *filename*
- es stehen auch nicht exportierte Variablen zur Verfügung
- das rufende Environment kann geändert werden
  (auch versehentlich)
- typische Anwendung sind Änderungen im *.bashrc* oder *.kshrc*, die aktiviert werden sollen
- es können jedoch keine Argumente übergeben werden

Übung

1. Setzen Sie die Shell-Variable SHOWVAR ohne diese zu exportieren:

   ```
   SHOWVAR=12345
   ```

2. Listen Sie die Variable auf:

   ```
   echo $SHOWVAR
   ```

3. Führen Sie das Shell-Script *show.demo* in derselben Shell aus:

   ```
   . show.demo
   ```

4. Listen Sie nach Ausführung des Scripts wiederum die Variable auf.

Beenden eines Shell-Scripts

exit [*n*]

- beendet das aktuelle Shell-Script
- Achtung: falls es sich um ein Shell-Script handelt, das in der Login-Shell aufgerufen wurde (*. filename*), kehrt man zum Login-Prompt zurück bzw. wird das aktuelle Konsole-Fenster geschlossen
- durch n kann ein Return Code mitgegeben werden
- standardmäßig wird der Exit-Status des zuletzt ausgeführten Kommandos zurückgegeben

Aufruf eines Shell-Scripts III

Hinweis

Bei Ausführung in der gleichen Shell *(. Befehl)* kann die Shell-umgebung geändert werden, beispielsweise durch

- **cd** → Ändern des Working Directory
- Umsetzen von wichtigen Shell Environment Variablen durch Gebrauch ihrer Namen, beispielsweise

PATH	Suchpfad → Programme werden nicht mehr gefunden
HOME	Heimatverzeichnis → Daten werden falsch oder gar nicht abgelegt
MOUNT	Automount-Verzeichnis → Zugriff zu NFS Daten geht nicht mehr
ENV	Shell-Profile → Shell Initialisierung geht nicht mehr
SHELL	Shell-Name → Anwendungen gehen evtl. nicht mehr
PS1	Prompt String → Systemprompt ist anders
DISPLAY	X-Server → X-Fenster sind nicht sichtbar, da sie evtl. woanders aufgemacht werden
TERM	Terminalemulation → Sonderzeichen und Tasten gehen nicht mehr

Aufgabe (A1)

Schreiben Sie ein Shell-Script *to*, in dem Sie zum Directory */etc* wechseln. Geben Sie in dem Script vor und nach dem Wechsel das aktuelle Working Directory aus.

Hinweis: Benutzen Sie die Kommandos **`pwd`** und **`cd`**.

1. Rufen Sie Ihr Script zuerst mit **`to`** auf.
2. Rufen Sie Ihr Script dann mit **`. to`** auf.
3. Wie lautet jeweils Ihr Working Directory nach Beenden des Scripts und warum?

Ausgeben von Daten - echo, print

	echo [***options***] ***text***	
oder:		
	print [***options***] ***text***	**(nur ksh!)**

- geben Text oder Shell-Variablen auf Standardausgabe aus
- soll die Ausgabe in eine Datei erfolgen, so muss die Ausgabe mittels I/O Redirection umgeleitet werden (`>`, `>>`)
- Option `-n` verhindert Zeilenumbruch nach der Ausgabe
- `echo -e` kann sowohl einige Symbole (Tabs \t, Newline \n, Linefeed \c) als auch Farb- und Attributänderungen
- **ksh:** `print` kann einige Symbole (\t, \n, \c)
- mit `echo` bzw. `print` ist keine spaltengerechte, tabellarische Ausgabe möglich; eine Folge von Blanks muss mit Hochkommata maskiert werden

Farben und Attribute I

echo -e "\033[*e*;*bf*m*Text*"

- die Ausgabe kann mit Farben und Attributen versehen werden
- gesteuert über sogenannte Escape-Sequenzen
- eingeleitet mit einem ESC-Zeichen (\033), gefolgt von [
- anschliessend die gewünschten Effekte, dann mit ";"getrennt Vorder- oder Hintergrund und danach die Farben
- die Sequenz schließt mit **m** ab
- mehrere Sequenzen können für einen Text angegeben werden

  Beispiel: `echo -e "\033[0;34mText1\033[0;45mText2"`
- diese Einstellung gilt bis zum Zurücksetzen mit

  reset

Beispiel: `echo -e "\033[0;34mBsp-Text"`

0 → kein Effekt
3 → Vordergrund
4 → blau

Farben und Attribute II

Folgende Farben und Attribute gelten für `echo -e`:

Zeichen/Effekte (e)

0	ohne Effekt
1	fett
4	unterstrichen (abhängig vom Terminal)
5	blinken (abhängig vom Terminal)
7	Vorder/Hintergrund tauschen (invertieren)
22	fett zurücksetzen
24	unterstrichen zurücksetzen
25	blinken zurücksetzen
27	invers zurücksetzen

Bereiche (b)

3	Vordergrund
4	Hintergrund

Farben (f)

0	schwarz
1	rot
2	grün
3	braun
4	blau
5	magenta
6	cyan
7	hellgrau
9	auf Normalfarbe zurücksetzen

Ausgeben von Daten – printf

printf "*format*" [*arguments*]

- `printf` konvertiert und formatiert entsprechend der *format* Angabe die Ausgabe
- gängige Formate sind:

%*n*s	String rechtsbündig der Länge *n*
%-*n*s	String linksbündig der Länge *n*
%c	erstes Zeichen eines Strings
%*n*i	Integerzahl der Länge *n*
%*n.m*f	Gleitkommazahl in Dezimalscheibweise
%*n.m*e	Gleitkommazahl in Exponentschreibweise

Beispiel:

```
echo   $NN $VN
echo   $NN          $VN
print "$NN          $VN"
printf "%-10s%-30s\n" $NN $VN
printf "%5d%10d\n" 333 70 214 12345678
```

Übung

1. Listen Sie das Shell-Script *write.demo* auf und veranschaulichen Sie sich die Möglichkeiten des **`echo`** Kommandos.
   Rufen Sie das Script anschließend auf:

   `write.demo` *vorname nachname*

2. Ändern Sie das Script so ab, dass der Text zusätzlich in der Konsole ausgegeben wird. Probieren Sie verschiedene Attribut- und Farbeinstellung mit **`echo -e`** und entsprechenden Escape-Sequenzen aus, auch unterschiedliche Farben in einer Zeile.
   Hinweis: Schauen Sie das Kommando **`tee`** nach, benutzen Sie die **`man`** Page oder suchen Sie online.

Einlesen von Daten - read

read [***options***] [***var1*** [***var2*** [...]]]

- liest genau eine Zeile von Standardeingabe
- weist die eingegebenen Zeichen/Worte, die durch Leerzeichen oder Tabs voneinander getrennt sind, den Shell Variablen var_n zu
- andere Trennzeichen können über die Shell-Variable **`IFS`** definiert werden (z.B. `IFS=:`)
- werden mehr Strings eingegeben, als Variablen vorhanden sind, so erhält die letzte Variable den Rest der Eingabezeile
- sind weniger Strings als Variablen vorhanden, so sind die restlichen Variablen leer
- fehlt var_n, wird die Zeile der Variablen **REPLY** zugewiesen

read - Optionen

- $options$ zur Steuerung können angegeben werden, beispielsweise:

  -r Raw Mode: Backslash wird nicht als special character interpretiert

  -u n Angabe von einem anderen File Descriptor; Default 0

- die *bash* bietet noch weitere $options$, beispielsweise:

  -p Angabe eines Prompt-String

  -s *silent* - Eingabe wird nicht angezeigt

  -n m maximale Anzahl m Zeichen, nach m Zeichen automatisch **`Enter`**

  -a Zeichenkette wird auf ein Feld gelesen

  -ei Inhalt einer Variablen kann editiert werden

Beispiele I

1
```
# Gebe 1. Spalte einer Datei als letzte aus
while read -r X Y; do
   echo $Y $X >> data.out
done < data.in
```

2
```
# Lese Datei Zeile fuer Zeile mittels File Descriptor
exec 3<exec.inp         # -> open 3
read -u3 VN NN          # -> 1. line
read -u3 STREET         # -> 2. line
read -u3 CITY           # -> 3. line
exec 3<&-               # -> close 3
echo "$NN, $VN, $CITY"
```

3
```
# Lese Daten mit Prompt String ohne Zeilenvorschub, nur ksh!
read X?"Eingabe von x und y:" Y
```

4
```
# anderes Trennzeichen zum Einlesen ueber IFS definiert
ZEILE="root:x:0:1:Super-User:/:/usr/bin/ksh"
IFS=:
read f1 f2 f3 f4 f5 f6 f7 <<EOF
  $ZEILE
EOF
```

Beispiele II

Die folgenden Beispiele gelten nur für die *bash*

1
```
# Lese Daten mit Prompt String
read -p "Eingabe von x und y: " X Y
```

2
```
# Eingabe eines Kennwort ohne Ausgabe
read -s -p "Kennwort eingeben: " VAR
```

3
```
# Eingabe besteht nur aus einem Zeichen
# Enter nicht noetig
read -n1 -p "Antwort (j/n): " VAR
```

4
```
# Variableninhalt kann editiert werden
# und wird auf VAR eingelesen
VAR=abc
read -ei $VAR -p "Variableninhalt: " VAR
```

5
```
# Zeichenkette auf Feld einlesen
read -a ARR <<EOF
$(date)
EOF
```

Übungen

1. Listen Sie das Shell-Script *read.demo* auf und veranschaulichen Sie sich die Möglichkeiten des `read` Kommandos. Rufen Sie das Script anschließend auf:

   ```
   read.demo
   ```

2. Vollziehen Sie die Kommandos aus den Beispielen I und II nach. Geben Sie für die Beispiele II auch die eingelesenen Werte aus.

Aufgabe (A2)

1. Schreiben Sie ein Shell-Script *nuser*, das die Uhrzeit in der Form hh:mm:ss und die Anzahl unterschiedlicher Benutzer im System ausgibt.

   Hinweis: Benutzen Sie `date`, `sort`, `who`, `wc`
2. Für die Fehlermeldung beim Hersteller benötigen Sie folgende Angaben:
   - den Betriebssystemnamen
   - die Version mit Release-Angabe
   - den Hostnamen Ihres Rechners
   - die aktuelle Konfiguration (Devices, Adressen)

   Schreiben Sie die Angaben mit dem Script *getsystem* in die Datei *getsystem.out*.

   Hinweis: Benutzen Sie `uname`, `hostname`, `lsdev`.

Tipp: Probieren Sie die Kommandos mit Optionen einzeln aus und schauen Sie sich jeweils die Ausgabe an.

Parameterübergabe

Überblick

Zur Übergabe von Parametern an das Shell-Script gibt es drei Möglichkeiten:

- indirekt über exportierte Variablen (1)
- direkt über Keyword Parameter (2)
- direkt über Positional Parameter (3)

exportierte Variablen

export ***variablenliste***
filename

- alle Variablen in *variablenliste* werden als exportiert markiert
- sie gelten in allen aufgerufenen Shells ab der Shell, in der sie exportiert wurden

  Beispiel:

```
export VAR1=otto
ksh script1 &    => $VAR1 ist otto
ksh script2 &    => $VAR1 ist otto
```

- sie gelten auch in Nachfolge-Shells, d.h. in geschachtelt aufgerufenen Scripts

  Beispiel:

```
export VAR1=otto
ksh script1      => $VAR1 ist otto
|  VAR1=hugo
|  ksh script11  => $VAR1 ist hugo
|  |   VAR1=eva  => $VAR1 ist eva
|  |_  exit      => $VAR1 ist hugo
|_ exit          => $VAR1 ist otto
```

Keyword Parameter

var1=value1 [var2=value2 [...]] filename

- die Variablen var_n werden angelegt
- Gültigkeitsbereich wie exportierte Variablen
- nach Beendigung des Scripts sind sie nicht mehr verfügbar

Beispiel:

```
unset VAR1
VAR1=otto script1  => $VAR1 ist otto
|   VAR1=hugo
|   script11       => $VAR1 ist hugo
|   |   VAR1=eva   => $VAR1 ist eva
|   |_  exit       => $VAR1 ist hugo
|_  exit           => $VAR1 ist leer
```

Übung

Rufen Sie das Shell-Script *keyword.demo* mit zwei Keyword Parametern **VAR1** und **VAR2** auf.
Listen Sie beide Variablen nach Beendigung des Scripts auf.

Positional Parameter

filename arg1 [arg2 [[...] argn]]

- Leerzeichen dienen als Trennzeichen zwischen den Argumenten
- Argumente, die Blanks enthalten, müssen in Hochkommata eingeschlossen werden:
  "..." $\rightarrow$ mit Parameterersetzung
  '...' $\rightarrow$ ohne jedwede Ersetzung
- Argumente werden referiert über **`$1`**, **`$2`**, ..., **`$9`** oder **`${1}`**, **`${2}`**, ..., **`${9}`**
- als ksh/bash-Erweiterung gibt es darüber hinaus gehend **`${10}`**, **`${11}`**, ...
  Wichtig: **`$10`** (ohne {}) bedeutet Wert von **`$1`**, also 1.Parameter, verkettet mit 0 und nicht 10. Parameter!
- unbesetzte Parameter sind leer: " "

Übung

Rufen Sie das Shell-Script *parms.demo* mit unterschiedlichen Parametern auf und begründen Sie die Parameterzuweisung:

```
parms.demo 1 2 3 4 5 6 7 8 9 10 11
parms.demo 'a b' c d
parms.demo "a b" c d

UEB='Dies ist ein Test'
parms.demo $UEB !
parms.demo "$UEB !"
parms.demo '$UEB !'

parms.demo `ls`
```

Aufgabe (A3)

Schreiben Sie ein Shell-Script *x*, das das *execute* Zugriffsrecht für den Eigentümer einer spezifizierten Datei setzt.

Aufruf: **x** *filename*

Sonderparameter

- folgende Shell-Variablen stehen immer zur Verfügung:

`$#`	Anzahl der an die Prozedur übergebenen Parameter
`$0`	Name der Prozedur, die gerade ausgeführt wird
`$1,...`	positionale Parameter
`$*`	alle übergebenen Parameter als ein Zeichenstring; darin durch Blank getrennt
`$?`	der Exit-Status des letzten ausgeführten Kommandos
`$$`	die Prozess-Id der ausführenden Shell; nützlich zur Erzeugung eindeutiger Dateinamen
`$!`	die Prozess-Id des letzten im Hintergrund ausgeführten Prozesses
`$-`	die gegenwärtig gesetzten Shell-Optionen

Übung

Rufen Sie das Shell-Script *special.demo* mit unterschiedlichen Parameterlisten auf und schauen Sie sich die Sonderparameter an:

```
special.demo 1 2 3 4 5 6 7 8 9 10
special.demo "Es war einmal ..."
special.demo $HOME
special.demo "A + B" c 'D E F' 123
```

Aufgabe (A4)

Ändern Sie Ihr Script *x* aus Aufgabe 3 so, dass Sie das *execute* Zugriffsrecht für alle spezifizierten Dateien setzen. Nennen Sie das neue Script *xx*.

Aufruf: **xx** *fn1* [*fn2* [...]]

Parameterverschiebung

shift [*n*]

- bewirkt eine Verschiebung der Parameter um n Positionen nach links ($1 ← ${n+1})
- die vorderen Parameter fallen raus
- der Default für n ist 1
- sinnvoll bei variablen Parameterlängen (z.B. bei der Übung *parms.demo* letzter Aufruf)
- notwendig bei nicht ksh/bash-Scripts, um auf die Parameter ab dem 10. Parameter zugreifen zu können

Beispiel:

```
# Das Script teste enthalte:
echo $1              # 1. Parameter
shift; echo $1       # 2. Parameter
shift 3; echo $1     # 5. Parameter
```

```
$ ksh teste 1 2 3 a b c
1 2 b
```

Shell-Variablen

Zuweisung

	NAME=[value]
referiert über:	***$NAME***
	${NAME}

- $NAME$ ist der Variablenname und beginnt mit einem Buchstaben und kann Ziffern und _ enthalten; Groß-/Kleinschreibung wird unterschieden; vereinbarungsgemäß sollten nur Großbuchstaben verwendet werden
- die 2. Form der Referenz muss benutzt werden, wenn der Name nicht eindeutig aus dem Kontext hervorgeht:

```
VAR=abc
echo $VAR123        →   leer
echo ${VAR}123      →   abc123
echo $VAR 123       →   abc 123
echo ${VAR} 123     →   abc 123
```

Zuweisung

- *value* ist der Wert der Variablen; er kann auch leer sein:

```
NAME=
NAME=''
NAME=""
```

  Wenn er Leerzeichen enthält, muss er in Hochkommata eingeschlossen werden:

```
NAME='string with blanks'
NAME="string with blanks"
```

  Er kann auch das Ergebnis eines Shell Kommandos sein:

  NAME=`***command*` oder **NAME=$(***command***)**

- vor und hinter dem Gleichheitszeichen dürfen **keine Leerzeichen** stehen
- Shell-Variablen werden prinzipiell als Zeichenketten behandelt

Zuweisung

Beispiel:

```
A=123
STR1=Otto
STR2='Otto & Emma'
STR3="$HOME/doc/kapitel"
lpr -Pzam23 $STR3.1    => druckt die Datei kapitel.1
MYCC='cc -o temp ctmp.c'
$MYCC                  => führt Befehl aus
STR_LS=`ls`
echo $STR_LS           => enthält Dateien
LEER=""
```

Übung

1. Listen Sie das Shell-Script *var.demo* auf.
2. Schauen Sie sich die Zuweisungen an.
3. Führen Sie das Script anschließend aus und prüfen Sie die Zuweisungen anhand der Ausgabe.
4. Wo steckt der Fehler und was passiert?

Shell Environment-Variablen

- einige Variablen werden für die Shell Umgebung mitgeliefert.
  Sie sollten nur gezielt verändert werden:

HOME	Heimatverzeichnis
PATH	Suchpfad für Kommandos
PS1	Prompt-String
PS2	Folgeprompt-String
SHELL	Login-Shell
USER	Login-Username
TERM	Terminal-Type
PRINTER	Default-Drucker
DISPLAY	X11- Server Bildschirm

- FZJ spezifisch:

MOUNT	Automounter Mountpoint
WORK	Directory für Arbeitsdateien

- eine Liste der gesetzten Shell-Variablen erhält man mit:
  `env` oder `set`
- den Wert einer Variablen erhält man mit: `echo` $NAME

Felder von Variablen I

	FELD [index]=value	
referiert über:		
	${*FELD [index]*}	(1)
	${*FELD [*]*}	(2)

- $FELD$ ist der Name des Feldes
- $index$ ist der Index des Feldelementes; er wird durch eckige Klammern [. . .] markiert
- Form (1) referiert einzelne Feldelemete, Form (2) das gesamte Feld (zum Auflisten aller Elemente)
- es sind nur eindimensionale Felder erlaubt
- die maximale Anzahl der Elemente ist system–abhängig; Standard sind 512, 1024 oder auch 4096 (beginnend bei 0)
- für das 0. Element gilt: $FELD$ und $FELD$[0] sind identisch; entsprechend die Referenzen **$**$FELD$ und **${**$FELD$[0]}

  Beispiel: `A=0; A[1]=0; A[2]=0`

Felder von Variablen II

- nicht besetzte Feldelemente sind leer

  Beispiel:
  ```
  ADDR[2]=KFA
  ADDR[3]="52425 Jülich"
  --> ADDR[0],ADDR[1]=''
  ```

- Felder können initialisiert werden:

  set -A *FELD* *value-list*

- initialisiert werden *FELD*[0], *FELD*[1], ... durch Werte aus *value-list*
- alte Feldelemente werden überschrieben
- falls das alte Feld länger war, wird automatisch ein **unset** auf nicht benötigte Elemente gemacht

  Beispiel:
  ```
  set -A LISTE 0 0 0
  set -A DATUM `date`
  echo ${DATUM[*]}  --> DATUM[0]=wochentag
                        DATUM[1]=monat
                        DATUM[2]=tag im monat
                        ...
  ```

Felder von Variablen III

- weitere Möglichkeit der Zuweisung:

  FELD=(val1 val2 ...)

- initialisiert werden *FELD*[0], *FELD*[1], ... durch *val1 val2* ...
  Beispiel:
```
DATUM=($(date))
echo ${DATUM[*]}  --> DATUM[0]=wochentag
                      DATUM[1]=monat
                      DATUM[2]=tag im monat
                      ...
```
- alte Feldelemente werden überschrieben
- falls das alte Feld länger war, wird automatisch ein **unset** auf nicht benötigte Elemente gemacht
- ***${FELD[*]:start:laenge}***
  liefert den angegebenen Bereich ab *start* mit *laenge* Elemente

Übung

Listen Sie das Shell-Script *array.demo* auf und schauen Sie sich die Zuweisungen an. Führen Sie das Script anschließend aus und prüfen Sie die Zuweisungen anhand der Ausgabe.

Aufgabe (A5)

Schreiben Sie ein Shell-Script *getls*, das nur die Spalten 1 (Inode), 3 (Zugriffsrechte), 7 (Größe) und 11 (Dateiname) der Ausgabe des Befehls

`ls -lisa` *filename*

ausgibt. Das Script wird mit dem gewünschten Dateinamen als Parameter aufgerufen.
Aufruf: **`getls`** *filename*

Länge von Variablen

${*#NAME*}	(1)
${*#FELD [index]*}	(2)
${*#FELD [*]*}	(3)

- Anzahl der Zeichen bei einfachen Variablen (1) oder Array-Elementen (2)
- Anzahl der besetzten Elemente bei Arrays (3)

Beispiel:

```
A=123
A_LEN=${#A}              => 3
B[1]=1; B[10]=10
B_ANZ=${#B[*]}           => 2 nicht 11
variable Felderweiterung:
C[0]=0; C[1]=10; C[2]=100
C_L1=${#C[1]}                 => 2
C_ANZ=${#C[*]}                => 3
C[${#C[*]}]=1000              => C[3]=1000
C_ANZ=${#C[*]}                => 4
```

Substrings von Variablen I

rechtsseitig:

${*NAME#pattern*}	(1)
${*NAME##pattern*}	(2)

- gesucht wird von links nach rechts ($\rightarrow$);
  der rechte (hintere) Teil ist der Substring
- *pattern* ist ein Muster, nach dem gesucht wird und kann die bei Dateinamen üblichen Wildcard-Zeichen enthalten *****,**?**,**[...]**
- Form (1) bedeutet Abtrennen hinter dem ersten Auftreten von *pattern* in **$***NAME* von links
- Form (2) heißt Abtrennen hinter dem letzten Auftreten von links

Beispiel:

```
MYBIN=/u/zdv/zdv045/bin
${MYBIN#*v}          => /zdv045/bin
${MYBIN##*v}         => 045/bin
${MYBIN#*/}          => u/zdv/zdv045/bin
${MYBIN##*/}         => bin
```

Substrings von Variablen II

linksseitig:

${*NAME%pattern*}	(1)
${*NAME%%pattern*}	(2)

- gesucht wird von rechts nach links ($\leftarrow$);
  der linke (vordere) Teil ist der Substring
- *pattern* ist ein Muster, nach dem gesucht wird und kann die bei Dateinamen üblichen Wildcard-Zeichen enthalten *****,**?**,**[...]**
- Form (1) bedeutet Abtrennen hinter dem ersten Auftreten von *pattern* in **$***NAME* von rechts
- Form (2) heißt Abtrennen hinter dem letzten Auftreten von rechts

Beispiel:

```
MYBIN=/u/zdv/zdv045/bin
${MYBIN%z*}              => /u/zdv/
${MYBIN%%z*}             => /u/
${MYBIN%/*}              => /u/zdv/zdv045
${MYBIN%%/*}             => ""
SRC=/u/zdv045/pgm.f
OBJ=${SRC%.f}.o          =>/u/zdv045/pgm.o
```

Substrings von Variablen III

${*NAME*:start:laenge}

- Teilstring ab $start$ mit $laenge$ Zeichen
- **${*NAME*::laenge}**
  fehlt $start$, wird ab Position 0 mit $laenge$ Zeichen ausgegeben
- **${*NAME*:start}**
  fehlt $laenge$, wird ab $start$ bis zum Ende ausgegeben
- **${*NAME*: -start}**
  ist $start$ negativ, wird vom Ende nach vorne gezählt
  (das Blank zwischen **:** und **-** ist **zwingend**!)

Beispiel:

```
MYBIN=/u/zdv/zdv045/bin
${MYBIN:3:3}            => zdv
${MYBIN::7}             => /u/zdv/
${MYBIN:6}              => /zdv045/bin
```

Übungen

Schauen Sie sich folgende Shell-Scripts

- *length.demo*
- *l_substr.demo*
- *r_substr.demo*
- *substr.demo*

an und führen Sie die Scripts anschließend aus.
Prüfen Sie die Ausgabe.

Aufgabe (A6)

1. Schreiben Sie ein Shell-Script *dir_info*, das zum aktuellen Verzeichnis den Pfad und den Verzeichnisnamen ausgibt.

   Beispiel:

   ```
   aktuelles Verzeichnis: /home/unix/bin
   Pfad: /home/unix
   Name: bin
   ```

2. Schreiben Sie ein Shell-Script *mk_user*, das zu übergebenen Vor- und Nachnamen jeweils die Länge ausgibt.
   Aus den ersten 4 Buchstaben des Vor- und Nachnamens erzeugen Sie einen Usernamen und geben diesen aus.

   Aufruf: **`mk_user`** *vname nname*

Attribute und Umwandlung

typeset ***options variablenliste***

- die Attribute für die in *variablenliste* spezifizierten Variablen werden gesetzt
- ob eine Typumwandlung vorgenommen wird, falls der Wert nicht typgerecht ist, hängt vom Betriebssystem und der Shell ab
  (entsprechend unterschiedlich verhalten sich die Scripte)
- *options* spezifizieren den Typ und können u.a. sein
  (siehe hierzu `man typeset`):

-i*n*	Integerdarstellung der Variablen; *n* ist die Basis und kann zwischen 2 und 36 sein; Default ist 10 (Dezimalsystem)
-l	Kleinschreibung (Lowercase)
-u	Großschreibung (Uppercase)
-L*n*	linksbündig mit der Länge *n*
-LZ*n*	linksbündig ohne führende Nullen
-R*n*	rechtsbündig mit der Länge *n*
-RZ*n*	rechtsbündig mit führenden Nullen

Beispiele

```
# Integer Konvertierung
typeset -i KK JJ I
KK=4711     => $KK = 4711
JJ=chanel   => irgendein Wert
I=1.0       => syntax error...
A=1.22
I=$A        => syntax error...

# Gross- Kleinschreibung
typeset -u UP
typeset -l LOW
UP=Bond     => $UP = BOND
LOW=$UP     => $LOW = bond
```

```
# rechtsseitig mit fuehrenden
# Nullen
typeset -RZ3 NUM3
NUM3=7        => $NUM3 = 007

# Zuweisung String
STR=0123456789

# Teilstring von rechts
typeset -R4 RIGHT4
RIGHT4=$STR   => $RIGHT4 = 6789

# Teilstring von links
typeset -L3 LEFT3
LEFT3=$STR    => $LEFT3 = 012
```

Wertzuweisungen

Das Resultat bei nicht typgerechten Wertzuweisungen ist implementationsabhängig. Folgende Tabelle zeigt die Unterschiede bei Zuweisung auf einem Integertyp auf:

Wert	**SuSE Linux**	**klassische UNIX-Systeme (z.B. AIX)**
`4711`	`4711`	`4711`
`47.11`	syntax error	`47`
`47,11`	`11`	`11`
`chanel`	`0`	syntax error

Aufgabe (A7)

1. Schreiben Sie ein Shell-Script *addphone*, das als Parameter einen Namen und eine Telefonnummer erhält. Speichern Sie beide Informationen in eine Zeile einer Datei *phonebook*, die in Ihrem Heimatverzeichnis liegt. Der Name soll dabei großgeschrieben werden. Wenn das Script mehrmals aufgerufen wird, sollen die alten Einträge nicht zerstört werden.

   Aufruf: **addphone** *name number*

2. Schreiben Sie ein Shell-Script *lsphone*, das zu einem Namen alle Nummern auflistet. (Hinweis: Benutzen Sie `grep -i`.)

   Aufruf: **lsphone** *name*

3. Schreiben Sie ein Shell-Script *rmphone*, das zu einem Namen alle Einträge löscht.

   Hinweis: Benutzen Sie `grep -v`, legen Sie eine Zwischendatei an.

   Aufruf: **rmphone** *name*

Shell-Arithmetik

Überblick

1. Bourne Shell kompatibel
   - Benutzung von **`expr`**
   - Benutzung von **`bc`**[1]
     erwartet den Ausdruck als Pipe
     ```
     echo '4*7' | bc
     ```
     kann auch positive Dezimalrechnung
     ```
     (echo scale=10; echo '37/7') | bc
     ```
     kann Konvertierung von dezimal nach dual, oktal, hexadezimal
     ```
     (echo obase=16; echo 32) | bc
     ```
     man Page:
     ```
     man bc | a2ps -o bc.ps
     ```
2. ksh/bash Erweiterungen
   - Definition der Variablen mit **`typeset`**
   - Benutzung von **`let`** bzw. **`((...))`**
   - **`RANDOM`** liefert eine Zufallszahl

[1] Wird nicht weiter in diesem Kurs behandelt.

Operatoren

1 Arithmetische:

*	Multiplikation
/	Integer Division
%	Rest der Integer Division
+	Addition
-	Subtraktion
<<n	Shift nach links (mal 2^n)
>>n	Shift nach rechts (durch 2^n)

2 Boolsche:

&	bitweise Und
^, |	bitweise exkl. Oder, Oder
&&	logisches Und
||	logisches Oder

3 Logische (1 = richtig, 0 = falsch):

<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich
!=	ungleich

4 sonst:

=	Zuweisung

expr Kommando

expr ***expression***

oder

***var*=`expr** ***expression***`

- *expression* ist ein Ausdruck aus Variablen mit Integerwerten oder Integerkonstanten und Operatoren (auch geklammert):

  term1 *op* *term2* [*op* *term3* [...]]
- jedes Element des Ausdrucks muss durch ein Leerzeichen getrennt sein
- namesgleiche Operatoren mit Shell Meta-Zeichen wie *****,**<**,**>**,**(**,**)** müssen durch Hochkommata **'** **'** oder Backslash \ vor der Shell Expansion geschützt werden
- der Wertebereich ist von $-2^{31} \leq x \leq 2^{31} - 1$
- Backquotes **`...`** bewirken, dass das Ergebnis des **expr** Kommandos einer Variablen zugewiesen wird

expr – Beispiele

```
$ A=5
$ A=`expr $A + 4`
$ echo $A
 9
$ B=2
$ expr $A / $B
 4
# Multiplikation geht vor Addition
$ C=`expr '(' $A + $B ')' '*' 2`
$ echo $C
 22
# Rechenoperation geht vor Vergleich
$ D=4
$ expr $D == $B + $B
 1       (entspricht wahr)
$ expr $D == $D \* 3
 0       (entspricht falsch)
```

Übung

Folgende Ausdrücke werden im Shell-Script *arith.expr.demo* mit **expr** berechnet:

$$
\begin{aligned}
x &= 2 && => 2 \\
x &= x + 1 && => 3 \\
y &= (x + 1) * (x - 1) && => 8 \\
z &= \frac{x^2 + y^2}{4} && => 18 \\
v &= \frac{1 - y}{z^2} && => 0
\end{aligned}
$$

Was ist das Ergebnis von:

x sowohl kleiner als y als auch kleiner als z?

Was ist das Ergebnis von:

x kleiner oder gleich y oder z kleiner als v?

typeset Kommando I

typeset -i *variablenliste*
var=expression

- *expression* ist ein Ausdruck aus Variablen mit Integerwerten oder Integerkonstanten und Operatoren (auch geklammert):
  term1 op term2 [*op term3* [...]]
- zwischen Elementen des Ausdrucks darf **kein** Leerzeichen sein
- nur ein Klammerpaar zu Beginn eines Ausdrucks muss durch Hochkommata ' ' oder Backslash \ vor der Shell Expansion geschützt werden

Beispiele:

```
typeset -i K
K=3*(5+1)           => $K = 18
K=(5+1)*3           => $K = 6 => falsch!
K='('5+1')'*3       => $K = 18
K=\(5*(1+1)\)*4     => $K = 40
```

typeset Kommando II

- alle Shift-/Vergleichsoperatoren müssen durch Hochkommatas `' '` oder Backslash \ vor der Shell Expansion geschützt werden
  (sonst Interpretation als I/O Redirection)

  Beispiele:

```
typeset -i K
K=2'<<'2        => $K = 8
K=2<<2          => Prompt fuer IO (Here Document)
K=2'<='4        => $K = 1 => TRUE
K=2<=4          => Zuweisung wird ignoriert
                   =4: cannot open
```

- wahlweise referiert über Variablenname VAR oder Wert **$**$VAR$

  Beispiele:

```
typeset -i N1 N2 SUM1 SUM2
N1=2
N2=3
SUM1=N1+N2              => $SUM1 = 5
SUM2=$N1+$N2            => $SUM2 = 5
```

let bzw. ((...)) Kommando

- bei `let` bzw. `((...))` müssen keine Zeichen vor der Shell Expansion geschützt werden, weder Klammern im Ausdruck noch Shift- und Vergleichsoperatoren

  Beispiele:

```
let "K=(5+1)*3"  => $K = 18
let "K=2<<2"     => $K = 8
let "K=2<=4"     => $K = 1
                 => TRUE
```

```
((K=(5+1)*3))  => $K = 18
((K=2<<2))     => $K = 8
K=$((2<=4))    => $K = 1
               => TRUE
```

- wahlweise referiert über Variablenname VAR oder Wert **$** VAR

  Beispiele:

```
N1=2; N2=3
let "SUM1=$N1+$N2"      => $SUM1 = 5
SUM2=$((N1+N2))         => $SUM2 = 5
```

- es ist eine C-like Syntax möglich

  Beispiele:

```
a=1
let "a+=10"   => $a = 11
((a++))       => $a = 12
```

Übung

Formulieren Sie einige der Ausdrücke aus der vorherigen Übung mit `expr` durch Verwendung von `typeset` und `let` bzw. `((...))`.

(Ergebnis: *arith.typeset.demo* und *arith.let.demo*)

Aufgabe (A8)

Schreiben Sie ein Shell-Script *compute1*, das zwei Zahlen als Parameter erhält und folgende Werte bestimmt:

1. die Summe der beiden Zahlen,
2. das Produkt der beiden Zahlen,
3. eine 0 ausgibt, falls die Summe ungerade ist,
   eine 1, falls die Summe gerade ist.

Rufen Sie Ihr Script mit unterschiedlichen, auch negativen Zahlen auf.

Aufruf: **`compute1`** *zahl1 zahl2*

Kontroll-Statements

Prüfen von Informationen - test, [...], [[...]]

Bourne Shell kompatibel:

test ***condition***

oder:

[***condition*** **]**

ksh/bash Erweiterung:

[[***condition*** **]]**

- die eckigen Klammern `[...]` bzw. `[[...]]` sind Befehle und keine Symbole für wahlweise Spezifikation
- Leerzeichen hinter der öffnenden bzw. vor der schließenden Klammer sind **zwingend**
- dient zur Prüfung und Abfrage von:
  - Datei Informationen (Existenz, Größe, Rechte, Typ)
  - Variablen und Zeichenketten
    (Länge, Gleichheit, numerische und logische Vergleiche)

Optionen

- liefert als Ergebnis einen Exit-Status **$?**:
  0 wenn Bedingung erfüllt (true)
  ungleich 0 wenn Bedingung nicht erfüllt (false)
- die einfachste Form von *condition* kann u.a. wie folgt sein und ist wahr, wenn gilt:
  bei Dateien:

-f ***file***	Datei existiert und ist ein regulärer File
-d ***file***	Datei existiert und ist ein Directory
-s ***file***	Datei existiert und ist nicht leer
-r ***file***	Datei existiert und ist lesbar
-w ***file***	Datei existiert und ist schreibbar
-x ***file***	Datei existiert und ist ausführbar
-L ***file***	Datei existiert und ist ein Symbolic Link
f1 **-ot** ***f2***	Datei *f1* ist älter als Datei *f2*
f1 **-nt** ***f1***	Datei *f1* ist jünger als Datei *f2*

Optionen/Operatoren

bei Zeichenketten:

-z *s1*	*s1* ist ein Leerstring
-n *s1*	*s1* enthält mindestens ein Zeichen
s1	*s1* enthält mindestens ein Zeichen
s1* = *s2	*s1* ist identisch mit *s2*
s1* == *s2	*s1* ist identisch mit *s2*
s1* != *s2	*s1* ist ungleich *s2*

bei Integervariablen:

n1* -eq *n2	*n1* ist gleich *n2*
n1* -ne *n2	*n1* ist ungleich *n2*
n1* -gt *n2	*n1* ist größer als *n2*
n1* -ge *n2	*n1* ist größer oder gleich *n2*
n1* -lt *n2	*n1* ist kleiner als *n2*
n1* -le *n2	*n1* ist kleiner oder gleich *n2*

mehrere Bedingungen

- *condition* kann aus mehreren einfachen *conditions* zusammengesetzt werden, die durch boolesche Operatoren verknüpft werden. Dabei gilt:

  für `test` bzw. `[...]` :

`!` *condition*	logisches nicht
cond1 `-a` *cond2*	logisches UND
cond1 `-o` *cond2*	logisches ODER

  für `[[...]]` :

cond1 `&&` *cond2*	logisches UND
cond1 `||` *cond2*	logisches ODER

Erweiterungen

- bei einem numerischen Vergleich können sowohl die Leerzeichen als auch das $-Zeichen vor Variablen weggelassen werden

  (siehe `let` bzw. `((...))` , siehe Folie 75)

  Beispiel: `((anzahl==20)) && ...`
- ein Mustervergleich ist unter folgenden Bedingungen möglich:
  - das Muster darf **nur rechts** stehen
  - das Muster darf **nicht** in "..." stehen
  - der Ausdruck ist wahr, wenn sich die linke Seite durch das Muster beschreiben lässt
  - Metazeichen (*,?, Folie 7), Klassen (z.B. [0-9], Folie 8) und Konstrukte (*(muster1|muster2|...) etc., Folien 9-10) sind möglich

  Beispiele:

```
[[ $n == [0-9] ]] && echo "einstellige Zahl"
[[ $n == [1-9]+([0-9]) ]] && echo "mehrstellige Zahl"
[[ $VAR == +([a-z]) ]] && echo "Buchstaben"
```

Beispiele

```
# Gibt es Parameter?
  [ $# = 0 ]
# Wird Online Help verlangt?
  [ "$1" = "-?"  -o  "$1" = "-help" ]
  [[ "$1" = "-?"  ||  "$1" = "-help" ]]
  [ "$1" = "-?" ] || [ "$1" = "-help" ]
# Existiert Datei und ist lesbar?
  FILE=myfile
  [ -r $FILE ]
# Existiert Datei nicht?
  [ ! -f $FILE ]          => normale Datei
  [ ! -s "$FILE" ]        => auch Directory
# Ist Zeichenstring leer?
  HOST=
  [ -z $HOST ]            => Fehler: specify ...
  [ -z "$HOST" ]
# Ist Zeichenstring gesetzt?
  [ $HOST = '' ]          => [ = ] Fehler
  [ $HOST = 'ABC' ]       => [ = ABC ] Fehler
  [ "$HOST" = 'ABC' ]     => richtig
  [ $HOST. = 'ABC.' ]     => richtig
  [ $HOST. = `hostname`. ]
```

Bedingte Ausführung - if I

```
if   command_list                                    (1)
then
  command_list_true
fi
```

```
if   command_list                                    (2)
then
  command_list_true
else
  command_list_false
fi
```

- bedingte Ausführung von Befehlen, abhängig vom Exit-Status von *command_list*:
  - gleich 0 → `then`-Zweig
  - andernfalls → `else`-Zweig
- *command_list* kann sein:
  - eine Abfrage der Form `test`, `[...]`, `[[...]]`
  - ein Shell-Kommando bzw. eine Liste von Shell-Kommandos; dann gilt der Exit-Status des letzten Befehls

Bedingte Ausführung - if II

- Schlüsselwörter **then**,**else**,**fi** sollten in separaten Zeilen stehen
- Ausnahmen sind möglich
- auf **then**/**else** können direkt Kommandos folgen
  (übersichtlicher bei Einzeilern)

```
if id $1 >/dev/null 2>/dev/null
  then echo "User $1 exists"
  else echo "User $1 doesn't exist"
fi
```

- **then** kann durch **;** getrennt hinter **if** stehen
  (übersichtlicher, wenn nur ein Fall)

```
if [ $# = 0 ] ; then
  echo "No parms specified"
fi
```

- eine *nop*-Anweisung, auch Pseudobefehl, ist durch **:** möglich
  (Vermeidung von Nicht-Abfragen)

```
if test -d $1
then :
else
  mkdir $1; chmod a+x $1
fi
```

Bedingte Ausführung - if III

- geschachtelte `if`-Statements sind möglich:

```
if [ -d $I ]
then echo "$I ist ein Directory"
else
  if [ -f $I ]
  then echo "$I ist eine Datei"
  else
    echo "$I ist weder Datei noch Directory"
  fi
fi
```

- eine kürzere Schreibweise dafür ist:

if ***command_list1*** (3)
then
command_list_true1
elif ***command_list2***
then
command_list_true2
...
else
command_list_false
fi

Aufgabe (A9)

1. Ändern Sie Ihr Script *compute1* aus Aufgabe 8 so ab, dass Sie ausgeben:
   - "Die Summe ist gerade“, wenn die Summe gerade ist
   - "Die Summe ist ungerade“, wenn die Summe ungerade ist.

   Speichern Sie es unter dem neuen Namen *compute2*.

   Rufen Sie Ihr Script mit unterschiedlichen, auch negativen Zahlen auf.
2. Ändern Sie Ihr Script *rmphone* aus Aufgabe 7.3 so ab, dass Sie zuvor prüfen, ob es mehr als einen Eintrag gibt. Nur wenn genau einer existiert, soll er gelöscht werden; ansonsten geben Sie eine Meldung aus. Speichern Sie das geänderte Script als *rmphone2*.

   Hinweis: Benutzen Sie `grep -c`.

Schleife mit fester Wiederholung - for I

```
for   VAR   [ in   words ]
do
      command_list
done
```

- die Klammern [...] bedeuten, dass die Angabe optional ist
- fehlt die Angabe von $words$, wird die Schleife für die übergebenen Parameter (**`$1 $2 ...`**) ausgeführt
- wiederholt $command\_list$ für jedes Wort ($w1\ w2\ \ldots$) im Zeichenstring $words$ bzw. für jeden Übergabeparameter
- die Shell Variable VAR erhält jeweils das aktuelle Wort als Schleifenindex
- Trennzeichen für die Worte ($w1\ w2\ \ldots$) sind Leerzeichen, Tab oder Newline (definiert über Variable **`IFS`**; kann neu definiert werden, z.B. **`IFS=":"`**)

Schleife mit fester Wiederholung - for II

```
for  VAR  in  {start..ende}
do
      command_list
done
```

- Schleife über feste Anzahl von Werten von $start$ bis $ende$
- die Shell Variable VAR erhält jeweils den aktuellen Wert als Schleifenindex

```
for  ((VAR=start;VAR<=ende;VAR++))
do
      command_list
done
```

- C-like Syntax
- Schleife über feste Anzahl von Werten von $start$ bis $ende$
- die Shell Variable VAR erhält jeweils den aktuellen Wert als Schleifenindex

for – Beispiele

1
```
# Schleife ueber Aufrufparameter $1, $2, ...
for I
do echo "Parm is: $I"
done
```

2
```
# Schleife ueber 3 Worte
DAT="dat1 dat2 dat3"
for I in $DAT; do
  if [ -d "$I" ]; then echo "$I Dir";fi
done
```

3
```
# Schleife von 1 bis 10
for I in {1..10}
do echo $I
done
```

Beispiele/Übung

4
```
# Schleife ueber Ausgabe des ls-Befehls
for I in $(ls)
do echo "Datei ist $I"
done
```

5
```
# Schleife ueber Inhalt der Datei data
for I in `cat data`
do echo "Data is: $I"
done
```

Übung

Legen Sie eine Datei *data* mit folgendem Inhalt an:

```
Dies ist die 1. Zeile der Datei
und hier folgt die 2. Zeile
```

Benutzen Sie dazu einen Editor, I/O Redirection oder die Möglichkeit des Here-Documents. Probieren Sie das 5. Beispiel (s.o.) mit dieser Datei aus.

Schleife solange Bedingung wahr - while

```
while   list
do
        command_list
done
```

- die `while`-Schleife wird wiederholt, **solange** der Exit-Status von *list* **gleich 0** ist, also *list* erfolgreich war
- *list* kann sein:
  - eine Abfrage der Form `test`, `[...]`, `[[...]]`
  - ein Shell-Kommando
  - eine Liste von Shell-Kommandos; dann gilt der Exit-Status des letzten ausgeführten Kommandos
- *command_list* sind ein oder mehrere Befehle, die wiederholt werden; es können Unix Befehle oder Shell-Kontroll-Statements sein

while – Beispiele

1
```
# Schleife von 1 bis 100
# C-like Syntax
N=100
A=1
while ((A<=N)); do
  echo "Wert von A = $A"
  ((A+=1))
done
```

2
```
# Abarbeiten der Parameterliste
while [ $# -ne 0 ]
do
  echo $1
  shift
done
```

3
```
# Endloses Einlesen von Konsole bis EOF
while read INP
do
   if [ "$INP" = "EOF" ] ; then break ; fi
   echo $INP
done
```

Übung

Korrigieren Sie die Fehler im Script *while.demo*.

Schleife bis Bedingung wahr - until

```
until   list
do
        command_list
done
```

- die `until`-Schleife wird solange wiederholt, **bis** der Exit-Status von *list* **gleich 0** ist, also bis *list* eingetreten ist
- *list* kann sein:
  - eine Abfrage der Form `test`, `[...]`, `[[...]]`
  - ein Shell-Kommando
  - eine Liste von Shell-Kommandos; dann gilt der Exit-Status des letzten Kommandos
- *command_list* sind ein oder mehrere Befehle, die wiederholt werden; es können Unix Befehle oder Shell-Kontroll-Statements sein

until – Beispiele

1
```
# Abarbeiten der Parameterliste
I=1
until [ $# -eq 0 ]
do
  echo "$I-ter Parameter: $1"
  shift
  ((I=I+1))
done
# Skript ausführen:
$ ksh prtparm a1 a2 a3
1-ter Parameter: a1
2-ter Parameter: a2
3-ter Parameter: a3
```

2
```
# C-like Syntax
LIMIT=10
VAR=0
until (( VAR>LIMIT ))
do
  echo $VAR
  ((VAR++))
done
```

3
```
# Test till someone logged in
until who | grep $1 >/dev/null 2>&1
do
   sleep 60
done
echo "$1 has logged in"
```

Sprung ans Schleifenende - break

break [*n*]

- bewirkt ein Verlassen einer **for**, **while** oder **until** Schleife
- *n* gibt bei geschachtelten Schleifen an, wieviele Schleifen beendet werden sollen; Default ist 1
- es wird hinter das entsprechende **done** Statement verzweigt

Beispiel:

```
# Endlos-Prompt fuer Ja/Nein Eingabe
while :
do
  echo "OK to delete $DIR? \c"
  read ANSWER
  [[ $ANSWER == y* ]] && rmdir $DIR && break
  [[ $ANSWER == n* ]] && echo $DIR kept && break
  echo "Enter y or n"
done
```

Sprung an Schleifenanfang - continue

continue [*n*]

- bewirkt, dass der Rest einer **for**, **while** oder **until** Schleife übersprungen und mit einer neuen Iteration begonnen wird
- *n* gibt bei geschachtelten Schleifen an, bei welcher Schleife wieder aufgesetzt werden soll; Default ist 1

Beispiel:

```
# Verteilen von Files an Benutzer
cat table | while read UID DIR
do
  mkdir /usr/$UID/$DIR || continue
  for FILE in .prof .dsm .init
    if cp /usr/std/$FILE /usr/$UID/$DIR
    then :
    else
      echo " Copy $FILE to $UID failed"
      continue 2
    fi
  done
done
```

Aufgabe (A10)

Ändern Sie Ihr Script *compute2* aus Aufgabe 9 so ab, dass Sie die beiden Zahlen jeweils paarweise einlesen.
Syntax:

```
while read A B
do
   ...
done
```

Lesen Sie solange neue Zahlenpaare ein, bis Sie keine gültige Integerzahl, sondern **`quit`**, **`stop`** oder **`end`** eingeben.
Speichern Sie das geänderte Script auf *compute3*.

Fallunterscheidung - case

```
case   $VAR   in
       pattern1  )  command_list1 ;;
       pattern2  )  command_list2 ;;
       ...
esac
```

- $\$VAR$ wird von oben nach unten mit den $pattern_n$ verglichen
- bei Übereinstimmung wird die entsprechende Folge von Kommandos $command\_list_n$ ausgeführt; die restlichen $pattern$ werden übersprungen
- für den Mustervergleich in $pattern_n$ können Wildcard-Zeichen (`*`, `?`, `[ ]`) verwendet werden
- mehrere $pattern$ können durch logische ODER (|) verknüpft werden
- sie müssen auf eine Zeile passen und mit ")" enden
- $command\_list_n$ wird ausgeführt, wenn ein $pattern$ zutrifft

case – Beispiele

```
# 1.) Prüfen der Eingabe
read ANSWER
case $ANSWER in
    y* | Y*) echo "Yes requested";;
    n* | N*) echo "No requested";;
    *      ) echo "invalid answer";;
esac
# Hinweis: *) agiert wie ein "Otherwise"

# 2.) Prüfen von Dateinamen
for I in $(ls)
do
  case $I in
    a*   ) echo "Datei fängt mit a an";;
    *.f  ) echo "FORTRAN Programm";;
    s.*.f) echo "s-FORTRAN Programm";;
    *) ;;
  esac
done
```

Achtung: Die Reihenfolge ist entscheidend! Im 2. Beispiel wird s.*.f nie gefunden, da es bereits in *.f enthalten ist.

Menüs - select

```
select VAR in words
do
    command_list
done
```

- die Worte in $words$ werden als nummerierte Liste ausgegeben
- als Eingabe wird die Nummer der gewünschten Auswahl erwartet
- die eingegebene Nummer steht auf der Shell-Variablen **REPLY** zur Verfügung
- das zugehörige Wort aus $words$ steht auf VAR zur Verfügung
- der Prompt-String ist standardmäßig **#?**, kann aber mittels der Shell Variablen **PS3** umgesetzt werden
- **`Ctrl-d`** beendet die Schleife

Menüs - select – Beispiele

```
# Script prtdoc zum Drucken von Dokumenten
PS3="Enter your number:"
select NUM in "FORTRAN Doc" "C Doc" \
       "Text Doc" "ADSM Doc" "End"
do
  case $REPLY in
    1) lpr f90.ps;;
    2) lpr gnu_cc.ps;;
    3) lpr tex.ps;;
    4) lpr backup.ps;;
    5) exit 0;;
  esac
done
```

```
$ ksh prtdoc
1) FORTRAN Doc
2) C Doc
3) Text Doc
4) ADSM Doc
5) End
Enter your number:
```

```
# Auswahl aus FORTRAN Programmen
select PGM in *.f
do
  f77 -o ${PGM%.f}.o $PGM
done
```

Fehlerbehandlung I

- jeder Befehl liefert am Ende einen Exit-Status auf der Shell Variablen **$?**
- folgende Fehler können auftreten:
  - I/O Redirection fehlerhaft (1)
  - Befehl existiert nicht (1)
  - Befehl ist nicht ausführbar (1)
  - Befehl endete nicht normal (128 + signal code)
  - Befehl endete fehlerhaft (*n*)
- direkte Abfrage des Exit-Status:

```
command
if [ "$?“ = "0“ ]
    then command1
    else command2
fi
```

führt *command1* aus, wenn von *command* der Exit-Status 0 ist, andernfalls wird *command2* ausgeführt

Fehlerbehandlung II

- indirekte Abfrage des Exit-Status:
  - **if** ***command***
    then ***command1***
    else ***command2***
    fi
    führt *command1* aus, wenn der Exit-Status von *command* 0 ist, andernfalls wird *command2* ausgeführt
  - ***command1 && command2***
    führt *command2* nur aus, wenn der Exit-Status von *command1* 0 ist
  - ***command1 || command2***
    führt *command2* nur aus, wenn der Exit-Status von *command1* ungleich 0 ist

Debugging und Tracing

Setzen von Shell Optionen

- zwei Shell Optionen steuern das Tracing:

  -x Ausgabe jeder Zeile vor der Ausführung, aber nach der Interpretation durch Shell

  -v Ausgabe jeder Zeile vor der Ausführung und vor der Interpretation durch die Shell

- ein **+** am Anfang kennzeichnet die Trace-Ausgabe
- das Setzen und der Aufruf in der aktuellen Shell erfolgt durch:

```
set options
. filename
```

- für eine neue Shell können die Optionen auch beim Aufruf des Scripts mitgegeben oder im Script gesetzt werden:

```
ksh options filename
```

- beendet wird das Tracing durch Beenden der Shell bzw. durch:

```
set +options
```

Variablen zum Debugging

PS4	mit **PS4** kann die Zeilenkennzeichnung (+) geändert werden
LINENO	enthält die Zeilennummer der aktuellen Zeile
ERRNO	enhält die Fehlernummer des zuletzt fehlgeschlagenen System Calls

Listing *set_x.demo*:

```
#!/bin/ksh
PS4='$LINENO: '
set -x
A=1
B=`expr $A + 1`
echo "Inkrement von A ist: $B"
```

Aufruf (im Linux):

```
$ set_x.demo
4: A=1
5: expr 1 + 1
5: B=2
6: echo Inkrement von A ist: 2
Inkrement von A ist: 2
```

Aufgabe (A11)

1. Fügen Sie an den Anfang ihres Scripts *compute3* die Definition von **PS4** (wie im Beispiel) und das Starten des Trace ein. Rufen Sie danach Ihr Script auf.
2. Prüfen Sie in Ihrem Script *rmphone2* aus Aufgabe 9.2, ob das Erstellen der Zwischendatei korrekt gelaufen ist. Erst dann soll sie auf die Originaldatei umbenannt werden. Falls ein Fehler auftrat, soll die Zwischendatei gelöscht werden, wenn sie existiert. Speichern Sie das geänderte Script unter *rmphone3*.
   Hinweis: Zum Testen schreiben Sie die Zwischendatei in das Verzeichnis */home*

Shell für Fortgeschrittene

Variablensubstitution I

- neben der normalen Zuweisung eines Zeichenstrings an eine Variable gibt es Möglichkeiten der Substitution
- Substitution bei Nullstring mit Wertänderung

  ${*VAR*:=*value*}

  - falls VAR leer ist, wird der Ersatzwert $value$ benutzt und VAR auch zugewiesen
  - falls VAR besetzt ist, wird der Originalwert genommen und bleibt unverändert
- Substitution bei Nullstring ohne Wertänderung

  ${*VAR*:-*value*}

  - falls VAR leer ist, wird der Ersatzwert $value$ benutzt, aber nicht VAR zugewiesen
  - falls VAR besetzt ist, wird der Originalwert genommen und bleibt unverändert

Variablensubstitution II

Beispiel:

```
# Create file and mark as Version 2
# If no new name is given,
# use the old one plus extension v2
# Format: crtvx file [ext]
DEST=${2:-$1.v2}
echo "# Version 2" > $DEST
cat $1 >> $DEST
```

- Substitution bei Wert ohne Wertänderung

${*VAR*:+*value*}

  - falls VAR leer ist, wird Nullstring benutzt und bleibt unverändert
  - falls VAR besetzt ist, wird der Ersatzwert $value$ genommen, aber nicht VAR zugewiesen

Variablensubstitution III

- Meldung und Exit

  ${*VAR*:?*value*}

  - falls VAR leer ist, wird $value$ ausgegeben und das Script abgebrochen (Fehlermeldung)
  - falls VAR besetzt ist, wird der Originalwert unverändert benutzt

  Beispiel:

```
VAR=''
echo ${VAR:?"variable has no value"}
--> VAR: variable has no value
```

Variablensubstitution IV

- Prüfung auf Definition statt auf Wert
  Die folgenden Variablenausdrücke entsprechen den oben genannten in der Funktion, nur dass sie auf Definition der Variablen testen und nicht auf den Wert.

  ${*VAR*=*value*}
  ${*VAR*-*value*}
  ${*VAR*+*value*}
  ${*VAR*?*value*}

- Die Definition einer Variablen kann mit folgendem Befehl rückgängig gemacht werden

  unset ***VAR***

Beispiel

Beispiel:

```
DIR='/usr/local/bin'
echo ${DIR?}
unset DIR
echo ${DIR?}

--> /usr/local/bin
    DIR: Parameter null or not set.
```

Übung

Listen Sie das Shell-Script *substitute.demo* auf und veranschaulichen Sie sich die Ersetzungen.
Rufen Sie das Script auf.

Zerlegen von Zeichenketten - set

set ***word1*** **[*word2* [...]]**

- eine bequeme Art Zeichenketten zu parsen; besonders dann, wenn die einzelnen Worte durch unterschiedliche Anzahl Blanks getrennt sind (**`cut`** versagt dabei)
- die Variablen $word_n$ werden den Parameter-Variablen **$**$n$ zugewiesen (**$1** enthält $word1$, ...)
- die Variablen **$*** und **$#** werden aktualisiert
- wenn $word_n$ leer ist oder fehlt, werden die gesetzten Shell-Variablen aufgelistet
- wenn der Wert von $word1$ mit einem **-** beginnt, wird $word1$ als Option für den **`set`** Befehl interpretiert

  Umgehung: **`set -- word1 [word2 [...] ]`**

Beispiele

1
```
# Setzten der Parametervariablen
set fruits:  apple banana    peach

-->   $1=fruits: $2=apple $3=banana $4=peach
       $#=4
```

2
```
# Einlesen einer Datei und aufsplitten jeder Zeile
I=0
cat table | while read INP
do
  I=`expr $I + 1`
  if [ "$INP" = "" ]
     then continue
     else set $INP
  fi
  echo "Worte in Zeile $I: $# 1.Wort: $1"
done
```

Zweistufige Ersetzung - eval

eval ***VAR1*** **[*VAR2* [...]]**

- normalerweise testet die Shell nur einstufig auf Wildcard-Zeichen,Variablen, Kommandos
- **`eval`** bewirkt ein Rescan, so dass nochmals Ersetzungen stattfinden; danach wird die Zeile als Kommando ausgeführt

Beispiele:

1
```
FORMAT='+%A, %d %B %Y, %T'
DATUM='date "$FORMAT"'
echo $DATUM
eval $DATUM
```

2
```
# Change number of month to string
M1=Jan M2=Feb M3=Mar M4=Apr M5=May M6=Jun
M7=Jul M8=Aug M9=Sep M10=Oct M11=Nov M12=Dec
echo "Month number? " ; read MON
eval echo '$M'$MON
```

Zweistufige Ersetzung bash - ${!*POS*}

Die bash liefert noch eine weitere Möglichkeit, in einer zweistufigen Ersetzung auf Positionalparameter zuzugreifen:

echo ${!*POS*} [...]

- *POS* enthält eine Zahl, die im ersten Schritt ersetzt wird
- im zweiten Schritt wird der entsprechende Positionalparameter ersetzt

Beispiele:

```
# Zerlegen der Ausgabe des ls-Befehls
ls -l | while read INP
do
  set -- $INP
  echo "$# Worte - erstes Wort: $1 - letztes Wort: ${!#}"
done
```

Aufschlüsseln von Options - getopts I

getopts ***optstring*** ***VAR*** **[*args*]**

- `getopts` dient dazu Optionen und Argumente, die dem UNIX Standard entsprechen, zu parsen
  Beispiel: `cmd -o1o2 a2 -o3 a31,a32 p1 p2 ...`
- *optstring* ist die Liste der erlaubten Optionen
  - ein Zeichen pro Option ist zulässig
  - hat eine Option Argumente, ist dies durch Doppelpunkt ":" zu kennzeichnen
  - Argumente für Optionen sind niemals optional

  Beispiel: `om:a` oder `m:ao`

  → 2 Optionen (o,a) ohne Argumente,
  1 Option (m) mit Argumenten
- *VAR* ist die Shell-Variable, die das Zeichen der gerade gefundenen Option enthält
- *arg* ist der Zeichenstring, der zu parsen ist (normalerweise **$***)

Aufschlüsseln von Options - getopts II

- `getopts` muss wiederholt aufgerufen werden, bis zum Ende der im Aufruf angegebenen Optionen
- das Ende wird durch einen Exit-Status ungleich 0 markiert
- jeder Aufruf liefert von links nach rechts ein oder zwei Argumente
- es werden 3 lokale Variablen gesetzt:

VAR	Name der Option
OPTARG	Argumente der Option
OPTIND	Index des nächsten Positionalparameters, der noch nicht abgearbeitet wurde

Beispiel:

```
while getopts ab:c OPT; do
  case $OPT in
    a | c) echo $OPT: $OPTIND;;
    b)     echo $OPT: $OPTARG $OPTIND;;
  esac
done
shift $(expr $OPTIND - 1)
echo "$1 $2 $3 ..."
```

Aufschlüsseln von Options - getopts III

- falsche Optionen werden standardmäßig durch die Shell abgefangen:
  getopts: *opt* ***bad option(s)***
- eine eigene Behandlung ist möglich durch:
  - Doppelpunkt an den Anfang von *optstring*
  - Abfragen von **?**, weil die Variable dann den Wert **?** hat
  - Zeichen der falschen Option steht dann auf **OPTARG**

Listing *getopts.demo*:

```
while getopts :ab:cyz OPT
do
  case $OPT in
    a | c) echo $OPT: $OPTIND;;
    b )    echo $OPT: $OPTARG $OPTIND;;
    \?)    echo invalid Option $OPTARG;;
    * )    echo other Option $OPT;;
  esac
done
((OPTIND--)); shift $OPTIND
echo "$1 $2 $3 ..."
```

Shell Funktionen

Definition

function_name **()** **{** ***command_list*** **}**	**(1)**
	oder
function ***function_name*** **{** ***command_list*** **}**	**(2)**

- bei Schlüsselwort `function` dürfen **keine ( )**
- **ohne function** kennzeichnen `()` den Namen als Funktion
- `{...}` enthält die Anweisungen der Funktion
- Shell Funktionen gelten für die aktuelle Shell
- die Definitionen sollten am Anfang des Shell-Scripts stehen
- sie werden bei der Interpretation in den Speicher geladen
- ausgeführt werden sie Shell-intern
  - keine neue Shell
  - alle Variablen sind bekannt (außer Positionalparameter **$1**,...)
  - Änderungen von Variablen wirken auch nach Funktionsende

Shell Funktionen

Such-Hierarchie

1. reserviertes Wort (z.B. **`case, for, if, until, while,`** ...)
   die entsprechende Shell Routine verarbeitet die weitere Eingabe für die damit verbundene Aktion
2. Built-in Kommando
   (z.B. **`break, cd, echo, kill, read, set, shift,`** ...)
3. Shell-Funktion
4. Alias (csh, ksh)
   die durch den **`alias`** Befehl definierte Aktion wird ausgeführt
5. Beginnt die Eingabe mit "/", wird das durch den absoluten Pfad spezifizierte Kommando ausgeführt.
   Andernfalls wird in den in **`$PATH`** angegebenen Directories das entsprechende Kommando gesucht.

Hinweis: Die Such-Hierarchie ist noch **vor** dem Suchpfad von Kommandos!

Beispiel: Eine Funktion names ***ls*** wird vor dem Unix Kommando **`ls`** ausgeführt, das nur noch mit vollem Pfadnamen */usr/bin/ls* erkannt wird.

Shell Funktionen

lokale Variablen

Definition lokaler Variablen in Funktionen in der bash:

[function] ***function_name*** **[()] {**
    local *VAR1*=...
oder:
    typeset *VAR2*=...
}

- mit `local` oder `typeset` sind die angegebenen Variablen nur lokal in der Funktion definiert
- mit Beenden der Funktion verfallen die Variabeln
- gibt es gleichnamige globale Variablen, werden in der Funktion die **lokalen** verwendet
- jede Funktion hat eigene Positionalparameter **$1**, **$2**,...

Shell Funktionen

Aufruf

function_name **[*args*]**

- es können beliebig viele Parameter übergeben werden
- jede Funktion hat eigene Positionalparameter **$1**, **$2**,...
- die Positionalparameter vom Aufruf des Scripts bleiben unverändert

Beispiel:

```
flaeche() {
  ((FL=$1*$2))
  echo "Flaeche: $FL"
}
...
flaeche 2 2    # => liefert Flaeche: 4
...
flaeche $1 $2  # => Positionalparameter des Scripts!
...
```

Shell Funktionen

Verlassen / Beenden

return [*n*]

oder:

exit [*n*]

- mit `return` verläßt man die Shell Funktion und kehrt hinter den Aufruf zurück
- `exit` beendet das Shell-Script und kehrt zur rufenden Shell zurück
- n ist der Exit-Status, der mitgegeben werden kann; wird n nicht angegeben, wird der Exit-Status des zuletzt in der Funktion ausgeführten Kommandos genommen (n: 0-255)
- eine Shell Funktion wird automatisch verlassen, wenn das Ende `}` erreicht wird; dies wirkt wie `return`

Shell Funktionen

Beispiele I

1. Emulation des alias-Befehls in der Bourne Shell

```
# Wunsch: rm -i als alias fuer rm
rm()
{
  /bin/rm -i $* # absoluter Pfad notwendig
}
```

2. Beim Verzweigen in das angegebene Directory soll ein Inital File *.init* ausgeführt werden. Wenn die Datei nicht existiert, wird eine Datei-Liste erstellt (sinnvoll zum Testen von Voraussetzungen für Andwendungen).

```
function to {
  cd $1
  if [ -s .init ]
    then . .init
    else /usr/bin/ls -aF
  fi
}
```

Shell Funktionen

Beispiele II

3 gleichnamige globale und lokale Variablen

```
#!/bin/bash
# lokale Variablen in der Funktion
info ()
{
  local host=$1
  local count=5
  ...
}
...
count=10
...
info zam859
echo $count  # => count = 10, da die lokale Variable
             #     nicht mehr bekannt ist
...
```

Shell Funktionen

Rekursive Aufrufe

- rekursive Aufrufe von Shell-Scripts oder Shell Funktionen sind möglich
- es muss ein Abbruchkriterum vorhanden sein, sonst werden unendlich viele Prozesse erzeugt

Beispiel: Das folgende Beispiel ist nicht sehr sinnvoll, veranschaulicht jedoch die Rekursion

```
# Funktion: Ausgabe der Eingabeparameter
# Format:   showargs p1 [p2 [...]]
#
echo $1
if [ $# -gt 1 ]; then
   shift 1
   showargs $*
fi
```

Aufgabe (A12)

Wandeln Sie Ihr Script *compute3* so ab, dass Sie die Bestimmung, ob das Ergebnis gerade oder ungerade ist, in eine Shell Funktion legen, und bestimmen Sie dies nun sowohl für die Summe als auch für das Produkt.

Das geänderte Script speichern Sie unter *compute5*.

Signalbehandlung

- Signale werden mit **`kill`** an Prozesse geschickt

  kill ***signal pid***

- durch Tastenkombinationen wie **`Ctrl-c`** oder **`Ctrl-z`** ausgelöst
- Signale haben Nummern und Namen; sie sind zum Teil standardisiert
  (AIX: /usr/include/sys/signal.h, Linux: /usr/include/bits/signum.h)
  → u.a.:

```
            0   /* shell/function exit
SIGHUP      1   /* hangup
SIGINT      2   /* interrupt
SIGQUIT     3   /* quit
SIGILL      4   /* illegal instruction
SIGKILL     9   /* kill
SIGALRM    14   /* alarm clock timeout
SIGTERM    15   /* software termination
SIGCONT    18   /* continue
SIGTSTP    19   /* interactive stop
SIGXCPU    24   /* cpu time limit exceeded
SIGXFSZ    25   /* file size limit exceeded
```

Signalbehandlung – nur sh

- Shell-Scripts können diese Signale abfangen und ggf. darauf reagieren; danach sollte das Shell-Script immer mit `exit` beendet werden
- die Signalbehandlung steht in der Regel am Anfang des Shell-Scripts:

  trap *'command_list; exit'* ***signals***

- *command_list* ist die Befehlsfolge, die bei Auftreten des Signals durchlaufen werden soll
- ist *command_list* leer, so werden die entsprechenden Signale ignoriert
- fehlt *command_list*, wird die Signalbehandlung auf die Standardaktion zurückgesetzt (normalerweise Beenden des Script)
- *signals* sind ein oder mehrere durch Blank getrennte Signalnummern, für die die Aktion gelten soll
- ksh spezifisch gibt es symbolische Namen:

  HUP $\rightarrow$ 1, INT $\rightarrow$ 2, QUIT $\rightarrow$ 3, ..., KILL $\rightarrow$ 15 und EXIT $\rightarrow$ 0
- das Signal ERR (keine eindeutige Signalnummer) wird bei jedem nicht erfolgreichen Return Code (**$?**) eines Kommandos aktiv

Signalbehandlung – Beispiele

- Abfangen von `Ctrl-c` → Signal 2

```
#!/bin/sh
# funktioniert NUR mit /bin/sh
trap 'echo "terminated by user" ; exit' 2
```

- Ignorieren der Signale 1, 2 und 15

```
#!/bin/sh
# funktioniert NUR mit /bin/sh
trap '' 1 2 15
```

Aufgabe (A13)

Wandeln Sie Ihr Script *compute5* so ab, dass ein Interrupt von der Konsole (Signal 2) ignoriert wird.

Das geänderte Script speichern Sie unter *compute6*.

Inhalt

Literatur / Hilfe

Literatur I

- Patrick Ditchen: **Shell-Skript Programmierung**
  2003, mitp-Verlag, ISBN 3-8266-0883-6
  deutsch, mit CD
  **** (gut zum Nachschlagen/Lernen, Beispiele)
- J. Valley, S.G. Kochan, P.H. Wood:
  UNIX Desktop Guide to the Korn Shell
  1992, Hayden Books, ISBN 0-672-48513-3
  engl., 115-132, 165-242, 291-324
  **** (gut zum Nachschlagen/Lernen, Beispiele)
- Paul W. Abrahams, B.R. Larson: **UNIX for the Impatient**
  1992, Addison-Wesley, ISBN 0-201-82376-4
  engl.
  ** (gut zum Nachschlagen/Lernen, recht kurz)

Literatur / Hilfe

Literatur II

- Kaare Christian: **The UNIX Operating System**
  1988, John Wiley & Sons, ISBN 0-471-84781-x
  engl., 160-203
  ** (recht kurz, mehr Unix)
- B.W. Kernighan, R. Pike: **The UNIX Programming Environment**
  1984, Prentice Hall, ISBN 0-13-937681-x
  engl., 71-100, 133-170
  * (verstreut, viel Unix)
- J. Gulbins, K.Obermayr: **UNIX**
  1995, Springer Verlag, ISBN 3-540-58864-7
  dt., 335-370
  * (verstreut, viel Unix)
- U. Schmidt: **Korn Shell Programming**
  FZJ- ZAM- RFK- 0011 (Referenzkarte)
  WWW: http://www2.fz-juelich.de/jsc/docs/rfk_d.html

Literatur / Hilfe

Online Dokumentation

- Linux-Dokumentation
  man ksh
  $\rightarrow$ ca. 4000 Zeilen Dokumentation