Funktionen

Inhalt

Funktionen

- Wie in vielen anderen Programmiersprachen können auch in der Shell **Funktionen** definiert werden.
  - Haben einen **Namen** (*Groß/Kleinschreibung beachten*) und bestehen aus einer Folge von **Kommandos**.
  - Beim Aufruf sind **Argumente** übergebbar (*analog zu Skripten*).
  - Auch **eigene Kommandos** lassen sich damit realisieren (*auf Kommandozeile über Funktionsnamen aufrufbar*).
  - Müssen **vor** dem 1. Aufruf definiert werden (*außerhalb aller Kontrollstrukturen!*).

Funktionen

- Funktionen werden von der **aktuellen Shell** ausgeführt, es wird also im Gegensatz zum Aufruf eines Shell-Skriptes **kein neuer Prozeß** erzeugt.
  - D.h. **alle** in einem Skript definierten **Variablen** können in Funktionen verwendet und verändert werden.
  - Auf eine Funktion beschränkte **lokale Variablen** sind nur in der `bash` / `ksh` möglich, die Syntax lautet:

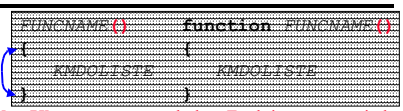
```
local VAR       (bash)
typeset -l VAR  (local ksh)
```


Funktionen

- Vorteile von Funktionen:
  - Programmteile lassen sich unter einem Namen **zusammenfassen** ("**kapseln**") und einfach **wiederverwenden**.
  - Das Programm kann in **logische Einheiten** mit **sauberen Schnittstellen** zerlegt werden (→ *leichter zu testen und zu warten*).
  - Ein Funktions-Aufruf ist im Gegensatz zu einem Skript-Aufruf **sehr schnell** (*da er keine Subshell startet*).
  - Die Funktionen eines Skripts können sich **gemeinsame Variablen** teilen.

Funktionen - Definition

- Syntax:



  - Die **runden Klammern** `()` nach dem Funktionsnamen sind notwendig und müssen leer sein.
  - Die in **geschweiften Klammern** { } eingeschlossene *KMDOLISTE* wird "**Funktions-Rumpf**" genannt.
  - Vor *FUNCNAME* kann das Schlüsselwort **`function`** stehen (*nur* `bash`/`ksh`).
  - Einrückung von *KMDOLISTE* ist sinnvoll.

Funktionen - Aufruf

- Die Kommandos im **Funktions-Rumpf** werden immer dann ausgeführt, wenn der **Name** der Funktion als Kommando (*ohne Klammern*) aufgerufen wird:

```
FUNCNAME
```

- **Beispiel**

```
hallo_welt()                    (Definition)
{
    echo "Hallo Welt"
}

hallo_welt                      (Aufruf)
```

Funktionen - Definition

- Funktionsdefinition in **einer Zeile**:

```
hallo_welt() { echo "Hallo Welt"; }
```

  - **Nach** dem { muß ein Leerzeichen stehen.
  - **Nach** dem letzten Kommando (*vor* }) muß ein **;** stehen.
- **Anzeige** aller / einer definierten Funktionen:

```
set (sh)  oder  typeset -f [FUNC] (bash / ksh)
```

- Funktion **löschen** mit `unset -f FUNC`
- Funktionen werden (*wie Aliase*) **nicht** an Subshells **vererbt**.

Funktionen - return

- `return [STATUS]`

  Beendet eine Funktion (*vorzeitig*), der Kontrollfluß kehrt wieder zur Aufrufstelle zurück.
  - Gibt **Exit-Code** `STATUS` oder den des letzten vorher ausgeführten Kommandos zurück.
  - Fehlt `return`, so wird die Funktion nach dem **letzten Kommando** im Funktions-Rumpf verlassen und dessen **Exit-Status** zurückgegeben.
  - `return` kann **beliebig oft** in einer Funktion vorkommen.

Funktionen - return

Beispiel

```
einmaleins()                                (Definition)
{
    if [ "$INPUT" = "1*1" ]
    then
        return 1
    elif [ "$INPUT" = "2*2" ]
    then
        return 4
    ...
    fi
}

INPUT="1*1"; einmaleins; echo $?        (Aufruf → 1)
INPUT="5*5"; einmaleins; echo $?        (Aufruf → 25)
```


Funktionen - Parameter-Übergabe

- Innerhalb einer Funktion kann über die **Positions-Parameter** `$1`, `$2`, ... auf die Argumente des Funktionsaufrufes zugegriffen werden.
  - Die **Argumente** sind beim Aufruf nach dem Funktionsnamen anzugeben (*ohne Klammern*):

```
FUNCNAME ARG1 ARG2 ...
```

  - Der **Skript-Name** `$0` bleibt unverändert.
  - **Außerhalb** von Funktionen sind die **normalen Skript-Argumente** wieder verfügbar (*als Parameter oder Variable an die Funktion übergeben, falls sie dort benötigt werden*).

Funktionen - Parameter-Übergabe

Beispiel

```
error()                                     (Funktions-Definition)
{
    if [ "$2" ]                             ($2 = Meldungstext)
    then
        echo "error: $2" 1>&2
    fi
    echo "usage: $0 FILE..." 1>&2           ($0 = Skript-Name)
    exit $1                                 ($1 = Exit-Code)
}

error 1 "Datei nicht gefunden"              (Funktions-Aufruf)
error 100 "Option -x unbekannt"
error 0                                     ($2 leer)
```


Funktionen - Werte-Rückgabe

- Die **Standard-Ausgabe** *aller* Kommandos (z.B. per `echo`) im Funktions-Rumpf kann als "**Ergebnis**" einer Funktion zurückgegeben werden, indem **Kommando-Substitution** verwendet wird:

```
VAR=`FUNCNAME`          (sh)
VAR=$(FUNCNAME)         (nur bash, ksh)
```

- Ein weiteres Ergebnis einer Funktion ist natürlich der **Exit-Status** des letzten Kommandos oder des Befehls `return STATUS` (*nur die Werte 0-255 möglich*).

Funktionen - Werte-Rückgabe

Beispiel

```
f1() { ls; echo "abc"; }        (Standard-Ausgabe)
VAR1=`f1`                       (Kommando-Substitution)

f2() { pwd; echo "def"; }       (Standard-Ausgabe)
VAR2=$(f2)                      (Kommando-Substitution)

f3() { return 100; }            (Exit-Code)
if f3
then
    echo "Ablauf ok"
else
    echo "Fehler trat auf"
fi
```


Übung 14

- **Zerlegen** Sie `menu.sh` in die **Funktionen** `PrintMenu`, `GetInput` und `ExecuteCmd`.
  - Verwenden Sie bei `ExecuteCmd` statt einer globalen Variable die **Übergabe** der Benutzer-Eingabe **als Parameter**.
  - Verwenden Sie bei `GetInput` statt einer globalen Variable die **Rückgabe** der Benutzer-Eingabe per `echo` + **Kommando-Substitution**.
  - Wie können Sie den **Abbruchfall** aus `ExecuteCmd` sauber an das Hauptprogramm zurückgeben? (`break` *in einer Funktion hat keine Wirkung!*)

Funktionen - Source-Operator

- Um Funktionen **mehrfach zu verwenden**, können sie in eine getrennte **Datei** ausgelagert werden.
  - Der **Source-Operator** liest diese Datei dann an beliebiger Stelle in Skripten ein:

```
DATEI                           (sh / bash / ksh)
source DATEI                    ((t)csh / bash / ksh)
```

  - Bei der **Suche** nach dieser Datei wird (*wie bei normalen Kommandos*) die **Pfad-Suche** über `$PATH` verwendet.

Funktionen - Source-Operator

Beispiel

```
# Inhalt der Datei include.sh
error()
{
    [ "$2" ] && echo "error: $2" 1>&2
    echo "usage: $0 FILE..." 1>&2
    exit $1
}

#!/bin/sh
. include.sh    # Datei include.sh einlesen
error 100 "Option -x nicht erlaubt"
```


Übung 15

- **Lagern** Sie die Funktionen von `menu.sh` in eine Datei `menu.inc` aus.
- Erstellen Sie eine Funktion `GetChar`, die das `dd`-Kommando zum Einlesen eines einzelnen Zeichens kapselt.
  - Verwenden Sie diese Funktion `GetChar` zur Abfrage der Benutzereingabe in `menu.sh` und lagern Sie sie ebenfalls nach `menu.inc` aus.