Werkzeuge der Sprachverarbeitung

In den folgenden Skripten sind neue Begriffe, wenn sie das erstemal auftauchen *hervorgehoben* gedruckt. An der entsprechenden Stelle sollte dann auch immer eine kurze Erklärung oder Definition des Begriffs zu finden sein.

1 Einführung in den Computer am Beispiel LINUX

Vorneweg ein paar Begriffsdefinitionen:

COMPUTER
= Maschine, die Zahlen (Symbole) manipulieren kann. Computer = Hardware und Software (+ Netz)

HARDWARE
= Processor (CPU), Speichermedien (Diskettenlaufwerk, Platte, CDROM, Bandlaufwerk,...), Motherboard, Einsteckkarten (Audio, Graphik, Netzwerk, etc), I/O-Peripherie (Monitor, Maus, Tastatur)

SOFTWARE
=OS (Operating System = Betriebssystem) + Programme

Das Betriebssystem bestimmt weitgehend, wie ein Computer zu benutzen ist. Auf derselben Hardware können mehrere OS laufen (allerdings normalerweise nicht gleichzeitig). Wir verwenden hier in diesem Kurs LINUX, weil

- es das am meisten verwendete OS am Institut ist
- es lizenzfrei ist, d.h. jeder kann es auf seinem eigenen Rechner installieren
- es ein echtes UNIX OS ist und daher sehr mächtig in der automatischen Verarbeitung von Daten

Der Benutzer kommt mit dem OS nur sehr wenig in Kontakt; das ideale OS läuft sozusagen 'im Hintergrund' und erledigt alles, ohne dass der Benutzer es explizit merkt. Der Benutzer kommuniziert mit dem OS über eine sogenannte *Shell* (Shell deshalb, weil dieses Programm das eigentliche OS umschließt wie eine Muschel, so dass man nur die Schale zu Gesicht bekommt). Die Shell ist eigentlich nur ein Programm wie alle anderen, aber dieses Programm ist speziell dazu geschrieben, damit es der Benutzer leicht hat, mit dem Computer zu arbeiten. Es stellt die Schnittstelle zwischen der Maschine und dem Menschen dar, d.h. es übersetzt die Befehle des bedienenden Menschen in Maschinen-Kommandos und stellt die Ergebnisse einer Rechenoperation so dar, dass sie für uns verständlich sind.

Um eine Shell zu starten und mit dem Computer zu arbeiten, muss sich der Benutzer *Anmelden* (*Einloggen*). Dazu benötigt er ein *Kürzel* (oder *user name*, *account name*) und ein zugehöriges *Passwort*. Das Passwort muss unbedingt geheim bleiben, um Missbrauch zu verhindern. Es sollte mindestens 9 Zeichen lang sein, kein Wort oder Name sein (auch nicht rückwärts!) und mindestens zwei Zahlen oder Sonderzeichen (Satzzeichen) enthalten. Ein sehr gutes Passwort ist z.B. 'AbHzdW.'. Das ist ein Akronym des Satzes 'Am blauen Himmel ziehen die Wolken.'

- User Name (ekurs2) eingeben und RETURN-TASTE drücken
- Passwort (wird vom Kursleiter mitgeteilt) eingeben und RETURN-Taste drücken (In Zukunft lasse ich das 'RETURN-Taste drücken' weg und sage nur noch eingeben)

Eine sogenannte *Oberfläche* (Window-Manager) erscheint. Diese wurde von der Shell gestartet, weil diese zu recht davon ausging, dass wir sie wollen (das muss aber nicht so sein: In einfachsten Falle zeigt sich die Shell nur mit der Zeile `'cip1 % '`, was bedeutet, dass man ihr nun Befehle geben kann; die Maus wäre in diesem Falle sinnlos). Die Oberfläche hat viele Eigenschaften, auf die wir hier nicht eingehen; für uns ist am wichtigsten, dass sie zwei Fenster (Terminal-Windows) öffnet, in denen ebenfalls die Shell (die sogenannte Bash-Shell oder kurz bash) läuft. (Wenn sich kein Fenster öffnet, klicken Sie auf das Symbol mit der Muschel vor dem Bildschirm (Terminal-Programm).)

Wir erkennen Shell-Fenster am *Prompt* auf der linken Seite: z.B. `'cip5 % '`; oft wird jedoch auch der aktuelle Benutzer und der Rechner, an dem man gerade arbeitet, angezeigt: `'ekurs2@cip5:> '`

Hinter dem Prompt ist eine Schreibmarke (cursor), mit der man nun Befehle eingeben kann. Das ganze nennt man die *Kommandozeile*.

Die Shell wartet nun auf Befehle. Wir werden die allerwichtigsten Befehle jetzt ausprobieren:

1.1 Wo bin ich?

```
cip1 % pwd
/homes/ekurs
```

Der Befehl pwd (print working dir) gibt das *aktuelle Directory* (das aktuelle Verzeichnis) auf den Bildschirm aus. Das ist die Position im *Dateibaum*, die man gerade einnimmt; diese kann sich verändert, wenn man sich im Dateibaum 'bewegt'.

Die Shell hat beim Einloggen dafür gesorgt, dass wir in unserem *Home-Directory* landen. Das ist der private Datenbereich, der nur dem Benutzer gehört. Directories können als *absoluter Pfad* beginnend mit der *Root* (Wurzel) '/' angegeben werden, oder als *relative Pfade* beginnend im aktuellen Directory. Der Punkt ('.') ist eine Abkürzung für das aktuelle Directory; Punkt Punkt ('..') ist das übergeordnete Directory (Vater-Directory). In Zukunft schreiben wir kurz 'Dir' für 'Directory'.

Beispiele:

```
/homes/ekurs2                # absoluter Pfad (Home-Dir)
cip1/subdir                  # relativer Pfad, 2 Ebenen tiefer
/homes/ekurs2/cip1/subdir    # absoluter Pfad, genau der gleiche wie eben
..                           # relativer Pfad, eins nach oben = /homes
../..                        # relativer Pfad, zwei nach oben = /
./cip1                       # relativer Pfad, dass gleiche wie 'cip1'
```

1.2 Was ist hier gespeichert?

```
cip1 % ls
```

Der Befehl ls (list) gibt den *Inhalt* des aktuellen Dir oder eines angegebenen Dirs aus oder sagt uns etwas über eine angegebene Datei.

```
cip1 % ls                       # Inhalt des aktuellen Dir
cip1 % ls SS10/cip1/subdir      # Inhalt des Subdirectory 'subdir'
cip1 % ls SS10/cip1/testfile    # listet das File 'testfile'
cip1 % ls /                     # listet den Inhalt des Root-Dirs
```

ls hat viele *Optionen*. Optionen sind optionale *Argumente*, die man hinter dem Befehl in die Kommandozeile schreibt. Gewöhnlich beginnen Optionen mit einem '-' und bestehen aus einem einzelnen Buchstaben; was auch vorkommt, ist die Form '–option=value' oder auch 'option=value'.

Z.B.:

```
cip1 % ls -a
cip1 % ls --color=always
```

Die Option '-a' bewirkt, dass ls alle, auch verborgene Dateien ausgibt. Verborgene Dateien werden normalerweise nicht angezeigt und beginnen alle mit einen '.'. Meistens sind das Konfigurationsdateien für die Shell oder Programme.

Dateinamen sind *case sensitive*, d.h. 'Testfile' ist nicht das gleiche wie 'testfile' oder gar 'testFile'.

Frage: Welche Files sind in unserem Home-Dir 'verborgen'?

Anderes Beispiel für eine Option:

```
cip1 % ls -l SS10
```

gibt mehr Information als nur den Namen pro File aus:

```
RECHTE         USER     GROUP        SIZE MOD.DATE     NAME
total 0
drwxr-xr-x   2 ekurs    users          59 Oct 18 11:23 cip1/
drwxr-xr-x   2 ekurs    users          59 Oct 18 11:23 cip2/
drwxr-xr-x   2 ekurs    users          59 Oct 18 11:23 cip3/
drwxr-xr-x   2 ekurs    users          59 Oct 18 11:23 cip4/
drwxr-xr-x   2 ekurs    users          59 Oct 18 11:23 cip5/
drwxr-xr-x   2 ekurs    users          59 Oct 18 11:23 cip6/

cip1 % ls -l SS10/cip1/test
-rw-r--r--   1 ekurs    users         264 Oct 18 11:23 cip1/test
```

In der ersten Spalte sind die *Zugriffsrechte* für die Datei kodiert. In den Spalten USER und GROUP steht, wem diese Datei gehört und zu welcher Gruppe dieser gehört. In SIZE steht die Dateigröße (in Bytes). MOD.DATE beschreibt Datum und Uhrzeit der letzten Änderung und die letzte Spalte NAME schließlich beschreibt den Namen der Datei.

1.3 Ich gehe nach...

Bewegen im Dateisystem (das aktuelle Dir wird verlegt):

```
cip1 % cd /tmp
```

Der Befehl cd (change directory) ändert das aktuelle Directory, in diesem Fall nach '/tmp'. Das Directory '/tmp' ist auf jedem UNIX-Rechner vorhanden. Dort darf jeder Daten zeitweise ablegen (temporär), aber die Daten sollen dort nicht permanent gespeichert werden (manche OS löschen diesen Bereich beim Abschalten des Rechners). Daher dort niemals wichtige Daten lagern!

Gibt man 'cd' ohne *Argumente* ein, wird man automatisch in das eigene Home-Dir zurückgebracht.

Ein Argument ist keine Option, sondern ein Wert, der dem Befehl übergeben wird. 'cd' hat z.B. gar keine Optionen und genau ein Argument. 'ls' hat Optionen und Argumente (Dateien oder Dirs). Diese Unterscheidung ist wichtig:

```
cip1 %   ls -l    test
            ^^    ^^^^
            |     |ARGUMENT
            |
            |OPTION
```

Grundsätzlich sollte jede Gruppe nur in ihrem *Gruppen-Directory* arbeiten; also z.B. die Gruppe am Rechner 'cip1' im Sub-Directory 'cip1'. Zu Beginn jeder Stunde sollte daher die Gruppe nach dem Einloggen folgenden Befehl ausführen:

```
cip1 % cd SS10/cip1
```

Directory: /homes/ekurs/SS10/cip1

1.4 Was ist in dieser Datei?

```
cip1 % cd
cip1 % cd SS10/cip1
cip1 % cat test
```

Der Befehl cat (concatenate and type) gibt den Inhalt einer Datei oder mehrerer hintereinander auf den Bildschirm, genauer gesagt auf *standard output*. Auf diese Weise kann man sich z.B. Text-Dateien anschauen. Damit die Daten nicht so schnell vorbeirauschen, gibt es den Befehl 'less'.

```
cip1 % less test.txt
```

less hat sehr viele Befehle zu Anzeigen von Dateien. Am wichtigsten für uns ist die Tatsache, dass wir mit den *Cursortasten* hinauf- und hinuterfahren können und mit 'q' wieder herauskommen. Man kann nach Begriffen suchen, indem man '/' und ein Suchmuster eingibt:

```
/Dies
```

Dann springt less an die erste gefundene Stelle, die auf das Muster passt. Mit der Taste 'n' springt man zum nächsten Muster; mit 'N' zum vorherigen Muster.

less kann Dateien nur anzeigen, d.h. man kann nichts kaputtmachen. Zum Kaputtmachen verwendet man einen *Editor*.

1.5 Wie ändere ich das? (wie mache ich etwas kaputt?)

```
cip1 % pico testfile
```

Mit dem Editor pico können wir Zeichen löschen, ändern und neue hinzufügen. Die wichtigsten Befehle für Pico sind:

```
Strg-X : verlassen und abspeichern
Strg-W : Suchen nach Mustern
Strg-O : Sichern
Strg-G : Online Hilfe
```

pico ist praktisch, weil er in einem einfachen Terminal-Fenster funktioniert und sehr einfach zu bedienen ist. Ein Nachteil von pico ist, dass er beim Schreiben automatisch zu lange Zeilen umbricht, auch wenn man das nicht will.

Bessere (aber auch schwierigere) Terminal-Editioren sind *vi* und *emacs*. Ein auch sehr einfacher, guter, aber graphischer Editor ist *kwrite* ('graphisch' heißt, er funktioniert nicht in einem Terminal-Fenster, sondern man braucht dazu eine graphische Oberfläche, wie z.B. das X-System auf Linux oder das Windows-System auf PCs).

1.6 Wie drucke ich ein File?

Text-Files und Postscript-Files (ein genormtes Druck-Format) können direkt mit dem Befehl 'lphp' auf den Netzdrucker geschickt werden.

```
cip1 % lphp test
```

(Der Befehl lphp3 druckt auf dem Netzdrucker im Projektzimmer im 4. Stock; lphp2 auf dem Drucker im 1. Stock.)

1.7 Übungen

1. Gehe in dein Gruppen-Dir und erzeuge ein Text-File namens 'Uebung1' mit dem Inhalt: "Sehr geehrter Herr Kursleiter,"
2. Gib den Inhalt des Files 'Uebung1' auf das Display.
3. Berechne die Anzahl der Zeichen im File 'Uebung1'. (Befehl 'wc')
4. Editiere ein neues File und schreibe folgenden Text: ïch möchte Ihnen versichern, dass dieser Kurs der beste ist, den ich jemals besucht habe."Dann speichere das Ergebniss unter einem neuen Filenamen 'Uebung2' ab.
5. Hänge die Files 'Uebung1' und 'Uebung2' hintereinander und gib das Ergebnis zuerst zur Kontrolle auf den Schirm.
6. Leite das soeben erzielte Ergebnis in ein neues File 'Brief' um.

   ```
   (cip1 % cat Uebung1 Uebung2 > Brief)
   ```

7. Ersetze alle 'i' im File 'Brief' durch das Zeichen 'y'.
8. Drucke die Datei 'Brief' aus.

2 Weitere UNIX-Befehle

Weitere nützliche Befehle zum selber ausprobieren:

```
cip1 % mkdir newdir       (erzeuge neues Verzeichnis)
cip1 % cp file1 file2         (kopiere file1 nach file2)
cip1 % mv file1 file2         (bewegen/umbennen)
cip1 % rm file1               (löschen)
cip1 % du dir                 (Dateigrößen ausgeben)
cip1 % who           (wer ist noch hier?)
cip1 % whoami                 (wer bin ich?)
cip1 % echo 'Hello, world!'  (drucke auf Standard-Output)
cip1 % locate xemacs   (finde mir...)
cip1 % cat file1 | tr 'x' 'y' > file2  (ersetze alle 'x' durch 'y')
cip1 % cat file1 | tr -d 'x'  > file2  (lösche alle 'x')
cip1 % sort file   (sortiere zeilenweise)
cip1 % uniq file              (sortiere doppelte Zeilen aus)
```

2.1 Wie helfe ich mir selbst?

Am einfachsten natürlich ist, ein Buch zu lesen (z.B. 'UNIX' von H. Schröpfer, dtv, oder viele andere Linux-Bücher). Ansonsten gibt es zu jedem Befehl eine *man- oder info-Seite*. Z.B. bekommt man eine detailierte Beschreibung des Befehls 'ls' mit

```
cip1 % man ls
```

Auch die Shell (bash) selber hat eine man page (eine sehr lange und nicht ganz leicht zu verstehende):

```
cip1 % man bash
```

Alternativ gibt es auch das Kommando 'info'

```
% info paste
```

2.2 Wie erleichtert mir die Shell die Arbeit?

Joker

Zum Beipiel mit *Jokern*. Ein Jokerzeichen ist eine Art 'Platzhalter' in Dateinamen. Der wichtigste ist '*', er steht für alles mögliche. Daher ist der Befehl

```
cip1 % ls *
```

so ähnlich wie

```
cip1 % ls
```

Was ist der Unterschied? Warum?

Die Shell *expandiert* das Jokerzeichen '*' zu allen Namen im aktuellen Directory, die darauf passen (und es passen alle, außer denen, die mit einem '.' beginnen) und gibt diese dann erst an das Kommando 'ls' weiter.

Man kann den Joker aber auch in Dateipfaden verwenden, z.B.

```
cip1 % ls subdir/file*
```

expandiert zu

```
cip1 % ls subdir/file1 subdir/file2 subdir/file3 subdir/file10
```

Ein anderes Jokerzeichen ist '?'; es steht für genau ein Zeichen.

```
cip1 % ls subdir/file?
```

hat fast den gleichen Effekt wie der obige Befehl. (Was ist der Unterschied?)

Dann gibt es als Drittes noch die eckigen Klammern '[ ]'. Die stehen auch genau für ein Zeichen, aber nicht ein beliebiges, sondern aus einer definierten Menge.

```
cip1 % ls subdir/file[12]
```

expandiert zu

```
cip1 % ls subdir/file1 subdir/file2
```

d.h. nur die beiden ersten Files werden ausgegeben. Innerhalb der Klammern können beliebig viele Zeichen stehen oder auch Bereiche, z.B.

```
cip1 % ls subdir/file[1-3]
```

expandiert zu

```
cip1 % ls subdir/file1 subdir/file2 subdir/file3
```

Ist das erste Zeichen innerhalb der '[ ]' ein '^', dann wird die Menge der nachfolgenden Zeichen negiert, z.B.

```
cip1 % ls subdir/file[^12]
```

expandiert zu

```
cip1 % ls subdir/file3
```

Dateinamen-Vervollständigung

Die Shell versucht aus Teilstücken auf die ganzen File-Namen zu schliessen. Ausgelöst wird das durch die Taste 'TAB'. Wenn z.B. im aktuellen Directory das File 'Haushaltsaussschuss' gespeichert ist, genügt es folgendes einzugeben:

```
cip1 % cd SS10/cip1
cip1 % ls Ha<tab>
```

Die Shell vervollständigt die Zeile automatisch zu

```
cip1 % ls Haushaltsaussschuss
```

Voraussetzung ist, dass es eine eindeutige Ergänzung ist, d.h. dass nicht noch ein File vorhanden ist, das mit 'Ha' beginnt. Ist eine Vervollständigung mehrdeutig, kann man sich durch erneutes Drücken der Taste 'TAB' alle möglichen Alternativen anzeigen lassen:

```
cip1 % ls H<tab><tab>
Haushaltsausschuss  Hotel
```

Genauso wie Dateinamen, werden auch Kommandos vervollständigt, wenn der eingegebene Teilstring eindeutig ist. Z.B.

```
cip1 % mkd<tab>
```

wird vervollständigt zu

```
cip1 % mkdir
```

2.3 Was bedeutet 'standard out'?

Die drei Standard-Kanäle

Außer den direkten Lesen und Schreiben von Dateien verwendet Linux drei 'Kanäle', um Information in ein Kommando hinein- und herauszuleiten:

- Standard Output : der Kanal, in den ein Kommando seine Ausgabedaten schreibt (wenn es nicht in eine Datei schreibt),
- Standard Input : der Kanal, von dem ein Kommando seine Eingabedaten schreibt (wenn es nicht aus einer Datei liest), und
- Standard Error : der Kanal, wohin ein Kommando Fehlermeldungen schreibt.

(Für Interessierte:
Formal öffnet ein Programm eine Datei im Dateibaum, namens `/dev/stdout, /dev/stdin` oder `/dev/stderr` für seine Lese- oder Schreiboperationen. Linux interpretiert das dann so, als ob nicht in eine Datei sondern in einen der drei Standardkanäle geschrieben/gelesen wird.)

Normalerweise ist der Kanal standard input immer mit der Tastatur verbunden, und die Kanäle standard output und standard error mit dem Display.

Deswegen sehen wir eine Ausgabe, wenn wir 'ls' eingeben, und keine Ausgabe, wenn wir 'ls > liste' eingeben; in letzterem Falle wird der Kanal standard output, der normalerweise mit dem Display verbunden ist, in eine Datei 'liste' umgelenkt.

Pipes und Umlenkung des standard output

Pipes sind aneinander gereihte Kommandos, die wie in einer Eimerkette Daten weiterreichen (pipeline). Das Verknüpfungszeichen ist '|' (auf PC-Tastaturen: Alt-Gr + '<>' bzw. rechte Alt-Taste und '<>').

```
cip1 % ls -l
cip1 % ls -l | cut -c 1-5
```

Das Pipe-Symbol verknüpft den standard output des vorangegangenen Kommandos mit dem standard input des nachfolgenden Kommandos.

Der 'ls' Befehl gibt seinen Output nicht mehr auf das Display (standard output) sondern 'piped' ihn in das nächste Kommando 'cut'. cut liest - da es kein File als Argument bekommen hat - von standard input, d.h. einem Kanal, der normalerweise unsere Tastatur ist, und verarbeitet das, was dort kommt, entsprechend seiner Optionen. Hier soll es die Spalten 1 - 5 herauschneiden und weitergeben. Da nach den 'cut' Befehl nicht wieder ein '|' Pipe-Zeichen steht, gibt 'cut' sein Ergebniss auf den Bildschirm (standard output). Alternativ könnten wir auch eingeben:

```
cip1 % ls -l | cut -c 1-5 | lphp
```

Dann wird das Ergebnis sofort auf den Netzdrucker geschickt, weil auch das Kommando lphp von standard input liest, wenn es keine Datei als Argument bekommt.

Auf diese Weise können beliebig lange Verarbeitungs-Pipelines mit einem einzigen Kommando erzeugt werden. Beispiele folgen später noch viele. Will man das Ergebnis einer Pipe (oder auch nur eines Befehls) in ein File speichern, so hängt man das *Umlenkungssymbol* > an die Pipe und schreibt den Dateinamen dahinter:

```
cip1 % ls -l | cut -c 1-5 > ergebnis-file
```

Existiert die Datei bereits, so wird sie überschrieben. Will man das Ergebnis nur an die Datei anhängen, verwendet man '>>' statt '>':

```
cip1 % ls -l | cut -c 1-5 >> ergebnis-file
```

Ein paar Beispiele

Alle Files auflisten:

```
cip1 % ls *
```

Nur Files mit 'H' am Anfang auflisten:

```
cip1 % ls H*
```

Nur Files mit genau 5 Buchstaben im Namen auflisten:

```
cip1 % ls ?????
```

Nur Files mit einem 'U' oder einem 'H' am Anfang auflisten:

```
cip1 % ls [HU]*
```

Text auf Bildschirm ausgeben:

```
cip1 % echo 'Hello, world!'
```

Text in ein File schreiben:

```
cip1 % echo "Hier kommt der Text." > newfile
```

Inhalt eines Files auf Bildschirm ausgeben:

```
cip1 % cat newfile
```

Inhalt eines Files in ein anderes File schreiben:

```
cip1 % cat newfile > newfile2
```

Inhalt eines Files an ein anderes Files anhängen:

```
cip1 % cat newfile >> newfile2
```

Von Tastatur (standard input) in eine Datei schreiben:

```
cip1 % cat > newfile2
Dies ist ein Text.
^D                                  (Strg-D)
cip1 % cat newfile2
Dies ist ein Text.
```

Umlenkung des standard input

Genau wie der standard output mit dem Umlenkungssymbol > in eine Datei gerichtet wird, kann auch der Kanal standard input an eine Datei gehängt werden:

```
cip1 % cat < quelldatei > zieldatei
```

In diesem Beispiel liest der Befehl cat nicht aus einer Datei (weil kein Argument gegeben ist), aber auch nicht von Tastatur, sondern aus der Datei quelldatei. Er schreibt dann alles, was er liest nach standard output, hier also in die Datei zieldatei. Das ganze entspricht einem Kopiervorgang, dem man auch mit dem Befehl cp hätte ausführen können:

```
cip1 % cp quelldatei zieldatei
```

2.4 Skripte und Variablen

Die Shell kann nicht nur einfache Befehle von der Kommandozeile ausführen, sondern auch ganze Programme, sog. *Skripte* (scripts) ausführen. Skripte stehen normalerweise in Text-Dateien, die dann von der Shell gelesen werden und die darin enthaltenen Befehle der Reihe nach ausgeführt werden (wie wenn man sie auf der Tastatur eingegeben hätte). Man kann aber auch auf der Kommandozeile kleine Skripte eingeben.

Skripte sind mehr als nur hintereinander ausgeführte Linux-Kommandos. Mit *Kontrollkommandos* können bestimmte Kommandos nur unter bestimmten Bedingungen ausgeführt werden (*if-Verzweigung*) oder Kommando-Serien mehrfach durchlaufen werden (*Schleife*).

Dazu ist es sinnvoll (und notwendig), dass wir Daten wie Dateinamen, Zahlen etc. nicht immer explizit in das Skript schreiben, sondern dafür *Variablen* verwenden. Eine Variable ist einfach ein Platzhalter mit einem (fast) frei wählbarem Namen, der einen bestimmten Inhalt haben kann. In der Shell-Skriptsprache ist dies immer eine Kette von Zeichen (ein String). Je nach Kontext wird dieser string als Wort oder Zahl interpretiert.

Variablen werden mit dem Befehl '<name>=<wert>' gesetzt. Dabei wird vor dem Variablenname *kein* '$' geschrieben. Später aber muss man immer vor dem Namen ein '$' schreiben, damit wirklich der Inhalt der Variable und nicht ihr Name verwendet wird. Wichtig ist auch, vor und nach dem Gleichheitszeichen ('=') keine Leerzeichen zu verwenden!

Wir wollen zunächst eine Variable namens *input* erzeugen und anschließend den Inhalt dieser Variable in einer if-Verzeigung prüfen.

Beispiel 1:
Wir wollen ein Skript schreiben, das 'File' ausgibt, wenn der als Argument übergebene Name eine normale Datei ist oder 'Dir' ausgibt, wenn es ein Verzeichnis ist.
(Die Textteile nach den Kommmentarzeichn '#' müssen Sie *nicht* eingeben!)

```
cip1 % input=subdir                    # schreibe den Wert 'subdir' in die
                                       # Variable namens 'input'
cip1 % echo $input                     # Variable ausgeben; beachte das '$'
subdir
cip1 % echo input                      # '$' fehlt -> input ist nur ein String
input
cip1 % if [ ! -e $input ]; then echo "$input existiert nicht"
> elif [ -f $input ]; then echo "File"
> else echo "Dir"
> fi
```

oder in einer Datei (Skript), die wie ein Programm ausgeführt werden kann:

```
cip1 % pico filecheck
```

Folgenden Text eingeben und Datei abspeichern:

```
#!/bin/bash
input=$1
if [ ! -e $input ]; then echo "$input existiert nicht"
elif [ -f $input ]; then echo "File"
else echo "Dir"
fi
```

Die neue Datei `filecheck` ausführbar machen:

```
cip1 % chmod 755 filecheck
```

und Skript starten:

```
cip1 % ./filecheck subdir
Dir
cip1 % ./filecheck subdir/file1
File
cip1 % ./filecheck XXXXX
XXXXX existiert nicht
```

Die if-Verzweigung hat einen Teil, der ausgeführt wird, wenn die erste Bedingung '! -e $input' (= $input not exist) erfüllt ist, und einen Teil, der ausgeführt wird, wenn die erste Bedingung *nicht* erfüllt ist aber dafür die zweite Bedingung '-f $input' (= $input is file) nach dem 'elif' (= 'else if') Befehl. Und wenn diese zweite Bedingung auch nicht erfüllt ist, dann wird der dritte Teil nach dem Befehl 'else' ausgeführt.

Man könnte das ganze Skript in Worten so formulieren:

> Lies einen Namen als erstes Argument von der Kommandozeile ($1) und schreibe ihn in die Variable 'input'.
> Wenn das Objekt, dessen Name in der Variable 'input' steht, nicht existiert, gib den Text '$input existiert nicht' aus.
> Anderfalls prüfe, ob das Objekt ein normales File ist. Wenn ja, gib den Text 'File' aus. Wenn nein, gib den Text 'Dir' aus.

Variablen, deren Name nur aus einer Nummer besteht sind, spezielle Variablen, die automatisch beim Starten des Skripts mit den Argumenten der Kommandozeile belegt werden:

```
variable=Donnerstag
cip1 % testscript arg1 7897 "Florian Schiel" $variable
```

Innerhalb des Skripts 'testscript' existieren nach diesem Aufruf folgende Variablen:

```
$1 = arg1
$2 = 7897
$3 = Florian Schiel
$4 = Donnerstag
```

Ein anderes nützliches Kontrollkommando ist 'for'. 'for' beginnt eine *Schleife* über eine Liste von Argumenten, die nacheinander einer *Variablen* zugewiesen werden. Auf diese Weise kann man sich sehr viel Tippen ersparen.

Beispiel 2:
Wir wollen wissen, wieviele Wörter insgesamt in den Text-Files im Subdirectory subdir stehen:
(Die Textteile nach den Kommmentarzeichn '#' müssen Sie *nicht* eingeben!)

```
cip1 % cd SS10/cip1
cip1 % count=0          # die Variable 'count' wird auf Null gesetzt
cip1 % echo $count      # Zur Kontrolle ausgeben
0
cip1 % for file in subdir/file* ; do # Beginn einer Schleife, in der nach
                                     # einander der Variablen 'file' die
                                     # Namen der files in subdir
                                     # zugewiesen
                                     # werden

> echo File = $file                  # Zur Kontrolle ausgeben

> anz=‘cat $file | wc -w‘            # Das File in der Variablen 'file' wird
                                     # in das Kommando 'wc -w' ge-piped.
                                     # Das Ergebnis von 'wc -w'
                                     # wird der Variable 'anz' zugewiesen.
                                     # Beachte die schräggestellten
                                     # Hoch-Kommata!
> count=$(($count + $anz))           # Der Wert wird auf die Variable count
                                     # aufaddiert. $(( ... )) klammert
                                     # arithmetische Berechnungen

> done                               # Ende der for-Schleife

cip1 % echo $count                   # Ausgabe des Ergebnisses
...
```

Ein paar Erklärungen zu diesem Beispiel:

- mit dem Prompt '>' zeigt uns die Shell, dass wir uns jetzt innerhalb einer Schleife befinden, dass also alle Befehle, die wir hier eintippen ev. mehrfach ausgeführt werden. Erst mit der Eingabe '> done' beendet man die Schleife und gibt gleichzeitig den Befehl, die Schleife jetzt durchlaufen zu lassen.
- der Befehl 'echo' ist sehr nützlich, wenn man wissen will, welche Werte Variablen gerade haben. 'echo' nimmt alles, was hinter dem Befehl steht, macht alles, was die Shell machen würde, bevor sie die Kommandozeile ausführt und gibt das Ganze auf Standard-Output (also den Bildschirm).
- alles, was in nach links gekippten Hochkommas (‘...‘) eingeschlossen ist, wird von der Shell wie ein Befehl ausgeführt und das Ergebnis, d.h. das was der Befehl nach stdout schreibt, wird an die Stelle der Hochkommas in der Kommandozeile kopiert. In unserem Beispiel wird also zunächst die Pipe 'cat $file | wc -w' ausgeführt: das File, dessen Name in der Variable 'file' gespeichert ist, wird nach 'wc -w' ge-piped, welches die Anzahl der Wörter zählt und auf stdout ausgibt. Diese Ausgabe (eine Zahl) wird von der Shell an die Stelle der schrägen

Hochkommas kopiert, und dann erst der Rest der Kommandozeile ausgeführt. In unserem Falle wird das Ergebnis der Variable 'anz' zugewiesen.

- der '$((...))'-Klammerung dient zum Rechnen mit Variablen (nur Bash-Shell!). Die allgemeine Form dieses Befehls ist:

  ```
  variable=$((formel))
  ```

  Die Variable, die das Ergebnis der Rechnung enthalten soll, kommt zuerst und wird wie bei der Zuweisung sonst auch OHNE das '$' geschrieben. Danach kommt die Formel, deren Ergebnis der Variable zugewiesen werden soll.

  Zum Beispiel:

  ```
  ergeb=$(( 10 * 80 ))          # die Variable 'ergeb' wird zu 800 gesetzt
  ergeb=$(( $var1 + $var2 ))    # die Variable 'ergeb' enthält die Summe
                                # der Variablen 'var1' und 'var2'
  ergeb=$(( $ergeb + 1 ))       # die Variable 'ergeb' wird um eins erhöht
  ```

2.5 Übungen

1. Sie haben vergessen, wo das File 'bash.bashrc' im Dateibaum gespeichert ist. Wie können Sie es wiederfinden? Wo ist es gespeichert?

2. Aus der letzten Uebungsstunde sind noch einige Files in ihren Gruppen- Directory übriggeblieben. Da Sie ein ordentlicher Mensch sind, wollen Sie die Dateien aufräumen. Erzeugen Sie sich ein neues Directory 'Uebung_1' und bewegen Sie alle Files dorthin.

3. Im Sub-Directory 'subdir' befinden sich vier Dateien und ein Subdirectory. Schauen Sie sich die Dateien an. Entfernen Sie mit einem geeigneten Filter die hässlichen '>' an Anfang jeder Zeile (Tip: man tr). Hängen Sie alle drei Files zu einem File 'gesamt' zusammen und drucken Sie dieses aus.

4. Geben Sie eine Pipe ein, die in einem Verzeichnis ausgeführt, von allen Dateien nur die Größe und den Namen in einer Tabelle ausgibt. (Tip: man cut)

5. Schreiben Sie ein Skript namens 'char_count' analog zu obigem Beispiel, das alle Dateien im aktuellen Directory öffnet, die Zeichen (nicht die Wörter!) aufaddiert und ausgibt. (Tips: - in einem Skript muss in der ersten Zeile `'#!/bin/bash'` stehen - nach Fertigstellung den Befehl 'chmod 755 char_count' eingeben - Skript starten mit './char_count')

6. Erweitern Sie Ihr Skript, so dass es nur die Dateien durchsucht, die keine Directories sind.

   Tipp: Verwenden Sie Ihr Skript 'filecheck'. Um den Inhalt einer Variablen `var` mit einer Zeichenkette (string) zu vergleichen, verwenden Sie

   ```
   if [ $var == "File" ]; then
   ...
   fi
   ```

7. Erweitern Sie Ihr Skript 'char_count', so dass es nicht im aktuellen Dir Dateien durchsucht, sondern in dem Dir, das als Argument ($1) auf der Kommandozeile übergeben wird.

8. Erweitern Sie Ihr Skript 'char_count', so dass es trotzdem im aktuellen Dir Dateien durchsucht, wenn das Argument fehlt.