Skript zur Vorlesung

Einführung in die Informatik

Version 1.00

Klaus Wachtler

03.09.2002

Inhaltsverzeichnis

Kapitel 1

Einleitung

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, weil die auch alles vergessen.

Teile dieses Skripts sind an andere Werke angelehnt; teilweise auch schamlos übernommen ([**?**]klinge-grundlagen, diverse c't-Artikel, weitere Bücher aus dem Literaturverzeichnis; siehe **C Literatur** auf Seite 95).

Mit diesem Skript kann man sicher nicht für alle Zeiten auf tiefergehende Literatur verzichten. Wesentlich detailliertere Informationen findet man über das Literaturverzeichnis; siehe **C Literatur** auf Seite 95.

Insofern darf man sich nicht zuviel erwarten: als Tischvorlage zur Vorlesung sollte der Text reichen.

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.

Copyright: Der vorliegende Text stammt von:

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

Teil I

Vorlesung

Kapitel 2

Grundbegriffe

Als lockerer Rundumschlag werden im folgenden einige wichtige Begriffe erwähnt und erklärt, die zum Verständnis des restlichen Skripts nötig sind.

Informatik ist die Wissenschaft von der automatischen Datenverarbeitung.

Die ersten Schritte wurden mit mechanischen Konstruktionen unternommen. Seit etwa der Mitte des Jahrhunderts begann die Entwicklung mit elektronischen Schaltungen. Dies waren anfangs Röhren und Relais, inzwischen fast ausschließlich Halbleiterschaltungen.

Zumindest in Teilbereichen wird in den nächsten Jahren und Jahrzehnten ein Umstieg auf optische Bauteile stattfinden.

Dies alles betrifft aber nur die sogenannte *Hardware* direkt. Mit Hardware bezeichnet man alle Bestandteile eines Rechensystems, die man anfassen kann: Gehäuse, Stromversorgung (nein, die bitte nicht anfassen!), Platinen, die Bauteile darauf, Diskettenlaufwerke, Disketten, Festplatten, Magnetbänder und so weiter[1], und kaputtgeht, wenn man mit dem Hammer draufhaut.

Der (künstliche) Gegenbegriff ist die *Software*. Damit ist alles gemeint, was die Hardware erst mit Leben füllt, nämlich Programme und Daten, also das Wissen eines Computers. Dies Wissen (die Daten) werden meistens unabhängig von der

[1] Wie jedes Fach hat auch die Informatik eine eigene Sprache entwickelt, die durch extrem schnelle Entwicklungen entsprechend dynamisch ist. Wie grundlegend und relativ alt der Begriff *hardware* ist, sieht man daran, daß er ausnahmsweise nicht künstlich geschaffen ist, sondern aus dem täglichen Leben praktisch ohne Bedeutungsverlust übernommen wurde: hardware steht für Eisenwaren, also das was man im Eisenwarengeschäft kaufen kann (Schaufel, Nägel, Beil...)

Hardware betrachtet[2]

Ein Informatiker wird sich sicher mehr auf die Softwareseite konzentrieren. Trotzdem muß er auch für die verwendete Hardware ein Grundwissen und Verständnis haben, um sinnvoll arbeiten zu können.

Grundsätzlich sind zwei Arten von Rechnern möglich:

- Analogrechner
- Digitalrechner

Analogrechner haben nur noch geringe Bedeutung; wenn man von Rechnern oder Computern spricht, meint man im allgemeinen Digitalrechner. Analogrechner rechnen nicht tatsächlich, sondern stellen die Werte von physikalischen oder technischen Problemen wieder als physikalische Werte dar, und zwar meist als elektronische Werte (Spannungen). Hat man beispielsweise ein technisches Gebilde, in dem Drücke und Strömungen vorkommen, und stellt diese Werte in einem Analogrechner durch Spannungen und elektrische Ströme dar, dann kann man den Analogrechner auf das gestellte Problem „programmieren“, indem man das gegebene technische System durch elektrische Addierer, Multiplizierer, Invertierer etc. ersetzt. Gibt man nun dem abgebildeten System (auf dem Analogrechner) irgendwelche Randbedingungen vor, kann man von den elektrischen Ausgängen des Analogrechners auf die Reaktionen an den Ausgängen des realen technischen Systems schließen. Analogrechner eignen sich also im wesentlichen nur zur Simulation von physikalischen Systemen und dadurch zur Lösung von gewissen Differentialgleichungen. Die „Rechenzeit“ ist klein, die Genauigkeit auch und der ganze Aufwand enorm; die Ergebnisse sind teilweise schwer reproduzierbar und oft höchst unsicher. Zu alledem haben Analogrechner einen sehr begrenzten Einsatzbereich.

Analogrechner hatten ihre große Zeit, als preiswerte Digitalrechner nicht erhältlich waren und für die meisten Probleme der Simulation nicht leistungsfähig genug waren. Der Name Analogrechner kommt daher, daß jeder Größe des originalen Problems eine sich analog verändernde Größe im Simulationsgerät entspricht, also

[2] Um irgendwelchen Diskussionen vorzubeugen: Wenn man mit dem Hammer auf eine Diskette eindrischt, macht man die Hardware kaputt, aber eigentlich nicht die Software. Man kommt an die Software nur nicht mehr heran. Analog: Man programmiert seinen Manta, indem man den Zündschlüssel umdreht und Vollgas gibt. Wenn man dann am Baum hängt, kann man feststellen, daß die Hardware ziemlich komplett zerstört ist und meinetwegen sogar der Zündschlüssel abgebrochen ist. Aber die *Stellung* des Zündschlosses kann trotzdem noch die alte sein, und gerade das ist die Software am Auto (beziehungsweise ein Teil davon)! Mit defekter Hardware hilft die Stellung des Zündschlosses halt auch nicht weiter.

einem hohen Druck im realen System entspricht eine hohe Spannung im Rechner und so weiter.

Digitalrechner dagegen stellen Zahlen durch diskrete Spannungswerte dar anstatt durch analoge, d.h. an jeder Leitung im Digitalrechner werden nicht beliebig viele Spannungen unterschieden, sondern nur eine sehr begrenzte Anzahl.

Im Regelfall werden nur zwei Spannungen unterschieden. Innerhalb der aktuellen Rechner sind das die Spannungen etwa 0 V (0 V–1.5 V) und etwa 5V (4 V–5 V). In der Übertragung zwischen Rechnern beispielsweise kommen auch andere Spannungen vor, wie etwa +12 V und etwa -12 V gegenüber Masse (z.B. RS-232-Schnittstelle bzw. V24 und andere Normen). Den beiden Spannungswerten ordnet man eine 0 und eine 1 zu, so daß man mit den beispielsweise 8 Spannungen (0 oder 5 V) auf 8 Leitungen eine achtstellige Binärzahl (0 oder 1) darstellen kann[3].

0 V könnte also vereinbarungsgemäß eine logische (gedachte) 0 sein, 5 V soll 1 sein; oder -12 V soll 1 sein und +12 V die 0.

Eine Binärzahl ist eine Zahl im Zweiersystem, in dem alle derzeitigen Computer rechnen. Als normal denkender Mensch rechnet man im Zehnersystem, in dem man die zehn Ziffern von 0 bis 9 zur Verfügung hat. Genauso kann man im Zweiersystem (Binärsystem, Dualsystem) rechnen, wo man nur die beiden Ziffern 0 und 1 hat. Auch wenn man selten etwas davon sieht, rechnen doch alle Computer (auch Taschenrechner) im Zweiersystem. Bevor man allerdings Ergebnisse sieht (wenn überhaupt), werden die Zahlen aus dem Zweiersystem wieder ins Zehnersystem umgewandelt.

Eine Stelle einer Binärzahl (also das, was entweder 0 oder eins sein kann), bezeichnet man als *Bit*. Das ist einerseits ein Kunstwort (<u>bi</u>nary digi<u>t</u>, also *binäre Ziffer*), aber andererseits auch ein tatsächlich verwendetes englisches Wort („ein Stückchen“).

Einige wenige Bits (heute meist 8) werden zu einem *Byte* zusammengefaßt. Vier Bit werden oft als *nibble* bezeichnet; dies entspricht genau einer Ziffer einer Hexadezimalzahl.

Größere Datenmengen werden in runden Zahlen gemessen; aber *rund* meint nicht rund im Sinne des Zehnersystems, sondern im Zweiersystem: Der Vorsatz *kilo* be-

[3] 0 V und 5 V waren jahrzehntelang die üblichen Spannungspegel in Logikschaltungen. Innerhalb von Rechnern werden aber zunehmend niedrigere Spannungen verwendet, weil bei den heutigen hohen Taktraten (über ein GHz hinaus) hohe Leistungen umgesetzt werden, die sich deutlich verringern lassen, wenn man mit niedrigeren Spannungen arbeitet. Deshalb liegen hoch getaktete Prozessoren schon unter 2 V für die Darstellung einer logischen 1.

deutet nicht 1000, sondern 1024 (also 2^{10}); ein *Mega* steht für kilo∗kilo=1048576 (2^{20}); ein *Giga* steht dementsprechend für kilo∗Mega=1073741824 (2^{30}); ein *Tera* steht für kilo∗Giga=1099511627776 (2^{40}).

Wenn ein Rechner also einen Arbeitsspeicher von 256 MB (Mega Byte) hat, dann stehen im Speicher 256∗1048576=268435456 Bytes zu je 8 Bits, also 2147483648 Bits zur Verfügung[4].

[4] Bei Festplatten geben die Hersteller allerdings im klaren Wiederspruch zu einschlägigen Normen die Speicherkapazitäten mit 1 000 für kilo, 1 000 000 für Mega etc. an; dadurch sehen die Kapazitäten etwas größer aus.

Kapitel 3

Geschichte

Zur Geschichte der Informatik siehe [Lit. C: Klingenberg: Grundlagen der Angewandten Informatik].

Ein lesenswerter Überblick ist auch in [Lit. C: Ebeling: Geist aus der Maschine] zu finden.

Kapitel 4

Schaltalgebra

Kapitel 5

Übliche Struktur eines Digitalrechners

Aufbau Rechner: RAM, ROM, PROM, EPROM, EEPROM, Flash, CPU, Register: a..., D..., PC, SP, Speicher, cache

realer/virtueller Speicher, Multitasking, Multiuser

5.1 Der Prozessor

Register, Adreßregister, Datenregister, PC, Framepointer, FPU, FPU-Register, Stackpointer

Kapitel 6

Betriebssysteme

CPM

DOS

Windows 3...., 95, 98, ME

OS/2

NT, Windows2000

DEC/VMS

Unix, Linux

Mach

QNX

Embedded/Controller

virtueller Speicher, multi user, multi tasking, Cluster

Kapitel 7

Programme, Prozesse

Ein *Programm* ist eine Menge von Arbeitsanweisungen, die auszuführen sind um ein vorgegebenes Problem zu lösen.

Insofern ist ein Programm eine konkrete Ausformulierung eines Algorithmus in Form einer Befehlsfolge.

Ein *Prozeß* dagegen ist ein Programm, das gerade von einem Rechner ausgeführt wird. Dazu wird das bis dahin nur gespeicherte Programm auf einer konkreten Maschine ausgeführt.

7.1 Programm

Ein Programm kann in einer Vielzahl von Sprachen formuliert sein:

- eine natürliche Sprache nach Wahl des Programmerstellers oder desjenigen, der das Programm auszuführen oder umzusetzen hat.

  Beispiel:

  ```
  Erstens:  Kontrolliere, ob Du mindestens 50.-- DM hast.
  Zweitens: Wenn Du weniger als 50.-- hast, dann hole 100.-- vom
            Automaten
  Drittens: gehe bis spätestens 20:00 Uhr in die Destille.
  Viertens: Versuche, aufrecht nach Hause zu kommen
  ```

Weiteres Beispiel des selben Programms in einer anderen Sprache:

```
erschd amohl schaust noch, obbsd gnuuch Gäld hast; mindestens an
Fuufzichä wäärsd woscheinli braung.
Wenns nedd langd, dann holst Der halt an Hunderder vom Ahtomahdn.
Bis achda schaust, dassd in dä Disdilln bisd.
Kumm obba mit Anstand haam, und versumbf nedd.
```

- eine künstliche Fachsprache, beispielsweise um ein Programm beziehungsweise einen Algorithmus klarer und kompakter auszudrücken als dies in natürlicher Sprache möglich ist.

  Beispiel (aus[Lit. C: Knuth: Fundamental Algorithms], Seite 141):
  Program M (*Find the maximum*). Register assignments: rA $\equiv m$, rI1 $\equiv n$, rI2 $\equiv j$, rI3 $\equiv k$, $X[i] \equiv \text{cell}(\mathtt{X}+i)$.

Line no.	LOC	OP	ADDRESS	Times	Remarks
01	X	EQU	1000		
02		ORIG	3000		
03	MAXIMUM	STJ	EXIT	1	Subroutine linkage
04	INIT	ENT3	0,1	1	*M1. Initialize.* $k \leftarrow n$.
05		JMP	CHANGEM	1	$j \leftarrow n, j \leftarrow X[n], k \leftarrow n-1$.
06	LOOP	CMPA	X,3	$n-1$	*M3. Compare.*
07		JGE	*+3	$n-1$	
08	CHANGEM	ENT2	0,3	$A+1$	*M4. Change m.* $j \leftarrow k$.
09		LDA	X,3	$A+1$	$m \leftarrow X[k]$
10		DEC3	1	n	*M5. Decrease k.*
11		J3P	LOOP	n	*M2. All tested?*
12	EXIT	JMP	*	1	Return to main program. ▮

  Knuth hat einen gedachten Computer erfunden, dessen Maschinensprache MIX ebenfalls frei erfunden ist. Die Assemblersprache von MIX heißt MIXAL, und darin werden in den Büchern von Knuth die Algorithmen vorgestellt und untersucht. In eben dieser Sprache ist das obige Beispiel formuliert.

- eine künstliche Programmiersprache, beispielsweise FORTRAN 66, Fortran 77, ADA, Pascal, BASIC und so fort; siehe dazu auch **11 Programmiersprachen** auf Seite 77.

- Maschinensprache eines bestimmten Computertyps; dies ist ebenfalls eine künstliche Sprache die im wesentlichen vom Prozessorhersteller vorgegeben wird.

  Ein Programm in dieser Sprache besteht aus einer Folge der möglichen Befehle des Prozessors. Dabei ist für jeden Befehl eine genaue Bitfolge definiert. Die Aneinanderreihung der passenden Bitfolgen ergibt das Programm.

> Bei der späteren Abarbeitung eines solchen Programms als Prozeß erscheinen die Werte dieser Bits als Spannung an den Eingangsleitungen des Prozessors und bewirken so die Ausführung der gewünschten Programmbefehle.

Teilweise kann ein Programm automatisch (von einem Konvertierprogramm) von einer Sprache in eine andere überführt werden. Beispiele hierfür sind zuvorderst natürlich Compiler, die einen Quelltext in Maschinensprache übersetzen. Es gibt aber auch Programme wie `f2c` (konvertiert einen FORTRAN 77-Quelltext nach C) oder `p2c` (konvertiert Pascal nach C).

Bei der Ausführung eines Programms gibt es zwei grundsätzliche Möglichkeiten:

- das Programm liegt in Maschinensprache des jeweiligen Rechners vor, und kann direkt von dessen Prozessor ausgeführt werden.

  Um ein solches Programm zu erstellen, wird üblicherweise der Quelltext des Programms in einer geeigneten Programmiersprache geschrieben, und dann von einem speziellen Programm, dem *Compiler*, in die Maschinensprache übersetzt.

  Eine Variante hiervon sind die sogenannten Assemblersprachen. Diese sind eine Klarschriftversion jeweils einer bestimmten Maschinensprache. Anstatt eines bestimmten Bitmusters um beispielsweise den Wert 2 zum Prozessorregister `D0` zu addieren, wird in einer solchen Sprache ein sogenanntes *mnemonic* des Befehls als Text hingeschrieben, etwa `ADD #2,D0`. Jede Assembleranweisung entspricht dabei genau einem Maschinenbefehl. Um ein Programm aus dieser Form in die wirkliche Maschinensprache zu übersetzen, wird ein sogenannter *Assembler* benötigt.

- Das auszuführende Programm liegt in einer Sprache vor, die nicht direkt vom Prozessor verstanden wird, und muß von einem anderen Programm (das in Maschinensprache vorliegen muß), dem sogenannten *Interpreter*, Anweisung für Anweisung ausgeführt werden.

Die Programme einiger Programmiersprachen werden (zumindest fast) ausschließlich von einem Compiler in Maschinensprache übersetzt (FORTRAN, C, C++, Pascal) und als Maschinenprogramme ausgeführt.

Programme einiger anderer Sprachen dagegen werden fast ausschließlich interpretiert (Shellskripte, die meisten BASIC-Dialekte).

Für einige Programmiersprachen sind sowohl Interpreter als auch Compiler verfügbar (beispielsweise einige BASIC-Versionen).

Dazwischen gibt es auch noch eine Mischform. Dabei wird das Programm aus dem Quelltext von einem Compiler in sogenannten *Pseudocode* (p-Code) übersetzt. Dieser Pseudocode ist einer Maschinensprache ähnlich, dafür ist aber tatsächlich kein Prozessor verfügbar. Stattdessen wird dieser Pseudocode auf einer realen Maschine wiederum von einem Interpreter ausgeführt. Dies wird bei der Sprache *Java* dazu genutzt, Programme unabhängig von dem verwendeten Zielsystem zu machen: Der Ersteller eines Programms übersetzt seinen Quelltext in den Pseudocode (ohne wissen zu müssen, auf welchem konkreten System das Programm laufen wird), und gibt das Programm in dieser Form an den Anwender weiter. Der Anwender benötigt auf seinen System einen Interpreter für den Pseudocode (dieser ist Teil der *run time*-Umgebung von Java, und ist abhängig von der Maschine, aber unabhängig vom konkreten Programm).

Eine andere Mischform wird von der Sprache Perl benutzt: Das Programm wird im Quelltext erstellt und gegebenenfalls an die Endanwender verteilt. Ähnlich wie bei einem wirklich interpretierten Programm wird zum Ausführen ein Hilfsprogramm benötigt. Dieses interpretiert aber nicht Anweisung für Anweisung während das gewünschte Programm ausgeführt wird, sondern übersetzt erst alle Anweisungen in Maschinencode, und startet diesen dann.

Noch eine andere Variante wird von VisualBASIC umgesetzt: Das erstellte Programm wird (in kompakter Form, beispielsweise mit verkürzten Variablennamen und ohne Kommentare) im Anschluß an einen kompletten Interpreter abgespeichert, und kann in dieser Form als ausführbares Programm verteilt werden. Die so erstellten Programme sind dabei größer als bei nur interpretierte Sprachen, weil in jedem erstellten Programm ein kompletter Interpreter enthalten ist.

Einige grundsätzliche Eigenschaften der möglichen Varianten im Überblick:

	Compiler	Interpreter	p-Code	Kompilieren bei Start	Interpreter enthalten
Beispiel	C, C++, Fortran, Pascal,	BASIC, Shell-skripte, LISP	Java	Perl	VisualBASIC
lauffähig auf verschiedenen Systemen	nein	ja	ja	ja	nein
effiziente (schnelle) Programme	ja	nein	nein	ja	nein
ohne weitere Programme lauffähig	ja	nein	nein	nein	ja
Quelltext vom Anwender sichtbar/änderbar	nein	ja	nein	ja	nein

Dazu folgende Anmerkungen:

- Die Zeile *lauffähig auf verschiedenen Systemen* bezieht sich auf das fertige Programm. Wenn der Quelltext zur Verfügung steht, kann der möglicherweise auf anderen Systemen in Maschinensprache übersetzt werden und ist dann natürlich auf dem anderen System lauffähig.

  Programme, die ohne erneutes Kompilieren auf vielen Systemen lauffähig sind, haben besondere Vorteile bei einem großen Kundenkreis. Beispiel: eine Bank möchte ihren Kunden Onlinebanking anbieten. Dann ist es ungeschickt, ein Programm nur für ein bestimmtes Betriebssystem anzubieten (wie es beispielsweise die Sparkassen versuchen), indem man es beispielsweise für Windows95/98 erstellt und verteilt. Selbst bei der starken Verbreitung dieser beiden Varianten treten Probleme auf, weil 60% Marktanteil eines Systems umgekehrt 40% der potentiellen Kunden ausschließt, die entweder gar kein Windows95/98 haben, oder intelligenterweise nicht gerade für sensiblen Datenverkehr verwenden wollen. Zudem kommen nach Windows95/98 irgendwann andere Versionen, für die das Programm wieder neu erstellt werden müßte. In einem solchen Fall kann man vielleicht besser ein Javaprogramm erstellen und verteilen; dann kann es jeder Kunde verwenden (so geschehen beispielsweise bei Consors).

- Wie *effizient* ein kompiliertes Programm ist, hängt natürlich auch von der Qualität des Compilers beziehungsweise Interpreters ab. Aber generell gilt, daß zu Maschinensprache kompilierte Programme meistens wesentlich schneller sind.

- Wenn ein Programm nur mit weiteren Hilfsprogrammen (insbesondere Interpreter, aber auch Datenbankserver oder ähnliches) lauffähig ist, dann hat das möglicherweise folgende Nachteile:

  - Die Hilfsprogramme sind unter Umständen nicht frei verfügbar, sondern der Endkunden muß Lizenzen erwerben. Wenn ich also ein Programm schreibe, das von einem Interpreter eines anderen Herstellers abhängig ist, dann muß mein Kunde den Interpreter bezahlen, ohne daß ich daran verdiene. Die finanzielle Mehrbelastung schreckt möglicherweise den Kunden ab, und ich habe nichts davon.
  - Die Fehlersuche beim Erstellen (oder noch schlimmer erst beim Endkunden) ist wesentlich schwieriger, wenn mehrere Programme beteiligt sind, auf die man nur bedingt Einfluß hat, und deren Interna meistens unbekannt sind.
  - Die Installation und eventuelle Updates werden komplizierter und fehlerträchtiger, weil meistens undurchsichtige Randbedingungen einzuhalten sind (welche Version meines Programms benötigt welche Version der Hilfsprogramme? Welche anderen Programme brauchen ebenfalls die Hilfsprogramme und in welcher Version?).

- Wenn ein Quelltext vom Anwender einsichtbar ist, kann das Vor- und Nachteile haben.

  Kommerzielle Anbieter lassen sich nicht gerne über die Schulter gucken, denn wenn ein Programm im Quelltext abgegeben wird, kann es von jedermann unkontrolliert übernommen werden (und komplett oder in Teilen als Raubkopie verwendet werden); man gibt das gesamte investierte *know how* preis. Zudem kann der Ersteller beim Auftreten von Problemen nicht mehr sicher davon ausgehen, daß wirklich das ausgelieferte Programm die Fehler verursacht; denn möglicherweise hat der Endkunde das Programm manipuliert und versehentlich Fehler eingebracht[1].

  Dies ist besonders problematisch bei aufwendigen Projekten, wie sie beispielsweise in AutoCAD auftreten. Viele Anbieter von AutoCAD-Erweiterungen erstellen ihre Programme zumindest in wesentlichen Teilen in LISP, und haben keine Möglichkeit, den Quelltext zu schützen, da diese auf dem Rechner des Endkunden liegen und dort kopiert oder geändert werden können.

[1]Wenn man nachfragt, hat natürlich keiner irgendetwas geändert. . .

Andererseits kann unter Umständen der Endanwender auch kleinere Fehler selbstständig beheben, möglicherweise im Rahmen einer *Telefonseelsorge*.

7.1.1 Überblick über verschiedene Programme

Im folgenden werden einige wichtige Typen von Programmen vorgestellt, gegebenenfalls mit typischen oder wichtigen Vertretern.

Teilweise lassen sich die einzelnen Klassen nicht genau voneinander abgrenzen; trotzdem sollen wenigstens die wichtigsten Begriffe kurz erklärt werden.

Rechte an Software In allen halbwegs fortschrittlichen Staaten existiert eine Vorstellung von *Urheberrecht*; allerdings mit bedeutenden nationalen Unterschieden.

In den meisten Ländern wird Software gleich oder sehr ähnlich literarischen Werken behandelt.

Halbwegs einheitlich ist die Grundregel, daß ein Hersteller von Software die Rechte an den Programmen in dem Sinne hat, daß die Software nur mit Einwilligung des Herstellers verwendet werden kann. Der Hersteller kann dieses Recht nutzen, indem er gegen einmalige (Softwarekauf) oder laufende Zahlungen (Softwaremiete) einem Kunden die Erlaubnis erteilt, die von ihm erstellten Programme zu verwenden oder weiter zu vertreiben.

Eine Verwendung oder eine Weitergabe, insbesondere mit kommerziellem Interesse, ohne Erlaubnis des Urhebers ist prinzipiell zivil- als auch strafrechtlich mit Vorsicht zu genießen.

Im Detail gelten aber beispielsweise zwischen den USA und Deutschland viele Unterschiede mit sehr weitreichenden Konsequenzen. Zum Beispiel ist die *EULA* („end user license agreement"), die man beim Installieren von Microsoft-Produkten mit einem Knopfdruck bestätigen muß (I accept), für einen deutschen Kunden, der im Supermarkt einen PC kauft, schlicht und ergreifend ungültig. Daß die darin enthaltenen Restriktionen (als Knebel für den Kunden) und Haftungsausschlüsse (als Bonbon für den Hersteller) ungültig sind, bedeutet aber im Gegenzug nicht, daß der Hersteller keine Rechte oder Pflichten mehr hätte. Stattdessen gelten dann halt entsprechende allgemeine Regeln, die allerdings durch die Komplexität des Rechtsgebietes und relativ neue Problemen in alten juristischen Regeln oft nicht eindeutig sind. Viele Punkte sind gesetzlich nicht klar geregelt, und Gerichtsentscheidungen oft nicht vorhersehbar.

Teilweise ganz andere Regeln für Urheberrecht und Gewährleistung gelten übrigens für Software, die nicht erst entwickelt, und dann vom Kunden möglicherweise gekauft wird, sondern stattdessen im Kundenauftrag entwickelt wird (Stichwort Werkvertrag). Das Urheberrecht ist übrigens auch dann nur sehr begrenzt anwendbar, wenn Software im Rahmen eines abhängigen Arbeitsverhältnisses entsteht.

Neben direkten Abgabepreisforderungen kann der Urheber an die Verwendung auch andere Bedingungen knüpfen; beispielsweise wird häufig die Verwendung durch nichtkommerzielle Anwender (Privatleute, Schulen, Universitäten etc.) frei zugelassen, während von kommerziellen Anwendern (Firmen) Preise gefordert werden.

Kommerzielle Software wird normalerweise nur in kompilierter Form vertrieben, und zudem mit strengen Auflagen, was die Weitergabe und die Anfertigung von Kopien betrifft. Teilweise wird auch versucht, diese vom Hersteller vorgegebenen Spielregeln zu erzwingen, indem bei der Installation bestimmte Freigabecodes eingegeben werden müssen, die bei anderer Hardware nicht funktionieren, oder indem bestimmte Hardwareerweiterungen zwingend notwendig sind (beispielsweise sogenannte *dongles*, die meistens in die parallele Schnittstelle gesteckt werden müssen, damit die Software läuft). So soll verhindert werden, daß für ein verkauftes Exemplar eines Produkts mehrere Raubkopierer damit arbeiten können.

Share ware: Viele Anbieter versuchen, Geld zu verdienen, indem sie ihre Produkte frei verteilen, aber dabei mit der Auflage (oder zumindest der Bitte) versehen, bei Gefallen einen (üblicherweise recht geringen) Obulus zu entrichten. Diese Software wird als sogenannte *share ware* vertrieben, und ist ähnlich zu betrachten wie zeitlich begrenzte Testversionen.

public domain: Andere verteilen ihre Programme zwar frei und ohne direkte kommerzielle Interessen (*public domain*), behalten jedoch den Quelltext zurück; nur die kompilierten Programme werden in Umlauf gebracht. Solche nichtkommerziellen Programme sind erst mal ganz nett, aber oft bekommt man keine neueren Programmversionen oder Fehlerbereinigungen, weil der Ersteller früher oder später das Interesse verliert. Manchmal wird das Programm auch relativ erfolgreich, und der Hersteller will dann mit neueren Versionen doch noch Geld verdienen, und verlangt dann dementsprechende Preise für die Weiterentwicklung (so beispielsweise bei *secure shell* (`ssh`) geschehen[2]).

Freie Software dagegen liegt dann vor, wenn jedermann sowohl die fertig kompilierten Programme verwenden kann, als auch die Quelltexte dafür frei verfügbar sind. Soweit der Ersteller selbst für die Verwendung der Quelltexte keine konkreten Einschränkungen vorsieht, kann jeder Programmierer auch Teile des Quelltextes für eigene Arbeiten mehr oder weniger direkt weiterverwenden.

Solche geistigen Plünderungen sind teilweise ausdrücklich erlaubt.

Freie Software hat eine recht lange Tradition, auch wenn sie sich im Moment besonders stark ausbreitet; insbesondere in Form von GNU-Linux als komplettes modernes Betriebssystem mit einer Fülle von darauf laufenden Anwendungen.

Der Begriff *GNU* ist eine rekursive Abkürzung, und steht für *„Gnu is Not Unix“*. Dies ist ein Projekt, das seit etlichen Jahren darauf abzielt, gute Software von fähigen Entwicklern erstellen zu lassen, ohne daß der Endanwender dafür bezahlen muß (http://www.gnu.org).

Das lange unerreichbare Ziel war die Entwicklung eines vollständigen Betriebssystems mit allen nützlichen Programmen. Da als technisches Vorbild weitgehend Unix diente, aber das GNU-Projekt im Gegensatz zu Unix frei von Lizenzen an Hersteller sein sollte, kam es zum Namen *„Gnu is Not Unix“*.

Die Software aus dem GNU-Projekt ist wie jedes andere geistige Eigentum urheberechtlich geschützt; die Nutzung und Weitergabe ist allerdings recht freizügig geregelt in der *GNU Public License* (GPL); siehe **A GNU GENERAL PUBLIC LICENSE** auf Seite 81.

Getragen wird das Projekt von der *Free Software Foundation* (FSF), die von Spenden lebt, sowie vom Verkauf der Software zu Preisen knapp über den Kosten der Datenträger. Solche Produkte zu kaufen kann auch schon als Spende angesehen werden, weil ja alle Programme frei weitergegeben werden kann, und damit überall im Internet ohen Einschränkungen erreichbar ist; niemand wird gezwungen, etwas von der FSF zu kaufen, wenn er die Produkte verwenden will. Auch eine kommerzielle Verwendung ist problemlos möglich. Die Einnahmen der FSF reichen damit natürlich bestenfalls, den laufenden Betrieb sicherzustellen, aber nicht um nennenswert die Entwicklung von Software zu finanzieren. So bestehen

[2] was in diesem Fall wiederum zu einer parallel entwickelten freien Version (*open ssh*) geführt hat

die einzelnen Bestandteile des GNU-Projekts weitgehend aus Entwicklungen von Universitäten, oder von Mitarbeitern kommerzieller Fimen, die sich für die eigene Arbeit irgendwelche Hilfsprogramme bauen, und der Öffentlichkeit zur Verfügung stellen.

Im Laufe der Zeit entstand so eine immense Fülle von kleineren und größeren Projekten, die in der Summe schon fast das erzielte Betriebssystem darstellten (und in der Regel unter Unix oder ähnlichen Betriebssystemen laufen konnten, aber großteils auch auf anderen Systemen wie Windows verwendbar waren). Nur der innerste Kern eines Betriebssystems, der Kernel, schien lange Zeit nicht machbar, bis 1991 der finnische Student Linus Torvalds für sich selbst einen unixähnlichen Kernel baute, um die Prozeßumschaltung auf einem Intelprozessor (80386) zu studieren. Er begann eine Diskussion darüber im Internet, und es kam dann die Idee auf, diesen Kernel noch auszubauen, und mit der bereits vorhandenen GNU-Software zu einem kompletten, freien Betriebssystem zusammen zu bauen. Daraus ist dann das heute schon überraschend ausgereifte System *GNU-Linux* entstanden.

Über weite Strecken ist freie Software nicht nur billiger, sondern auch wesentlich ausgereifter, weil einerseits die Entwicklung langsamer und ohne Zeitdruck verläuft, und andererseits viel mehr kritische Augen die Entwicklung beobachten, als dies bei Entwicklung innerhalb einer Firma der Fall ist.

Unter den Bedingungen der GPL ist inzwischen ein Compiler oder Interpreter für praktisch jede interessante Programmiersprache erhältlich (wobei besonders die Compiler für C, C++, Fortran und Pascal inzwischen sehr gut sind, Maschinencode für jeden gängigen Prozessor erzeugen können, und sogar als Crosscompiler genutzt werden können, also auf einem System Programme für ein anderes erzeugen können), einfache bis hin zu sehr komplexen Editoren, Datenbanksysteme (DBMS), alle wichtigen Hilfsprogramme zur Programmentwicklung, Zeichenprogramme, Benutzeroberflächen, und vieles mehr.

Was offensichtlich noch fehlt, ist ein vernünftiges CAD-Programm.

7.1.1.1 Spiele

Wohl die weiteste Anwendung von Computern ist die Verwendung unzähliger Spiele.

Programmiertechnisch sind diese inzwischen sehr interessant, und zudem hat sich in diesem Feld eine Wirtschaft mit enormen Umsätzen aufgebaut. Davon entfällt nur ein Teil auf übliche PCs; vielmehr wird wohl der größte Umstz mit speziellen Spielkonsolen (Nintendo, Sony Playstation etc.) und zugehörigen Programmen erzielt.

7.1.1.2 Texterstellung, Dokumentation

Wohl der größte ernsthafte Einsatzbereich von Computern liegt im Erstellen von Dokumenten im weitesten Sinne.

Das beginnt bei Texteditoren, mit denen sich unformatierte Dateien (freie Texte, Programmquelltexte, Konfigurationsdateien. . .) erstellen lassen, geht über wesentlich aufwendigere Textverarbeitungen zum Erstellen von Texten mit Formatierungen (Fettschrift, unterstrichen, Schrägschrift, verschiedene Schriftgrößen, Einfügen von Bildern, Querverweise etc.).

Zu den einfachen **Texteditoren** gehören erstmal als Grundausstattung jeweils die, die ein Hersteller seinem Betriebssystem beilegt. Das ist bei Unixrechnern traditionell der `vi`. Neben dem „Hauseditor“ `xedit`[3] des IBM-Betriebssystems VM/CMS ist der `vi` eines der Programme mit den erbittertsten Feinden und den glühendsten Anhängern. Anfänger tun sich immer schwer damit; mit der Zeit wird es bei einigen Benutzern besser. Er hat den Vorteil daß er nach wie vor auf jedem Unixrechner in jedem Fall verfügbar ist; alleine deshalb sollte man die grundlegende Bedienung des `vi` beherrschen. Mit entsprechender Übung kann man damit recht gut arbeiten, aber es ist halt alles Geschmackssache[4]. Nachdem Windows ohnehin zu den Glanzleistungen des 20. Jahrhunderts gehört, ist der Standardeditor `notepad` eine qualitativ angemessene Beigabe. Man kann ihm immerhin bescheinigen, daß man Texte aufrufen, ändern und abspeichern kann, solange sie nicht größer als 64 kB sind, keine Sonderzeichen vorkommen mit denen bei Microsoft niemand gerechnet hat und die Umstände auch sonst recht günstig sind. Zum Anzeigen einer `readme.txt` reicht er, wenn sie nicht zu groß ist. Sein größer Vorteil ist, daß man ihn nicht nehmen muß, es gibt auch andere.

Neben dieser Minimalausstattung gibt es für jedes Betriebssystem auch recht brauchbare Editoren, mit denen auch Vielschreiber unter den Anwendern glücklich werden. Welcher im konkreten Einzelfalll der beste ist, hängt in erster Linie davon ab, welchen Experten man fragt. Mein erklärter Liebling ist der EMACS.

[3] der Editor `xedit` von VM/CMS ist wohl sicher der schlimmste Editor, den ich je benutzt habe. Wenn man nach Stunden noch nicht im mindestens damit klar kommt, schreibt man seine Texte lieber auf einem anderen Rechner, und überträgt sie mit `ftp`.

[4] zum `vi` und seinen Anhängern fällt mir immer spontan ein: *meine Katze mag die Mäuse roh; ich mag sie nicht mal gekocht!*

Damit kann man (entsprechende Übung vorausgesetzt) sehr schnell und effektiv schreiben, aber nebenbei auch emails schreiben und lesen, Dateien und Verzeichnisse löschen/umbenennen/erzeugen/verschieben, selbst das Verhalten des Editors insgesamt oder für bestimmte Dateitypen frei definieren, und außer Kaffee kochen eigentlich alles machen. Es gibt Schnittstellen zu anderen Programmen, um beispielsweise direkt vom Editor aus Programme zu kompilieren, aufzurufen oder zu debuggen. Das ganze auch noch für alle Betriebssysteme und für alle Textarten. Durch den Umstand, daß man nur einen Editor erlernen muß und damit alle Arten von Dateien auf allen möglichen Systemen ohne zusätzlichen Lernaufwand editieren kann, hat man viel Nutzen für wenig Aufwand.

In sogenannten *integrierten Entwicklungsumgebungen* (siehe **7.1.1.4 Programmentwicklung** auf Seite 37) ist entweder ein Editor zwangsweise enthalten, oder man hat glücklicherweise die Möglichkeit, einen Editor nach eigenen Wünschen einzubinden.

Eine **Textverarbeitung** zielt dagegen nicht auf reinen Text ab, sondern soll die Erstellung von Dokumenten mit Formatierungen unterstützen. Der wohl bekannteste Vertreter dieses Genre ist Word von Microsoft; von etlichen anderen Herstellern gibt es für Windows ebesno wie für andere Betriebssysteme ähnlcihe Systeme. Sie verfolgen ein Konzept, das vor ein paar Jahre als WYSIWYG („what you see is what you get“) als Modewort populär war. Dahinter steckt die Idee, gleich bei der Eingabe den Text genau so zu sehen, wie er später auf dem Papier ausgedruckt erscheinen soll, also wenn ich beispielsweise eine ander Schrift wähle, dann sehe ich die neu eingegegebenen Zeichen gleich in der geänderten Schrift. Das ist ein sehr lobenswertes Ziel, und erleichtert besonders Anfängern den Einstieg in die Arbeit. Man wird heute kaum noch eine Sekretärin finden, der man etwas anderes zumuten kann. Für kurze Texte im Bereich von maximal wenigen Dutzend Seiten und nicht zu vielen Querverweisen, Stichworteinträgen und ähnlichem sind diese Programme durchweg gut geeignet und sinnvoll zu verwenden. Anders sieht es aus, wenn ernsthaft größere Dokumente erstellt werden. Dann kommen Programme dieser Art auch auf schnellen Rechnern mit viel Speicher schnell an ihre Grenzen.

Da für Dokumentationen neben dem reinen Text auch Grafiken benötigt werden, versuchen die meisten Anbieter neben einer Einbindung vorhandener Bilder auch eine Lösung zur Erstellung und Manipulation von Grafiken in die Textverarbeitung zu intergrieren. Dadurch verschwwimmen auch hier die Grenzen; im Bereich *DTP* (desktop publishing) wird Text weitgehend als Grafik betrachtet.

Einen ganz anderen Ansatz verfolgt das **Textsatzsystem** TeX (gesprochen: „tech“). Dieses Programm wurde von Donald E. Knuth geschaffen, weil der einige Bücher ([Lit. C: Knuth: Fundamental Algorithms] und folgende) schreiben wollte, und keine vernünftigen Programme fand, um sich die Arbeit zu erleichtern. Gedacht war das Programm TEX hauptsächlich, um technische Dokumente zu schreiben; der Schwerpunkt lag anfangs auf der Darstellung mathematischer Formeln. Der Umgang mit TEX direkt ist eher schwierig und umständlich (Knuth ist Mathematiker). Aber das Programm ist sehr flexibel und ermöglicht dank einer eingebauten Programmiersprache die Verwendung von Erweiterungen. So entstanden im Laufe der Jahre etliche Aufsätze auf TEX, um die Arbeit damit angenehmer zu gestalten. Die bekannteste und inzwischen mit Abstand am meisten eingesetzte ist LATEX von Leslie Lamport ([Lit. C: Lamport: LATEX]). Damit ist auch dieses Dokument geschrieben. Die Arbeitsweise mit einem solchen System ist vollkommen entgegengesetzt zu dem WYSIWYG-Ansatz: man schreibt mit einem ganz normalen Texteditor eine Datei, die aus dem gewünschten Text und eingestreuten Direktiven besteht, mit denen TEX oder LATEX zu bestimmten Aktionen veranlaßt werden. Während man diesen Text mit den Direktiven schreibt, sieht man eben noch nicht die Auswirkung der Direktiven. Erst nach dem Abspeichern des Textes läßt man das Programm TEX laufen. TEX erstellt dabei aufgrund der Eingabedatei die gesamte Formatierung des Textes (welcher Buchstabe aus welchen Zeichensatz kommt an welche Stelle der Seite? Wo werden Zeilen umgebrochen? Wo werden Seiten umgebrochen?), numeriert Kapitel, Abschnitte, Unterabschnitte, kann auf Wunsch Inhaltsverzeichnis oder Stichwortverzeichnis erstellen, und so weiter. Nachdem TEX beendet ist, kann man sich das Ergebnis betrachten, und bei Bedarf die Eingabedatei ändern. Die Änderungen sieht man aber natürlich erst nach einem erneuten Lauf von TEX. Diese Arbeitsweise ist der Programmentwicklung sehr ähnlich (Quelltext schreiben, kompilieren lassen, ausprobieren, Quelltext ändern. . .). Deshalb wird TEX beziehungsweise LATEX in erster Linie von technisch orientierten Menschen verwendet (Programmierer, Ingenieure, Mathematiker, Physiker. . .).

Auf den ersten Blick ist eine solche Arbeitsweise recht umständlich, aber der Verzicht auf sofortige optische Kontrolle hat in der Praxis drastische Vorteile:

- nach einer Eingewöhnung ist das Arbeiten sehr effektiv, weil die Konzentration nicht auf das Aussehen eines Dokuments gerichtet ist, sondern mehr auf den Inhalt.
- Die optische Qualität der erzeugten Dokumente ist sehr gut, ohne daß der Beutzer sich um die Formatierung im Detail kümmern muß. Man greift in die Formatierung nur dann manuell ein, wenn man glaubt es besser zu wissen als LATEX. (Meistens hat allerdings LATEX in Geschmacksfragen recht.)
- Da die Formatierung im Detail fast ausschließlich automatisch erfolgt, geraten selbst längere Texte optisch einheitlich. Auch wenn die Arbeit auf

mehrere Ersteller verteilt wird, kann man leicht ein einheitliches Bild erreichen.

- Da die Eingabedatei ein reiner Text mit einem klar definierten Format ist, kann man die Eingabedatei mit fertigen oder selbsterstellten Programmen automatisch weiterverarbeiten. Vieles gibt es fertig, was es nicht gibt kann man sich bauen: man kennt ja den Eingabetext, den man selbst schreibt.

  Dieses Dokument wird beispielsweise in LaTeX erstellt, und daraus wird mit TeX eine druckbare Ausgabe erzeugt, ebenso wie mit einem frei erhältlichen System LaTeX2HTML eine HTML-Version, die von ALLEN Browsern vernünftig dargestellt werden kann.

- Die Anbindung an automatische Dokumentationssysteme ist nahtlos möglich, beispielsweise um Klassendefinitionen und weitere Informationen maschinell aus Quelltexten zu extrahieren, und wieder zu einer TeX-Eingabe zu machen.

- Es gibt keine ernsthaften Beschränkungen hinsichtlich Umfang und Komplexität von Dokumenten. Bereits auf einem Atari ST (8 MHz, 1 MB Speicher) vor 15 Jahren waren Dokumente mit hunderten Seiten und Tausenden Querverweisen problemlos zu erstellen. Die so erstellten Dateien waren selbstredend mit den selben Programmen auf dem damaligen Großrechner des Rechenzentrums ebenso austauschbar wie mit allen heutigen TeX-Systemen.

- Die gesamte Erstellung von Dokumenten kann mit freier Software erfolgen, die seit Jahrzehnten kontinuierlich gepflegt wird. Probleme der Art *„ich habe vor drei Jahren eine Datei mit Word 6.0 angespeichert, meine Textverarbeitung kann sie nicht mehr öffnen. Was kann ich tun?“* gibt es einfach nicht. Alle Dokumente sind über alle Betriebssystemgrenzen hinweg frei austauschbar, und auch nach Jahren noch verwendbar; Änderungen werden dann eingeführt, wenn sie nötig und sinnnvoll sind und nichts bewährtes eliminieren; bei kommerziellen Systemen dagegen werden Änderungen nach dem Kalender verabschiedet, und durch ständig neue und undokumentierte Dateiformate wird versucht, die Benutzer möglichst schnell auf neue Versionen zu zwingen.

- Alle verwendeten Programme sind sehr ausgereift und stabil. Das typische Gejammer *„Meine Diplomarbeit ist fast fertig, aber ich kann sie nicht mehr öffnen“* oder *„seit gestern landen die Fußnoten auf den falschen Seiten, und morgen muß ich abgeben“* stammt nicht von LaTeX-Nutzern.

Aber halt auch wieder Geschmackssache. Jeder ist für sein Unglück selbst verantwortlich. Aber es soll nachher keiner sagen, er wäre nicht gewarnt worden!

7.1.1.3 Grafik

vektororientiert/pixelorientiert

CAD

Visualsierung

Photoshp, Paintshop, gimp

Multimedia

7.1.1.4 Programmentwicklung

Zur Entwicklung eigener Programme (siehe auch **7.1.2 Programmerstellung** auf Seite 38) existiert eine riesige Vielfalt großer und kleiner Programme.

Compiler, Linker, make, integrierte Entwicklungsumgebungen, Versionsverwaltung RCS (revision control system), CVS (concurrent versions system)

7.1.1.5 Systemprogramme

Datensicherung

7.1.1.6 DBMS

SQL

7.1.1.7 Numerik und Simulation

FE, Simulation, Differentialgleichungen

7.1.2 Programmerstellung

Wenn man ein Programm in einer interpretierten Sprache erstellen will, dann ist prinzipielle Vorgehensweise recht einfach: Mit einem Editor (siehe **7.1.1.2 Texterstellung, Dokumentation** auf Seite 33) wird der Quelltext erstellt. Mithilfe des Interpreters läßt man das erstellte Programm ausführen.

Deutlich umständlicher erscheint zumindest anfangs die Erstellung von Programmen, die letztlich in Maschinencode vorliegen und damit direkt ausführbar sein sollen:

1. Ebenfalls mit einem Editor wird ein Quelltext erstellt (beziehungsweise bei größeren Projekten mehrere Quelltexte).

2. Eventuell wird der Quelltext von einem Präprozessor in eine andere Datei umgesetzt, die formal wieder ein Quelltext für einen Compiler darstellt. Bei C und C++ ist dies generell der Fall. Unter Umständen wird dies aber auch bei Programmen in anderen Sprachen gemacht (und bei C/C++ gegebenenfalls zusätzlich von einem weiteren Präprozessor), beispielsweise um in Fortran eine Anbindung an SQL-Datenbanken zu vereinfachen.

3. Der Quelltext beziehungsweise die Ausgabe des Präprozessors wird von einem Compiler in Maschinensprache und ergänzende Informationen umgesetzt.

   Bei manchen Systemen wird die Maschinensprache nicht wirklich als Binärcode erzeugt, sondern als lesbarer Text in Form der jeweiligen Assemblersprache.

   Ansonsten heißen die so aus dem Quelltext entstandenen Dateien *Objektdateien*, falls der Compiler tatsächlich bereits die binäre Darstellung des Programms erzeugt.

   Solche Objektdateien sind auf den meisten Systemen noch nicht ausführbar, sondern enthalten nur die Teile eines Programms, die konkret in diesem Projekt entwickelt wurden und im Quelltext abgelegt sind. Normalerweise werden aber noch weitere Programmteile benötigt, die auch von anderen Programmen genutzt werden können (Funktionen zur Ein- und Ausgabe, allgemeine mathematische Operationen und ähnliches). Diese noch fehlenden Teile liegen ebenfalls als Objektdateien in sogenannten *Bibliotheken* oder *libraries* vor.

4. Wenn der Compiler Assemblercode erzeugt hat, dann muß dieser im nächsten Schritt von einem Assembler in eine Objektdatei übersetzt („assembliert“) werden.

5. Die vom Compiler oder vom Assembler erzeugten Objektdateien und die benötigten Bibliotheken werden vom *Binder* oder *linker* zu einem ausführbaren Programm zusammengeschnürt, meistens in Form einer Datei.

   Derartige Bibliotheken wurden früher generell komplett oder teilweise (soweit vom jeweiligen Programm benötigt) zum ausführbaren Programm gebunden. Das hat natürlich den Nachteil, daß sowohl die ausführbare Datei auf dem Massenspeicher (Festplatte oder ähnliches) als auch nach dem Starten im Speicher Programmcode enthält, der vollkommen identisch in anderen Programmen enthalten ist. Um dieser Verschwendung Einhalt zu gebieten, wurde auf den modernen Systemen das Konzept von *dynamischen Bibliotheken* entwickelt. Dabei wird nicht mehr der komplette Programmcode aus den Bibliotheken zu einem ausführbaren Programm hinzugefügt, sondern stattdessen nur noch ein Verweis auf die zugehörige Bibliothek. In diesem Fall spricht man von *dynamischen Binden* (dynamic linking). Unter Unix heißen solche dynamischen Bibliotheken *shared objects* (oft mit der Endung `.so` im Dateinamen); unter Windows *dynamic link libraries* (meist endet der Dateiname auf `.dll`).

6. Um ein solches Programm auszuführen, wird ein bestimmter Teil des Betriebssystems aktiv, der sogenannte *Lader* oder *loader* aktiv.

   Dieser lädt die Datei mit dem ausführbaren Programm in den Arbeitsspeicher, und führt dann das Relozieren (relocation) durch. Dies ist nötig, weil Compiler und Linker noch nicht wissen können, wo das Programm später bei seiner Ausführung im Speicher liegen wird. Deshalb nehmen sie jeweils für den Bereich (Segment), in dem der Maschinencode und die verschiedenen Datenbereiche untergebracht sind, irgendwelche Adressen an (beispielsweise die Adresse 0), und vermerken in den Objekt- beziehungsweise ausführbaren Dateien, an welchen Stellen solche vermuteten Adressen verwendet wurden. Beim Relozieren muß der Lader nun alle solchen vermuteten um die tatsächliche Anfangsadresse korrigieren.

   Gegebenenfalls muß außerdem im Falle von dynamischen Bibliotheken vom Lader dafür gesorgt werden, daß die entsprechenden Bibliotheken geladen werden (falls sie sich nicht schon durch ein anderes gestartetes Programm im Speicher befinden).

   Anmerkung: Dynamische Bibliotheken müssen nicht zwangsläufig beim Linken und Laden berücksichtigt werden, sondern es ist je nach Betriebssystem auch möglich, erst zur Laufzeit dynamische Bibliotheken nachzuladen. Dies kann genutzt werden, um vorhandene Programme, zu denen dem Endanwender kein Quelltext zur Verfügung steht, erweiterbar zu machen. Nach diesem Prinzip funktionieren die Plugins verschiedener Internetbrowser und Zeichenprogramme ebenso wie die Laufzeiterweiterungen von AutoCAD (ARX

> = AutoCAD run time extension*) oder die meisten COM-Objekte alias* ActiveX controls *unter Windows.*

Entweder innerhalb des Compilers oder als eigenes Hilfsprogramm kann als weiterer Zwischenschritt noch ein sogenannter *Optimierer* (*optimizer*) laufen. Dieser versucht, den Assembler- oder Maschinencode so umzuformen, daß er entweder schneller ausgeführt wird, oder kompakter wird (am besten natürlich beides). Gerade maschinell erzeugter Code enthält oft überflüssige Anweisungen, oder ungünstig formulierte Programmschritte. Beispielsweise muß für ein nur kurzfristig benötigtes Zwischenergebnis manchmal keine Variable angelegt werden, sondern es kann direkt in einem Register gehalten werden. Die Verwendung der Variable kann also einfach wegoptimiert werden. Ein anderes Beispiel für eine mögliche Optimierung ist die Berechnung von invarianten Ausdrücken (also Ausdrücke, deren Wert sich nicht ändert) nur einmalig vor Schleifen, anstatt mehrfach innerhalb von Schleifen.

Debuggen ist ein weiterer Schritt auf dem steinigen Weg zum Superprogramm für den zufriedenen Kunden.

Erfahrungsgemäß gelingt es selten, ein Programm mit mehr als 10 Zeilen Quelltext auf Anhieb fehlerfrei zu erstellen.

Wenn man die Fehler beseitigt hat, die bereits vom Compiler oder vom Linker erkannt werden können, sind meist immer noch Fehler vorhanden, die erst zur Laufzeit des Programms in der Testphase auftreten. Solche Fehler werden teilweise am Verhalten des Programms erkannt, und sind schnell behoben.

Weniger offensichtliche Fehler (falsche Rechenergebnisse, unvermittelte Programmabstürze) sind oft nur schwer zu lokalisieren.

In diesem Stadium benötigt man einen *Debugger*. Damit kann man ein Programm während seiner Ausführung kontrollieren, auf Wunsch in Einzelschritten ausführen lassen oder in Abhängigkeit von Variablenwerten oder an bestimmten Stellen des Quelltextes anhalten lassen. Während ein Programm angehalten ist, kann man in Ruhe die Werte von Variablen betrachten, oder sogar testweise manipulieren.

Wer mit seinem Debugger gut umgehen kann, hat ein wesentlich leichteres Leben. Noch leichter tut man sich natürlich, wenn man die Fehler bereits im Vorfeld vermeidet. Aber das ist ein anderes Thema, und sprengt den Rahmen dieses Skripts deutlich.

7.2 Prozeß

Ein Prozeß ist ein Programm, welches gerade auf einem konkreten System ausgeführt wird. Folgende Eigenschaften charakterisieren einen Prozeß zu einem bestimmten Zeitpunkt (gegebenenfalls je nach Betriebssystem):

- eine eindeutige Kennung, die vom Betriebssystem für jeden momentan laufenden Prozeß vergeben wird; dies ist in den meisten Fällen einfach eine fortlaufende Zahl (*process id*, `pid`)
- Die Argumente an ein Programm (die meisten Betriebssysteme lassen es zu, einem Programm bei seinem Start einige Zeichenfolgen als Argumente mitzugeben, beispielsweise kann einem Compiler mit solchen Argumenten mitgeteilt werden, welche Quelltexte er zu übersetzen hat).
- das *environment* ermöglicht bei den meisten Betriebssystemen zusätzlich zu den Argumenten weitere Informationen zu hinterlegen, die ein Prozeß nutzen kann. Dies ist eine Sammlung von sogenannten Umgebungsvariablen, mit denen jeweils einem Namen eine Zeichenkette (String) als Wert zugeordnet wird.

  Die bekannteste Variable dieser Art hat den Namen `PATH`. Ihr Wert gibt unter Windows (in allen seinen Variationen) ebenso wie unter allen Unixversionen an, in welchen Verzeichnissen im Dateisystem nach ausführbaren Programmen gesucht wird.
- die auszuführende Befehlsfolge (in Maschinensprache) ab einer bestimmten Startadresse im Speicher
- gegebenenfalls Speicherbereiche für initialisierte Daten (oft *block storage segement* (BSS, `.bss`) genannt) und für Daten, die bei Programmstart nicht initialisiert werden müssen (oft *data segment*, `.data` genannt)
- ein Programmstack, der für sogenannte automatische Variablen und für Unterprogrammaufrufe verwendet wird (siehe **7.2.2 Der Programmstack** auf Seite 44).
- der aktuelle Zustand des Prozessors (Inhalt der Daten- und Adreßregister, Stackpointer (siehe **7.2.2 Der Programmstack** auf Seite 44), Statusregister, program counter PC; siehe dazu auch **5.1 Der Prozessor** auf Seite 19)

7.2.1 Ein Prozeß im Speicher

Beim Starten eines Prozesses wird von einem Teil des Betriebssystems, dem Lader, eine ausführbare Programmdatei gelesen, und großteils in einen freien Speicherbereich kopiert. Bei einfachen Betriebssystemen wie CP/M oder MS-DOS und Windows gibt es einen Speicherbereich, in dem alle gerade laufenden Prozesse zu finden sind; dann können die einzelnen Prozesse versehentlich oder absichtlich sich gegenseitig oder womöglich das Betriebssystem beeinflussen. Unter Betriebssystemen mit einer virtuellen Speicherverwaltung (DEC-VMS, Unix, Windows NT) sieht jeder Prozeß nur einen ihm zugewiesenen Speicher, und das Betriebssystem kann darauf achten, daß nicht illegalerweise anderer Speicher verwendet wird als der vom Betriebssystem für diesen Prozeß vorgesehene (sie dazu auch **6 Betriebssysteme** auf Seite 21).

Die zu kopierenden Bereiche sind üblicherweise in verschiedene *Segmente*[5] unterteilt. Dies kommt daher, daß der Compiler, Assembler und Linker (siehe **7.1.2 Programmerstellung** auf Seite 38) ihre jeweiligen Eingabedateien sequentiell durchlaufen, und dabei aber verschiedenartige Informationen sammeln: ausführbarer Maschinencode fällt vermischt mit anzulegenden Variablen oder Konstanten an. Deshalb werden jeweils vom Typ her ähnliche Informationen in Segmenten gesammelt: eines für Maschinencode, eines für Konstanten et cetera.

So brauchen der Compiler und der Assembler nur bei jedem Teil ihrer Ausgabe vermerken, zu welchem Segment es zugeschlagen werden soll. Erst der Linker sortiert die zusammengehörigen Teilstücke dann in den Segmenten ein. Ein Beispiel dafür ist die Anweisung `.text` im Assemblerlisting in **7.2.2.1 Vom C-Programm zum Prozeß** auf Seite 45. Sie weist den Linker an, alles danach folgende (bis zu einer anderen Segmentdirektive, `.data` beispielsweise) in das `.text`-Segment zu packen. Weil in diesem Beispielprogramm keine globalen und konstanten Daten verwendet werden, und alles im `.text`-Segment landet, ist dies die einzige Segmentanweisung.

Es gibt praktisch immer ein Segment, in dem Maschinencode zu finden ist. Dieses heißt manchmal `TEXT`, `.text`, `.CODE` oder so ähnlich.

Ein weiteres Segment besteht aus den globalen Variablen eines Programms, die

[5] Diese Verwendung des Begriffs *Segment* darf nicht verwechselt werden mit der Segmentierung unter MS-DOS und 16bit-Windows (also Windows bis 3.11), sowie teilweise immer noch unter Windows 95 und 98. Dabei geht es um die Adressierung von mehr als 64 kB Speicher mit nur 16 bit langen Adressen. Außer dieser Warnung wird diese Art der Segmentierung in diesem Dokument nicht mehr weiter erwähnt; sie stirbt hoffentlich noch schneller aus als der Rest von Windows.

bei Programmstart initialisiert sein müssen. Dieses Segment kann `DATA`, `.data` oder so ähnlich heißen.

Globale Variablen, die keinen bestimmten Anfangswert bei Programmstart benötigen, brauchen auch nicht in voller Länge in der Programmdatei abgelegt zu werden; es reicht wenn die Länge des dafür nötigen Bereichs in der Datei vermerkt ist. Ein solches Segment heißt *block storage segment*, `BSS`, oder `.bss`. Erst beim Erzeugen eines Prozesses muß dieser Bereich im Speicher angelegt werden.

Je nach Fähigkeiten des Betriebssystems kann es auch noch feinere Unterteilungen geben; beispielsweise könnten für globale Daten zwei verschiedene Segmente verwendet werden: eines für solche, die während der Laufzeit konstant sind, und eines für die restlichen, die sich ändern dürfen. In diesem Fall kann das Betriebssystem einen versehentlichen schreibenden Zugriff auf Konstanten unterbinden[6].

Anmerkung: lokale, automatische Vairbalen werden nicht in einem besonderen Segment im ausführbaren Programm abgelegt, sondern entstehen erst zur Laufzeit auf dem Stack; siehe **7.2.2 Der Programmstack** *auf Seite 44.*

Neben den Bereichen der Programmdatei, die zum Start in den Speicher kopiert werden müssen, enthält die Programmdatei je nach Betriebssystem noch Informationen für den Lader, beispielsweise wie groß die einzelnen Segmente sind, wieviel Stackspeicher (siehe **7.2.2 Der Programmstack** auf Seite 44) benötigt wird, gegebenenfalls Informationen über besondere Zugriffsrechte, oder ähnliches.

TODO:

Verwaltung freier Speicher manuell (`malloc()`/`free()` beziehungsweise `new`/`delete`) oder automatisch (Java, LISP, *garbage collection*)

statischer Speicher und globale Variablen, Konstanten, lokale/automatische/globale/statische Variablen, Adressen, Zeiger

[6]Wenn man unter Linux versucht, eine Stringkonstante zur Laufzeit zu ändern, erhält man einen „Speicherzugriffsfehler“. Dies liegt daran, daß Konstanten unter Linux in einem Segment `.rodata` (`ro` wie *read only*); normale globale Variablen dagegen in einem Segment `.data`. Der Inhalt des `.rodata`-Segments wird während des Programmlaufs überwacht und kann tatsächlich nicht verändert werden.

7.2.2 Der Programmstack

Stackpointer (Motorola: SP alias A7; Intel: ESP), frame pointer (Motorola: beliebig, beispielsweise A6; Intel: EBP)

Soviel abgelegt wird, soviel muß auch wieder aufgeräumt werden.

einen Wert ablegen: den Stackpointer um die Byteanzahl des zu abzulegenden Werts erniedrigen, dann den Wert an die Adresse kopieren, die im Stackpointer steht

aufräumen: den Stackpointer um die Anzahl der abgelegten Bytes erhöhen

begrenzte Größe: Schachtelungstiefe von Unterprogrammen, Rekursion

Nach Ende eines Unterprogrammaufrufs wird der freigegebene Bereich für die nächsten Aufrufe genutzt.

Was kommt im Stack vor?

- Je nach System alles, was der Aufrufer eines Unterprogramms vorher in Sicherheit bringen möchte, und ebenfalls alles, was das aufgerufene Unterprogramm retten möchte. Das ist zumindest die Rücksprungadresse, an der nach dem Ende des Unterprogramms das aufrufende Programm fortgesetzt werden muß.

- Argumente, die beim Aufruf eines Unterprogramms an dieses übergeben werden sollen.

  Bei C gilt die Regel, daß erst die im Quelltext rechts stehenden Werte abgelegt werden, dann erst die weiter links stehenden. Zudem muß in C/C++ der Aufrufer einer Funktion auf dem Stack die Werte aufräumen, die er abgelegt hat. Dadurch kann eine Funktion mit mehr Argumenten aufgerufen werden, als die Funktion erwartet; zudem kann problemlos ein Aufruf von Funktionen mit einer variablen Anzahl und Größe realisiert werden (beispielsweise `printf()`). In Prospero-Fortran und Microsoft-Fortran dagegen kann das aufgerufene Unterprogramm sich darauf verlassen, daß es mit der richtigen Anzahl an Argumenten aufgerufen wurde, und kann deshalb selbst den Stack aufräumen. Das geht unter Umstämden geringfügig schneler, weil es ohnehin den Stackpointer erhöhen muß; die weitere Erhöhung kostet keine zusätzliche Zeit.

- alle lokalen Variablen eines jeden Unterprogramms, die erst beim Aufruf des

Unterprogramms erzeugt werden (in C und C++ sind das die sogenannten automatischen Variablen, also alle lokalen Variablen, die nicht als `static` oder `const` deklariert sind)

In C/C++ wird beim Start einer Funktion der Platz für alle von der Funktion benötigten automatischen Variablen geschaffen, indem der Stackpointer um die entsprechende Anzahl Bytes herabgesetzt wird.

- Zwischenergebnisse von komplizierten Ausdrücken

Also legt der Aufrufer eines Unterprogramms eventuell Registerinhalte oder ähnliches ab, soweit er fürchten muß, daß der Aufruf diese zerstört, dann zumindest die Argumente und die Rücksprungadresse.

Das aufgerufene Unterprogramm rettet als erstes meist den Wert des Registers, das als frame pointer verwendet wird auf den Stack, anschließend den dann aktuellen Stackpointer in das frame pointer-Register. Während des Abarbeitens dieses Unterprogramms wird der frame pointer nicht mehr verändert und dazu verwendet, um relativ zu der darin enthaltenen Adresse auf die übergebenen Argumente und die eigenen automatischen Variablen zuzugreifen. Dadurch daß dafür nicht mehr der Stackpointer, sondern der frame pointer verwendet wird, steht es dem Unterprogramm frei, den Stackpointer zu manipulieren: zum Aufruf weiterer Unterprogramme müssen ja Werte auf dem Stack abgelegt werden, was den Stackpointer wieder verändert.

Wenn ein Unterprogramm seine Arbeit erledigt hat, kann es umgekehrt den Wert des frame pointers in den Stackpointer zurückkopieren. Anschließend kann über den Stackpointer die Rücksprungadresse in den program counter geladen werden, und der Programmlauf liegt wieder beim Aufrufer.

Der Aufrufer räumt (zumindest in C-/C++-Programmen) die Argumente vom Stack, indem er den Stackpointer um soviele Bytes erhüht, wie er als Argumente abgelegt hat.

Ein vollständiges kleines Beispiel ist im nächsten Abschnitt durchexerziert.

7.2.2.1 Vom C-Programm zum Prozeß

```
/* addbetr.c 31.12.2000 kw
 * Kleines Beispiel
 */
```

```
int betrag( int wert )
{
  if( wert<0 )
    {
      return -wert;
    }
  else
    {
      return wert;
    }
}

int addierebetraege( int summand1, int summand2 )
{
  int summe;

  summe = betrag( summand1 ) + betrag( summand2 );

  return summe;
}

int main( int nargs, char **args )
{
  int i, j, erg;

  i = 12;
  j = 13;

  erg = addierebetraege( i, j );

  return 0;
}
```

```
                    .text


                    ;   1:addbetr.c     **** /* addbetr.c 31.12.2000 kw
                    ;   2:addbetr.c     ****  * Kleines Beispiel
                    ;   3:addbetr.c     ****  */
                    ;   4:addbetr.c     ****
                    ;   5:addbetr.c     **** int betrag( int wert )
                    .globl betrag
```

```
                                .type    betrag,@function


                        ;   6:addbetr.c     **** {
                        betrag:
0000 55                         pushl %ebp
0001 89E5                       movl %esp,%ebp


                        ;   7:addbetr.c     ****   if( wert<0 )
0003 837D0800                   cmpl $0,8(%ebp)
0007 7D0B                       jge .L3


                        ;   8:addbetr.c     ****     {
                        ;   9:addbetr.c     ****       return -wert;
0009 8B5508                     movl 8(%ebp),%edx
000c F7DA                       negl %edx
000e 89D0                       movl %edx,%eax
0010 EB0E                       jmp .L2


                        ;  10:addbetr.c     ****     }
0012 EB0C                       jmp .L4
                        .L3:


                        ;  11:addbetr.c     ****   else
                        ;  12:addbetr.c     ****     {
                        ;  13:addbetr.c     ****       return wert;
0014 8B5508                     movl 8(%ebp),%edx
0017 89D0                       movl %edx,%eax
0019 EB05                       jmp .L2
001b 908D7426                   .p2align 4,,7
     00
                        .L4:


                        ;  14:addbetr.c     ****     }
                        ;  15:addbetr.c     **** }
                        .L2:
0020 89EC                       movl %ebp,%esp
0022 5D                         popl %ebp
```

```
0023 C3                 ret
0024 8DB60000           .align 16
     00008DBF
     00000000


               ;  16:addbetr.c     ****
               ;  17:addbetr.c     **** int addierebetraege( int summand1, int summand2 )
               ;  18:addbetr.c     **** {
               .globl addierebetraege
                       .type    addierebetraege,@function
               addierebetraege:
0030 55                 pushl %ebp
0031 89E5               movl %esp,%ebp
0033 83EC14             subl $20,%esp
0036 53                 pushl %ebx


               ;  19:addbetr.c     ****   int summe;
               ;  20:addbetr.c     ****
               ;  21:addbetr.c     ****   summe = betrag( summand1 ) + betrag( summand2 );
0037 83C4F4             addl $-12,%esp
003a 8B4508             movl 8(%ebp),%eax
003d 50                 pushl %eax
003e E8FCFFFF           call betrag
0043 83C410             addl $16,%esp
0046 89C3               movl %eax,%ebx
0048 83C4F4             addl $-12,%esp
004b 8B450C             movl 12(%ebp),%eax
004e 50                 pushl %eax
004f E8FCFFFF           call betrag
     FF
0054 83C410             addl $16,%esp
0057 89C0               movl %eax,%eax
0059 8D1418             leal (%eax,%ebx),%edx
005c 8955FC             movl %edx,-4(%ebp)


               ;  22:addbetr.c     ****
               ;  23:addbetr.c     ****   return summe;
005f 8B55FC             movl -4(%ebp),%edx
0062 89D0               movl %edx,%eax
0064 EB00               jmp .L5
```

```
                    ;  24:addbetr.c     **** }
                    .L5:
0066 8B5DE8                 movl -24(%ebp),%ebx
0069 89EC                   movl %ebp,%esp
006b 5D                     popl %ebp
006c C3                     ret
006d 8D7600                 .align 16

                    ;  25:addbetr.c     ****
                    ;  26:addbetr.c     **** int main( int nargs, char **args )
                    ;  27:addbetr.c     **** {
                    .globl main
                            .type    main,@function
                    main:
0070 55                     pushl %ebp
0071 89E5                   movl %esp,%ebp
0073 83EC18                 subl $24,%esp

                    ;  28:addbetr.c     ****   int i, j, erg;
                    ;  29:addbetr.c     ****
                    ;  30:addbetr.c     ****   i = 12;
0076 C745FC0C               movl $12,-4(%ebp)
     000000

                    ;  31:addbetr.c     ****   j = 13;
007d C745F80D               movl $13,-8(%ebp)
     000000

                    ;  32:addbetr.c     ****
                    ;  33:addbetr.c     ****   erg = addierebetraege( i, j );
0084 83C4F8                 addl $-8,%esp
0087 8B45F8                 movl -8(%ebp),%eax
008a 50                     pushl %eax
008b 8B45FC                 movl -4(%ebp),%eax
008e 50                     pushl %eax
008f E8FCFFFF               call addierebetraege
     FF
```

```
0094 83C410               addl $16,%esp
0097 89C0                 movl %eax,%eax
0099 8945F4               movl %eax,-12(%ebp)


                  ;  34:addbetr.c     ****
                  ;  35:addbetr.c     ****   return 0;
009c 31C0                 xorl %eax,%eax
009e EB00                 jmp .L6


                  ;  36:addbetr.c     **** }
                          .p2align 4,,7
                  .L6:
00a0 89EC                 movl %ebp,%esp
00a2 5D                   popl %ebp
00a3 C3                   ret
```

Ein Intel-Prozessor führt beim Befehl `call` folgendes aus:

1. Der aktuelle Wert des program counters (von Intel EPI genannt) wird auf dem Stack abgelegt. Der abgelegte Wert ist zu diesem Zeitpunkt die Adresse des Befehls nach dem `call`, also die Adresse zu der nach Ende des Unterprogramms zurückgesprungen werden muß, um die Befehle nach dem `call` auszuführen.

2. Es wird zu der Adresse verzweigt, die als Operand angegeben ist. Im Falle des hier verwendeten *kurzen Sprungs* mit OP-code 0xEP ist dies ein 1 Byte großer Wert, der die Differenz zwischen der anzuspringenden Adresse und dem Wert des program counters angibt. So sind relative Sprünge von -128 bis +127 Byte, bezogen auf den folgenden Befehl (Wert von EPI alias program counter).

Kapitel 8

Zahlendarstellung

```
Taschenrechner
rechnen tatsächlich mit einer merkwürdigen Kreuzung aus Zweier- und
Zehnersystem, der sogenannten BCD-Arithmetik (BCD = binary coded decimal).
Dabei wird nicht die gesamte Zahl ins Zweiersystem übertragen, wie bei
anderen Rechnern, sondern jede Ziffer des Zehnersystems einzeln als kurze
Zahl im Zweiersystem.

Stack: Postscript, UPN

Speicher: konstant, variabel

CDC Cyber 76: CPU 1:30h, real 1-2 Tage bzw. Wochenende
Cyber 990:    20-30 min
Atari ST/FPU: 7h
AMD K7/600:   0.5sec




negative Zahlen:

a) ein Bit für Vorzeichen; der Rest ergibt den Betrag

b) Einerkomplementdarstellung
```

```
   -|x| <= ( 2^n - 1 ) - |x|

   01111111 =  127
   ...
   00000001 =  1
   00000000 =  0
   11111111 = -0
   11111110 = -1
   ...
   10000000 = -127

   Negieren: Bits invertieren

c) Zweierkomplement:

   -|x| <= ( 2^n ) - |x|

   01111111 =  127
   ...
   00000001 =  1
   00000000 =  0
   11111111 = -1
   11111110 = -2
   ...
   10000000 = -128

   Negieren: Bits invertieren

d) Excess/BIAS

   11111111 = 127
   ...
   10000000 = 0
   01111111 = -1
   ...
   00000000 = -128




Der folgende Text ist aus einem Programmkommentar entnommen, und nur
```

für mich als Merkhilfe gedacht:

```
// Dabei sollten sowohl Systeme mit einer internen Darstellung
// als "big endian" (Sun, Motorola 68xxx, PowerPC) als auch Systeme
// mit "little endian" (Intel) funktionieren.
// DEC Vax: gemischt:


Die Zahl 0x12345678
                    Big endian:    Little endian:   Vax (evtl.):
hohe Adressen...
                        0x12            0x78          0x56
                        0x34            0x56          0x78
                        0x56            0x34          0x12
                        0x78            0x12          0x34
niedrige Adressen...
```

Festkommazahlen:
feste Anzahl Vor- und Nachkommastellen
problematisch bei kleinen und großen Zahlen

Gleitkommazahlen: großer Bereich bei fester relativer Genauigkeit

rationale Zahlen

IEEE 754: Literatur

big/little endian byte order für double-Werte herausfinden,
dabei nebenbei prüfen, ob die Bitmuster IEEE 754 entsprechen.
Das funktioniert prinzipiell ähnlich wie bei den int-Werten:
in einer union liegen eine Gleitkommazahl (8 Byte) und ein char[]-
Feld parallel.
Wenn man jetzt bestimmte Werte in die Gleitkommazahl schreibt,
kann man im char[]-Feld die Werte der einzelnen Bytes angucken.

```
Byte <---3--><---2--><---1--><---0-->
Bit  3322222222221111111111
```

```
     10987654321098765432109876543210
     seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

BIAS für e:

Aufbau einer double nach IEEE ([3], [4]):

Byte <---7--><---6--><---5--><---4--><---3--><---2--><---1--><---0-->
Bit  6666555555555544444444443333333333222222222211111111111
     3210987654321098765432109876543210987654321098765432109876543210
     seeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm

Bit  63 (s) ist das Vorzeichen (0 ist '+', 1 ist '-')
Die 11 Bits 52-62 (e) stellen den Exponenten dar, verschoben um
den Wert 1023 ins Positive.
Die 52 Bits 0-51 (m) sind die Mantisse.
Bei normalisierten Zahlen (Wert für den Exponenten von -1022 bis 1023,
verschoben in den Bereich 1-2046 für e) wird vor der Binärziffernfolge
von m eine "1." gedacht, und nicht mit abgespeichert.

Beispiel 1:
Die Dezimalzahl 1.0 ist binär ebenfalls 1.0, also 1.0*2^0.
Die "1." wird nicht gespeichert, die gesamte Mantisse m ist 0.
Der Exponent ist 0, und wird um 1023(dez)=1111111111(binär)
verschoben.
s ergibt:
Byte <---7--><---6--><---5--><---4--><---3--><---2--><---1--><---0-->
Bit  6666555555555544444444443333333333222222222211111111111
     3210987654321098765432109876543210987654321098765432109876543210
     seeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
bin: 0011111111110000000000000000000000000000000000000000000000000000
hex:    3   f   f   0   0   0   0   0   0   0   0   0   0   0   0   0

Beispiel 2:
Die Dezimalzahl 1.03125 (1+2^-5) ist binär 1.00001, also 1.00001*2^0.
Die "1." wird nicht gespeichert, die gesamte Mantisse ist 0000100...
Der Exponent ist 0, und wird um 1023(dez)=1111111111(binär)
verschoben.
Das ergibt:
Byte <---7--><---6--><---5--><---4--><---3--><---2--><---1--><---0-->
Bit  6666555555555544444444443333333333222222222211111111111
     3210987654321098765432109876543210987654321098765432109876543210
     seeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
```

```
bin: 0011111111110000100000000000000000000000000000000000000000000000
hex:    3   f   f   0   8   0   0   0   0   0   0   0   0   0   0   0

Beispiel 3:
Die Dezimalzahl
1+(2^(-5))+(2^(-14))+(2^(-23))+(2^(-32))+(2^(-41))+(2^(-50))
ist binär
1.0000100000000100000000100000000100000000100000000100,
also 1.0000100000000100000000100000000100000000100000000100*2^0.
Die "1." wird nicht gespeichert, die gesamte Mantisse ist 0000100...
Der Exponent ist 0, und wird um 1023(dez)=1111111111(binär)
verschoben.
Das ergibt:
Byte <---7--><---6--><---5--><---4--><---3--><---2--><---1--><---0-->
Bit  666655555555554444444444333333333322222222221111111111
     3210987654321098765432109876543210987654321098765432109876543210
     seeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
bin: 0011111111110000100000000100000000100000000100000000100000000100
hex:    3   f   f   0   8   0   4   0   2   0   1   0   0   8   0   4
```

denormalisierte Zahlen, NAN, $+\infty$, $-\infty$

Kapitel 9

Noch zu erledigen

TODO Referate nachtragen:

- Schnittstellen
- Prozessoren
- Netzwerkprotokolle

Schaltalgebra, binäres Rechnen, 2 Bits zu einem Ergebnisbit und Übertrag addieren, Volladdierer, Schaltung umformen

Grundelemente Rechner

Teil II

Referate

In diesem Teil sind die Referate des letzten Semesters eingefügt. Außer einigen offensichtlichen Fehlern, die ich (kw) leise korrigerte, sind die Texte bisher weitgehend ungeprüft und unkorrigiert.

Kapitel 10

Codierungsverfahren

Das Referat *Codierungsverfahren* wurde von Thorsten Wiggers und Patrick Thelen erarbeitet und vorgetragen.

10.1 Warum Codierungsverfahren

Besonders durch die gestiegene Popularität des Internet durch die Möglichkeiten der Präsentation im World Wide Web hat die Bildcodierung enorm an Wichtigkeit gewonnen.

Zum einen locken besonders bunte Bilder und ansprechende Illustrationen die Menschen auf bestimmte Webseiten, zum anderen bewirken gerade große Bilder erhöhte Wartezeiten durch die im Vergleich zu Festplatten oder CD-ROMs kümmerlich kleine Bandbreite einer Netzübertragung.

Besonders durch die gestiegene Popularität des Internet durch die Möglichkeiten der Präsentation im World Wide Web hat die Bildcodierung enorm an Wichtigkeit gewonnen. Zum einen locken besonders bunte Bilder und ansprechende Illustrationen die Menschen auf bestimmte Webseiten, zum anderen bewirken gerade große Bilder erhöhte Wartezeiten durch die im Vergleich zu Festplatten oder CD-ROMs kümmerlich kleine Bandbreite einer Netzübertragung.

Kaum daß die Größe eines digitalen Bildes auf der Festplatte durch eine mittlerweile Mindestgröße neuer Platten von 1 GB fast unerheblich geworden ist, bedeutet die Reduzierung der Übertragungszeit durch Bildkompression von z.B.

5 Minuten auf 20 Sekunden (BMP auf JPEG bei einem 1024x768 TrueColor Landschaftsbild) einen erheblichen Zeitgewinn.

Unkomprimierte digitale Videos dagegen beanspruchen so viel Speicherplatz, daß auch heutige Festplatten schnell am Ende sind, von der Übertragungsrate ganz zu schweigen.

Bei der Codierung unterscheidet man grundsätzlich zwischen verlustbehafteter (lossy) und verlustfreier Codierung.

Bei verlustfreier Codierung werden nur die Redundanzen eines Bildes zusammengefaßt, z.B. wird bei aufeinanderfolgenden Pixeln gleicher Farbe nur einmal die Farbe und die Anzahl der Pixel gespeichert (variable Lauflängencodierung - variable length coding, VLC). Die Wiederherstellung des Ursprungsbildes ist jederzeit vollständig möglich. Man spricht hierbei von Datenkompression.

Zusätzlich werden bei verlustbehafteter Codierung (lossy) die irrelevanten Informationen entfernt, also Informationen, die das menschliche Auge sowieso nicht erkennen oder unterscheiden kann. Es ist dabei möglich, den Fehlergrad einzustellen, so daß zur besseren Datenreduktion auch Informationen weggelassen werden können, die kaum wahrnehmbar sind. In der Realität gibt es allerdings kein Verfahren, das absolut keine Störungen erzeugt. Diese Bildstörungen, Artefakte genannt, verstärken sich natürlich bei steigender Kompressionsrate. Bei diesen Methoden ist eine Rekonstruktion des Ausgangsbildes natürlich nicht möglich, dafür sind im Allgemeinen die Kompressionsraten erheblich höher als bei verlustfreier Codierung.

10.2 Möglichkeiten der Codierung

Wie schon erwähnt beruht jede Kompression auf der Reduzierung oder sogar Beseitigung von Redundanzen. Diese können räumlich, spektral oder bei Filmen zusätzlich noch zeitlich sein.

Räumliche Redundanz bedeutet, daß die gleiche oder zumindest eine ähnliche Bildinformation mehrfach vorkommt, etwa bei sich wiederholenden Mustern.

Die **spektrale Redundanz** bezieht sich auf Ähnlichkeiten der Farbkomponenten, z.B. bei einem einfarbigen Bildausschnitt.

Bei Filmen verändert sich oft nur ein Ausschnitt, z.B. eine Person, während das restliche Bild, der Hintergrund, unverändert bleibt. Dieses bezeichnet man mit **zeitlicher Redundanz**.

Die Kompressionsverfahren entfernen die gefundenen Redundanzen, idealerweise ohne daß der Betrachter eine Beeinträchtigung der Bildqualität feststellt.

Grundsätzlich läßt sich jeder Kompressionsvorgang in vier Phasen aufteilen, wobei nicht jede Phase bei jedem Verfahren vorhanden sein muß:

- Vorverarbeitung
- reversible Transformation
- Quantisierung
- Umcodierung zur Kompression

Die **Vorverarbeitung** dient dazu die Bildparameter so zu verändern, daß nachfolgend eine bessere Kompression möglich ist. Das wird durch Filterung, Löschen unerwünschter Bildabschnitte oder auch Transformation der Bildelemente erreicht.

Die **reversible Transformation** stellt den verlustfreien Teil der Umcodierung dar. Hier werden die Bildelemente bijektiv, daher reversibel, auf eine andere Menge von Bildelementen abgebildet.

Die dritte Phase,**Quantisierung**, führt die Irrelevanzreduktion durch. Sie ist bei den verlustfreien Verfahrenen nicht vorhanden.

Zum Abschluß werden die transformierten und evtl. quantisierten Bildelemente **umcodiert**, so daß idealerweise häufig vorkommende Elemente kurze, selten vorkommende Elemente lange Codes erhalten. Dies wird **Entropiecodierung** genannt.

Dabei ist Entropie definiert als die untere Schranke für die Codelänge einer Codekomprimierung ohne Beachtung eines Kontextes, also nach vollständiger Entfernung aller Redundanzen. Dies wird durch Huffman- oder arithmetische Codierung erreicht.

Diese Codierungsverfahren werden bei fast allen Graphikformaten benutzt.

10.2.1 Huffman Codierung

Die Huffman-Kodierung geht schon auf den guten alten Morse und sein Morse-Alphabet zurück. Das Prinzip bildet die häufiger auftretenden Symbole auf kürzere, selten auftretende auf die langen Codes ab. Der in der Datenverarbeitung sehr weit verbreitete ASCII-Code codiert jedes Zeichen mit 8 Bit. Codiert man nun nach Huffman die häufiger auftretenden Zeichen mit weniger Bits und die seltener auftretenden Zeichen mit mehr Bits so spart man offenkundig Volumen ein.

Zur Festlegung, welches Zeichen mit wievielen Bits codiert werden soll, müssen also Informationen über die Zeichenhäufigkeiten vorhanden sein, wofür drei Möglichkeiten existieren:

statisch: Die Zeichenhäufigkeiten werden vorher festgelegten Tabellen entnommen.

dynamisch: Die Daten werden einmal ganz gelesen, um die dort geltenden Häufigkeiten zu bestimmen.

adaptierend: Es wird mit festen Vorgaben begonnen (z.B.: alle Zeichen treten gleich oft auf, oder das e ist das häufigste Zeichen, usw..), die während der Codierung an die realen Daten angepaßt werden.

Das Verfahren nach Huffman sorgt nicht nur für eine Codierung, die eindeutig, sondern auch optimal ist, d.h., die Länge des codierten Textes wird minimal.

Zur Verdeutlichung des Verfahrens bearbeiten wir das folgende Beispiel: Das Wort **„abrakadabra“** soll mit dem Huffman-Verfahren codiert werden.

Bestimmung der Zeichenhäufigkeiten: Die dynamische Bestimmung der Zeichenhäufigkeiten ergibt:

‘a’: 5 mal

‘b’: 2 mal

‘r’: 2 mal

‘k’: 1 mal

‘d’: 1 mal

Die beiden seltensten Zeichen in „abrakadabra“ sind ‘k’ und ‘d’. Zur Ermittlung der Huffman-Codierung -**wenige Bits für häufige Zeichen, viele Bits für seltene Zeichen** - bauen wir einen Binärbaum auf. Dazu beginnen wir mit den beiden seltensten Zeichen und codieren diese mit 0 und 1. Codieren wir also k mit 0 und d mit 1. Die Codierung lautet bis jetzt:

‘k’: 0

‘d’: 1

Nun fassen wir die Häufigkeiten dieser beiden Zeichen zusammen und suchen wieder die beiden seltensten. Das sind dann ‘r’ und ‘k’ mit ‘d’ zusammen. Hiervon wird wieder eines mit 0, das andere mit 1 codiert. Existiert für ein Zeichen schon ein Code, so wird die 0 oder 1 einfach davorgestellt. Die Codierung lautet dann, wenn wir ‘r’ mit 0 codieren und vor ‘k’ und ‘d’ folglich eine 1 stellen:

‘r’: 0

‘k’: 10

‘d’: 11

Wieder werden die Häufigkeiten zusammengefaßt: ‘r’, ‘k’ und ‘d’ treten zusammen 4 mal auf. Die beiden seltensten Zeichen sind nun ‘b’ und ‘r’, ‘k’ und ‘d’ zusammen. Das b codieren wir mit 0, die anderen Codes bekommen jeweils eine 1 vorangestellt und wir erhalten:

‘b’: 0

‘r’: 10

‘k’: 110

‘d’: 111

Bei der letzten Wiederholung dieses Verfahrens wird ‘a’ mit 0 codiert und alle anderen erhalten eine vorangestellte 1:

‘a’: 0

‘b’: 10

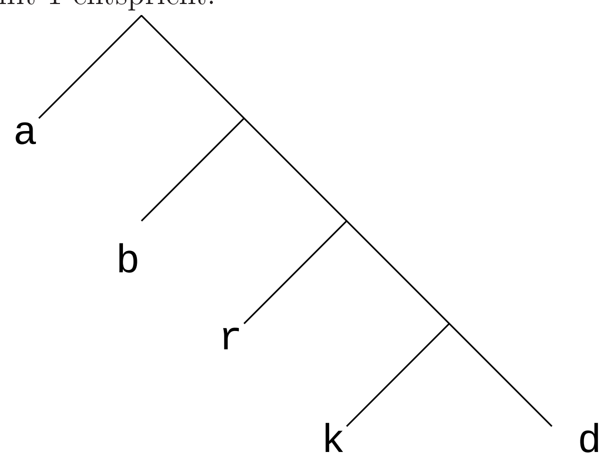
‘r’: 110

‘k’: 1110

‘d’: 1111

Das Wort „abrakadabra“ wird dann also mit 01011001110011110101100 codiert.

Bei der eben dargestellen Codierung können wir den folgenden Binärbaum aufgestellen, wobei ein Pfad nach links einer Kodierung mit 0 und ein Pfad nachrechts einer Kodierung mit 1 entspricht:



Zur Decodierung lesen wir einfach Bit für Bit ein und wandern dabei durch den Binärbaum. Wenn wir ein Zeichen als Blatt gefunden haben, ist dieses decodiert und wir fangen mit dem nächsten Bit wieder an der Wurzel des Baumes an.

Diese Codierung benötigt für das Wort „abrakadabra“ **23 Bit**:
01011001110011110101100

Eine Codierung in 8-Bit ASCII verschlingt **88 Bit**.
01100001 01100010 01110010 01100001 01101011 01100001 01100100 01100001 01100010 01110010 01100001

10.2.2 LZW-Codierung

LZW ist wohl das bekannteste Lempel Ziv-Verfahren schlechthin. Es ist sehr weit verbreitet und hat wahrscheinlich im Felde der LZ- Verfahren die meisten

Varianten und Verbesserungen erfahren. Auch heutzutage ist LZW immer noch interessant genug um Entwicklungen auf seiner Basis zu führen.
Terry A. Welch hat LZW mit dem Hintergedanken einer Hardwareimplementierung entwickelt. Für diesen Zweck muß der zugehörige Algorithmus einigen Voraussetzungen genügen. Zum einen muß er für den Benutzer und das Computersystem transparent sein. Zum anderen muß die Kompression und Dekompression sehr schnell sein. Für eine universelle Nutzung der Hardware soll die Komprimierung außerdem ungeachtet der im Datenstrom auftretenden Redundanzarten möglichst optimal sein. LZW erfüllt diese Voraussetzungen aufgrund seiner Struktur weitestgehend.

LZW basiert auf einem "Wörterbuchverfahren" mit einem Wörterbuch von bis zu 4096 möglichen Einträgen und einer festen Codelänge von 12 bit. Das Wörterbuch wird wie bei allen LZ-Verfahren während des Kompressions- bzw. Dekompressionsvorgangs aufgebaut. Eine Ausnahme bilden die ersten 256 Einträge, die von vornherein mit den Abbildungen der einzelnen Bytes vorbelegt sind. Zur Laufzeit werden aus den Eingabedaten Muster gewonnen, die, sofern sie noch nicht bekannt sind, ins Wörterbuch eingetragen werden. Im anderen Fall wird nicht das Muster sondern sein Index im Wörterbuch ausgegeben.

Wieder eine Erklärung am Beispiel des Wortes
"abrakadabra"

Eingabe	Erkanntes Muster	Wörterbucheintrag
abrakadabra	a	ab $< 256 >$
brakadabra	b	br $< 257 >$
rakadabra	r	ra $< 258 >$
akadabra	a	ak $< 259 >$
kadabra	k	ka $< 260 >$
adabra	a	ad $< 261 >$
dabra	d	da $< 262 >$
abra	ab	$< 256 >$
ra	ra	$< 258 >$

Jede Zeile im Beispiel zeigt jeweils den Inhalt des Eingabestrings in jedem Schritt der Kodierung, das in diesem Schritt längste im Wörterbuch gefundene Zeichenmuster und neu hinzugekommene Wörterbucheinträge mit ihren Indexnummern.

Im ersten Schritt wird das längste gefundene Muster zwangsläufig nur ein einziger Buchstabe sein, da beim LZW-Verfahren das Wörterbuch mit allen Einzelzeichen mit den Codewerten von 0 bis 255 vorbelegt ist, aber keine längeren Zeichenketten enthält. In unserem Beispiel findet der Algorithmus also das 'a'. Im gleichen

Schritt wird noch ein weiteres Zeichen der Eingabe betrachtet ('b') und an das 'a' angehängt. Die so erzeugte Zeichenkette ist garantiert noch nicht im Wörterbuch enthalten, wird also unter dem Index 256 neu eingetragen. Im nächsten Schritt wird vom Eingabestrom die im letzen Schritt längste gefundene Zeichenkette (also das 'a') entfernt und ausgegeben.
Das 'b' wird nun zum ersten Zeichen des neuen Eingabestrings. Hier beginnt das gleiche Spiel von vorne. 'b' ist das längste bekannte Muster. 'br' wird unter dem Index 257 ins Wörterbuch aufgenommen. 'b' wird von der Eingabe entfernt, ausgegeben und ein neuer Durchlauf begonnen. 'r' ist nun längste im Wörterbuch eingetragene Zeichenkette, 'ra' wird mit Indexnummer 258 neu aufgenommen und das 'r' aus der Eingabe gestrichen und zur Ausgabe umgeleitet.
'a' ist nun längste im Wörterbuch eingetragene Zeichenkette, 'ak' wird mit Indexnummer 259 neu aufgenommen und das 'a' aus der Eingabe gestrichen und zur Ausgabe umgeleitet.
'k' ist nun längste im Wörterbuch eingetragene Zeichenkette, 'ka' wird mit Indexnummer 260 neu aufgenommen und das 'k' aus der Eingabe gestrichen und zur Ausgabe umgeleitet.
'a' ist nun längste im Wörterbuch eingetragene Zeichenkette, 'ad' wird mit Indexnummer 261 neu aufgenommen und das 'a' aus der Eingabe gestrichen und zur Ausgabe umgeleitet.
'd' ist nun längste im Wörterbuch eingetragene Zeichenkette, 'da' wird mit Indexnummer 262 neu aufgenommen und das 'd' aus der Eingabe gestrichen und zur Ausgabe umgeleitet.
Im nächsten Schritt passiert endlich etwas interessanteres; das längste bekannte Muster ist nun eine Zeichenfolge, die wir selbst ins Wörterbuch eingetragen haben ('ab' mit Indexnummer 256). Jetzt werden im nachfolgenden Schritt nicht zwei Einzelzeichen ausgegeben, sondern der Index des Musters aus dem Wörterbuch (256).
Das selbe geschieht nun auch bei der nächsten Zeichenkette 'ra' ist in unserem Wörterbuch unter der Indexnummer 258 eingetragen. Es wird also wieder nur der Index des Musters ausgegeben.
Die Ausgabe würde also wie folgt aussehen:

a b r a k a d $< 256 >$ $< 258 >$

Dekodierung:

Eingabezeichen	C	Neuer Wörterbucheintrag	P
a			a
b	b	ab=< 256 >	b
r	r	br=< 257 >	r
a	a	ra=< 258 >	a
k	k	ak=< 259 >	k
a	a	ka=< 260 >	a
d	d	ad=< 261 >	d
< 256 >	a	da=< 262 >	ab
< 258 >	r	abr=< 263 >	ra

Ausgabe: a b r a k a d ab ra

Bei der Dekodierung fangen wir ebenfalls mit einem schon vorinitialisierten Wörterbuch an. Wieder sind Einträge von 0 bis 255 vorhanden. In unserem Beispiel ist 'a' die längste bekannte Zeichenkette. Sie wird deshalb auch ausgegeben und in der Variablen P (wie 'Präfix') festgehalten. Die nächste Eingabe ist ebenfalls ein bekanntes Zeichen. Dieses wird zunächst in der Variablen C gespeichert. Nachfolgend wird P mit C konkateniert und das Ergebnis ins Wörterbuch unter der Indexnummer 256 aufgenommen. Man beachte, daß 'ab' sowohl im Dekompressor als auch im Kompressor an der gleichen Stelle im Wörterbuch vorkommt (256).

Nachfolgend wird die letzte Eingabe ('b') in die Variable P kopiert und P ausgegeben. Der nachfolgende Schritt (mit 'r') funktioniert genauso.

Kommen wir zu dem Punkt an dem der erste Index eingelesen wird. Hier wird nur das erste Zeichen des Wörterbucheintrags nach C übernommen ('a'). Dann wird wieder P und C konkateniert und ein neuer Wörterbucheintrag erzeugt ('da' mit Index 262). Jetzt wird P der Index 256 zugewiesen und der dazugehörige Wörterbucheintrag ausgegeben.

Problemfall K[omega]K

Bei dieser Eingabekombination kann es bei der Implementierung des Dekoders Probleme geben. Hierbei ist K ein beliebiges Zeichen, gefolgt von einer Zeichenkette [omega], wiederum gefolgt von K. Entscheidend ist hierbei, daß sich K[omega] bereits im Wörterbuch befindet. Die Kodierung selbst bereitet keine Probleme;

diese ergeben sich erst im Dekoder. In diesem Fall hinkt sozusagen der Dekoder dem Enkoder hinterher. Der Enkoder schickt hier den Code für seinen letzten Wörterbucheintrag. Dieser Eintrag ist dem Dekoder jedoch zu dieser Zeit noch unbekannt.

Beispiel: **APAPAPAPAP**

Kodierung:

Eingabe	Erkanntes Muster	Neuer Wörterbucheintrag
APAPAPAPAPAP	A	AP $< 256 >$
PAPAPAPAPAP	P	PA $< 257 >$
APAPAPAPAP	AP	APA $< 258 >$
APAPAPAP	APA	APAP $< 259 >$
PAPAP	PA	PAP $< 260 >$
PAP	PAP	$< 260 >$

Ausgabe: A P $< 256 >< 258 >< 257 >< 260 >$

Dekodierung:

Eingabezeichen	C	Neuer Wörterbucheintrag	p
A			A
P	P	AP $< 256 >$	P
$< 256 >$	A	PA$< 257 >$	AP
$< 258 >$	???		

Trick: K[omega]K Fall ist aufgetreten, C ist erstes Zeichen der letzten Ausgabe (also 'A')

	A	APA$< 258 >$	APA
$< 257 >$	P	APAP$< 259 >$	PA
$< 260 >$	???		

gleiches Problem wie oben, d.h. C muß 'P' sein

	P	PAP$< 260 >$	PAP

Ausgabe: A P AP APA PA PAP

10.3 Bildformate

10.3.1 GIF

Der GIF-Standard wurde von Compuserve entwickelt, um die Übertragungsgeschwindigkeiten dieses Onlinedienstes so gering, wie möglich zu halten.

GIF ist ein verlustfreies Kompressionsverfahren, allerdings können nur 8 Bits pro Pixel gespeichert werden, was zu einer maximalen Farbanzahl von 256 führt. Somit gibt es selbstverständlich einen Qualitätsverlust, wenn Bilder mit mehr als 256 Farben als GIF komprimiert werden, da die Farbanzahl angepaßt werden muß.

Die Basis der GIF-Kompression ist das LZW-Verfahren, das sich wiederholende Symbolfolgen vergleicht. Ein Nachteil dieses Verfahrens bei der Bildkompression ist, daß der LZW-Codierer zeilenweise arbeitet und somit z. B. Waagerechte Farbverläufe nicht komprimiert werden können.

Die Kompressionsrate von GIF liegt bei ca. 5:1.

10.3.2 JPEG

Das JPEG-Format ist verlustbehaftetes Kompressionsverfahren, welches sich besonders für die Komprimierung von digitalen Fotos eignet.

Die Basis dieses Formates ist DCT-Verfahren (Discrete-Cosinus Transformation). Die Idee hinter diesem Verfahren ist, daß die Information über jeden Pixel in Relation zu seinem Vorgängerpixel angegeben werden kann.

Beispiel:

	0	1	2
0	rot	0.5*(0/0)	2*(0/1) <— entspricht wieder Pixel (0/0)

Somit werden nur noch die Veränderungskoeffizienten der jeweiligen Pixel gespeichert, was einen erheblich geringeren Datenaufwand verursacht.

10.4 MP3

MP3 bedeutet 'MPEG Layer 3'. MPEG steht für 'Moving Pictures Expert Group'. Diese Gruppe hat sich zum Ziel gesetz, Standards für die Kodierung von Filmdaten zu entwickeln. Da Filme auch Audiodaten enthalten, ist dieser Bereich auch ein Teil des MPEG- Standards.

Der MPEG-Standard setzt sich aus mehreren teilen zusammen:

1. Kodierung von Videosignalen
2. Kodierung von Audiosignalen
    1. Layer 1
    2. Layer 2
    3. Layer 3

Die einzelnen Layer der Audiosignale unterscheiden sich durch die Kompressionsrate

Bezeichnung	Kompression	kbps (Datenübertragung)
Layer 1	1:4	384
Layer 2	1:6...1:8	256...192
Layer 3	1:10...1:12	128...112

Derzeit haben sich MPEG-1 und MPEG-2 als Videostandards etabliert. Wobei sich der Audiobereich der beiden Formate dahingehend unterscheidet, daß bei MPEG-2 jetzt auch Signale zur Ansteuerung von Multispeaker-Systemen (Surround-Systeme) mit verarbeiten lassen.

Da MP3 den Layer 3 des MPEG Standards beschreibt hier eine kleine Übersicht der unterschiedlichen Qualitätsstufen:

Kompressionsrate	Qualität	kbps	Modus
1:96	Telefon	8	mono
1:48	Kurzwelle	16	mono
1:24	Langwelle	32	mono
1:24...26	Ultrakurzwelle	56..64	stereo
1:14...16	unterhalb CD	96...112	stereo
1:12...14	CD	112...128	stereo

Da es nun die Möglichkeit gibt, Audiodaten mit relativ geringem Datenaufwand zu übertragen, bieten sich viele Möglichkeiten, dieses Format zu nutzen. Radio und Fernsehsender können nun aus aller Welt Liveinterviews und andere Berichte über die Telefonleitung übertragen. Das ist grundsätzlich nichts Aufregendes, aber durch das Layer 3 Format kann das alles jetzt in erheblich besserer Qualität erfolgen. Hat man früher noch hören können, daß der Reporter durch ein Telefon spricht, so wird die Stimme nun erst in das Layer 3 Format umgewandelt und dann via ISDN in Echtzeit an den Sender geschickt. Dort werden die Daten wieder in Audiosignale umgesetzt und ohne merkliche Verzögerung gesendet.

Um die Vorteile des MPEG Layer 3 Formates darzustellen, hier ein kleines Beispiel:

Wird z.b. eine CD auf die Festplatte aufgenommen (im WAV-Format), so braucht man für eine Minute ungefähr 10.000Kb = 10Mb. Nehmen wir nun eine sehr gute Computerverbindung via ISDN, so brauchen wir mehr als 20 Minuten, um diese Minute zu übertragen. Werden die Audiodaten vor der Übertragung erst in das MP3 Format umgewandelt, so braucht man nur noch 1/11-1/12 der Zeit. Somit ist es selbst mit einem 56K-Modem möglich eine 45 Minuten CD in knappen 90 Minuten zu übertragen.

Und genau das ist das Problem, das die Musikindustrie heute hat! Denn durch MP3 ist es möglich, Musikstücke in relativ geringer Zeit mit nahezu keinem Qualitätsverlust über das Internet auszutauschen.

10.4.1 Funktionsweise von MP3

Die meisten werden sie jetzt sagen, das eine derartige Reduzierung der Datenmenge ohne Qualitätsverlust nicht geht. MP3 basiert aber auf dem Gedanken, das der Mensch nur einen bestimmten Frequenzgang auch hören kann. Es werden also zu einen die Frequenzbereiche 'abgeschnitten', die das menschliche Ohr nicht wahrnehmen kann. Desweiteren ist das Ohr sehr träge. Dazu ein Beispiel: ein Orchesterstück wechselt sehr schnell von vielen schnellen Instrumenten auf ein einzelnes Leises. Das Ohr bekommt nun von der ersten 1/10 Sekunde des leisen Instrumentes gar nichts mit. Da man diesen Teil nicht hören kann, werden diese Daten auch zu Reduzierung benutzt.

Diese Möglichkeiten wurden ausgiebig beim Entwickler des MP3 Formates, dem deutschen Fraunhofer Institut, getestet. Für diese Test wurden Menschen mit dem sogenannten 'Goldenen Ohr' unterschiedliche Musikstücke im normalen und im komprimierten Zustand vorgespielt. Da diese Menschen sowieso erheblich besser hören, als der normal lärmgeschädigte Durchschnittsmensch, konnten die Ergebnisse problemlos für das MP3-Komprimierungsvefahren eingesetzt werden.

Kapitel 11

Programmiersprachen

Referate:

- generelle Regeln für wissenschaftliche Arbeiten (Recherche, Literatur, Quellenangaben, Nachprüfbarkeit)
- angemessen an Hörer
- was erzähle ich?
- wieviel erzähle ich?
- Reaktionen beachten
- Höflichkeit: Vortragender, Hörer
- Tafel verwenden
- Tafelbild
- langsam beschreiben, Bilder nicht zu schnell auflegen und ersetzen
- technisches Funktionieren sicherstellen
- Zeit einhalten
- pro Bild nicht zu viel darstellen (6—7 Punkte)
- ...

Teil III

Anhang

Anhang A

GNU GENERAL PUBLIC LICENSE

```
		    GNU GENERAL PUBLIC LICENSE
		       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
                       59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

			    Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
```

in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another

language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on

```
    the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

    a) Accompany it with the complete corresponding machine-readable
    source code, which must be distributed under the terms of Sections
    1 and 2 above on a medium customarily used for software interchange; or,

    b) Accompany it with a written offer, valid for at least three
    years, to give any third party, for a charge no more than your
    cost of physically performing source distribution, a complete
    machine-readable copy of the corresponding source code, to be
    distributed under the terms of Sections 1 and 2 above on a medium
    customarily used for software interchange; or,

    c) Accompany it with the information you received as to the offer
    to distribute corresponding source code.  (This alternative is
    allowed only for noncommercial distribution and only if you
    received the program in object code or executable form with such
    an offer, in accord with Subsection b above.)
```

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent

infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
  Yoyodyne, Inc., hereby disclaims all copyright interest in the program
  ‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Anhang B

Der Quelltext von Windows 98

```
/* Netscape already did it and now Microsoft has done it.
 * The source Windows 98 available for everybody!
 */

/*
 * TOP SECRET Microsoft(c)  Code
 * Project: Chicago(tm)
 * Projected release-date: Summer 1998
 * Original version date : Aug 1994
 */

#include "win31.h"
#include "win95.h"
#include "evenmore.h"
#include "oldstuff.h"
#include "billrulz.h"
#define INSTALL = HARD

char make_prog_look_big[1600000];

void main()
{
  while(!CRASHED)
    {
      display_copyright_message();
      display_bill_rules_message();
      do_nothing_loop();
```

```
    if (first_time_installation)
      {
        make_50_megabyte_swapfile();
        do_nothing_loop();
        totally_screw_up_HPFS_file_system();
        search_and_destroy_the_rest_of_OS/2();
        hang_system();
      }
    write_something(anything);
    display_copyright_message();
    do_nothing_loop();
    do_some_stuff();
    if (still_not_crashed)
      {
        display_copyright_message();
        do_nothing_loop();
        basically_run_windows_3.1();
        do_nothing_loop();
        do_nothing_loop();
      }
  }
if (detect_cache())
  disable_cache();

if (fast_cpu())
  {
    set_wait_states(lots);
    set_mouse(speed, very_slow);
    set_mouse(action, jumpy);
    set_mouse(reaction, sometimes);
  }

/* printf("Welcome to Windows 3.11"); */
/* printf("Welcome to Windows 95"); */
printf("Welcome to Windows 98");
if (system_ok())
  crash(to_dos_prompt);
else
  system_memory = open("a:\swp0001.swp", O_CREATE);

while(something)
  {
    sleep(5);
```

```
      get_user_input();
      sleep(5);
      act_on_user_input();
      sleep(5);
    }
  create_general_protection_fault();
}
```

Anhang C

Literatur

[Linux - Unix Shells]

Helmut Herold: Linux - Unix Shells
Bourne-Shell, Korn-Shell, C-Shell, bash, tcsh 3. Auflage
Addison Wesley, 1999
ISBN 3-8273-1511-5

[Ebeling: Geist aus der Maschine]

Adolf Ebeling: Geist aus der Maschine
Das erste Jahrtausend des Computers neigt sich seinem Ende zu
ct́ 26/99, Seite 74ff.; Heise Verlag, 1999

Lesenswerter Überblick über die wichtigsten Schritte zum heutigen Computer

[Borchers/Benning/Kuri: Internet]

Detlef Borchers, Maria Benning, Jürgen Kuri: ‘Hätt ich dich heut erwartet ...´

Das Internet hat Geburtstag
c't 21/99, Seite 128ff.; Heise Verlag, 1999

Interessantes über die ersten Anfänge des Internet

[0 ist false, alles andere true]

Klaus Wachtler: 0 ist false, alles andere true
Eine Einführung in ANSI-C
Soltau, 1999

für lau!

[The Linux Programmer's Guide]

Goldt, v.d.Meer, Burkett, Welsh: The Linux Programmer's Guide
(Teil des *Linux Documentation Project*)

[Stallman: GNU Emacs Manual]

Richard Stallman: GNU Emacs Manual
`http://www.fsf.org`

[Tanenbaum: Minix]

Andrew S. Tanenbaum: Design and Implementation of Operating Systems oder so ähnlich TODO
Addison Wesley
ISBN

Um verständlich zu machen, wie man ein echtes Unix-ähnliches Betriebssystem baut, entwirft Tanenbaum ein vereinfachtes, aber funktionierendes System, nämlich Minix. Bevor Linux aufkam, war Minix das einzige im Quelltext frei verfügbare Unix. Als tatsächlich einsetzbares System ist Linux wesentlich fortgeschrittener; aber als Lehrstück ist Minix allein deshalb gut, weil es viel übersichtlicher und besser dokumentiert ist (nämlich in diesem Werk). Sehr gutes Buch!

[Postscript Language]

Adobe Systems: Postscript Language Reference Manual
Second Edition
Addison Wesley, 1990
ISBN 0-201-18127-4

Sehr gute, genaue und trotzdem verständliche Einführung und Referenz zu Postscript. Ein ausgesprochen gelungenes Buch!

[Stroustrup: C++]

Bjarne Stroustrup: Die C++-Programmiersprache
4., aktualiserte und erweiterte Auflage
Addison Wesley, 2000
ISBN 3-8272-1660-X

DAS Buch zu C++. Entsprechend der Programmiersprache teilweise etwas hartes Brot. Nachdem die erste und zweite Auflage ziemlich planlos waren und die deutschen Übersetzungen auch noch dagegen absackten, sind die dritte und jetzt die vierte Auflage wirklich gut, auch in der deutschen Ausgabe. Sehr umfassend.

[Starke: Schaltungslehre]

Lothar Starke: LaTeX
Schaltungslehre der Elektronik

Frankfurter Fachverlag
ISBN 3-87234-031-X

Übersichtlicher und praxisgerechter Rundumschlag durch die Schaltungslehre; setzt allerdings weiteres physikalisches und elektrotechnisches Wissen voraus.

[Lamport: LaTeX]

Leslie Lamport: LaTeX
A Document Preparation System
Addison Wesley
ISBN 0-201-15790-X

Die Beschreibung des Textsatzsystems LaTeX, welches auf dem TeX-System von Donald E. Knuth aufbaut. Das gängige deutsche Buch zu LaTeX ist von Helmut Kopka.

[Knuth: Fundamental Algorithms]

Donald E. Knuth: The Art of Computer Programming
Volume 1: Fundamental Algorithms
Addison Wesley
ISBN 0-201-03809-9

Einige elementare Verfahren, Laufzeitüberlegungen, Listen, Felder, Bäume; Beschreibung der auch in den weiteren Bänden verwendeten fiktiven Assemblersprache `MIX`

[Knuth: Seminumerical Algorithms]

Donald E. Knuth: The Art of Computer Programming
Volume 2: Seminumerical Algorithms

Addison Wesley
ISBN 0-201-03822-6

Zahlendarstellungen, binäre Arithmetik, Zufallszahlen, eher mathematisch orientiert.

Trotz der theoretischen Orientierung und des trockenen Stoffs gut lesbar.

[Knuth: Sorting, Searching]

Donald E. Knuth: The Art of Computer Programming
Volume 3: Sorting and Searching
Addison Wesley
ISBN 0-201-03803-X

Umfassende Darstellung der wichtigen Verfahtren zum Sortieren und Suchen von Daten; eher mathematisch orientiert.

[Müllges/Reutter: Numerische Mathematik]

G. Engeln-Müllges, F. Reutter: Numerische Mathematik für Ingenieure
Bibliographisches Institut Mannheim
ISBN 3-411-01688-4

Sehr praxisgerechte Darstellung der wichtigsten numerischen Verfahren aus der Technik (Fehleranalyse, Lösen von Gleichungen, numerische Lösung von linearen und nichtlinearen Gleichungssystemen, Approximation, Interpolation, numerisches Integrieren, Anfangswertprobleme).

Als Ergänzung ist eine Formelsammlung [Lit. C: Müllges/Reutter: Formelsammlung] erhältlich.

[Müllges/Reutter: Formelsammlung]

G. Engeln-Müllges, F. Reutter: Formelsammlung zur Numerischen Mathematik mit Standard-FORTRAN 77-Programmen
Bibliographisches Institut Mannheim
ISBN 3-411-03125-5

Formelsammlung zum Nachschlagen als Ergänzung zu [Lit. C: Müllges/Reutter: Numerische Mathematik]; enthält alle Verfahren in Fortran 77 formuliert. Analoge Ausgaben sind mit BASIC, Pascal und C erhältlich.

[Stevens: Unix Network Programming Vol. 1]

W. Richard Stevens: Unix Network Programming
Vol. 1: Network APIs: Sockets and XTI
Second Edition
Prentice Hall
ISBN 0-13-490012-X

Sehr saubere und umfassende Darstellung der Netzwerkprogrammierung, vor allem unter Unix.

[Stevens: Unix Network Programming Vol. 2]

W. Richard Stevens: Unix Network Programming
Vol. 2: Interprocess Communications
Second Edition
Prentice Hall
ISBN 0-13-081081-9

Wichtiges Thema (IPC), sehr gutes Buch.

[Rago: UNIX System V Network Programming]

Stephen A. Rago: UNIX System V Network Programming
Addison Wesley, 1993
ISBN 0-201-56318-5

Dieses Buch hat im Vorspann eine sehr übersichtliche und gut verständliche Einführung in die Programmierung unter Unix; daher auch sehr interessant, wenn man gar nichts mit Netzwerkprogrammierung zu tun hat!

Für Netzwerkprogrammierung selbst ist es meistens eher uninteressant, weil der Stoff nur unter System V R4 benutzbar ist (die ganzen `t_`*...-Funktionen gemäß X/Open Transport Interface XTI). Hier ist* **C Unix Network Programming Vol. 1** *auf Seite 100 geeigneter.*

[Stevens: Advanced Programming Unix]

W. Richard Stevens: Advanced Programming in the Unix Environment
Addison Wesley, 1993
ISBN 0-200-56317-7

Hervorragendes Buch über die Verfahren zur Programmierung unter Unix. Sehr gut verständliches Englisch. Es werden genau die Unterschiede zwischen den verschiedenen Unixablegern verdeutlicht. Selten so ein gutes und hilfreiches Buch gesehen! Der Bereich der inter process communication *(IPC) wird allerdings komplett ausgeklammert; hierfür gibt es [Lit. C: Stevens: Unix Network Programming Vol. 1]und [Lit. C: Stevens: Unix Network Programming Vol. 2].*

[Herold: Systemprogrammierung]

Helmut Herold: Linux-Unix-Systemprogrammierung
2., überarbeitete Auflage
Addison Wesley, 1999
ISBN 3-8273-1512-3

Trägt das **Linux** *nur im Titel, weil es gerade so Mode ist. Ist in den interessanten Bereichen von [Lit. C: Stevens: Advanced Programming Unix] abgeschrieben (leider unvollständig). Die anderen Bereiche sind teilweise nett gemeint (Einführung in ANSI-C beispielsweise), aber viel zu ungenau.*

[Herold: Tools]

Helmut Herold: Linux-Unix-Profitools
awk, sed, lex, yacc und make
3., überarbeitete Auflage
Addison Wesley, 1999
ISBN 3-8273-1448-8

Etwas grober und gelangweilter, aber verwendbarer Überblick über einige wichtige Werkzeuge.

[Klingenberg: Grundlagen der Angewandten Informatik]

Prof.Dr. Rene Klingenberg: Grundlagen der angewandten Informatik
Fachhohschule Hannover, 2000,

Index