# UNIX–Kurs
# Hochschulrechenzentrum TH–Darmstadt

Stefan Grüber[1]
Manfred Lang[2]
Rainer Radimersky[3]
Klaus Kaltwasser[4]
Roger Kehr[5]

25. Februar 1993

[1]email: grueber@hrz.th-darmstadt.de tel: 16-37 27
[2]email: mln@hrz... tel: 16-55 65
[3]email: radi@hrz... tel: 16-56 08
[4]email: kalth2o@hrz...
[5]email: kehr@hrz...

# Inhaltsverzeichnis


**1 Einführung in UNIX** **1**
- 1.1 Zum Script . . . 1
- 1.2 Zum Aufbau des Kurses . . . 1
- 1.3 Kurseinteilung . . . 1
- 1.4 Was der Kurs nicht bietet ! . . . 2

**2 Einführung** **2**
- 2.1 Was ist UNIX ? . . . 2
- 2.2 Historie . . . 3
- 2.3 Charakteristika von UNIX . . . 3
- 2.4 Grundsätzlicher Aufbau von UNIX . . . 3
  - 2.4.1 Kernel (Betriebssystemkern) . . . 4
  - 2.4.2 Prozeßverwaltung . . . 5
- 2.5 Handhabung eines UNIX-Systems . . . 6

**3 Die Anmeldung am System (login)** **6**
- 3.1 Der Benutzer . . . 6
  - 3.1.1 Setzen des Passworts . . . 7
  - 3.1.2 Terminalbenutzung . . . 8
- 3.2 Der aktuelle Status . . . 8
  - 3.2.1 Was macht der Rechner ? . . . 8
  - 3.2.2 Prozeßkommandos . . . 8

**4 Filesysteme** **10**
- 4.1 Basiswissen über Filesysteme . . . 10
- 4.2 Verzeichnisse . . . 10
- 4.3 Grundregeln . . . 11
  - 4.3.1 Regeln . . . 11
  - 4.3.2 Allgemeine Empfehlungen . . . 11
  - 4.3.3 Common Sense . . . 12
- 4.4 Etwas mehr Theorie . . . 13
  - 4.4.1 Implementation . . . 13
  - 4.4.2 Harddisks (Platten) . . . 13
  - 4.4.3 Verzeichniseintrag für Dateien . . . 13
  - 4.4.4 Details für Verzeichnisse . . . 13
- 4.5 Zugriffsrechte . . . 14
- 4.6 Erste Befehle . . . 14
- 4.7 Dateimanipulationsbefehle . . . 15
  - 4.7.1 Befehle . . . 15

**5 Einführung in die Shell** **18**
- 5.1 Was ist die Shell ? . . . 18
- 5.2 Fähigkeiten der Shell ? . . . 18
- 5.3 Was macht die Shell ? . . . 19
- 5.4 Shell-Geschichte . . . 19

**6 Die Shell auf der Kommandoebene** **19**

**7 Das Variablensystem der Shell** **19**
7.1 Variablen im Environment . . . 19
7.2 UNIX-System-Variablen . . . 20
7.3 Besondere Variablen . . . 21

**8 Kommandozeilenverarbeitung** **21**
8.1 Wie wird eine Zeile bearbeitet ? . . . 21
8.2 Was macht die Shell mit der eingegebenen Zeile ? . . . 21
8.2.1 Eingabe lesen und parsen . . . 21
8.2.2 Verbose–Tracing . . . 22
8.2.3 Parameter-Ersetzung . . . 22
8.2.4 Kommandoersetzung . . . 22
8.2.5 Ein-/Ausgabe-Umlenkung . . . 23
8.2.6 Pipes . . . 23
8.2.7 IFS-Trennung . . . 23
8.2.8 Filenamen–Expansion . . . 24
8.2.9 Beispiele zur Filenamen–Expansion . . . 24
8.2.10 Ausführungs-Trace . . . 24
8.2.11 Kommandoausführung . . . 24

**9 Prozeß-Hierarchie** **25**
9.1 Was passiert beim Programmaufruf ? . . . 25

**10 Befehle der Shell** **26**
10.1 Direkte Befehle (Shell-Builtins) . . . 26
10.2 Kontrollstrukturen . . . 27
10.3 Testkommandos . . . 28

**11 Die Startup–Files** **29**
11.1 `/etc/profile` . . . 30
11.2 `$HOME/.profile` . . . 31

**12 Das Quoting** **31**
12.1 Einführung Quoting . . . 31
12.2 Die Quote-Zeichen . . . 32
12.2.1 Beispiele . . . 32
12.3 Die Backquotes . . . 33
12.3.1 Beispiele . . . 33

**13 Reguläre Ausdrücke** **34**
13.1 Reguläre Ausdrücke für ein einzelnes Zeichen . . . 34
13.2 Reguläre Ausdrücke für mehrere Zeichen . . . 36
13.3 Beginn und Ende einer Zeile . . . 37

**14 Der Stream-Editor sed** **38**
14.1 Der Aufruf des sed . . . 38
14.2 Die Anweisungen des sed . . . 38

**15 Der Reportgenerator awk** **43**
15.1 Aufruf von awk . . . 43
15.2 Das awk-Programm (Skript) . . . 43
15.3 awk-Sprachelemente . . . 44
15.3.1 awk-Konstanten . . . 45
15.3.2 awk-Variablen . . . 45
15.3.3 awk-Ausdrücke . . . 45
15.3.4 awk-Aktionen . . . 46
15.3.5 Die Funktionen des awk . . . 47
15.4 Übergabe von Shellparametern . . . 48

**16 Weitere Grundbefehle** **49**
16.1 date - Ausgabe des Systemdatums . . . 49
16.2 find - Suchen von bestimmten Dateien . . . 49
16.3 grep - Suchprogramm . . . 49
16.4 head, tail - Ausgabe von Dateien . . . 50
16.5 ln - Verweis auf eine Datei . . . 50
16.6 lpr - Druckausgabe von Dateien . . . 51
16.7 man - Ausgabe einer Befehlsbeschreibung . . . 51
16.8 pack, unpack - Komprimieren von Dateien . . . 51
16.9 sort - Sortieren von Ausgaben . . . 52
16.10 tar - Sicherungsbefehl . . . 52
16.11 time - Dauer eines Prozeßes ausgeben . . . 53
16.12 wc - Zählen von Zeichen, Zeilen und Wörtern . . . 53
16.13 compress - Komprimierung . . . 53

**17 Programmentwicklung unter UNIX** **55**

**18 Vorgänge beim Programmieren** **55**
18.1 Die Programmierumgebung . . . 56
18.2 Allgemeines zu Compilern unter UNIX . . . 56
18.3 Die Programmiersprache C . . . 57
18.4 C-Compileraufruf (cc) . . . 57
18.4.1 Der Präprozessor cpp . . . 57
18.5 Allgemeines zur C–Programmierung . . . 58
18.5.1 Sprachkurzbeschreibung . . . 58
18.5.2 Kontrollstrukturen in C . . . 58
18.5.3 Speicherklassen . . . 65
18.5.4 Datentypen . . . 67
18.5.5 Grundtypen . . . 67
18.5.6 Vektoren und Zeiger . . . 71
18.5.7 Felder (Arrays) . . . 71
18.5.8 Strukturen . . . 71
18.5.9 Varianten–Records(Union) . . . 73
18.5.10 Typdefinitionen . . . 73
18.5.11 Funktionen . . . 73
18.5.12 Sonstiges . . . 75
18.5.13 Runtime Library . . . 75
18.6 C–Beispielprogramm . . . 76
18.7 FORTRAN–Programmierung . . . 78

18.8 Assembler–Programmierung . . . . . 78
18.9 Grundsätzliches zum Assembler . . . . . 78

**19 Werkzeuge** **78**
19.1 Die Bibliotheksbearbeitung . . . . . 79
19.2 Symboltabellenanzeiger nm . . . . . 79
19.3 Der Linker ld . . . . . 80
19.4 Das make–Kommando . . . . . 80
19.4.1 Besonderheiten von `make` . . . . . 81
19.4.2 Beispiel Makefile . . . . . 83
19.5 Source Level Debugger dbx . . . . . 84

**20 Werkzeuge für die C-Entwicklung** **85**
20.1 C–Programm Checker lint . . . . . 85
20.2 Generierung von Software . . . . . 86
20.3 Generierung von Software . . . . . 86

# 1 Einführung in UNIX

## 1.1 Zum Script

Dieses Script liegt nun in der 2. Fassung vor. Gegenber der Vorläuferfassung haben wir den Kurs aktualisiert und Unzulänglichkeiten entfernt. Insebesondere sind die Kapitel über die Softwareentwicklungsumgebeung und die Einfhrung intensiv modifiziertt worden. Es existiert jetzt auch eine Sonderausgabe ztum Thema X-Windows.

## 1.2 Zum Aufbau des Kurses

„UNIX ist ein hervorragendes Betriebssystem“
- sagen die Eingeweihten
„UNIX ist chaotisch“
- sagen die Anfänger
Dies ist kein Widerspruch.
Die Aufgabe der UNIX-Einführung soll es sein, den chaotischen Eindruck durch Strukturierung zu entwirren. Hierzu einige Kernsätze, die aus der Sprachdefinition für C++ entnommen wurden und auf das Thema Betriebssysteme angepasst wurden.

1. Alle Funktionen sollen elegant und sauber in das System integriert sein.
2. Alle Funktionen sollen kombinierbar sein.
3. Es soll so wenig wie möglich Sonderfunktionen geben.
4. Wie eine Funktion ihre Aufgabe erledigt, ist für ihre Benutzung nicht relevant.
5. Ein Benutzer benötigt nur soviel Wissen über eine Funktion, wie er für die Lösung seiner Aufgabenstellung benötigt.

Die Punkte 1–3 sind Vorgaben, die das UNIX unserer Meinung nach erfüllt; die Punkte 4+5 sind Elemente, die wir besonders in unserem Kursaufbau berücksichtigt haben. Der Kurs will neben der Theorie auch praktische Übungen anbieten. Dies hat dazu geführt, daß die Einführung nicht dem klassischen Aufbau vieler Lehrbücher folgt, sondern sich in seinem Zeitzuschnitt den Übungen anpasst.

## 1.3 Kurseinteilung

Der Kurs unterteilt sich in 3 große Blöcke.

1. Tag — Grundlagen von UNIX
   - Zielgruppe: Anfänger und Quereinsteiger von anderen Systemen
   - Voraussetzungen: keine
2. + 3. Tag — Shells - die Kommandooberfläche
   - Zielgruppe: alle UNIX Benutzer
   - Voraussetzungen: UNIX Grundlagen
4. Tag — Softwareentwicklung unter UNIX
   - Zielgruppe: SW-Entwickler, die unter UNIX arbeiten wollen

- Voraussetzungen: UNIX Grundlagen Kenntnisse der Shells

5. Tag — Einführung in X-Windows

- Zielgruppe: alle UNIX Benutzer die an einer Workstation arbeiten
- Voraussetzungen: UNIX Grundlagen, Shells

Sonderkapitel — Einführung in UNIX Texteditoren vi oder me

- Zielgruppe: Benutzer, die keinen Editor unter UNIX beherschen
- Voraussetzungen: keine

### 1.4 Was der Kurs nicht bietet !

- Unix specials für Eingeweihte
- netzwerkbezogene UNIX-Komponenten
  (mail, Filetransfer, etc)
- UNIX Systemadministration
  (Wie installiere/pflege ich UNIX)
- Programmiersprachen

## 2 Einführung

### 2.1 Was ist UNIX ?

UNIX ist heute eine Familie gleichartiger Betriebssysteme; der Name wurde von Brain Kerighan 1970 geprägt.
Warum ist UNIX eine Familie?
Von Unix gibt es heute eine Menge verschiedener Implementationen mit unterschiedlichen Erweiterungen.
Hier sind zunächst 2 Grundrichtungen festzustellen. Eine wird mit dem Schlagwort System V, die andere mit dem Schlagwort BSD belegt. Die folgende Tabelle zeigt die gängisten Varianten und der Zuordung zu den Grundrichtungen.

| Betriebssystem | System V | BSD |
|---|---|---|
| aix | x | |
| bsd 4.3 | | x |
| ultrix | | x |
| xenix | x | |
| sinix | x | |
| solaris | x | |
| uxpm | x | |

Die beiden Richtungen unterscheiden sich im wesentlichem in den Interna (und in der Programmierschnittstelle). Der Stand der heutigen Implementationen stellt in der Regel beide Programmierschnittstellen zur Verfügung.
Die Tendenz der zukünftigen Entwicklungen gehen dahin, beide Grundrichtungen zusammenzufassen zu einem neuen konsolitierten Konzept. Schlagwort ist hier OSF (Open Software Foundation).

## 2.2 Historie

UNIX begann mit seine Wurzeln 1968/69 in den Bell Laboratorien von AT&T.
Ken Thomson befasste sich zunächst primaer mit Dateisystemen. Da zu dieser Zeit die Rechnerkapazitäten knapp waren, und es eine Gelegenheit gab, eine ausrangierte PDP7 (Digital) zu benutzen, wurde das Dateisystem und ein rudumitärer Betriebssystemkern per Crossassemblierung auf dieses System übertragen. Auf 1971 datiert das erste Handbuch, mit dessen Hilfe das UNIX-System innerhalb der Laboratorien eingesetzt wurde. Zu diesem Zeitpunkt war das gesamte System noch in Assembler geschrieben und lief auf einer PDP 11.
Die Arbeiten an und um das System in Assembler waren zeitraubend; aus diesem Grunde wurde das System 1973 in C, welches parallel und sich gegenseitig beeinflussend entwickelt worden war, reimplemetiert. Diese Version wurde dann 1975 öffentlich zugänglich gemacht.
Das letzte große Projekt innerhalb der UNIX-Entwicklung durch die Bell Laboratorien war die Transformation des Systems auf eine hardwareunabhängige Grundlage. Dies wurde 1979 abgeschlossen.
Nach diesem Termin wurde die Entwicklung an unterschiedlichen Stellen fortgesetzt.

## 2.3 Charakteristika von UNIX

UNIX ist

- hardwareunabhängig
  lauffähig auf allen gängigen Prozessortypen
- funktionskompatibel über alle Implementationen
  gleiche Bedienung auf allen Rechnern
- Philosophie der verketteten Funktionen
- viele Programme sind public domain und im Source erhältlich
- interaktiv ausgerichtet

## 2.4 Grundsätzlicher Aufbau von UNIX

Hardwarevoraussetzungen aus der Sicht der Autoren

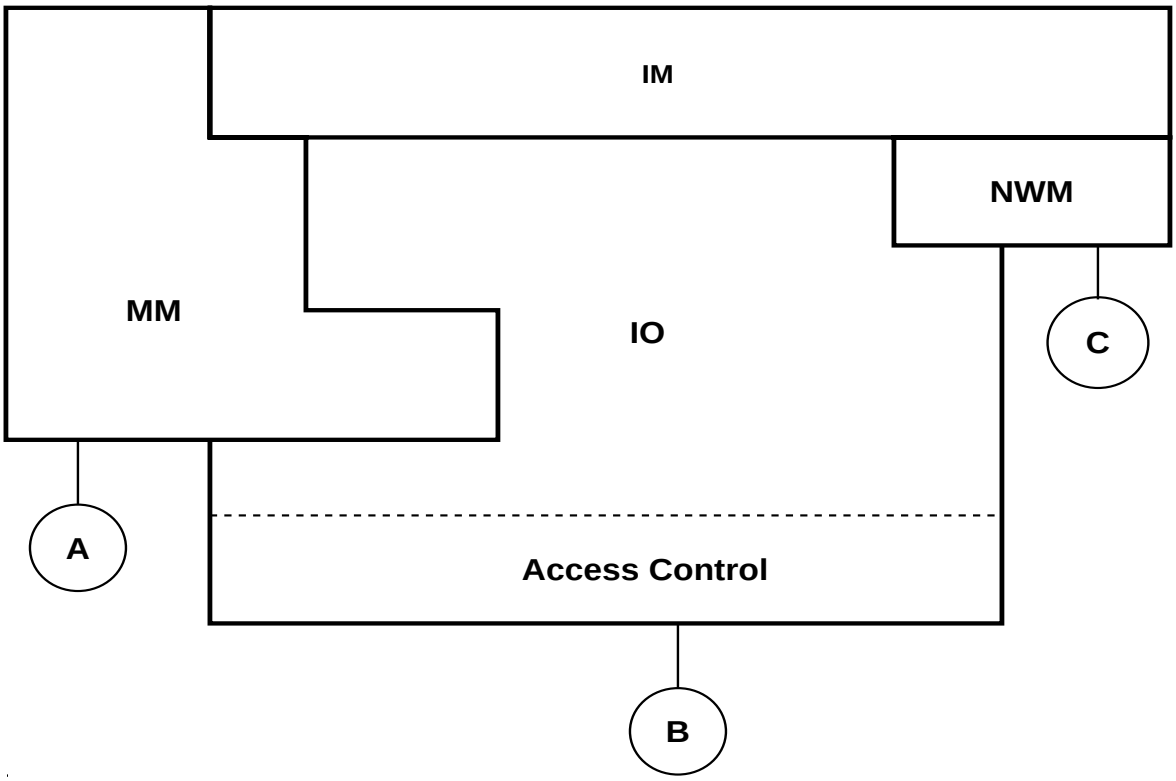
- Virtual Memory(VM)- Möglichkeiten $\Longrightarrow$ Paging
- 32-Bit Prozessorbreite
- Hauptspeicherausbau $\geq$ 16 MB
- Pagingspace $\geq$ 32MB (2-facher Hauptspeicherausbau)
- Terminalanschlüße pro CPU $\leq$ 8

### 2.4.1 Kernel (Betriebssystemkern)

Der Betriebssystemkern stellt das unverzichtbare Zentralstück des Systems dar; er wird beim Booten (Hochladen) des Hardwaresystems geladen.
Der Kernel besteht aus 2 Bereichen, einer Prozeßverwaltung (PV) und Dienstleistungsroutinen.
Die Dienstleistungsroutinen bilden einige Subsysteme, die im folgenden beschrieben werden.

- das IM-Subsystem
- das MM-Subsystem
- das NWM-Subsystem
- das IO-Subsystem
- das Util-Subsystem

Übersichtsbild Subsysteme




- Das Interrupt-Management(IM)-Subsystem ist der hardwareabhängige Teil des Betriebssystemkerns, in dem die Geräte (und auch teilweise das Memory) behandelt werden. Geräte arbeiten asynchron und melden ihre Veränderung durch eine Unterbrechung (Interrupt). Für jedes Gerät gibt es einen Treiber (device driver).

- Das Memory-Management(MM)-Subsystem stellt die Funktionalität des Virtual Memory zur Verfügung und hat unter anderem eine Programmierschnittstelle, um Speicherplatz zur Verfügung zu stellen (siehe A)
- Das NetWork-Management(NWM)-Subsystem behandelt alle Aufgaben, die mit dem Netzwerk in Verbindung stehen und stellt eine Schnittstelle (siehe C) zur Verfügung.

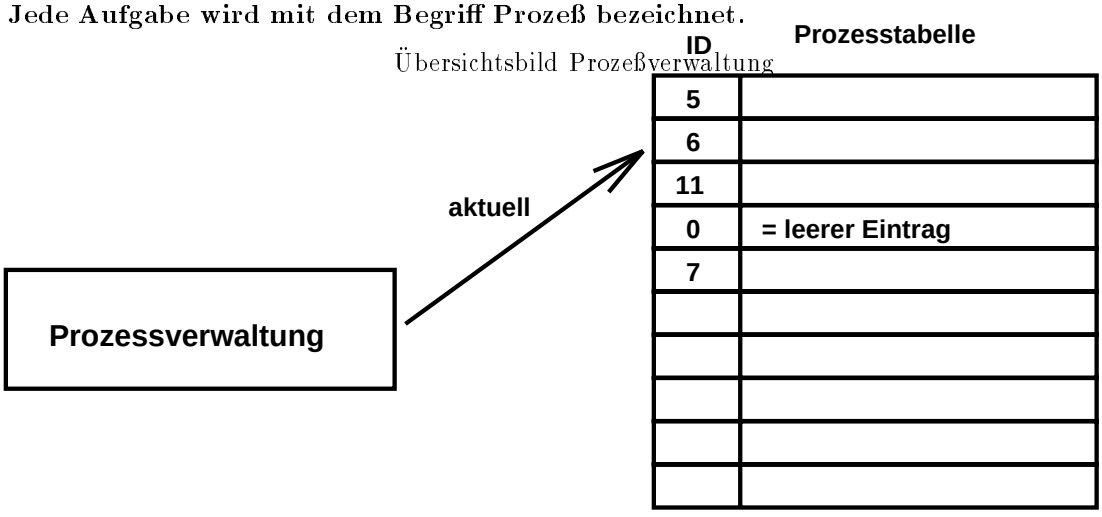
  Dieser Teil wird im weiteren nicht behandelt.
- Das Input/Output-Management(IO)-Subsystem behandelt alle Aufgabenstellungen, die mit der Ein- und Ausgabe von Daten auf externe Geräte zu tun haben und schließt die Zugriffskontrolle mit ein. Sie realisiert auch das Dateisystem. (siehe Schnittstelle B)
- Das Utilities(Util)-Subsystem erscheint in dem Übersichtsbild nicht, da es nicht eindeutig einordnenbar ist. Es umfasst eine Menge allgemeiner Funktionen, wie Datum, Zeit, Hardwareidentifikation, etc.

Es sei nochmals ausdrücklich darauf hingewiesen, daß alle Untersysteme sind, die nur durch den Anstoß durch externe Vorgänge in Betrieb genommen werden und sich wieder gezielt terminieren.

### 2.4.2 Prozeßverwaltung

Die Prozeßverwaltung (PV) hält den Status des Systems und kontrolliert die Resource CPU.
Die PV wird in Zeitintervallen per Interrupt angestossen (Zeitscheibe), oder per Softwareinterrupt aktiviert (event interrupt).
Die PV hält in einer Tabelle die jeweiligen aktuellen Aufgaben und deren Status. Bei der Aktivierung der PV wird der Status der zum Zeitpunkt aktiven Aufgabe gespeichert, der Status der zyklisch nachfolgenden Aufgabe geladen und dann die PV wieder verlassen.
**Jede Aufgabe wird mit dem Begriff Prozeß bezeichnet.**

Übersichtsbild Prozeßverwaltung




| PID Prozess-ID | Status | Pri | Owner | Start-Adr. | Akt.-Adr. | Father-Prozeß |
|---|---|---|---|---|---|---|

Eintrag in der Prozeßtabelle

Wichtig ist, daß der Prozeß eindeutig über eine Nummer identifiziert wird (und dies ist nicht der Index innerhalb der Prozeßtabelle).
Der Status eines Prozesses kann Verschiedenes enthalten. Insbesondere ist zu erwähnen, daß hier vermerkt wird, ob der Prozeß auf eine I/O Operation wartet. Ist dies der Fall, so wird der Prozeß solange nicht zyklisch aktiviert, bis sich der Status ändert.
Die Priorität kann bei der zyklischen Auswahl des neu zu aktivierenden Prozesses verwendet werden.
Die PV hat Möglichkeiten, einen Prozeß neu in die Tabelle aufzunehmen, bzw. zu entfernen.
Nach dem Booten (im ersten Moment) ist nur der Betriebssystemkern geladen. Danach wird automatisch ein Initialprozeß gestartet, der eine Startupdatei abarbeitet und gegebenfalls andere Prozesse startet.
Typische Aufgaben, die gestartet werden, sind:

- Systemtests
- Networkinitialisierung
- Druckerinitialisierung
- Servicejobs (Datenbankserver, etc)
- Loginprozeß

### 2.5 Handhabung eines UNIX-Systems

Bevor wir uns in die Welt der UNIX-Systemsoftware begeben, möchten wir zunächst, und damit es nicht in Vergessenheit gerät, einige Bemerkungen über die Handhabung von UNIX-Systemen machen.

1. Ein Unixsystem benötigt Administration; bitte überlassen Sie diese einem benannten Systemadministrator.
2. Wenn sich mehrere Benutzer eine Maschine teilen, denken Sie daran, daß es bei den Resourcen zu Kollisionen kommen kann; geben Sie alles frei, wenn Sie es nicht mehr benötigen.
3. Unix-Systeme sind Multiuser/Multiprozessysteme. Die bedeutet, daß diese Computer auch dann arbeiten (können), wenn sich auf dem Schirm nichts regt. Deshalb sollten Sie stets die Shutdownprozedur verwenden, um die Maschine auszuschalten.

## 3 Die Anmeldung am System (login)

### 3.1 Der Benutzer

Ein Unixsystem läßt nur eingetragene Benutzer zum Arbeiten zu. Die Eintragung erfolgt durch den Systemadministrator; dies ist ein Benutzer mit weitgehenden Rechten.
Der Systemadministrator benötigt zum Eintrag von dem Benutzer einen Namen (welcher im System eindeutig sein muß) und die Zuordung zu einer Gruppe von Benutzern.
Der Systemverwalter vervollständigt diese Angaben zu folgenden den Benutzer im System komplett beschreibenden Satz.

| | |
|---|---|
| user id (uid) | eindeutige Benutzernummer |

| | |
|---|---|
| user name | eindeutiger Benutzername |
| group id (gid) | eindeutige Gruppennummer |
| group name | eindeutiger Gruppenname |
| password | Passwort |
| home directory | Beginn des benutzerspezifischen Verzeichnisses |
| initial program | Kommandointerpreter |

user id und user name sind 2 gleichberechtigt eindeutige Begriffe, wobei die uid systemintern verwendet wird und der user name im Dialog mit dem Benutzer, um ihm damit einen Code mit besserer Begrifflichkeit zu geben.
Das gleiche gilt für die gid und den group name. Der Sinn der Gruppe wird im Zusammenhang mit den Dateisystem erläutert. Die weiteren Elemente werden im späteren Vorgehen, dann wenn sie benötigt werden, erläutert.
Wenn Sie Kontakt zum Rechner aufnehmen (über Terminal oder Netz) und sind direkt am Rechner angekommen, geben Sie solange *RETURN* (CR,Wagenrücklauf,ENTER, ⏎ ,etc) bis Sie zu Identifikation aufgefordert werden. Dies geschieht durch

```
login:
```

Geben Sie dann ihren Benutzernamen an und schliessen Sie die Eingabe ab. Der Abschluß jeder Eingabe geschieht durch die Taste *RETURN*, wenn es nicht anders vermerkt ist. Bitte achten Sie darauf (wie auch in allen späteren Fällen), daß UNIX zwischen Klein- und Großbuchstaben unterscheidet. Danach erscheint die Eingabeaufforderung

```
Password:
```

Geben Sie nun Ihr Passwort ein. Sollte kein Meldung mit dem Inhalt *login invalid* erscheinen und ein Prompt (Eingabeaufforderung) dargestellt werden, sind Sie erfolgreich eingeloggt und können mit dem System arbeiten. Im Fehlerfalle geben Sie wiederum solange *RETURN* ein, bis die Aufforderung zum login erscheint. Achten Sie diesmal darauf, daß keine falsche Eingabe geschieht. Sollte abermals keine Verbindung gelingen, halten Sie Rücksprache mit dem Systemadministrator.
Sonderfälle:
Bei dem ersten Einloggen wird bei manchen Systemen nicht nach dem Passwort gefragt oder direkt zum Ändern des Passworts aufgefordert.

### 3.1.1 Setzen des Passworts

Das Passwort ist ein Codewort, daß nur dem jeweiligen Benutzer bekannt ist (auch nicht dem Systemadministrator), um so eine unberechtigten Zugang unter dieser Benutzerkennung zu verhindern.
Das Verändern des Passwortes sollte auf Rechner, die öffentlich genutzt werden, mindestens 1 mal pro Monat erfolgen. Das Program **passwd** setzt das Passwort eines Benutzers auf einen neuen Wert.

Geben Sie hierzu das Kommando ein und schliessen Sie es mit *RETURN* ab.
Wenn ein altes Passwort gesetzt war, werden Sie zur Eingabe des alten Passwortes aufgefordert (Riegel gegen Scherzbolde). Wenn dieses korrekt eingegeben wurde, erscheint die Abfrage des neuen Passwortes, das sodann nochmals einzugeben ist, um Tippfehler zu erkennen. Bitte beachten Sie, daß Passwörter nicht auf dem Schirm erscheinen. Ist alles korrekt erfolgt, ist das Passwort hiernach geändert. Sollte das Program nicht erfolgreich abgeschlossen werden, hat das alte Passwort nach wie vor seine Gültigkeit.

### 3.1.2 Terminalbenutzung

Regeln für die Terminalbenutzung unter UNIX (Spezialtasten)
Ctrl meint im weiteren, daß zugleich mit der Buchstabentaste die Ctrl oder Strg Taste gedrückt werden soll.

| | |
|---|---|
| *RETURN* | ist der Abschluß eine Kommandos oder Eingabe. Andere Bezeichnungen sind : *ENTER CR Wagenrücklauf* |
| *Ctrl–C* | ist der interaktive Abruch eines Kommandos. |
| *Ctrl–U* | löscht die angefangene Kommandozeile. |
| *BS* | löscht das letzte Zeichen der Kommandozeile. Andere Bezeichnungen sind : *RUBOUT, DEL* |
| *Ctrl–D* | hat 2 Bedeutungen<br>1. führt es zum Abbruch der Verbindung zum Rechner, wenn es als Kommando verwendet wird (logout)<br>2. hat es die Bedeutung End-Of-Text (EOT) |

## 3.2 Der aktuelle Status

### 3.2.1 Was macht der Rechner ?

In diesem Abschnitt soll erklärt werden, was Multiprozeßfähigkeit bedeutet, und zum anderen,wie sich der Benutzer auf dem System über den Status informieren kann.
Im Abschnitt über die Prozeßverwaltung haben wir festgestellt, daß jede Aufgabe innerhalb von UNIX als Prozeß bezeichnet wird. Die Prozeße haben in der Tabelle einen Eintrag, welchen Besitzer (owner) sie zugeordnet sind. Dies wird über die Angabe der uid erreicht. Somit ist es möglich, alle Prozesse, die einem Benutzer zugeordnet sind, zu erkennen.
Prozesse entstehen, indem Sie von anderen Prozessen initiert werden. Prozesse können auf die Beendigung der von ihnen erzeugten Prozesse warten.
Beispiel:
Ein denkbare Möglichkeit, aus dem Editor heraus zu drucken ist folgende:
Stellen Sie sich den Editor als Prozeß A (Aufgabe A des Systems). Dieser erzeugt eine temporäre Datei. Prozeß A initiert Prozeß B mit der Aufgabe „Drucke temp. Datei“ . Prozeß A wartet auf die Beendigung von Prozeß B. Prozeß A startet sodann einen Prozeß C mit der Aufgabe „Lösche temp. Datei“ und fährt mit dem Editieren fort.
Stellen wir nun fest, daß die Information über den Benutzer, welches Programm $\Longrightarrow$ Aufgabe $\Longrightarrow$ Prozeß als erstes zu starten ist, vorliegt, so haben wir nachvollzogen, daß nach dem Einloggen ein Prozeß existiert. Die erste Aufgabe, die geladen wird, ist der Kommandointerpreter.

### 3.2.2 Prozeßkommandos

Der erste Befehl im Zusammenhang mit Prozeßen ist **ps** (process status). Geben Sie diesen Befehl unmittelbar nach dem Einloggen wie folgt

```
ps -l
```

ein und schliessen die Eingabe mit *RETURN* ab. Sie erhalten dann eine Aufstellung aller Prozesse, die Ihrer Benutzerkennung zugeordnet sind.

```
     F S  UID   PID  PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
200001 R  271 32241 67422  12  66 20 768e   172           pts/6  0:00 ps
240801 S  271 67422 02908  01  60 20 16e3   144           pts/6  0:00 ksh
```

Wie im obigen Beispiel erkenne Sie in der Regel 2 Prozesse

1. den Prozeß des Kommandointerpreters ksh
2. den Prozeß des Kommandos ps selbst

Von Interesse sind für den Beginner nur die Spalten S, PID, TIME, CMD.

| | |
|---|---|
| S | bezeichnet des Status und hat die Werte R (running) S (sleeping $\Longrightarrow$ wartet auf Prozeß) W (waiting $\Longrightarrow$ wartet auf IO). |
| PID | zeigt die Nummer des Prozesses an. |
| TIME | die verbrauchte CPU-Zeit des Prozesses |
| CMD | zeigt das geladene Programm an. |

Interessant ist, daß der Kommandointerpreterprozeß (MP) auf einen Prozeß wartet (Status S) und der ps-Prozeß (SP) arbeitet (Status R).
Zu interpretieren ist dies wie folgt:
Der MP hat den SP gestartet und wartet auf dessen Beendigung.
Der SP Prozeß erfüllt seine Aufgabe.
Der MP-Prozeß begleitet Sie durch die gesamte Sitzung (session).
Als nächstes werden wir, um einige Dinge veranschaulichen zu können, den Befehl `sleep` einführen.
`sleep` [*n*] wartet, wenn es gestartet wurde, n Sekunden und beendet sich dann.
Setzen wir also das Kommando `sleep 30` ab, so erscheint 30 Sekunden lang kein Prompt.
Der Kommandointerpreter hat eine Option, die es erlaubt, eine Unterprozeß zu starten, ohne auf dessen Beendigung zu warten. Hierzu wird das Zeichen

```
&
```

dem Befehl nachgestellt.
Auf den Befehl

```
sleep 30 &
```

wird die Prozeßnummer von sleep ausgegeben und es erscheint wieder der Prompt.
Setzen wir nun wieder ein `ps`-Kommando mit dem Argument `-l` innerhalb von 30 Sekunden nach dem `sleep`, so erhalten wir folgenden beispielhaften Output.

```
     F S  UID   PID  PPID   C PRI NI ADDR    SZ    WCHAN    TTY  TIME CMD
200801 S  271 04516 36396  00  64 24 cdb9   100           pts/3  0:00 sleep
200001 R  271 09381 36396  12  66 20 81b2   168           pts/3  0:00 ps
240801 S  271 36396 66347  01  60 20 1b42   152           pts/3  0:00 ksh
```

Hier können Sie nun parallele Prozesse (Multiprocessing) sehen.
Wenn Sie das Kommando `ps` ohne Argument verwenden, so erhalten Sie folgende Ausgabe.

```
  PID   TTY  TIME CMD
04516 pts/3  0:00 sleep
09381 pts/3  0:00 ps
36396 pts/3  0:00 ksh
```

Dies entspricht einer kurzen Form, die zu einem Kommando die jeweilige Prozessid anzeigt.
Geben Sie als nächstes ein Kommando

```
sleep 3600
```

ein.
Nun müßten Sie 1 Stunde warten, bis Sie das nächste Kommando eingeben könnten. Wenn Sie jedoch *Ctrl–C* eingeben, wird das Programm sofort abgebrochen.
Wenn Sie dagegen das Kommando

```
sleep 3600 &
```

eingeben, so läuft dieser Prozeß 1 Stunde im Hintergrund. Eine Möglichkeit, ihn vorher zu beenden, ist wie folgt. Mit `ps` ermitteln Sie zunächst die pid des sleep-Prozesses.
Mit dem Kommando

`kill` ⟨*pid*⟩

wird der Prozeß sofort beendet (Kontolle über `ps` möglich).

# 4 Filesysteme

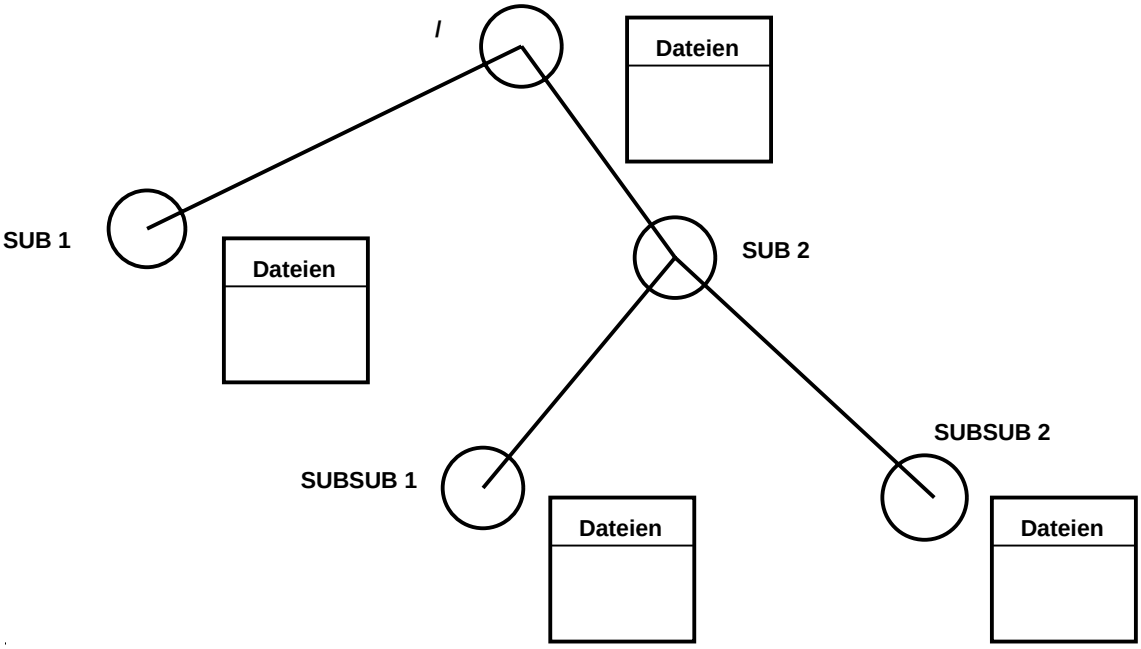
## 4.1 Basiswissen über Filesysteme

Das Dateisystem war eines der Ausgangspunkte von UNIX. Seine herausragende Eigenschaft ist die einfache generelle Struktur.
Grundlage der Überlegungen ist der Datenstrom; jede Datei wird als Aneinanderreihung von Bytes verstanden, deren Struktur (Beziehung untereinander) dem Dateisystem unbekannt sind und nur von der Anwendung definiert werden. Ob der Datenstrom aus einem Peripheriegerät, Terminal, Netz oder Platte kommt, ist hierbei unmaßgeblich.
Die 2. Überlegung ist, daß es nur Dateien gibt. Geräte, Verzeichnisse (Directories) und Daten sind alles Dateien; alle sind Datenströme.

## 4.2 Verzeichnisse

Ein UNIX-System hat nur ein Verzeichnis ( mit Unterverzeichnissen (mit ...)); dies bedeutet, daß es ein hierachisch-organisiertes Dateisystem gibt.
Die Wurzel des Dateisystems ist das Verzeichnis / (root directory). Hieran anschliessend existiert ein Baum.

Übersichtsbild Verzeichnisstruktur




Jedes Verzeichnis kann Dateien und Unterverzeichnisse enthalten. Verzeichnisse, die nur Daten enthalten, sind die Endknoten.
Ein Kommandointerpreter befindet sich von seinem Verständnis her an einer bestimmten Stelle im Verzeichnisbaum. Diese Stelle wird Arbeitsverzeichnis (working directory) genannt und hat den symbolischen Namen ".".
Wenn Sie sich einloggen, befindet sich der Kommandointerpreter innerhalb des Baumes in dem Ursprungsverzeichnis (home directory). Diese Stelle ist diejenige Verzweigung innerhalb des Gesamtverzeichnisses, die als Anfangselement der Benutzerstruktur vorgegeben wird. Diese Stelle wird bei der Benutzerdefinition mit festgelegt.

## 4.3 Grundregeln

### 4.3.1 Regeln

In diesem Abschnitt werden einigen Regeln und Empfehlungen im Zusammenhang mit Dateien angesprochen.
**Eine Regel:**
In einem Dateinamen darf das Zeichen "/" nicht vorkommen. Diese Zeichen ist reserviert, um Verzeichnisnamen voneinander zu trennen.
Eine zweite implizite Regel: Ein Dateiname besteht mindestens aus einem Zeichen.

### 4.3.2 Allgemeine Empfehlungen

- Dateinamen bis zu einer Länge von 14 Zeichen gehen immer (in allen Uniximplementationen); viele Systeme können mehr.

  - Verwenden Sie sinnvolle Dateinamen.
  - Dateien, die auf andere Systeme übertragen werden sollen, sollten der 14-er Empfehlung folgen.
  - Lokale Datei können nach den Systemmöglichkeiten benannt werden.
  - Für Programmierer gilt: Wenn ich Dateinamen fest vorsehe, sollten sie 14 Zeichen nicht überschreiten. Wenn Dateinamen einlesen werden sollen, soll einen gößerer Puffer vorsehen werden.

- Vermeiden Sie Sonderzeichen in Dateinamen, insbesondere an der 1. Stelle.
  - Alles, was druckbar ist, ist ein Zeichen.
  - Regelzeichen sind alle Buchstaben (klein und groß) sowie Ziffern.
  - Sonderzeichen ist der Rest außer "/"
  - Die Sonderzeichen sind in 2 Gruppen zu unterteilen:
    * Gruppe 1 enthält ",", "_", "-", "+".
    * Gruppe 2 enthält den Rest.
  - Sonderzeichen der 1. Gruppe dürfen in Dateinamen vorkommen. Wer Zeichen der 2. Gruppe verwendet, betreibt schlechten Stil.

- Dateien vergleichbaren Inhalts sollten gleiche Namensfortsätze (extensions) haben; diese werden durch das Zeichen "." abgetrennt. Extensions sind nicht in der Länge beschränkt.

  Ausführbare Programme haben keine extensions.

- Verzeichnisse dienen zur Strukturierung von Daten; Verzeichnisse mit mehr als 30 Dateien sollten die Ausnahme sein.

### 4.3.3 Common Sense

Grundempfehlungen, die allgemein gelten:

- Ein Benutzerverzeichnis hat in der Regel folgenden Aufbau:

  `./tmp` Verzeichnis der temporären Dateien
  `./app1` Verzeichnis der Applikationen
  `./app2`
  `./app3`
  ⋮
  `./app` $n$
  `./src` Verzeichnis der Quellcodes
  `./bin` Verzeichnis für eigene Programme

- Das Systemverzeichnis ist wie folgt aufgebaut:

  `/bin` Programme wie ls cd pwd
  `/dev` Geräte
  `/etc` Umgebungseinstellungen
  `/lib` Biliotheken
  `/tmp` Arbeitsbereich
  `/usr` Benutzerdaten

- Kleinbuchstaben sind Großbuchstaben vorzuziehen.

## 4.4 Etwas mehr Theorie

Wenn wir früher einiges Basiswissen über das Dateisystem angeführt haben, so wollen wir hier weitere Fakten einführen.

### 4.4.1 Implementation

Das Dateisystem mit seiner fixierten Schnittstelle gegenüber dem Restsystem gibt es heute in vielen Implementationsvarianten. Diese Varianten unterschieden sich im Verhalten bei Systemabbrüchen oder sind für große Dateien optimiert. Die Erwähnung ist wichtig, da diese Begriffe häufig in Beschreibungen vorkommen. Begriffe wie NFS, JFS, etc. haben jedoch auf die Benutzung von UNIX keinen Einfluß.

### 4.4.2 Harddisks (Platten)

Platten werden intern und durch Firmware in 1 .. n Partionen unterteilt. Will man eine Partition unter UNIX nutzen, so entspricht Sie nach der Formatierung einem Verzeichnis. Im Falle der Systempartition ist dieses Verzeichnis "/", in allen anderen Fällen ist es ein Unterverzeichnis. Die Datenmenge innerhalb einen solchen Verzeichnisses ist durch die Größe der Partition beschränkt.

### 4.4.3 Verzeichniseintrag für Dateien

Folgende Information über eine Datei liegt vor:

| | |
|---|---|
| filename | Name der Datei |
| size | Größe der Datei |
| create/modify date | |
| owner | uid |
| group | gid |
| access code | |
| access date | |
| disk adr. | |

Das create/modify date hält Datum und Zeit des letzten Schreibvorgangs auf die Datei fest.
Das access date hält Datum und Zeit des letzten Zugriffs auf die Datei fest.
Der access code wird im nächsten Abschnitt erklärt.
Die disk adr. ist der Hardwareverweis.

### 4.4.4 Details für Verzeichnisse

Jedes Verzeichnis enthält mindestens 2 Dateien.

1. Die Datei "." enthält den Inhalt des Verzeichnisses.
2. Die Datei ".." enthält eine Verweis auf das übergeordnete Verzeichnis.

Beide Dateien sind selbst vom Charakter her Verzeichnisse.

## 4.5 Zugriffsrechte

UNIX kennt drei Arten von Zugriffsrechten

- Datei darf gelesen werden `r` (read).
- Datei darf geschrieben werden `w` (write).
- Datei darf benutzt werden `x` (access).

Diese Rechte sind additiv.

UNIX kennt 3 Gruppen von Dateibenutzern

- Der Eigentümer der Datei `u` (user).
- Die Gruppeneigentümer der Datei `g` (group).
- Alle anderen `o` (other).

Da die Zugriffrechte für jeden der Zugreifenden angewendet wird, hat somit jede Datei 9 Accesscodes. Diese werden als 9 Bits organisiert.

```
rwx rwx rwx
u   g   o
```

Bit gesetzt $\Longrightarrow$ Recht erteilt.

In diesem Zusammenhang wird auch der Begriff Gruppe in seiner Anwendung klar. Benutzer mit gleichen Aufgaben (Projekte, etc) können sich somit Dateien teilen, die für andere nicht zugreifbar sind. Dies beeinträchtigt aber nicht die eigenen Rechte.

## 4.6 Erste Befehle

`pwd` (Print working directory) ist ein Befehl, der Ihnen den Namen des Arbeitsverzeichnisses auf dem Bildschirm ausgibt. Mit ihm können Sie sich jederzeit über Ihre Position im Dateibaum informieren.
Der Befehl hat keine Argumente

`ls` (list) ist eine Befehl, der den Inhalt des aktuellen Verzeichnisses auf den Bildschirm ausgibt .

`mkdir` (make directory) ist eine Befehl, um ein neues Verzeichnis anzulegen. Als Argument wird der Name des neu anzulegenden Verzeichnisses angegeben.

`cd` (change directory) ist ein Befehl, der das aktuelle Verzeichnis wechselt.

- ohne Argument wechselt der Befehl in das homedirectory.
- ist das Argument ein Verzeichnis, so wird das working directory auf dieses Verzeichnis eingestellt.
- ist das Argument .. , wechselt das Programm aus dem aktuellen Verzeichnis in das übergeordnete Verzeichnis.

`touch` (touch file) ist ein Befehl, der später eingehend erläutert wird. Mit einem Argument versehen, erzeugt er eine Datei mit dem Namen Argument, wenn diese nicht existiert, anderfalls geschieht nichts.

## 4.7 Dateimanipulationsbefehle

Kehren wir wieder zu den Befehlen zurück. Zunächst vorangestellt werden soll, wie sich ein Befehl in der Regel aufbaut.
`Befehl` [*Option*] [*Datei.1*] [*Dateien.2*] .....
Am Anfang der Kommandozeile steht zunächst stets immer der Befehl. Dann folgen die Optionen (switches); diese beginnen in der Regel mit einem "-" und können mehrfach auftreten. Danach folgt die zu bearbeitenden Datei und zu Schluß die Hilfsdateien.
Beispiele:

- `ls` ohne Option
- `ps -l` die Option l wird angegeben.
- `touch file` die zu bearbeitende Datei wird angegeben.
- `frt -c test.f` Eine Kombination der Möglichkeiten

Die Dateien können stets absolut oder relativ zum Arbeitsverzeichnis angegeben werden. Beginnt ein Dateiname mit einem "/" ist es eine absolute Angabe. Ist es ein sonstiges Zeichen, wird der Inhalt des pwd-Kommandos vorangestellt.

### 4.7.1 Befehle

`ls` list
Diesen Befehl haben wir bereits früher eingeführt. Dieser Befehl hat auch Optionen und die Möglichkeit, Datei zu spezifizieren.
Folgende Optionen sind gebräuchlich:

- `-l` zeige die Verzeichnisinformation in der ausführlichen Form an.
- `-a` zeige auch Dateien an, deren Namen mit "." beginnen.
- `-t` zeige die Verzeichnisinformation in der Reihenfolge des creation date an.

Wenn Sie Dateien spezifizieren, wird nur für diese Dateien die Information dargestellt (wichtig im Zusammenhang mit `-l`).
Beispiel:

```
ls -a -l
```

zeigt folgende Ausgabe

```
total 16
drwxr-xr-x   2 mln      hrz          512 Jul 20 13:10 .
drwxrwxrwx  18 mln      hrz         2048 Jul 20 13:10 ..
-rw-r--r--   1 mln      hrz            0 Jul 20 13:10 file1
-rw-r--r--   1 mln      hrz            0 Jul 20 13:10 file2
```

Hier können Sie nun erkennen,

- welche Zugriffsrechte auf die Datei existieren
- wer der Besitzer der Datei ist
- welcher Gruppe die Datei zugeordnet ist
- die Größe der Datei

- das creation/modification date
- den Namen der Datei
- die Art der Datei

Die Art der Datei wird durch das erste Zeichen der Zeile bestimmt:
`d` bedeutet Verzeichnis.
Die Dateien "`.`" und "`..`" haben wir bereits vorher besprochen.

`rm` (remove) löscht Dateien.

`rm` [*Optionen*] ⟨*file*⟩ ...

Gängige Optionen sind :
`-i` fragt zunächst, ob die Datei gelöscht werden soll.

Um Verzeichnisse zu löschen existiert der Befehl
`rmdir`
der analog zu rm verwendet wird.
`mv` (move) ist eine Befehl, der Dateien innerhalb des Verzeichnisbaums bewegt.
`mv` [*Optionen*] ⟨*file1*⟩ ⟨*file2*⟩
`mv` [*Optionen*] ⟨*file*⟩ ... ⟨*dir*⟩

Dieser Befehl verschiebt die Datei(en) in das Ziel, bzw wenn sie sich beide im gleichen Verzeichnis befinden, werden sie umbenannt. Ist das Ziel ein Verzeichnis, so werden die Dateien dort hineingeschoben.
Existiert das Ziel, so wird es überschrieben.

Gängige Optionen sind :
`-i` fragt zunächst, ob das Ziel überschrieben werden soll.

Der Befehl `cp` (copy) entspricht den Befehl mv, nur daß das Original erhalten bleibt. Er hat eine weitere Option `-r` , bei beim Verzeichnis kopieren alle Unterverzeichnisse mitkopiert.
Die nächsten 2 Befehle modifizieren die Verzeichniseinträge.

`touch` setzt das creation date bei einer bestehenden Datei auf die aktuelle Uhrzeit, ansonsten legt der Befehl eine leere Datei an.
`touch` ⟨*file1*⟩ ...

`chmod` (change modus) verändert die Zugriffsrechte.
`chmod` ⟨*Optionen*⟩ ⟨*file1*⟩ ...

Die Option besteht stets aus bis zu 3 Buchstaben

- Buchstabe 1 Dateibenutzer (ugo)
- Buchstabe 2 +/- setze Recht/Lösche Recht
- Buchstabe 3 Art des Zugriffsrechts (rwx)

Entfällt der 1. Buchstabe, so gilt die Rechtezuweisung für alle Gruppen.
Sinnvolle Rechtezuweisung sind in der Regel so, daß die Rechte vom Benutzer zu allen anderen hin abnehmen.
Die Rechte können auch als Oktalzahlen spezifiziert werden; dies soll hier nicht erläutert werden.
Beispiele:
chmod +w file $\Longrightarrow$ alle haben Schreibrechte
chmod o-w file $\Longrightarrow$ nur Gruppe und Besitzer haben Schreibrechte
Befehle, die den Plattenplatz transparent machen.

`du` (disk usage) zeigt, wieviel Plattenplatz durch das Verzeichnis belegt wird. Dabei werden auch die Unterverzeichnisse mit aufsummiert.

`du` [*dir*]
Das optionale Argument kann ein Verzeichnis sein, für das die Summierung durchzuführen ist; bei fehlendem Argument wird das aktuelle Verzeichnis dargestellt.
Beispiel der Ausgabe:

```
128568 lang/adb
8 lang/aph
8 lang/gph
800 lang/pcb/tmp
12288 lang/pcb
8 lang/pst
8 lang/gdi
8 lang/sim
16 lang/tmp
40 lang/cmd
8 lang/din
8 lang/dout
80 lang/drill
416 lang/edb
48 lang/lst
8 lang/pif
40 lang/plot
8 lang/prof
8 lang/text
3296 lang/cxf
8688 lang/ka/adb
19352 lang/ka
172864 lang
```

Die Zahlenangaben beziehen sich auf 512-Byte Einheiten.

`df` (disk free) zeigt, wie gross die Platte jeweils ist, und den Grad ihrer Belegung.

`df` [*dir*]
Das optionale Argument kann ein Verzeichnis sein, für das die Darstellung durchzuühren ist.
Beispiel der Ausgabe:

```
Filesystem    Total KB    free %used   iused %iused Mounted on
rs3:/usr/local  524288   40072   92%   -       -    /tmp_mnt/usr/local/etc
rs3:/udsk      2097152   51276   97%   -       -    /tmp_mnt/home/mln
rs3:/usr/local  524288   40072   92%   -       -    /tmp_mnt/usr/local/X11
rs3:/udsk      2097152   51276   97%   -       -    /tmp_mnt/home/kehr
rs3:/usr/local  524288   40072   92%   -       -    /tmp_mnt/usr/local/bin
rs3:/usr/local  524288   40072   92%   -       -    /tmp_mnt/usr/local/man
rs3:/usr/local  524288   40072   92%   -       -    /tmp_mnt/usr/lpp/info/mnt
```

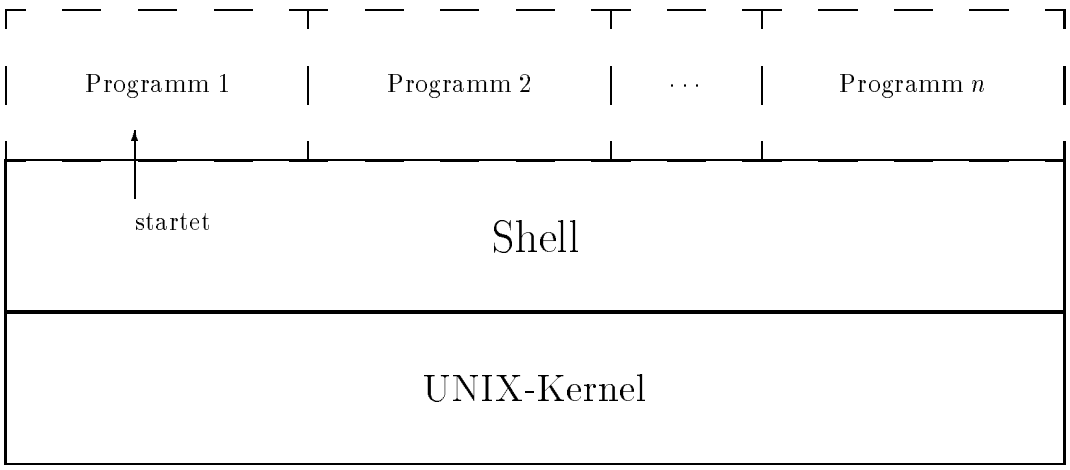
Die erste Spalte zeigt an, wo sich die Platte befindet.
Die zweite Spalte zeigt an, wie groß die Platte ist.
Die dritte Spalte zeigt an, wieviel freier Platz noch zur Verfügung steht.

Die letzte Spalte zeigt, welchem Verzeichnis dies entspricht.
Weitere spezielle Befehle und herausragende Dateiverzeichnisse werden im Anschluß an die Kommandoprozeduren später erklärt.

# 5 Einführung in die Shell

## 5.1 Was ist die Shell ?

Der Begriff *Shell* bedeutet Muschel. Gemeint ist damit eine Schale um den UNIX-Kernel herum die dem Benutzer den Dialog mit dem Kern-System (*Kernel*) ermöglicht. Die Shell ist zunächst nicht anderes als ein ganz normales Programm von denen UNIX einige hundert bis tausend besitzt.
Dieses Programm besitzt aber Funktionen, die es zu einem ganz besonderen Teil eines UNIX-Systemes werden lassen.




## 5.2 Fähigkeiten der Shell ?

- Interaktive Benutzung von UNIX (**Kommandoebene**)

  Die Shell kann UNIX-Kommandos entgegennehmen und starten, sobald der Benutzer der Shell die eingegebene Zeile übergibt. Erst nach Eingabe der Return-Taste „sieht“ die Shell die Zeile und beginnt mit deren *Interpretation*.

- Programmierung (**Shellskripts**)

  Shellskripts sind Text-Dateien die zeilenweise von der Shell interpretiert werden. Die Shell ist also auch eine sehr leistungsfähige Interpretersprache. In jeder Zeile eines Shellskripts stehen also die Befehle, die man auch von Tastatur eingeben könnte.

  *Wichtig für die Entwicklung von Programmen ist also die Möglichkeit die Befehle vor dem Programmieren auszuprobieren, da sie sich genauso später im Shellskript verhalten !*

### 5.3 Was macht die Shell ?

- Kommandozeilen-Interpretation
- Programm-Ausführung
- Filenamen-Substitution
- Ein-/Ausgabe-Umlenkung
- Pipeline-Handling
- Umgebungs-Kontrolle (*Environment*)

### 5.4 Shell-Geschichte

- UNIX ist nicht auf eine einzige Shell beschränkt
- Da die Shell „nur“ ein Programm wie jedes andere ist, kann auch je nach Bedarf eine andere Shell benutzt werden.
- Es gibt verschiedene Shells u.a.
  - Bourne-Shell (`bsh`); ist im Normalfall immer verfügbar (Quasi-Standard)
  - C-Shell (`csh`) als Shell mit C-ähnlicher Notation (unter Systemadministratoren verbreitet). Im PD-Bereich gibt es die kompatible `tcsh`.
  - Korn-Shell (`ksh`) als Weiterentwicklung der Bourne-Shell. Darüberhinaus gibt es noch die `bash` des GNU–Projektes, die einige Features der C–Shell mit denen der Korn–Shell verbindet.

Wir beschäftigen uns hier im folgenden mit der Korn-Shell, da sie wesentliche Erweiterungen gegenüber der Bourne-Shell besitzt und immer beliebter wird.

## 6 Die Shell auf der Kommandoebene

Die Syntax von Befehlen auf der Kommandoebene haben folgende Gestalt :

⟨*kommando*⟩ [⟨*optionen*⟩]... [⟨*argumente*⟩]... ⏎

## 7 Das Variablensystem der Shell

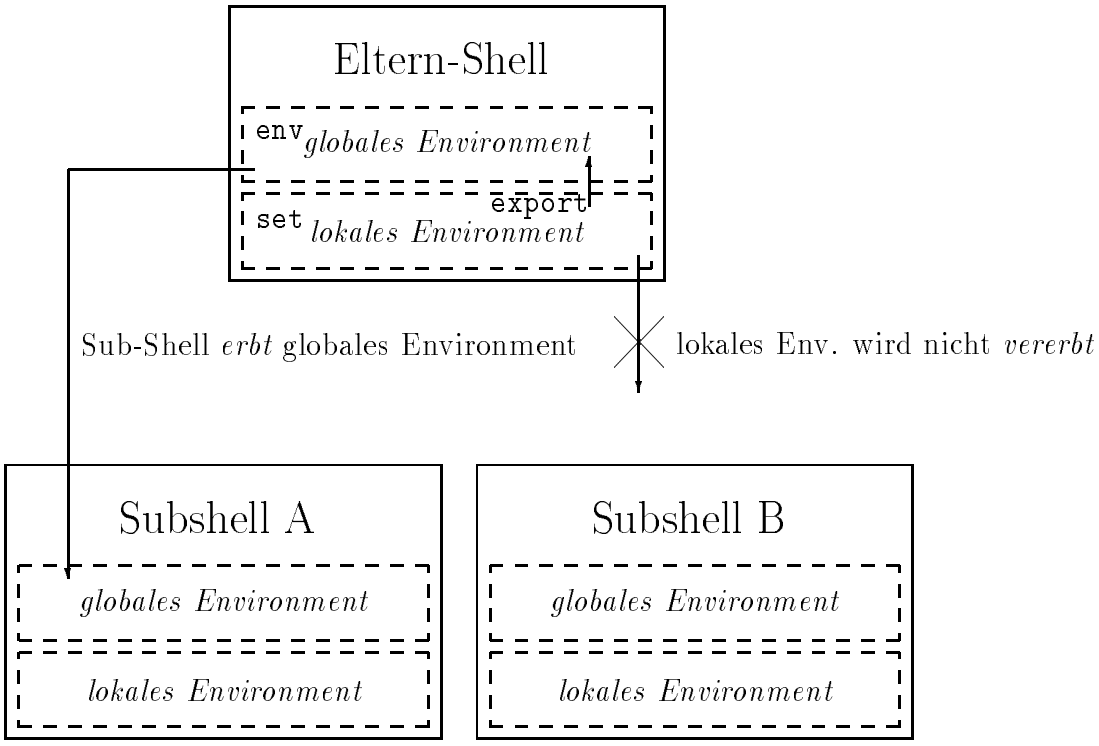
### 7.1 Variablen im Environment

- In der Umgebung (*local Environment*) können Variablen definiert und abgelegt werden. Variablen sind Platzhalter für Texte deren Wert sich durch Neubelegung geändert werden kann.
- Eine Zuweisung hat die Form:

  ⟨*variablenname*⟩=⟨*wert*⟩

  ♢ Keine Leerzeichen vor dem Gleichheitszeichen !
- Namenseinschränkungen: [a–z A–Z][a–z A–Z _]...
  insg. 14 Zeichen lang
  UNIX-Befehle : `set` zum setzen und definieren von Variablen, `unset` zum Löschen von Variablen

- Variablen werden über `$`⟨*variable*⟩ angesprochen, bzw. mit `$`{⟨*variable*⟩}
- Variablen können mit `export` ⟨*variable*⟩ in die globale Umgebung weitergegeben werden. Dort sind sie dann auch für Sub-Shells erreichbar.
  UNIX-Befehle : `env` zum ermitteln der globalen Umgebung




## 7.2 UNIX-System-Variablen

| | |
|---|---|
| `HOME` | enthält das Home-Verzeichnis des Users |
| `LOGNAME` | Name des Users |
| `IFS` | enthält die Trennzeichen für IFS-Trennung |
| `PATH` | enthält die durch einen Doppelpunkt getrennten Verzeichnisse, in denen nach auszuführenden Programmen gesucht wird |
| `PS1` | Standard-Promptzeichenfolge |
| `PS2` | die Promptzeichenfoge wenn die Shell nach weiteren Eingaben verlangt |
| `TERM` | der Terminaltyp |
| `SHELL` | Name der laufenden Shell |

## 7.3 Besondere Variablen

Allgemeine Variablen :

| | |
|---|---|
| `$?` | liefert den Wert des letzten Exit-Codes |
| `$$` | liefert die Prozeß-ID der laufenden Shell. (u.a. für die Benennung von Temporärdateien wichtig) |
| `$-` | die gesetzten Shellflags |

Im Zusammenhang mit Shellskripts :

| | |
|---|---|
| `$0` | liefert den Namen des Shellskripts |
| `$1..$9` | liefert den 1. bis 9. übergebenen Parameter<br>`shift` $\langle n \rangle$ verschiebt Parameter um $\langle n \rangle$ Positionen nach vorne |
| `$#` | liefert die Anzahl der übergebenen Argumente |
| `$*` | den Satz der übergebenen Argumente als Zeichenkette<br>`$*` $\Longrightarrow$ `$1␣$2␣...␣$n` |

# 8 Kommandozeilenverarbeitung

Der kompliziertesteste Teil ist in der Regel der Verarbeitungsprozeß einer Kommandozeile.
Aus diesem Grund soll hier mit besonderer Ausführlichkeit darauf eingegangen werden.

## 8.1 Wie wird eine Zeile bearbeitet ?

Reihenfolge der Interpretation :

1. Eingabe lesen und parsen
2. Verbose-Tracing
3. Parameter-Ersetzung
4. Kommando-Ersetzung
5. Ein-/Ausgabe-Umlenkung
6. IFS-Trennung
7. Filenamen-Expansion
8. Ausführungs-Trace
9. Kommandoausführung

## 8.2 Was macht die Shell mit der eingegebenen Zeile ?

### 8.2.1 Eingabe lesen und parsen

- Einlesen der Kommandozeile vom Terminal oder aus einer Datei
- Speichern der Eingabezeichen in einem Puffer.
  Die Shell bearbeitet die Eingabezeile erst nach dem ⏎ bzw. dem Newline-Zeichen.

- Durchsuchen der Zeile nach den folgenden Zeichen, die als Stoppzeichen interpretiert werden :

| | |
|---|---|
| Semikolon | ; |
| Background-Task | `&` |
| Logisches UND | `&&` |
| Logisches ODER | `\|\|` |
| Newline-Zeichen | \n |

- Teilen der Zeile in Worte durch die Suche nach Leerzeichen und Tabulatorzeichen (sog. *White Space*)

### 8.2.2 Verbose–Tracing

Das Verbose–Tracing erlaubt eine Ausgabekontrolle aller Shellkommandos.
Mit dem Befehl `set` ⟨*flags*⟩ können Shellflags gesetzt bzw. zurückgesetzt werden.
Ein Aufruf der Form

`ksh` ⟨`-`*Flags*⟩ ⟨*datei*⟩ ⟨*argument*⟩...

ruft eine neue Shell mit den gesetzten Flags auf und führt darin das Shellskript ⟨*datei*⟩ mit den angegebenen Argumenten ⟨*argument*⟩ aus.
Falls das Verbose-Flag (`-v`) gesetzt ist, wird die eingelesene Zeile auf *stderr* ausgegeben.

*Diese Möglichkeit sollte man bei der Shell-Programmierung ausgiebig nutzen.*

### 8.2.3 Parameter-Ersetzung

- Ein Parameter beginnt immer mit dem Dollar-Zeichen (`$`)
- Es gibt folgende Parameterarten
  - Positionale Variablen (`$0`...`$9`)
  - Umgebungs-Variablen-Ersetzungen (`$`⟨*variable*⟩)
  - sonstige Shellvariablen (`$# $$`...)
  - sonstige besondere Ersetzungsformen

### 8.2.4 Kommandoersetzung

Um z.B. die Ausgabe eines Kommandos einer Variablen zuzuweisen ( Bsp: Bearbeiten des Datums mit dem `date`–Befehl ) muß eine spezielle textuelle Ersetzung für Kommandos stattfinden.

- Dies betrifft alle Kommandos, die mit *Backquotes* ( ` ) umgeben sind
  - Die entsprechenden Kommandos werden ausgewertet und ausgeführt
  - Die Textausgabe auf *stdout* dieser Kommandos ersetzt den ursprünglichen Ausdruck der Kommandozeile
- Das auszuführende Kommando darf enthalten :
  - Sequentielle Kommandos
  - Pipelines
  - mit Klammern gruppierte Kommandos
- Zusätzliche Blanks, Tabs oder Newlines werden später von der IFS-Trennung entfernt

### 8.2.5 Ein-/Ausgabe-Umlenkung

- Es wird nach Ein-/Ausgabe-Umlenkungen gesucht.
  Die dafür zuständigen Zeichen sind `< > << und >>`.
- Falls eine Umlenkung existiert :
  1. der Orginal-Filedeskriptor wird geschlossen
     0 ist der *stdin*–Filedeskriptor (default: Tastatur)
     1 ist der *stdout*–Filedeskriptor (default: Bildschirm, Sofortausgabe)
     2 ist der *stderr*–Filedeskriptor (default: Bildschirm–Ausgabe, aber keine Sofortausgabe)
  2. der Filedeskriptor wird mit dem neuen Wert geöffnet
  3. die Umlenkungszeichen werden von der Kommandozeile entfernt

Beispiele:

```
$ latex texfile.tex 2>/dev/null
$ more < textfile
$ ls 1> dateien
$ ksh -vx shellskript 1>testlauf 2>/dev/null
```

### 8.2.6 Pipes

Pipes sind eine Form der Verkettung von Kommandos ohne Temporäre Dateien:
Beispiel:

```
$ grep "heute" mytext > mytext.heute
$ grep "morgen" mytext.heute > mytext.heute.morgen
$ more mytext.heute.morgen
$ rm mytext.heute.morgen
```

Mit Hilfe von Pipes kann man kürzer scheiben:

```
$ grep "heute" mytext | grep "morgen" | more
```

Bei Pipes wird *stdout* des $n$. Befehls auf *stdin* des $(n+1)$. Befehls gesetzt. Dadurch braucht sich der Benutzer nicht um das Anlegen von Temporärdateien zu kümmern, wie es im ersten Beispiel der Fall war. Bei Pipes werden die „unsichtbaren“ Dateien vom Betriebssystem verwaltet und auch wieder gelöscht.

### 8.2.7 IFS-Trennung

- IFS bedeutet *Inter-Field-Separation*.
- IFS wird benötigt, da sich die Kommandozeile durch die vorangegangenen Substitutionen verändert haben kann.
- Die Kommandozeile wird neu geparst bezüglich der IFS-Zeichen :
  - Die IFS-Zeichen werden in der Umgebungsvariable `IFS` gespeichert.
  - Jedes Zeichen der Kommandozeile, welches in der IFS-Variable enthalten ist wird mit einem Leerzeichen ersetzt. Leerzeichen begrenzen die Worte.
  - Alle nichtgequoteten Blanks, Tabs und Newlines werden entfernt.
  - Alle gequoteten Variablen-Zuweisungen sind vor der IFS-Trennung geschützt.

  ♢ Innerhalb von Worten wird nicht getrennt !

### 8.2.8 Filenamen–Expansion

- Die Shell sucht nach Worten, welche die *Metacharakter* (`* ? [ ]`) enthalten.

| | |
|---|---|
| `*` | steht für eine beliebige Zeichenkette (Ausnahme: Files mit führendem Punkt) |
| `?` | steht für ein beliebiges einzelnes Zeichen |
| `[`⟨*chars*⟩`]` | steht für einen der Buchstaben in der Zeichenliste.<br>Mit `[aAcC]` ist also einer der 4 angegebenen Buchstaben verlangt.<br>Mit `[a-z]` sind alle kleinen Buchstaben von a–z verlangt. Mit `[!abcd]` sind alle Zeichen *außer* a,b,c und d verlangt. |

- Ein *Patternmatching* wird im aktuellen oder spezifizierten Directory durchgeführt.
- Wurde kein Match gefunden, so bleiben die Metacharakter unverändert.
- Wurden Matches gefunden, so ersetzen alle Matches die ursprünglichen Metacharakter.

### 8.2.9 Beispiele zur Filenamen–Expansion

Ein Verzeichnis enthalte folgende Dateien :

```
abc.c   abcd.c  Makefile
ver.0   ver.1   ver.2
```

Folgende Befehle werden ausgeführt :

```
$ ls a*
abc.c abcd.c
$ ls ver.[01]
$ ver.0 ver.1
$ ls [A-Z]*
Makefile
$ ls ?????
abc.c ver.0 ver.1 ver.2
```

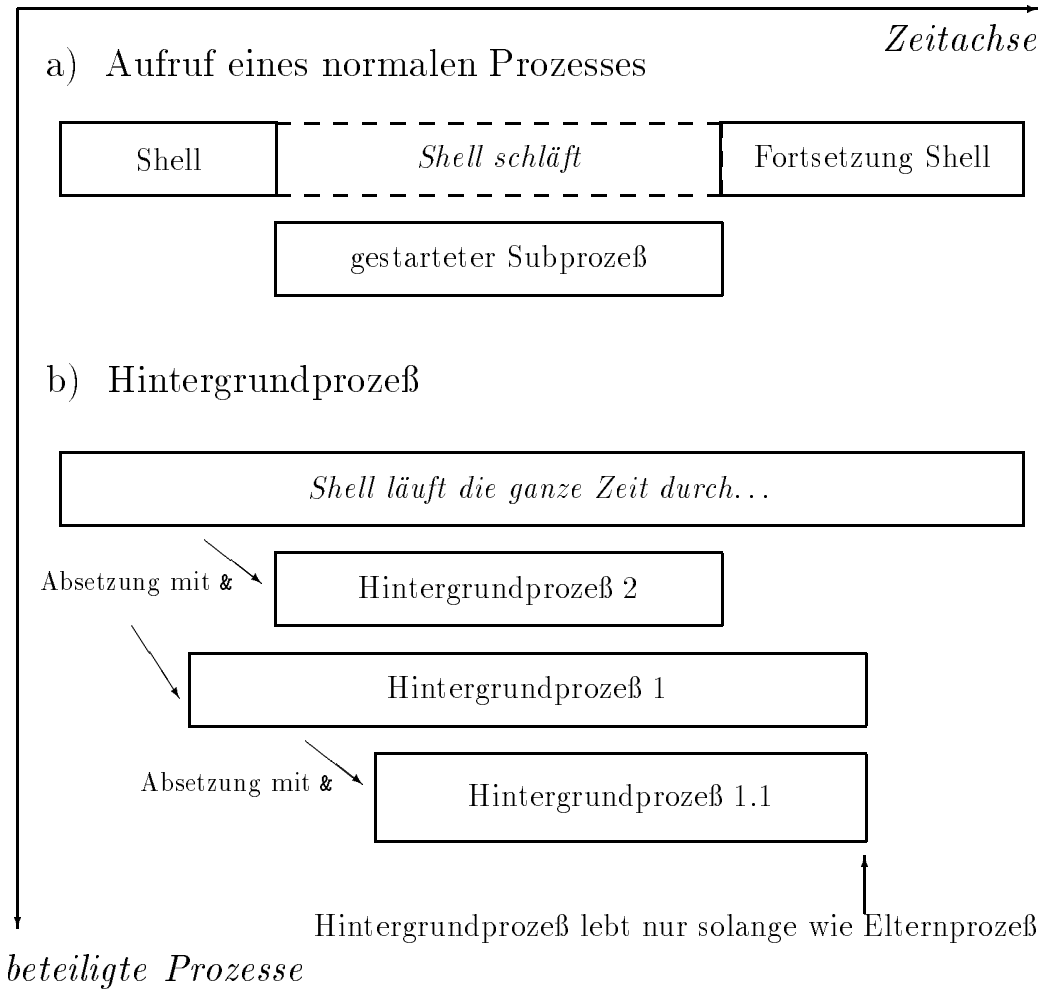
### 8.2.10 Ausführungs-Trace

- Falls das Ausführungs-Trace-Flag (`-x`) gesetzt ist (siehe Verbose-Tracing), so wird die Kommandozeile im momentan substituierten Zustand auf *stderr* ausgegeben. Die Zeile wird mit dem Präfix (`+`) ausgegeben.
- Variablen-Zuweisungen werden ohne einen Präfix ausgegeben.

### 8.2.11 Kommandoausführung

- Handelt es sich um ein in der Shell eingebautes Kommando (*Shell-Builtin*) so wird es jetzt direkt ausgeführt.
- Ansonsten wird das Programm in den Speicher geladen und gestartet. Die Shell „legt sich schlafen“ solange bis der Kind-Prozeß beendet ist.

# 9 Prozeß-Hierarchie




## 9.1 Was passiert beim Programmaufruf ?

1. Die aktuelle Shell stellt für das aufzurufende Programm ein Environment zur Verfügung, in dem sich dann auch alle exportierten Variablen befinden.
2. Das Programm wird als Kindprozeß gestartet.
3. Die aktuelle Shell „legt sich schlafen“ bis der Kindprozeß beendet ist.

Ausnahmen :

1. wurde der Kind–Prozeß als *Background-Task* gestartet, so läuft die Shell weiter und wartet nicht auf die Beendigung des Kind-Prozesses.
2. In einer Pipe werden die beteiligten Kommandos ebenfalls als Hintergrund-Prozesse abgesetzt.

*Werden in einem Sub–Prozeß beispielsweise Verzeichnisse gewechselt, so haben sich diese Verzeichniswechsel nicht auf die darüberliegende Shell ausgewirkt.*

# 10 Befehle der Shell

Die Shell verfügt über Befehle, die nicht als externe Kommandos realisiert sind, sondern direkt von der Shell interpretiert werden (sog. *Shell Builtins*).

## 10.1 Direkte Befehle (Shell-Builtins)

`#` [*irgendetwas*]
Kommentar–Befehl

`:`
Dies ist ein nichtstu–Kommando mit Exitcode 0

`;`
Kommando–Separator; trennt Befehle in einer Zeile voneinander

`(`⟨*kommandoliste*⟩`)`
Zusammenfassen von Kommandos in einer Subshell

`{`⟨*kommandoliste*⟩`}`
Zusammenfassen von Kommandos in der aktuellen Shell

`echo` ⟨*zeichenkette*⟩
Ausgabe (kann umgelenkt werden) Besondere Zeichen: `\t` Tabulator, `\n` Zeilenvorschub, `\c` unterdrückt Zeilenvorschub am Ende

`read` ⟨*variable*⟩...
Liest eine Eingabezeile von *stdin* und verteilt die gelesenen Worte wie folgt auf die Variablen: 1. Variable = 1. Wort, 2.Variable = 2. Wort,..., letzte Variable = „verbliebene Worte“

`exit` [⟨*exitcode*⟩]
verläßt die aktuelle Shell mit dem Exit-Code des letzten Befehls oder dem anggegebenen Exitcode.

`.` ⟨*shellskript*⟩
führt das Shellskript in der momentanen Shell aus.

*Es wird kein Subprozeß gestartet (Bsp.: Verzeichniswechsel zeigen Wirkung) ! Wirkung, als ob die Befehle direkt in der interaktiven Shell eingetippt worden wären.*

`exec` ⟨*progname*⟩
ersetzt die momentane Shell durch ⟨*progname*⟩. Die alte Shell wird abgebrochen.

`eval` ⟨*ausdruck*⟩
Mit Hilfe von `eval` ist es möglich eine Zeichenkette von der Shell direkt von Grund auf neu parsen und ausführen zu lassen. `eval 'ls -la'` liefert das gleiche Ergebnis wie `ls -la`.

`expr` ⟨*ausdruck*⟩
wertet den folgenden Ausdruck arithmetisch aus. Erlaubte Operatoren sind:

| | |
|---|---|
| `+ -` | Addition und Subtraktion |
| `\* /` | Multiplikation und Division |
| `%` | Divisionsrest (Modulo) |
| `\( \)` | dürfen zur Klammerung verwendet werden |

*Alle Operatoren sind nur ganzzahlig und müssen durch Blanks getrennt sein !*

```
$ expr 2 \* 5 - \( 6 / 2 \)
7
```

`type` ⟨*kommando*⟩
ermittelt, ob ⟨*kommando*⟩ ein Shell-Builtin-Befehl oder ein externer Befehl ist (dann mit Pfadangabe).

```
$ type alias
alias is a shell builtin
$ type mkdir
mkdir is /bin/mkdir
```

`alias` [⟨*name*⟩=⟨*ersetzung*⟩]
ermöglicht die Definition von Kurznamen für Kommandos (Bsp. `alias ll="ls␣-la"`)

`umask` [⟨*3-stellige oktalnummer*⟩]
definiert die Zugriffsrechte für Dateien, die von der Shell und ihren Subprozessen angelegt werden.

`#!` ⟨*progname*⟩
in der 1. Zeile eines Shellskripts legt fest, welches Programm dieses Skript interpretieren soll ( z.B. `#!/bin/ksh`).

`function` ⟨*funktionsname*⟩ { *funktionskörper* } | ⟨*funktionsname*⟩ `()` { *funktionskörper* }
hiermit kann eine Shell-Funktion definiert werden. Besonderheiten im Umgang mit Funktionen sind:

- sie werden intern gespeichert
- Aliases werden beim Einlesen der Funktion aufgelöst
- die Ausführung erfolgt wie bei Kommandos mit positionalen Parametern

```
function findit {
  find . -name $@ -print 2>/dev/null
}
```

`return` [⟨*returncode*⟩]
hiermit wird eine Shell-Funktion verlassen und ein Returncode spezifiziert.

## 10.2 Kontrollstrukturen

- Verzweigung

`if` ⟨*bedingung*⟩
`then`
⟨*kommandoliste*⟩
[`else`]
[⟨*kommandoliste*⟩]
[`elif`] ⟨*bedingung*⟩
[⟨*kommandoliste*⟩]
`fi`

- Pre-Condition-Schleife

```
while ⟨bedingung⟩
do
   ⟨kommandoliste⟩
done
```

⟨*kommandoliste*⟩ bestehe hierbei aus einer Folge von Anweisungen, die wiederum selbst Kontrollstrukturen enthalten können.
⟨*bedingung*⟩ bezeichnet eine Befehlskombination, deren Exit-Code ($0 \rightarrow$ wahr, $\neq 0 \rightarrow$ falsch) die Schleifenbedingung darstellt.

- Fallunterscheidung

```
case ⟨variable⟩ in
     ⟨ match1 | match2 ⟩ ) ⟨kommandoliste⟩ ; ;  # oder
     ⟨match3⟩            ) ⟨kommandoliste⟩ ; ;  #
     *                   ) ⟨kommandoliste⟩ ; ;  # otherwise
esac
```

⟨*match*⟩ kann aus den bereits bekannten Wildcards `* ?  [...]` bestehen und bezieht sich hierbei auf Zeichenketten

- Feste Schleife

```
for ⟨variable⟩ [ in ⟨liste⟩]
do
   ⟨kommandoliste⟩
done
```

Wird die Option `in` ⟨*liste*⟩ weggelassen, so nimmt ⟨*variable*⟩ nacheinander alle an das Shellskript übergebenen Worte an. Im 2. Fall steht hinter dem Bezeichner `in` eine Wortliste, die ⟨*variable*⟩ dann durchläuft. ⟨*liste*⟩ bezeichnet eine Folge von Wörtern (z.B. `hallo alle aufgewacht`) oder z.B. `$*`.

Die reservierten Worte `if`, `while`, `done`, ... müssen einem Semikolon folgen oder in einer neuen Zeile beginnen.

| | |
|---|---|
| `break` [⟨*n*⟩] | bricht die inneren ⟨*n*⟩ Schleifen ab und setzt nach dem folgenden `done` fort |
| `continue` [⟨*n*⟩] | bricht den aktuellen Iterationsschritt ab und startet den nächsten Schleifendurchlauf |
| `true` | liefert immer wahr (Exit-Status 0) |
| `false` | liefert immer falsch (Exit-Status ungleich 0) |

## 10.3 Testkommandos

Als besonderes Kommando eignet sich das `test`-Kommando.

`test` ⟨*ausdruck*⟩ | [ ⟨*ausdruck*⟩ ]

prüft den Ausdruck auf seinen Wahrheitswert und liefert entsprechend dem Ergebnis einen Exitcode, der z.B. von `if` oder `while` ausgewertet werden kann.

Es stehen u.a. folgende Vergleichsoperatoren zur Auswahl:

| | |
|---|---|
| `-d` ⟨*name*⟩ | (**d**irectory) : Verzeichnis ⟨*name*⟩ existiert |
| `-f` ⟨*name*⟩ | (**f**ile) : Datei ⟨*name*⟩ existiert |
| `-s` ⟨*name*⟩ | (**s**ize) : Datei ⟨*name*⟩ existiert und ist nicht leer |
| `-r` ⟨*name*⟩ | (**r**ead) : ⟨*name*⟩ existiert und ist lesbar |
| `-w` ⟨*name*⟩ | (**w**rite) : ⟨*name*⟩ existiert und ist schreibbar |
| `-x` ⟨*name*⟩ | (e**x**ecutable) : ⟨*name*⟩ existiert und ist ausführbar |

Die Korn-Shell verfügt noch über weitere Testmöglichkeiten, die mit Hilfe von

[[ ⟨*ausdruck*⟩ ]]

ausgewertet werden:

- ⟨*string*⟩ : falls ⟨*string*⟩ nicht der Nullstring ist
- ⟨*string*⟩ [`!=|=`] ⟨*pattern*⟩ : prüft Matchen des Strings auf das Pattern.
- ⟨*string1*⟩ [`<|>`] ⟨*string2*⟩ : prüft kleiner/größer von Strings
- `-eq -ne -gt -ge -lt -le` für den Vergleich numerischer Werte

  *Wichtig ist, daß alle Argumente des* `test`*-Kommandos durch Leerzeichen voneinander getrennt sind !*

  *Numerische Werte werden auch als Zeichenketten verwaltet !*

Um umfangreichere Konstrukte zu ermöglichen gibt es noch eine AND, OR und NOT-Verknüpfung.

| | |
|---|---|
| `-a` | (AND) entspricht einer logischen UND-Verknüpfung |
| `-o` | (OR) entspricht einer logischen ODER-Verknüpfung |
| `!` | (NOT) entspricht einer logischen Verneinung |

Weiterhin gibt es noch folgende Konstrukte:

- ⟨*bedingung*⟩ `&&` ⟨*kommando*⟩ $\Longleftrightarrow$ `if` ⟨*bedingung*⟩ `then` ⟨*kommando*⟩
- ⟨*bedingung*⟩ `||`⟨*kommando*⟩ $\Longleftrightarrow$ `if !`⟨*bedingung*⟩ `then` ⟨*kommando*⟩

# 11 Die Startup–Files

Beim Start einer Shell werden einige Kommandos aufgeführt, bevor das Promptzeichen erscheint. Diese Kommandos finden sich in den folgenden Shellskripts:

`/etc/profile`
: Diese Datei wird bei jedem Aufruf einer Shell durchgeführt.

`$HOME/.profile`
: Diese Datei im Homeverzeichnis eines jeden Users wird direkt nach der `/etc/profile` ausgeführt. Dort kann jeder User eine eigenen Variablen oder Aliases definieren.

## 11.1 /etc/profile

Hier ein Beispiel für eine /etc/profile

```
# @(#)27        1.16  com/cfg/etc/profile, bos, bos320 2/4/91 15:11:29

trap "" 1 2 3
readonly LOGNAME

# Automatic logout, include in export line if uncommented
# TIMEOUT=120

# The MAILMSG will be printed by the shell every MAILCHECK seconds
# (default 600) if there is mail in the MAIL system mailbox.
MAIL=/usr/spool/mail/$LOGNAME
MAILMSG="[YOU HAVE NEW MAIL]"

# If termdef command returns terminal type (i.e. a non NULL value),
# set TERM to the returned value, else set TERM to default hft.
TERM_DEFAULT=hft
TERM=`termdef`
TERM=${TERM:-$TERM_DEFAULT}

export LOGNAME MAIL MAILMSG TERM

trap 1 2 3

##### Default-Pfad einstellen   (aus default-user .profile)

PATH=/bin:/usr/bin:/etc:/usr/ucb/:usr/bin/X11:
export PATH

# TimeZone
TZ=CET-1CEDT-2,J89/02:00,J271/02:00
export TZ

##### nur wenn Userid nicht root

if [ "$LOGNAME" != root ]
        then    :

                # ist Mail vorhanden ?          (aus default-user .profile)

                if [ -s "$MAIL" ]
                        then    echo "$MAILMSG"
                fi

                # lokale Erweiterungen aktivieren

                PATH=/usr/local/bin:/bin:/usr/bin:/etc:/usr/ucb:/usr/bin/X11:
                if [ "$LOGNAME" != "modem" ]; then
                        export PATH
                        # Display HRZ-motd  AL 25.3.92
                        if [ -r /usr/local/etc/motd ]
                                then    cat /usr/local/etc/motd
                        fi
                fi
fi
```

### 11.2 $HOME/.profile

Hier ein Beispiel für eine $HOME/.profile

```
#
# Hier koennen Befehle eingetragen werden, die von der Shell direkt nach
# login automatisch ausgefuehrt werden.
#

PATH=.:$HOME/bin:/usr/local/X11/bin:$PATH

PAGER=less

LESS="-csu"

EDITOR=/u/kehr/bin/mm

export CDPATH PATH PAGER LESS EDITOR


case "$SHELL" in
        /bin/ksh        )       export  ENV=$HOME/.environment.ksh
                                export  FCEDIT="mm"
                                export  HISTSIZE=999
                                ;;
esac


#echo $PATH


case "$TERM" in                         # look at TERM
        *\;*    )       eval "$TERM"    # if it contains a semicolon, then
esac                                    # execute the contents.


export TAPE=/dev/rmt0.1
export TEXINPUTS=.:/u/kehr/tex/styles:/usr/local/lib/tex/inputs/latex/sty
export MANPATH=.:/usr/local/beta/man:/usr/local/man:/usr/man:/u/kehr/man

alias ll="ls -la"
alias lll="ls -la | less"
alias lf="ls -aF"
alias cls="clear"
alias rd="rmdir"
alias h="history"
alias xlock="xlock -mode flame "

```

*Bitte erweitern Sie in Ihrer* `.profile` *den Pfad auf die hier angeführte Weise durch Anhängen von* `:$PATH` *um die Voreinstellungen zu erweitern und nicht zu überschreiben.*

# 12 Das Quoting

## 12.1 Einführung Quoting

Bestimmte Zeichen wie `< > ; ␣ & ( ) | \` haben für die Shell eine Sonderbedeutung. Man nennt sie *Metazeichen.* Ein Blank ist auch ein Metazeichen und hat die Funktion eines Trennzeichens. Will

man nun solche Zeichen in einem String unterbringen, so muß diese Sonderbedeutung unterdrückt werden. Diesen Vorgang nennt man *maskieren* oder *quoten*. Um z.B. ein Leerzeichen zu quoten gibt es folgende Möglichkeiten:

1. `var1=cd\ /usr/u1/bin`
2. `var2='cd /usr/u1/bin'`
3. `var3="cd /usr/u1/bin"`

Alle drei Variablen besitzen nun den gleichen Wert. Gibt man nun auf Kommandoebene `$var1` ein, so wird das Directory auf `/usr/u1/bin` eingestellt.

## 12.2 Die Quote-Zeichen

- \ (backslash)
  Der Backslash quotet das unmittelbar darauffolgende Zeichen (auch Blanks und Backslashes)
- ' (single quotes)
  `'...'` maskiert alle eingeschlossenen Zeichen (außer ' selbst)
- " (double quotes)
  "..." quotet alle eingeschlossenen Zeichen (außer `$ \ ` `)
  Parameterersetzungen sind also genauso erlaubt, wie Kommandosubstitierungen durch Backquotes. Dagegen werden enthaltene single quotes selbst gequotet.

### 12.2.1 Beispiele

Beispiele:

- ```
  $ A=$HOME  B=\$HOME  C='$HOME'  D="$HOME" ⏎
  $ echo $A  $B  $C  $D ⏎
  /usr/u1  $HOME  $HOME  /usr/u1
  ```

  Bei Variable B wurde mit dem Backslash das $ als Variablen-Präfix gequotet. Somit fand keine Variablenersetzung statt. Bei C wurde die komplette Zeichenkette mit den Single-Quotes maskiert; ebenfalls keine Variablenersetzung. Im Fall D fand eine Ersetzung innerhalb der Double-Quotes jedoch statt.

- ```
  $ Stern=* ⏎
  $ echo $Stern ⏎
  ```

  hat als Ergebnis

  ```
  datei1 datei2 ... datein
  ```

  Da der Stern eine Sonderbedeutung als Filename-Wildcard hat, fand eine Ersetzung mit allen Dateien des aktuellen Verzeichnisses statt. In einem solchen Fall muß die Sonderbedeutung maskiert werden.

  ```
  $ Stern="*"   oder   Stern=\* ⏎
  $ echo $Stern ⏎
  ```

  hat als Ergebnis immer noch

  ```
  datei1 datei2 ... datein
  ```

weil zwar die Variable `Stern` jetzt den richtigen Wert besitzt, aber bei der Ausführung des `echo`-Befehles nach der Variablenersetzung immer noch eine Filenamen-Expansion durchgeführt wurde.

```
$ echo "$Stern" ⏎
*
```

liefert nun endlich das gewünschte Ergebnis, da das Ergebnis der Variablen-Ersetzung selbst wieder gequotet wurde.

## 12.3 Die Backquotes

- ` (backquotes)
  Der Ausdruck `` ` ...⟨*ausdruck*⟩... ` `` wird ausgewertet. Ein `` h=` pwd ` `` liefert also das momentane Verzeichnis an die Variable `h`. Insbesondere gilt:

  1. backquotes `` ` ...⟨*variable*⟩... ` `` quoten ⟨*variable*⟩ nicht ! (⟨*variable*⟩ wird also von der Shell ersetzt)
  2. sie werden durch ein vorangestelltes \ und zwischen '...' gequotet, aber nicht zwischen "...".

### 12.3.1 Beispiele

- ```
  $ d=/usr/mike/bin ⏎
  $ m=` ls $d | wc -l ` ⏎
  $ echo $m ⏎
  3  (wenn dieses Verzeichnis 3 Dateien enthält)
  ```

- ```
  $ n="Das aktuelle Verzeichnis ist ` pwd ` ." ⏎
  $ echo $n ⏎
  Das aktuelle Verzeichnis ist /usr/u1.
  $ cd .. ⏎
  $ echo $n ⏎
  Das aktuelle Verzeichnis ist /usr/u1.
  ```
  hierbei wurde die Variable bereits vor dem `cd` expandiert und der Verzeichniswechsel hat auf die Ausgabe keine Wirkung.

- ```
  $ n="Das aktuelle Verzeichnis ist \` pwd\` ." ⏎
  $ eval echo $n ⏎
  Das aktuelle Verzeichnis ist /usr/u1.
  $ cd .. ⏎
  $ eval echo $n ⏎
  Das aktuelle Verzeichnis ist /usr/u1.
  ```
  Das `eval`-Kommando bewirkt ein völliges neuparsen des `echo`-Befehles. Dies bewirkt auch eine vollständige Kommandosubstitution des `pwd`-Kommandos.
  ```

# 13 Reguläre Ausdrücke

*Reguläre Ausdrücke* sind Muster zur Suche von Texten.
Sie können – ähnlich wie bei der Dateinamenersetzung – Metazeichen enthalten, welche spezielle Bedeutung haben. Es gibt viele Regeln und Möglichkeiten zur Bildung von regulären Ausdrücken. Bevor die Syntax zur Bildung von regulären Ausdrücken systematisch dargestellt wird, sollen hier erst einführende Beispiele gebracht werden, welche die Philosophie und die Ähnlichkeiten und Unterschiede zur Dateinamensersetzung deutlich machen.

So passen folgende Muster auf folgende Dateinamen:

| **Muster** | **Dateiname** |
|---|---|
| `ABCDE` | `ABCDE` |
| `AB?DE` | `ABADE, ABBDE, ABCDE, ...` |
| `AB*` | `AB, ABA, ABB, ABAA, ...` |

Folgende Muster passen auf folgende Texte:

| **Muster** | **Text** |
|---|---|
| `ABCDE` | `ABCDE` |
| `AB.DE` | `ABADE, ABBDE, ABCDE, ...` |

Das erste Muster `ABCDE` paßt auf den Text `ABCDE`, da jeder Text ohne gewisse Sonderzeichen auf sich selbst paßt. Das Muster `AB.DE` paßt auf die angegebenen Beispiele, weil der Punkt in regulären Ausdrücken eine ähnliche Bedeutung wie das Zeichen `?` in Dateinamen hat.
Im folgenden werden die wichtigsten Regeln für die Bildung und die Interpretation von *regulären Ausdrücken* aufgeführt:

## 13.1 Reguläre Ausdrücke für ein einzelnes Zeichen

- *Normales* Zeichen:
  Ein normales Zeichen paßt auf sich selbst.

  **Beispiele:**

| **Muster** | **Text** | **paßt ?** |
|---|---|---|
| `A` | `A` | ja |
| `B` | `b` | nein |
| `=` | `=` | ja |

  ◇ Unter einem normalen Zeichen werden hier alle am Bildschirm darstellbaren Zeichen verstanden mit folgenden Ausnahmen:

| | |
|---|---|
| `.` | Punkt |
| `*` | Mal |
| `[` | öffnende eckige Klammer |
| `\` | Backslash |

  Bei den folgenden Zeichen hängt es davon ab, wo sie in einem regulären Ausdruck stehen:

| | |
|---|---|
| `^` Zirkumflex | Wird als spezielles Zeichen angesehen, wenn es das erste Zeichen des gesamten regulären Ausdruckes ist, oder wenn es einer öffnenden Klammer in einem Ausdruck der Art `[ ... ]` unmittelbar folgt. |
| `$` Dollarzeichen | Wird als spezielles Zeichen angesehen, wenn es das allerletzte Zeichen eines gesamten regulären Ausdruckes ist. |

- Der Punkt
  Ein Punkt steht für ein beliebiges Zeichen außer `NEWLINE`

- Aufzählung
  Die Konstruktion `[ ...]` steht für irgendeines der darin enthaltenen Zeichen. Es müssen Zeichen innerhalb der Klammern stehen.

**Beispiele:**

| **Muster** | **Text** | **paßt ?** |
|---|---|---|
| `[abcde]` | `a` | ja |
| `[abcde]` | `A` | nein |

Es können auch Bereiche verwendet werden. Diese werden mit Hilfe des Minus-Zeichens gebildet. Ein Bereich

```
f-y
```

steht für alle Zeichen von f bis y. Das Minus-Zeichen hat nicht diese Bedeutung (d.h. es steht für sich selbst), wenn es am Ende der Zeichen innerhalb `[ ...]` steht oder am Anfang oder nach der Kombination `[^` (siehe unten).

**Beispiele:**

| **Muster** | **Text** | **paßt ?** |
|---|---|---|
| `[A-Z]` | `A` | ja |
| `[A-Z]` | bel. Großbuchst. | ja |
| `[A-Z]` | `b` | nein |
| `[A-Z]` | `-` | nein |
| `[A-H123i-z]` | `A` | ja |
| `[A-H123i-z]` | `I` | nein |
| `[A-H123i-z]` | `i` | ja |
| `[A-H123i-z]` | `2` | ja |
| `[A-H123i-z]` | `4` | nein |
| `[1-9-]` | `1` | ja |
| `[1-9-]` | `-` | ja |

Steht hinter der öffnenden Klammer das Zeichen `^`, so steht der Klammerausdruck für alle Zeichen mit Ausnahme der restlichen in der Klammer aufgegührten Zeichen und mit Ausnahme von `NEWLINE`.

**Beispiele:**

| Muster | Text | paßt ? |
|---|---|---|
| `[^aAbB]` | `a` | nein |
| `[^aAbB]` | `A` | nein |
| `[^aAbB]` | `c` | ja |
| `[^aAbB]` | `^` | ja |
| `[aAbB^]` | `a` | ja |
| `[aAbB^]` | `c` | nein |
| `[aAbB^]` | `^` | ja |
| `[^aAbB^]` | `^` | nein |

Soll das Zeichen `]`, welches einen solchen Klammerausdruck normalerweise beendet, in der Aufzählung der Zeichen vorkommen, so muß es das erste Zeichen nach `[` oder `[^` sein.

**Beispiele:**

| Muster | Text | paßt ? |
|---|---|---|
| `[]A-Z]` | `A` | ja |
| `[]A-Z]` | `]` | ja |
| `[^]A-Z]` | `A` | nein |
| `[^]A-Z]` | `]` | nein |

Die vier folgenden Zeichen haben innerhalb einer Klammer ihre normale Bedeutung:

`.  *  [  \`

**Beispiele:**

| Muster | Text | paßt ? |
|---|---|---|
| `[.,;]` | `.` | ja |
| `[.,;]` | `a` | nein |

## 13.2 Reguläre Ausdrücke für mehrere Zeichen

- Das Zeichen `*` :

  Wird ein regulärer Ausdruck für ein Zeichen von einem `*` gefolgt, so paßt dies auf ein beliebig häufiges Vorkommen des Zeichens; auch das *Nicht-Vorkommen* des Zeichens paßt.

  **Beispiele:**

  | Muster | Text | paßt ? |
  |---|---|---|
  | `a*` | `a` | ja |
  | `a*` | `aaaa` | ja |

- Aneinanderfügen von Ausdrücken für ein Zeichen

  Werden mehrere reguläre Ausdrücke für jeweils ein Zeichen hintereinandergeschrieben, so paßt der gesamte Ausdruck, wenn im Text die aufeinanderfolgenden Teile auf die einzelnen Teilausdrücke passen.

**Beispiele:**

| Muster | Text | paßt ? |
|---|---|---|
| `ABCDE` | `ABCDE` | ja |
| `ABCDE` | `ABCD` | nein |
| `abcde` | `AbCdE` | nein |
| `aa*` | `a` | ja |
| `aa*` | `aaaaaaa` | ja |
| `aa*` | `abc` | nein |

**Beispiele unter Verwendung von Sonderzeichen:**

| Muster | Text | paßt ? |
|---|---|---|
| `[a-zA-Z][0-9]` | `a1` | ja |
| `[a-zA-Z][0-9]` | `A1q` | nein |
| `[a-zA-Z][0-9]` | `AA` | nein |
| `[a-zA-Z][a-zA-Z0-9]*` | `X67b5` | ja |
| `[a-zA-Z][a-zA-Z0-9]*` | `767b5` | nein |
| `[a-zA-Z][a-zA-Z0-9]*` | `X67_b5` | nein |
| `[a-zA-Z][a-zA-Z0-9_]*` | `X67_b5` | ja |

## 13.3 Beginn und Ende einer Zeile

Steht das Zeichen `^` am Beginn des gesamten regulären Ausdruckes, so paßt dieser nur dann auf eine Zeile, wenn sich der dem Ausdruck ( ohne das `^` ) entsprechende Text am Zeilenanfang befindet.
Steht das Zeichen `$` am Ende des gesamten regulären Ausdruckes, so paßt dieser nur dann auf eine Eingabezeile, wenn sich der dem Ausdruck ( ohne das `$` ) entsprechende Text am Zeilenende befindet.

**Beispiele:**

| Muster | Text | paßt ? |
|---|---|---|
| `^[0-9].*` | `123 A=5` | ja |
| `^[0-9].*` | `77` | ja |
| `^[0-9].*` | `ABC=5` | nein |
| `.*&$` | `ls -l &` | ja |
| `.*&$` | `cat test` | nein |
| `^A.*Z$` | `A123Z` | ja |
| `^A.*Z$` | `A qwert asdfg Z` | ja |
| `^A.*Z$` | `% A qwert asdfg Z +` | nein |

# 14 Der Stream-Editor sed

**Name:**

`sed` – Das Programm **sed** ist ein nicht interaktiver Editor, der in einem *Stream-* bzw *Batch-Modus* betrieben wird. Die auszuführenden Editier-Anweisungen werden entweder aus einer Datei gelesen, oder sind Teil der Kommandozeile. Soll in mehreren Dateien die gleichen Änderungen durchgeführt werden, oder eine Vielzahl von Änderungen gemacht werden, so ist der **sed** ein geeignetes Hilfsmittel dafür.

Der **sed** arbeitet wie ein Filter. Er ließt von Standardeingabe und schreibt auf Standardausgabe.

## 14.1 Der Aufruf des sed

**Synopsis:**

`sed` [*-n*] [*-e* ⟨*Skript*⟩ | *-f* ⟨*Skript-Datei*⟩ ] [*datei*]...

**Beschreibung:**

Der **sed** bearbeitet die angegebenen Dateien oder – falls keine Datei angegeben wurde – die Daten der Standardeingabe und schreibt das Ergebnis auf die Standardausgabe. Die Editieranweisungen, d.h. die Angabe was mit den Eingabedaten geschehen soll, wird dem **sed**–Editor mit einem Art Programm, auch *sed–Skript* genannt, vorgegeben. Dieses Skript kann entweder beim Aufruf des **sed** als Parameter in der Form `-e skript` angegeben werden ( dann mit ' ... ' geklammert ) oder in einer *Skriptdatei* stehen ( zweite Form ). Beim Aufruf dürfen mehrere Skripts durch `-e` und `-f` (auch kombiniert) angegeben werden.

In *Skript* oder *Skript-Datei* steht jeweils eine **sed** – Anweisung pro Skriptzeile.

Die Abarbeitung geschieht in der Art, daß **sed** die erste Zeile der Eingabe in den Eingabepuffer liest, prüft welche Anweisung des Skript auf dieser Zeile ausgeführt werden sollen, diese Anweisungen nacheinander ausführt und das Ergebnis auf die Standardausgabe schreibt. Danach liest **sed** die nächste Zeile und wiederholt diesen Vorgang. Das bedeutet, daß bei einem Einfügen einer Zeile, die eingefügte Zeile ebenfalls bearbeitet wird. Weiterhin aendert sich die Zeilennummerierung während der Abarbeitung der Befehle bei einem Einfügen oder Löschen einer Zeile.

Die Option `-n` unterdrückt die automatische Ausgabe auf Standardausgabe. In diesem Fall werden nur Ausgaben mittels den Druckanweisungen **print** und **write** produziert. Die Texte können auch in einen temporären Puffer abgelegt und bei Bedarf ausgegeben werden. Dieser Puffer wird als *Haltepuffer* bezeichnet.

Bei einer Bearbeitung durch den **sed** wird die Eingabedatei **nicht** verändert.

## 14.2 Die Anweisungen des sed

Die Anweisungen an den **sed** im Parameter *Skript* oder *Skript-Datei* haben folgendes Format:

[ *Adresse* [, *Adresse* ] ] `Funktion` [*Argumente*]

`Funktion` ist der Befehl oder die Anweisung, welche auf die durch *Adresse* angegebenen Zeile angewendet werden soll. Die Zeilennummerierung beginnt bei 1. Bei zwei Adressen handelt es sich um einen Bereich `von, bis` wobei die beiden angegebenen Zeilen mit einbezogen werden.
Ohne Angabe einer Adresse wird die ganze Eingabedatei bearbeitet. Das Dollarzeichen `$` als Adresse bezeichnet die letzte Eingabezeile. Ein Textmuster der Form `/text/` als Adresse, bedeutet *von der ersten Zeile der Eingabe auf die das Textmuster paßt.* Als Textmuster sind Buchstaben, reguläre

Ausdrücke und zusätzlich noch `\n` als **`NEWLINE`** - Symbol erlaubt. Eine Gruppierung von Zeichenketten wird durch \( ... \) erreicht. Als Ersetzungsmuster gelten folgende Metazeichen:

**Funktionen im Ersetzungsmuster**

| | |
|---|---|
| n-ter Teilausdruck | `\n` |
| gefundene Zeichenkette | `&` |

Will man `Funktion` auf alle Zeichen ausführen, auf die das Muster bzw. der Zeilenbereich **nicht** paßt, so wird dies durch ein ! - Zeichen vor der Funktion erreicht. ( z.Bsp. `/^[0-9]/ !d` löscht alle Zeilen, die **nicht** mit einer Ziffer beginnen. )
Folgende **sed**-Kommandos stehen für **sed**-Skripten zur Verfügung:

`a\`
*text* — Der nachfolgende Text wird in die Ausgabe nach der aktuellen Zeile geschrieben und erst danach werden weitere Eingaben verarbeitet. Der einzufügende Text beginnt im Skript erst auf der nächsten Zeile und endet mit einer Zeile, die nicht mit \ aufhört. Der \ wird nicht in die Ausgabe kopiert.

`b` *marke* — Es wird zu der Marke ( in der Form: `:marke` ) des Skripts gesprungen und dort die Abarbeitung des Skripts fortgesetzt. Fehlt die Angabe der Marke, so wird an das Ende des Skriptes gesprungen.

`c\`
*text* — Der Text des angegebenen Bereichs wird durch den neuen Text *text* ersetzt. Der Ersetzungstext beginnt in der folgenden Zeile. Erstreckt er sich über mehrere Zeilen, so wird mit einem \ am Ende der Zeile seine Fortsetzung angezeigt. Der \ wird nicht in die Ausgabe kopiert.

`d` — Der Text des angegebenen Bereichs wird gelöscht und die nächste Eingabezeile gelesen.

`D` — Der erste Teil des angegebenen Bereichs bis zum ersten Zeilenende wird gelöscht.

`g` — Der Text des angegebenen Bereiches ( des Eingabepuffers ) wird durch den Inhalt des Haltepuffers ersetzt.

`G` — Der Inhalt des Haltepuffers wird am Ende des vorgegebenen Bereichs angefügt.

`h` — Der Inhalt des Haltepuffers wird durch den Text des Bereichs ersetzt. Der alte Inhalt des Haltepuffer geht verloren.

`H` — Der Text des gewählten Bereichs wird am Ende des Haltepuffers angehängt.

`i\`
*text* — Der Text wird in die Ausgabe vor der Ausgabe der angegebenen Zeile geschrieben. Der Text beginnt in der nächsten Zeile und endet mit einer Zeile ohne ein \ am Ende. Der \ wird nicht in die Ausgabe kopiert.

`l` — Der Text des angegebenen Bereichs wird auf die Ausgabe geschrieben, wobei nicht druckbare Zeichen als 2- oder 3-Zeichen-ASCII-Zeichen in der oktaler Darstellung \ooo ausgegeben und überlange Zeilen in mehrere einzelne Zeilen unterteilt werden.

`n` — Der Text des Bereichs wird ohne eine Änderung in die Ausgabe kopiert und es wird die nächste Eingabezeile gelesen.

`N` — Die nächste Zeile der Eingabe wird an den Eingabepuffer angehängt ( die Eingabezeile wird um eins weitergezählt ).

`p` — Der Text des Bereichs bzw. des Eingabepuffers wird auf die Ausgabe geschrieben.

`P` — Der erste Teil des Bereichs bzw. des Eingabepuffers bis zum ersten Zeilenende wird auf die Ausgabe geschrieben.

`q` — Es wird die aktuelle Zeile ausgegeben, zum Ende des Skripts gesprungen und die Bearbeitung des **sed** beendet.

`r` *datei* — Die angegebene Datei wird gelesen und ihr Inhalt ohne eine weitere Bearbeitung auf die Ausgabe kopiert. Erst danach wird die nächste Eingabe gelesen und verarbeitet. Zwischen `r` und dem Dateinamen muß genau ein Leerzeichen stehen !

`s`*/muster/text/modus*

In dem angegebenen Bereich sollten Textstücke, auf die das Textmuster *muster* (regulärer Ausdruck) paßt, durch *text* ersetzt werden. Die Klammerung von *muster* und *text* braucht nicht durch das Zeichen / erfolgen, sondern kann auch durch jedes andere Zeichen geschehen.
Z.Bsp.: `s#Unix#UNIX#g` → ersetzt alle *Unix* durch *UNIX*.
*modus* gibt dabei an, wie dies geschehen soll. *modus* darf folgende Werte haben:

| | |
|---|---|
| *n* | Es wird nur das *n*-te passende Textstück ersetzt ($1 \leq n \leq 255$) |
| `g` | Es wird nicht nur das erste, sondern **alle** passenden ( sich nicht überlappenden ) Textstücke des Eingabepuffers ersetzt. |
| `p` | Sofern eine Ersetzung stattfindet, wird der Text des Bereichs ( der neue Text ) ausgedruckt. |
| `w` *datei* | Sofern eine Ersetzung stattgefunden hat, wird der Text des Bereichs ans Ende der angegebenen Datei geschrieben. Alle Dateien, welche im Ablauf eines `sed`-Skriptes aufgerufen werden, werden vor der ersten Verarbeitung angelegt. Zwischen dem `w` und dem Dateinamen muß **genau ein Leerzeichen** stehen ! |

| | |
|---|---|
| `t` *marke* | Wurde in der aktuellen Zeile eine Ersetzung vorgenommen, so wird (wie bei `b` ) zur angegebenen Marke gesprungen. Fehlt *marke*, so wird zum Ende des Skripts gesprungen. |
| `w` *datei* | Der Text des Bereichs wird ans Ende der angegebenen Datei geschrieben. ( Besonderheiten zum `w`-Kommando siehe unter dem `s`-Kommando) |
| `x` | Der Text im Eingabepuffer wird mit dem Text im Haltepuffer vertauscht. |
| `y`*/t1/t2/* | Es werden im Text des Bereichs alle Zeichen, die in der Zeichenfolge *t1* vorkommen, durch die entsprechenden Zeichen der Zeichenfolge *t2* ersetzt. *t1* und *t2* müssen gleich lang und dürfen kein regulärer Ausdruck sein ! |
| `!`*funktion* | Die angegebene Funktion wird auf jene Zeilen ausgeführt, auf die der angegebene Bereich nicht zutrifft. |
| `:`*marke* | Definiert eine Sprungmarke für die Funktionen `b` und `t`. |
| `=` | Die aktuelle Zeilennummer wird als eigene Zeile auf die Ausgabe geschrieben. |
| { ...} | Klammert eine Gruppe von Funktionen. Die einzelnen Funktionen werden jeweils durch ⏎ syntaktisch getrennt. Alle diese Funktionen werden für den angegebenen Bereich ausgeführt. |
| `#` *text* | Diese Zeile wird als Kommentarzeile betrachtet. `#` muß das erste Zeichen der Zeile sein ! |

**Beispiele**

- `$ sed '/^[ ]*$/d' alt > neu` ⏎
  → bearbeitet die Datei *alt* und schreibt als Ergebnis nach *neu*. Es werden alle Leerzeilen ( Zeilen ohne ein Zeichen oder nur mit Leerzeichen ) in der Eingabe gelöscht.

- `$ sed 'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/'` ⏎
  → arbeitet als Filter und ersetzt alle Kleinbuchstaben des Eingabetextes durch Großbuchstaben.
  ♢ bei dem `y`-Kommando dürfen keine regulären Ausdrücke benutzt werden, deshalb ist die Kurzschreibweise `y/[a-z]/[A-Z]/` **nicht** möglich !

- `$ sed -n -e "1,20 w"\`
  `-e "30,40 w≫ neu < alt` ⏎
  → Gibt nur die Zeilen 1 bis 20 und 30 bis 40 aus. Die Ausgabe der restlichen Zeilen wird durch die Option `-n` unterdrückt. **sed** liest aus der Datei `alt` und schreibt in die Datei `neu`.

# 15 Der Reportgenerator awk

**Name:**

`awk` – Mit **awk** können Dateien ähnlich wie mit **sed** bearbeitet werden. **awk** zerlegt die Zeilen automatisch in Zeichenketten, bzw. *Felder*. **awk** ermöglicht eine Bearbeitung von ASCII-Dateien, deren Modifizierung, das Rechnen mit *Feldern* (wenn es sich um Zahlen handelt), sowie die formatierte Ausgabe. Aus diesem Grund spricht man bei **awk** von einem Reportgenerator

## 15.1 Aufruf von awk

**Synopsis:**

`awk` [*-Fz*] ⟨ *awk-programm* | *-f awk-skript* ⟩ [ *parameter* | *datei* ... ]

**Beschreibung:**

Das Programm **awk** bearbeitet die Eingabedatei, bzw. falls keine Datei ( oder – ) angegeben wurde, die Standardeingabe. Die Bearbeitungsschritte werden in Form eines Programms (auch *awk-Skript* genannt) angegeben. Dieses Programm kann auf zwei Arten vorgegeben werden. Entweder wird es als Parameter *awk-programm* beim Aufruf des **awk** spezifiziert und ist dann in der Regel mit ' ... ' geklammert oder es steht in einer Datei und wird dann mit `-f` *awk-skript* angegeben. *parameter* sind Namen von **awk**-internen Variablen und deren Werte in der Form *variable=wert*. Mit `-Fz` kann das Zeichen `z` als Feldtrennzeichen vorgegeben werden.

Das Ergebnis der Bearbeitung wird auf die Standardausgabe geschrieben.

## 15.2 Das awk-Programm (Skript)

Ein **awk**-Programm besteht aus drei Teilen:

- der Initialisierung
- der Verarbeitung
- dem Abschluß.

Die Initialisierung und der Abschluß sind optional.

### Grundstruktur eines awk-Programmes

```
BEGIN   {
        Preblock
        }

⟨Pattern⟩ { ⟨kommandoliste⟩
        }

⟨Pattern⟩ { ⟨kommandoliste⟩
        }

        { ⟨kommandoliste⟩
        }

END     {
        Postblock
        }
```

Die mit Preblock bezeichneten Aktionen werden immer vor der Bearbeitung der ersten Eingabezeile ausgeführt, die mit Postblock bezeichneten Aktionen werden immer nach der letzten Eingabezeile ausgeführt, selbst dann, wenn die Bearbeitung durch die `exit`-Anweisung vorzeitig beendet wird.
Der Allgemeine Aufbau des Hauptblockes sieht folgendermaßen aus:

⟨*Pattern*⟩ { ⟨*kommandoliste*⟩}

`awk` ließt eine Zeile und prüft, ob *Pattern* auf diese Zeile zutrifft. Ist dies der Fall, so werden die durch ⟨*kommandoliste*⟩ bezeichneten Aktionen ausgeführt.Die nächste Eingabezeile wird erst gelesen, sobald alle angegebenen Muster geprüft worden sind. Fehlt die Angabe von ⟨*Pattern*⟩, so wird die jede Eingabezeile durch die entsprechende ⟨*kommandoliste*⟩ bearbeitet. Fehlt ⟨*kommandoliste*⟩, so wird die betreffende Eingabezeile ohne Bearbeitung auf Standardausgabe geschrieben.

⟨*Pattern*⟩ kann sein:

- ein regulärer Ausdruck in der Form: /textmuster/
- ein Auswahlbereich in der Form:
  /textmuster1/,/textmuster2/
- ein relationaler Ausdruck,
- eine Verknüpfung der obigen Möglichkeiten.
- Außer den normalen regulären Audrücken sind noch folgende Metazeichen erlaubt.

| | |
|---|---|
| `?` | keine oder eine Wiederholung |
| `+` | eine oder mehrere Wiederholungen |
| a \| b | a oder b, wobei a und b reguläre Ausdrücke sind |

Im einfachsten Fall handelt es sich um einen Text oder ein Textmuster in der Form */muster/*.

**Beispiele:**

- `$ awk '/Unix/' info` ↵
  → gibt alle Zeilen der Datei *info* aus, in denen *Unix* vorkommt.
- `$ awk '/Anfang/,/Ende/' info` ↵
  → sucht in der Eingabe nach der ersten Zeile, welche das Textstück *Anfang* enthält. Diese und alle folgenden Zeilen bis zur Zeile, die das Textmuster *Ende* enthält werden ausgegeben.
- `$ awk '/^[0-9]+/ {print $1}' info` ↵
  → gibt das erste Wort aller Zeilen der Datei *info* aus, die mit einer Ziffernfolge beginnen.
- `$ awk '/^[0-9]+|^#/' info` ↵
  → gibt alle Zeilen der Datei *info* aus, die entweder mit mindestens einer Ziffer oder dem Zeichen # beginnen.
- `$ awk 'lenght() > 50 { print }' info` ↵
  → es werden nur die Zeilen ausgegeben, die länger als 50 Zeichen sind.

## 15.3 awk-Sprachelemente

Wie die meisten anderen Programmiersprachen, kennt **awk** Konstanten, Variablen, Ausdrücke, Funktionen und Anweisungen.
Bei den Konstanten und Variablen wird der Wert abhängig von der Verwendung als Zeichenkette oder als numerischer Wert interpretiert.

### 15.3.1 awk-Konstanten

**Numerische Konstanten** sind entweder ganze Zahlen (z.Bsp. 127) oder Gleitpunktzahlen, wobei folgende Formen erlaubt sind: `1.2, .3, 23e2, 3.4e-3, 34e+4, 12E15, 12.3E-2`. Diesen darf jeweils ein + oder ein – vorangestellt sein.
**Textkonstanten** werden durch " ... " geklammert. "" stellt dabei die leere Zeichenkette dar. Soll das Zeichen " selbst im Text vorkommen, so muß dies durch \" angegeben werden;
`n` steht für `NEWLINE` und
`t` für `TABULATOR` und \\ für das Zeichen\ selbst.

### 15.3.2 awk-Variablen

Variablen haben einen Bezeichner (Namen), der mit einem Buchstaben beginnt. Ohne daß man die Variablen explitzit definiert, werden sie bei der ersten Verwendung im **awk**-Programm oder durch ihre Defintion beim Aufruf von **awk** ( in der Form *variable=wert* ) angelegt. Der initiale Wert einer **awk**-Variablen ist die leere Zeichenkette. Es sind auch Felder bei Variablen möglich, wobei dann, wie in C, das Feldelement mit *variable*`[`*index*`]` angegeben wird. Im Gegensatz zu C, darf beim **awk** der Index jedoch nicht nur ein numerischer Ausdruck sein, sondern er darf auch aus einer Zeichenkette bestehen, womit man einen *Namen* als Index verwenden kann.
Neben den vom Benutzer vorgegebenen Variablen und den Feld-Variablen `$0, $1, ..., $`*n* kennt der **awk** bereits eine Reihe von Variablen mit fester Bedeutung. Diese werden jedoch im Gegensatz zur Shell **nicht** mit vorangestelltem $ benutzt !

| | |
|---|---|
| `ARGC` | Gibt die Anzahl der Argumente der Kommandozeile an |
| `ARGV` | Ist ein Feld, dessen Elemente die Argumente der Kommandozeile enthalten. |
| `FILENAME` | Ist der Name der aktuellen Eingabedatei |
| `FNR` | Gibt die Nummer der Zeile (des Satzes) der aktuellen Eingabedatei an. |
| `FS` | (*input field seperator*) Gibt das oder die Trennzeichen für Felder der Eingabe an. Der Standardwert sind Leerzeichen und Tabulatorzeichen (auch *white spaces* gennant). Mit `FS="`*z*`"` wird *z* zum neuen Trennzeichen zwischen Feldern. Standardwerte sind Leer- und Tabulatorzeichen. |
| `NF` | Gibt an, wieviel (jeweils durch das Trennzeichen `FS` unterteilte) Felder in der Eingabezeile vorhanden waren. |
| `NR` | Gibt die Nummer der aktuellen Zeile (bzw. des aktuellen Satzes) der Eingabe an. |
| `OFMT` | Gibt das Standardformat von Zahlenwerten in der Ausgabe an. (Standard in C-Syntax: `%.6g`). |
| `OFS` | Gibt an, welche Trennzeichen jeweils zwischen zwei Feldern in der Ausgabe stehen sollen. Der Standardwert ist das *Leerzeichen*. |
| `ORS` | Gibt an, mit welchem Zeichen (bzw. mit welchen Zeichen) ein Satz in der Ausgabe abgeschlossen sein soll. Der Standardwert hierfür ist `NEWLINE`. Ein Satz ist damit bei der Ausgabe äquivalent zu einer Zeile. |
| `RS` | Gibt an, welche Zeichen in der Eingabe als Satztrennzeichen betrachtet werden sollen (Standard: `NEWLINE`). |

### 15.3.3 awk-Ausdrücke

In numerischen Ausdrücken im **awk** dürfen die Operatoren `+, -, *, /, %` für Modulo , ( ... ) zur Gruppierung sowie numerische Funktionen vorkommen.

In Textausdrücken wird der Operator `Leerzeichen` als Zeichen für eine Textkonkationation verwendet.
In *logischen Ausdrücken* dürfen

- die Vergleichsoperatoren `<, <=, ==, !=, >=, >` vorkommen,
- die logischen Verknüpfungen

  `&&` ( für **UND**, d.h. beide Operanden müssen wahr sein),
  `||` ( für **ODER**, d.h. einer der beiden Operanden muß wahr sein ),
  `!` ( für die **Negation** ) sowie die Operatoren,
  `~` ( für *ist enthalten in* ) und
  `!~` ( für *ist nicht enthalten in* ).

Ein Ausdruck wird abhängig von seinen Elementen als ein numerischer Ausdruck oder als Operationen mit Zeichenketten betrachtet. Will man erreichen, daß ein Ausdruck als numerischer Wert interpretiert wird, so kann man das durch `+0` im Ausdruck erzielen; soll er als Textausdruck interpretiert werden, so geschieht dies durch Textkonkatenation mit `""`. In einem numerischen Ausdruck wird ein Text als `0` behandelt.

### 15.3.4 awk-Aktionen

Eine Aktion in der Syntax

⟨*Pattern*⟩ { [*aktion*] }

kann aus keiner, aus einer oder aus mehreren Anweisungen bestehen. Die Anweisungen haben eine Syntax ähnlich der von C. Eine Anweisung wird durch ein Semikolon `;`, durch `NEWLINE` oder durch eine schließende Klammer } beendet. Es kann sich hierbei um eine Zuweisung der Form `variable=`*ausdruck* handeln. Der *ausdruck* wird ausgewertet, und das Ergebnis der Variablen zugewiesen. Neben dem Zuweisungsoperator = sind auch folgende von C bekannten Operatoren erlaubt:
`+=, -=, *=, /=, %=, ++, --`

Weiterhin existieren folgende Anweisungen und Kontrollstrukturen:

**if** *( bedingung ) anweisung1* [**else** *anweisung2*]
: Ist die Bedingung erfüllt, so wird *anweisung1* ausgeführt; ist sie nicht erfüllt, so wird - sofern angegeben - der **else**-Teil ausgeführt.

**while** *( bedingung ) anweisung*
: Die Bedingung wird ausgewertet, und falls sie erfüllt ist, die angegebene *anweisung* ausgeführt, bis die Bedingung nicht mehr erfüllt ist.

**for** *(ausdruck1; bedingung; ausdruck2) anweisung*

*ausdruck1* wird ausgewertet, danach die angegebene Bedinung überprüft. Ist diese erfüllt, so wird *ausdruck2* ausgewertet. Dies wird solange wiederholt, bis die Bedingung nicht mehr erfüllt ist.

**for** *( variable* **in** *feld )*

Die *Variable* nimmt nacheinander die Werte der einzelnen Feldelementen an. Die Anweisung wird solange durchlaufen, wie *feld* Elemente besitzt.

**break**

Die **break**-Anweisung beendet die Abarbeitung einer **for**- oder **while**-Schleife. Die Abarbeitung wird hinter der Schleife fortgesetzt.

**continue**

Es wird an das Ende einer **for**- oder **while**-Schleife gesprungen und erneut ein Durchlauf gestartet.

**print** [*ausdruckliste*] [*umlenkung*]

**printf** *format* [*ausdruckliste*] [*umlenkung*]

Die Ausdrücke der Liste werden ausgewertet und ausgegeben. **printf** kann auch in C-Syntax geschrieben werden. Die **format**-Angabe wird ebenfalls in C-Syntax erwartet (siehe C-Manual).

Die *umlenkung* erfolgt wie bei der normalen Standardumlenkung der Shell mit > oder >>.

{ *anweisung ...* }

Sollen dort, wo die obige Syntax nur eine Anweisung vorschreibt, mehrere Anweisungen erfolgen, so müssen diese Anweisungen wie bei C durch { ... } geklammert werden.

**next**

Die noch verbleibenden *Pattern* des Skriptes werden übersprungen und die nächste Eingabezeile wird gelesen.

**exit** [ *ausdruck* ]

Durch diese Anweisung wird der Rest der Eingabedatei übersprungen, der **END**-Teil abgearbeitet und der Exit-Status erhält den Wert von *ausdruck*.

**return** [ *ausdruck* ]

Rücksprung aus einer Funktion mit dem Ergebniswert *ausdruck*

**#** *kommentar*

Das Zeichen **#** leitet einen Kommentar ein. Dieser erstreckt sich bis zum Ende der Zeile.

### 15.3.5 Die Funktionen des awk

Neben den bereits aufgezählten Programmkonstrukten besitzt **awk** eine Reihe von internen Funktionen (ohne nähere Erklärung):

**Numerische Funktionen:** atan2( x ), cos( x ), exp( x ), int( x ), log( x ),rand( x ), sin( x ), sqrt( x ), srand( x ), length( *ausdruck* ).
(length ohne Argument liefert die Länge der aktuellen Eingabezeile).

**Funktionen auf Zeichenketten:** gsub( *ra, neu, text* ), index( *t1, t2* ), match(*t, ra* ), split( *ausdruck, bezeichner* [, *trennzeichen* ] ), sprintf( *format, ausdruck, ...*), sub( *ra, neu* ), substr( *zk, m* [, *n*]).

**Ein-/Ausgabefunktionen und generelle Funktionen:** close( *datei* ), getline, system( *kommando* ).

## 15.4 Übergabe von Shellparametern

Im **awk**-Skript kann die Übergabe der Shellvariablen nur über einen Trick durch geschicktes Quoting erreicht werden, und zwar:

z.Bsp: `$ awk 'BEGIN { laenge = '$SHELLVARIABLE' } ... '` ⏎

Dieser Aufruf nutzt den Umstand, daß $SHELLVARIABLE nicht innerhalb einer Quotierung steht und dadurch durch die Shell ersetzt wird. Dieser Umstand kann auch in einer Skript-Datei verwendet werden.

Beim Aufruf läßt sich dieses auch durch die direkte Angabe der Variablenbelegung wie folgt erreichen:

z.Bsp: `$ awk 'lenght() > laenge { print ... } ' laenge=30` ⏎ Dieses Beispiel kann auch wie folgt aussehen:

`$ awk 'BEGIN { laenge=ARGV[1] }' 30` ⏎

# 16 Weitere Grundbefehle

## 16.1 date - Ausgabe des Systemdatums

**Name:**

`date` – Gibt das aktuelle Systemdatum, sowie die Systemzeit aus.

**Synopsis:**

`date`

## 16.2 find - Suchen von bestimmten Dateien

**Name:**

`find` – Sucht rekursiv nach einer Datei ab dem Verzeichniss *pfadname*, die dem *Ausdruck* entspricht.

**Synopsis:**

`find` ⟨*pfadname*⟩ [*Ausdruck*]

Mögliche Werte für *Ausdruck*

| | |
|---|---|
| `-name` *filename* | Berücksichtigt alle Dateien, die den Dateinamen *filename* haben, oder auf ein Muster welches in *filename* angegeben ist passen. Wird ein Muster, also ein *filename* mit Wildcards angegeben, so muß dieses Muster maskiert, also in Hochkommas eingefaßt sein. |
| `-user` *username* | Es werden alle Dateien berücksichtigt, die dem Benutzer *username* gehören, und auf die der Aufrufende Zugriff hat. |
| `-print` | Diese Aussage ist immer wahr. Alle Dateien die auf die anderen Muster zutreffen werden ausgegeben. |
| `-ls` | Erzeugt eine Ausgabe, die in etwa dem Aufruf `ls -lagi` entspricht. |

## 16.3 grep - Suchprogramm

**Name:**

`grep` – Die `grep`–Programme durchsuchen die angegebenen Dateien nach dem angegebenen *muster*. Die Zeilen der Dateien, in denen das *muster* gefunden wird, werden auf Standardausgabe ausgegeben. Wird mehr als eine Datei angegeben, so wird der Dateiname mit ausgegeben.

Bei `fgrep` werden die Metazeichen nicht umgesetzt. In `egrep` kann der Ausdruck sich auch aus Metazeichen (regulären Ausdrücken) zusammensetzen.

**Synopsis:**

`grep` [*-bcilnsv*] ⟨*muster*⟩ [*datei* ...]

`fgrep` [*-bcefilnvx*] ⟨*muster*⟩ [*datei* ...]

`egrep` [*-bcefhalnvxy*] ⟨*muster*⟩ [*datei* ...]

Zusätzliche reguläre Ausdrücke für `egrep`:

⟨*ausd1*⟩ | ⟨*ausd2*⟩ entweder ausd1 oder ausd2 trifft zu.

⟨*ausd1*⟩ ⟨*ausd2*⟩ ausd1 und ausd2 treffen zu.

**Optionen:**

| | |
|---|---|
| `-c` | Die Anzahl der Zeilen, auf die das *muster* zutrifft werden gezählt. |
| `-e` *ausdruck* | Falls *ausdruck* mit einem '-' beginnt. *ausdruck* darf mit oder ohne Zwischenraum auf `-e` folgen. |
| `-f` *datei* | Der Ausdruck, nach dem gesucht werden soll, steht in der angegebenen *datei*. |
| `-h` | Der Dateiname wird in der Ausgabezeile weggelassen. |
| `-i` | Groß- und Kleinbuchstaben gleichbehandeln |
| `-n` | Zeilennummern mit ausgeben |
| `-s` | Ausgabe unterdrücken, nur Status zurückliefern.<br>Rückgabewerte: 0 - gefunden,<br>1 - nicht gefunden,<br>2 - Syntaxfehler |
| `-v` | Alle Zeilen ausgeben, auf die das *muster* nicht zutrifft. |

## 16.4 head, tail - Ausgabe von Dateien

**Name:**

`head tail` – Gibt die ersten ( `head` ), bzw. letzten ( `tail` ) Zeilen der angegebenen Dateien aus.

**Synopsis:**

`head` [*-n number*] ⟨*datei*⟩...

`tail` [*-n number -f*] ⟨*datei*⟩...

**Optionen:**

| | |
|---|---|
| `-n` *number* | Mit dieser Option läßt sich die Standardeinstellung von 10 Zeilen verändern. -n 20 wären dann z.Bsp. 20 Zeilen. |
| `-f` | ( Nur `tail` ) Der Befehl terminiert nicht, sondern merkt sich das Dateiende, und überprüft in gewissen Zeitintervallen, ob sich die Datei vergrößert hat, und zeigt den nun eventuell vorhandenen neuen Text mit an. |

## 16.5 ln - Verweis auf eine Datei

**Name:**

`ln` – Dieser Befehl ermöglicht es, eine Datei unter mehreren Namen ansprechbar zu machen. Die Datei existiert nur einmal auf dem Rechner.
`datei1` bezeichnet die Originaldatei, auf die verwiesen wird, und `datei2` bezeichnet den Namen ( bzw. Pfad und Namen ) unter dem die `datei1` auch ansprechbar ist.

**Synopsis:**

`ln` [*-fs*] ⟨*datei1*⟩ ⟨*datei2*⟩

**Optionen:**

| | |
|---|---|
| `-f` | Ist `datei2` eine existierende Datei mit eingeschränkter Schreibberechtigung, so wird eine Nachfrage ob der Link erzeugt werden soll unterdrückt. |
| `-s` | Symbolischer Link. Ein Verweis auf ein Verzeichniss, sowie ein Verweis auf eine Datei in einem anderen Device ist nur mit dieser Option möglich. |

## 16.6 lpr - Druckausgabe von Dateien

**Name:**

`lpr` – Die angegebene ⟨*datei*⟩ wird auf den Drucker, bzw. in die Warteschlange des Standarddruckers übertragen.

**Synopsis:**

`lpr` [*-P queue*] [*-r*] ⟨*datei*⟩ ...

**Optionen:**

`-P` Gibt die gewünschte Warteschlange und somit Verarbeitungsweise an, wie zum PCL, Postscript o. ä. Die Konfiguration ist lokal unterschiedlich.

`-r` Löscht die Datei, sobald eine Kopie davon in der Warteschlange abgelegt ist.

## 16.7 man - Ausgabe einer Befehlsbeschreibung

**Name:**

`man` – gibt die Befehlsbeschreibung für das `kommando` formatiert aus. Die Ausgabe erfolgt meistens mit pg oder more.
In diesem Zusammenhang ist auch der Befehl `apropos` aufzuführen. `apropos` sucht nach Abschnitten in den Manuals, in deren Titel die angegebenen *schlüsselwörter* vorkommen.

**Synopsis:**

`man` ⟨*kommando*⟩

`apropos` ⟨*schlüsselwort*⟩...

## 16.8 pack, unpack - Komprimieren von Dateien

**Name:**

`pack unpack` – Die angegebenen Dateien werden komprimiert und mit der Endung **.z** versehen. Eine Komprimierung erfolgt nicht, falls Verweise auf diese Datei existieren, kein Platzgewinn erzielt wird, die Datei nicht gelesen werden kann, oder ein Fehler auftritt. unpack dekomprimiert ein **.z**-Datei wieder, und entfernt die **.z**-Endung.

**Synopsis:**

`pack` [- *-f*] ⟨*datei*⟩ ...

`unpack` ⟨*datei*⟩...

**Optionen:**

`-` Zeichenhäufigkeit einzelner Zeichen und deren Kodierung wird ausgegeben.

`-f` Die `datei` wird selbst dann komprimiert, wenn auch kein Platzgewinn erzielt wird.

## 16.9 sort - Sortieren von Ausgaben

**Name:**

`sort` – sortiert die Zeilen der Eingabe und gibt diese auf die Standardausgabe aus.

**Synopsis:**

`sort` [*-bcdfimMnru -o ausgabedatei -t trennzeichen* ] [*+skip1* [*-skip2*]] [*datei* . . . ]

**Optionen:**

| | |
|---|---|
| `+skip1 -skip2` | Das Sortierkriterium sind die Felder, die `pos1` vom Zeilenanfang (Feld 0) entfernt sind. Die Felder ab `pos2` werden ignoriert. |
| `-c` | Eine Ausgabe der Zeilen erfolgt nur, falls die Eingabe bereits wie angegeben vorsortiert ist. |
| `-d` | Sonderzeichen werden nicht beachtet. |
| `-n` | Ausgabe wird nach Nummern sortiert. Beginnen die Zeilen nicht mit Nummern, so wird alphabetisch sortiert. |
| `-r` | Die Sortierung erfolgt in umgekehrter Reihenfolge. |
| `-o` *ausgabedatei* | Die Ausgabe erfolgt auf *ausgabedatei*. |
| `-t` *trennzeichen* | Mit `-t` kann das *trennzeichen* der Felder eingestellt werden. |
| `-u` | Doppelte Schlüssel werden unterdrückt. |

## 16.10 tar - Sicherungsbefehl

**Name:**

`tar` – sichert die angegebenen Dateien in eine Datei oder auf Band oder liest diese Dateien wieder vom Medium ein. Es muß eine der Optionen `crtux` angegeben werden.

Die Verwendung von absoluten Dateibezeichnungen sollte vermieden werden, da diese Vorgehensweise beim vorgesehenen Entladen der Verzeichnisse sehr leicht zu Konfliktsituationen führt. Dateiverzeichnisse sollten daher direkt aus dem übergeordneten Verzeichnis heraus angesprochen werden.

**Synopsis:**

`tar -{c|r|t|u|x}` [*blmovw*] [*f dateiname*] ⟨*datei*⟩. . .

**Optionen:**

| | |
|---|---|
| `-c` | Ein neues "Band" wird angelegt und das Sichern beginnt am "Band"anfang. |
| `-r` | Die Dateien werden am "Band"ende angefügt. |
| `-t` | Das "Band" wird nach den vorgegebenen Namen durchsucht, und die gefundenen Namen ausgegeben. Es wird damit eine Art von Inhaltsverzeichnis des "Bandes" erstellt. |
| `-u` | Die angegebenen Dateien werden nur dann auf "Band" geschrieben, wenn sie entweder noch nicht auf dem "Band" stehen, oder neueren Datums sind. |

Zusatzbuchstaben, die nur in Verbindung mit einer der o.g. Optionen angegeben werden können.

| | |
|---|---|
| `f` *datei* | *datei* ist der Name des Gerätes oder der Datei, auf die gesichert, oder von der gelesen werden soll. |
| `o` | Die eingelesenen Dateien sollen statt ihrer bisherigen Benutzer- und Gruppennummer die des aufrufenden Benutzers erhalten. |
| `v` | Gibt die gesicherten, bzw. gelesenen Dateinamen während des Schreib-, bzw. Lesevorganges aus. |

## 16.11 time - Dauer eines Prozeßes ausgeben

**Name:**

`time` – gibt nach Beendigung des *kommandos* die dafür benötigte Ausführungszeit mit Verweildauer (*real*), Zeit im Benutzermodus (*user*) und die Zeit im Systemmodus (*sys*) jeweils in Sekunden aus.

**Synopsis:**

`time` ⟨*kommando*⟩

## 16.12 wc - Zählen von Zeichen, Zeilen und Wörtern

**Name:**

`wc` – gibt die Anzahl von Zeichen, Zeilen oder Wörtern der Eingabedatei aus. Wird keine *datei* angegeben, so wird von Standardeingabe gelesen. Wird keine der Optionen angegeben, so werden alle Angaben ausgegeben. Wörter sind durch Sonderzeichen getrennte Zeichenketten.

**Synopsis:**

`wc` [*-clw*] [*datei ...*]

**Optionen:**

`-c` (characters) Anzahl der Zeichen

`-l` (lines) Anzahl der Zeilen

`-w` (words) Anzahl der Wörter

## 16.13 compress - Komprimierung

**Synopsis:**

`compress datei ...`

`uncompress datei ...`

**Beschreibung:**

Die angegebenen Dateien werden komprimiert und mit der Endung **.Z** versehen. Eine Komprimierung erfolgt nicht, falls Verweise auf diese Datei existieren, kein Platzgewinn erzielt wird, die Datei nicht gelesen werden kann, oder ein Fehler auftritt.
uncompress dekomprimiert eine **.Z**-Datei wieder, und entfernt die **.Z**-Endung.

Beispiele:

```
$ ls -al oper.*
-rw-r--r--   1 ka  20480 Dec 03 11:30 oper.tar

$ compress oper.tar

$ ls -al oper.*
-rw-r--r--   1 ka   8037 Dec 03 11:30 oper.tar.Z
```

# 17 Programmentwicklung unter UNIX

Compiler $\leftrightarrow$ Interpreter

- C
- FORTRAN

- alle anderen Compiler wie
- PASCAL
- ASSEMBLER
- PEARL
- COBOL
- BASIC
- PROLOG
- LISP
- MODULA–2
- ADA

# 18 Vorgänge beim Programmieren

- Editieren einzelner Module $\Rightarrow$ Quellcode
- Übersetzen des Codes $\Rightarrow$ Maschinencode
- Binden der Objektdateien $\Rightarrow$ Programm
- Testen des ausführbaren Programms $\Rightarrow$ Zeitverlust

## 18.1 Die Programmierumgebung

- C-Compiler (`cc`) Präprozessor (`cpp`)
- Fortran-Compiler (`xlf od. frt`)
- Assembler (`as`)
- Bibliotheksverwalter (`ar`) Erzeugung von Bibliotheks–Archiven
- Linker (`ld`)
- Header–(Include)–Dateien (Definitionen, Konstanten, Makros)

Typische Dateien, die mit der Softwareentwicklung in Verbindung stehen:

- C–Quelltext–Dateien (`*.c`)
- FORTRAN–Quelletxt–Dateien (`*.f`)
- Assembler–Programm–Dateien (`*.s`)
- Pascal–Programm–Dateien (`*.p`)
- Objekt–Dateien (`*.o`)
- Bibliotheken (`*.a`)

## 18.2 Allgemeines zu Compilern unter UNIX

UNIX standardisiert die Bedienung der Compiler in weiten Bereichen:

Die wichtigsten Optionen beim Aufruf der Compiler:

| | |
|---|---|
| -g | Debug Informationen |
| -c | nur compilieren (nicht linken) |
| -o name | Ausführbare Datei `name` |
| -O | Optimize–Anweisung |

diese Schalter sind immer mit NO vorbelegt. Im Falle des -o Schalters ist a.out voreingestellt.

*Beispiele:*

*cc -O -c programm1.c*
*xlf -o xxx.out xxx.f*

Es ist wichtig zu beachten, daß ein Compileraufruf in der Regel auch automatisch den Binder aufruft. Somit können in der Regel Kommandooptionen für den Binder mit in die Optionen des Compilers mit aufgenommen werden.

## 18.3 Die Programmiersprache C

C–Sprachkonzept
Unter den zwei grossen Klassen von Programmiersprachen, den maschinen-orientierten (Assembler) und den problemorientierten wie Fortran, Pascal, Basic usw. ist C der letzteren zuordnen.
C-Programme sind unter Einhaltung einfacher Regeln voll portabel. Somit werden die Schnelligkeit einer systemnahen Programmiersprache mit der Portabilität verbunden.
Der Sprachumfang von C ist im Vergleich zu anderen Sprachen relativ klein und beschränkt sich auf Zuweisungen, Vergleiche, strukturierende Elemente und Speicheroperationen.
Die Stärke von C liegt in der großen Anzahl von Funktionen, vergleichbar mit Unterprogrammen. Was innerhalb eines Programmes aussieht wie der Aufruf eines Befehles ist das Starten einer Funktion.
Als Beispiel sei die `printf`-Funktion genannt, die für eine Ausgabe auf dem Bildschirm benutzt werden kann.

Beispiel :

```
printf("hello world\n");
```

Der Parameter der `printf`-Anweisung ist `hello world`, eingegrenzt durch die beiden Anf'ührungszeichen.
Er wird der `printf`- Funktion übergeben ; es erfolgt die Ausgabe.
Ein C-Programm besteht aus einer Anzahl von vordefinierten oder eigendefinierten Funktionen, die kombiniert werden.
Häufig verwendete Funktionen können vorübersetzt und in Bibliotheken abgelegt werden. Einige Funktionen sind (z.B. durch ANSI-Norm 1989) standardisiert auf allen Rechnersystemen verfügbar.

## 18.4 C-Compileraufruf (cc)

- Front–End für Benutzer, bestehend aus:
  1. Präprozessor cpp
  2. Compiler
  3. Linker ld (optional)
- Die wichtigsten Optionen beim Aufruf des cc
  (ohne die allgemein gültigen):

| | |
|---|---|
| -Dname=wert | `name` erhält den Wert `wert` |
| -Dname | `name` erhält den Wert `true` |
| -Iverzeichnis | `Verzeichnis`, in dem kann Dateien erwartet werden |

### 18.4.1 Der Präprozessor cpp

Der cpp-Präprozessor ist ein generelles Tool zur Bearbeitung von ACSII-Datein; er ist ein Textersetzungsprogramm, der über Konditionen gesteuert werden kann. Er wird häufig nur mit C-Code in Verbindung gebracht, da er dem C-Compiler vorgeschaltet ist.

`Cpp-Anweisungen:`

- #include <*Dateiname*> aus `/usr/include`

- #include "*Dateiname*" aus aktuellem Verzeichnis
- #define *name*
- #define *name text*
- #define *name(argument, ...) text*
- #undef name
- #if *c_ausdruck*
- #ifdef *name*, #ifndef *name*
- #else
- #endif

## 18.5 Allgemeines zur C–Programmierung

### 18.5.1 Sprachkurzbeschreibung

- Kommentare (alles zwischen /* und */)
- Namen (Unterscheidung Groß–Kleinschrift)
- Reservierte Worte (auto extern...)
- Konstanten(entsprechend den Datentypen `int, long`,...) und Stringkonstanten

  1. integer_zahl=300;
  2. gleitkomma_zahl=1.2345;
  3. ein_zeichen='A';
  4. eine_zeichenkette="hallo welt";

- Operatoren für Grundrechenarten sowie math. Standardfunktionen

### 18.5.2 Kontrollstrukturen in C

- `if`–Anweisungen

  Die `if`-Anweisung eroeffnet die Möglichkeit, Anweisungen auszuführen, wenn bestimmte Bedingungen erfüllt sind.
  *Syntax* *if(Bedingung){*
  *Befehlsblock*
  }
  Dem Schlüsselwort `if` folgt in Klammern die Bedingung. Ist diese erfüllt, so wird der Befehlsblock ausgefürt.

  Anwendung
  Die `if` -Anweisung ist eine Kontrollanweisung, die bei einfachen Entscheidungen verwendet wird.

*Beispiel* :

```
if(a > b)   {
                c = a;
                a = b;
                b = c;
                printf("   a   und   b   vertauscht");
}
```

Fehlerquellen
Steht direkt hinter der Bedingung ein Semikolon, so wird eine leere Anweisung erzeugt. Der Befehlsblock wird nicht ausgeführt.

- `else–if` Ketten

Diese Anweisung wird in Verbindung mit dem Schlüsselwort → `if` verwendet und eröffnet die Möglichkeit der alternativen Bewertung von Ausdrucken.

```
if(Bedingung1){
                Befehlsblock
}
elseif(Bedingung2){
                Befehlsblock
}
else{
                Befehlsblock
}
```

Die Bedingungen werden nacheinander bewertet. Der dazugehörige Befehlsblock wird ausgeführt, wenn die Bedingung wahr ist. Der Programmablauf wird dann nach der gesamten Kontrollstruktur fortgesetzt. Trifft keine der Bedingungen zu, wird der Befehlsblock der `else` -Anweisung ausgefürt.

Fehlerquellen
Bei folgender Struktur wird ein häufig auftretender Fehler deutlich :

```
if(a > 100)
        if(a > 50)
                a − −;
elsea + +;
```

Die `else`-Anweisung bezieht sich auf das letzte davor stehende `if`. Sollte es sich aber auf das erste `if` beziehen, so mußdie Struktur folgendermassen aussehen :

```
if(a > 100)
        {if(a > 50)
                a − −;
         }
elsea + +;
```

- `while`–`for` Schleifen

  Die `while`- Anweisung wird zur Konstruktion von Schleifen verwendet.

  *Syntax*

  $$while(Bedingung)$$
  $$\{$$
  $$\quad Befehlsblock$$
  $$\}$$

  Ist die Bedingung wahr, so wird der Befehlsblock ausgeführt und die Bedingung wieder überprüft. Dieser Ablauf wiederholt sich bis die Bedingung nicht mehr erfüllt ist.
  Anwendung
  *Beispiel* :

  $$while((c = getchar())! =' E')$$
  $$\quad putchar(c);$$

  Zeichen, die über die Tastatur eingegeben werden ( Funktion getchar()) erscheinen auf dem Bildschirm (Funktion putchar(c)). Bei Eingabe des Buchstabens E wird die Schleife abgebrochen.
  Fehlerquellen
  *Beispiel* :

  $$x = 10;$$
  $$\quad while(x > 0);$$
  $$\quad\quad x - -;$$

  Durch das Semikolon direkt nach der `while`-Anweisung wird eine Endlosschleife erzeugt.

- `for` Schleifen

  Der zur `for`-Anweisung gehoerige Befehlblock wird in Abhängigkeit von den drei Parametern der Anweisung ausgefürt.
  *Syntax*

  $$for(\ Ausdruck1;\ Ausdruck2;\ Ausdruck3)$$
  $$\{$$
  $$\quad Befehlsblock$$
  $$\}$$

  Ausdruck1 ist die Initialisierung, Ausdruck2 die Bedingungsabfrage und Ausdruck3 die Schrittweite. Sie sind alle drei optional und werden durch ein Semikolon getrennt.
  Anwendung
  Die `for` -Schleife sollte benutzt werden, wenn Initialisierung und Reinitialisierung einfache Ausdrücke sind.
  *Beispiel :*

  $$for(x = 0, y = 1; x < 10; y + +, x + +)$$
  $$\{$$
  $$\quad Befehlsblock$$
  $$\}$$

  Im ersten Ausdruck erfolgt die Initialisierung der Schleifenlaufvariablen. Im Beispiel wird x auf Null und y auf Eins gesetzt. Zu beachten ist das Komma zwischen den Initialisierungen.
  Der zweite Ausdruck prüft, ob die Variable x kleiner als Zehn ist. Ist dies nicht der Fall, so wird die Schleife beendet und das Programm nach dem Befehlsblock fortgesetzt.
  Solange die Bedingung aber erfüllt ist, wird der Befehlsblock abgearbeitet, nach jedem Durch-

lauf x und y inkrementiert und die Bedingung wieder überprüft.
Fehlerquellen
Der häufigste Fehler bei der `for`- Anweisung ist die Konstruktion von Endlosschleifen und leeren Schleifen.
*Beispiel :*

```
for(i = 0; i < 10; j++) {
        a = i + 1;
        printf("  a   gleich  %d");
}
```

Die Variable i wird nicht inkrementiert, es entsteht eine Endlosschleife.
*Beispiel :*

```
for(x = 0; x < 10; x++);
{
        Befehlsblock
}
```

Durch das Semikolon nach der `for`-Anweisung wird der Befehlsblock nicht in Abhängigkeit von der Schleifenvariablen x ausgefürt. Vielmehr wird die Schleife zehnmal mit einer leeren Anweisung durchlaufen und dann verlassen. Der Befehlsblock wird nicht bearbeitet.

- `do-while` Schleifen

  Die `do`-Anweisung dient zum Aufbau einer `do-while`-Schleife. Die Schleife wird ausgeführt solange die `while`-Bedingung am Ende der Schleife erfüllt ist.

  Syntax

```
do
{
        Befehlsblock
}while(Bedingung);
```

  Anwendung
  *Beispiel* :

```
k = -10;
do
{
    k++;
    printf("k  ist  negativ");
}while(k < 0);
```

  Die Variable k wird bis zum Wert 0 inkrementiert und jeweils auf dem Bildschirm ausgegeben.

  Allgemein sollte diese Schleifenkonstruktion da eingesetzt werden, wo der Befehlsblock auf jeden Fall mindestens einmal durchlaufen werden muß. Ansonsten ist eine einfache `while`- oder `for`-Schleife vor- zuziehen.

  Fehlerquellen
  Aus einer `do-while` Schleifen kann sehr leicht eine Endlosschleife entstehen, wenn die `while`-Bedingung immer erfüllt ist. Würde im obigen Beispiel k nicht inkrementiert, so wird die Schleife unendlich oft durchlaufen.

- `switch`

  Diese Kontrollstruktur ermöglicht es ganzzahlige Ausdrücke mit den zur Auswahl stehenden Werten zu vergleichen.
  Syntax

```
switch(Ausdruck)
{
        case : Konstante1
               Befehlsblock;
               break;
        case : Konstante2
        case : Konstante3
        case : Konstante4
               Befehlsblock;
               break;
        ...
               ...
        ...
               ...
        default Befehlsblock;
}
```

  Dem Schlüsselwort `switch` folgt in der Klammer der Ausdruck (vom Typ Integer oder Character), der mit den Konstante bei den `case`-Anweisungen verglichen werden soll.
  Fällt der Vergleich positiv aus, so wird der zugehörige Befehlsblock ausgeführt. Ein Befehlsblock mußübrigens nicht durch geschweifte Klammern begrenzt werden. Die `break`-Anweisung bewirkt dann das sofortige Verlassen der Struktur. Wie oben ersichtlich können auch mehrere `case`- Konstanten einem Befehlsblock zugeordnet werden.
  Kann beim Vergleichen keine passende `case`- Konstante gefunden werden, wird der `default`-Befehlblock ausgeführt.

  Anwendung
  *Beispiel :*

```
switch(c)
{
        case 'h' :
        case 'H' :
               printf("Hilfe");
               break;
        case 'e' :
        case 'E' :
               exit(0); /* Programm verlassen */
        case 'i' :
        case 'I' :
               printf("   Inhaltsverzeichnis");
               break;
        default : printf("   Keine   gueltige   Eingabe");
}
```

  Das Bedingungsargument 'c' ist vom Typ Character.

Fehlerquellen
Es ist darauf zu achten nach jeder `case`- Konstanten einen Doppelpunkt zu setzen. Die `case`-Anweisung sollte immer mit `break` abgeschlossen werden um etwaige Probleme bei der Modifikation der Struktur zu vermeiden. Es dürfen keine zwei `case`- Konstanten mit dem gleichen Wert auftreten. Jede Konstante muß zu allen anderen alternativ sein.

- `break`

  Die `break`-Anweisung wird benutzt den Ablauf von Schleifen und `switch`-Kontrollstrukturen zu steuern. Nach Ausführen der Anweisung wird die innerste →`for`-, →`while`- → `do-while`-Schleife oder →`switch`- Struktur verlassen.

  *Syntax*

  $break;$

  Anwendung

  Der Ablauf des Programmes wird bewußt gesteuert. Nach Verlassen einer Schleife bzw. `switch`-Struktur mit `break` wird mit den naechst folgenden Anweisungen fortgefahren.
  Man vermeidet somit die →`goto`-Anweisung, die zu unübersichtlichen Programmen führen kann.

  Fehlerquellen
  `Break` ist nur in Verbindung mit Schleifen oder `switch`- Kontrollstrukturen sinnvoll und darf nicht isoliert im Programm stehen.

- `continue`

  Mit der `continue`-Anweisung ist es möglich, den Ablauf von → `for`-, → `while`- oder → `do-while`- Schleifen zu steuern.
  Sie bewirkt das sofortige Durchführen des naechsten Schleifendurchlaufes.
  *Syntax*

  $continue;$

  Anwendung
  *Beispiel* :

  $for(i = 0; i < 10; i++)$
  {
  $\quad if(i == 5)$
  $\quad\quad continue;$
  $\quad printf(" \quad \%d");$
  }

  In der `for`-Schleife wird i von 0 bis 10 inkrementiert. Hat i den Wert 5, so erfolgt sofort der naechste Schleifendurchlauf, ohne dass die folgende `printf`-Anweisung ausgeführt wird.
  Bei der `do-while`-Schleife wird nach der `continue`-Anweisung die `while`-Bedingung am Ende der Schleife überprüft.
  Durch das `Überspringen` von Anweisungen kann die Geschwindigkeit eines Programmes erhöht werden.

  Fehlerquellen
  Die `continue`-Anweisung kann nur sinnvoll in Verbindung mit den oben erwaehnten Schleifen

eingesetzt werden. Steht sie isoliert im Programm, so ist das ein Fehler.

- `return`

  Die `return`-Anweisung dient dazu von einer Funktion in die Funktion zurückzukehren, von der aus der Funktionsaufruf erfolgte.
  *Syntax*

  *return(Ausdruck);*

  Der Ausdruck ist das Argument der `return`-Anweisung. Er ist optional und kann ein arithmetischer Ausdruck, eine Integervariable, ein einzelnes Zeichen oder ein Pointer (`Zeiger`) sein.

  Anwendung
  Mit `return` wird der Programmablauf gesteuert und Parameter oder Ausdrücke zwischen Funktionen ausgetauscht.
  Beispiel:

```
main( )
{
        int j;
        j = zahl();
        printf("Die Zahl ist : %d", j);
}

int zahl( )
{
        printf("Hier Funktion 'Zahl'") ;
        return(5) ;
}
```

  Der Hauptfunktion `main` wird mittels der `return`-Anweisung die Zahl 5 übergeben.

  Fehlerquellen
  Der → `Datentyp` der Funktion, in der die `return`- Anweisung eingesetzt wird, muß mit dem des zurückgegebenen Wertes übereinstimmen. Im Beispiel oben ist sowohl die Funktion (`zahl()`) als auch der zurückgegebene Wert (`5`) vom Typ Integer.

- `continue`

  Mit der `continue`-Anweisung ist es möglich, den Ablauf von → `for`-, → `while`- oder → `do-while`- Schleifen zu steuern.
  Sie bewirkt das sofortige Durchführen des naechsten Schleifendurchlaufes.
  *Syntax*

```
continue;
```

  Anwendung
  *Beispiel* :

```
for(i = 0; i < 10; i++)
{
        if(i == 5)
                continue;
        printf(" %d");
}
```

In der `for`-Schleife wird i von 0 bis 10 inkrementiert. Hat i den Wert 5, so erfolgt sofort der naechste Schleifendurchlauf, ohne dass die folgende `printf`-Anweisung ausgeführt wird.
Bei der `do-while`-Schleife wird nach der `continue`-Anweisung die `while`-Bedingung am Ende der Schleife überprüft.
Durch das `Überspringen` von Anweisungen kann die Geschwindigkeit eines Programmes erhöht werden.

Fehlerquellen
Die `continue`-Anweisung kann nur sinnvoll in Verbindung mit den oben erwaehnten Schleifen eingesetzt werden. Steht sie isoliert im Programm, so ist das ein Fehler.

- `Leere Anweisung ';'`

  Eine leere Anweisung steht anstelle einer von der Syntax erforderlichen Anweisung. Sie bewirkt keine weitergehende Aktionen.
  Syntax ;

  Anwendung
  Mit der leeren Anweisung lassen sich Schleifen schreiben, die zur Zeitverzögerung im Programm dienen (Leerschleifen).
  *Beispiel* :

  $$for(x = 0; x < 16; x++)$$
  $$;/*LeereAnweisung*/$$

  Fehlerquellen
  Sehr vorsichtig muss mit der leeren Anweisung bei Schleifen umgegangen werden.
  *Beispiel* :

  $$for(x = 0; x < 10; );/*Endlosschleife*/$$
  $$x++;$$

  Die Variable `x` wird nicht inkrementiert. Die Leerschleife wird zur Endlosschleife.

### 18.5.3 Speicherklassen

- `auto`:

  Anwendungsbereich

  Variablen der `auto`-Speicherklasse werden lokal innerhalb einer Funktion eingesetzt.
  Die Variable wird bei Aufruf der Funktion `auto`-matisch erzeugt und verschwindet wieder bei Verlassen.
  *Syntax*

  *auto Datentyp Variablenname ;*

  oder

  *auto Datentyp Variablenname1, Variablenname2, ... ;*

  Dem Wort `auto` folgt der Datentyp der Variablen und ihr Name. Bei mehreren Variablennamen muessen diese jeweils durch ein Komma getrennt werden.

  Anwendung
  Die `auto`-Variable muß im Kopf der jeweiligen Funktion deklariert und dort oder später im

Laufe des Programmes initialisiert werden.
*Beispiel :*

*auto intx = 10; /* Deklaration und Initialisierung */*— Soll eine Variable auch anderen Funktionen bekannt werden, so mußzur → `extern` oder `global`-Speicherklasse übergegangen werden.
Variablen, die ohne Speicherklasse deklariert werden, sind `auto`-Variablen.

*Beispiel :*

*int y ;*

entspricht

*auto int y ;* Bei häufiger Benutzung einer `auto`-Variable ist zur → `register`-Speicherklasse überzugehen. Die Zugriffszeite werden dadurch minimiert.

Fehlerquellen:
Eine fehlende Initialisierung führt zu Fehlern.

- `extern`:

**Anwendungsbereich**

Variablen der `extern`-Speicherklasse sind allen Funktionen eines Programmes bekannt.
*Syntax*

*extern Datentyp Variablenname ;*

oder

*extern Datentyp Variablenname1, Variablenname2, ... ;*

Dem Wort `extern` folgt der Datentyp der Variablen und ihr Name. Bei mehreren Variablennamen müssen diese jeweils durch ein Komma getrennt werden.

**Anwendung**

`Extern`-Variablen behalten im Gegensatz zu solchen vom Typ → `auto` oder → `register` (lokale Variablen) ihren Wert auch bei Verlassen einer Funktion bei.
Sie belegen also während des Programmlaufes dauernd Speicherplatz. Um nicht zu viel Speicherplatz zu belegen, sollte, wenn möglich, lokale Variablen benutzt werden.

Die Deklaration erfolgt vor allen Funktionen.
*Beispiel : int y ;*
Auf das Schlüsselwort `extern` kann hier verzichtet werden.

Mit : int $y = 5$;

erfolgt die Initialisierung.

Innerhalb der einzelnen Funktionen, in denen auf die `extern`-Variable zugegriffen wird, ist die Vereinbarung dann folgendermaßen vorzunehmen :
*extern int y ;*

**Fehlerquellen**

Die Deklaration von `extern`-Variablen kann leicht zu unübersichtlichen Programmen führen, die zudem schlecht zu modifizieren sind. Ausserdem können einzelne Funktionen dann nicht so leicht in andere Programme eingebaut werden.

- `static`: Die Lebensdauer des Objektes geht bis zum Programmende

  1. intern: innerhalb einer Funktion sind die `static`– Variablen privat, außerhalb sind sie unbekannt
  2. extern: innerhalb einer Quelltext–Datei ist eine `static`–Variable allen Funktionen bekannt

- `register`: Hinweis auf Optimierung an den Compiler
  $\Rightarrow$ schneller, kompakter Code

### 18.5.4 Datentypen

Die Speicher–Reservierung der Datentypen ist **maschinenabhängig**. Die Angaben in Klammern geben übliche Werte an (Vax/780, 3B2, RS6000).

### 18.5.5 Grundtypen

- `char` - Variable
  Geltungsbereich
  `Char`-Variablen sind Variablen für einzelne Zeichen.
  Der im Praktikum verwendete Compiler wandelt sie bei vielen Operationen in die entsprechende $\rightarrow$`int`-Variable um.
  Der Code des jeweiligen Zeichens wird unter der `char`-Variablen abgelegt.

  Deklaration

  char Variablenname ;
  oder
  char Variablenname1, Variablenname2, . . . ;

  Dem Wort `char` folgt der Name der Variablen und das Semikolon.
  Bei mehreren Namen müssen diese jeweils durch ein Komma getrennt werden.

  Initialisierung
  Eine Initialisierung ist folgendermaßen vorzunehmen :

  char Variablenname = 'Zeichen' ;
  *Beispiel :*
  *char c = 'h' ;*

  Fehlerquellen
  Unter einer `char`-Variablen kann nur ein einzelnes Zeichen abgespeichert werden. Bei mehreren

Zeichen muß zu einem Ausdruck in folgender Form übergegangen werden :

*char *string = Text";* Weitere Information zu diesem Thema ist unter → Zeiger zu erhalten.

- `short` - Variable
  Geltungsbereich
  Durch eine Deklaration als `short`- Variable wird der verfügbare Speicherbereich dieser Variablen halbiert. So kann besonders bei längeren Datenfeldern Speicherplatz gespart werden.

  Deklaration

  short Datentyp Variablenname ;

  oder

  short Datentyp Variablenname1, Variablenname2 , ... ;

  Dem Wort `short` folgt der Datentyp der Variablen gefolgt von ihrem Namen. Bei mehreren Namen müssen diese jeweils durch ein Komma getrennt werden. Enthält die `short`-Deklaration keine weiteren Datentypen, so wird für die Variable der Datentyp `integer` angenommen.
  *Beispiel* : *shortzahl*;
  $DieVariableistvomTyp$`shortinteger`.
  $SiekannWertevon -32768bis+32768annehmen.$

  Initialisierung
  Eine Initialisierung ist folgendermaßen vorzunehmen :

  short Datentyp Variablenname = Wert ;

  *Beispiel* :

  $shortzahl = 6457;$

  Fehlerquellen
  Eine Variable vom Typ `short` hat den Wertebereich einer `integer`-Variablen (-32768 bis +32768). Es muß darauf geachtet werden, daß dieser nicht überschritten wird.

- `Integer` - Variable
  Geltungsbereich
  Eine Variable vom Typ `int` (integer) ist eine ganze Zahl, deren Größe vom verwendeten Computer abhängt. Bei den im Praktikum verwendeten Maschinen kann sie den Bereich von -32768 bis +32767 abdecken. Bei der Deklaration als →`unsigned int` geht der Bereich von 0 bis 65635.

  Deklaration

  int Variablenname ;

  oder

  int Variablenname1, Variablenname2, ... ;

Der Festlegung des Datentyps durch `int` folgt der Variablenname und ein Semikolon. Bei mehreren Namen müßen diese jeweils durch ein Komma getrennt werden.

Initialisierung
Eine Initialisierung ist folgendermaßen vorzunehmen :

int Variablenname = Wert ;

*Beispiel* :

$intZahl = 5;$

Fehlerquellen
Der zulässige Bereich der `int`-Variable darf nicht überschritten werden, da sonst Fehler auftreten können. Eine Erweiterung des Bereiches ist durch den → `long`-Datentyp oder wie oben erwähnt durch :

unsigned int Variablenname ;

zu erreichen.

- `float` - Variable
  Geltungsbereich
  Variable vom Typ `float` werden für Gleitkommazahlen von mittlerer Genauigkeit verwendet. Das Maß für die Genauigkeit ist die Anzahl der Nachkommastellen. Bei den im Praktikum verwendeten Maschinen sind für diesen Variablentyp 8 Stellen nach dem Komma vorgesehen. Ist dies Anzahl der Stellen größer, so wird gerundet.

  Deklaration

  float Variablenname ;

  oder

  float Variablenname1, Variablenname2, ... ;

  Dem Wort `float` folgt der Name der Variable und das Semikolon. Bei mehreren Namen müßen diese jeweils durch ein Komma getrennt werden.

  Initialisierung
  Eine Initialisierung ist folgendermaßen vorzunehmen :

  float Variablenname = Wert ;

  *Beispiel* :

  $floatzahl = -4.87634;$

  oder

  $floatzahl = -0.487634E - 1$

  Die zweite Schreibweise ist die Darstellung der Gleitkommazahl mit Vorzeichen, Dezimalbruch und Exponent.

Das Vorzeichen ist negativ oder positiv (das `+` kann weggelassen werden), der Dezimalbruch ist eine Zahl kleiner als eins und der Exponent wird wie oben dargestellt (das `E` kann auch klein geschrieben werden).

Fehlerquellen
Durch die beschränkte Anzahl der Nachkommastellen können Rundungs- fehler auftreten.

- `long` - Variable
  Geltungsbereich
  Durch eine Deklaration als `long`-Variable wird der verfügbare Speicherbereich dieser Variablen verdoppelt. Bei denen im Praktikum verwendeten Maschinen hat eine Variable vom Typ `long` die Größe von vier Byte.

  Deklaration
  long Datentyp Variablenname ;
  oder
  long Datentyp Variablenname1, Variablenname2, ... ;

  Dem Wort `long` folgt der Datentyp der Variablen, ihr Name und ein Semikolon. Bei mehreren Variablennamen müßen diese jeweils durch ein Komma getrennt werden. Enhält die `long`-Deklaration keinen weiteren Datentypen, so wird für die Variable der Datentyp `integer` angenommen.

  Beispiel :
  long zahl ;

  Die Variable mit dem Namen `zahl` ist vom Typ `long integer`. Sie kann Werte von -2147483648 bis +2147483648 annehmen (wenn eine Variable vom Typ `integer` 2 Byte Speicherplatz zur Verfügung hat).

  Eine weitere Moglichkeit der Deklaration ist :

  unsigned long Variablenname ;

  Nun kann die Variable Werte von 0 bis 4294967295 annehmen.

  Initialisierung
  Eine Initialisierung ist wie folgt vorzunehmen :

  long Datentyp Variablenname = Wert ;
  *Beispiel* :
  $longzahl = 1034577;$

  Fehlerquellen
  Wenn unsicher ist, ob eine Variable vom Typ `integer` im Laufe des Programmes ihren Wertebereich übersteigt, sollte sie als `long`-Variable deklariert werden.

- `double` doppeltgenaue float (64 Bit)
- `void`

  Datentyp für Funktionen, welche keinen Rückgabewert erfordern.

### 18.5.6 Vektoren und Zeiger

- T a ( ); $a$ ist eine Funktion vom Ergebnistyp $T$
- T a [ ]; $a$ ist ein Feld (Array) mit Elementen des Typs $T$
- T * a; $a$ ist ein Zeiger (Pointer) vom Typ $T$
- T ** a; $a$ ist ein Zeiger (Pointer) auf Zeiger vom Typ $T$

### 18.5.7 Felder (Arrays)

- ganzahlige Indizes $0 \ldots n-1$
- Dimensionen $> 2$ möglich (z.B. $a[i][j][k]$)
- auch Arrays von Funktionen möglich ,z.B.
  *long*( a [ ] )( );*
  tt a ist ein Feld von Zeigern auf Funktionen des Typs `long` )

### 18.5.8 Strukturen

In einer Struktur werden Datenobjekte , deren Typ unterschiedlich sein kann, zusammengefaßt. Man nennt die Datenobjekte Komponenten der Struktur. Sie können einzeln oder als ganzes angesprochen werden.

Deklaration

```
struct Strukturname
{
        Datentyp1 Name1 ;
        Datentyp2 Name2 ;
        ...
        ...
} ;
```

Dem Wort `struct` folgt der Name der Struktur. Innerhalb der ge- schweiften Klammern stehen die Komponenten, die sich aus dem Datentyp und einem Namen zusammensetzen. Der schließenden Klammer muß ein Semikolon folgen. Die Deklaration der Struktur sollte am Anfang des Programmes erfolgen.

Anwendung
Der Struktur kann ein Strukturvariablenname zugewiesen werden. Der Strukturvariablenname wird folgendermaßen festgelegt :

*struct Strukturname Strukturvariablenname ;*

Es ist auch eine ganze Liste von Namen möglich :

```
struct Strukturname Strukturvariablenname_1 ,
                    Strukturvariablenname_2 ,
        ...
        ...
        ...
        ...;
```

oder

*struct Strukturname Strukturvariablenname*[3];

Dieser wird verwendet, um die einzelnen Komponeneten der Struktur direkt anzusprechen :

*Strukturvariablenname.Name_der_Komponente*

Entscheidend ist der Punkt '.' zwischen Variable und Name. Der Ausdruck kann nun wie eine normale Variable benutzt werden.

*Beispiel :*

```
struct ergebnisse /* Ergebnisse olymp. Spiele */
{
        char *vorname ;
        char *nachname ;
        float zeit ;
} ;

main()
{
        struct ergebnisse athlet[3] ;
        /* Zuweisung Strukturvariablenname */

                athlet[0].vorname = "Ben   'The   Steroit'";
                athlet[0].nachname = "Johnson";
                athlet[0].zeit = 9.79 ;

                athlet[1].vorname = Carl   'The   Brikett'";
                athlet[1].nachname = Louis";
                athlet[1].zeit = 9.88 ;

                athlet[2].vorname = "Woody   'The   Rabbit'";
                athlet[2].nachname = Ällen";
                athlet[2].zeit = 27.87 ;
```

```
        printf("Winners of the 100 meters final\n") ;

        for( n = 0;n < 3;n++)
                printf("%s %s %f",athlet[n].vorname,\
                        athlet[n].nachname, athlet[n].zeit) ;

}
```

### 18.5.9 Varianten–Records (Union)

Gesonderte Anordnung der Elemente zum sparen von Speicher
(heute nicht mehr wichtig)

### 18.5.10 Typdefinitionen

Erstellung neuer Datentypen. Z.B. eines Datentyps, welcher unter dem Namen `Messwert` drei `float`-Variablen und eine String-Varieble für 256 Zeichen bereitstellt.

```
typedef struct {
        float x, y, z;
        char bla[256];
} Messwert;
```

Später kann dann der neue Datentyp `Messwert` z.B. folgendermaßen verwendet werden:

```
Messwert *daten;
...
daten− >x=25;
daten− >y=45;
daten− >z=35;
strcpy(daten− >bla,"fasel");
...
```

### 18.5.11 Funktionen

Funktionen ähneln Unterprogrammen in Fortan bzw. Prozeduren Pascal. Mit Ihnen können Programmbestandteile zusammengefasst werden, welche im weiteren Verlauf intern nicht weiter beachtet werden (Blackbox) müssen, da lediglich ihre Funktionalität interessiert, und nicht die Art wie dies realisert wurde.

- C kennt nur Funktionen, keine Prozeduren
- liefern einen Ergebniswert eines bestimmten Datentyps zurück (default `int`)
- Call By Value $\leftrightarrow$ Call By Reference
- Rücksprung aus einer Unterroutine entweder bei Erreichen des Endes des Funktionsblockes oder der Anweisung `return`
- Der Datentyp `void` dient der Funktionsdeklaration von Funktionen, deren Rückgabewert nicht ausgewertet werden soll

- Der Rückgabewert der Funktion `main()` wird an die aufrufende Shell zurückgegeben und kann dort ausgewertet werden

Beispiel:

```
float addiere(Zahl1, zahl2)
        float Zahl1, Zahl2:
{
        return(Zahl1+Zahl2);
}
main( )
{
        float z1, z2, z3, addiere( );
        ...
        z1=123.4;
        z2=567.89;
        z3=addiere(z1,z2);
        ...
}
```

### 18.5.12 Sonstiges

- Typumwandlungen können mittels des Typumwandlungs–Operators durchgeführt werden (`casting`):

  Im folgenden Beispiel ist die Funktion `malloc( )` aufgezeigt, die eine Menge an Speicher für Variablen bereitstellt. Standardmäßig geschieht dies für Variablen des Typs `char`. Hier wird zunächst Platz füer 256 Bytes für die Variable `string` reserviert, nämlich 256 mal der Größe eines Zeichens (meist 1 Byte). Im 2. Beispiel wird für einen neuen Datantyp `MESSWERT` Platz für eine Variable `daten` von genau der Größe der Struktur (siehe oben `Struktur`),

```
int i;
long l;
char        *string;
MESSWERT    *daten;
...
...
string      = (char *)malloc(256*sizeof(char));
...
daten       = (MESSWERT *)malloc(sizeof(MESSWERT));
...
i           = (int)l;
```

### 18.5.13 Runtime Library

Damit nicht alle Programm-Bausteine von jedem Programmierer neu geschrieben werden müssen, liefert der Hersteller eines Betriebssystems zusätzlich zur Systemsoftware meist ein Software--Entwicklungspaket mit, welches eine mehr oder weniger vollständige Sammlung von Standard-Bibliotheken für den Programmierer bereitstellt.
Stellt Standard–Funktionen zur Verfügung für

- Stellt Standard–Funktionen zur Verfügung für
- Zeichenverwaltung z.B. `toascii, isalpha`
- Stringverwaltung z.B. `strcpy, index`
- Standard I/O–Operationen z.B. `read, write, printf`
- File I/O–Operationen z.B. `fread, fwrite, fopen, fseek`
- Allgemeine Bibliotheken–Funktionen z.B. `abort, malloc, free, system`
- Mathematische–Funktionen z.B. `abs, cos, sinh, exp, log10, pow...`

## 18.6 C–Beispielprogramm

```
1  /*
2          Primitives C-Beispielprogramm
3          Author: Felix  Wed Feb 26 19:58:03 GMT 1992
4  */
5  #include <stdio.h>
6
7  #define UP 1
8  #define DOWN 2
9  #define LEFT 3
10 #define RIGHT 4
11
12 usage (progname)
13
14          char *progname;
15
16 {
17         fprintf (stderr, "Usage: %s -[u|d|l|r] <text>, where:\n", progname);
18         fprintf (stderr, "u = print upwards, d = print downwards,\n");
19         fprintf (stderr, "l = write from right to left, r = write from left ");
20         fprintf (stderr, "to right.\n");
21         exit (1);
22 }
23
24
25
26 printout (word, direction)
27
28          char *word;
29          int direction;
30
31 {
32         int counter, stringlength = strlen (word);
33
34         for (counter = 0; counter < stringlength; counter++)
35         {
36                 if (direction == UP || direction == LEFT)
37                         putchar (word[stringlength - counter - 1]);
38                 else
39                         putchar (word[counter]);
40                 if (direction == UP || direction == DOWN)
41                         putchar ('\n');
42         }
43         if (direction == UP || direction == DOWN)
44                 putchar ('\n');
45         else
46                 putchar (' ');
47 }
48
49
50
51 main (argc, argv)
52
53 int argc;
54 char **argv;
55
```

```
{
        int direction, counter;

        if (argc < 2)
                usage (argv[0]);

        /* Optionen parsen */
        if (argv[1][0] == '-')
        {
                if (strlen (argv[1]) != 2 || argc < 2)
                {
                        usage (argv[0]);
                }
                else
                {
                        switch (argv[1][1])
                        {
                                case 'u':
                                        direction = UP;
                                        break;

                                case 'd':
                                        direction = DOWN;
                                        break;

                                case 'l':
                                        direction = LEFT;
                                        break;

                                case 'r':
                                        direction = RIGHT;
                                        break;

                                default:
                                        usage (argv[0]);
                                        break;
                        }
                }
                for (counter = 2; counter < argc; counter++)
                {
                        if (direction == DOWN || direction == RIGHT)
                                printout (argv[counter], direction);
                        else
                                printout (argv[argc - counter + 1], direction);
                }
        }
        putchar ('\n');
}
```

## 18.7 FORTRAN–Programmierung

`FORTRAN` wurde Mitte der 50ér Jahre entwickelt und ist somit die älteste höhere Programmiersprache von Bedeutung. Eine erste Standardisierung fand 1966 mit `FORTRAN 66` statt, welche in den 70érn durch `FORTRAN 77` abgelöst wurde.
Der Aufruf des Compilers ist auf den verschiedenen Plattformen unterschiedlich, wohingegen die Optionen und Argumente weitgehend betriebssystemsunabhängig sind. Bei vielen Systemen lautet das Kommando zum Aufruf des Compilers `f77` . Beim AIX von IBM lautet der Aufruf `xlf` (ähnlich dem C-Compiler-Aufruf `xlc`). Auf dem HHLR heißt das Kommando `frt`. Im Zweifelfall den Sysadmintrator fragen.

## 18.8 Assembler–Programmierung

## 18.9 Grundsätzliches zum Assembler

Assembler ist eine Programmiersprache, deren Befehle sich schon sehr an den tatsächlich ausgeführten Befehlen orientieren. Der größte Unterschied zum ausführbezaren Programm ist, daß die Befehle nicht mehr aus Bitkombinationen bestehen, sondern aus „Mnemonics“. Dies sind Wörter oder Abkürzungen, deren Bedeutung in Bezug auf den Benutzer die gleiche ist wie ihre Wirkung als Kommando, d.h. im Idealfall ist ein Programm ein kleiner Aufsatz, den man sich durchlesen und sofort verstehen kann. So wird ein Programm nicht zur „Bitpfriemelei“, sondern ist lesbar und abänderbar; Voraussetzungen also zur erfolgreichen Programmierung.
Durch dieses Konzept ist nach dem Schreiben eines Programms eine Übersetzung desselben nötig. Diese Arbeit erledigt der Assembler, der so heißt wie die Programmiersprache, die er verarbeitet. Näheres dazu beim Assembleraufruf.

# 19 Werkzeuge

Die Programmier-Tools stellen wichtige Werkzeuge für den Programmierer bereit, Angefangen von der Fehlersuche, über das Verwalten von Versionen, bis hin zum automatisierten bearbeiten von Projekten. Die Verwendung dieser Tools bleibt nicht auf den Programmierer beschränkt, sondern es gibt viele andere Bereiche, in denen diese Hilfsmittel sinnvoll einsetzbar sind.

- `ar, ranlib` Archivbearbeitung
- `nm` Zerlegen von Archiv–Objektdateien
- `ld` Binder, Loader
- `make` Projektbearbeitung
- `sdb, dbx, adb, gdb` Debugger (Entlauser)
- `strip` Entfernen der Debug–Informationen
- `prof` Tool zum Optimieren von Routinen
- `time` Kontrolle des Laufzeitverhaltens
- `rcs` Revision Control System oder `sccs` Source Code Control System

## 19.1 Die Bibliotheksbearbeitung

Aufruf:
*ar <kommando>archivname files*

- Zusammenfassen von mehreren Dateien zu einer Bibliothek
  (in der Regel Objektdateien, jedoch auch für Quelldateien möglich)
  $\Rightarrow$ Platz– und Zeitersparnis und bessere Übersicht

- Einige Parameter von `ar`

  | | |
  |---|---|
  | c | Erstellung einer neuen Bibliothek (create) |
  | x | Auspacken des Bibliotheksinhalts (extract) |
  | t | Anzeigen des Bibliotheksinhalts (type) |
  | d | Löschen der spezifizierten Dateien aus Archief (delete) |
  | m | Verschieben der spezifizierten Dateien ans Ende (move) |
  | p | Drucken der angegebenen Moduln (print) |
  | v | Zusatzangaben zum Bibliotheksinhalt zeigen (verbose) |

- Beispiel: ar cv fasel.a *.o

Aufruf:
*ranlib archivname*
aktualisiert das Inhaltsverzeichnis des Archivs, ohne die Daten selbst zu verändern

## 19.2 Symboltabellenanzeiger nm

Aufruf:
$nm[-gnoprsua][[filename]...$
Dient zum Feststellen, welche Funktionen in einer Bibliothek oder in einem Objektmodul vorhanden sind

- Ausgeben der Symboltabellen von Objekt–und Archivdateien

- Einige Optionen:

  | | |
  |---|---|
  | -a | gibt alle Symboltabellen aus |
  | -g | gibt nur globale Symbole aus |
  | -n | sortiert numerisch und nicht alphabetisch |
  | -u | undefinierte Symbole ausgeben |

- Beispiel: `nm /usr/lib/libc.a`

## 19.3 Der Linker ld

- `ld` bindet Module (Objektdateien) zu lauffähigen Programmen
- C Standard Library z.B. `libc.a, libm.a usw.` enthält Funktionen und Makros für den Programmierer, die ebenfalls eingebunden werden
- Integration von weiteren (nicht Standard) Bibliotheksfunktionen
  z.B. `-lX11` bindet die Bibliothek `libX11.a` ein, wechselselbige sich in `/usr/lib` befinden muß
- Beispiele für Standard Libraries (`lib*.a`)
  libc.a libcposix.a libm.a
- Shared Libraries (`lib*_s.a`)
- Eigene Bibliotheken mit einbinden als Ergebnis von `ar`
  z.B. `-lfasel` bindet die Bibliothek libfasel.a ein, falls sie im spezifizierten Pfad liegt – sonst ist auch folgendes möglich: Vollständige Angabe des Bibliotheknamens z.B. `./libfasel.a`

## 19.4 Das make–Kommando

Das `make`- Kommando dient zum Steuern und aktualisieren von Projekten "jeglicher" Art. Es ist nicht auf Programmierer beschränkt, sondern liefert durch seine Einfachheit in der Handhabung für viele tägliche Aufgaben gute Dienste. Ein Projekt, welches in einem `makefile` spezifiziert wurde, wird auf Aktualität gegenüber anderen, sog. Abhängigkeitsdateien, überpruft. Dies geschieht durch Betrachtung des Zeitpunktes des letzten schreibenden Zugriffs. Ist also beispielsweise eine der Abhängigkeitsdateien verändert worden, so gilt das Projektziel als veraltet und muß aktualisiert werden. Die geforderten Aktionen werden in Kommandozeilen festgeschrieben und gegebenenfalls ausgeführt.

Ein paar Stichworte:

- Bewältigung von Projekten (nicht auf C beschränkt)
- Wartung, Installation, Archivierung...
- Aufruf: make [Optionen] [Makrodefinition] [Aktion/Ziel]
- Definition von Makros
  OBJS= test.o test2.o
  LIBS= -lm -lc
  PROGNAME= test
  LD= cc
- bereits vordefinierte Standard–Makros vorhanden: `CC AS CFLAGS`
- Einführung des Begriffs der Abhängigkeiten
  (Datum des letzten schreibenden Dateizugriffs)
- Abhängigkeiten: z.B. *.o → *.c
  test1.o: test1.c def.h
  test2.o: test2.c math.h

- Ausführen einer Kommandozeile als Folge einer Abhängigkeit
  ${PROGNAME}: ${OBJS}
  <Tabulator> ${LD} -o ${PROGNAME} ${OBJS} ${LIBS}
- Kommentare wie in der `shell` alles hinter dem '#'–Zeichen ist Kommentar
- Standardmäßig wird die Datei `makefile` der Datei `Makefile` vorgezogen.

Beispiel für eine Datei `makefile`, in der als Projekt das Drucken eines textes in Abhängigkeit der auszudruckenden Datei `Druckdatei.ps` festgelegt wurde. Diese ist selbst wiederum von der Datei `Druckdatei.txt` abhängig. Um den Zeitpunkt des letzten Ausdruckens festzulegen, wird die Datei `drucken` verwendet und mit dem `touch`–Kommando der Zeitpunkt des Druckens "aufgeprägt". Ist die zu druckende Datei `Druckdatei.ps` neuer als die Datei `drucken`, so wird zunächst geprüft, ob nicht vielleicht sogar die Textdatei `Druckdatei.txt` neuer ist als `Druckdatei.ps`. Ist dies der Fall, so muss erst die Kommando– Folge zum aktualisieren von `Druckdatei.ps` ausgeführt werden, bevor gedruckt werden kann `printps`.

```
Druckdatei.ps:          Druckdatei.txt
                        txt2ps  Druckdatei.txt
...
                        ...
drucken:            Druckdatei.ps
                        touch  drucken
                        printps  Druckdatei.ps
                        rm  Druckdatei.ps
...
                        ...
```

Einige Optionen von make:

| | |
|---|---|
| -f dateiname | alternatives Makefile `dateiname` |
| | default: makefile bzw. Makefile |
| -n | macht garnichts zeigt an was gemacht würde |
| -k | ohne Abbruch bei Fehler |
| -p | Ausgabe aller Makrodefinitionen |
| -d | Testmodus mit ausführlichen Infos |
| -r | abschalten der internen Abhängigkeitsregelen |
| -s | abschalten der Ausgabe (silent) |
| -t | 'touched' alles, anstelle des 'makens' |

### 19.4.1 Besonderheiten von `make`

Im Folgenden seien:
Ziel_1...Ziel_n die `Ziele` der „linken Seite",
Abhk_1...Abhk_n deren Abhängigkeitsdateien,
Komm_1...Komm_n Kommandos der Kommandozeile

- Die Reihenfolge der Abhängigkeits-Beschreibungen ist frei

- Mehrere Ziele in einer Abhängigkeits–Zeile:

  *Ziel_1 Ziel_2 :* *Abhk_1 Abhk_2*
  *Komm_1*

- Mehrere Kommandozeilen:

  *Ziel_1 :* *Abhk_1 Abhk_2*
  *Komm_1*
  *Komm_2*

- Mehrere Kommandos in einer Kommandozeile:

  *Ziel_1 :* *Abhk_1 Abhk_2*
  *Komm_1 ; Komm_2*

- Ein Ziel in mehreren Abhängigkeits–Zeilen kann nur in `einer` Gruppe Kommandos enthalten, Ausnahme:
  Das entsprechende Ziel muss durch zwei Doppelpunkte gekennzeichnet werden und zwar in `alle` Abhängigkeits–Zeilen!

  *Ziel_1 ::* *Abhk_1*
  *Komm_1*
  *Ziel_1 ::* *Abhk_2*
  *Komm_2*
  ist identisch mit:

  *Ziel_1 ::* *Abhk_1 Abhk_2*
  *Komm_1*
  *Komm_2*

- Mehrere Abhängigkeits–Zeilen:

  *Ziel_1 :* *Abhk_1 Abhk_2*
  *Komm_1*
  *Ziel_1 :* *Abhk_4*
  *Ziel_1 :* *Abhk_3*

  ist identisch mit:

  *Ziel_1 :* *Abhk_1 Abhk_2 Abhk_3 Abhk_4*
  *Komm_1*

- Kommandos können beliebige Shell-Kommandos sein
- Verwendung der Zeichen `>`, `<`, und `|` ist erlaubt
- Builtin-Shell-Kommandos gelten nicht “uber die Kommando-Zeile hinaus!!
  *list:*
  *cd dir*
  *ls*

listet nicht den Inhalt von `dir`, sondern den des aktuellen Verzeichnis ⇒ richtig lautet es folgendermassen:

*list:*

```
cd dir ; ls
```

- Die Shell-Variablen `$* $# $!` gelten nicht innerhalb von `make`
  Ausnahme:
  `$$`-Var liefert den Wert der Variblen Var der Shell.

- Eine `for`-Schleife wird als ein Kommando angesehen, und mußdeshalb an jedem Zeilenende das \-Zeichen enthalten!

  *test:*

```
for     i in *.c ;          do \
        mv $$i $$i.alt; \
        echo                $$i ; \
done
```

### 19.4.2 Beispiel Makefile

```
1  #
2  # Makefile fuer German-LATEX-Dateien
3  #
4  # Hinweis: AIX-Make verlangt TABs am Zeilenanfang, keine Spaces !
5  #
6  .SUFFIXES: .tex .dvi
7  #
8
9  VORTRAG=        vortrag
10 SCRIPT=         script
11
12 diffs:
13         for i in *.tex;\
14         do\
15                 echo $$i;\
16                 diff $$i v1.0/$$i;\
17         done
18
19 VORTRAGSRC=     0_folie.tex     23_folie.tex    411_folie.tex    \
20         4_folie.tex  1_folie.tex     24_folie.tex    41_folie.tex  \
21         99_folie.tex  21_folie.tex    2_folie.tex     42_folie.tex \
22         22_folie.tex    3_folie.tex     43_folie.tex    ${VORTRAG}.tex
23
24 SCRIPTSRC=      A_teil.tex B_teil.tex
25
26 all: ${VORTRAG}.dvi
27
28 clean:
29         rm -f *.dvi *.aux *.log *~
30
31 Anweisung.dvi:          Anweisung.tex
32
33
34 ${SCRIPT}:              ${SCRIPT}.dvi
35                         latex ${SCRIPT}
```

```
36
37 ${VORTRAG}.dvi:          ${VORTRAGSRC}
38                          latex ${VORTRAG}
39
40 ${SCRIPT}.dvi:           ${SCRIPTSRC}
41                          latex ${SCRIPT}
42
43 print${SCRIPT}:
44                          dviprt ${SCRIPT}
45
46 print${VORTRAG}:
47                          dviprt ${VORTRAG}
48
49 view${SCRIPT}:           ${SCRIPTSRC}
50                          xdvi -s 4 ${SCRIPT}
51
52 view${VORTRAG}:          ${VORTRAGSRC}
53                          xdvi -s 4 ${VORTRAG}
54 #
55 .texiddvi:
56         glatex $<
```

## 19.5 Source Level Debugger dbx

Aufruf:

$dbx[-ffcount][-i][-Idir][-k][-kbd]$
$[-Pfd][-r][-sstartup][-srtstartup]$
$[objfile[corefile|process-id]]$

- einige Optionen des Debuggers:

  | | |
  |---|---|
  | -s startup | lese Startup–Skript |
  | -r | sofortiger Programmstart |
  | -k | Kernel Debugging |
  | -I dir | Objektdateien können auch in `dir` liegen |

- vielfältige Navigationsmöglichkeiten s.U.
- verschiedene File Access Commands verfügbar
- verschiedene Shell Funktionen verfügbar

- einige interne Kommandos des dbx:

  $run[args][< infile][> \mid >> outfile]$
  $rerun[args][< infile][> \mid >> outfile]$
  $cont[atsourceline][sigsignal]$
  $trace[infunction][ifcondition]$
  $traceexpressionatsourceline[ifcondition]$
  $tracevariable[infunction][ifcondition]$
  $stopatsourceline[ifcondition]$
  $stopinfunction[ifcondition]$
  $stopvariable[ifcondition]$
  $clear[sourceline]$
  $ignore[signal[, signal]...]$
  $step[n]$
  $next[n]$
  $display[expression[, expression]...]$
  $whatisidentifier$
  $setvariable = expression$
  $clear[sourceline]$
  $help[command]$

# 20 Werkzeuge für die C-Entwicklung

- `lex, yacc` Syntax Analysator und Parser, Programmgeneratoren
- `lint` Syntax–Checker
- `xref, ctags` Aufstellen von Querreferenzen
- `cflow` Aufbau eines Referenzgraphen (C, LEXX, YACC)
- `cb, indent` C–Beautyfier

## 20.1 C–Programm Checker lint

Aufruf:

$/usr/bin/lint[-abcghnpquvxzO][-Dname[= def]]$
$[-Idirectory][-llibrary][-ooutputfile]$
$[-Uname]filename...$

- Sinvoll bei großen Paketen und bei der Portierung von anderen Maschinen
- führt Typprüfungen aus
- kontrolliert Parameterübergabe
- findet unereichbare Funktionsbereiche
- meldet unbenutzte Variablen

- einige Optionen:
  - -a Zuweisungen von `long` an nicht
  - -b nicht erreichbare breaks anzeigen
    `long`–Variablen prüfen
  - -p Portabilitätscheck
  - -n keine Kompatibilitätsprüfung
    gegenüber Standard Bibliothek
  - -x unbenutzte externe Variablen anzeigen

## 20.2 Generierung von Software

Zunehmend dramatisch, wenn ...

- ...Quell–Code bereits auf gleicher Plattform läuft
- ...auf beiden Maschinen gleiches Unix läuft (BSD SVR4)
- ...auf beiden Maschinen güberhaupt Unix läuft und terminalunabhängig programmiert wurde (Verwendung der X11 oder der Curses Bibliotheken
- ...andere bereits Portierungsversuche für unterschiedliche Plattformen unternommen haben
- ...Standards verwendet wurden (K & R, Ansi, Posix )
- ...der Programmierer keine Betriebssystesspezifischen oder Hardwareabhängigen Routinen eingebaut hat
- ...Programme anständig dokumentiert sind

## 20.3 Generierung von Software

- FTP-Server `ftp.th-darmstadt.de`
  viel Public–Domain Software
  Anwendung ähnlich dem `telnet`–Kommando

  ```
  ftp ftp.th-darmstadt.de
  ```
- News-Server `news.th-darmstadt.de`
  Unzählige News–Gruppen z.B. comp.unix.aix, thd.hrz.announces
  Vorraussetzung: vorhandensein eines lokalen Newsreaders (`nn, rn, xrn...`) und Netzanbindung an Server
- Mail-Server `mailserver.th-darmstadt.de`
  "Jeder" mit Internet–Zugang kann national und international electronic-mail verschicken und empfangen (gesonderter Kurs halbjährlich)

  Für Systemadministratoren: HRZ–Sendmail–Konfigurationskit !!!
- monatlicher Unix–Treff in *Darmstadt*: jeden 3. Dienstag im Monat im "Da Nino", Alexanderstr. ab 18:00 Uhr

- monatlicher Unix–Treff in *Frankfurt*: jeden 1. Montag im Monat im Restaurant der Bürgerhaus in Frankfurt Bornheim

# Index


`$@`, 21
`&`, 9
`.profile`, 29
`/etc/profile`, 29
`$*`, 21
`$1..$9`, 21
`$?`, 21
`$#`, 21
`$$`, 21

`alias`, 27
`ar`, 79
Archivierung, *siehe* Backup
`ARGC`, 45
`ARGV`, 45
Assembler, 78
awk
    Beispiele, 44
    Funktionen, 47
    Kontrollstrukturen, 46
    Pattern, 44
    Programmierung, 43
    Systemvariablen, 45
`awk`, 43

Backquotes, 33
Backslash, 32
Backup, 52
`bash`, 19
Binder, *siehe* Linker
Bourne-Shell, 19
`break`, 28, 63
BSD, 2
`bsh`, 19

C, 57
    Beispielprogramm, 76
    Datentypen, 67
    Kontollstrukturen, 58
    Kontrollstrukturen, 59, 62
    Kurzbeschreibung, 58
    Speicherklassen, 65
    Strukturen, 71
    Syntaxprüfung, 85
    Variablen, 67
C-Shell, 19
`case`, 28
`cc`, 57
Compiler
    Beispielaufrufe, 56
    Optionen, 56
`compress`, 53
`continue`, 28, 63, 64
`cpp`, 57
`csh`, 19

`date`, 49
Datei
    Extension, 11
    Filtern von Dateien, 38
    Komprimierung, 51, 53
    Links, *siehe* Links
    Namensregelungen, 11
    Sicherung, *siehe* Backup
    Sortieren von Zeilen, 52
    Statistik, 53
    suchen von Dateien, 49
    Teile von Dateien, 50
    Verweise, *siehe* Links
    Zugriffsrechte, 13
Datum, 49
`dbx`, 84
Debugger, 84
`do`, 61
Dokumentation, *siehe* Hilfe
Double-Quotes, 32
Drucker
    Ansteuerung, 51

`echo`, 26
`egrep`, 49
`eval`, 26
`exec`, 26
`exit`, 26
Expansion
    Filenamen, 24
`expr`, 26

`false`, 28
`fgrep`, 49
`FILENAME`, 45
Filesystem, 10
`find`, 49
`FNR`, 45

`for`, 28, 60
FORTRAN, 78
`FS`, 45
`function`, 27

`grep`, 49

`head`, 50
Header–Dateien, 56
Hilfe
    Manual-Pages, 51
`HOME`, 20

`if`, 27, 58
`IFS`, 20, 23

Kernel, *siehe* Unix–Kernel
`kill -9`, 10
`kill`, 10
Korn-Shell, 19
`ksh`, 19

`ld`, 80
Library, 75, 79
    Systemlibraries, 75
Link
    hard, 50
    symbolischer, 50
Linker, 80
`lint`, 85
`ln`, 50
Login, 7
`LOGNAME`, 20
`lpr`, 51
`ls`, 14

make
    Abhängigkeitsregel, 82
    Kommandozeilen–Optionen, 81
    Makros, 80
    Shellvariablen, 83
    Variablen, 80
`make`, 80, 81
`man`, 51
Metazeichen, 24
Muster, 49
    Reguläre Ausdrcke, *siehe* Reguläre Ausdrücke
    suche von, 49

`NF`, 45
`nm`, 79
`NR`, 45

`OFMT`, 45
`OFS`, 45
`ORS`, 45

`pack`, 51
`passwd`, 7
Passwort, 7
`PATH`, 20
Pattern, 49
    suche von, 34, 49
Patternmatching, 24, 34, 38
`pcat`, 51
Pipes, 23
Präprozessor, 57
`profile`, 29
Programmierung
    C-Tools, 85
    Entwicklung, 55
    Sprachen, 55
    Tools, 78
    Umgebung, 56
Prozeß
    Background, 25
    Hierarchie, 25
    Hintergrund, 10
    Kommandos, 8
    Priorität, 6
`ps`, 8
`PS1`, 20
`PS2`, 20
`pwd`, 14

Quotes
    double, 32
    single, 32
Quoting, 31

`ranlib`, 79
`read`, 26
Reguläre Ausdrcke, 38
Reguläre Ausdrücke, 34
`return`, 27, 64
`RS`, 45

`sed`, 38
`set`, 22
`SHELL`, 20
Shell, 18

Aufgaben, 19
Befehle, 26
Beispiele Quoting, 32
Funktionen, 27
Kontrollstrukturen, 27
Quoting, 31
Typen, 19
Variablen, 19
`sleep`, 9
Sonderzeichen
Maskierung der, 31
`sort`, 52
Startup, 29
stderr, 23
stdin, 23
stdout, 23
Subsystem, 4
Symboltabelle, 79
System V, 2
Systemadministrator, 6
Systemuhrzeit, 49, 53
Systemverwalter, *siehe* Systemadministrator

`tail`, 50
`tar`, 52
`tcsh`, 19
`TERM`, 20
`test`, 28
Textfilter, 38, 50
`time`, 53
`touch`, 14
`true`, 28
`type`, 27

`umask`, 27
`uncompress`, 53
Unix
Entwicklung von, 3
Kernel, 4
`unpack`, 51

Variablen, 19
awk
definierbare Variablen, 45
Systemvariablen, 45
Systemvariablen, 20
Verzeichnis
Zugriffsrechte, 13
Verzeichnisstruktur
Übersicht über, 10
Systemverzeichnis, 12

`wc`, 53
`while`, 28, 60
Wildcards, 24

`zcat`, 53
Zugriffsrechte, *siehe* Datei–Zugriffsrechte