A Fool with a Tool is still a Fool

Überblick über einige wichtige Programmiertools

Klaus Wachtler

10. Juni 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

Dieser Text ist als Arbeitsunterlage für das Programmieren mit *C* und *C++* gedacht, hauptsächlich als Ergänzung zu den beiden Büchern [KW:0 ist false] und [KW:C++]. In letzteren wird die jeweilige Sprache nur behandelt ohne jeglichen Bezug zur Rechnerumgebung; das zum Arbeiten unter gängigen Systemen (mit Schwerpunkt auf unixähnlichen Betriebssystemen wie GNU/Linux) fehlende wird hier nachgeholt.

Dabei werden „moderne" Wege der Programmentwicklung ausgeklammert (diverse integrierte Entwicklungsumgebungen insbesondere). Diese sollten jeweils eine geschlossene Dokumentation beinhalten, so daß eine Beschreibung hier überflüssig ist. Vielmehr geht es in diesem Buch darum, einen Überblick über die wichtigsten „klassischen" Werkzeuge zu geben, mit denen ich vorwiegend arbeite. Diese sind zwar jeweils für sich mehr oder weniger gut dokumentiert; allerdings findet man schwerlich eine Zusammenstellung der nötigen Dokumentationen für alle Werkzeuge, die man sinnvollerweise zur Programmentwicklung benötigt.

Die Auswahl der beschriebenen Werkzeuge ist insofern willkürlich, als ich mich auf das beschränke, was ich für meine Arbeit als sinnvoll erachte, und aus Erfahrung etwas dazu sagen kann. Beispielsweise behandle ich als Editor den EMACS. Sicher gibt es auch andere gute Editoren; aber für mich hat er sich als hervorragend erwiesen. Wer zu anderen Programmen (vi?) etwas beisteuern kann, darf gerne mit Ergänzungen (oder einem allgemein zugänglichen Link) zu einem umfassenderen Werk beitragen.

Dieses Skript entstand aus meinen Unterlagen zu Seminaren und Vorlesungen an der Fachhochschule Hannover im Fachbereich Informatik[1].

Form und Inhalt sind:

Breidingstr. 17
29614 Soltau
Tel.: 05191/70271
email: AnyWare@Wachtler.de
http://www.wachtler.de

Für Verbesserungsvorschläge, Fehlerkorrekturen und ähnliches bin ich jederzeit dankbar.

Für weitere Kopien des Buches im jeweils aktuellen Stand bitte ebenfalls bei obiger Adresse nachfragen.

Am Ende des Buches befindet sich ein Stichwortverzeichnis.

Der Text wurde mit TEX und LATEX gesetzt.

Voraussetzungen:

Vorausgesetzt werden:

- Grundlegende Programmierkenntnisse
- Die Programmiersprachen *C* und/oder *C++*

Wenn diese Voraussetzungen fehlen, ist es sinnvoll zuerst weitere Grundlagenliteratur zu konsultieren, beispielsweise [KW:0 ist false].

[1] http://www.inform.fh-hannover.de

Kapitel 2

make

2.1 Allgemeines

Beim Schreiben von Quelltexten hat man früher oder später den Wunsch, den Quelltext in mehrere Module aufzuteilen.

Dann möchte man (um Zeit zu sparen) natürlich nicht bei jeder Änderung in einem Quelltext alles neu kompilieren, sondern man überlegt sich, auf welche Module sich die Änderungen auswirken, kompiliert nur diese Module, und linkt dann alles zusammen.

2.2 Modulweises Kompilieren

Ein Hauptprogramm `h.cpp` und die beiden Module `up.cpp`/`up.h` (mit irgendeinem Unterprogramm `up()`) sowie `db.c`/`db.h` (mit Funktionen für einen Datenbankzugriff) soll geschrieben werden.

Zuerst werden die beiden Module vorgestellt. Das Modul `db` simuliert die „Datenbank“ nur, indem es einen Typ für die Datensätze definiert (`record_t`), und davon einfach ein Feld anlegt. Die einzige Funktion `lesedb()` füllt das Feld ganz stumpf mit lauter gleichen Werten:

```
// Time-stamp: "(21.11.01 20:24) db.cpp [Klaus Wachtler (aw38)]"
//
```

```
// Implementationsdatei (Funktionsdefinitionen) für Modul db

// Damit der Compiler die Deklarationen in db.h mit den Definitionen
// in dieser Datei vergleichen kann, sicherheitshalber die zugehörigen
// Deklarationen einfügen (hier sind der Typ record_t und das Feld
// arr[] deklariert):
#include "db.h"

// Das Feld wird hier auch definiert:
record_t arr[N_RECORDS];

// könnte Werte aus einer Datenbank in das Feld arr[] lesen; setzt
// hier aber nur ein paar fiktive Werte:
void lesedb( void )
{
  for( int i=0; i<N_RECORDS; i++ )
    {
      arr[i].i = 12;
    }
}
```

Die zugehörigen Deklarationen für die Anwender des Moduls stehen in db.h:

```
// Time-stamp: "(21.11.01 20:31) db.h [Klaus Wachtler (aw38)]"
//
// Funktionsdeklarationen für Modul up

// Damit mehrfaches #includen nicht zu Problemen führt:
#ifndef _DB_H_
#define _DB_H_

#define N_RECORDS  (1000)           // soviele Datensätze

typedef struct { int i; } record_t; // Datentyp für einen Record

extern record_t arr[N_RECORDS];     // alle Records in einem Feld

void lesedb( void );

#endif /// ifndef _DB_H_
```

Das Modul up enthält nur ein Unterprogramm, um das Feld von `lesedb()` füllen zu lassen; hier die Implementationsdatei:

```
// Time-stamp: "(21.11.01 20:26) up.cpp [Klaus Wachtler (aw38)]"
//
// Implementationsdatei (Funktionsdefinitionen) für Modul up

// Damit der Compiler die Deklarationen in up.h mit den Definitionen
// in dieser Datei vergleichen kann, sicherheitshalber die zugehörigen
// Deklarationen einfügen:
#include "up.h"

void up( void )
{
  lesedb();  // in Modul db.cpp/db.h
}
```

Und wieder die zugehörigen Deklarationen für die Anwender von `up`:

```
// Time-stamp: "(21.11.01 20:27) up.h [Klaus Wachtler (aw38)]"
//
// Funktionsdeklarationen für Modul up

// Damit mehrfaches #includen nicht zu Problemen führt:
#ifndef _UP_H_
#define _UP_H_

// Deklarationen für Modul db, weil up() eine der db-Funktionen
// benötigt:
#include "db.h"

void up( void );

#endif /// ifndef _UP_H_
```

Das Hauptprogramm benötigt Deklarationen aus beiden Modulen (weil es erst `up()` aufruft, sowie das Feld `arr[]` verwendet), und bindet deshalb auch beide Headerdateien ein:

```
// Time-stamp: "(21.11.01 20:31) haupt.cpp [Klaus Wachtler (aw38)]"
//
```

```
// Hauptprogramm eines kleinen Beispielprojekts

#include <iostream>

using namespace std;

#include "up.h"  // Deklarationen für Modul up
#include "db.h"  // Deklarationen für Modul db

int main( int nargs, char **args )
{

  up();      // in Modul up.cpp/up.h

  cout << " Wert = "
       << arr[0].i // arr ist in db.h deklariert
       << endl;

  return 0;
}
```

(Daß dadurch versehentlich `db.h` doppelt eingebunden wird, soll nicht weiter stören; hier greift der Mechanismus mit `#ifndef _DB_H_`....)

Theoretisch könnte man jetzt das fertige Programm mit einem einzigen Compiler- und Linkeraufruf übersetzen und linken lassen:

```
g++ -Wall haupt.cpp up.cpp db.cpp -o haupt
```

Das ist zwar soweit korrekt, aber bei größeren Projekten nicht wünschenswert. Wenn es nicht wie hier nur um wenige Zeilen Quelltext geht, sondern um Dutzende Quelldateien mit jeweils einigen Tausend Zeilen, dann benötigt das Übersetzen doch eine gewisse Zeit. Wenn man aber nur an wenigen Stellen etwas geändert hat, wäre es interessant, auch nur die zugehörigen Module zu kompilieren und dann alles zu linken.

Dazu übersetzt man beim ersten Mal nicht mit dem einen obigen Aufruf, sondern Modul für Modul:

```
g++ -Wall -c haupt.cpp
```

```
g++ -Wall -c up.cpp
g++ -Wall -c db.cpp
```

Dadurch werden die beiden Module und das Hauptprogramm nur jeweils einzeln kompiliert (wegen der Option -c), und in je einer Objektdatei abgelegt (haupt.o, up.o, db.o).

Erst anschließend fügt man diese Objektdateien mit dem Linker zu einem ausführbaren Programm zusammen:

```
g++ -Wall haupt.o up.o db.o -o haupt
```

Insofern hat man bisher nichts gewonnen; mit mehr Kommandos hat man das selbe Ergebnis wie in einfachen Fall zu Beginn. Der Vorteil ist aber, daß die Objektdateien (haupt.o, up.o, db.o) nach dem Linken noch erhalten bleiben, und bei Bedarf wieder verwendet werden können.

Das ist der Fall, wenn man jetzt an einzelnen Stellen im Quelltext etwas ändert.

Dafür ein paar einfache Beispiele:

- Hat man am Hauptprogramm haupt.cpp etwas geändert, dann muß man nur diese eine Datei kompilieren; die beiden anderen Objektdateien sind auf dem aktuellen Stand und deshalb müssen db.cpp und up.cpp nicht kompiliert werden:

  ```
  g++ -Wall -c haupt.cpp
  g++ -Wall haupt.o up.o db.o -o haupt
  ```

- Hat man analog etwas in up.cpp geändert, dann muß halt up.cpp kompiliert werden, und alles gelinkt:

  ```
  g++ -Wall -c up.cpp
  g++ -Wall haupt.o up.o db.o -o haupt
  ```

- Entsprechend bei einer Änderung in db.cpp:

  ```
  g++ -Wall -c db.cpp
  g++ -Wall haupt.o up.o db.o -o haupt
  ```

- Wenn mehrere Quelltexte geändert wurden, muß entsprechend mehr kompiliert werden.

Insoweit ist die Lage recht übersichtlich. Was passiert aber, wenn man an den Headerdateien Änderungen vornimmt?

- eine Änderung an up.c erfordert ein Neukompilieren von up.cpp und von haupt.cpp, weil diese beiden Quelltexte up.h einbinden.
- eine Änderung an db.h bedingt sogar das Kompilieren aller drei Module.

2.3 Aufstellen der Abhängigkeiten

Egal wie reizvoll das teilweise Übersetzen für die Praxis auch ist: bei jeder Änderung an den Quelltexten muß sorgfältig entschieden werden, was veraltet ist und damit neu erzeugt werden muß.

Bei größeren Projekten ist diese Entscheidung nicht mehr zuverlässig manuell zu erledigen (geschweige denn komfortabel).

Um die Abhängigkeiten der Dateien untereinander zu veranschaulichen, kann man sie wie in Bild 2.1 darstellen.

In dieser Darstellung sind die weiter rechts stehenden Dateien von den weiter links stehenden abhängig, und können bei Bedarf mit den an die Pfeile geschriebenen Kommandos neu erzeugt werden. Wenn jetzt also eine Datei neuer ist als eine auf ihrem Weg entlang der Pfeile zu der zu aktualisierenden Zieldatei, dann müssen alle auf dem Weg an die Pfeile geschriebenen Kommandos ausgeführt werden.

Anhand einer solchen Darstellung könnte man leichter entscheiden, welche Kommandos nach Änderungen am Quelltext nötig sind, um die Zieldatei (beispielsweise haupt) zu aktualisieren.

Je nachdem wie kompliziert das Projekt wird, können neben Kompilieren und Linken von Programmteilen natürlich auch noch beliebige andere Dateien mit ihren Abhängigkeiten und zugehörigen Kommandos in einer solchen Darstellung untergebracht werden, beispielsweise das Generieren von Skripten, um Datenbanktabellen anzulegen, Erzeugen und Drucken von Programmdokumentation,

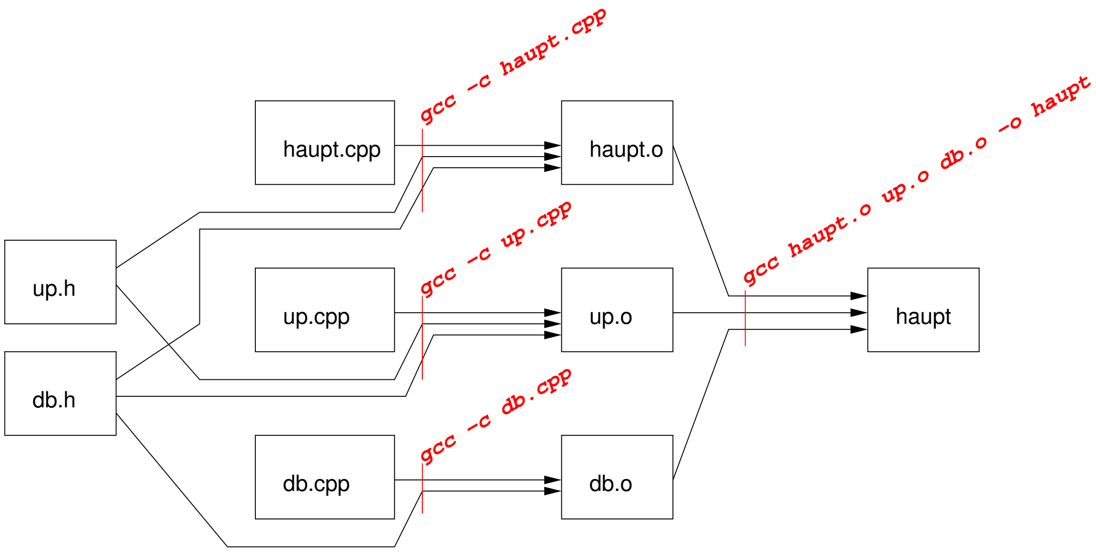

Abbildung 2.1: Abhängigkeiten Beispielprojekt

Versenden von emails mit Programmänderungen an den Projektleiter, Versenden von Beispielergebnissen an den Kunden, und so weiter.

2.4 Verwendung von make

Die im letzten Abschnitt gezeigte grafische Darstellung von Abhängigkeiten und Kommandos kann hilfreich sein, um im Einzelfall zu entscheiden, welche Kommandos zum Aktualisieren einer Zieldatei nötig sind.

Da diese Arbeit aber –falls man erst einmal die Abhängigkeiten wie beschrieben aufgestellt hat– vollkommen mechanisch erfolgt (anhand des letzten Änderungsdatums aller Dateien die nötigen Wege der Grafik entlanggehen, und die an den Pfeilen stehenden Kommandos ausführen), kann man sie auch einem Programm überlassen.

Dieses Programm existiert wohl auf jedem Unix- und GNU/Linux-System, und heißt make[1].

Leider kann make mit einem gezeichneten Diagramm nicht viel anfangen; vielmehr liest es eine Datei, in der man in Textform exakt die Informationen der

[1]Wenn auf einem Windowssystem VC++ installiert ist, dann existiert dort ein sehr ähnliches Programme namens nmake.

Grafik eintragen muß. Diese Datei heißt in aller Regel makefile. Beim Start von make wird die Datei mit dem Namen makefile gelesen, daraus die Abhängigkeiten und die zugehörigen Kommandos gelesen, und –je nach dem letzten Änderungsdatum aller beteiligten Dateien– werden Kommandos ganz, teilweise oder gar nicht ausgeführt. Dabei zählen nicht existierende Dateien als „sehr alt“, so daß sie bei Bedarf neu erzeugt werden.

Um jetzt diese Abhängigkeiten und die Kommandos eintragen zu können, muß man eine besondere Syntax beachten, die im Folgenden (etwas vereinfacht) vorgestellt wird.

Die meisten Bestandteile des makefile sind in je einer *logischen Zeile* untergebracht. Eine solche logische Zeile entspricht meistens einer tatsächlichen Zeile der Datei. Allerdings kann man eine lange logische Zeile auf mehrere tatsächliche Zeilen verteilen, wenn sie sonst zu lang wird (identisch zu *C/C++*). Dazu trennt man die Zeile einfach auf, und beendet jede tatsächliche Zeile außer der letzten mit einem *back slash*(\). Nach dem \ darf außer dem Zeilenendezeichen (LF, *line feed*) nichts mehr kommen, auch kein Leerzeichen oder Tabulator!

Die Datei besteht hauptsächlich aus:

- **Kommentar**: alles, was in einer Zeile nach einem #-Zeichen folgt (falls das #-Zeichen nicht gerade in einem String vorkommt), ist Kommentar und wird von make ignoriert.
- **Leerzeilen** werden großteils ignoriert, sind aber an einigen Stellen zum Trennen von einzelnen Abschnitten nötig.
- **Makrodefinitionen**: Man kann innerhalb eines makefile Makros definieren. Ein Makro besteht aus einem Namen (der meistens groß geschrieben wird), und einem Inhalt (der auch leer sein kann). Die Definition beginnt an einer beliebigen Stelle mit dem Namen, nach Belieben Leerzeichen, dann einem Gleichheitszeichen, wieder beliebig Leerzeichen, und dann (falls das Makro nicht leer sein soll) bis zum Rest der (logischen) Zeile der Inhalt des Makros.

  An jeder anderen Stelle kann man ein solches Makro expandieren lassen, indem man den Makronamen geklammert und mit einem $ vorweg schreibt.

  Beispiel:

  ```
  CC = gcc
  CFLAGS = -Wall -c
  ```

  definiert zwei Makros mit den Namen CC und CFLAGS. Damit kann man irgendwo im makefile schreiben:

```
$(CC) $(CFLAGS)
```

; dies wird dann zu:

```
gcc -Wall -c
```

expandiert.

Für den Namen des Makros darf man alle Zeichen verwenden außer *white space* (zumindest nicht am Anfang oder Ende des Namens), `#`, `=` sowie `:`. Groß-/Kleinschreibung wird unterschieden.

Anstelle der runden Klammern sind auch geschweifte Klammern erlaubt.

Wenn der Name des Makros nur aus einem Zeichen besteht (und zwar nicht aus `$`, `(` und `{`), dann kann man die Klammern auch ganz weglassen. `$(A)` ist gleichbedeutend mit `$A`.

Ein Dollarzeichen `$` an allen anderen Stellen im makefile muß zur Unterscheidung als `$$` geschrieben werden (beispielsweise in Dateinamen).

- **explizite Regeln**: eine *explizite Regel* gibt für genau eine Datei (*target* genannt) an, von welchen anderen Dateien (*prerequisites* genannt) sie abhängt, und mit welchem/n Kommando(s) sie aktualisiert werden kann.

  Im obigen Beispiel würde eine solche explizite Regel sinngemäß heißen: *Die Datei* haupt.o *hängt ab von* haupt.cpp*,* up.h *und* db.h*, und kann mit dem Kommando* `gcc -c haupt.cpp` *aktualisiert werden.*

  Die für make lesbare Form dieser Regel ist allerdings etwas anders:

```
haupt.o : haupt.cpp up.h db.h
	gcc -c haupt.cpp
```

  In der ersten (logischen) Zeile steht zuerst ein Target (oder mehrere durch Leerzeichen getrennt). Nach beliebigen Leerzeichen folgt ein Doppelpunkt, dann eins oder mehrere Leerzeichen, und dann alle *prerequisites*, also alle Dateien, von denen das Target abhängt (voneinander mit mindestens je einem Leerzeichen getrennt).

  Nach dieser Zeile mit den Abhängigkeiten folgen eines oder mehrere Kommandos (jeweils in einer logischen Zeile), um das Target auf den aktuellen Stand zu bringen. Bei einer Objektdatei, die von *C*- oder *C++*-Quelltexten abhängt, ist das meist der entsprechende Compileraufruf. Eine solche Zeile mit einem Kommando muß immer mit einem Tabulatorzeichen beginnen; Leerzeichen stattdessen sind nicht zulässig (außer bei dem nmake von Microsoft).

Nach dem Tabulatorzeichen, aber vor dem eigentlichen Kommando darf noch das Zeichen `@` und/oder ein `-` eingefügt werden. Mit einem `@` erreicht man, daß das von `make` ausgeführte Kommando nicht zur Standardausgabe geschrieben wird. Mit einem `-` wird verhindert, daß `make` sich mit einer Fehlermeldung beendet, wenn das aufgerufene Programm nicht den Wert 0 liefert. Programme sollten ja im Fehlerfall einen Wert ungleich 0 liefern, und im Erfolgsfall eine 0 an den Aufrufer zurückgeben. `make` verwendet diesen Rückgabewert üblicherweise, um zu entscheiden, ob die restlichen Kommandos zur Erzeugung des Targets überhaupt noch Sinn machen (wozu sollte man den Linker aufrufen wollen, wenn ein Quelltext nicht kompiliert werden kann?).

Wenn mehrere Kommandos nötig sind, um das Target zu erzeugen, folgen diese Kommandos jeweils einzeln auf einer (logischen) Zeile.

Jedes Kommando wird in einer eigenen Shell abgearbeitet. Deshalb ist es sinnlos, in einem Kommando das Verzeichnis zu wechseln, um in einem weiteren Kommando dann in dem neuen Verzeichnis beispielsweise etwas zu kompilieren.

Anstatt:

```
dir/haupt.o : dir/haupt.cpp dir/up.h dir/db.h
        cd dir
        gcc -c haupt.cpp
```

ist es also nötig zu schreiben:

```
dir/haupt.o : dir/haupt.cpp dir/up.h dir/db.h
        cd dir; gcc -c haupt.cpp
```

Dadurch läuft das gesamte Kommando `cd dir; gcc -c haupt.cpp` in einer gemeinsamen Shell, und der Verzeichniswechsel mit `cd dir` wirkt sich auch auf den Compilerlauf aus.

In einem solchen Kommando kann man alles schreiben, was die aufgerufene Shell als Kommando verträgt. Es können hier also entsprechende `for`– `if`–, `while`–Anweisungen geschrieben werden, oder was einem als Shellkommando alles einfällt.

- **implizite Regeln**: Eine implizite Regel enthält nicht ein konkretes Target, das zu aktualisieren wäre, sondern ist als Schablone zu betrachten, wie aus einer Datei mit einer bestimmten Endung eine Datei mit einer anderen Endung zu erzeugen ist; beispielsweise wie man aus einer *irgendwas*.`cpp` eine *irgendwas*.`o` erzeugt werden kann.

Eine implizite Regel beginnt mit der Endung (inklusive dem Punkt) der Abhängigkeit (beispielsweise `.cpp`), direkt gefolgt von der Endung der zu erzeugenden Datei, wie etwa `.o`. Nach einer beliebigen Folge von Leerzeichen kommt ein Doppelpunkt. Abhängigkeiten wie bei einer expliziten Regel werden hier kaum vorkommen.

In einer oder mehreren folgenden logischen Zeilen werden dann die Kommandos geschrieben, die zur Erzeugung des Targets nötig sind. In diesen Kommandos kann man spezielle vordefinierte Makros verwenden, um den Namen des Targets, der Abhängigkeiten und so weiter einzusetzen. Diese werden noch im Folgenden erläutert; um das Beispiel komplett schreiben zu können aber schon im Voraus: der Name der ersten (und in diesem Fall einzigen) zu kompilierenden Datei (also des *prerequisites*) kann in einer Regel als `$<` geschrieben werden.

Beispiel: Um aus einer beliebigen *C++*-Datei eine Objektdatei zu kompilieren, kann man die folgende Regel verwenden:

```
.cpp.o:
	g++ -Wall -c $<
```

Dabei wird `$<` durch den Namen der ersten (und damit der einzigen) Abhängigkeit ersetzt, also durch *irgendwas*`.cpp`.

Auch hier muß vor dem Kommando exakt ein Tabulatorzeichen stehen.

Eine implizite Regel wird von make nur verwendet, wenn für eine zu erzeugende Datei keine explizite Regel vorhanden ist.

Beim Start von make ist bereits ein Satz von impliziten Regeln vorhanden. Diese kann man sich neben anderen Voreinstellungen mit: `make -p -f /dev/null` oder
`make -qp`
anzeigen lassen.

Wenn man dann noch beachtet, daß man zwar nicht innerhalb von Regeln, aber zwischen selbige Leerzeilen einstreuen kann, um die Datei etwas übersichtlicher zu machen, dann kann man für das obige Beispiel aus drei Quelltexten folgendes makefile schreiben:

```
# Time-stamp: "(25.11.01 09:00) makefile [Klaus Wachtler (aw38)]"
# makefile für ein kleines Projekt aus haupt.cpp, up.cpp/.h,
# sowie db.cpp/.h.

haupt: haupt.o up.o db.o
```

```
        g++ haupt.o up.o db.o -o haupt

haupt.o: haupt.cpp up.h db.h
        g++ -c haupt.cpp

up.o: up.cpp up.h db.h
        g++ -c up.cpp

db.o: db.cpp db.h
        g++ -c db.cpp
```

Durch die Verwendung einer impliziten Regel kann man die Datei etwas vereinfachen:

```
haupt: haupt.o up.o db.o
        g++ haupt.o up.o db.o -o haupt

haupt.o: haupt.cpp up.h db.h

up.o: up.cpp up.h db.h

db.o: db.cpp db.h

-cpp.o:
        g++ -c $<
```

(Bitte beachten: Vor den Kommandos `g++`... stehen keine Leerzeichen, sondern jeweils ein Tabulator!)

Mit dem Aufruf:
`make`
wird dann das Programm make gestartet. Dieses liest die Datei makefile und findet als erstes *target* die Datei haupt.

Da diese Datei laut makefile von haupt.o, up.o sowie von db.o abhängt, prüft make als nächstes, ob diese drei Dateien existieren und auf dem neuesten Stand sind, also neuer als ihre eigenen *prerequisites*. Gegebenenfalls werden sie aktualisiert mit den angegebenen Kommandos, und dann wird –falls nötig– haupt neu gelinkt.

Davon abgesehen, daß make das erste gefundene Target zu aktualisieren versucht, ist die Reihenfolge der Regeln im makefile gleichgültig.

Wenn man nicht das erste im makefile aufgeführte target aktualisiert haben möchte, gibt man einfach das gewünschte Target (oder mehrere) als Argumente an:

```
make up.o db.o
```

würde die Dateien up.o und db.o aktualisieren, ohne sich um haupt zu kümmern.

Falls man den Namen makefile nicht verwenden möchte, kann man beim Aufruf von make mit der Option `-f` einen anderen Namen vorgeben.

2.4.1 Verwendung von Makros

Eine Möglichkeit, das makefile vielseitiger zu machen und leichter pflegen zu können, ist die geschickte Verwendung von Makros.

Beispielsweise ist es eleganter, nur an einer Stelle im makefile festzulegen, wie der Compiler heißt, und mit welchen Optionen er zum Kompilieren aufgerufen werden soll.

Gerade für solche allgemeinen Fälle gibt es bereits vordefinierte Makros, die man ohne Neudefinieren verwenden kann, oder mit eigenem Inhalt überschrieben kann.

Einige vom GNU-make vordefinierten Variablen und ihre Standardwerte sind:

`AR`	Archivprogramm; Standard: `ar`.
`ARFLAGS`	Weitere Programmargumente dafür; Standard: `rv`.
`AS`	Assembler; Standard: `as`.
`ASFLAGS`	Weitere Programmargumente dafür (für `.s` or `.S` Dateien).
`CC`	Compiler für *C*-Quelltexte; Standard: `cc`.
`CFLAGS`	Weitere Programmargumente dafür.
`CXX`	Compiler für *C++*; Standard: `g++`.
`CXXFLAGS`	Weitere Programmargumente dafür.
`CO`	extrahieren einer RCS-Datei; Standard: `co`.
`COFLAGS`	Weitere Programmargumente dafür.
`CPP`	Aufruf des *C*-Präprozessors; Standard: `$(CC) -E`.
`CPPFLAGS`	Weitere Programmargumente dafür.
`FC`	Compiler für Fortran- und Ratfor; Standard: `f77`.
`FFLAGS`	Weitere Programmargumente dafür.
`RFLAGS`	Weitere Programmargumente dafür (wenn Ratfor kompiliert wird).
`LDFLAGS`	Weitere Programmargumente für alle Compiler, wenn sie zum Linken aufgerufen werden.
`GET`	Programm, um eine SCCS-Datei zu extrahieren; Standard: `get`.
`GFLAGS`	Weitere Programmargumente dafür.
`LEX`	Konvertieren einer Lex-Grammatik nach C oder Ratfor; Standard: `lex`.
`LFLAGS`	Weitere Programmargumente dafür.
`PC`	Compiler für Pascal; Standard: `pc`.
`PFLAGS`	Weitere Programmargumente dafür.
`YACC`	Konvertieren einer Yacc-Grammatik nach C; Standard: `yacc`.
`YFLAGS`	Weitere Programmargumente dafür.
`YACCR`	Konvertieren einer Yacc-Grammatik nach Ratfor; Standard: `yacc -r`.
`MAKEINFO`	Konvertieren einer Texinfo-Datei nach Info; Standard: `makeinfo`.
`TEX`	Programm zum Konvertieren von TEX nach DVI; Standard: `tex`.
`TEXI2DVI`	Programm zum Konvertieren von TeXinfo nach DVI; Standard: `texi2dvi`.
`WEAVE`	Programm zum Konvertieren von Web nach TeX; Standard: `weave`.
`CWEAVE`	Programm zum Konvertieren von C Web nach TeX; Standard: `cweave`.
`TANGLE`	Programm zum Konvertieren von Web nach Pascal; Standard: `tangle`.
`CTANGLE`	Programm zum Konvertieren von C Web nach C; Standard: `ctangle`.
`RM`	Löschkommando; Standard: `rm -f`.

Weiterhin können in impliziten ebenso wie in expliziten Regeln Makros verwendet werden, die zu den jeweiligen target- und prerequisite-Namen expandieren:

`$*`	„stem“; das ist der gemeinsame Teil des Namens von Target und prerequisite (beispielsweise `haupt` für `haupt.cpp` und `haupt.o`)
`$(*D)`	Der Teil des Namens in `$*`, der das Verzeichnis bildet (ohne `/`)
`$(*F)`	Der eigentliche Dateiname in `$*` ohne das Verzeichnis
`$@`	Das Target einer Regel
`$(@D)`	Der Teil des Namens in `$@`, der das Verzeichnis bildet (ohne `/`)
`$(@F)`	Der eigentliche Dateiname in `$@` ohne das Verzeichnis
`$%`	Das Target einer Regel
`$(%D)`	Der Teil des Namens in `$%`, der das Verzeichnis bildet (ohne `/`)
`$(%F)`	Der eigentliche Dateiname in `$%` ohne das Verzeichnis
`$<`	Der Name der ersten Abhängigkeit (prerequisite)
`$(<D)`	Der Teil des Namens in `$<`, der das Verzeichnis bildet (ohne `/`)
`$(<F)`	Der eigentliche Dateiname in `$<` ohne das Verzeichnis
`$?`	Alle prerequisites, die neuer sind als das Target
`$(?D)`	Liste der Teile der Namen in `$?`, die das jeweilige Verzeichnis bilden (ohne `/`)
`$(?F)`	Liste der Dateinamen in `$?` ohne die Verzeichnisse
`$^`	Alle prerequisites, mit Leerzeichen getrennt (mehrfach vorkommende tauchen hier nur einmal auf)
`$(^D)`	Liste der Verzeichnisse der Namen in `$^` (ohne `/`)
`$(^F)`	Liste der Dateinamen in `$^` ohne die Verzeichnisse
`$+`	Alle prerequisites, mit Leerzeichen getrennt (mehrfach vorkommende tauchen gegebenenfalls mehrfach auf)

Damit könnte man das makefile auch so schreiben:

```
CXX = g++               # der zu verwendende C++-Compiler
CXXFLAGS = -c           # Optionen zum Kompilieren von C++

CC = gcc                # Compiler für C und gleichzeitig Linker

haupt: haupt.o up.o db.o
        $(CXX) $^ -o $@

haupt.o: haupt.cpp up.h db.h
        $(CXX) $(CFLAGS) $<

up.o: up.cpp up.h db.h
```

```
        $(CXX) $(CFLAGS) $<

db.o: db.cpp db.h
        $(CXX) $(CFLAGS) $<
```

, beziehungsweise mit einer impliziten Regel:

```
CXX = g++               # der zu verwendende C++-Compiler
CXXFLAGS = -c           # Optionen zum Kompilieren von C++

CC = gcc                # Compiler für C und gleichzeitig Linker

haupt: haupt.o up.o db.o
        $(CXX) $^ -o $@

haupt.o: haupt.cpp up.h db.h

up.o: up.cpp up.h db.h

db.o: db.cpp db.h

.c.o:
        $(CXX) $(CFLAGS) $<
```

Der Vorteil ist bei so wenigen Zeilen nicht offensichtlich; aber bei größeren Projekten kann man so mit weniger Änderungen an der Datei (und dementsprechend weniger Fehlern) das makefile pflegen, und leichter Teile eines makefile in einem anderen weiter verwenden.

Neben der impliziten Regel und einigen Makrodefinitionen enthält die Datei nur noch Abhängigkeiten, sowie das Kommando zum Linken.

Im folgenden Abschnitt wird gezeigt, wie man das Erstellen der Abhängigkeiten automatisieren kann.

2.4.2 Automatisches Erzeugen der Abhängigkeiten mit gcc/g++

Als Ausgangspunkt der folgenden Gedanken dient das kleine makefile aus dem letzten Abschnitt:

```
CXX = g++              # der zu verwendende C++-Compiler
CXXFLAGS = -c          # Optionen zum Kompilieren von C++

CC = gcc               # Compiler für C und gleichzeitig Linker

haupt: haupt.o up.o db.o
        $(CXX) $^ -o $@

haupt.o: haupt.cpp up.h db.h

up.o: up.cpp up.h db.h

db.o: db.cpp db.h

.c.o:
        $(CXX) $(CFLAGS) $<
```

Das Linkkommando ist wirklich projektspezifisch, und kann beim Erstellen des makefile leicht festgelegt werden; es wird sich selten ändern, und eine solche Änderung ist auch nicht fehlerkritisch.

Alle anderen projektspezifischen Angelegenheiten (außer den Abhängigkeiten der Quelltexte) können mit Makrodefinitionen so angelegt werden, daß sie leicht wartbar sind.

Der einzige fehlerkritische Punkt besteht in den Abhängigkeiten der Quelltexte untereinander. Diese zu finden ist intellektuell nicht sehr anspruchsvoll: eine *irgendwas*.o hängt von *irgendwas*.cpp ab; weiterhin muß man den Quelltext nach `#include "..."`-Anweisungen durchsuchen, alle so gefundenen Dateinamen als weitere *prerequisites* vermerken, sowie die gefundenen Includedateien rekursiv wiederum nach `#include "..."`-Anweisungen durchsuchen.

Auch wenn diese Arbeit prinzipiell nicht schwierig ist, so ist sie doch ab einer gewissen Projektgröße mühsam und fehlerträchtig, zumal man bei jeder Änderung an `#include "..."`-Anweisungen das makefile sorgfältig korrigieren muß.

Da man eine solche mühsame, aber eigentlich nicht zu schwierige Aufgabe am besten delegiert, stellt sich die Frage, wem man das Herausfinden der Abhängigkeiten überlassen könnte. Es drängt sich geradezu auf, ein Programm zu schreiben (unter vernünftigen Systemen vielleicht als Shellskript?), welches die Regeln kennen muß, nach denen der Compiler Includedateien sucht, und dementsprechend die Abhängigkeiten liefert.

Wer mit einem GNU-System arbeiten darf, hat auch hier Glück: Der gcc beziehungsweise g++ kennen die Optionen `-M` und `-MM`. Damit aufgerufen kompilieren sie die angegebenen Quelltexte nicht, sondern liefern nur zur Standardausgabe die Abhängigkeiten, praktischerweise gleich im passenden Format für das makefile.

Die beiden Varianten unterscheiden sich darin, daß `-M` alle Includedateien auswertet, während `-MM` nur die mit `"..."` eingebundenen Dateien durchsucht (nicht die mit `<...>` eingebundenen).

Beispielaufruf:

```
klaus@aw38:~/skript_cpp_20011124/module > gcc -MM *.cpp
db.o: db.cpp db.h
haupt.o: haupt.cpp up.h db.h
up.o: up.cpp up.h db.h
```

Diese Ausgabe könnte man jetzt manuell in das makefile übernehmen; eleganter ist es aber, die Abhängigkeiten in einer eigenen Datei stehen zu lassen (beispielsweise dep.make) und stattdessen in das makefile eine `include`-Anweisung zu schreiben. Dadurch wird der Inhalt der angegebenen Datei beim Lesen des makefile mit ausgewertet. Der Vorteil dieser Vorgehensweise liegt darin, daß man jetzt eine eigene Datei für die Abhängigkeiten hat, die man als Target im Makefile angeben kann:

```
CXX = g++               # der zu verwendende C++-Compiler
CXXFLAGS = -c           # Optionen zum Kompilieren von C++
CC = gcc                # Compiler für C und gleichzeitig Linker

haupt: haupt.o up.o db.o
        $(CXX) $^ -o $@

dep.make:
        $(CXX) $(CFLAGS) -MM haupt.cpp up.cpp db.cpp > $@

include dep.make
```

Beim ersten Aufruf existiert die Includedatei noch nicht; weil aber eine Regel für das Target `dep.make` vorhanden ist, wird die Datei von make automatisch erzeugt.

Als Lohn der Mühe muß man sich nicht mehr im Detail um die Abhängigkeiten der Quelltexte kümmern, sondern braucht nur nach Änderungen an Includeanweisungen einmalig:
`make dep.make`
aufrufen, um das makefile zu aktualisieren.

2.4.3 Bereinigen eines Projekts: make clean

Manchmal will man nicht nur Dateien erzeugen lassen, sondern ein Projekt auf die nötigsten Dateien reduzieren, die nötig sind, um die Targets zu bauen. Beispielsweise zum Versenden per email oder für ein Backup ist es nicht wünschenswert, alle Objektdateien, fertig kompilierte Programme oder Bibliotheken, vom Editor angelegte Sicherungskopien etc. zu behalten.

Zum Löschen aller überflüssigen oder aus den Quelldateien erzeugbarer Dateien wird üblicherweise in einem makefile ein Target mit dem Namen `clean` geschrieben:

```
clean:
        rm -f haupt
        rm -f haupt.o up.o db.o
        rm -f *.bak *~  # emacs erzeugt *~ als Sicherheitskopie
```

Dann kann man das Arbeitsverzeichnis mit:
`make clean`
auf das Wesentliche reduzieren, um es beispielsweise zu archivieren; danach kann wiederum mit make wieder das Projekt neu gebaut werden.

2.4.4 Erzeugen aller Targets: make all

In vielen makefiles findet sich auch das Gegenstück zu `clean`, nämlich ein Target namens `all`. Dies wird verwendet, um alle relevanten Zieldateien des Projekts zu aktualisieren (Clientprogramme, Serverprogramme, zugehörige Bibliotheken, Dokumentation etc.).

Der Aufruf erfolgt dann mit: `make all`

2.4.5 Verwendung von Textfunktionen

Neben dem Expandieren von selbstdefinierten und den vordefinierten Makros kann man weiterhin Funktionen aufrufen, die zumindest im GNU-`make` eingebaut sind. Dazu schreibt man wie bei Makros `$(...)`; allerdings kommen in die Klammern der Funktionsname und gegebenenfalls nach einem Leerzeichen die Argumente an die Funktion, getrennt mit Kommata.

Die Argumente können wiederum Funktionsaufrufe oder Makroexpansionen enthalten. Allerdings darf in den Argumenten die Klammerung, die für das umgebende Makro verwendet wird (also `{}` oder `()`) nur paarweise auftreten, weil sich `make` sonst bei der Suche nach dem Ende des Funktionsaufrufs verläuft.

Die Funktion wird ausgeführt, und liefert als Ergebnis einen Text, der genau wie eine Makroexpansion in den umgebenden Kontext eingefügt wird. Ein Funktionsaufruf darf ebenso wie eine Makroverwendung sowohl in den Abhängigkeiten als auch in den Kommandos von expliziten und impliziten Regeln auftauchen.

Hier eine Liste der Funktionen:

- `$(subst` *ALT*`,`*NEU*`,`*TEXT*`)` liefert den *TEXT*, aber jedes Vorkommen von *ALT* ist durch *NEU* ersetzt.

  Beispiel:`$(subst Verona,Naddel,Liebe Verona)` liefert `Liebe Naddel`

- `$(patsubst` *ALT*`,`*NEU*`,`*TEXT*`)` liefert den *TEXT*, aber jedes Vorkommen *ALT* ist durch *NEU* ersetzt; dabei kann *ALT* ein Suchmuster sein. Der *TEXT* wird wortweise betrachtet, und die einzelnen Worte tauchen im Ergebnis jeweils mit einem Leerzeichen getrennt auf. In dem Suchmuster steht ein `%` für eine beliebige Folge von Zeichen. In *NEU* darf ebenfalls ein `%` stehen; an dessen Stelle wird dann der zum `%` in *ALT* passende Text eingesetzt.

  Beispiel:`$(patsubst V%a,V_%_a,Liebe Verona ! Liebe Viktoria)` expandiert zu `Liebe V_eron_a ! Liebe V_iktori_a`

  Ein reales Beispiel wäre die Ersetzung von Dateiendungen:
  `$(patsubst %.c,%.o,a.c b.c c.c)` liefert `a.o b.o c.o`.

  Will man ein `%` in `ALT` oder `NEU` haben, dann muß man es mit einem vorangestellten `\` markieren. Ein beabsichtigtes `\` wird dementsprechend als `\\` geschrieben.

- `$(strip` *STRING*`)` entfernt von *STRING* alle führenden und schließenden *white space* (also im Wesentlichen Tabulatoren, Leerzeichen, Zeilenvorschub), und ersetzt innerhalb von *STRING* jede *white space*-Folge durch genau ein Leerzeichen.

- `$(findstring` *SUCHTEXT*`,`*STRING*`)` liefert *SUCHTEXT*, falls *SUCHTEXT* in *STRING* vorkommt; ansonsten einen leeren String.

- `$(filter` *MUSTER1 MUSTER2...*`,`*STRING*`)` liefert alle Worte aus *STRING*, die einem der *MUSTER* entsprechen, und unterdrückt alle anderen Worte.

  Beispiel:
  ```
  ALLEDATEIEN = a.c b.c c.cpp a.h b.h
  ZUKOMPILIERENDEDATEIEN = $(filter %.c %.cpp,$(ALLEDATEIEN))
  ```

- `$(filter-out` *MUSTER1 MUSTER2...*`,`*STRING*`)` liefert alle Worte aus *STRING*, die keinem der *MUSTER* entsprechen, und unterdrückt alle anderen Worte, also genau das Gegenteil von `filter`.

- `$(sort` *LISTE*`)` liefert eine sortierte Liste der Wörter in *LISTE*, wobei mehrfach vorkommende Wörter unterdrückt werden. Im Ergebnis sind die Wörter jeweils durch genau ein Leerzeichen getrennt.

- `$(dir` *LISTE*`)` extrahiert von allen Dateinamen in *LISTE* die Verzeichnisangabe (falls vorhanden) ohne `/`, und liefert diese als Liste zurück. Wenn ein Dateiname keine Verzeichnisangabe hat, dann wird dafür `./` vermutet und geliefert.

  Beispiel: `$(dir /boot/vmlinuz printcap ./a.c)` liefert `/boot ./ ./`

- `$(notdir` *LISTE*`)` extrahiert von allen Dateinamen in *LISTE* die Namen ohne Verzeichnisangabe, und liefert diese als Liste zurück.

  Beispiel: `$(notdir /boot/vmlinuz printcap ./a.c)` liefert `vmlinuz printcap a.c`

- `$(suffix` *LISTE*`)` extrahiert von allen Dateinamen in *LISTE* die Dateiendungen (falls vorhanden), und liefert diese als Liste zurück.

  Beispiel: `$(suffix /boot/vmlinuz printcap ./a.c)` liefert `.c`

- `$(basename` *LISTE*`)` extrahiert von allen Dateinamen in *LISTE* die vollen Namen, aber ohne Dateiendungen (falls vorhanden), und liefert die Namen als Liste zurück.

  Beispiel: `$(basename /boot/vmlinuz printcap ./a.c)` liefert `/boot/vmlinuz printcap ./`

- `$(addsuffix` *SUFFIX*`,`*LISTE*`)` hängt an jeden Namen in *LISTE* das *SUFFIX* an.

  Damit kann das Beispiel-makefile noch wartungsfreundlicher formuliert werden:

```
CXX = g++               # der zu verwendende C++-Compiler
CXXFLAGS = -c           # Optionen zum Kompilieren von C++
CC = gcc                # Compiler für C und gleichzeitig Linker

MODULE = haupt up db

haupt: $(addsuffix .o,$(MODULE)) # expandiert zu: haupt.o up.o db.o
        $(CXX) $^ -o $@

dep.make:
        $(CXX) $(CFLAGS) -MM $(addsuffix .cpp,$(MODULE)) > $@

include dep.make

clean:
        rm -f haupt
        rm -f $(addsuffix .o,$(MODULE))
        rm -f *.bak *~  # emacs erzeugt *~ als Sicherheitskopie
```

Wenn jetzt ein Modul dazukommt, muß nur noch sein Name an `MODULE` angehängt werden. Die restliche Datei bleibt unverändert!

- `$(addprefix` *PREFIX*`,`*LISTE*`)` hängt vor jeden Namen in *LISTE* das *PREFIX*.
- `$(join` *LISTE1*`,`*LISTE2*`)` liefert eine Liste mit den Kombinationen der beiden ersten Wörter aus *LISTE1* und *LISTE2*, den jeweils zweiten Wörtern, usw..

  Beispiel: `$(join db up,.cpp .h)` liefert `db.cpp up.h`
- `$(word` *N*`,`*LISTE*`)` liefert das *N*-te Wort aus *LISTE*; *N* beginnt bei 1. Zu hohe Werte von *N* liefern einen leeren String.
- `$(words` *LISTE*`)` liefert die Anzahl Worte in *LISTE*.
- `$(wordlist` *N1*`,`*N2*`,`*LISTE*`)` liefert aus *LISTE* die Teilliste vom Index *N1* bis *N2*.
- `$(firstword` *LISTE*`)` liefert das erste Wort aus *LISTE*, ist also identisch mit `$(word 1,`*LISTE*`)`
- `$(wildcard` *PATTERN*`)` expandiert zu einer Liste existierender Dateien, die dem *PATTERN* entsprechen (in *PATTERN* werden wild cards wie in der Shell üblich verwendet, also `*`, `?` und `[...]`).

- `$(foreach` *VARIABLE*`,`*LISTE*`,`*VORLAGE*`)` expandiert zuerst *VARIABLE* und *LISTE*. Dann wird für jedes Wort *LISTE* einmal an die Variable mit dem aus *VARIABLE* enstandenen Namen das Wort zugewiesen, und damit *VORLAGE* expandiert. Rückgabe der Funktion ist eine Liste der so entstandenen Expandierungen von *VORLAGE*.

  Beispiel: `$(suffix .c,a b c)` liefert `a.c b.c b.c`, und könnte gleichwertig durch `$(foreach v,a b c,$(v).c)` ersetzt werden.

- `$(if` *BEDINGUNG*`,`*THEN-TEXT*`)`
  `$(if` *BEDINGUNG*`,`*THEN-TEXT*`,`*ELSE-TEXT*`)` entfernt von *BEDINGUNG* führende und schließende *white space* und expandiert den Rest . Falls dies einen nicht-leeren String liefert, dann wird der *THEN-TEXT* expandiert und zurückgeliefert; ansonsten der *ELSE-TEXT* (falls angegeben).

  Es gibt auch noch eine direkte Unterstützung von Fallunterscheidungen; siehe **2.4.6 Fallunterscheidung** auf Seite 30.

- `$(call` *VARIABLE*`,`*P1*`,`*P2*...`)` expandiert die Parameter *P1*, *P2*... und definiert damit die Makros `$(1)`, `$(2)`, ... Zudem erhält `$(0)` den Inhalt von *VARIABLE*. Damit wird dann die *VARIABLE* expandiert (die zweckmäßigerweise `$(1)`, `$(2)`, ... enthalten sollte).

  Beispiel:

  ```
  concat = $1$2
  ...
  $(call concat,abc,def)
  ```

  expandiert zu `abcdef`.

- `$(origin` *VARIABLENNAME*`)` liefert einen String, der angibt, woher der Wert der angegebenen Variable stammt:

  - `undefined`: gar kein Wert
  - `default`: vordefiniert vom Start
  - `environment`: aus Umgebungsvariable und make wurde ohne `-e` aufgerufen
  - `environment override`: aus Umgebungsvariable und make wurde mit `-e` aufgerufen
  - `file`: im makefile definiert
  - `command line`: auf Kommandozeile definiert[2]
  - `override`: aus einer `override`-Direktive

[2]wobei ich nicht weiß, wie man auf der Kommandozeile Werte definieren können soll

  - `automatic`: automatisch gesetzt in einer Regel wie etwa `$*`).

- `$(shell` *KOMMANDO*`)` führt das angegebene *KOMMANDO* aus, und erzeugt aus der Ausgabe eine Liste, indem die einzelne Zeilen mit jeweils einem Leerzeichen aneinandergehängt werden.

  Beispiel: `$(shell find . -name "*.cpp")` liefert eine Liste aller `*`.cpp-Dateien im aktuellen Verzeichnis sowie in allen Unterverzeichnissen.

2.4.6 Fallunterscheidung

TODO...

2.5 Anwendungen von make

Prinzipiell ist die Anwendung von make nicht auf Kompilieren und Linken beschränkt. Von den bereits eingebauten Makrodefinitionen und einigen impliziten Regeln abgesehen, weiß make eigentlich gar nichts über den Sinn der aufgerufenen Kommandos.

Vielmehr ist make ein sehr universelles Werkzeug, mit dessen Hilfe man alle Dateien, die durch irgendwelche Programmläufe aus anderen Dateien entstehen, aktualisieren kann.

Das Erstellen von Programmen ist sicher ein häufiger Anwendungsfall, aber beileibe nicht der einzige.

Die vorliegende Dokumentation beispielsweise ist mit einem üblichen Editor geschrieben (keiner Textverarbeitung), und wird durch Aufrufe der Programme TEX/LATEX, dvips, LATEX2HTML (und weiteren) verarbeitet. Auch hier wird make sinnvollerweise eingesetzt. Die Vorgehensweise ist prinzipiell dem beschriebenen Programmbeispiel sehr ähnlich. Nach einer Änderung der manuell geschriebenen Eingabedateien kann man mit einem make-Aufruf die druckbare Postscriptdatei oder die HTML-Dateien neu erzeugen lassen.

Besseres Beispiel! Tatsächlich wird auch in diesem makefile wieder kompiliert und gelinkt, nämlich die Beispielprogramme. Wenn sich an den Beispielquelltexten etwas ändert, dann wird durch Regeln der folgenden Art:

```
intarr: intarr.cpp
```

```
intarr.aus: intarr
        - ./intarr >intarr.aus 2>&1
```

die zugehörige Beispielausgabe aktualisiert, und als Datei intarr.aus wieder in die Dokumentation eingebunden, in diesem Beispiel für das Programm intarr (siehe **?? Überladen von Operatoren** auf Seite **??**).

Weitere potentielle Anwendungen von make sind automatisches Aktualisieren von Konfigurationsdateien in der Systemverwaltung, fallgesteuertes Backup von Daten, Abgleichen von gespiegelten Daten, und viele weitere.

2.6 Empfehlung des Hauses: EMACS

Der EMACS (**4 EMACS** auf Seite 143) bietet auch beim Schreiben von makefiles ganz nette Gimmicks, die einem das Leben leichter machen.

Ein Problem in den meisten Editoren ist die Frage, ob an einer Stelle, an der man nichts sieht, wirklich nichts steht, oder ob da Leerzeichen oder Tabulatorzeichen stehen. In einem makefile ist das durchaus wichtig.

Den EMACS kann man dazu überreden, nach wilden Regeln (die man weitestgehend abkupfern wird, aber auch selbst vorgeben kann) den Quelltext bunt einzufärben.

Dazu ein Beispiel (einzufügen in die Konfigurationsdatei .emacs im Homeverzeichnis):

```
;...
(defun kw-ergaenzungen-zu-makefile-mode()
  ( font-lock-mode 1 )
  ; ggf. noch weitere Kommandos, die man beim Wechsel in den
  ; 'makefile-mode' ausgeführt haben möchte, z.B.:
  ; ( setq case-fold-search nil )  ; unterscheidet beim Suchen groß/klein
  ; ( setq abbrev-mode t )         ; Abkürzungen werden expandiert
  )

; Die eben definierte Funktion kw-ergaenzungen-zu-makefile-mode soll
; immer beim Wechsel in den 'makefile-mode' ausgeführt werden (für
; den aktuellen buffer):
(add-hook 'makefile-mode-hook 'kw-ergaenzungen-zu-makefile-mode )
```

```
; und zwar möchte ich alle Tabs in einem dreckigen Braun ("Khaki")
; und alle Leerzeichen und Tabs am Zeilenende in einem
; leuchtenden Gelb ("Gold"); dazu ist der font-lock-mode nötig:
(custom-set-faces
 '(my-tab-face            ((((class color)) (:background "Khaki"))) t)
 '(my-trailing-space-face ((((class color)) (:background "Gold"))) t))
(add-hook 'font-lock-mode-hook
          (function
           (lambda ()
             (setq font-lock-keywords
                   (append font-lock-keywords
                           '(("\t+" (0 'my-tab-face t))
                             ("[ \t]+$" (0 'my-trailing-space-face t))))))))
```

Wenn man damit dann ein makefile öffnet, sieht das etwa aus wie in Bild 2.2.

Hier sieht man schön daß vor den Kommandos Tabulatoren stehen (wie es auch sein soll), aber am Ende einer fortgesetzten Zeile nach dem *back slash* (\) versehentlich ein Leerzeichen zuviel eingefügt wurde.

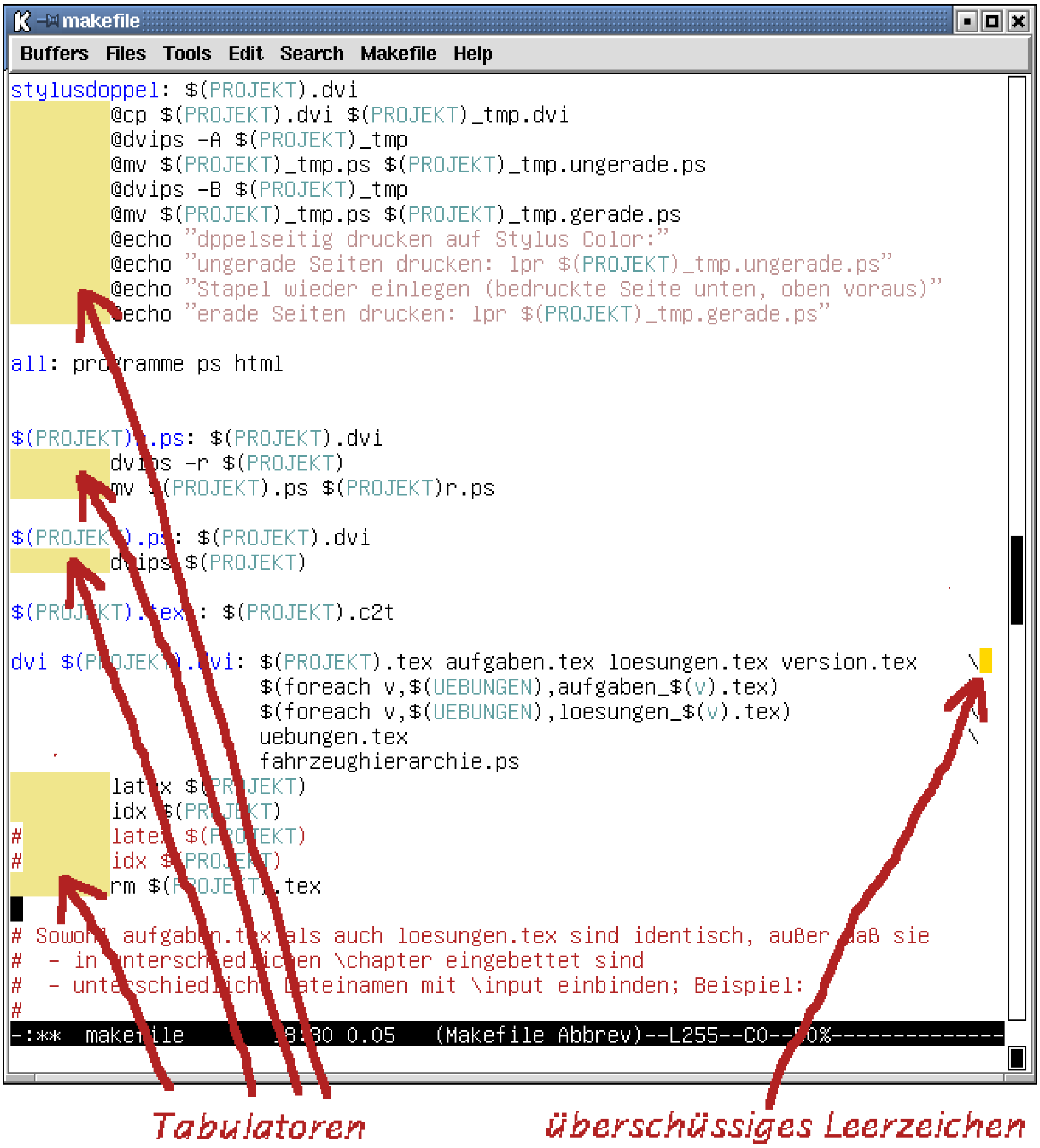

Abbildung 2.2: Beispielfenster `makefile` im EMACS

Kapitel 3

Shell: Der Kommandointerpreter

3.1 Einleitung

3.1.1 Allgemeines

Die üblichen Kommandointerpreter, über die ein Benutzer mit einem Unix-Rechner kommuniziert (`sh`, `csh`, `ksh`, `bash`), werden alleine durch interaktives Arbeiten lange nicht ausgereizt.

Für viele Programmierarbeiten ist eine solche Shell wesentlich praktischer als eine echte Programmierumgebung.

Die Erstellung von Shellprogrammen ist natürlich keine Zauberkunst, und wird hier vorgestellt.

Der vorliegende Text ist als Gerüst einer Vorlesung gedacht. Einerseits ist er für mich eine Gedächtnisstütze, weil ich sonst die Hälfte vergesse, und andererseits soll er den Studenten als Merkhilfe und Arbeitsunterlage dienen.

Mit diesem Skript kann man sicher nicht für alle Zeiten auf tiefergehende Literatur verzichten. Wesentlich detailliertere Informationen wird man beispielsweise in [Herold: Shell] oder [man bash] beziehungsweise in [info bash] zu finden.

Insofern darf man sich nicht zuviel erwarten: als Tischvorlage zur Vorlesung sollte der Text reichen; als einzige Arbeitsunterlage ist er sicher zu schwach, um ausgiebig Shellprogramme zu schreiben.

Trotz der geringen Ansprüche an dieses Werk bin ich für Kommentare aller Art an die folgende Adresse dankbar. Angefangen von Rechtschreibkorrekturen bis zu grundlegenderen Anmerkungen (zuviel Gefasel, Unklarheiten, inhaltliche Fehler...) werde ich jede Kritik mit Fassung tragen.

Klaus Wachtler
Breidingstr. 17
29614 Soltau
Tel. 05191/70 2 71, 0171/45 53 039
Fax: 05191/70 2 72
email: anyware@wachtler.de
http://www.wachtler.de

(Anmerkung: Wenn der Text im Laufe der Zeit einen befriedigenden Stand erreicht hinsichtlich Umfang und Qualität, dann habe ich vor ihn zusammen mit anderen Skripten von mir der GPL zu unterstellen. Dann ist der gesamte Inhalt tatsächlich frei verfügbar.)

Marken und Warenzeichen werden im laufenden Text verwendet, ohne besonders gekennzeichnet zu werden.

Ohne Anspruch auf Vollständigkeit:

- **Linux** ist ein eingetragenes Warenzeichen von Linus Torvalds.
- **Unix** ist ein eingetragenes Warenzeichen von The Open Group[1]

[1] Aus www.netbsd.org: If something looks like a duck, walks like a duck, and quacks like a

- **SuSE** ist ein eingetragenes Warenzeichen der SuSE GmbH in Nürnberg.
- Die Bezeichnung **AnyWare** ist eine eingetragene Marke von Klaus Wachtler.

In einigen der folgenden Beispielen ist zur Verdeutlichung die Benutzereingabe und die erzeugte Programmausgabe farblich hervorgehoben.

3.1.2 Was fehlt noch?

- logische Zeilen mit `\`
- Felder
- der Abschnitt über Signale ist noch nicht gar
- Prozesse (process id, parent process)
- Zugriffsrechte (`drwxr-x---`...)
- environment (Umgebungsvariablen, wie etwa `PATH`, `PS1`, `HOME` ...)
- Quoting von Sonderzeichen (`"..."`, `'...'`)
- die wichtigsten Kommandos der Kommandozeile, cat, cp, ls, find, tar
- alias

Im nächsten größeren Schritt wird dieses Skript Teil einer Kurzanleitung für verschiedene Hilfsprogramme werden; die Shell ist dann nur ein Kapitel darin neben `make` und bei Gelegenheit weiteren.

---

duck, what is it?
The answer, of course, depends on whether or not the name 'duck' is a trademark! If it is, then the closest that something can get, without permission of the owner of the trademark, is 'duck-like.'
'UNIX' is a trademark of The Open Group, and NetBSD has not been branded with that trademark. Therefore, NetBSD is not UNIX. We refer to it as 'UNIX-like' or 'UN*X-like.'

3.1.3 Was ist eine Shell? Was ist ein Shellprogramm?

Eine *Shell* ist unter Unix ein Kommandointerpreter[2]. Man tippt Kommandos ein (wie beispielsweise `cp *.txt /tmp`, `ls /home/meier` etc.) und die Shell versucht dann nach bestem Wissen und Gewissen das zu tun, was man mit dem Kommando gemeint hat.

Wenn man so arbeitet, dann nutzt man die Shell *interaktiv*, weil man jeweils ein Kommando eingibt, und hoffentlich sieht, was man damit angerichtet hat. Dann gibt man das nächste Kommando ein.

Daneben kann ein Unixrechner[3] beziehungsweise eine darauf laufende Shell (wie in den meisten anderen Betriebssystemen auch) im *Stapelbetrieb* (zumindest hieß das früher so, neudeutsch: batch) arbeiten. Dazu schreibt man die Kommandos in eine Datei, und bringt dann eine Shell dazu, ihre Kommandos nicht von der Tastatur zu lesen, sondern aus der Datei (das entspricht einem Stapel Lochkarten, auf dem Karte für Karte die Befehle eingestanzt sind; daher die Bezeichnung *Stapelbetrieb*). Diese Datei (oder der Kartenstapel) ist dann ein *Shellprogramm*. Wem das Wort Shellprogramm zu lang ist, der kann auch *Shellskript*, *Skript*, *shell script* oder *script* dazu sagen.

Die Kommandos, die einem zur Verfügung stehen, kann man in *interne* und *externe Kommandos* unterteilen.

Interne Kommandos sind in die Shell fest eingebaut, externe Kommandos werden nach dem Aufruf erst nachgeladen (entweder als Binärprogramm, oder als Shellprogramm, das in der aktuellen Shell läuft, oder als Shellprogramm, für das erst noch eine weitere Shell nachgeladen werden muß).

Um die Sache nicht zu einfach zu machen, gibt es natürlich nicht eine Shell, sondern mehrere.

Die ursprüngliche Shell, die anfangs auf Unixrechnern als `/bin/sh` zu finden war, ist die sogenannte *Bourne-Shell* (nach ihrem Ziehvater Steve R. Bourne). Sie gilt nach wie vor als Standardshell unter Unix. Ihre Sprache ist wohl die am häufigsten verwendete Programmiersprache der Welt, da die neuere Kornshell und jetzt die aktuelle `bash` im wesentlichen Erweiterungen der Bourne-Shell sind.

Die *C-Shell* (`/bin/csh`) ist nicht kompatibel zur Bourne-Shell. Um effektiver zu

[2]Wer aus der DOS/Windowswelt kommt, kennt vielleicht `command.com`. Dieses Programm entspricht etwa einer Shell unter Unix, wenn auch mit deutlichen Abstrichen.

[3]Unix ist ein benutzerfreundliches System. Es sucht sich aber seine Freunde sehr gut aus…

arbeiten, sind früher viele Leute auf die `csh` umgestiegen. Da sie eben inkompatibel zur `sh` ist, wurden nicht allzuviele Skripte damit erstellt. Sie wurde weitgehend zur interaktiven Arbeit verwendet. Aus der `csh` ist die `tcsh` entstanden, die ebenfalls hauptsächlich interaktiv benutzt wird.

Eine direkte Erweiterung der `sh` ist die *Korn-Shell* (benannt nach ihrem Schöpfer *David Korn*). Sie bietet gute Erweiterungen, auch zum interaktiven Arbeiten, und ist weitgehend kompatibel zur `sh`. Leider unterliegt sie in ihrer ursprünglichen Version der generellen Unixlizenz, sie ist keine freie Software. Als Nachbau entstand die frei verfügbare `pdksh` (public domain ksh), die sich aber nicht allzu weit durchsetzen konnte. Sie wird inzwischen von der ebenfalls freien `bash` (Bourne again shell) verdrängt, die leistungsfähiger ist und ebenfalls weitgehend kompatibel zu `sh` und `ksh`.

Inzwischen schwören viele Leute auf einen neuen Trend: `perl` verdrängt in einigen Bereichen die `sh`-Skripte. `perl` ist erstaunlich schnell entstanden, und wird von vielen Programmierern als das einzig Wahre proklamiert. Von ideologischen Auswüchsen abgesehen, ist sie einerseits sehr leistungsfähig, da sie einen sehr mächtigen Sprachumfang hat, und ihre Syntax klar definiert ist (wenn auch nicht sehr verständlich). Andererseits muß man sich intensiv damit beschäftigen, um damit ein Bein auf den Boden zu bekommen. `perl` ist vollkommen inkompatibel zu allen anderen Skriptsprachen, und auch zu sich selbst, da inzwischen mehrere Versionen existieren, die sich gar nicht vertragen. Es ist halt eine sehr junge Sprache, die sich noch schnell weiterentwickelt.

Um die Sache jetzt doch wieder einfacher zu machen: In diesem Text wird es in erster Linie um die Sprache von `sh` und `bash` gehen.

3.1.4 Wozu ist Shellprogrammierung gut, und wozu nicht?

Mit Hilfe eines Shellprogramms kann man viele Dinge im Alltag schnell und einfach erledigen. Fast alles, was im interaktiven Betrieb aufwendig ist (Umbenennen von vielen Dateien nach einem bestimmten Strickmuster, Änderungen von Texten nach festen Regel, ...) kann man in einem Shellprogramm effektiver machen.

Vorteile eines Shellprogramms:

- Direktes Arbeiten, da die Programme nicht kompiliert werden, sondern nur interpretiert. Kleine Programme können sehr schnell erstellt und getestet werden.

- Wenn man sich auf den standardisierten Sprachumfang beschränkt, sind die erstellten Programme innerhalb der Unix-Welt sehr portabel.

  Shellprogramme funktionieren im günstigsten Fall unabhängig von Betriebssystem und darunter liegender Hardware.

  Stellenweise kann man sich mit einem Shellprogramm auch über Unix hinauswagen: Für Windows 95/98/NT existiert eine angepaßte Version der bash, die anstelle der schwachbrüstigen MSDOS-Eingabeaufforderung ziemlich alles beherrscht, was man von einer Unix-Shell erwartet. Unter Windows halt deutlich langsamer als unter Linux, aber es geht[4].

- Auch wenn man in einem Shellprogramm eigentlich nicht mehr machen kann, als beim interaktiven Arbeiten, hat die Sache gegenüber dem direkten Arbeiten einen wesentlichen Unterschied: Der Ersteller des Shellprogramms muß nicht identisch sein mit dem Anwender.

  Wenn beispielsweise ein reiner Anwender, der von den Details eines Unixsystems recht wenig weiß, irgendwelche kryptischen Kommandos eingeben müßte, dann kann ein Shellprogramm sehr nützlich sein: ein Unixkundiger legt die nötigen Befehle in einem Skript ab, und der Unixunkundige (oder gegebenfalls mehrere) können das Skript nutzen, ohne es zu verstehen zu müssen.

  Auch wenn Ersteller und Anwender die selbe Person sind, kann das von Vorteil sein: Wenn man für eine gelegentlich auftretende Arbeit eine ziemlich komplizierte Kommandos zusammenbauen muß, hat man die Details bis zum nächsten Mal vielleicht schon wieder vergessen. Gießt man seine Erkenntnisse aber gleich beim ersten Mal in ein Shellprogramm, dann kann man es beim nächsten Mal einfach aufrufen. Gerade für vergeßliche Menschen wie mich ist das Gold wert. Beispiel: Audio-CD unter Linux kopieren. Dazu muß man mit einer länglichen Kommandozeile vom Original die einzelnen Tracks auf die Festplatte kopieren, und wieder mit einer kryptischen Kommandozeile auf einen Rohling brennen. Anfangs mußte ich jedesmal wieder nachdenken, wie das denn nun genau ging, und hatte es bis zur nächsten CD zwei Wochen später wieder vergessen. Jetzt habe ich ein Shellprogramm, das die einzelnen Tracks auf die Platte kopiert, und dabei gleich ein weiteres Shellprogramm generiert, mit dem man den Rohling brennen kann. Jetzt muß ich nur noch jedesmal das Shellprogramm wiederfinden.

- Eine Erweiterung dieses Gedankens berührt die Systemsicherheit. Angenommen, ein normaler Benutzer muß für irgendeine Arbeit manchmal eine

[4] „If you are suffering Windows 95 (or 98), GNU-Win32 will install just fine for you too. There are a few things that either don't work or are a little flaky, but that's what you get for running 32 bit extensions and a graphical shell for a 16 bit patch to an 8 bit operating system originally coded for a 4 bit microprocessor, written by a 2 bit company, that cannot stand 1 bit of competition.“ Pete Jordan

(kostenpflichtige) Wählverbindung zum Onlineprovider aufbauen. Die Firma möchte aber nun erreichen, daß dieser Verbindungsaufbau ausschließlich für eine bestimmte Funktion im Zusammenhang mit der Arbeit möglich sein soll. Wenn man jetzt dem Benutzer generell den Aufbau der Verbindung ermöglicht, könnte er natürlich nach Belieben auch für andere Zwecke die Verbindung herstellen, sei es absichtlich oder unabsichtlich.

Ein ähnlicher Fall ist gegeben, wenn normale Anwender (ohne Administratorfunktion) für ganz bestimmte Zwecke auf ein Bandlaufwerk zugreifen müssen, um beispielsweise Daten an ein Band anzufügen, ohne die bereits auf dem Band befindlichen Daten löschen zu dürfen. Gibt man ihnen die Berechtigung dazu direkt (beispielsweise nach Setzen von `chmod 777 /dev/tape /dev/ntape`), können sie versehentlich oder absichtlich auch das Band an den Anfang spulen, und damit vorhandene Daten (vielleicht Daten anderer Benutzer?) beim nächsten Schreiben zerstören.

In solchen Fällen kann ein Administrator (`root`) ein Skript erstellen, mit dem nur die beabsichtigten und zulässigen Aktionen möglich sind (Daten auf ein Band nur schreiben, wenn es an das Ende gespult ist beispielsweise). Dieses Shellprogramm kann so installiert werden, daß es mit `root`-Rechten läuft, aber von normalen Benutzern gestartet werden darf. So braucht der Benutzer selbst keine Rechte haben, um das Bandlaufwerk anzusprechen, oder eine Onlineverbindung herzustellen, und kann es indirekt über das Shellprogramm doch - soweit das Shellprogramm es zuläßt.

Nachteile:

- Da die Programme im Quelltext vorliegen müssen, sind sie leicht kopier- und manipulierbar. Das kann in einem kommerziellen oder sicherheitsrelevanten Umfeld kritisch sein.
- Größere Projekte sind schwer zu debuggen.
- Durch das Interpretieren zur Laufzeit können Programme sehr langsam werden.

Entsprechend dieser Vor- und Nachteile kann man im Einzelfall abschätzen, ob sich ein Shellprogramm lohnt.

Trotz der vielen Programmiersprachen ist die eierlegende Wollmilchsau noch nicht erfunden. Shellprogramme sind oft nützlich, in vielen Fällen gibt es bessere Möglichkeiten. Vor allem, wenn ein Programm droht etwas aufwendiger zu werden, dann fährt man mit Perl oder mit kompilierten Programmen oft besser.

Typische Anwendungsfälle sind:

- kleine Hilfsprogramme, die recht eng mit den vielen Unixkommandos verzahnt sind (`sed`, `grep`, `find`, `awk` ...)
- Installationsprogramme, Setupprogramme, Konfigurationsprogramme
- einfache automatische Textverarbeitung (in Quelltexten Adressen anpassen, Suchen nach Schlüsselwörtern)
- komplexe Programmaufrufe je nach System für den einfachen Anwender erleichtern (X-Windows starten, Netscape aufrufen)

Sicher ungeeignet ist die Shellsprache um:

- ein Zeichenprogramm zu bauen
- ein Finite-Elemente-System zu bauen
- einen C++-Compiler zu implementieren

3.2 Aufruf eines Shellskripts

Es gibt mehrere Möglichkeiten ein Shellprogramm aufzurufen:

- die populärste Möglichkeit ist sicher, in einer bereits laufenden Shell den Namen des gewünschten Shellprogramms einzugeben, danach eventuelle Parameter, und dann *Return* zu drücken.

  Dadurch wird eine neue Shell gestartet (als Kind der aktuell laufenden Shell, in der man das Kommando eingegeben hat). Diese neue Shell erbt praktisch das gesamte Environment sowie die Standardeingabe, die Standardausgabe, und die Standardfehlerausgabe von ihrem Aufrufer, und führt dann die Kommandos im Shellprogramm aus. Durch das Starten einer neuen Shell werden Änderungen an Variablen oder ein Wechsel des Arbeitsverzeichnisses, die das Shellprogramm vielleicht ausführt, nicht beim Aufrufer wirksam.

  Voraussetzung für einen solchen Aufruf ist allerdings, daß die Rechte zum Lesen und Ausführen des Shellprogramms gesetzt (also zumindest `-r-x------` wenn der Aufrufer der Eigentümer der Programmdatei ist, oder `----r-x---` wenn er die Gruppenrechte der Gruppe hat, zu der die Programmdatei gehört, oder `-------r-x` sonst).

  Wenn der Aufrufer das Shellprogramm nur mit dem Namen (ohne Pfadangabe) aufruft, dann muß es in einem der Verzeichnisse enthalten sein, die beim Aufrufer in der `PATH`-Variable aufgelistet sind.

  Beispiel 1 *Erzeugen eines kleinen Shellprogramms* `t`*, und Ausprobieren desselben in einer eigenen Shell:*

  neues Skript erzeugen:

  ```
  klaus@aw35:~ > cat >t                                 Skript schreiben[5]
  echo das aktuelle Verzeichnis:
  ls
  echo Und Tschüß!                             diese Zeile mit Return
                                                 und Strg-D beenden!
  klaus@aw35:~ > t                                    Skript aufrufen
  bash: ./t: Keine Berechtigung                          Oh, schade!
  klaus@aw35:~ > ls                                Gibt es das Skript?
  t                                                              ja!
  klaus@aw35:~ > ls -l                              ist es ausführbar?
  -rw-r--r-- 1 klaus users 51 Apr 12 10:35 t                   nein!
  ```

[5] Anstelle von `cat` könnte man natürlich auch jeden beliebigen Editor nehmen!

```
klaus@aw35:~ > chmod 700 t                              ändern
klaus@aw35:~ > ls -l                     ist es jetzt ausführbar?
-rwx------ 1 klaus users 51 Apr 12 10:35 t                   ja!
klaus@aw35:~ > t                                   neuer Versuch
das aktuelle Verzeichnis:
t
Und Tschüß!                                                  Ah!
```

Wenn die erste Zeile des Skripts mit den beiden Zeichen `#!` beginnt, und danach der Name eines ausführbaren Programms kommt, dann wird dieses Programm als Interpreter des Skripts verwendet (Beispiel: `#!/bin/sh`). Ist diese erste Zeile so nicht vorhanden, dann wird stattdessen die Standardshell des Benutzers verwendet (`/bin/sh` oder `/bin/bash` meistens).

- Eine weitere Möglichkeit wäre, dem Programmaufruf einen Punkt und ein Leerzeichen voranzustellen. Im Vergleich zur ersten Möglichkeit bewirkt das im Wesentlichen zwei Dinge:

  1. Es wird keine neue Shell gestartet, um das Skript auszuführen, sondern die Kommandos im Shellprogramm werden von der aktuellen Shell interpretiert. Wird beispielsweise im Skript das Arbeitsverzeichnis geändert, oder eine Variable manipuliert, dann wirkt sich das auch auf den Aufrufer aus.
  2. es ist nur das Leserecht nötig (`r`), nicht das Recht zum Ausführen (`x`).

Beispiel 2 *Ein neues Skript in der aktuellen Shell ausführen:*
Erzeugen eines neuen Skripts `t2`*:*

```
klaus@aw35:~ > cat >t2
echo EGON war bisher $EGON
EGON=Krenz
echo EGON ist jetzt $EGON                 diese Zeile mit Return
                                             und Strg-D beenden!
```

Variable EGON definieren:

```
klaus@aw35:~ > export EGON=Waldschrat
```

Skript `t2` *in der aktuellen Shell ausprobieren:*

```
klaus@aw35:~ > . t2
EGON war bisher Waldschrat
EGON ist jetzt Krenz
```

Die Variable kontrollieren:

```
klaus@aw35:~ > echo $EGON
Krenz
```

EGON hat sich geändert! Nochmal auf alten Wert setzen:

```
klaus@aw35:~ > export EGON=Waldschrat
```
und das Skript `t2` *in einer eigenen Shell ausführen lassen:*
```
klaus@aw35:~ > chmod 700 t2
klaus@aw35:~ > t2
EGON war bisher Waldschrat
EGON ist jetzt Krenz
```
Wie lautet der Wert jetzt?
```
klaus@aw35:~ > echo $EGON
Waldschrat
```
Hat sich also diesmal durch `t2` *nicht geändert!*

- Man kann weiterhin natürlich auch eine Shell direkt mit ihrem Namen aufrufen, und ihr das auszuführende Skript als Parameter mitgeben:
```
klaus@aw35:~ > bash t2
```

- Letztlich kann man ein Skript (wie jede andere ausführbare Datei auch) von einem eigenen Programm aus mit `exec()` oder `system()` starten.

Beispiel 3 *Erzeugen eines kleinen C-Programms, mit dem das Skript* `t` *(siehe oben) aufgerufen wird.*

Erzeugen des C-Programms:
```
klaus@aw35:~ > cat >meinprg.c
#include <stdlib.h>
#include <stdio.h>

int main( int nargs, char **args )
{
  printf( "meinprg: gleich gehts los.\n" );
  system( "t" );
  printf( "meinprg: fertig!\n" );
  return 0;
}
```
diese Zeile mit Return und Strg-D beenden!
Programm übersetzen und ausführen:
```
klaus@aw35:~ > cc meinprg.c -o meinprg
klaus@aw35:~ > meinprg
meinprg: gleich gehts los.
das aktuelle Verzeichnis:
meinprg meinprg.c t
Und Tschüß!
```

```
meinprg: fertig!
```

Dabei wird aber letztlich natürlich nicht das Skript als Prozeß gestartet, sondern eine neue Shell (`sh`, `bash` oder was auch immer), die dann das Skript ausführt.

3.3 Die Shellsprache

3.3.1 Kommentare

Wie jede andere Programmiersprache kennt eine Shell auch Kommentare.

Nach einem einzeln stehenden `#` ist der Rest einer Zeile Kommentar, und damit für die Ausführung eines Skripts gleichgültig.

Vollständig leere Zeilen, oder Zeilen, in denen nur *white space* vorkommt (also Leerzeichen, Tabulatoren) werden ebenfalls ignoriert.

Jetzt kommt natürlich die übliche Empfehlung: reichlich, deutlich und für Doofe verständlich kommentieren!

Auch wenn es immer wieder die gleiche Leier ist: reichlich, deutlich und für Doofe verständlich kommentieren! Alles andere ist Murks.

3.3.2 Variablen, Parameter

Variablen heißen in der Shell nicht Variablen, sondern Shellparameter. Da aber in allen anderen Sprachen Variablen halt Variablen heißen, und Parameter die Dinger sind, die man an ein Programm oder an eine Funktion übergibt, werde ich Variablen auch weiterhin so nennen.

Variablen werden in der Shell als Zeichenketten (Strings) gespeichert. Es gibt einfach keine numerischen Variablen. Strukturen (`struct`, `class`), oder so etwas wie `union` oder `Record` sowieso nicht.

Um eine Variable zu verwenden, schreibt man ein Dollarzeichen `$`, gefolgt vom Variablennamen. Den Wert der Variablen `EGON` kann verwenden, indem `$EGON` schreibt.

Wenn direkt an den Variablennamen weiterer Text anschließt, kann die Shell natürlich nicht erkennen, wo der Name nun enden soll. Angenommen, man hat die Variable `BASIS` auf den Wert `10` gesetzt, und möchte jetzt mit dem Ausdruck `$BASISE3` den String `10E3` erzeugen, dann wird das nicht klappen, weil die Shell nicht nach dem Namen `BASIS` sucht, sondern nur nach `BASISE`, Als Lösung bietet es sich dann an, den gewünschten Variablennamen in geschweifte Klammern zu setzen: `${BASIS}E3` liefert tatsächlich `10E3`.

Wenn man eine Variable `NAME` verwendet, die nicht definiert ist, dann liefert `$NAME` einen leeren String. Insofern sieht man meistens keinen Unterschied zwischen einer Variablen, die gar nicht gesetzt ist, und einer Variablen, der ein leerer String zugewiesen ist. Manchmal ist der Unterschied aber von Belang, siehe **3.3.3 Vordefinierte und spezielle Variablen, shift, set** auf Seite 53. Eine gänzlich undefinierte Variable liefert nämlich bei ihrer Verwendung einen *Nullwert*, der sich in Ausdrücken manchmal anders verhält als ein leerer String. Will man eine bereits vorhandene Variable wirklich löschen, so daß sie den Nullwert liefert, dann kann man das mit `unset` machen; Beispiel:

```
klaus@aw35:~ > unset NAME
```

Wenn die Variable vorher gar nicht existiert hatte, schadet der Befehl auch nichts.

Die Parameter eines Skripts lassen sich mit dem (internen) Kommando `getopts` leichter auswerten als zu Fuß. Das setzt aber voraus, daß die Optionen Unix-üblich vor weiteren Argumenten kommen, und nur aus einem Bindestrich und einem weiteren Buchstaben bestehen (wie beispielsweise bei einem Druckaufruf `lpr -c -P lp2 -s datei1 datei2`).

Mit `getopts` lassen sich Parameter verarbeiten, die keine Argumente haben (`-c` und `-s` im `lpr`-Beispiel), oder genau ein Argument haben (`-P` im `lpr`-Beispiel). Die restlichen Argumente (hier `datei1` und `datei2`) müssen nach den Optionen kommen, und werden von `getopts` nicht verarbeitet.

`getopts` wird mit (mindestens) zwei Argumenten aufgerufen. Das erste Argument ist ein String, in dem die zu erkennenden Optionen angegeben sind. Eine Option, die ein zusätzliches Argument erhält, kennzeichnet man durch einen nachfolgenden Doppelpunkt (`:`). Um die `lpr`-Optionen für das obige Beispiel zu erkennen, müßte man also den String `"P:cs"` angeben.

Das zweite Argument ist der Name einer Variablen, in der die erkannte Option gespeichert wird (ohne Bindestrich).

`getopts` liefert mit jedem Aufruf eine weitere Option aus der Parameterliste, und gibt `true` (also 0) zurück, wenn eine Option gefunden wurde, oder `false`, wenn keine Option mehr gefunden werden kann.

Wenn eine der Optionen ein weiteres Argument hat (im String mit dem Doppelpunkt (`:`) markiert), dann wird dieses Argument in der Variablen `OPTARG` hinterlegt.

Die Parameterliste selbst wird nicht verändert.

Beispiel 4 *Mit dem Skript:*

```
#!/bin/sh

# Die Optionen -i, -j, -k und -l sollen erkannt werden,
# wobei für -k ein Argument angegeben werden muß:
while getopts "ijk:l" option
do
        case $option in
        i)   echo Die Option i
             ;;
        j)   echo die Option j
             ;;
        k)   echo die Option k mit Argument $OPTARG
             ;;
        l)   echo die Option l
             ;;
        \?)  echo falsche Argumente
             ;;
        *)   echo sollte nicht vorkommen
             ;;
        esac
done
```

erhält man beim Aufruf mit den Optionen `-i -k blabla -l` *folgende Ausgabe:*

```
Die Option i
die Option k mit Argument blabla
die Option l
```

`getopts` muß sich von einem Aufruf zum nächsten merken, wie weit es die Argumente schon ausgewertet hat. Dazu verwendet es die Variable `OPTIND`. Diese wird beim Starten eines Skripts automatisch auf `1` gesetzt, und von `getopts` bei jedem Aufruf weiter gezählt. Dies kann man nutzen:

1. um nach dem letzten `getopts`-Aufruf an weitere Argumente des Shellprogramms heran zu kommen (im `lpr`-Beispiel die Dateinamen `datei1` und `datei2`)

2. wenn man die Optionen ein zweites Mal ausgewertet haben will, setzt man einfach `OPTIND` selbst wieder auf `1`, und kann dann wieder in einer Schleife `getopts` aufrufen, bis es `false` liefert.

Beispiel 5 *Wenn man also an weitere Parameter (nach den Optionen) heran kommen möchte, kann man das Skript so erweitern:*

```
#!/bin/sh

# Die Optionen -i, -j, -k und -l sollen erkannt werden,
# wobei für -k ein Argument angegeben werden muß:
while getopts "ijk:l" option
do
        case $option in
        i)   echo Die Option i
             ;;
        j)   echo die Option j
             ;;
        k)   echo die Option k mit Argument $OPTARG
             ;;
        l)   echo die Option l
             ;;
        \?)  echo falsche Argumente
             ;;
        *)   echo sollte nicht vorkommen
             ;;
        esac
done

# OPTIND hat den Wert (Anzahl der ausgewerteten Parameter)+1.
# Diese Argumente brauchen wir schon nicht mehr, und verschieben
# deshalb die Argumente um $OPTIND-1 nach unten:
shift `expr $OPTIND - 1`

# Anzeige der restlichen Parameter:
echo $*
```

Bei einem Aufruf mit den Parametern `-k keinegnade -l blabla abc` *erhält man die Ausgabe:*

```
die Option k mit Argument keinegnade
die Option l
blabla abc
```

Siehe dazu auch **3.3.10 Numerik light mit expr** *auf Seite 71,* **3.3.12.2 while-Schleife** *auf Seite 77,* **3.3.3 Vordefinierte und spezielle Variablen, shift, set** *auf Seite 53, und* **3.3.13.2 case** *auf Seite 82.*

3.3.2.1 Namen von Variablen

Die Namen von Variablen dürfen aus den 26 kleinen (`a-z`) Buchstaben, den 26 großen (`A-Z`) Buchstaben, den 10 dezimalen Ziffern (`0-9`), und dem Unterstrich `_` (under score) gebildet werden.

Groß- und Kleinschreibung wird unterschieden.

Wie in vielen anderen Sprachen auch darf ein Name nicht mit einer Ziffer beginnen.

3.3.2.2 Setzen von Variablen, Exportieren

Bestehende Variablen kann man überschreiben, oder nicht bestehende neu anlegen, indem man ihnen einfach einen Wert zuweist:

```
klaus@aw35:~ > EGON=Waldschrat
klaus@aw35:~ > ZAHL=35
klaus@aw35:~ > PI=3.14159265
```

So definierte Variablen werden an Kind-Shells erstmal nicht weiter vererbt (im Gegensatz zu `PATH` beispielsweise).

Dazu muß man der Shell mit `export` sagen, daß man die Variablen vererbt haben möchte.

Beispiel 6 *Weitervererben einer Variable an Kind-Shells*
Setzen einer Variable und testweise Ausgabe:

```
klaus@aw35:~ > FUENF=5
klaus@aw35:~ > echo $FUENF
5
```

Aufruf einer Kind-Shell:

```
klaus@aw35:~ > bash
```

Kennt diese Shell die Variable?

```
klaus@aw35:~ > echo $FUENF

```

Offenbar nicht. Kind-Shell wieder beenden.

```
klaus@aw35:~ > exit
```

Jetzt sind wir wieder in der ursprünglichen Shell.
Variable exportieren:

```
klaus@aw35:~ > export FUENF
```

Nochmal eine Kind-Shell aufrufen:

```
klaus@aw35:~ > bash
```

Kennt diese Shell die Variable?

```
klaus@aw35:~ > echo $FUENF
5
```

Offenbar ja.

```
klaus@aw35:~ > exit
klaus@aw35:~ >
```

Das Setzen einer Variablen und das Exportieren kann man auch zusammenfassen:

```
klaus@aw35:~ > EGON=Waldschrat
klaus@aw35:~ > ZAHL=35
klaus@aw35:~ > PI=3.14159265
klaus@aw35:~ > export EGON ZAHL PI
```

ist gleichbedeutend mit:

```
klaus@aw35:~ > export EGON=Waldschrat
klaus@aw35:~ > export ZAHL=35
klaus@aw35:~ > export PI=3.14159265
```

3.3.3 Vordefinierte und spezielle Variablen, shift, set

Beim Starten einer Shell werden viele Variablen bereits vordefiniert, die großteils veränderbar sind (`HOME`, `PATH` usw.). Ein kleinerer Teil kann nicht geändert werden (`PPID`).

Weiterhin gibt es Variablen, die eigentlich gar keine sind, sondern bei ihrem Aufruf etwas erzeugen und zurückgeben.

Die wichtigsten im Überblick:

Ausdruck	liefert:
`$#`	Anzahl Argumente
`$0`	Name
`$1`...`$9`	Parameter, bash auch: `${1}`...`${9}` und weitere
`$*`	alle Parameter
`"$*"`	alle Parameter als ein String
`$@`	alle Parameter
`"$@"`	alle Parameter als je ein String
`$$`	process id
`$?`	Rückgabewert/exit-Status des letzten Kommandos
`${`*name*`-`*wert*`}`	Wenn *name* definiert ist, dann `$`*name* , sonst *wert*
`${`*name*`=`*wert*`}`	Wenn *name* definiert ist, dann `$`*name* , sonst (*name*=*wert*)
`${`*name*`+`*wert*`}`	Wenn *name* definiert ist, dann *wert* , sonst Nullwert
`${`*name*`?}`	Wenn *name* definiert ist, dann `$`*name* , sonst (Abbruch)
`${`*name*`?`*wert*`}`	Wenn *name* definiert ist, dann `$`*name* , sonst (Ausgabe von *wert*, Abbruch)
`${`*name*`:-`*wert*`}`	Wenn *name* nicht Nullwert ist, d. `$`*name* , sonst *wert*
`${`*name*`:=`*wert*`}`	Wenn *name* nicht Nullwert ist, d. `$`*name* , sonst (*name*=*wert*)
`${`*name*`:+`*wert*`}`	Wenn *name* nicht Nullwert ist, d. *wert* , sonst Nullwert
`${`*name*`:?}`	Wenn *name* nicht Nullwert ist, d. `$`*name* , sonst (Abbruch)
`${`*name*`:?`*wert*`}`	Wenn *name* nicht Nullwert ist, d. `$`*name* , sonst (Ausgabe von *wert*, Abbruch)

Die Positionsparameter (`1` etc., `*`, `@`) und die Anzahl (`#`) beziehen sich innerhalb einer Funktion auf die Parameter, mit denen die Funktion aufgerufen wurde (siehe **3.3.8 Funktionen** auf Seite 67). Außerhalb von Funktionen beziehen sie sich auf die Parameter des Shellprogramms.

Die Variablen `1`..., `*`, `@`, und `#` kann man mit dem Befehl `shift` beeinflussen.

`shift` (ohne Argumente) verwirft den ersten Parameter `$1` unwiderruflich, und rückt die anderen um eine Position auf. `*`, `@`, und `#` werden entsprechend angepaßt. `shift` mit einer nachfolgenden Zahl verschiebt die Parameter analog dazu um mehrere Plätze.

Mit dem (internen) Kommando `set` kann man einerseits Optionen setzen, mit denen das Verhalten der Shell selbst beeinflußt wird. Beispielsweise kann man mit `set -o vi` bewirken, daß das Aufrufen und Ändern von alten Kommandos mit `vi`-Befehlen erfolgt (`k` für: *eine Zeile aufwärts*, `j` für: *eine Zeile abwärts*, `h` bzw. `l` für: *Cursor nach links/rechts*, `i` um etwas einzufügen, ...). Mit `set -o emacs` kann man wieder vernünftigerweise auf die Funktionen zurückschalten, wie man sie vom EMACS kennt.

Andererseits kann man mit `set` die Positionsparameter `$1` etc. auf beliebige Werte setzen. Die davon abhängigen Variablen `#`, `*` und `@` ändern sich dabei automatisch mit.

Beispiel 7 *Setzen der Variablen* `1`*,* `2`*, und* `3`*:*

```
klaus@aw35:~ > set abc def ghi
klaus@aw35:~ > echo $2
def
klaus@aw35:~ > echo $#
3
klaus@aw35:~ > echo $*
abc def ghi
```

3.3.4 Verarbeitung eines Kommandos, eval

Jedes Kommando wird von der Shell einer *command expansion* unterzogen, bevor es gestartet wird. Dazu wird nacheinander ausgeführt:

- *brace expansion*: aufeinanderfolgende geschweifte Klammerpaare werden zu einer Liste von Wörtern expandiert.

  Beispiel: Die Kommandozeile
  `echo {kohl,schäuble}.spende`
  wird expandiert zu
  `echo kohl.spende schäuble.spende`

  Mehrere solcher Ersetzungen können kombiniert werden.
  So wird das Kommando `touch {lesen,schreiben}.{cpp,h}` expandiert zu `touch lesen.cpp lesen.h schreiben.cpp schreiben.h`.

- *tilde expansion*: Eine Tilde (`~`) am Anfang eines Wortes wird durch das *home directory* des aktuellen Benutzers ersetzt, wenn kein Name folgt. Ist nach dem `~` dagegen ein Name angegeben, dann werden die Tilde und der Name durch das Homeverzeichnis des angegebenen Benutzers ersetzt.

- *parameter and variable expansion*: Das Dollarzeichen (`$`) und der nachfolgende Name werden entsprechend der definierten Variablen ersetzt (siehe dazu **3.3.2.2 Setzen von Variablen, Exportieren** auf Seite 51).

- *command substitution*: Ein Ausdruck der Form `` ` ``*Kommando*`` ` `` oder `$(`*Kommando*`)` wird ersetzt durch die Standardausgabe von *Kommando*.

  In der ersten Form (mit den Gegenapostrophen `` ` ``, auch *back quotes*) werden `\$`, ``\` `` und `\\` ersetzt durch `$`, `` ` `` und `\`.

  In der Version der Form `$(`*Kommando*`)` erfolgt diese Ersetzung nicht.

  In beiden Fällen werden mehrere Zeilen der Standardausgabe von *Kommando* zu einer Zeile zusammengefaßt, indem der Zeilentrenner (*line feed*, `\n`) durch einen Worttrenner ersetzt wird (das Leerzeichen).

- *arithmetic expansion*: arithmetische Ersetzung eines Ausdrucks der Form `$((` *Ausdruck* `))` durch seinen Wert (siehe **3.3.9 Numerische Ausdrücke $((...))** auf Seite 68).

- *word splitting*: Anhand der Worttrennzeichen in der Variablen `IFS` werden alle Wörter der Eingabezeile aufgetrennt (diese Variable besteht normalerweise aus einem Leerzeichen, einem Tabulator und dem Zeilenvorschubzeichen `\n`).

- *filename expansion*: Wörter, die `*`, `?`, oder `[` enthalten, werden entsprechend den vorhandenen Dateinamen ersetzt (mit `set -f` kann dieses Verhalten abgeschaltet werden, mit `set +f` wieder an).

  Wenn eines dieser *wild card*-Zeichen in einem Wort enthalten ist, dann wird das Wort durch eine alphabetisch sortierte Liste von Dateinamen ersetzt, auf die das angegebene Muster zutrifft (*globbering*).

  Dabei steht `?` für genau ein Zeichen, `*` repräsentiert keines, eines oder beliebig viele Zeichen, und die Folge der Zeichen von einem `[` bis zum nächsten `]` steht für genau ein Zeichen, das einem der Zeichen zwischen den eckigen Klammern entsprechen muß. In diesen Klammern dürfen auch Bereiche angegeben werden (*erstes Zeichen*-*letztes Zeichen*).

  Ein Punkt am Anfang eines Namens muß normalerweise explizit angegeben werden (`*` und `?` können zwar generell auch für Punkte stehen, aber eben nicht am Anfang eines Wortes). Dieses Verhalten kann aber in der Shell mit dem Befehl `shopt -s dotglob` dahingehend geändert werden, daß alle Datei- und Verzeichnisnamen mit Punkten (außer `.` und `..`) wie andere Namen auch behandelt werden. Mit `shopt -u dotglob` kann man das ursprüngliche Verhalten wieder herstellen.

  Schrägstriche sind von der Ersetzung immer ausgenommen, weil sie nicht zu den Namen gezählt werden.

  Beispiele:

  – `*` trifft auf alle Datei- und Verzeichnisnamen (die nicht mit einem Punkt beginnen üblicherweise) des aktuellen Verzeichnisses zu

  – `/tmp/*` trifft auf alle Dateien und Verzeichnisse in `/tmp` zu

  – `*.*` trifft auf alle Namen zu, die mindestens einen Punkt enthalten (wichtiger Unterschied zu DOS/Windows!)

  – `urlaub??.txt` trifft auf alle Namen zu, die mit `urlaub` beginnen, dann genau zwei weitere beliebige Zeichen haben, und auf `.txt` enden.

  – `[0-9a-fA-F]*.xy` paßt auf alle Namen, die mit einer hexadezimalen Ziffer beginnen (also einer Ziffer von `0` bis `9` oder einem Kleinbuchstaben `a` bis `f` oder einem Großbuchstaben `A` bis `F`), dann irgendwie weitergehen und auf `.xy` enden.

Für Details (insbesondere, wann und wie Sonderzeichen berücksichtigt oder ignoriert werden): siehe `info bash`.

Manchmal ist es nötig, die Auswertung eines Kommandos mehrfach durchführen zu lassen. Hat man beispielsweise in einer Variablen `i` eine Zahl, und will anhand dieser Variablen auf den soundsovielten Positionsparameter zugreifen, hat man ein Problem. Angenommen, `i` hat den Wert zwei, und man will dann den Wert von `$2` verwenden. Dann liegt es nahe, einen Ausdruck der Art `${$i}` zu verwenden. Das klappt nur leider nicht, weil die Variablenersetzung nur „einstufig“ durchgeführt wird: `$i` wird zwar durch eine 2 ersetzt, aber der daraus resultierende Teilausdruck `${2}` erzeugt eine Fehlermeldung, weil zu diesem Zeitpunkt keine weitere Parameterersetzung mehr durchgeführt wird (die Variablenersetzung fand ja bereits statt).

Eine Lösung dafür hat man mit dem internen Kommando `eval`.

`eval` wird mit einem nachfolgenden Kommando aufgerufen, und läßt das ihm übergebene Kommando ein zweites Mal durch die obige *command expansion* laufen, führt dann das Kommando aus, und liefert selbst den Rückgabewert des Kommandos zurück.

Diesen Mechanismus kann man auch mehrfach anwenden: Wenn ein *Kommando* einmal die Ersetzung durchläuft, dann führt `eval` *Kommando* zu einer doppelten Ersetzung, und `eval eval` *Kommando* zu einer dreifachen.

Beispiel 8 *Ausgeben des zweiten Positionsparameters anhand des Werts* `2` *der Variablen* `i`*:*

```
klaus@aw35:~ > set wer würmer hat ist nie allein
klaus@aw35:~ > i=2
klaus@aw35:~ > echo \$$i
$2
klaus@aw35:~ > echo $$i
783i
klaus@aw35:~ > echo ${$i}
bash: ${$i}: bad substitution
klaus@aw35:~ > eval echo \${$i}
würmer
```

Der erste Versuch (`\$$i`*) bringt nicht das erhoffte Ergebnis:* `\$` *wird zwar zu einem einzelnen Dollarzeichen, und das folgende* `$i` *zu einer* `2` *expandiert. Es fehlt aber daran, das resultierende* `$2` *nochmals auszuwerten. Stattdessen wird* `$2` *ausgegeben.*

Im zweiten Versuch hat man mit `$$i` *auch nicht mehr Glück:* `$$` *wird durch die aktuelle* process id *der Shell ersetzt (hier:* `783`*), und mit einem nachfolgenden* `i` *ausgegeben.*

Der dritte Versuch (`${$i}`*) bringt einen auch nicht weiter: das resultierende* `${2}` *ist nach der Ersetzung von Parametern ein ungültiger Ausdruck.*

Freude kommt erst beim letzten Versuch auf: durch das `eval` *wird das ganze Kommando zweimal durch die* command expansion *geschickt, bevor es ausgeführt wird. Beim ersten Durchlauf läßt man* `$i` *zu einer* `2` *expandieren, und beim zweiten Durchlauf mit einem vorangestellten* `$` *zum Wert des zweiten Positionsparameters (*`würmer`*). Damit dieses vorangestellte Dollarzeichen aber überhaupt unbeschadet die erste Ersetzung übersteht, muß man einen* back slash *(\) voranstellen. Beim ersten Ersetzen wird also* `\$` *zu einem Dollarzeichen, und* `$i` *zu einer* `2`*. Beim zweiten Ersetzen entsteht aus dem dann vorliegenden* `${2}` *tatsächlich der Wert der Variablen, nämlich* `würmer`*.*

Die Angabe der geschweiften Klammern ist hier übrigens nicht nötig; das Kommando `eval echo \$$i` *ist gleichwertig, und funktioniert im Gegensatz zu der Lösung mit Klammern nicht nur in der* `bash`*, sondern auch mit der Bourne-Shell (*`/bin/sh`*). Ich finde aber die Version mit Klammern verständlicher*[6]*.*

3.3.5 Standarddateien

Unter Unix (und den meisten gängigen anderen Betriebssystemen ebenso) haben neu gestartete Programme drei Dateien zur Verfügung, die nicht explizit geöffnet werden müssen:

- die *Standardeingabe*
- die *Standardausgabe*
- die *Standardfehlerausgabe*

Bei interaktiven Systemen sind die Standarddateien meistens mit den Ein- und Ausgabegeräten verbunden, mit denen der Benutzer arbeitet. Also wird Tastatureingabe in der Standardeingabe landen, und alles zur Standardausgabe und zur Standardfehlerausgabe geschriebene wird auf den Bildschirm oder in ein Fenster dort ausgegeben werden.

Bei einem Programmlauf im Stapelbetrieb wird entsprechender Ersatz geschaffen werden, beispielsweise Standardeingabe aus einem Lochkartenstapel oder einem Modem, und die Ausgabekanäle möglicherweise zu einem Drucker.

[6]In Franken gibt es für Leute, die einen recht schrägen Geschmack haben, den Kommentar: „Mei Katz moch die Meis roh, iich mogs nedd amol g'kocht“

Wie alle anderen Dateien auch, werden (innerhalb von C-Programmen) diese mit den üblichen Funktionen (`read()`, `write()`, `fcntl()`, `dup()`, ...) verwendet.

Da die Dateien aber schon bei Programmstart geöffnet sind, ist ein `open()` beziehungsweise `creat()` sinnlos. Bei explizit geöffneten Dateien liefern diese beiden Funktionen aber den *Dateideskriptor* (eine ganze Zahl), der für alle Dateioperationen nötig ist.

Stattdessen werden für die vorgeöffneten Dateien feste Zahlen verwendet, und zwar für die Standardeingabe der Wert `0`, für die Standardausgabe `1` und für die Standardfehlerausgabe `2`; mit diesen Konstanten als Dateideskriptoren kann man also die Dateifunktionen ansprechen[7].

Viele Programme verwenden nun für die Ein- und Ausgabe gar keine konkreten Dateien, sondern lesen einfach nur von der Standardeingabe und schreiben das gewünschte Ergebnis zur Standardausgabe. Treten Fehler auf (oder sollen andere Informationen ausgegeben werden, die nicht zum normalen Ergebnis gehören; beispielsweise Warnungen oder statistische Angaben wie der verbrauchten CPU-Zeit), dann wird dafür die Standardfehlerausgabe verwendet.
Anmerkung: Solche Programme, die diesem einfachen Schema *Lesen einer Zeichenfolge von der Standardeingabe, irgendwie verarbeiten, Schreiben zur Standardausgabe* entsprechen, heißen *Filterprogramme*.

Dies vereinfacht die Programme deutlich, weil keine Dateinamen erfragt und benutzt werden müssen, und macht sie zumindest unter unixähnlichen Systemen gleichzeitig viel flexibler. Die Zuordnung der drei vordefinierten Dateideskriptoren zu konkreten Dateien oder zur Verknüpfung mit anderen Programmen kann nämlich vom Aufrufer in der Shell vorgenommen werden, ohne daß sich die aufgerufenen Programme darum kümmern müssen.

Auf die Voreinstellung der Standarddateien kann mittels *Pipelines* (**3.3.5.1 Pipeline: Datenfluß durch mehrere Kommandos** auf Seite 60) und *Ein-/Ausgabeumlenkung* (**3.3.5.2 Ein-/Ausgabeumlenkung** auf Seite 62) Einfluß genommen werden.

Für die folgenden Beispiele wird das Unixkommando `sed` verwendet, ein ganz typisches Filterprogramm. `sed` steht für *stream editor*: das Programm liest einen Zeichenstrom von der Standardeingabe, bearbeitet ihn wie ein Editor anhand von Kommandos, die als Parameter übergeben werden, und schreibt den möglicherweise geänderten Text wieder zur Standardausgabe. Ein Kommando kann mit

[7] Die vorgeöffneten Dateien `stdin`, `stdout` und `stderr`, die in Programmen ANSI-C verwendet werden können, sind intern wieder mit `0`, `1` und `2` verknüpft.

der Option `-e` angegeben werden, dem der auszuführende Befehl folgt (siehe `man 1 sed`). Hier wird nur ein Kommando verwendet, nämlich `s` (für *substitute*) zur Ersetzung von Text. Hinter `s` folgen drei Schrägstriche; zwischen dem ersten und dem zweiten steht der zu ersetzende Text, zwischen dem zweiten und dritten der Ersatztext.

Damit könnte ein Aufruf von `sed` so aussehen:

```
klaus@aw35:   > sed -e s/gewonnen/verloren/
wir haben die Wahl gewonnen!
wir haben die Wahl verloren!
die anderen sind die schlechteren.
die anderen sind die schlechteren.
Wie gewonnen, so zerronnen!
Wie verloren, so zerronnen!
```

Ein weiteres in den folgenden Beispielen verwendete Kommando ist `fgrep`; dieses Kommando bekommt in seiner einfachsten Form ein Argument übergeben, liest seine gesamte Eingabe und leitet zur Ausgabe alle Zeilen weiter, in denen das übergebene Argument enthalten ist:

```
klaus@athlon1:/lap2/klaus/skript+shellprogrammierung > fgrep onnen
wir haben die Wahl gewonnen!
wir haben die Wahl gewonnen!!
die anderen sind die schlechteren.
Wie gewonnen, so zerronnen!
Wie verloren, so zerronnen!
```

Hier wird die zweite Eingabezeile unterdrückt, weil sie nicht den Text `onnen` enthält.

3.3.5.1 Pipeline: Datenfluß durch mehrere Kommandos

Eine *Pipeline* besteht aus einem oder mehreren einfachen Kommandos, die (falls es mehrere sind) durch das *pipeline symbol* (das ist das Zeichen `|`) getrennt sind.

Alle Kommandos einer Pipeline laufen gleichzeitig, wobei das erste Kommando die normale Standardeingabe zu sehen bekommt, während alle anderen Kommandos als Eingabe jeweils die Standardausgabe ihres vorhergehenden Kommandos erhalten. Die Standardausgabe des letzten Kommandos der Pipeline wird dann zur Ausgabe der gesamten Pipeline.

Die Standardfehlerausgabe aller Kommandos wird durch diesen Mechanismus nicht beeinflußt.

Beispiel 9 *Die Eingabe wird nacheinander durch drei Kommandos gefiltert: ein* sed*, welches alle* `zerronnen` *durch* `vergoren` *ersetzt, dann ein* fgrep*, welches nur Zeilen passieren läßt, die den Text* `gewonnen` *enthalten, und dann ein weiteres* sed*, welches* `gewonnen` *durch* `verloren` *ersetzt (die lange Eingabezeile ist mit einem \ umgebrochen, damit sie auf die Seite paßt):*

```
klaus@aw35: ~ > sed -e s/zerronnen/vergoren/ | \
                fgrep gewonnen | sed -e s/gewonnen/verloren/
wir haben die Wahl gewonnen!
die anderen sind die schlechteren
Wie gewonnen, so zerronnen.
wir haben die Wahl verloren!
Wie verloren, so vergoren.
```

Anmerkung: Durch die interne Pufferung wird jetzt nicht mehr nach jeder Eingabezeile sofort die Ausgabe sichtbar, sondern erst nach dem Ende der Eingabe mit *Strg-D*. Dies hat aber auf die beschriebenen Mechanismen keinen weiteren Einfluß. Aber in jedem Fall kann man sich nicht darauf verlassen, wann die Ausgabe (bezogen auf die Eingabe) erfolgt. Allerdings kann man sich darauf verlassen, daß die Reihenfolge innerhalb des Eingabestroms und innerhalb der Ausgabeströme von der Shell nicht manipuliert wird.

(Man kann sich zum Verständnis eine Pipeline auch so vorstellen, daß das erste Kommando wie üblich aufgerufen wird, aber seine gesamte Ausgabe irgendwo heimlich gesammelt wird. Anschließend wird das zweite Kommando aufgerufen, und mit der gesammelten Ausgabe des Vorgängers gefüttert, wobei dessen Ausgabe wiederum gesammelt wird, und so weiter.)

Damit können mehrere Kommandos zu einem komplexeren zusammengefaßt werden, und erscheinen von außen betrachtet wieder wie ein Kommando.

Optional darf vor der gesamten Pipeline noch

```
  time
```

oder

```
  time -p
```

geschrieben werden.

Dadurch wird die anschließend angegebene Pipeline wie üblich ausgeführt, und zuletzt die verbrauchte Rechen- und Gesamtzeit ausgegeben.

In der `-p`-Variante erfolgt diese Ausgabe nach POSIX-Standard (in der `bash`).

Tip: Wenn man in einer Pipeline wissen möchte, was zwischen zwei Kommandos übertragen wird, kann man das leicht protokollieren. Dazu gibt es das Kommando `tee` (gesprochen wie der englische Buchstabe T). Es wird mit einem Dateinamen aufgerufen, und funktioniert wie ein T-Stück in der Pipeline: alles, was zu seiner Eingabe kommt, wird sowohl unverändert zur Ausgabe kopiert, als auch in die angegebene Datei geschrieben:

```
...kommando_n-1 | tee a.log | kommando_n...
```

erzeugt in der Datei `a.log` eine Kopie der Daten, die von `kommando_n-1` nach `kommando_n` fließen.

3.3.5.2 Ein-/Ausgabeumlenkung

Für eine gesamte Pipeline (**3.3.5.1 Pipeline: Datenfluß durch mehrere Kommandos** auf Seite 60), also im einfachsten Fall auch für ein einfaches Kommando, können die drei Standarddateien auf andere Dateien umgeleitet werden.

Diese drei Umlenkungen können einzeln verwendet werden, oder kombiniert. Die Reihenfolge untereinander ist insoweit egal.

In jedem Fall gelten die Umleitungen für die gesamte Pipeline; die Eingabe wird also bei mehreren Programmen in der Pipeline für das erste Kommando umgeleitet, und die beiden Ausgaben für das letzte Kommando der Pipeline.

Für die Eingabe geschieht dies, indem hinter[8] der Pipeline nach einem Kleinerzeichen (`<`) ein Dateiname angegeben wird:

```
fgrep gewonnen <a.txt
```

führt wie sonst auch das Kommando `fgrep gewonnen` aus, allerdings wird die Eingabe nicht von der Tastatur gelesen, sondern aus der Datei `a.txt`.

Analog kann man die Standardausgabe mit `>`*Dateiname* und die Standardfehlerausgabe mit `2>`*Dateiname* in eine Datei umleiten (die Ziffer `2` in `2>` kommt davon, daß die Standardfehlerausgabe intern den Dateideskriptor 2 hat). Generell kann man vor das Umleitungszeichen den zugehörigen Dateideskriptor schreiben, sodaß `<` nur ein Synonym für `0<` ist, und analog `>` und `1>` identisch sind.

[8] Eigentlich dürfen die Umleitungen auch vor dem Kommando oder zwischen den Argumenten stehen; aber weil es sich eingebürgert hat, alle Umlenkungen erst nach den Argumenten zu schreiben, sind die Kommandos so leichter lesbar.

Entsorgen unerwünschter Ausgaben: Will man eine der Ausgaben überhaupt nicht haben, kann sie unter Unix nach `/dev/null` umgeleitet werden (oder beide zusammengelegt, wie oben beschrieben).

Umleitung von/zu geöffneten Dateien: Weiterhin kann man anstatt eines Dateinamens auch (nach einem `&`-Zeichen) den Deskriptor einer geöffneten Datei angeben (falls man ihn kennt natürlich).

Dies wird meistens dazu genutzt, die Standardfehlerausgabe mit der Standardausgabe zusammenzulegen:
  Kommando `2>&1`
leitet die Fehlerausgabe (2) auf die Standardausgabe (1) um.Dabei muß man allerdings beachten, daß eine weitere Umleitung der Standardausgabe die Fehlerausgabe nicht „mitnimmt“. Will man also beide Ausgaben in einer einzigen Datei wiederfinden, muß man das so schreiben (die Umleitungen werden immer von links her ausgeführt!):
  Kommando `>`*Dateiname* `2>&1`
Im umgekehrten Fall (`2>&1 >`*Dateiname*) würde erst die Fehlerausgabe zur Standardausgabe umgelenkt; letztere dann in die Datei, aber ohne die Fehlerausgabe mitzunehmen. *Bei der* `bash` *gibt es zum Zusammenlegen der beiden Ausgaben in eine Datei auch die Kurzform* `&>`Dateiname.

Analog kann man auch für ein Kommando in einer Pipeline erreichen, daß seine Fehlerausgabe in die Pipeline einfließt:
  Kommando1 `2>&1 |` *Kommando2*
Dadurch werden sowohl Standardausgabe als auch die Standardfehlerausgabe von *Kommando1* an *Kommando2* weitergereicht. Wenn es nur die Fehlerausgabe sein soll, geht das natürlich auch:
  Kommando1 `2>&1 >/dev/null |` *Kommando2*
(die Reihenfolge ist wichtig!).

Die direkte Angabe der Dateideskriptoren kann man weiterhin nutzen, um in einem eigenen Programm Dateien zu öffnen, und anschließend ein weiteres Programm aufzurufen, wobei dessen Standarddateien auf die eigenen geöffneten umgebogen werden können:

```
int eingabe = open( "eingabe.txt", O_RDONLY );
int ausgabe = creat "ausgabe.txt", 0600 );
// ... eingabe, ausgabe prüfen ...
char kommando[1024];
sprintf( kommando, "prg arg <&%d 2>%d", eingabe, ausgabe );
system( kommando );
```

Dabei wird in `kommando` eine Kommandozeile zusammengebaut; wenn die Deskriptoren beispielsweise die Werte 12 und 14 haben, dann würde die Kommandozeile den Inhalt `prg arg <&12 2>%14` haben; also liest das mit `system()` gestartete Programm aus der Datei `eingabe.txt` und schreibt nach `ausgabe.txt`.

Anhängen statt Überschreiben: Eine als Ausgabe angegebene Datei wird durch die Umleitung gelöscht, falls sie bereits existiert (das kann man mit dem Kommando `set -C` verhindern). Will man an eine Datei etwas anhängen, verwendet man `>>` statt `>`.

Öffnen von Dateien Falls man mehr Dateien als die drei vordefinierten benötigt, kann man am Ende einer Pipeline weitere öffnen mit einer Konstruktion der Art *Deskriptor*<*Dateiname* (zum Lesen) beziehungsweise *Deskriptor*>*Dateiname* zum Schreiben. Dabei ist *Deskriptor* eine ganze Zahl (größer als 2 natürlich, weil 0, 1, und 2 ja schon vergeben sind). Von allen so geöffneten Dateien kann dann in der Pipeline lesen oder oder darauf schreiben durch Angabe des Deskriptors.

Es hängt von der Shell ab, wieviel Dateien geöffnet werden können. Portabel kann man bis zur Nummer 9 gehen; die `bash` limitiert nur durch die maximal mögliche Anzahl offener Dateien des Betriebssystems.

here-Dokumente: Eng mit der Eingabeumlenkung verwandt sind *here-Dokumente*. Dabei wird die Eingabe eines Kommandos direkt im Kommando erzeugt, indem als Operator `<<` angegeben wird, gefolgt von einer frei wählbaren Endemarkierung. Aus dem Skript (oder von der Tastatur bei einem interaktiven Kommando, da macht es nur keinen Sinn) wird dann Text gelesen und an das Kommando weitergereicht, bis die Endemarkierung am Anfang einer Zeile auftritt:

```
fgrep Hallo << ichhabefertig
Hallo, Max
Holdrio
Hallo, mein Name ist $USER
ichhabefertig
```

Hier wird alles nach dem `fgrep`-Kommando bis (ausschließlich) der `ichhabefertig`-Zeile als Standardeingabe an das `fgrep` gereicht.

In dieser Form wird innerhalb des here-Dokuments wie üblich eine Expansion von Variablen, Kommandos und arithmetischen Ausdrücken durchgeführt. Der

Endetext hinter Operator `<<` wird dagegen generell nicht ausgewertet.

Allerdings kann er ganz oder teilweise in Gänsefüßchen oder Apostrophe eingeschlossen werden (*quoted*); dann findet innerhalb des here-Dokuments auch keine Ersetzung mehr statt (beispielsweise für `$USER`).

Eine weitere Variante ist die Verwendung des Operators `<<-` anstatt `<<`. Dadurch werden im gesamten here-Dokument in jeder Zeile führende Leerzeichen und Tabulatoren entfernt. Dadurch lassen sich Skripte schöner schreiben, ohne die Einrückungen zu unterbrechen.

3.3.6 Listen

Eine oder mehrere Pipelines (und damit auch einzelne Kommandos) können zu einer Liste zusammengefaßt werden, indem hinter jede Pipeline einer der vier Operatoren `&`, `;`, `&&`, oder `||` geschrieben wird (oder zumindest zwischen aufeinanderfolgende Pipelines). Hinter der letzten Pipeline ist allerdings nur `&`, `;` oder ein Zeilenvorschub zulässig.

Dabei haben `&` und `;` gleiche Priorität beim Zusammenfassen, und zwar eine niedrigere als `&&` und `||` (die auch wieder gleichen Vorrang haben).

Zwei mit `;` getrennte Pipelines werden in der angegebenen Reihenfolge abgearbeitet (von links nach rechts). Die rechts stehende wird erst gestartet, wenn die links stehende beendet ist (unabhängig vom Rückgabewert der linken Pipeline). Das Ergebnis der linken Pipeline wird verworfen, nur die ganz rechts in einer solchen Liste stehende Pipeline liefert den Rückgabewert des Gesamtausdrucks.

Eine mit `&` beendete Pipeline wird als Hintergrundprozeß gestartet; ihr Rückgabewert kann nicht verwertet werden. Stattdessen wird der Wert `0` angenommen.

Sind zwei Pipelines mit dem Operator `&&` getrennt, dann wird erst die linke Seite ausgeführt und es wird auf ihr Ende gewartet. Falls sie dann den Wert `0` (also *erfolgreich*) liefert, wird die rechte Seite gestartet. Rückgabewert des Gesamtausdrucks ist `0`, wenn beide Teilausdrücke `0` lieferten (logisches UND, *short circuit evaluation*; siehe dazu auch **3.3.11 test-Ausdrücke, true, false** auf Seite 72).

Analog wird bei `||` der linke Ausdruck ausgewertet; nur wenn er einen Wert ungleich `0` liefert, wird der rechte ausgeführt. Das Ergebnis ist `0`, wenn mindestens einer der beiden `0` ergab (logisches ODER).

3.3.7 Gruppen

Eine Liste (mit `;`, `&`, `&&` oder `||` zusammengefaßte Pipelines oder Kommandos, siehe **3.3.6 Listen** auf Seite 65) kann mit runden oder geschweiften Klammern zu einer Kommandogruppe erklärt werden, die von außen wie ein einzelnes Kommando betrachtet wird. Das ermöglicht, beispielsweise die Standardausgabe mehrerer aufeinanderfolgender Kommandos gemeinsam in eine Datei umzuleiten.

Bei der Verwendung von runden Klammern
`(` *Liste* `)`
wird die eingeschlossene Liste in einer eigenen Shell ausgeführt; bei den geschweiften Klammern:
`{` *Liste;* `}`
dagegen innerhalb der aktuellen Shell. Im letzteren Fall muß innerhalb der Klammern mindestens ein Leerzeichen stehen; weiterhin muß die Liste mit einem Semikolon angeschlossen sein.

Das Ergebnis der Liste wird als Ergebnis der Gruppe verwendet.

3.3.8 Funktionen

In einer Shell kann man Funktionen (Unterprogramme) definieren.

Funktionsnamen müssen nach den selben Regeln wie Variablennamen gebildet werden (siehe **3.3.2.1 Namen von Variablen** auf Seite 51). Dabei dürfen in der Original-`sh` eine Funktion und eine Variable nicht denselben Namen haben; in der `bash` dagegen doch.

Eine Funktion wird einfach mit ihrem Namen aufgerufen, und gegebenenfalls (wie bei Programmaufrufen) mit Parametern.

Die Parameter werden nicht bei der Definition einer Funktion mit deklariert. Sie sind innerhalb der Funktion mit `$*` etc. zu erreichen, wie die Parameter des Skripts außerhalb von Funktionen (siehe **3.3.2 Variablen, Parameter** auf Seite 47).

Eine Funktion liefert bei ihrem Aufruf (wie jeder Ausdruck) einen Rückgabewert, den man verwenden kann, aber nicht muß. Den Rückgabewert kann man setzen, indem man die Funktion mit `return` *wert* verläßt. Ansonsten wird das Ergebnis (der exit-Status) des letzten ausgeführten Kommandos in der Funktion als Rückgabewert verwendet.

Beispiel 10 *Im folgenden Beispiel wird eine Funktion definiert, und mit zwei Parametern aufgerufen. Die Funktion gibt ihre Parameter aus:*

```
#!/bin/bash
funny()
{
        echo "ich bin funny!"
        for f
        do
                echo Argument ist $f
        done
}
echo "es geht los"
funny 123 456
echo fertig
```

3.3.9 Numerische Ausdrücke $((...))

Innerhalb des Operators `$(( ... ))` wird ein arithmetischer Ausdruck ausgewertet. Dieser wird mit langen Ganzzahlen (vier Byte) vorzeichenbehaftet berechnet; Ergebnis ist ein String, der die berechnete Zahl repräsentiert. Die Operanden können Konstanten (in oktaler Schreibweise, wenn sie mit der Ziffer `0` beginnen, in hexadezimaler Schreibweise, wenn sie mit `0x` oder `0X` beginnen, sonst dezimal) sein oder die üblichen Variablen (**3.3.2 Variablen, Parameter** auf Seite 47).

Eine Schachtelung solcher arithmetischen Ausdrücke ist möglich.

Als *wahr* und *falsch* gilt wie in C und ebenso wie in `eval-Ausdrücken` (**3.3.10 Numerik light mit expr** `auf Seite 71`) „0 ist false, alles andere true", also entgegengesetzt der in der Shell sonst verwendeten Faustformel, daß ein Kommando *wahr* beziehungsweise *erfolgreich* mit dem Wert `0` signalisiert (**3.3.11 test-Ausdrücke, true, false** auf Seite 72).

Beispiel 11 *Die Zeile* `i=4; j=5; j=$(( $((k = 2*i+j)) -2 ))` *setzt* `i` *auf den Wert* `4`*,* `j` *auf anfangs* `5`*, dann* `k` *auf* `13` *und letztlich nochmals* `j` *auf* `11`*.*

Man muß also die Variablen nicht explizit durch ein vorangestelltes `$`-Zeichen expandieren, dürfte es aber: `$((k = 2*i+j))` kommt auf das selbe Ergebnis wie `$((k = 2*$i+$j))`; in letzterem Fall würden die Variablen in einem früheren Schritt durch ihren Wert ersetzt werden und die Berechnung erfolgt dann letztlich mit Konstanten (dem Ergebnis der Expansion).

Bei einer Zuweisung an eine Variable (oder beim Inkrementieren mit `++` oder Dekrementieren mit `--`) darf allerdings vor den Variablen kein `$` stehen, weil die Veränderung der expandierten Konstanten nicht möglich ist.

Neben dem Operator `$(( ))` gibt es noch die `let`-Anweisung, der man einen oder mehrere zu berechnende Ausdrücke als Argumente übergibt. Dabei muß man allerdings die Ausdrücke meistens durch *quoten* vor einer zu frühen Auswertung durch die Shell schützen, beispielsweise

```
let "k = 3 + 6"
```

Derselbe Ausdruck ohne Gänsefüßchen würde nicht funktionieren, weil vor dem Ausführen von `let` die Shell bereits das Gleichheitszeichen entdeckt und dadurch eine direkte Zuweisung versucht, ohne den arithmetischen Ausdruck auszuwerten. (Ohne Leerzeichen, also `let "k=3+6"`, würde es zufällig klappen, weil das Gleichheitszeichen für die Shell in einem Wort versteckt ist.)

Folgende Operatoren stehen zur Verfügung:

- `++`, `--`: Inkrement- und Dekrementoperatoren in Prä- und Postfixnotation wie in C erhöhen beziehungsweise erniedrigen den Wert einer Variablen.

  Steht der Operator vor dem Operanden (Präfixnotation, also `++`*Variable* oder `--`*Variable*), dann wird die Variable erst um `1` erhöht oder erniedrigt; der geänderte Wert wird gegebenenfalls für den umgebenden Ausdruck verwendet.

  Folglich gibt `i=4; j=$(( ++i )); echo i=$i j=$j` aus:
  `i=5 j=5`
  weil `i` erst erhöht wird, und dann der manipulierte Wert an `j` zugewiesen.

  In der Postfixnotation dagegen wird die Variable zwar ebenfalls erhöht oder erniedrigt, aber im umgebenden Ausdruck wird noch der alte Wert verwendet.

  In der Anweisung `i=4; j=$(( i++ )); echo i=$i j=$j` wird ausgegeben:
  `i=5 j=4`
  `i` wurde also von `4` auf `5` erhöht; an `j` wurde aber noch der alte Wert zugewiesen.

- `+` und `-` als unäre Operatoren (also mit nur einem nachfolgenden Operanden) dienen wie üblich als Vorzeichen.

- `!` ist die logische Verneinung (also ein Test auf gleich `0`), `~` ist die bitweise Negation

- `**` ist eine Exponentiation

- `*` dient zum Multiplizieren, `/` zum Dividieren und `%` berechnet den Divisionsrest (modulo)

- `+` und `-` als binäre Operatoren addieren beziehungsweise subtrahieren

- Mit `<<` kann man den linken Operanden um soviele Bit nach links schieben (*shift*), wie der rechte Operand angibt; von rechts her werden Nullen eingefügt. Dementsprechend schiebt `>>` nach rechts; von links her werden Nullen eingefügt (falls der linke Operand positiv ist) oder Einsen (wenn negativ).

- `<` und `>` vergleichen auf „kleiner“ und „größer“; `<=` und `>=` auf „kleiner gleich“ und „größer gleich“.

- `==` vergleicht zwei Werte auf Gleichheit, `!=` auf Ungleichheit

- `&` verknüpft zwei Operanden bitweise mit UND
- Mit `^` werden zwei Operanden bitweise XOR (exklusives ODER) verknüpft.
- Mit `|` werden zwei Operanden bitweise OR (ODER) verknüpft.
- `&&` ist ein logisches UND
- `||` ist ein logisches ODER
- Der ternäre Operator `?:` hat drei Operanden: *Ausdruck1* `?` *Ausdruck2* `:` *Ausdruck3* wertet *Ausdruck1* aus; wenn dieser *true* ergibt (also ungleich `0` ist), dann wird *Ausdruck2* berechnet und dessen Ergebnis geliefert; ansonsten *Ausdruck3*.

  Der Ausdruck `i=10 let "k=i?2:3"` weist demnach an `k` den Wert `2` zu.
- Das Gleichheitszeichen `=` dient, wie bereits in den Beispielen gezeigt, der Zuweisung.

  Ähnlich wie in C gibt es davon noch etliche Varianten, die eine Zuweisung an den ersten Operanden einer anderen Operation zusammenfassen, nämlich: `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`.

  Dabei entspricht `a+=b` dem Ausdruck `a=a+b`; analog ist `a-=b` identisch mit `a=a-b`, und so fort.
- Ebenfalls von C entlehnt wurde der Kommaoperator:
  Ausdruck1 `,` *Ausdruck2*
  führt *Ausdruck1* aus, ignoriert dessen Rückgabewert, und führt anschließend *Ausdruck2* aus. Dessen Rückgabewert steht dann für den gesamten Ausdruck.

  Weil solche Ausdrücke von links her zusammengefaßt werden, kann man auch längere Ketten bilden:
  Ausdruck1 `,` *Ausdruck2* `,` *Ausdruck3* `,` ... `,` *Ausdruck n*
  führt alle Ausdrücke aus, und liefert das Ergebnis des letzten.

3.3.10 Numerik light mit expr

`expr` ist ein internes Kommando, und liefert den String `"0"` für *false* (zur Standardausgabe), oder `"1"` (außer für den `:`-Operator) für *true*.

Achtung! Dies ist wie in C-Programmen (0 ist *false* und wie bei den eingebauten arithmetischen Ausdrücken (siehe **3.3.9 Numerische Ausdrücke $((...))** auf Seite 68), alles andere *true*); aber nun gerade entgegengesetzt zu `test` und den üblichen exit-Statuswerten (siehe auch **3.3.11 test-Ausdrücke, true, false** auf Seite 72).

Um das Ergebnis (für die Standardausgabe) zu bestimmen, werden die übergebenen Argumente rechnerisch (und ausschließlich ganzzahlig) ausgewertet. Dabei müssen alle Operanden und Operatoren als einzelne Argumente bei `expr` ankommen. Man kann die Summe von fünf und zwei also nicht mit `expr 5+2` bestimmen, sondern muß entsprechend Leerzeichen einfügen, damit einzelne Argumente entstehen: `expr 5 + 2`.

Neben den Grundrechenarten (`+`, `-`, `*`, und `/`) gibt es noch *modulo* (`%`).

Zwei Werte können mit `=`, `<`, `<=`, `>`, und `>=` verglichen werden.

Bei einem String *str* und einem regulären Ausdruck (RE) *re* liefert `expr` *`str`* `:` *`re`* dann `"0"`, wenn *re* nicht auf *str* zutrifft. Ansonsten wird die Anzahl der Zeichen ausgegeben, die in *str* mit dem *re* übereinstimmen.

3.3.11 test-Ausdrücke, true, false

`test` ist ein internes Kommando der Shell. Es wertet anhand seiner Argumente einen Ausdruck aus, und liefert (je nachdem, ob der ausdruck für *wahr/true* oder *falsch/false* erkannt wird) 0 oder 1 als exit-Status zurück. Dabei steht 0 für *true*, und 1 für *false*.

Achtung! Die selbe Konvention (0 für *true*, ungleich 0 für *false*) wird üblicherweise bei Rückgabewerten von Prozessen eingehalten. Beispielsweise wird ein Compiler mit dem Wert 0 terminieren, wenn er alle Quelltexte korrekt übersetzen konnte, und er wird mit einem anderen Wert terminieren, wenn Fehler auftreten. Wenn man eigene Programme schreibt, sollte man sich unbedingt an diese Konvention halten, weil dadurch diese Programme in Shellprogrammen in Schleifen und Fallunterscheidungen verwendet werden können.
Man darf sich aber nicht dadurch verwirren lassen, daß innerhalb von C genau die umgekehrte Regelung gilt: 0 ist *false*, alles andere ist *true*.
`expr` (siehe **3.3.10 Numerik light mit expr** auf Seite 71) und die eingebaute Arithmetik (**3.3.9 Numerische Ausdrücke $((...))** auf Seite 68) verhalten sich konform zu C, und damit entgegengesetzt zu `test`.

Für kleine Versuche kann man den Rückgabewert eines vorherigen `test`-Kommandos mit der vordefinierten Shellvariable `?` abfragen.

Beispiel 12 *Abfrage, ob 12 größer ist als 10:*

```
klaus@aw35:~ > test 12 -gt 10
klaus@aw35:~ > echo $?
0
```

Da der Ausdruck `12 -gt 10` wahrscheinlich mit *true* bewertet wird, liefert `test` den Wert 0 als exit-Status. Mit `echo $?` kann man diesen Wert sichtbar machen.

Ein einzeiliger Test wäre beispielsweise `test 12 -gt 10 && echo "stimmt!"`.

Für `test` gibt es noch eine alternative Schreibweise. Statt:
  `test` *Ausdruck*
kann auch geschrieben werden:
  `[` *Ausdruck* `]`
Dabei muß man darauf achten, daß innerhalb der eckigen Klammern zum Ausdruck hin jeweils mindestens ein Leerzeichen stehen muß.

Ausdruck	*liefert:*
`-n` *string*	t, wenn Stringlänge>0
`-z` *string*	f, wenn Stringlänge>0
`-b` *file*	t, wenn Datei *file* existiert und block dev. ist
`-c` *file*	t, wenn Datei *file* existiert und character dev. ist
`-d` *file*	t, wenn Verzeichnis *file* existiert
`-e` *file*	t, wenn Datei *file* existiert
`-f` *file*	t, wenn reguläre Datei *file* existiert
`-g` *file*	t, wenn Datei *file* existiert und setguid gesetzt ist
`-G` *file*	t, wenn Datei *file* existiert und zu eff. gid gehört
`-k` *file*	t, wenn Datei *file* existiert und sticky bit gesetzt ist
`-h` *file*	t, wenn Datei *file* existiert und symb. Link ist
`-L` *file*	t, wenn Datei *file* existiert und symb. Link ist
`-O` *file*	t, wenn Datei *file* existiert und zu eff. uid gehört
`-u` *file*	t, wenn Datei *file* existiert und suid gesetzt ist
`-p` *file*	t, wenn Datei *file* existiert und named pipe ist
`-r` *file*	t, wenn Datei *file* existiert und lesbar ist
`-s` *file*	t, wenn Datei *file* existiert und größer 0 Bytes ist
`-S` *file*	t, wenn Datei *file* existiert und ein socket ist
`-t` *fd*	t, wenn file descriptor *fd* an einem Terminal geöffnet
`-t`	t, wenn file descriptor 1 an einem Terminal geöffnet
`-w` *file*	t, wenn Datei *file* existiert und beschreibbar ist
`-x` *file*	t, wenn Datei *file* existiert und ausführbar ist
f1 `-nt` *f2*	t, wenn Datei *f1* neuer ist als *f2* ist
f1 `-ef` *f2*	t, wenn *f1* und *f2* identischen inode haben (wenn also *f1* und *f2* den gleichen Namen ergeben, oder hard links auf eine Datei sind)

(Anmerkung: Für `-n` *String* kann man auch einfach nur *String* schreiben, also `-n` einfach weglassen.)

`test` kennt einige numerische Vergleiche (`-eq` (gleich), `-ge` (größer oder gleich), `-gt` (größer), `-le` (kleiner oder gleich), `-lt` (kleiner), `-ne` (ungleich)), sowie Stringvergleiche (`=`, `!=`).

Für die numerischen Vergleiche werden aus den Argumenten (Strings) ganze Zahlen gelesen, die dann für die Vergleiche herangezogen werden.

Als Sonderfall kann man laut `info bash` *als ganzzahliges Argument auch die Länge eines Strings* s *verwenden, indem man* `-l` s *angibt. Das funktioniert aber bei mir nicht. Kann aber auch an mir liegen.*

Man kann Ausdrücke mit runden Klammern zusammenfassen, um eine Auswertungsreihenfolge zu erzwingen, und Teilausdrücke mit `-a` (logisches UND) und `-o` (logisches ODER) vergleichen.

Mit einem vorangestellten Ausrufezeichen (`!`) kann man einen Ausdruck negieren.

Wenn man bestimmte Rückgabewerte produzieren will, dann kann man dafür `true` oder `false` aufrufen. Diese (internen) Kommandos liefern 0 beziehungsweise 1 zurück.

Zum Probieren in Shellprogrammen ist manchmal folgendes Programm `gibmir.c` ganz nützlich:

```
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include <string.h>

int main( int nargs, char **args )
{
  if( nargs==2 )
    {
      return strtol( args[1], NULL, 0 );
    }

  return 0;
} /* main( int nargs, char **args ) */
```

Dieses Programm gibt seinen ersten und einzigen Parameter als Rückgabewert (exit-Status) zurück. Damit entspricht `gibmir 0` einem `true`, und `gibmir 1` einem `false`.

3.3.12 Schleifen

Es gibt vier Schleifentypen in der Shell: `for`, `while`, `until`, und `select`.

Alle Typen können vorzeitig mit `break` verlassen werden, oder mit `continue` kann mit dem nächsten Durchlauf beziehungsweise dem nächsten Prüfen der Bedingung fortgefahren werden (es wird also nur der Rest des Schleifenrumpfes übersprungen).

Wenn `break` oder `continue` ohne Argument angegeben werden, dann betreffen sie nur die innerste umgebende Schleife. Man kann sie aber auch mit einem positiven Argument aufrufen; dann betreffen sie entsprechend soviele der innersten umgebenden Schleifen.

3.3.12.1 for-Schleife

Eine `for`-Schleife unterscheidet sich von ähnlichen Schleifen in anderen Sprachen dadurch, daß sie nicht Anfangs- und Endwerte von Schleifenzählern beziehungsweise irgendwelche Abbruchbedingungen bekommt, und dann die Zwischenwerte selbst berechnet, sondern vielmehr muß man ihr alle Werte geben, für die die Schleife ausgeführt werden soll. Also nicht *für Werte von i von 1 bis 5 in Einserschritten*, sondern stattdessen *für die Werte (*`"1"`*,* `"2"`*,* `"3"`*,* `"4"`*, und* `"5"`*).*

Dabei sind die angegebenen Werte in Wirklichkeit wieder gar keine Zahlen, sondern nur Strings.

Damit ist die `for`-Schleife keine Zählschleife, wie man vielleicht erwartet hätte, sondern eine Schleife über eine Liste von Strings.

Eine reine Zählschleife kann man dagegen mit `while` (siehe **3.3.12.2 while-Schleife** auf Seite 77) und beispielsweise `expr` (**3.3.10 Numerik light mit expr** auf Seite 71) bauen.

Die Struktur einer solchen Schleife sieht so aus:

```
for name in ersterwert zweiterwert dritterwert...
do
   # Verwenden von $name
   # Beim 1. Durchlauf hat $name den Wert "ersterwert",
   # beim 2. Durchlauf hat $name den Wert "zweiterwert", ...
done
```

Beispiel 13 *Anzeige von 4 Dateinamen, und Löschen dieser Dateien:*
```
for f in a.f77 b.cpp a.o b.cpp
do
    echo Ich lösche die Datei $f
    rm $f
done
```

Man kann eine `for`-Schleife auch einzeilig schreiben. Das ist besonders dann interessant, wenn man nicht in einem Skript arbeitet, sondern die Schleife interaktiv eingibt. Die obige Schleife würde so aussehen:
```
for f in a.f77 b.cpp a.o b.cpp; do echo Ich lösche die Datei $f; rm $f; done
```
Man muß dabei allerdings an einigen Stellen mit einem Semikolon (`;`) der Shell einen Hinweis geben, wo die Liste der Schleifenwerte beziehungsweise die Liste der Argumente eines Kommandos endet.

Wenn man das Schlüsselwort `in` und die Liste der Schleifenwerte wegläßt, dann läuft die Schleife über alle Werte in `$*` (siehe **3.3.3 Vordefinierte und spezielle Variablen, shift, set** auf Seite 53).

Die Schleife:
```
for name
do
   # Verwenden von $name
done
```
ist also identisch mit dieser Version:
```
for name in $*
do
   # Verwenden von $name
done
```

Bevor die Schleife ausgeführt wird, wird in der Liste der Schleifenwerte eine Dateinamenexpansion durchgeführt. Dies ist sehr praktisch, wenn man eine Schleife über bestimmte Dateien laufen lassen will.

Beispiel 14 *Löschen aller Dateien mit der Endung* `.bak` *im aktuellen Verzeichnis:*
```
for f in *.bak
do
    rm $f
done
```

Wenn man jetzt nicht nur die Dateien eines Verzeichnisses anfassen will, sondern auch in allen Unterverzeichnissen, dann muß dafür wie immer `find` herhalten.

Beispiel 15 *Löschen aller Dateien mit der Endung* `.bak` *im aktuellen Verzeichnis und allen darunter liegenden Verzeichnissen:*

```
for f in `find ~/aweho000411 -name "*.bak"`;
do
    rm $f
done
```

3.3.12.2 while-Schleife

Eine `while`-Schleife führt die Anweisungen im Schleifenrumpf solange aus, wie ein Test des angegebenen Ausdrucks (vor Eintritt in den Schleifenrumpf) 0, also *false* ergibt.

Als Bedingung kann man jedes Kommando nehmen; häufig wird dafür `test` beziehungsweise `[ ... ]` verwendet (siehe **3.3.11 test-Ausdrücke, true, false** auf Seite 72).

Die Struktur einer solchen Schleife sieht so aus:

```
while Ausdruck
do
   # Schleifenrumpf
   Kommando1
   ...
done
```

Zusammen mit `expr` (siehe **3.3.10 Numerik light mit expr** auf Seite 71) kann man damit auch echte Zählschleifen bauen.

Beispiel 16 *Das folgende Beispiel gibt die Zahlen von* `-1` *bis* `10` *aus:*

```
i=-1
while [ $i -le 10 ]
do
        echo $i
        i=`expr $i + 1`
```

```
done
```

Beispiel 17 *Statt* `expr` *kann man wahlweise auch* `bc` *nehmen. Die Schleife würde dann so aussehen:*

```
i=-1
while [ ‘echo $i "<"= 10 | bc‘ != 0 ]
do
        echo $i
        i=‘echo $i + 1 | bc‘
done
```

Beispiel 18 *Eine letzte Variante derselben Schleife nutzt weder* `expr` *noch* `bc`*, sondern die interne Arithmetik der* `bash`*. Dieses Skript ist deshalb nur mit der* `bash` *lauffähig!*

```
#!/bin/bash
# nur BASH!!!
i=-1
while [ $i -le 10 ]
do
        echo $i
        i=$[ $i + 1 ] # oder:  i=$(($i+1)) oder : $((i=$i+1))
done
```

Die einzeilige Version sieht so aus:
`while` Ausdruck; `do` *Kommando1*;*...*; `done`

3.3.12.3 until-Schleife

Die `until`-Schleife entspricht der `while`-Schleife; im Gegensatz dazu wird die Bedingung aber negiert verwendet (die Schleife läuft nicht -wie bei `while`- *solange* die Bedingung `true` (0) liefert, sondern vielmehr *solange sie NICHT* `true` (0) liefert).

Siehe **3.3.12.2 while-Schleife** auf Seite 77.

Die Struktur einer `until`-Schleife sieht so aus:

```
until Ausdruck
do
   # Schleifenrumpf
   Kommando1
   ...
done
```

Die einzeilige Version sieht so aus:
`until` Ausdruck; `do` *Kommando1*;*...*; `done`

3.3.12.4 select-Schleife

Mit `select` kann man schnell und einfach ein kleines Menü zaubern. Sozusagen fast food für die Shellprogrammierung.

`select` zeigt in einer Endlosschleife eine Auswahl für den Benutzer an. Die Schleife verläßt man bei Bedarf mit `break` im Skript.

Die Struktur ähnelt sehr einer `for`-Schleife:

```
for name in ersteauswahl zweiteauswahl dritteauswahl...
do
   # In $name steht der Text der Auswahl
done
```

Beispiel 19 *Es werden ein paar (nicht sehr intelligente) Auswahlmöglichkeiten angeboten. Die Schleife wird verlassen, wenn der Benutzer eine* `6` *(für* `ende`*) eingibt:*

```
#!/bin/sh
echo soll alles eventuell nicht formatiert werden\?
select name in ja nein "weiß nicht" "keine Hilfe!" "vielleicht" "ende"
do
    echo ausgewählt wurde $name
    if [ "$name" = ende ]
    then
        break
    fi
done
echo fertig
```

Ein Beispieldialog könnte so aussehen, wenn der Benutzer nacheinander die Werte `1`*,* `4`*, und* `6` *eingibt:*

```
soll alles eventuell nicht formatiert werden?
1) ja           3) weiß nicht    5) vielleicht
2) nein         4) keine Hilfe!  6) ende
#? 1
ausgewählt wurde ja
1) ja           3) weiß nicht    5) vielleicht
2) nein         4) keine Hilfe!  6) ende
#? 4
ausgewählt wurde keine Hilfe!
1) ja           3) weiß nicht    5) vielleicht
2) nein         4) keine Hilfe!  6) ende
#? 6
ausgewählt wurde ende
fertig
```

3.3.13 Fallunterscheidungen

Für die bedingte Ausführung von Teilen eines Skripts kann man zwischen `if` und `case` wählen. Ein `if` verwendet den Rückgabewert (exit-Status) eines Ausdrucks als Bedingung, während `case` einen String nacheinander gegen mehrere Muster prüft, und beim ersten Treffer den entsprechenden Zweig ausführt.

3.3.13.1 if

Mit `if` kann man einen Teil eines Skripts in Abhängigkeit einer Bedingung ausführen lassen. Die Angabe mehrerer Alternativen mit `elif` und maximal ein `else` ist möglich. Die `if`-Anweisung wird mit einem `fi` beendet.

Die Struktur kann also etwas variieren. Die einfachste Form sieht so aus:

```
if Bedingung
then
    kommando1
    kommando2
    ...
fi
```

In voller Schönheit hat die Fallunterscheidung diese Form:

```
if EineBedingung
then
    kommando1
    kommando2
    ...
elif NochEineBedingung
then
    nocheinkommando1
    nocheinkommando2
    ...
...
else
    undnocheinkommando1
    undnocheinkommando2
    ...
fi
```

Als *Bedingung* wird häufig `test` beziehungsweise `[ ... ]` verwendet, siehe **3.3.11 test-Ausdrücke, true, false** auf Seite 72.

Beispiel 20 *Kompilieren eines Programms mit Erfolgsmeldung:*

```
if gcc meinprg.c -o meinprg
then
   echo "hat geklappt!"
else
   echo "Schade!"
fi
```

3.3.13.2 case

Mit der `case`-Anweisung kann man einen String *Ausdruck* gegen mehrere Muster testen lassen. Die Kommandos, die zum ersten passenden Muster gehören, werden dann ausgeführt:

```
case Ausdruck in
     muster1) Kommando1
                  ...
                  ;;
     muster2) AnderesKommando1
                  ...
                  ;;
     ...
esac
```

Jeder Block, der zu einem Muster gehört, muß mit doppelten Semikolon `;;` abgeschlossen werden.

Als Muster kann man beliebige Strings verwenden, wobei `*` auf beliebige Zeichen (auch auf keines) zutrifft, `?` auf genau ein Zeichen, und mehrere Muster mit einem `|` (für logisches ODER) zusammengefaßt werden können.

Beispiel 21 *Für alle Dateien im aktuellen Verzeichnis wird versucht, den Dateityp zu erraten:*

```
for f in *
do
     case $f in
         *.c|*.cpp|*.h)
                  echo $f ist ein Quelltext.
                  ;;
         *.txt)
                  echo $f ist eine Textdatei.
                  ;;
         *)       echo $f hat einen unbekannten Dateityp"
                  ;;
     esac
done
```

3.4 Eingabe vom Benutzer lesen: read

Mit dem Kommando `read` kann man vom Benutzer Eingaben (als Text) lesen. Mit jedem `read` wir genau eine Zeile gelesen. Es wird also alles gelesen, was der Benutzer eingibt, bis zum abschließenden *line feed* (*Enter*- oder *Return*-Taste).

Hinter dem `read` gibt man eine oder mehrere Variablen, die mit der Eingabe gefüllt werden. Das erste Wort der Eingabe wird der ersten angegeben Variablen zugewiesen, das zweite Wort der zweiten Variablen, und so weiter. Wenn mehr Variablen angegeben sind, als die Eingabe Wörter enthält, dann sind die überzähligen Variablen leer. Wenn die Eingabe mehr Wörter enthält, als Variablen angegeben sind, dann erhält die letzte Variable den ganzen Reste der Eingabezeile.

Als Trennzeichen der eingegebenen Wörter werden alle Zeichen verstanden, die in der Variablen `IFS` abgelegt sind (normalerweise das Leerzeichen).

Wenn man gar keine Variablen angibt, dann wird die eingegebene Zeile in der Variablen `REPLY` gespeichert.

Beispiel 22 *Einlesen von drei Werten in drei Variablen:*

```
klaus@aw35:~ > read a b c
eins 222 drei
klaus@aw35:~ > echo $a
eins
klaus@aw35:~ > echo $b
222
klaus@aw35:~ > echo $c
drei
```

Beispiel 23 *Einlesen eines Werts in drei Variablen:*

```
klaus@aw35:~ > read a b c
eins
klaus@aw35:~ > echo $a
eins
klaus@aw35:~ > echo $b

klaus@aw35:~ > echo $c
```

Beispiel 24 *Einlesen von fünf Werten in drei Variablen:*

```
klaus@aw35:~ > read a b c
eins zwei 333 Issos Keilerei
klaus@aw35:~ > echo $a
eins
klaus@aw35:~ > echo $b
zwei
klaus@aw35:~ > echo $c
333 Issos Keilerei
```

Mit der Option `-p`*prompt* kann man einen Eingabeprompt angeben, der angezeigt werden soll.

Beispiel 25 *Einlesen von drei Werten in drei Variablen (mit Eingabeprompt* `Bitte drei Werte: ` *):*

```
klaus@aw35:~ > read -p "Bitte drei Werte: " a b c
Bitte drei Werte: eins 222 drei
klaus@aw35:~ > echo $a
eins
klaus@aw35:~ > echo $b
222
klaus@aw35:~ > echo $c
drei
```

Beispiel 26 *Einlesen einer Zeile in die Variable* `REPLY`*:*

```
klaus@aw35:~ > read -p "Bitte Ihre Meinung: "
Bitte Ihre Meinung: ich spende gern
klaus@aw35:~ > echo $REPLY
ich spende gern
```

Beispiel 27 *Mit* `read` *kann man natürlich aus einer Datei lesen, indem man die Standardeingabe umlenkt:*

```
read a <dateiname
```

liest die erste Zeile der angegebenen Datei in die Variable `a`.

Will man mehrere Zeilen lesen, dann muß man etwas mitdenken. Denn so:

```
read a <dateiname
read b <dateiname
read c <dateiname
```

liest man nicht etwa die ersten drei Zeilen nach `ab`, `b`, *und* `c`. *Vielmehr wird mit jedem* `read` *die Datei neu geöffnet, und alle drei Variablen bekommen den Inhalt der ersten Zeile.*
Schreibt man dagegen:

```
{ read a; read b; read c; }<dateiname
```

dann klappt's auch mit dem Nachbarn: `a` *erhält die erste Zeile,* `b` *die zweite, und* `c` *die dritte.*

Nur `bash`: Wenn man die Option `-e` benutzt, dann wird für die Eingabe der *readline*-Mechanismus genutzt, das heißt der Benutzer kann sich mit den Pfeiltasten durch die letzten Eingaben blättern, oder mit *Strg-r* (`^R`) rückwärts beziehungsweise mit *Strg-s* (`^R`) vorwärts in der letzten Eingabe suchen (wie in der Eingabezeile der `bash` sonst auch.)

3.5 Signale

3.5.1 Was sind Signale?

Unabhängig von der Shell sind *Signale* eine Möglichkeit, Information zwischen Prozessen auszutauschen.

Innerhalb eines (Unix-)Systems kann jeder Prozeß jedem anderen oder sich selbst Signale senden. Ein Versenden zwischen Prozessen auf verschiedenen Rechnern ist nicht möglich.

Der Informationsgehalt eines Signals ist gering: 1 Bit (Signal vorhanden oder nicht).

Es gibt (je nach System etwa 20 bis 30) verschiedene Signale, die verschickt werden können. Jedem dieser Signale ist ein Zahlenwert zugeordnet. Davon sind einige für bestimmte Zwecke reserviert (beispielsweise Abschießen eines Prozesses ohne weitere Diskussion mit `SIGKILL` mit der Nummer 9). Andere kann man für eigene Zwecke benutzen. Extra für eigene Mechanismen reservierte Signale sind `SIGUSR1` und `SIGUSR2`; sie werden vom System überhaupt nicht verwendet. Aber die meisten anderen Signale können auch mehr oder weniger sinnvoll nach eigenem Gusto benutzt werden.

Ein oft genutztes Signal ist `SIGINT`. Startet man beispielsweise ein Programm aus der Shell heraus, und es macht Unfug (möglicherweise habe ich mal wieder sauber eine Endlosschleife programmiert?), dann kann man es (meistens) mit Strg-C (`^C`) abbrechen. Wieso endet das Programm aber? Die Shell, von der man das Programm startet, „fängt" das `^C` ab, und schickt an das laufende Programm daraufhin das Signal `SIGINT` (*signal interrupt*). Die meisten Programme reagieren darauf mit einem Abbruch.

Ein häufig „mißbrauchtes" Signal ist `SIGHUP`. Es ist gedacht für Programme, die über eine serielle Leitung ein Modem oder etwas ähnliches betreiben. Wenn die Verbindung unterbrochen wird, weil beispielsweise die Gegenseite auflegt (*hang up*), dann wird dem Prozeß, der das Terminal geöffnet hat, das Signal `SIGHUP` (*signal hang up*) geschickt. Dieser Prozeß braucht dann gar nicht mehr versuchen, von dieser Verbindung zu lesen; das würde den Prozeß blockieren, weil ja keine Zeichen mehr kommen können. Weil die unter Unix häufigen „Dämonen" (Hintergrundprogramme wie FTP-Server `ftpd`, WEB/HTTP-Server `httpd`, etc.) zwangsläufig mit keinem Terminal direkt verbunden sind (sie laufen ja im Hintergrund, ohne Benutzereingriff) reagieren solche Programme auf das Signal `SIGHUP`, indem sie ihre Konfigurationsdatei(en) neu einlesen. Ändert man beispielsweise etwas an `/etc/exports` (in dieser Datei stehen die Verzeichnisse, die über den NFS-Dämon, also mit dem Programm `nfsd` im Netz freigeben werden sollen), dann würden sich die Änderungen erst nach dem nächsten Starten des `nfsd` auswirken. Um den `nfsd` davon zu informieren, daß sich etwas geändert hat, schickt man ihm das Signal `SIGHUP`. Die meisten Implementationen solcher Programme reagieren auf dieses Signal, indem sie ihre Konfigurationsdateien neu einlesen und berücksichtigen.

Halbwegs portabel ist die Verwendung der nach POSIX.1 definierten Signale. Zumindest diese werden von allen aktuellen Unixversionen unterstützt (Linux, BSD 4.3, SunOS, Solaris, HP-UX,...). Je nach System ist aber die Zuordnung der Signale zu ihren Nummern unterschiedlich; deshalb muß man die symbolischen Namen verwenden, also beispielsweise `SIGKILL` statt `9`.

Im Folgenden die Ausgabe von `man 7 signal` mit den möglichen Signalen:

```
SIGNAL(7)           Linux Programmer's Manual           SIGNAL(7)


NAME
       signal - list of available signals

DESCRIPTION
       Linux  supports  the  signals listed below. Several signal
       numbers are  architecture  dependent.  First  the  signals
       described in POSIX.1.


       Signal     Value     Action   Comment
       -------------------------------------------------------------------------
       SIGHUP        1        A      Hangup detected on controlling terminal
                                     or death of controlling process
       SIGINT        2        A      Interrupt from keyboard
       SIGQUIT       3        A      Quit from keyboard
       SIGILL        4        A      Illegal Instruction
       SIGABRT       6        C      Abort signal from abort(3)
       SIGFPE        8        C      Floating point exception
       SIGKILL       9       AEF     Kill signal
       SIGSEGV      11        C      Invalid memory reference
       SIGPIPE      13        A      Broken pipe: write to pipe with no readers
       SIGALRM      14        A      Timer signal from alarm(2)
       SIGTERM      15        A      Termination signal
       SIGUSR1   30,10,16     A      User-defined signal 1
       SIGUSR2   31,12,17     A      User-defined signal 2
       SIGCHLD   20,17,18     B      Child stopped or terminated
       SIGCONT   19,18,25            Continue if stopped
       SIGSTOP   17,19,23    DEF     Stop process
```

SIGTSTP	18,20,24	D	Stop typed at tty
SIGTTIN	21,21,26	D	tty input for background process
SIGTTOU	22,22,27	D	tty output for background process

Next various other signals.

Signal	Value	Action	Comment
SIGTRAP	5	CG	Trace/breakpoint trap
SIGIOT	6	CG	IOT trap. A synonym for SIGABRT
SIGEMT	7,-,7	G	
SIGBUS	10,7,10	AG	Bus error
SIGSYS	12,-,12	G	Bad argument to routine (SVID)
SIGSTKFLT	-,16,-	AG	Stack fault on coprocessor
SIGURG	16,23,21	BG	Urgent condition on socket (4.2 BSD)
SIGIO	23,29,22	AG	I/O now possible (4.2 BSD)
SIGPOLL		AG	A synonym for SIGIO (System V)
SIGCLD	-,-,18	G	A synonym for SIGCHLD
SIGXCPU	24,24,30	AG	CPU time limit exceeded (4.2 BSD)
SIGXFSZ	25,25,31	AG	File size limit exceeded (4.2 BSD)
SIGVTALRM	26,26,28	AG	Virtual alarm clock (4.2 BSD)
SIGPROF	27,27,29	AG	Profile alarm clock
SIGPWR	29,30,19	AG	Power failure (System V)
SIGINFO	29,-,-	G	A synonym for SIGPWR
SIGLOST	-,-,-	AG	File lock lost
SIGWINCH	28,28,20	BG	Window resize signal (4.3 BSD, Sun)
SIGUNUSED	-,31,-	AG	Unused signal

(Here - denotes that a signal is absent; there where three values are given, the first one is usually valid for alpha and sparc, the middle one for i386 and ppc, the last one for mips. Signal 29 is SIGINFO / SIGPWR on an alpha but SIGLOST on a sparc.)

The letters in the "Action" column have the following meanings:

A Default action is to terminate the process.

B Default action is to ignore the signal.

C Default action is to dump core.

D Default action is to stop the process.

E Signal cannot be caught.

F Signal cannot be ignored.

G Not a POSIX.1 conformant signal.

```
CONFORMING TO
       POSIX.1

BUGS
       SIGIO and SIGLOST have the same value.  The latter is com
       mented  out in the kernel source, but the build process of
       some software still thinks that signal 29 is SIGLOST.

SEE ALSO
       kill(1), kill(2), setitimer(2)
```

3.5.2 Versenden von Signalen

Der Versender eines Signals hat keine Kontrolle darüber, wie der Empfänger reagiert.

Das Signal wird an einen bestimmten Prozeß (anhand dessen *process id*) geschickt, und fertig.

Aus Programmen heraus kann man Signale mit der Funktion `kill()` schicken (siehe `man 2 kill`), oder an sich selbst (an die eigene *process id*) mit der ANSI-C-Funktion `raise()`.

Beispiel 28 *Schicken eines Signals an den aufrufenden Prozeß* `papa`*:*

```
#include <unistd.h>
...
pid_t papa;
// die process id meines Aufrufers beschaffen:
papa = getppid();
// Das Signal SIGUSR1 an meinen Aufrufer schicken:
kill( papa, SIGUSR1 );
```

Von der Shell aus (interaktiv oder aus einem Skript) kann man ebenfalls mit einem Aufruf des Kommandos `kill` ein Signal verschicken (`man 1 kill`).

Beispiel 29 *Ebenfalls ein Signal* `SIGUSR1` *an meinen Aufrufer schicken:*

```
#!/bin/sh
kill -s SIGUSR1 $PPID
```

3.5.3 Empfangen von Signalen

Der Empfänger hat keine Möglichkeit, den Sender ausfindig zu machen. Das Signal ist beim Empfänger ein asynchrones Ereignis; es kann zu irgendeinem beliebigen Zeitpunkt in Form eines Interrupts auftreten. Der Empfänger hat keinerlei Einfluß auf den Zeitpunkt. Das heißt, daß der gerade ablaufende Maschinencode unterbrochen wird, vom Betriebssystem der gesamte Prozeßstatus (Stack, Register, etc.) irgendwohin gerettet wird, und dann das Signal vom Empfängerprozess verarbeitet werden kann. Danach wird der Prozeß an der unterbrochenen Stelle wieder aufgesetzt, und läuft weiter. Es ist sogar möglich, daß ein Interrupt auftritt, während der Empfänger auf die Rückkehr einer aufgerufenen Betriebssystemfunktion wartet (z.B. `read()` aus einer Datei oder von einem Socket). Gerade an diesem Punkt unterscheidet sich das Verhalten verschiedener Unixversionen.

Signale werden nicht gepuffert, und es können daher auch Signale verlorengehen. Wenn ein bestimmtes Signal zum zweitenmal gesendet wird, bevor die Reaktion auf das erste beendet ist, dann geht das zweite verloren. Denn der Kernel hat je Prozeß und Signaltyp nur jeweils ein Bit, in dem vermerkt wird, ob für diesen Prozeß ein bestimmtes Signal anliegt, oder nicht. Man kann Signale also nicht zum Zählen von Ereignissen nehmen (zumindest dann nicht, wenn es genau gehen soll)!

Ein Prozeß kann für jedes der möglichen Signale (außer `SIGKILL` und `SIGSTOP`) steuern, wie er darauf reagieren möchte: entweder ignorieren, oder gezielt darauf reagieren. Für jedes Signal gibt es ein Standardverhalten (*default signal handler*). Das ist systemweit für jedes Signal entweder auf *ignorieren* oder *Beenden des Programms* gesetzt. Für jeden Prozeß existiert eine individuelle Tabelle, wie auf jedes Signal reagiert werden soll. Diese Tabelle ist anfangs mit dem Standardverhalten für jedes Signal vorbelegt, und kann zur Laufzeit modifziert werden.

Um auf ein Signal reagieren zu können, muß man vorher (!) in C einen *signal handler* installieren. Das ist keine große Sache; man ruft nur die Funktion `signal()` auf, und übergibt ihr die betreffende Signalnummer (beispielsweise `SIGUSR1`) und die Adresse einer Funktion, die beim Eintreffen eines Signals aufgerufen werden soll. Diese Funktion ist der *signal handler*. Sie erhält einen Parameter, nämlich den Typ des aufgetretenen Signals. So kann man eine Funktion für verschiedene Signale verwenden wenn man will. Siehe `man 2 signal`.

Ein *signal handler* sollte nicht lange laufen, weil er weitere Signale blockieren kann. Wenn auf ein Signal hin etwas aufwendiges passieren soll (z.B. Öffnen und Lesen von Dateien), dann sollte man im *signal handler* nur vermerken, daß das Signal empfangen wurde (beispielsweise in einer globalen Variablen), und zu einem späteren Zeitpunkt anhand dieser Variablen entsprechend reagieren (beispielsweise in einer Hauptschleife des Programms, oder bei ohnehin regelmäßig auftretenden Timerereignissen).

Beispiel 30 *Installation der Funktion* `signal_f()` *als* signal handler*:*

```
static int KonfigurationMussNeuGelesenWerden = 0;
...
/* Bei einem SIGIO soll das Programm beendet werden,
 * bei SIGPIPE soll der socket geschlossen werden,
 * bei SIGHUP soll in (KonfigurationMussNeuGelesenWerden)
 * vermerkt werden, daß die Konfigurationsdatei neu gelesen
 * werden soll.
 */
void signal_f( int s )
{
  switch( s )
    {
    case SIGPIPE:
      {
        schliesseosocket();
      }
      break;

    case SIGIO:
      {
        schliesseosocket();
        exit( 2 );
      }
      break;

    case SIGHUP:
      KonfigurationMussNeuGelesenWerden++;
      break;

    default :

      break;
    } /* Ende switch */
} /* Ende signal_f() */
...
/* Signalhandler installieren (damit bei einem
```

```
     * SIGIO nicht das Programm beendet wird, sondern
     * nur in signal_f() der socket geschlossen wird, etc.):
     */
    signal( SIGPIPE, signal_f );
    signal( SIGIO, signal_f );
    signal( SIGHUP, signal_f );
    ...
    /* Hauptschleife des Programms (je nach Systemumgebung, hier nur
     * fiktiv):
     */
    while( ( E=HoleEreignis() )!=EreignisAbbruch )
    {
      switch( E )
        {
        case EreignisTastegedrueckt:
          VerarbeiteTastenDruck();
          break;

        case EreignisMausgedrueckt:
          MausQuietscht();
          break;

        case EreignisTimer:
          if( KonfigurationMussNeuGelesenWerden!=0 )
            {
              /* es ist ein Signal SIGHUP aufgetreten:
               */
              KonfigurationMussNeuGelesenWerden = 0;
              LeseKonfiguration();
            }
          break;

        default :
          break;
        }
    }
```

Wenn dann später dieses Signal auftritt, wird der installierte *signal handler*, also die eigene Funktion aufgerufen (hier `signal_f()`).

Um einen solchen *signal handler* in der Shell zu definieren, gibt es das (zwangsläufig interne) Kommando (nein, nicht `signal`, sondern) `trap` (siehe dazu `info bash`, darin `Bourne Shell Builtins`).

`trap` erhält ein Kommando mit den zugehörigen Argumenten (als String), das beim späteren Eintreffen eines Signals ausgeführt werden soll, und die betreffende Signalnummer.

Mit der Option `-p` listet `trap` die gesetzten *signal handler*.

Beispiel 31 *Das folgende Beispiel setzt eine eigene Behandlung für die Signale* `SIGINT`*,* `SIGKILL`*,* `SIGUSR1`*,* `SIGUSR2`*. Das Signal* `SIGKILL` *kann man nicht mit einer eigenen Behandlung übersteuern; deshalb bewirkt* `trap ... SIGKILL` *nichts.*

Dann gibt das Skript mit `trap -p` *aus, welche Signalbehandlungen gesetzt sind, und geht in eine Endlosschleife. Damit nicht die ganze CPU-Zeit verballert wird, habe ich noch ein* `sleep` *eingebaut. Normalerweise könnte man die Endlosschleife durch Drücken von* `^C` *knacken (das bewirkt das Senden von* `SIGINT`*), aber dieses Skript fängt* `SIGINT` *ab, und gibt dafür nur aus:*

```
ich habe hier ein Signal SIGINT
```

Beispielausgabe:

```
klaus@aw33:~/s2 > ../t
ursprünglich abgefangene Signale:
jetzt abgefangene Signale:
trap -- 'machwaswennSignal SIGINT' SIGINT
trap -- 'machwaswennSignal SIGKILL' SIGKILL
trap -- 'machwaswennSignal SIGUSR1' SIGUSR1
trap -- 'machwaswennSignal SIGUSR2' SIGUSR2
schnarch... (Prozeß 9411)
schnarch... (Prozeß 9411)
schnarch... (Prozeß 9411)
ich habe hier ein Signal SIGINT
schnarch... (Prozeß 9411)
schnarch... (Prozeß 9411)
schnarch... (Prozeß 9411)
...
```

Von einem anderen Terminal aus kann man andere Signale schicken, beispielsweise `kill -s SIGUSR1 9411` *würde den Text* `ich habe hier ein Signal SIGUSR1` *produzieren. Beenden kann man diesen Amokläufer nur noch durch* `kill -s SIGKILL 9411`*.*

Hier das Skript des Amokläufers:

```
#!/bin/sh

# Diese Funktion soll aufgerufen werden, wenn ein Signal kommt:
machwaswennSignal()
{
  echo ich habe hier ein Signal $1
}

# bisher gesetzte Signale ausgeben:
echo ursprünglich abgefangene Signale:
trap -p

# eigene Signalbehandlung setzen.
# Das Setzen von SIGKILL ist überflüssig,
# weil dieses Signal nicht angefangen werden kann.
trap  'machwaswennSignal SIGINT' SIGINT
trap  'machwaswennSignal SIGKILL' SIGKILL
trap  'machwaswennSignal SIGUSR1' SIGUSR1
trap  'machwaswennSignal SIGUSR2' SIGUSR2

echo jetzt abgefangene Signale:
trap -p

# Endlosschleife: Ausgabe der eigenen Process id,
# und dann nochmal kurz schlafen legen:
while true
do
    echo "schnarch... (Prozeß $$)"
    sleep 1
done
```

3.6 Denksportaufgaben

Die folgenden Aufgaben sind direkt aus dem wahren Leben entstanden. Die Musterlösungen im nächsten Abschnitt sind Skripte, die ich für mich selbst geschrieben habe, und zu denen ich für diesen Abschnitt nur die Aufgabenstellung neu geschrieben habe.

Wie immer beim Programmieren gibt es für jede Aufgabe viele mögliche Lösungen. Meine muß sicher nicht die beste oder gar die einzig richtige sein. Aber zumindest als Denkanstoß oder zum Abkupfern stehen sie hier.

3.6.1 Aufgabe: Abgleichen von Verzeichnissen: updt

Gesucht ist ein Skript `updt` (für *update*), mit dessen Hilfe Verzeichnisse auf einen gleichen Stand gebracht werden können.

Anwendungsbeispiel:

Ich habe auf einem Rechner (`aw33`), der immer zuhause steht, alle Programme und Dokumente liegen, die ich im Lauf der Jahre selbst erstellt habe, eingescannte Bilder, viele Treiber von anderen gesaugt, Onlinehandbücher, und so weiter, in einem riesigen Verzeichnis liegen (`/home/archiv`). Andere Rechner im lokalen Netz mounten dieses Verzeichnis über NFS, und so kann ich zuhause von jedem Rechner aus darauf zugreifen. So weit, so gut.

Da ich manchmal mit einem Laptop unterwegs bin, und auch mal unterwegs etwas schreibe (Rechnungen beispielsweise, lechz!) oder ändere (Termine), manchmal aber auch zuhause Änderungen mache (in `/home/archiv`), brauche ich erstens einen Teil des Archivverzeichnisses auch unterwegs (aber nicht alles, weil das zuviel wäre), und zweitens hatte ich immer das Problem, daß ich auf dem Laptop geänderte Dateien in das Archivverzeichnis kopieren muß, und im Archivverzeichnis geänderte Dateien dafür auf den Laptop. Wer weiß aber noch nach ein paar Tagen sicher, was er wo geändert hat?

Meine derzeit praktizierte Lösung sieht so aus: Auf dem zentralen Rechner `aw33` gibt es neben dem vollständigen Verzeichnis `aw33:/home/archiv` noch ein kleineres `aw33:/home/miniarchiv`. Das ist entstanden als Kopie von `aw33:/home/archiv`; von Hand habe ich dann alles darin gelöscht, was ich unterwegs ziemlich sicher nicht brauche. Auf dem Laptop `aw35` halte ich eine Kopie davon (also `aw35:/home/miniarchiv`). Alle aktuell interessanten Dateien sind jetzt dreifach vorhanden (`aw33:/home/archiv`, `aw33:/home/miniarchiv`, `aw35:/home/miniarchiv`).

Die nicht ganz so aktuellen Dateien sind nur in `aw33:/home/archiv` vorhanden.

Die aktuellen Dateien können jetzt entweder auf `aw33` oder auf dem `aw35` in `/home/miniarchiv` geändert werden. Wenn ich den Rechner wechsele (bisher auf `aw33` gearbeitet, in nächster Zeit auf `aw35`, oder umgekehrt), dann müssen die beiden `miniarchiv`-Verzeichnisse so abgeglichen werden, daß alle Dateien, die auf `aw33` neuer sind, von `aw33` nach `aw35` kopiert werden, und umgekehrt. Danach muß dann alles, was in `aw33:/home/miniarchiv` neuer ist als in `aw33:/home/archiv` noch in diese Richtung kopiert werden, aber nicht zurück. Das ist natürlich gut geeignet, mit einem Programm automatisiert zu werden.

Da ähnliche Fälle (Dateien in zwei oder mehr verschiedenen Verzeichnissen abgleichen, mal in beide Richtungen, mal nur in eine) öfter vorkommen, lautet die Aufgabenstellung:

Gesucht ist ein Shellskript `updt`*, das man mit den Namen von zwei oder mehr Verzeichnissen aufrufen kann. Die Dateien werden so kopiert, daß danach in allen Verzeichnissen die jeweils neueste Dateiversion zu finden ist. Mit einer Option (ich habe* `-1` *gewählt) kann man festlegen, daß nur in der Richtung kopiert wird, in der auch die Verzeichnisse angegeben sind (z.B. kopiert*

```
updt -1 /home/miniarchiv /home/archiv
```

neuere Dateien von `/home/miniarchiv` *nach* `/home/archiv`*, aber nicht zurück).*

Mit einer weiteren Option `-v` kann man sich anzeigen lassen, was alles wohin kopiert wird.

Musterlösung siehe **3.7.1 Lösung: Abgleichen von Verzeichnissen: updt** auf Seite 103.

3.6.2 Aufgabe: Automatisches Backup: fullbackup_install

Gesucht ist ein Mechanismus, um erstens gelegentlich ein Vollsicherung bestimmter Verzeichnisse zu machen (volles Backup, bei mir `/home`, `/usr/local`, `/etc`, `/var/spool/mail`, `/root`). Weil das eine ziemliche Menge ist (bei mir ca. 3 GB zur Zeit), und sich von dem vielen Zeug anteilig nicht viel täglich ändert, will ich dann zweitens noch eine tägliche (oder besser: nächtliche) Sicherung aller Dateien, die sich im laufenden Betrieb seit der letzten Sicherung geändert haben, oder die seither dazukamen (inkrementelles Backup).

Das volle Backup will ich nach eigenem Ermessen starten können; das inkrementelle soll jede Nacht zu einem bestimmten Zeitpunkt automatisch starten. Dazu lege ich nach dem vollen Backup eine leere Kassette ins Laufwerk (bei mir `/dev/tape` beziehungsweise `/dev/ntape`), und wünsche mir jeden morgen eine email, anhand derer ich sehe, ob die automatische inkrementelle Sicherung geklappt hat, während ich mich meinem wohlverdienten Schlaf hingegeben hatte. Die inkrementellen Sicherungen werden hintereinander auf das Band geschrieben. Wenn das Band voll ist, sehe ich das an der email, und lege entweder ein neues Band für die nächsten inkrementellen Sicherungen ein, oder mache gleich mal wieder ein volles Backup.

So habe ich alle Dateien regelmäßig gesichert, ohne jeden Tag ein Band verschwenden zu müssen (eine SLR-5-Kassette für 4 GB kostet knapp 100 DM).

Neben dem vollen und dem inkrementellen Backup kamen dann noch folgende Wünsche auf:

- Ermitteln der Größe aller Dateien, die von einem vollen Backup belegt werden
- Zurückladen aller inkrementellen Sicherungen, die auf einem Band hintereinander liegen

Insgesamt sind das jetzt vier Funktionen (volles Backup, inkrementelles Backup, Ermitteln der Größe, Zurückladen eines gesamten Bandes mit möglicherweise mehreren Sicherungen), die sich teilweise überschneiden, und ein paar gleiche Informationen brauchen.

Als fauler Mensch hatte ich keine Lust, vier ähnliche Skripte zu bauen, die irgendwie auf gleiche Parameter zugreifen müßten (welches Device ist das Bandlaufwerk? Welche Verzeichnisse sollen gesichert werden?). Bei irgendwelchen Änderungen müßten dann ja alle Skripte geändert werden.

Deshalb habe ich folgende Krücke genommen: Es gibt in `/etc/backup` ein Skript namens `fullbackup`. Weiterhin gibt es dort noch drei symbolische Links `incrbackup`, `du`, und `incrrestore`.

Man kann also mit vier verschiedenen Namen ein und dasselbe Skript aufrufen. Dieses Skript entscheidet dann anhand seines nullten Parameters `$0`, was es zu tun hat.

Bei einem Aufruf des vollen Backups muß irgendwie hinterlegt werden, wann die letzte Sicherung begonnen hatte (damit folgende inkrementelle Sicherungen alle neueren Dateien erkennen können), ebenso bei einem inkrementellen Backup. Bei einem Abbruch, beispielsweise wegen eines vollen Bandes, darf der Zeitpunkt der letzten korrekten Sicherung nicht überschrieben worden sein! Einfach beim erfolgreichen Ende der Sicherung Datum und Uhrzeit hinterlegen ist aber auch nicht der Weisheit letzter Schluß, weil dann bei der nächsten Sicherung alle Dateien vergessen werden, die *während der letzten* erfolgreichen Sicherung geändert wurden, und vielleicht in ihrer neuen Version nicht auf dem Band sind. Also: am Ende einer erfolgreichen Sicherung muß Datum und Uhrzeit des Beginns hinterlegt werden, wenn alles geklappt hat.

`/etc/backup/fullbackup` wird bei Bedarf von Hand gestartet.
`/etc/backup/incrbackup` wird durch den `cron`-Dämon angeworfen (siehe `man cron` und `man 5 crontab`).

Musterlösung siehe **3.7.2 Lösung: Automatisches Backup: fullbackup_install** auf Seite 111.

3.6.3 Aufgabe: Kopieren von Audio CDs: cpaudiocd

Das Kopieren von Audio-CDs ist unter Linux etwas umständlich, weil man erst mit dem Kommando `cdda2wav` mit etlichen Parametern (welcher SCSI-Adapter? Welche Stücke? Alle?) die einzelnen Tracks auf die Platte kopieren muß, und dann beispielsweise mit `cdrecord` diese Tracks wieder auf einen Rohling brennt. Wenn man (wie ich) zum Auslesen des Originals dasselbe Laufwerk verwendet wie zum Brennen, dann sind die Parameter zur Angabe des Laufwerks schon mal gleich für beide Aufrufe.

Eigentlich nicht schwer, aber wie oft kopiert man eine Audio-CD? Nach ein paar Tagen oder Wochen hat man das alles natürlich vergessen, und muß schon wieder nachdenken, wie das genau war. Ich zumindest.

Das schreit gerade nach einem Shellskript!

Musterlösung siehe **3.7.3 Lösung: Kopieren von Audio CDs: cpaudiocd** auf Seite 124.

3.6.4 Aufgabe: NFS ohne Warten: automount_nfs_noauto

Eine Firma hat ein paar Linuxrechner im Netz, die alle von einem Fileserver irgendwelche Verzeichnisse beim Start automatisch mounten (über NFS).

Das geschieht in der Datei `/etc/fstab` mit einem Eintrag folgender Art:

```
server:/fueralle /vomserverfueralle nfs defaults 0 0
```

Die Zeile bedeutet, daß etwas gemountet werden soll, und zwar:

- `server:/fueralle` bedeutet, daß vom Rechner `server` im lokalen Netz dessen Verzeichnis `/fueralle` geholt werden soll
- `/vomserverfueralle` bedeutet: an dieser Stelle im Verzeichnisbaum soll das gemountete Verzeichnis auf dem lokalen Rechner erscheinen
- `nfs` bedeutet, daß es sich um ein NFS-Verzeichnis handelt
- `defaults` gibt Optionen an. `defaults` steht für `rw` (Lesen und Schreiben), `suid` (bei gemounteten Dateien wird ggf. das suid-Bit berücksichtigt), `dev` (*device files* werden als solche behandelt), `exec` (gemountete ausführbare Binärprogramme sind auch auf dem lokalen Rechner ausführbar), `nouser` (nur `root` darf dieses Verzeichnis mounten), und `async` (asynchrone Übertragung erlaubt). Siehe auch `man 8 mount` und `man 5 fstab`.
- erste `0` hier egal (für `nfs`)
- zweite `0` hier egal (für `nfs`)

Problem ist mal wieder ein Laptop. Der Chef von dem Laden will natürlich schick bei seinen Kunden aufkreuzen, mit einem Laptop unterm Arm. Macht die Kiste an, und wartet 5 Kaffeetassen lang darauf, daß er sich endlich anmelden darf. Woran liegt's? Man hat ihm den Rechner natürlich so eingerichtet, daß er beim Hochfahren des Netzwerks versucht, irgendwelche Verzeichnisse vom Fileserver zu mounten. Und bis ein NFS-client aufgibt, das dauert.

In irgendeinem Skript wird beim Hochfahren des Rechners der Befehl `mount -at nfs` ausgeführt (bei SuSE-Linux ab 6.0 ist das in der Datei `/sbin/init.d/nfs`). Dieser Befehl `mount -at nfs` läßt vom Programm `mount` alle Verzeichnisse mounten, die in der `/etc/fstab` stehen, sofern sie als `auto` markiert sind (Option `-a`), und vom Typ `nfs` sind (Option `-t nfs`).

Also geht sein Knecht natürlich her, und ändert die entsprechende Zeile in der `/etc/fstab`:

```
server:/fueralle /vomserverfueralle nfs defaults 0 0
```

um in:

```
server:/fueralle /vomserverfueralle nfs noauto,exec,rw,suid,user 0  0
```

Jetzt wird beim Starten der Befehl `mount -at nfs` nicht mehr das Hochfahren ausbremsen, weil ja die Zeile kein `auto` enthält, sondern vielmehr ausdrücklich `noauto`.

Der Knecht dachte dabei schlauerweise gleich daran, die Option `user` zu setzen. Dann kann sich der Chef nämlich mit dem kleinen Befehl:

```
mount /vomserverfueralle
```

das Verzeichnis selbst mounten (ohne `root` zu sein!), wenn er im Büro sitzt und sein Rechner am Netz hängt.

Da hat der Knecht aber die Rechnung ohne den Wirt gemacht. Der Chef hat seine Kunden im Kopf, und ein `mount /vomserverfueralle` hat da keinen Platz mehr. Leute, die mit Rechnern groß geworden sind, und nicht nur in Word herumklicken, haben damit keine Probleme. Aber jemand, der kurz vor der Pension steht, und trotzdem noch einen Rechner anfaßt, will daß das Ding einfach nur funktioniert.

Die Stimmung ist schlecht. Gefragt ist ein Weg, der die Verzeichnisse mountet, sofern möglich, aber den Rechner nicht ausbremst, wenn er gar nicht am Netz hängt.

Was tun?

Musterlösung siehe **3.7.4 Lösung: NFS ohne Warten: automount_nfs_noauto** auf Seite 128.

3.7 Lösungen

Hier sind meine Musterlösungen zu den vorigen Aufgaben zu finden. Bei Bedarf (und mit Einverständnis natürlich) werde ich auch gerne interessante Lösungen der Studenten mit aufnehmen.

3.7.1 Lösung: Abgleichen von Verzeichnissen: updt

Hier ist die Musterlösung zu **3.6.1 Aufgabe: Abgleichen von Verzeichnissen: updt** auf Seite 96:

```
#!/bin/sh
#
#########################################################################
#
# Time-stamp: "29.09.02 15:38 updt klaus@wachtler.de"
#
# Shellskript, um zwei (oder mehr) Dateien und Verzeichnisse
# abzugleichen.
#
#########################################################################
#
#           ///| |||\    || ||||    //
#          ////| |||\\   || ||||   //
#         ////|| ||||\\  || ||||  //
#        //// || |||| \\ || |||| //
#       ////==|| ||||  \\|| ||||//
#      ////   || ||||   \\| |||//
#     ////    || ||||    \| ||//
#                           |//
#         ||||    /|    //  //  ///| ||||=====//  ////=======
#         ||||   /||   //  //  ////| ||||    //  ////
#         ||||  //||  //  //  ////|| ||||   //  ////
#         |||| // || //  //  //// || ||||  //  ////
#         ||||//  ||//  //  ////==|| ||||=((  ((((=======
#         |||//   |//  //  ////   || ||||  \\  \\\\
#         |||/    |/  //  ////    || ||||   \\  \\\\=======
#
#########################################################################
```

```
#
# Copyright:
############
#
# (C) 1999-2002 AnyWare
#
# Dipl.Ing. Klaus Wachtler
# Breidingstr. 17
# 29614 Soltau
#
# Tel. (+49)-5191-70 2 71, (+49)-171-45 53 039
# Fax: (+49)-5191-70 2 72
# eMail AnyWare@Wachtler.de
#
######################################################################
#
# Aufgabenstellung:
###################
#
# Dieses Skript wird mit mindestens zwei Verzeichnisnamen aufgerufen.
# Alle angegebenen Verzeichnisse werden miteinander abgeglichen, sind
# danach also identisch.
#
# Funktionsweise:
# Es wird mit cp -u ... von jedem Verzeichnis in alle anderen
# kopiert.
# Beispiel:
#    updt /home/klaus /home/klaus.archiv
# kopiert alle neueren oder fehlenden Dateien von /home/klaus nach
# /home/klaus.archiv, und dann die neueren oder fehlenden von
# /home/klaus.archiv nach /home/klaus.
# Oder:
#    updt a b c d
# kopiert entsprechend von a nach b, c, und d; dann von b nach
# a, c, d; dann von c nach a, b, d, und zuletzt von d nach a, b,
# und c.
#
# Optionen:
###########
#
# -v oder --verbose:
#      ausführlichere Ausgabe erzeugen.
#
```

```
# -h, -? oder --help:
#      es wird eine kurze Hilfe ausgegeben.
#
# -1 (eine Eins):
#      es wird nur in eine Richtung kopiert.
#      Beispiel:
#            updatedir -1 a b c d
#      kopiert von a nach b, c, und d; dann von b nach c und d, dann
#      von c nach d.
#
######################################################################
#
# Geschichte:
#############
#
# Datum:     Name: Vers. Aenderung:
# 14. 5.1999 kw    0.0   erster Entwurf
# 17.01.2002 kw          - fehlende Verzeichnisse werden ggf. neu
#                          angelegt
#                        - existierende Dateien werden gelöscht,
#                          bevor ein Verzeichnis mit dem selben
#                          Namen angelegt wird
#                        - existierende Verzeichnisse werden
#                          gelöscht, bevor Dateien desselben Namens
#                          kopiert werden
#
######################################################################
#
# Benoetigt:
############
#
#
######################################################################
#
# Umgebung:
###########
#
#
######################################################################

unset VERBOSE UNIDIRECT VERZEICHNISSE VERZEICHNISSER

#VERZEICHNISSE=""
```

```
#VERZEICHNISSER=""
#UNIDIRECT=""
#VERBOSE=""

usage()
{
            cat <<EOF
 $0 wird mit mindestens zwei Verzeichnisnamen aufgerufen.
 Alle angegebenen Dateien und Verzeichnisse werden miteinander
 abgeglichen, sind danach also identisch.

 Funktionsweise:
 Es wird mit cp -u ... von jedem Verzeichnis in alle anderen
 kopiert.
 Beispiel:
    updt /home/klaus /home/klaus.archiv
 kopiert alle neueren oder fehlenden Dateien von /home/klaus nach
 /home/klaus.archiv, und dann die neueren oder fehlenden von
 /home/klaus.archiv nach /home/klaus.
 Oder:
    updt a b c d
 kopiert entsprechend von a nach b, c, und d; dann von b nach
 a, c, d; dann von c nach a, b, d, und zuletzt von d nach a, b,
 und c.

 Optionen:
##########

 -v oder --verbose:
      ausführlichere Ausgabe erzeugen.

 -h, -? oder --help:
      es wird eine kurze Hilfe ausgegeben.

 -1 (eine Eins):
      es wird nur in eine Richtung kopiert.
      Beispiel:
            updatedir -1 a b c d
      kopiert von a nach b, c, und d; dann von b nach c und d, dann
      von c nach d.
EOF
}
```

```
# Schleife über alle Parameter, dabei jeweils eine Liste für den
# Vorwärtsdurchlauf und eine für den Rückwärtsdurchlauf
# zusammenbauen:
for arg
do
    case $arg in
        -h|-"?"|--help)
            usage
            true
            exit
            ;;
        -v|-verbose)
            VERBOSE="y"
            ;;
        -1)
            UNIDIRECT="y"
            ;;
        *)
            VERZEICHNISSE=$VERZEICHNISSE" "$arg
            VERZEICHNISSER=$arg" "$VERZEICHNISSER
            # if [ -r $arg ]
            # then
            #     VERZEICHNISSE=$VERZEICHNISSE" "$arg
            #     VERZEICHNISSER=$arg" "$VERZEICHNISSER
            # else
            #     # echo $arg" ist kein Verzeichnis!"
            #     echo $arg" ist nicht lesbar!"
            #     false
            #     exit
            # fi
            ;;
    esac
done

if [ $VERBOSE ]
then
    #echo "VERBOSE   ="$VERBOSE
    #echo "UNIDIRECT ="$UNIDIRECT
    #echo "abzugleichen: " $VERZEICHNISSE
    VERBOSE="-v"
fi

LETZTES=""
```

```
VERZEICHNISGEDREHT=""

# Vorwärtsdurchlauf durch alle Verzeichnisse/Dateien:
for d in $VERZEICHNISSE
do
    VERZEICHNISGEDREHT=$d" "$VERZEICHNISGEDREHT
    if [ $LETZTES ]
    then
        # if [ "$VERBOSE" ]
        # then
        #    echo $LETZTES" -> "$d
        # fi

        if [ ! -e $LETZTES ]
        then
            echo $LETZTES existiert nicht!
            false
            exit
        fi

        # Wenn die Quelle ein Verzeichnis ist, dann muß
        # 1) geprüft werden, ob das Ziel auch ein Verzeichnis ist.
        #    Wenn nicht, dann ggf. eine Datei mit dem Namen
        #    löschen, und ein leeres Verzeichnis anlegen.
        # 2) es muß kopiert werden in der Art:
        #      cp -... source/* dest
        #
        # Ist die Quelle kein Verzeichnis, dann muß
        # 1) das Ziel gelöscht werden, falls es ein Verzeichis ist
        # 2) es muß kopiert werden in der Art:
        #      cp -... source dest
        if [ -d $LETZTES ]
        then
            #echo Quelle $LETZTES ist ein Verzeichnis!
            if [ ! -d $d ]
            then
                #echo Ziel $d ist kein Verzeichnis!
                # Ggf. löschen:
                rm -rf $d
                # neu anlegen:
                mkdir -p $d
                # Zugriffsrechte kopieren:
                chmod --reference=$LETZTES $d
```

```
                chown --reference=$LETZTES $d
            fi
            #echo cp $VERBOSE -u -d -f -r -p $LETZTES/* $d
            cp $VERBOSE -u -d -f -r -p $LETZTES/* $d
            touch --reference=$LETZTES $d
        else
            #echo Quelle $LETZTES ist kein Verzeichnis!
            if [ -d $d ]
            then
                #echo Ziel $d ist ein Verzeichnis!
                # Ggf. löschen:
                rm -rf $d
            fi
            #echo cp $VERBOSE -u -d -f -p $LETZTES $d
            cp $VERBOSE -u -d -f -p $LETZTES $d

        fi

    fi
    LETZTES=$d
done

if [ ! "$UNIDIRECT" = "y" ]
then
    LETZTES=""
    for d in $VERZEICHNISGEDREHT
    do
        if [ $LETZTES ]
        then
            # if [ "$VERBOSE" ]
            # then
            #     echo $LETZTES" -> "$d
            # fi


            # Wenn die Quelle ein Verzeichnis ist, dann muß
            # 1) geprüft werden, ob das Ziel auch ein Verzeichnis
            #    ist.
            #    Wenn nicht, dann ggf. eine Datei mit dem Namen
            #    löschen, und ein leeres Verzeichnis anlegen.
            # 2) es muß kopiert werden in der Art:
            #      cp -... source/* dest
            #
```

```
            # Ist die Quelle kein Verzeichnis, dann muß
            # 1) das Ziel gelöscht werden, falls es ein Verzeichis
            #    ist
            # 2) es muß kopiert werden in der Art:
            #      cp -... source dest
            if [ -d $LETZTES ]
            then
                #echo Quelle $LETZTES ist ein Verzeichnis!
                if [ ! -d $d ]
                then
                    #echo  Ziel $d ist kein Verzeichnis!
                    # Ggf. löschen:
                    rm -rf $d
                    # neu anlegen:
                    mkdir -p $d
                    # Zugriffsrechte kopieren:
                    chmod --reference=$LETZTES $d
                    chown --reference=$LETZTES $d
                fi
                #echo cp $VERBOSE -u -d -f -r -p $LETZTES/* $d
                cp $VERBOSE -u -d -f -r -p $LETZTES/* $d
                touch --reference=$LETZTES $d
            else
                #echo  Quelle $LETZTES ist kein Verzeichnis!
                if [ -d $d ]
                then
                    #echo Ziel $d ist ein Verzeichnis!
                    # Ggf. löschen:
                    rm -rf $d
                fi
                #echo cp $VERBOSE -u -d -f -p $LETZTES $d
                cp $VERBOSE -u -d -f -p $LETZTES $d


            fi

        fi
        LETZTES=$d
    done
fi
```

3.7.2 Lösung: Automatisches Backup: fullbackup_install

Hier ist die Musterlösung zu **3.6.2 Aufgabe: Automatisches Backup: fullbackup_install** auf Seite 98.

Das Skript hat vor dem Installieren den Namen `fullbackup_install`, uns installiert sich selbst, wenn es mit diesem Namen aufgerufen wird.

Der erhoffte Lernerfolg dieser Musterlösung:

- ohne ausführlichen Kommentar ist alles nichts wert
- den Programmnamen (`$0`) bekommt man nicht zum Vergnügen immer übergeben, sondern man darf ihn auch ruhig mal verwenden
- wie man normale Bandlaufwerke (hier `/dev/tape`) und ihre *no rewind*-Abart (`/dev/ntape`) mit tar nutzen kann
- wie man Programme automatisch starten lassen kann
- wie man aus einem Shellskript heraus eben mal eine email verschickt

```
#!/bin/sh
#
######################################################################
#
# Time-stamp: "29.09.02 15:52 fullbackup_install klaus@wachtler.de"
#
######################################################################
#
# /etc/backup/fullbackup
# /etc/backup/append       -> /etc/backup/fullbackup
# /etc/backup/delete       -> /etc/backup/fullbackup
# /etc/backup/du           -> /etc/backup/fullbackup
# /etc/backup/incrbackup   -> /etc/backup/fullbackup
# /etc/backup/incrrestore  -> /etc/backup/fullbackup
# /etc/backup/list         -> /etc/backup/fullbackup
# /etc/backup/rewind       -> /etc/backup/fullbackup
#
# fullbackup macht ein volles Backup aller Verzeichnisse (ab /),
# die in $ZUSICHERN aufgeführt sind.
# Datum und Uhrzeit (vor der Sicherung)
# werden in /etc/backup/lastfull und
```

```
# /etc/backup/lastincr festgehalten.
# Nachfolgende inkrementelle Sicherungen können sich also auf
# /etc/backup/lastincr beziehen, und diese dann
# aktualisieren.
#
# /etc/backup/lastincr und /etc/backup/lastfull
# werden vor der Sicherung als
# /etc/backup/lastincr.tmp und /etc/backup/lastfull.tmp
# angelegt, und erst bei Erfolg umbenannt.
#
#
# Wenn der Name, mit dem das Skript aufgerufen wird, auf
# "incrbackup" endet, dann wird nur ein inkrementelles
# Backup durchgeführt.
# Wenn das Skript also als "/etc/backup/fullbackup"
# gespeichert ist, und man mit
# "cd /etc/backup/; ln -s fullbackup incrbackup"
# einen Link darauf anlegt, dann wird -wenn
# /etc/backup/incrbackup aufgerufen wird- nur
# ein inkrementelles Backup durchgeführt.
#
# In diesem Fall verhält sich das Skript etwas anders:
#
# - Die Meldungen lauten etwas anders
#
# - Es wird nicht die Merkerdatei "$MERKERFULL"
#   aktualisiert, sondern nur "$MERKERINCR".
#
# - Das Band wird vor der Sicherung ans Ende gespult
#
# - für die Sicherung wird nicht $TAPE, sondern $NTAPE verwendet.
#   Dadurch kann für die inkrementellen Sicherungen ein Band
#   durchgehend im Laufwerk bleiben (wenn $NTAPE nicht automatisch
#   an den Anfang gespult wird). Alle inkrementellen Sicherungen
#   kommen dann hintereinander auf ein Band, bis ein Schreibfehler
#   auftritt. Erst dann muß von Hand das Band gewechselt werden.
#
# - es werden nicht alle Dateien gesichert, sondern
#   nur diejenigen, welche seit dem Anlegen von
#   "$MERKERINCR" modifiziert wurden
#
#
# Endet der Skriptname auf "restore", dann wird das Band an den
# Anfang gespult, und solange von /dev/ntape tar-Archive gelesen,
# bis ein Fehler auftritt.
```

```
# Dazu kann man wieder einen symbolischen Link verwenden:
# "cd /etc/backup/; ln -s fullbackup incrrestore"
#
#
# Endet der Skriptname auf "rewind", dann wird das Band an den
# Anfang gespult.
#
#
# Endet der Skriptname auf "list", dann wird der Bandinhalt
# gelistet, und das Band an das Ende der Daten gespult.
#
#
# Endet der Skriptname auf "append", dann wird das Band an das
# Ende der Daten gespult.
#
#
# Endet der Skriptname auf "delete", dann wird das Band gelöscht.
#
#
# Endet der Skriptname auf "du", dann wird festgestellt, wieviel
# Speicherplatz auf dem Band für eine Vollsicherung benötigt wird.
#
#
# Vorgesehene Verwendung:
#
# 0) mit ./fullbackup_install wird das Skript installiert und
#    die Links werden angelegt
#
# 1) gelegentlich wird ein ausreichend großes Band eingelegt,
#    und manuell eine Vollsicherung gestartet (mit
#    "/etc/backup/fullbackup")
#
# 2) Ein weiteres Band für inkrementelle Sicherungen verbleibt
#    im Laufwerk.
#    Über den cron-Dämon wird beispielsweise jede
#    Nacht eine inkrementelle Sicherung angeworfen (mit
#    "/etc/backup/incrbackup").
#    Beispieleintrag für /etc/crontab:
#      # automatisches inkrementelles Backup:
#      0 4 * * *     root  /etc/backup/incrbackup
#    Damit wird jeden morgen um 4:00 ein inkrementelles Backup
#    gestartet.
#    Wenn das Band (für die inkrementellen Sicherungen) voll ist,
#    muß man entweder:
#    a) ein neues Band einlegen, und eine Vollsicherung starten
```

```
#       (mit /etc/backup/fullbackup),
#       danach ein leeres Band für die inkrementellen Sicherungen
#       (ggf. mit /etc/backup/delete löschen) im
#       Laufwerk liegen lassen.
#       Damit beginnt der Zyklus von vorne.
#    oder:
#    b) Die letzte Vollsicherung behalten, die Kassette mit den
#       bisherigen ersten inkrementelle Sicherungen als solche
#       kennzeichnen,
#       und eine weitere Kassette für die kommenden inkrementellen
#       Sicherungen ggf. löschen und im Laufwerk lassen.
#
# Zum Restaurieren aller Daten geht man in ein Verzeichnis mit
# reichlich Platz und macht folgendes:
# 1) Kassette mit der Vollsicherung einlegen
# 2) /etc/backup/incrrestore
# 3) Nacheinander für alle Kassetten mit den inkrementellen
#    Sicherungen (in der richtigen Reihenfolge!)
#    a) Kassette einlegen
#    b) /etc/backup/incrrestore
# 4) Wenn man die letzte Kassette jetzt wieder für die nächsten
#    inkrementellen Sicherungen verwenden will:
#    a) diese Kassette einlegen
#    b) an das Ende der Daten spulen (inzwischen überflüssig):
#       /etc/backup/append
#
# Alternativ kann man (wenn man nur eine Datei bzw. wenige wieder
# restaurieren möchte) anhand der Inhaltsverzeichnisse aller Bänder
# (abgelegt in /home/dir/tape*.dir.gz) nach den neuesten Sicherungen
# der Datei(en) suchen, und von Hand mit tar entpacken.
#
#####################################################################
#
# Änderungen:
#
# 31.04.2000 kw   - Selbstinstallierend
#                 - Vor einer inkrementellen Sicherung wird das
#                   Band automatisch ans Ende gespult
#                   (falls der Rechner neu geboootet hat, oder
#                   jemand das Band aus dem Laufwerk genommen hatte,
#                   dann mußte man bisher daran denken, das Band
#                   ans Ende zu spulen, um nicht schon bestehende
#                   Sicherungen am Anfang des Bandes zu
#                   überschreiben.
#                   Das geschieht jetzt automatisch vor jeder
```

```
#                   inkrementellen Sicherung.).
#                   Folge: Will man auf einem bereits beschriebenen
#                   Band mit inkrementellen Sicherungen neu beginnen,
#                   dann muß das Band vorher gelöscht werden, sonst
#                   würde die erste Sicherung ja am Ende angefügt
#                   werden (falls da überhaupt Platz ist).
#                 - Zum Löschen des Bandes gibt es dafür jetzt
#                   das Kommando /etc/backup/delete.
#                 - Weitere neue Kommandos:
#                   /etc/backup/append: spult das Band ans Ende
#                   /etc/backup/rewind: spult das Band an den Anfang
#                   /etc/backup/list:
#
# 02.05.2000 kw   Nach der vollen Sicherung bleibt das Band am Ende
#                 stehen.
#                 Es wird nur noch $NTAPE verwendet, $TAPE ist
#                 überflüssig geworden.
#
#
# 18.05.2001 kw   Bei der Installation wird das Verzeichnis für
#                 die Inhaltsverzeichnisse angelegt
#
#####################################################################
#
# Einstellbare Variablen:


# Liste der zu sichernden Verzeichnisse (ab /, anzugeben ohne
# führenden Schrägstrich, mehrere Verzeichnisse getrennt mit
# Leerzeichen).
# Beispiel:
#  ZUSICHERN="home root var/spool/mail etc usr/local"
#
# (ZUSICHERN wird nur gesetzt, wenn es nicht schon beim Aufruf des
# Skripts gesetzt ist)
ZUSICHERN=${ZUSICHERN-"home root var/spool/mail etc usr/local"}

# zu verwendendes Laufwerk:
# Band wird NICHT an den Anfang gespult!
NTAPE=/dev/ntape

# Pfad für die Inhaltsverzeichnisse aller Bänder:
VERZEICHNIS_FUER_INHALTSVERZEICHNIS=/home/dir

# Benutzer und Gruppe, auf die das Inhaltsverzeichnis gesetzt werden
# soll (mit anderen Worten: wer darf die Inhaltsverzeichnisse
```

```
# sehen?):
DIRUSR=root
DIRGRP=users

# Merkerdateien für Datum der letzten vollen/inkr. Sicherung:
MERKERFULL=/etc/backup/lastfull
MERKERINCR=/etc/backup/lastincr

# Protokolldatei:
LOGFILE=/etc/backup/backup.log

# Verzeichnis für dieses Skript und die Links:
INSTALLDIR=/etc/backup

#
######################################################################


STARTSTRING=`date +"%Y%m%d_%H%M"`

if [ `echo $0|grep '.*install$'` ]
then

    echo "Installation von $INSTALLDIR/fullbackup"
    if [ `whoami` != "root" ]
    then
        echo "Fehler: Installation nur als root möglich!"
        exit 1
    fi

    # evtl. alten Müll wegwerfen:
    rm -f $INSTALLDIR/append
    rm -f $INSTALLDIR/delete
    rm -f $INSTALLDIR/du
    rm -f $INSTALLDIR/incrbackup
    rm -f $INSTALLDIR/incrrestore
    rm -f $INSTALLDIR/list
    rm -f $INSTALLDIR/rewind

    # Verzeichnis anlegen:
    mkdir -m 700 -p $INSTALLDIR 2>/dev/null
    echo Verzeichnis $INSTALLDIR angelegt

    # Verzeichnis für Inhaltsverzeichnisse anlegen:
    mkdir -m 700 -p $VERZEICHNIS_FUER_INHALTSVERZEICHNIS 2>/dev/null
```

```
    echo Verzeichnis $VERZEICHNIS_FUER_INHALTSVERZEICHNIS angelegt


    # Das Skript selbst dahin kopieren:
    cp $0 $INSTALLDIR/fullbackup
    chown root:root $INSTALLDIR/fullbackup
    chmod 500 $INSTALLDIR/fullbackup

    # Und weitere Links darauf anlegen:
    cd $INSTALLDIR
    ln -s fullbackup append
    echo "link angelegt: $INSTALLDIR/append"
    ln -s fullbackup delete
    echo "link angelegt: $INSTALLDIR/delete"
    ln -s fullbackup du
    echo "link angelegt: $INSTALLDIR/du"
    ln -s fullbackup incrbackup
    echo "link angelegt: $INSTALLDIR/incrbackup"
    ln -s fullbackup incrrestore
    echo "link angelegt: $INSTALLDIR/incrrestore"
    ln -s fullbackup list
    echo "link angelegt: $INSTALLDIR/list"
    ln -s fullbackup rewind
    echo "link angelegt: $INSTALLDIR/rewind"

    true
    exit

fi

if [ `echo $0|grep '.*du$'` ]
then

    echo "Ermitteln des Platzbedarfs..."

    ( cd /; du -c  $ZUSICHERN )

    true
    exit

fi

if [ `echo $0|grep '.*restore$'` ]
then
```

```
    echo "Zurückspulen des Bandes..."
    $INSTALLDIR/rewind

    echo "Entpacken aller Archive von $NTAPE ins aktuelle"
    echo "Verzeichnis,"
    echo "bis ein Fehler auftritt..."

    true
    erg=$?
    while [ $erg == "0"  ]
    do
            tar xvf $NTAPE 2>/dev/null
            erg=$?
            if [ $erg == "0" ]
            then
                    echo
                    echo "  #####  Sicherungssatz gefunden  #####"
                    echo
            else
                    echo
                    echo "Ende der Sicherungen?"
                    echo "Das Band steht am Ende der Daten."
            fi
    done

    echo "Ende des Zurückladens vom Band."
    exit

fi


if [ `echo $0|grep '.*delete$'` ]
then

    echo "Löschen des Bandes..."
    mt -f $NTAPE erase
    echo "Fertig!"

    exit

fi



if [ `echo $0|grep '.*rewind$'` ]
then
```

```
    echo "Zurückspulen des Bandes..."
    mt -f $NTAPE rewind

    echo "Ende des Zurückspulens."

    exit

fi


if [ `echo $0|grep '.*list$'` ]
then

    echo "Zurückspulen des Bandes..."
    $INSTALLDIR/rewind

    echo "Listen aller Archive von $NTAPE,"
    echo " bis ein Fehler auftritt..."

    true
    erg=$?
    while [ $erg == "0"  ]
    do
               tar tvf $NTAPE 2>/dev/null
               erg=$?
               if [ $erg == "0" ]
               then
                       echo
                       echo "  #####  Sicherungssatz gefunden  #####"
                       echo
               else
                       echo
                       echo "Ende der Sicherungen?"
                       echo "Das Band steht am Ende der Daten."
               fi
    done

    echo "Ende des Inhaltsverzeichnisses vom Band."

    exit

fi
```

```
if [ `echo $0|grep '.*append$'` ]
then

    echo "Spulen des Bandes ans Ende..."
    mt -f $NTAPE eom
    echo "fertig!"

    exit

fi


if [ `echo $0|grep '.*incrbackup$'` ]
then

        # Programmname endet auf "incrbackup", also nur
        # inkrementelles Backup:
        incrbackup="y"
        unset fullbackup
        TEXT="INKREMENTELLES BACKUP"
        text="Inkrementelles Backup"
        NAME=tape.$STARTSTRING.incr

        mt -f $NTAPE eom

else

        # Programmname endet NICHT auf "incrbackup", also
        # volles Backup:
        unset incrbackup
        fullbackup="y"
        TEXT="VOLLBACKUP"
        text="Vollbackup"
        NAME=tape.$STARTSTRING

        mt -f $NTAPE rewind

fi

DIRFILE=$NAME.dir


cd /
```

```
# die Merkerdateien erstmal provisorisch anlegen
# (mit Datum/Uhrzeit vom Backupstart):
echo $STARTSTRING >$MERKERINCR.tmp
if [ $fullbackup ]
then

        echo $STARTSTRING >$MERKERFULL.tmp

fi


echo ""  >>$LOGFILE
echo "-----------------------------------------"  >>$LOGFILE
echo ""  >>$LOGFILE

echo $text" " `date +"%A %d.%m.%Y %H:%M"`   >>$LOGFILE
echo "(Rechner: " `uname -a` ")"  >>$LOGFILE

echo ""  >>$LOGFILE

if [ $fullbackup ]
then

        # Volles Backup
        tar cvf $NTAPE                                          \
                $ZUSICHERN                                      \
                -V "$NAME"                                      \
                >$DIRFILE  2>>$LOGFILE

else

        # Inkrementelles Backup
        find $ZUSICHERN -newer $MERKERINCR -and -type f -print0 |\
        tar cvf $NTAPE -V $NAME --null -T /dev/stdin            \
        >$DIRFILE 2>>$LOGFILE

fi


if [ $? != "0" ]
then

        # tar meldet einen Fehler

        echo "ABBRUCH! ("`date +"%d.%m.%Y %H:%M"`")"  >>$LOGFILE
```

```
        echo ""  >>$LOGFILE

        rm -f $MERKERINCR.tmp
        if [ $fullbackup ]
        then
                rm -f $MERKERFULL.tmp
        fi

        mail -s "$TEXT $STARTSTRING gescheitert!" root <<-EOF
        $0: $text ($STARTSTRING) gescheitert.
        Genaueres steht in $LOGFILE.
        Ende: `date +"%d.%m.%Y %H:%M"`
        EOF

else


        # tar scheint geklappt zu haben.
        # Dann kann man jetzt die Merker für das letzte Backup
        # (etc/backup/lastincr, $MERKERFULL) endgültig
        # anlegen und das Inhaltsverzeichnis komprimieren:

        gzip -9 $DIRFILE
        mv $DIRFILE.gz $VERZEICHNIS_FUER_INHALTSVERZEICHNIS
        chmod 640 $VERZEICHNIS_FUER_INHALTSVERZEICHNIS/$DIRFILE.gz
        chown $DIRUSR:$DIRGRP                                      \
              $VERZEICHNIS_FUER_INHALTSVERZEICHNIS/$DIRFILE.gz
        echo "Inhaltsverzeichnis steht in " \
        $VERZEICHNIS_FUER_INHALTSVERZEICHNIS/$DIRFILE.gz  >>$LOGFILE

        mv $MERKERINCR.tmp $MERKERINCR
        if [ $fullbackup ]
        then
                mv $MERKERFULL.tmp $MERKERFULL
        fi

        echo "ok, fertig ("`date +"%d.%m.%Y %H:%M"`")"  >>$LOGFILE
        echo ""  >>$LOGFILE

        mail -s "$text $STARTSTRING erfolgreich!" root <<-EOF
        $0: $text ($STARTSTRING) erfolgreich.
        Inhaltsverzeichnis steht in\
 $VERZEICHNIS_FUER_INHALTSVERZEICHNIS/$DIRFILE.gz
        Ende: `date +"%d.%m.%Y %H:%M"`
        EOF
```

```
fi

# Ende.
```

3.7.3 Lösung: Kopieren von Audio CDs: cpaudiocd

Hier ist die Musterlösung zu **3.6.3 Aufgabe: Kopieren von Audio CDs: cpaudiocd** auf Seite 100.

Was man daran sehen kann?

- Wie man aus der Infoausgabe eines Programms (hier `cdda2wav`) mit `grep` und `sed` die gewünschte Information zieht
- Lesen von Benutzereingabe mit `read`
- formatiertes Schreiben mit `printf`
- daß man aus einem Skript heraus einfach ein neues generieren kann (ist ja alles nur Text!)

```
#!/bin/sh
# kopiert eine Audio-CD komplett auf Platte, und schreibt ein
# Skript zum Brennen
#
# 27.04.2000 kw
#
# Das Unterdrücken der 2-sec-Pause soll laut "man cdrecord" so nur
# mit TEAC-Brennern funktionieren (ich habe einen TEAC CDR55S, da
# klappt das auch).
#

# SCSI-Bus,SCSI-Id,LUN
DEVICE0="0,0,0"


if [ $# -ge 1 ]
then
  projekt=$1
else
  echo -n "Name der CD und des Verzeichnisses: "
  read projekt
fi

echo "Das Verzeichnis für die Tracks ist $projekt"

echo -n "Der CD-Brenner hat SCSI-Bus,SCSI-Id,LUN: ($DEVICE0) "
```

```
read DEVICE

if [ ! "$DEVICE" ]
then
  DEVICE=$DEVICE0
fi

echo -n "Welche Tracks, z.B.: 1 5 8 10? (alle) "
read trackliste

echo -n "Zwischen den Stücken eine 2-sec-Pause j/n? (j) "
read jn
if [ "$jn" = "n" -o "$jn" = "N" ]
then
  pregap=" defpregap=0"
else
  pregap=""
fi

mkdir $projekt 2>/dev/null
cd $projekt

# Inhaltsverzeichnis lesen:

# diese Version hatte ich bis SuSE 6.2/cdda2wav 0.95beta07 verwendet:
# if cdda2wav -D $DEVICE -N -t1 -d1 -v 3 /dev/null 2>tmp.txt
# then
#   echo "Audio-CD erkannt."
# else
#   echo "keine Audio-CD?"
#   false
#   exit
# fi

# Das klappt jetzt leider nicht mehr, weil cdda2wav jetzt (SuSE 6.3)
# nicht mehr 0 zurückgibt, wenn alles ok läuft.
# Deshalb wird der Rückgabewert jetzt ignoriert:
cdda2wav -D $DEVICE -N -t1 -d1 -v 3 /dev/null 2>tmp.txt


# jetzt steht in tmp.txt etwa folgendes:
#   Read TOC CD Text failed (probably not supported).
#   #Cdda2wav version 0.95beta07 plus real time scheduling ...
#   AUDIOtrack pre-emphasis  copy-permitted tracktype channels
#         1-13           no               no     audio    2
```

```
#   Table of Contents: total tracks:13, (total time 39:37.00)
#     1.( 3:21.00),  2.( 2:31.00),  3.( 3:44.00),  4.( 1:23.00),...
#     6.( 5:35.00),  7.( 2:51.00),  8.( 5:39.00),  9.( 2:06.00),...
#    11.( 4:25.00), 12.( 1:20.00), 13.( 1:05.00),
#   CDDB discid: 0x9d09470d
#   samplefile size will be 176444 bytes.
#   recording 1.00000 seconds stereo with 16 bits @ 44100 Hz ->...
#   votes for little:  160,  votes for big:    0
#   little endian detected
#   track  1 successfully recorded
#
# Daraus extrahieren wir mit grep die Zeile:
#        1-13          no              no     audio    2,
# und daraus mit sed dann die Anzahl der Tracks (hier: 13).

#echo "cdda2wav sagt:"
#cat tmp.txt

TRACKS=`cat tmp.txt | grep "^ *1-[0-9]* "|sed 's/^ *1-\([0-9]*\).*$/\1/'`
rm tmp.txt


if [ "$trackliste" ]
then

  # Tracks angegeben, also nur die kopieren:
  for i in $trackliste
  do
    # Damit die Dateinamen von ls und der shell richtig sortiert
    # werden, schreiben wir in i2 die Tracknummer zweistellig, und
    # nehmen dann $i2, um einen Dateinamen daraus zu bilden.
    # ( || true, weil cdda2wav wieder irgendeinen Unfug zurückgibt)
    i2=`printf %02d $i`
    if cdda2wav -q -D $DEVICE -s -b 16 -r 44100                 \
                -t $i -H $projekt"_"$i2.wav                     \
       || true
    then
      echo "Track $i gelesen"
    else
      echo "Lesen von Track $i gescheitert."
      false
      exit
    fi
  done
```

```
else

  # alle Tracks sollen kopiert werden
  # ( || true, weil cdda2wav wieder irgendeinen Unfug zurückgibt)
  if cdda2wav -q -D $DEVICE -s -b 16 -r 44100                    \
              -t 1+$TRACKS -B -H $projekt                         \
     || true
  then
    echo "Tracks 1-$TRACKS gelesen"
  else
    echo "Lesen der CD gescheitert."
    false
    exit
  fi

fi

echo "#!/bin/sh" >brenn
echo "cdrecord -eject -speed=16 dev=$DEVICE -audio -pad$pregap "\
     *.wav >>brenn
chmod 755 brenn

echo "Brennen der CD mit:"
echo "  cd $projekt; ./brenn; cd .."

true
exit
```

3.7.4 Lösung: NFS ohne Warten: automount_nfs_noauto

Hier ist die Musterlösung zu **3.6.4 Aufgabe: NFS ohne Warten: automount_nfs_noauto** auf Seite 101.

Diese Lösung funktioniert exakt in dieser Form nur unter SuSE Linux ab 6.0. Im Prinzip sehr ähnliche Lösungen sind natürlich auf jedem Unix machbar, unterscheiden sich aber in Details (verschiedene *run level*, andere Verzeichnisse für die Startskripte).

Die Musterlösung erinnert etwas an ein Telefonbuch: viel Text, wenig Handlung. Der Kern des Ganzen ist ein unscheinbares `nohup mount $f >/dev/null 2>&1 &`. Damit wird ein `mount` im Hintergrund ausgeführt. Das vorangestellte `nohup` (*no hang up*) bewirkt, daß der Hintergrundprozeß ein `SIGHUP` überlebt. Dieses Signal wird an einen Prozeß geschickt, wenn er keinen Vaterprozeß mehr hat (wenn er also länger läuft als der Prozeß, der ihn gestartet hat).

Mit einem solchen `nohup mount $f >/dev/null 2>&1 &` wird ein `mount` im Hintergrund ausgeführt, ohne daß der Aufrufer auf den Erfolg (oder Mißerfolg) des Aufrufs wartet.

Damit wird gemountet, wenn möglich, aber der Aufrufer (während des Bootens) muß nicht auf das Ende des Vorgangs warten. Während `mount` gemütlich darauf wartet, ob sich jemand am anderen Ende des Kabels meldet, kann der Rechner ja weiter booten.

Interessant ist das Skript aus drei Gründen:

- Das `$f` in `nohup mount $f >/dev/null 2>&1 &` wird vorher aus der Datei `/etc/fstab` gewonnen, und zwar aus allen Zeilen, in denen in den entsprechenden Spalten ein `nfs` beziehungsweise ein `noauto` vorkommt, denn nur dann soll in diesem Skript gemountet werden. Dazu werden sogenannte *reguläre Ausdrücke* (RE) verwendet. Diese sind im Kommentar des Skripts grob beschrieben. Mehr dazu findet man mit `man 7 regex`.
- Mit diesem Skript kann man die Einbindung neuer Funktionen in den Startvorgang studieren. Leider unterscheiden sich die Unixversionen untereinander, aber die grundsätzliche Vorgehensweise ist seit Unix System V/R4 doch recht ähnlich.

  Jeder Unixrechner läuft zu einem bestimmten Zeitpunkt in einem definierten *run level*. Ein bestimmter run level ist der *default run level*, in den das System nach dem Einschalten hochfährt.

Der run level kann einen Wert von 0 bis 5 annehmen. Zwischen 0 und 1 denkt man sich noch den run level `S`.

Mit einem Aufruf von `init` kann der Administrator einen Wechsel in einen anderen run level erzwingen. Wenn der Rechner gerade in run level 2 ist, kann man mit `init 3` in den run level 3 wechseln.

Mit `init 6` wird der Rechner heruntergefahren (Reset), und startet automatisch wieder in den default run level.

Primär werden die Bedeutungen der run level mit der Datei `/etc/inittab` gesteuert.

Ein Beispiel von meinem Rechner (`aw33.wachtler.de`, SuSE 6.3):

```
#
# /etc/inittab
#
# Copyright (c) 1996 SuSE GmbH Nuernberg, Germany.  All rights reserved.
#
# Author: Florian La Roche <florian@suse.de>, 1996
#
# This is the main configuration file of /sbin/init, which
# is executed by the kernel on startup. It describes what
# scripts are used for the different run-levels.
#
# All scripts for runlevel changes are in /sbin/init.d/ and the main
# file for changes is /etc/rc.config.
#

# default runlevel
id:2:initdefault:

# check system on startup
# first script to be executed if not booting in emergency (-b) mode
si:I:bootwait:/sbin/init.d/boot

# /sbin/init.d/rc takes care of runlevel handling
#
# runlevel 0 is halt
# runlevel S is single-user
# runlevel 1 is multi-user without network
# runlevel 2 is multi-user with network
# runlevel 3 is multi-user with network and xdm
# runlevel 6 is reboot
l0:0:wait:/sbin/init.d/rc 0
l1:1:wait:/sbin/init.d/rc 1
l2:2:wait:/sbin/init.d/rc 2
```

```
l3:3:wait:/sbin/init.d/rc 3
#l4:4:wait:/sbin/init.d/rc 4
#l5:5:wait:/sbin/init.d/rc 5
l6:6:wait:/sbin/init.d/rc 6

# what to do in single-user mode
ls:S:wait:/sbin/init.d/rc S
~~:S:respawn:/sbin/sulogin

# what to do when CTRL-ALT-DEL is pressed
ca::ctrlaltdel:/sbin/shutdown -r -t 4 now

# special keyboard request (Alt-UpArrow)
# look into the kbd-0.90 docs for this
kb::kbrequest:/bin/echo "Keyboard Request -- edit /etc/inittab to let this work."

# what to do when power fails/returns
pf::powerwait:/sbin/init.d/powerfail    start
pn::powerfailnow:/sbin/init.d/powerfail now
#pn::powerfail:/sbin/init.d/powerfail now
po::powerokwait:/sbin/init.d/powerfail  stop

# for ARGO UPS
sh:12345:powerfail:/sbin/shutdown -h now THE POWER IS FAILING

# getty-programs for the normal runlevels
# <id>:<runlevels>:<action>:<process>
# The "id" field  MUST be the same as the last
# characters of the device (after "tty").
1:123:respawn:/sbin/mingetty --noclear tty1
2:123:respawn:/sbin/mingetty tty2
3:123:respawn:/sbin/mingetty tty3
4:123:respawn:/sbin/mingetty tty4
5:123:respawn:/sbin/mingetty tty5
6:123:respawn:/sbin/mingetty tty6

# ...
# fax getty (hylafax)
mo:23:respawn:/usr/lib/fax/faxgetty /dev/modem
# end of /etc/inittab
```

Beim Starten des Rechners wird der run level angesteuert, der in der Datei `/etc/inittab` mit `initdefault` deklariert ist:

```
id:2:initdefault:
```

bedeutet, daß der run level 2 erreicht werden soll, solange der `root` nicht mit einem `init`-Befehl dazwischenfunkt.

Der Hersteller kann definieren, welcher run level von 1 bis 5 welche Bedeutung hat. Allgemein gilt aber:

– run level 0 bedeutet: das System ist heruntergefahremn, es steht (bereit zum Ausschalten, oder Einstellen von BIOS-Funktionen, etc.)

– run level S ist üblicherweise ein Modus, in dem sich nicht mehrere Benutzer am Rechner anmelden können, sondern nur der `root`, und auch der nur an der Konsole. Es gibt kein Netzwerk, kein X-Windows. In diesem run level kann der `root` ungestört Festplatten restaurieren, Änderungen an der Konfiguration machen, oder was auch immer.

– run level 6 bedeutet: den Rechner herunterfahren (durch 0 hindurch), und wieder bis zum default run level (hier: 2) hochfahren

Unter SuSE-Linux ist der run level 1 der Modus, in dem sich zwar mehrere Benutzer anmelden können (an den virtuellen Konsolen 1 bis 6). Aber es läuft kein Netzwerk (also kein `inetd`, kein `lpd`, kein `nfs`, etc.) und kein X-Windows. Mit einem Wechsel zu run level 2 werden alle konfigurierten Netzwerkdienste gestartet. Wenn man sich in run level 2 angemeldet hat (an einer Konsole), dann kann man mit `startx` das X-Windows starten. In run level 3 geschieht dies automatisch, und man findet den `xdm` oder `kdm` vor, unter denen man sich gleich unter X-Windows anmelden kann, statt an der Konsole. Die run level 4 und 5 sind bei SuSE unbenutzt.

RedHat dagegen hat run level 5 als Anmeldung an `xdm` bzw. `kdm` ausgesucht.

In der `/etc/inittab` ist weiterhin definiert, was beim Ansteuern eines run level passieren soll. Unter SuSE ist dies ein Aufruf von `/sbin/init.d/rc` mit dem gewünschten run level als Parameter. Dadurch werden wiederum alle Skripte eines Verzeichnisses aufgerufen, die mit `S` beginnen (für *Start*), wenn von einem niedrigeren run level ein bestimmter neuer erreicht werden soll, oder alle Skripte, die mit `K` beginnen (für *Kill*), wenn von einem run level zu einem niedrigeren gewechselt werden soll. Das entsprechende Verzeichnis ist abhängig vom verwendeten System, und vom run level. Unter SuSE sind es ab Version 6.0 die Verzeichnisse `/sbin/init.d/rc0.d` (für run level 0) bis `/sbin/init.d/rc6.d` (für run level 6).

Beim Hochfahren werden die `S*`-Skripte eines run levels mit dem Parameter `start` aufgerufen (unter SuSE von `/sbin/init.d/rc`); beim Verlassen eines run levels die `K*`-Skripte mit dem Parameter `stop`.

SuSE hat jetzt eine weitere Eigenheit: Jeweils ein Start- und ein ein Kill-Skript sind identisch, und liegen in `/sbin/init.d`. Die aufgerufenen `S`- und `K`-Skripte sind nur symbolische Links auf eine wirkliche Datei.

Beispiel:

```
klaus@aw38:~ > ls -l /sbin/init.d/rc2.d/*nfsd*
lrwxrwxrwx 1 root root 9 Apr 3 07:10 /sbin/init.d/rc2.d/K23pcnfsd -> ../pcnfsd
lrwxrwxrwx 1 root root 9 Apr 3 07:10 /sbin/init.d/rc2.d/S17pcnfsd -> ../pcnfsd
```

Das tatsächlich aufgerufene Skript `/sbin/init.d/pcnfsd` erkennt am Parameter `start` bzw. `stop`, ob es sich um Hoch- oder Herunterfahren handelt, und startet oder stoppt dementsprechend das Programm `pcnfsd`.

- An diesem ganzen Mechanismus sieht man recht deutlich, daß ein Unix eigentlich nur aus einfachen Programmen besteht, die großteils sogar nur Skripte sind. Erst durch die enge Verzahnung wird daraus ein komplexes System, an dem sich aber letztlich fast alles mit Bordmitteln anpassen und korrigieren läßt.

Das Skript habe ich aus der vorhandenen Datei `/sbin/init.d/skeleton` übernommen, und nur abgeändert. Die englischen Kommentare stammen noch aus dem Original.

```
#! /bin/sh
# Time-stamp: "04.02.03 01:39 automount_nfs_noauto klaus@wachtler.de"
# (getetest auf aw41)
#
######################################################################
#
#           ///| |||\    || ||||    //
#          ////| |||\\   || ||||   //
#         ////|| ||||\\  || ||||  //
#        //// || |||| \\ || |||| //
#       ////==|| ||||  \\|| ||||//
#      ////   || ||||   \\| |||//
#     ////    || ||||    \| ||//
#                           |//
#         ||||    /|    //  //  ///| ||||=====//  ////=======
#         ||||   /||   //  //  ////| ||||    //  ////
#         ||||  //||  //  //  ////|| ||||   //  ////
#         |||| // || //  //  //// || ||||  //  ////
#         ||||//  ||//  //  ////==|| ||||=((  ((((=======
#         |||//   |//  //  ////   || ||||  \\  \\\\
#         |||/    |/  //  ////    || ||||   \\  \\\\=======
#
#####################################################################
#
```

```
# Copyright:
############
#
# (C) 2001-2002 AnyWare
#
# Dipl.Ing. Klaus Wachtler
# Breidingstr. 17
# 29614 Soltau
#
# Tel. (+49)-5191-70 2 71, (+49)-171-45 53 039
# Fax: (+49)-5191-70 2 72
# eMail AnyWare@Wachtler.de
#
#####################################################################
#
# Aufgabenstellung:
###################
#
# Mounten von NFS-Verzeichnissen im Hintergrund
#
#####################################################################
#
# Geschichte:
#############
#
# Datum:     Name: Vers. Aenderung:
#
# 07.02.2001 kw    0.0   erster Entwurf (übernommen aus
#                        /sbin/init.d/skeleton von SuSE 6.2)
#
# 09.01.2002 kw          Ergänzungen, mehrfache Versuche bis Erfolg
#
#
#####################################################################
#
# Dieses Skript kann mit dem Parameter "start" aufgerufen werden, und
# mountet dann alle Verzeichisse, die in der /etc/fstab mit dem Typ
# "nfs" (in der dritten Spalte) eingetragen sind, und bei den Optionen
# (vierte Spalte) "noauto" eingetragen ist.
# Diese Verzeichnisse werden im Hintergrund gemountet, und blockieren
# deshalb nicht das weitere Booten, wenn das Skript beim Starten des
# Rechners abgearbeitet wird.
#
# Sinnvoll ist das Skript vor allem bei Notebooks o.ä., die manchmal
# an einem Netz hängen (und dann etwas mounten sollen), manchmal aber
```

```
# ohne Netzanbindung booten (und dann nicht minutenlang auf eine
# Verbindung warten sollen, die es nie geben wird).
#
# Ab der Version vom 09.01.2002 wird das Mounten nicht nur einmal
# probiert, sondern das Skript ruft für jedes zu mountende
# Verzeichnis eine shell im Hintergrund auf, und versucht darin das
# Mounten dann etwas länger (für den Fall, daß ein Server erst nach dem
# Client hochkommt).
#
# Installation:
#
# Dieses Skript muß nach /sbin/init.d/automount_nfs_noauto (alte
# SuSE-Versionen, bis etwa 6.x) beziehungsweise nach /etc/rc.d (alle
# anderen Linux) kopiert werden.
# Dann müssen in /sbin/init.d/rc2.d (alte SuSE) beziehungsweise in allen
# gewünschten /etc/rc.d/rc?.d (alle anderen; beispielsweise in
# /etc/rc.d/rc3.d bis /etc/rc.d/rc5.d) zwei symbolische Links
# angelegt werden:
# (Beispiel alte SuSE:)
#   cd /sbin/init.d/rc3.d
#   ln -s  ../automount_nfs_noauto S40automount_nfs_noauto
#   ln -s  ../automount_nfs_noauto K40automount_nfs_noauto
#   cd /sbin/init.d/rc4.d
#   ln ...
#   ...
# (Beispiel alle anderen:)
#   cd /etc/rc.d
#   for i in 3 4 5; do
#       cd rc$i.d;
#       ln -s  ../automount_nfs_noauto S22automount_nfs_noauto;
#       ln -s  ../automount_nfs_noauto K02automount_nfs_noauto;
#       cd ..;
#   done
#
# Die S*- und die K*-Skripte (bzw. die über die symbolischen Links mit
# diesen Namen adressierten Programme) werden in ihrer alphabetischen
# Reihenfolge ausgeführt.
# In den obigen Beispielen sind S22... und K02... ggf. entsprechend
# abzuändern, so daß beim Starten erst alle Skripte ausgeführt werden,
# die von automount_nfs_noauto benötigt werden (insbesondere werden dies
# das Starten des Netzwerks an sich und das Routing sowie der port mapper
# sein, der von NFS benötigt wird), und erst nach diesem Skript dürfen
# die kommen, die von den gemounteten Verzeichnissen abhängen.
# Wird beispielsweise das Netzwerk mit S05network gestartet, das
# Routing mit S07route und der port mapper mit S08portmap, dann muß
```

```
# dieses Skript mindestens mit S09... beginnen, weil es sonst
# unsinnigerweise vor dem port mapper gestartet wird.
# Beim Herunterfahren mit den K-Skripten analog, wobei das vorliegende
# Skript beim Herunterfahren z.Zt. gar nichts macht (beim Beenden von
# NFS werden die gemounteten Verzeichnisse ohnehin gekappt, also
# brauchen wir das hier nicht zu machen).
# Insofern könnte man die K-Skripte beziehungsweise die symbolischen
# Links auch gleich weglassen...
#
#
######################################################################
#
# Umgebung:
############
#
# Windows 32 API
# AnyWare logfile
# VarArr<>
# Qt 1
# Qt 2
# omniORB 2.7.1
# omniORB 3.5
#
######################################################################
#
# Getestet:
###########
#
# SuSE Linux 6.2 bis 7.3, bash 2.05
#
######################################################################



# /etc/rc.config (falls vorhanden) in aktueller shell ausführen:
test -e /etc/rc.config && . /etc/rc.config


# Die Variable liste wird zu einem String, der -mit Leerzeichen
# getrennt- alle zu mountenden Verzeichnisse enthält.
#
# Zur Erklärung wird folgende /etc/fstab genommen:
#   # Demo-/etc/fstab
#   /dev/hda1    swap                 swap         defaults       0 0
#   /dev/hda2    /                    ext2         defaults       1 1
#
```

```
#   /dev/hdc     /cdrom              iso9660       ro,noauto,user 0 0
#
#   /dev/fd0     /floppy             auto          noauto,user    0 0
#
#   aw33:/home/klaus /home/aw33_klaus nfs        noauto,user,exec 0 0
#
#   aw33:/home/miniarchiv /home/miniarchiv nfs    noauto,user,exec 0 0
#
#   server:/home/fileserver /home/fileserver nfs  noauto,user,exec 0 0
#
#   none            /proc                   proc      defaults      0 0
#
# Erklärungen:
#
# 1) "reguläre Ausdrücke" (regular expression,
#    RE) bezeichnen Strings,
#    die nach bestimmten Regeln aufgebaut sind,
#    und auf einen gegebenen Text entweder zutreffen
#    oder halt nicht.
#    Beispiel: der RE "fängt mit 'A' an, dann kommen
#    zwei Ziffern, dann ein 'B', dann beliebig viele
#    Ziffern, dann ein 'Z'" trifft auf die
#    Texte "A79B123456789Z" und "A12B11Z" zu,
#    aber nicht auf den Text "AttBZ".
#    RE sind in "man 7 regex" beschrieben.
#    Leider gibt es davon zwei Varianten:
#    a) "basic RE" (alte, einfachere Version)
#       und
#    b) "modern" oder "extended" RE.
#    Tatsächlich werden REs natürlich nicht mit Worten beschrieben
#    ("fängt mit 'A' an..."), sondern etwas kryptischer.
#    Dazu mehr weiter unten und in 'man 7 regex'.
#
# 2) 'cat /etc/fstab' kopiert die Datei
#    /etc/fstab zur Standardausgabe
#
# 3) 'col -x' kopiert von Standardeingabe
#    zur Standardausgabe, ersetzt aber
#    alle TAB durch einige Leerzeichen.
#    Dadurch wird die Behandlung in den
#    folgenden "regulären Ausdrücken"
#    einfacher.
# 4) egrep filtert aus seiner Standardeingabe
#    alle Zeilen, auf die der
#    im einzigen Argument angegebene reguläre
```

```
#    Ausdruck zutrifft, und kopiert
#    diese Treffer zur Standardausgabe.
#    egrep verwendet "modern" REs.
#    In diesem Fall sind alle Zeilen aus
#    /etc/fstab gewünscht, deren
#    dritte Spalte 'nfs' heißt, und bei
#    denen in der vierten Spalte
#    das Wort 'noauto' vorkommt.
#    Es werden zwei grep verwendet:
#    - der erste läßt alles durch, was kein
#      Kommentar ist,
#    - der zweite läßt alles durch, was in der
#      dritten Spalte "nfs" enthält, und in der
#      vierten irgendwo "noauto".
#    Das erste grep hat den RE "^[ ]*#.*$".
#    Dabei steht '^' für den Zeilenanfang,
#    '[ ]*' für beliebig viele Leerzeichen,
#    '#' für das Kommentarzeichen selbst
#    und '.*' für beliebig viele Zeichen
#    bis zum Zeilenende '$'.
#    Der RE trifft also auf alle Kommentarzeilen
#    in der /etc/fstab zu.
#    Mit der Option -v wird die Wirkung des grep
#    umgedreht; es werden also alle
#    Nicht-Kommentarzeilen durchgelassen.
#    Das zweite grep hat dann den RE
#    "^[^ ]+[ ]+[^ ]+[ ]+nfs[ ]+[^ ]*noauto.*$".
#    Der reguläre Ausdruck beginnt
#    mit einem '^' (für: Zeilenanfang),
#    dann kommt '[^ ]+' (mindestens ein
#    Nicht-Leerzeichen für die erste
#    Spalte),
#    dann '[ ]+' (mindestens ein Leerzeichen
#    für den Platz zwischen erster
#    und zweiter Spalte),
#    dann wieder '[^ ]+' (mindestens ein
#    Nicht-Leerzeichen für die zweite Spalte),
#    dann '[ ]+' (mindestens ein Leerzeichen
#    für den Platz zwischen zwischen zweiter
#    und dritter Spalte), dann
#    'nfs', weil in der dritten Spalte
#    genau dieser Text stehen soll,
#    dann '[ ]+' (mindestens ein Leerzeichen
#    für den Platz zwischen zwischen dritter
#    und vierter Spalte), dann
```

```
#    '[^ ]*noauto.*' (keines, eines oder mehrere
#    Nicht-Leerzeichen, dann der Text 'noauto',
#    dann beliebige Zeichen), dann
#    '$' für das Zeilenende.
#    Alle Zeilen, auf die das beschriebene
#    Muster zutrifft, werden zur Standardausgabe
#    weitergereicht.
#    Die Standardausgabe von egrep besteht also
#    aus den folgenden Zeilen:
#      aw33:/home/klaus /home/aw33_klaus  nfs  noauto,user,exec   0 0
#      aw33:/home/miniarchiv /home/miniarchiv nfs noauto,user,exec  0 0
#      server:/home/fileserver /home/fileserver nfs noauto,user,exec 0 0
#
# 5) sed kopiert ebenfalls von der Standardeingabe
#    zur Standardausgabe. Dabei kann es aber den
#    Text editieren. Die Kommandos, wie editiert
#    werden soll, werden hinter der Option '-e'
#    angegeben.
#    Mit dem Kommando s/regulärer_Ausdruck/neuer_Text/
#    wird ein Text, auf den 'regulärer_Ausdruck'
#    zutrifft, durch den Text 'neuerText' ersetzt.
#    Der anzugebende RE ist jetzt leider -anders als
#    bei egrep- ein "basic RE".
#    Hauptunterschied: es gibt kein '+' für "eines
#    oder mehrere". Stattdessen muß man das was man
#    haben will, einmal hinschreiben, und dann nochmal
#    mit einem '*' dahinter (für: "keines, eines,
#    oder mehrere").
#    Zudem kann man bei sed im RE einzelne Bereiche
#    mit '\(' und '\)' klammern. Das hat für den
#    RE selbst erstmal keine Auswirkung, aber die
#    so markierten Bereiche aus einem zum RE passenden
#    Text können im Ersetzungstext wieder
#    verwendet werden. '\1' ist der erste
#    geklammerte Ausdruck, '\2' der zweite usw.
#    'sed -e "s/^[^ ][^ ]*[ ][ ]*\([^ ][^ ]*\).*$/\1/"'
#    ist also so zu lesen:
#    Ersetze aus der Standardeingabe alles, worauf der
#    RE '^[^ ][^ ]*[ ][ ]*\([^ ][^ ]*\).*$' zutrifft,
#    durch '\1', also den Inhalt des ersten Klammerpaares
#    im RE.
#    Zum RE:
#    '^' steht wieder für einen Zeilenanfang.
#    '[^ ][^ ]*' steht für: genau ein Nicht-Leerzeichen
#    ('[^ ]'), gefolgt von keinem, einem, oder mehreren
```

```
#    Nicht-Leerzeichen ('[^ ]*'); zusammen also mindestens
#    einem Nicht-Leerzeichen (für die erste Spalte).
#    Dann kommt eines oder mehrere Leerzeichen
#    ('[ ][ ]*') zwischen der ersten und der zweiten
#    Spalte.
#    Dann kommen wieder eines oder mehrere
#    Nicht-Leerzeichen ('\([^ ][^ ]*\)', für
#    die zweite Spalte). Dieser Bereich ist
#    mit '\(' und '\)' geklammert, und kann im
#    Ersetzungstext später als '\1' verwendet werden.
#    Danach dürfen beliebig viele beliebige Zeichen
#    ('.*') bis zum Zeilenende ('$') folgen.
#    Jeder Text, auf den dieses Muster zutrifft (das
#    ist in unserem Fall hoffentlich jede Zeile,
#    da unser egrep-Kommando so gebaut ist, daß nur solche
#    Zeilen geliefert werden) wird ersetzt durch '\1',
#    also der zweiten Spalte aus den gelieferten Zeilen,
#    weil dieser Bereich im RE entsprechend geklammert ist.
#    Die Standardausgabe von sed besteht also
#    aus den folgenden Zeilen:
#      /home/aw33_klaus
#      /home/miniarchiv
#      /home/fileserver
#
# 6) Diese ganze Kommandokette:
#      cat /etc/fstab|col...|sed -e /.../.../
#    würde also aus der Beispiel-/etc/fstab
#    die Ausgabe:
#      /home/aw33_klaus
#      /home/miniarchiv
#      /home/fileserver
#    erzeugen.
#
# 7) Wenn man in der shell ein Kommando (oder eine ganze
#    pipe, wie hier) mit verkehrten Apostrophen
#    klammert (also mit '`'), dann wird in der
#    Kommandozeile der geklammerte Bereich als Programm ausgeführt,
#    aus dem Kommando entfernt und durch seine eigene Standardausgabe
#    ersetzt.
#    Besteht die Ersetzung aus mehreren Zeilen, dann werden
#    die einzelnen Zeilen mit Leerzeichen getrennt
#    aneinandergereiht.
#
# Die folgende Anweisung entspricht also in unserem
# Beispiel der Zeile:
```

```
#    liste="/home/aw33_klaus /home/miniarchiv /home/fileserver"
#
liste=`cat /etc/fstab|col -x|\
       egrep -v "^[ ]*#.*$"|\
       egrep "^[^ ]+[ ]+[^ ]+[ ]+nfs[ ]+[^ ]*noauto.*$"|\
       sed -e "s/^[^ ][^ ]*[ ][ ]*\([^ ][^ ]*\).*$/\1/"`



echo $liste



# Determine the base and follow a runlevel link name.
base=${0##*/}
link=${base#*[SK][0-9][0-9]}

# Force execution if not called by a runlevel directory.
#   ???  test $link = $base && START_FOO=yes
#   ???  test "$START_FOO" = yes || exit 0

# The echo return value for success (defined in /etc/rc.config).
return=$rc_done

case "$1" in
    start)
#echo -n "Starting service foo"
## Start daemon with startproc(8). If this fails
## the echo return value is set appropriate.

#startproc /usr/sbin/foo || return=$rc_failed

        echo Versuche, NFS-Pfade im Hintergrund zu mounten...

        # f wird nacheinander auf einen der Werte
        # "/home/aw33_klaus", "/home/miniarchiv" und
        # "/home/fileserver" gesetzt:
        for f in $liste
        do
            echo Mounte $f im Hintergrund...
            # alte Version: einmal versuchen, zu mounten;
            #               dann aufgeben:
            # nohup mount $f >/dev/null 2>&1 &

            # Version 09.01.2002:
            # solange (im Hintergrund) versuchen, bis mount
```

```
            # geklappt hat.
            # Dazu wird nach dem ersten Versuch eine Pause von
            # 1 sec gemacht,
            # nach dem zweiten Versuch eine Pause von 2 sec, und
            # so fort bis 60 sec. Dann werden die Pausen nicht
            # mehr länger.
            # Im Erfolgsfall wird eine kurze Meldung nach
            # /dev/tty1 ausgegeben; das sollte die erste
            # Textkonsole sein.
            nohup bash -c '(export n=0;\
                            until mount '"$f"' >/dev/null 2>&1;\
                            do\
                               if [ $n -lt 60 ];\
                               then\
                                    n=`expr $n + 1`;\
                               fi;\
                               sleep $n;\
                            done;\
                            echo -e "\n'"$f"' gemounted!">/dev/tty1;\
                            echo -e "mfgkw">/dev/tty1\
                            )' >devnull 2>&1 &
        done

        #echo "Fertig mit automount_nfs_noauto!"
echo -e "$return"
;;
    stop)
echo "umounten der automount_nfs_noauto-Verzeichnisse"
## Stop daemon with killproc(8) and if this fails
## set echo the echo return value.

        # f wird nacheinander auf einen der Werte
        # "/home/aw33_klaus", "/home/miniarchiv" und
        # "/home/fileserver" gesetzt:
        for f in $liste
        do
            echo umounte $f im Hintergrund...
            nohup umount $f >/dev/null 2>&1 &
        done

#killproc -TERM /usr/sbin/foo || return=$rc_failed

echo -e "$return"
;;
    restart)
```

```
## If first returns OK call the second, if first or
## second command fails, set echo return value.
$0 stop  &&  $0 start  ||  return=$rc_failed
;;
    reload)
## Choose ONE of the following two cases:

## First possibility: A few services accepts a signal
## to reread the (changed) configuration.

#echo -n "Reload service foo"
#killproc -HUP /usr/sbin/foo || return=$rc_failed
#echo -e "$return"

## Exclusive possibility: Some services must be stopped
## and started to force a new load of the configuration.

$0 stop  &&  $0 start  ||  return=$rc_failed
;;
    status)
echo -n "Checking for service foo: "
## Check status with checkproc(8), if process is running
## checkproc will return with exit status 0.

#checkproc /usr/sbin/foo && echo OK || echo No process
;;
    probe)
## Optional: Probe for the necessity of a reload,
## give out the argument which is required for a reload.

#test /etc/foo.conf -nt /var/run/foo.pid && echo reload
;;
    *)
echo "Usage: $0 {start|stop|status|restart|reload[|probe]}"
exit 1
;;
esac

# Inform the caller not only verbosely and set an exit status.
test "$return" = "$rc_done" || exit 1
exit 0
```

Kapitel 4

EMACS

Kapitel 5

CVS

CVSup dient zur Verteilung mehrerer (synchroner) Repositories, und wird für größere Projekte (XFree86, FreeBSD) eingesetzt.

Kapitel 6

Was noch fehlt:

Kapitel 7

Glossar

In diesem Abschnitt werden in alphabetischer Reihenfolge wichtige Begriffe vorgestellt, und bei Bedarf wird auf weitere Erklärungen verwiesen.

Damit wird ein Mittelweg beschritten zwischen einem Stichwortverzeichnis (in dem alle Begriffe ohne Erklärung und Verweise auf zugehörigen Text stehen) und einer genauen Beschreibung (die für eine erste Information zu unübersichtlich ist).

Begriffe aus folgenden Gruppen werden in der gezeigten Form aufgeführt:

`beispielname` *(Schlüsselwort* C *und* C++*)*
: So wird ein Schlüsselwort aus *C* und *C++* beschrieben.

`beispielname` *(Schlüsselwort* C++*)*
: So wird ein Schlüsselwort beschrieben, das zu *C++* gehört (aber nicht zu *C* nach ANSI-Standard, auch wenn vielleicht manche Implementationen das Wort bereits verwenden)

`beispielname` **oder *Begriff*** (C- *und* C++-*Bibliothek)*
: Ein `beispielname` oder ein *Begriff* aus der Standardbibliothek von *C* ebenso wie von *C++*.

`beispielname` **oder *Begriff*** (C++-*Bibliothek)*
: Ein `beispielname` oder ein *Begriff* aus der Standardbibliothek von *C++*.

`beispielname` **oder *Begriff***
: sonstiger *Begriff*

Hier jetzt die Liste der Begriffe:

ANSI
: ANSI ist ein US-amerikanisches Normungsinstitut; etwa vergleichbar mit dem deutschen DIN.

Compiler
: Ein Compiler ist ein Programm, welches ein Programm von einer Programmiersprache (beispielsweise *C++*) in eine andere (meist eine konkrete Maschinensprache) übersetzt.

Free Software Foundation
: Die Free Software Foundation (FSF) ist eine Stiftung in den USA, die freie Software fördert.

FSF
: FSF ist die Abkürzung für *Free Software Foundation*.

GNU
: Ein ehrgeiziges (aber trotzdem von Erfolg gekröntes) Projekt der FSF ist GNU. Darin wird ein komplettes unixähnliches Betriebssystem entwickelt, das technisch mindestens so gut sein soll wie kommerzielle Unixvarianten, aber vollständig aus frei verfügbarer Software besteht.

  Ein heute üblicherweise „Linux“ genanntes System besteht tatsächlich aus einem Kernel, dem eigentlichen Linux, sowie aus einer Unmenge Software aus dem GNU-Projekt. Deshalb sollte das komplette System eigentlich nicht „Linux“ heißen, sondern „GNU/Linux“.

  Von der FSF wird aber auch die Entwicklung eines eigenen Kernels betrieben (das „Hurt“-Projekt). Ob damit allerdings gegen das weit verbreitete Linux anzukommen ist, erscheint fraglich.

  Aus dem GNU-Projekt entstammt auch die *Gnu compiler collection* (*gcc*), in der etliche Compiler enthalten sind, unter anderem für *C*, *C++*, Fortran, Pascal, Ada, Java, und andere.

  Weiterhin ist in die Rubrik GNU auch der EMACS einzuordnen, der möglicherweise einzig wahre Editor.

ISO
: ISO ist eine internationale Institution zur Erstellung von Normen.

  Die ANSI-Normen zu *C* und *C++* wurden als ISO-Normen übernommen.

Linker
: Ein Linker setzt mehrere Teilprogramme, die in Maschinensprache vorliegen

(also Ausgaben eines Compilers) zu einem lauffähigen Programm zusammen.

Maschinensprache
Maschinensprache ist die Programmiersprache, die ein bestimmter Prozessor direkt verarbeiten kann.

Microsoft

Microsoft ist ein Softwarehersteller aus den USA.

Präprozessor
Der Präprozessor ist ein Programm, das vor dem eigentlichen Übersetzen durch den *C*- oder *C++*-Compiler den Quelltext manipulieren kann.

Siehe [KW:0 ist false].

Literaturverzeichnis

[Strou I] Bjarne Stroustrup:
Die C++ Programmiersprache
1. Auflage
Addison-Wesley, 1985

[Strou II] Bjarne Stroustrup:
Die C++ Programmiersprache
2. Auflage
Addison-Wesley, 1992

[Strou IV] Bjarne Stroustrup:
Die C++ Programmiersprache
4. Auflage
Addison-Wesley, 1999

[Cline: C++ FAQs] Marshall Cline, Greg Lomow, Mike Girou:
C++ FAQs
2nd Edition
Addison-Wesley, 1999
Eine Sammlung vieler kleinerer Aspekte zu C++, CORBA und COM

[Josuttis: Stdbib.] Nicolai Josuttis::
Die C++-Standardbibliothek
Eine detaillierte Einführung in die vollständige ANSI/ISO-Schnittstelle
1. Auflage
ISBN 3-8273-1023-7
Addison-Wesley, 1996

[RRZN] *Die Programmiersprache C++ für C-Programmierer*
RRZN-Klassifikationsschlüssel SPR.C 2
3. Auflage

Revision B Regionales Rechenzentrum für Niedersachsen/Universität Hannover
Juni 1993

[KW:0 ist false] Klaus Wachtler:
0 ist false, alles andere true
Selbstverlag Klaus Wachtler
Breidingstraße 17
29614 Soltau
http://www.wachtler.de
September 2002

[KW:C++] Klaus Wachtler:
C++: Mehr als C
Selbstverlag Klaus Wachtler
Breidingstraße 17
29614 Soltau
http://www.wachtler.de
September 2002

[ct 9/94 Hartinger] Roland Hartinger:
Syntax-Puzzle
Die jüngsten C++-Erweiterungen
c't 9/1994, S. 184
Korrektur in c't 10/1994, S. 12
Kurzabriß der neueren Sprachelemente (außer templates):
Ausnahmebehandlung, RTTI, Namensräume (name spaces)

[ct 9/94 Hueskes] Ralf Hüskes:
Geduldsspiel
Der kommende C++-Standard
c't 9/1994, S. 180
Zur Standardisierung von C++, Interview mit Bjarne Stroustrup

[ct 11/94 Langner] Angelika Langner:
Instantware
Die neuen Standardklassen für C++
c't 11/1994, S. 272

[Eckel: TIC++ 1] Bruce Eckel:
Thinking in C++
Volume 1
http://www.mindview.net/Books/DownloadSites

[Eckel: TIC++ 2] Bruce Eckel:
Thinking in C++
Volume 2
http://www.mindview.net/Books/DownloadSites

[man make] Dennis Morse, Roland McGrath:
make - GNU make utility to maintain groups of programs
`man 1L make`
Linux man pages, 1989-2000 by the Free Software Foundation, Inc., `http://www.fsf.org`

[info make] Free Software Foundation:
make - GNU make utility to maintain groups of programs
`info make`
Linux man pages, 1989-2000 by the Free Software Foundation, Inc., `http://www.fsf.org`

[man bash] Brian Fox, Chet Ramey:
bash - GNU Bourne-Again Shell
`man 1 bash`
Linux man pages, 1989-1999 by the Free Software Foundation, Inc., `http://www.fsf.org`
Brian Fox, Free Software Foundation, `bfox@gnu.ai.MIT.Edu`
Chet Ramey, Case Western Reserve University, `chet@ins.CWRU.Edu`

[info bash] Brian Fox, Chet Ramey:
bash - GNU Bourne-Again Shell
info bash
Linux man pages, 1989-1999 by the Free Software Foundation, Inc., `http://www.fsf.org`
Brian Fox, Free Software Foundation, `bfox@gnu.ai.MIT.Edu`
Chet Ramey, Case Western Reserve University, `chet@ins.CWRU.Edu`

[Herold: Shell] Helmut Herold:
Linux - Unix Shells
Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh
3. Auflage
Addison Wesley, 1999
ISBN 3-8273-1511-5

[LaTeX2html] Nikos Drakos:
The LaTeX2`HTML` *Translator*
(Onlinehilfe zu LaTeX2`HTML` *Version 96.1)*
CTAN (Comprehensive TeX archive), z.B. `ftp.shsu.edu`, `ftp.dante.de`

Index