# Skript zur Vorlesung Computergrundlagen

Prof. Dr. H. J. Herrmann

WS 06/07


ETH Zürich, Oktober 2006

# Inhaltsverzeichnis

# Kapitel 1

# Einführung[1]

## 1.1 Was ist Computational Physics?

Automatische programmgesteuerte Datenverarbeitungsanlagen, auch Computer genannt, haben das Leben in den vergangenen zehn Jahren revolutioniert. Ihr Einsatz im Druckereigewerbe als *Desktop Publisher* hat den Beruf des Schriftsetzers zum Aussterben gebracht, ihre Vernetzung über alle irdischen Grenzen hinweg zum *World Wide Web* lässt Informationen und Meinungen –gelegentlich– in Sekunden um den Globus kreisen, und Mikroprozessoren in modernen Haushaltsgeräten besitzen heute im Prinzip eine größere Rechenkapazität als die Bordcomputer der Apollo Mondlandemissionen. Dieser Siegeszug lässt sich historisch nur noch mit der Erfindung der Buchdruckkunst durch Johannes Gutenberg vergleichen, jenem Gewerbe, dass sich die Computer inzwischen ganz und gar einverleibt haben. Eine Ursache für diese Revolution liegt in der fortschreitenden Miniaturisierung elektronischer Schaltkreise, mit denen die grundlegenden Funktionen eines Computers heute bei weitem überwiegend realisiert werden. So passt eine Großrechenanlage der siebziger Jahre heute in jede Aktentasche. Der eigentliche Grund für den Erfolg der Computer liegt aber in seiner ungeheuren Vielseitigkeit, die mit dem Computer als Gerät nur so viel zu tun hat, als er das Medium darstellt, in dem eine Folge von Anweisungen oder modern ausgedrückt, die *Software*, abläuft, die ihn steuert und die ihm die gerade gewünschte Funktionalität verleiht. Und seit die Realisierung ernstzunehmender Rechenanlagen keine raumfüllenden Anlagen mehr erfordert, konnte die Vielseitigkeit der Programmsteuerung ihre volle Tragweite offenbaren.

Der Computer ist keine neue Erfindung. Lediglich seine Realisierung durch die Mikroelektronik ist neu, und diese wird in Zukunft vielleicht durch optische Schaltkreise abgelöst, die von Lasern gespeist werden. Der Wunsch nach einer programmgesteuerten Rechenanlage geht in die Zeit des Mathematikers und Astrono-

[1] Dieses Kapitel wurde uns von Dr. M. Krech aus dem Skript seiner Vorlesung „Physik auf dem Computer – Grundlagen“ (WS 01/02) zur Verfügung gestellt. Vielen Dank!

men Wilhelm Schickard (1592 - 1635) und des Mathematikers Blaise Pascal (1623 - 1662) zurück, als man erkannte, dass die Ausführung numerischer Verfahren zur Lösung mathematischer Probleme eigentlich von einer Maschine, ähnlich einem Webstuhl, übernommen werden könnte. Einfache Rechenhilfen auf mechanischer Basis wie der Abakus sind noch wesentlich älter. Der erste Entwurf eines Computers im heutigen Sinne auf mechanischer Basis stammt von Charles Babbage um ca. 1850. Das Rechenwerk, der Prozessor, war als eine Art Getriebe konzipiert und sollte zumindest addieren und subtrahieren können. Für Eingabe und Zwischenergebnisse waren weitere Räderwerke als *Speicher* und *Register* vorgesehen, und die Programmsteuerung sollte über Lochstreifen erfolgen. Für die Darstellung von Rechenergebnissen waren Zifferntrommeln vorgesehen, die an eines der Register gekoppelt werden sollten. Dieser Entwurf nimmt weitgehend die heutige *Architektur* von Computern mit einem zentralen Rechenwerk (Prozessor) vorweg. Zu Lebzeiten von Charles Babbage wurden aus Kostengründen nur Fragmente seines Computers realisiert. Dass seine Entwürfe im Prinzip funktionstüchtig sind, wurde erst vor rund 10 Jahren am British Science Museum in London durch den Aufbau der sog. *Difference Engine* nachgewiesen. Charles Babbage war übrigens weitsichtig genug, um in einem seiner letzten Entwürfe (*Analytic Engine*) auch gleich einen Drucker vorzusehen. Vielleicht hatte er schon die Vorstellung vom papierlosen Büro als reine Utopie erkannt. Das Zeitalter des Computers wurde schließlich von Konrad Zuse im Jahre 1940 durch Entwurf und Bau der *Z2*, eines Computers mit elektromagnetischen Schaltern, eingeleitet. Einer seiner Relaisrechner, die Z4, war bis 1959 in St. Louis in Frankreich in Betrieb. Ob das Computerzeitalter schon 100 Jahre früher mit Charles Babbage hätte beginnen können, steht dahin, denn die Anforderungen an die mechanische Präzision bei der Herstellung seiner *Difference Engine* sind auch heute nicht im Handumdrehen zu erfüllen.

Wir können hier nur anekdotenhaft einige wenige Schlaglichter auf die Entwicklungsgeschichte des Computers setzen, aber man sollte diesen Abschnitt nicht beschließen, ohne John von Neumann und Alan Turing wenigstens erwähnt zu haben. Von ihnen stammen die grundlegenden theoretischen Überlegungen zum Aufbau, der *Architektur*, eines Computers. Dazu gehören Ein- und Ausgabegeräte, Speicher für Daten und Anweisungen und eine zentrale Recheneinheit (CPU = Central Processing Unit) mit internen Rechenwerken für die Grundrechenarten und den Vergleich von Zahlen, internem Speicher (Registern) und einer Ablaufsteuerung, die einen bestimmten Mindestsatz von Befehlen ausführen können muss. Ein typischer Befehlssatz umfasst das Lesen und Schreiben von Daten (Zahlen), einige Grundoperationen mit den Daten und Verzweigungen im Programmablauf in Abhängigkeit von Bedingungen, die mit Hilfe von Vergleichen überprüft werden. Das aus diesen Befehlen zusammengesetzte Programm und die Daten sind in einem Speicher, dem *Hauptspeicher* des Computers, ausserhalb der CPU abgelegt und werden von dieser ausgelesen und ausgeführt beziehungsweise nach Anweisung des Programms verändert. Zum weiteren theoretischen

Unterbau gehören die Entwicklungsprinzipien von durch den Menschen lesbaren *Programmiersprachen*, welche erst in eine von der CPU ausführbare Anweisungsfolge, das *Maschinenprogramm*, übersetzt werden müssen. Weitere Überlegungen betreffen die Struktur von Programmen, die Unterscheidung zwischen Programm und Daten, und vieles mehr. Neuere Entwicklungen auf dem Gebiet der Rechnerarchitekturen betreffen den gleichzeitigen Einsatz vieler CPUs in *Parallelrechnern* und *verteilten Rechenanlagen*, speziellen CPUs mit mehreren gleichartigen Rechenwerken in *Vektorrechnern* oder *neuronale Netze*, von denen man sich einen Zugang zur *künstlichen Intelligenz* verspricht. Vor diesem Hintergrund ist es um so bemerkenswerter, wie weitgehend Charles Babbage heute noch gebräuchliche *Architekturen* für Computer fast 100 Jahre vor ihrer Zeit in seinen Entwürfen bereits verwirklicht hatte.

Mit der *Physik auf dem Computer* kehren wir gewissermaßen zu den anfänglichen Motiven zurück, die zur Konstruktion von Computern geführt haben, nämlich der numerischen Behandlung von Problemen in Naturwissenschaft und Technik. Der Einsatz des Computers im Schriftsatz oder als ein *Knoten* in einem weltumspannenden Informationsnetz wird uns hier nur ganz am Rande begegnen. Wir werden den Computer, seine Steuerungssoftware, d.h. sein *Betriebssystem*, und die Programmiersprachen als gegebene Werkzeuge ansehen, mit denen wir physikalische Fragestellungen beantworten wollen.

Physik auf dem Computer besteht unter diesem Blickwinkel aus den folgenden für sich sehr umfangreichen Anwendungsfeldern:

- Die Interpretation experimenteller Daten und deren Vergleich mit theoretischen Vorhersagen, bzw. die Identifikation bisher nicht geklärter Gesetzmäßigkeiten in den Daten ist das vielleicht ürsprünglichste Anwendungsfeld. Die Datenanalyse mit statistischen Methoden, Interpolation und Anpassung von Parametern in theoretisch fundierten oder empirisch gefundenen funktionalen Zusammenhängen ist mit Computern effizient und für sehr große Datenmengen durchführbar. In einem zweiten Schritt lassen sich Computer in Abhängigkeit von den ausgewerteten Daten zur Steuerung von Experimenten oder von Prozessen in technischen Anwendungen einsetzen.

- Die mathematische Modellierung von physikalischen oder technischen Vorgängen führt in der Regel zu Gleichungen, "die man nicht lösen kann". Das bedeutet natürlich nicht, dass die Gleichungen keine Lösung besitzen, sondern lediglich, dass sich die Lösung nicht mit Hilfe bekannter transzendenter Funktionen darstellen lässt. In diesem Fall ist es zweckmäßig, zur Lösung der Gleichungen numerische Verfahren zu verwenden, die sich in Form von Computerprogrammen bequem implementieren lassen. Viele Programmiersprachen besitzen bereits eine Bibliothek transzendenter Funktionen, die die numerische Behandlung von analytisch gegebenen Gleichungen unterstützen.

- Auf mikroskopischer Basis aufgebaute Modelle für makroskopische Stoffeigenschaften sind häufig von Anfang an nur für eine numerische Auswertung konstruiert. Wegen der großen Diskrepanz zwischen Längen- und Zeitskalen, auf denen ein solches Modell definiert ist, und den Längen- und Zeitskalen, auf denen Phänomene untersucht werden sollen, ist der erforderliche Rechenaufwand für aussagekräftige Ergebnisse besonders hoch. Daher bilden diese *Computersimulationen* ein Haupteinsatzgebiet von Großrechnern der höchsten Leisungsklasse, zu denen die Parallel- und Vektorrechner zählen. Oft versteht man diesen rein numerischen Zugang zu physikalischen Phänomenen heute als den alleinigen Gegenstand der *Computational Physics*.

## 1.2 Was denkt sich ein Computer?

Seit der industriellen Revolution hat sich der Mensch daran gewöhnt, im Alltag Maschinen zu bedienen. Der Siegeszug des Computers durch fast alle Lebensbereiche bringt jedoch die für viele immer noch unheimlich wirkende Notwendigkeit mit sich, eine Maschine nicht nur bedienen, sondern mit ihr auch kommunizieren zu müssen. Gemessen an den menschlichen Fähigkeiten Informationen aufzunehmen und zu verarbeiten wirken die der Maschinen, d.h., Computer auch heute noch recht schwerfällig. So muss man in der Regel Anweisungen in schriftlicher Form erteilen, Systeme, die auf gesprochene Kommandos in der gewünschten Form reagieren, sind noch immer eher die Ausnahme, wenn auch beeindruckende Fortschritte auf diesem Gebiet zu verzeichnen sind. Der Einsatz solcher Systeme ist auch eine Kostenfrage, und so bleibt es in der Regel bei den althergebrachten *Schnittstellen* zwischen Mensch und Maschine, einer Tastatur und einem Drucker oder einem Bildschirm.

Der Wunsch nach einer automatisch gesteuerten Rechenmaschine ist untrennbar mit der Notwendigeit verknüpft, eine Kommunikationsplattform, d.h., eine Sprache zwischen Mensch und Maschine zu vereinbaren. Nach den eher kläglichen Fähigkeiten der Maschinen, Sprache zu verstehen, sollte diese so einfach wie möglich und in schriftlicher Form gegeben sein.

### 1.2.1 Die binäre Welt.

Die Realisierung des Computers auf digitaler Basis ist von Vornherein nicht zwingend. Allerdings bietet die digitale Datenverarbeitung so viele technische Vorteile, dass andere Konzepte heute kaum noch eine Rolle spielen. Wir werden daher auf Alternativen, z.B. Analogrechner, überhaupt nicht eingehen.

Die Basis jeglicher Schrift ist ein Alphabet, und das einfachste denkbare Alphabet, das für Mitteilungen geeignet ist, besteht aus nur zwei verschiedenen Zeichen. Dieses Alphabet lässt sich technisch sehr einfach durch die zwei Stellungen eines Schalters *aus* - *ein* realisieren. Ein Speicher ist dann eine Reihe von

Schaltern, in der jeder durch seine Stellung anzeigt, welchen der beiden Buchstaben er gerade darstellt. Will man ein solches Alphabet wie das lateinische zum Schreiben von Texten verwenden, so muss man aber im Auge behalten, dass das Leerzeichen als Trenner zwischen zwei Worten als eigenständiges Zeichen mitgezählt werden muss. Es bleibt dann nur noch ein Buchstabe übrig, sodass die Bedeutung eines Wortes *nur* in der Anzahl seiner Buchstaben, des einzig übriggebliebenen, steckt. Im Prinzip spricht nichts dagegen, einen digitalen Computer auf der Basis dieses Alphabets zu realisieren. In der Praxis gibt es jedoch ein wesentliches Hindernis, nämlich die mit der Zahl der gewünschten Funktionen linear zunehmende Wortlänge. Mehr Buchstaben in das Alphabet aufzunehmen ist zwar ein Ausweg, aber dann müsste z.B. ein Speicherelement mehr als zwei Zustände annehmen können, die technisch attraktive weil einfache Realisierung durch Schalter wäre nicht mehr möglich. Für den Bau eines Digitalcomputers gibt es noch eine andere Möglichkeit, mit zwei Zeichen auszukommen, man interpretiert die beiden Zeichen als die Ziffern, 0 oder 1, einer *Zahl in Dualdarstellung* und verbindet die Bedeutung mit dem *Wert* dieser Zahl. Der Vorteil liegt auf der Hand, mit jeder zusätzlichen Stelle *verdoppelt* sich die Anzahl der darstellbaren Werte, also die Anzahl der verschiedenen Worte von vorbestimmter Länge. Die kleinstmögliche Informationsmenge steckt dann in einer einstelligen Dualzahl, 0 oder 1, und diese Informationsmenge nennt man ein *Bit*. Die ersten Überlegungen in diese Richtung stammen von dem Philosophen und Mathematiker Gottfried Wilhelm Leibniz (1646 - 1716).

Jede reelle Zahl $x$ lässt sich eindeutig in der Form

$$x = \sum_{k=-\infty}^{n} a_k B^k \tag{1.1}$$

mit positiv ganzzahligem $B$ schreiben, wo die Koeffizienten $a_k$ nur die Werte $0, 1, \ldots, B-1$ annehmen können. Für $B = 10$ erhält man das auf dem arabischen Kontinent erfundene *Dezimalsystem*, $B = 2$ liefert das *Dualsystem* oder *Binärsystem*, und $B = 16$ charakterisiert das *Hexadezimalsystem*. Allen diesen Zahlensystemen ist das Symbol für die unbesetzte Stelle, die *Null*, gemeinsam. Die Erfindung der Null ist in ihrer Bedeutung für die menschliche Kulturgeschichte kaum hoch genug einzuschätzen, denn erst mit ihr kann man beliebige Zahlen nicht nur darstellen, man kann in dieser Darstellung auch mit ihnen *rechnen*. Das in Europa lange Zeit gebräuchliche römische Zahlensystem ist für das Rechnen dagegen ungeeignet. Die Beiträge zu Gl.(1.1) für $k = -1, -2, \ldots$ werden üblicherweise durch einen Punkt von den Beiträgen für $k \geq 0$ getrennt. So hat z.B. die Zahl 11.01 im Binärsystem ($B = 2$) im Dezimalsystem ($B = 10$) den Wert $2 + 1 + \frac{0}{2} + \frac{1}{4} = 3.25$. Sind alle $a_k$ für negatives $k$ Null, dann ist die dargestellte Zahl eine *ganze Zahl*. Sind alle $a_k$ ab einem gewissen $k < 0$ Null oder ist die Folge der $a_k$ für $k < 0$ *periodisch*, dann ist die durch Gl.(1.1) dargestellte Zahl *rational*. Zum Beispiel entspricht die Zahl $0.101010\ldots$ im Dualsystem der Zahl $0.6666\ldots$

im Dezimalsystem, also $\frac{2}{3}$. Hat die Folge der $a_k$ für $k < 0$ keine dieser beiden Eigenschaften, dann ist die in Gl.(1.1) dargestellte Zahl *irrational*. Die Ziffern des Hexadezimalsystems werden mit den Ziffern $0 \ldots 9$ aus dem Dezimalsystem und mit den Buchstaben $A \ldots F$ für die fehlenden Symbole mit den Stellenwerten $10 \ldots 15$ dargestellt. Im Hexadezimalsystem ist also 3.25 (dezimal) gleich 3.4 und $\frac{2}{3}$ ist $0.AAAA\ldots$.

Für die Interpretation einer Bitfolge, also einer Dualzahl, als ganze Zahl wird das führende Bit manchmal als Vorzeichenbit angesehen. Ist es Null, so ist die Zahl positiv, ist es 1, so ist sie negativ. Das Muster lässt sich schon an einer dreistelligen Binärzahl demonstieren:

| dual | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| dezimal | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| mit Vorzeichen | 0 | 1 | 2 | 3 | -4 | -3 | -2 | -1 |

Die binäre Zahlendarstellung erlaubt also bei Bedarf auch ein Vorzeichen, ohne ein neues Zeichen einführen zu müssen. In heutigen Computern werden negative ganze Zahlen in der Tat so dargestellt, wie oben in der Tabelle für drei Bit gezeigt.

Ein Problem bleibt aber noch, für ein Leerzeichen zwischen zwei binären Worten oder Zahlen bietet unser Alphabet keinen Platz mehr. Wenn wir also eine mehrteilige Nachricht binär verschlüsseln wollen, dann gibt es keine Möglichkeit anzuzeigen, wo eine Nachricht endet und wo die nächste beginnt. Aus diesem Dilemma gibt es nur einen Ausweg, man muss ein für allemal festlegen, wie lang ein Wort unserer binären Sprache sein soll. Diese Entscheidung ist vor geraumer Zeit gefallen, ein Wort soll immer 8 Buchstaben, also 8 Bit lang sein. Ein Paket von 8 Bit hat daher einen eigenen Namen bekommen, es heißt ein *Byte*. Diese Festlegung wirkt bis heute nach, denn ein Byte ist die kleinste Datenmenge, die von einem Computer gelesen oder geschrieben werden kann. Auch wenn nur ein Bit geändert werden soll, der Computer muss (mindestens) das Byte, im dem das fragliche Bit steht, zuerst als ganzes einlesen und nach erfolgter Manipulation wieder zurückschreiben. Auf den ersten Blick erscheint die Wahl der Länge 8 recht kurz gegriffen, den mit 8 Bit kann man nur die Zahlen $0 \ldots 255$ oder $-128 \ldots 127$ darstellen, das erscheint als Umfang für den Wortschatz auch einer Maschine unzureichend zu sein. Das ist in der Tat so, und in der Praxis kann man dem dadurch abhelfen, dass zumindest in einigen Bytewerten die Anweisung steckt, zur Vervollständigung der Botschaft auch noch die folgenden ein, zwei oder mehr Byte einzulesen. Diese Maßnahme macht den Umfang der möglichen verschiedenen Botschaften praktisch unerschöpflich.

### 1.2.2 Computer und die Logik.

Die Entscheidung für die binäre Darstellung von Informationen im Inneren eines Computers mag noch immer nicht zwingend erscheinen. Der Erfolg der Stellenschreibweise für Zahlen liegt darin begründet, dass sich jede dieser Darstellungen,

gleich in welcher Basis, direkt für das Rechnen eignet. Ähnlich verhält es sich in unserem Fall: Man kann mit Zahlen in der Binärdarstellung nicht nur unmittelbar rechnen, mit der Binärdarstellung aller Informationen gerät der Computer in das allgemeine Regelwerk der sogenannten *Booleschen Algebra*.

Die Boolesche Algebra ist die von dem englischen Mathematiker George Boole im Jahre 1854 formulierte allgemeine mathematische Struktur für Mengen von Variablen $\mathcal{A} = \{a, b, c, \ldots\}$, die mit zwei kommutativen Verknüpfungen $\odot$ und $\oplus$ ausgestattet sind:

$$\begin{aligned} \forall a, b \in \mathcal{A} \quad &: \quad a \odot b = b \odot a \in \mathcal{A} \\ \forall a, b \in \mathcal{A} \quad &: \quad a \oplus b = b \oplus a \in \mathcal{A}. \end{aligned} \tag{1.2}$$

In $\mathcal{A}$ gibt es zwei spezielle Elemente 0 und 1 mit den Eigenschaften

$$\forall a \in \mathcal{A} : a \oplus 0 = a \quad \text{und} \quad a \odot 1 = a \tag{1.3}$$

Schließlich gibt es zu jedem $a \in \mathcal{A}$ ein *Komplement* $\overline{a}$ mit

$$\forall a \in \mathcal{A} : \exists \overline{a} \in \mathcal{A} : a \odot \overline{a} = 0 \quad \text{und} \quad a \oplus \overline{a} = 1. \tag{1.4}$$

Beide Verknüpfungen erfüllen das *Assoziativgesetz*

$$\begin{aligned} a \odot (b \odot c) = (a \odot b) \odot c = a \odot b \odot c \\ a \oplus (b \oplus c) = (a \oplus b) \oplus c = a \oplus b \oplus c \end{aligned} \tag{1.5}$$

und das *Distributivgesetz*

$$\begin{aligned} a \odot (b \oplus c) &= (a \odot b) \oplus (a \odot c) \\ a \oplus (b \odot c) &= (a \oplus b) \odot (a \oplus c). \end{aligned} \tag{1.6}$$

In diese Struktur gehören die *Mengenalgebra* und die *Aussagenlogik* mit den Verknüpfungen Durchschnitt $\cap$ und Vereinigung $\cup$ bzw. logisches *und* $\wedge$ und logisches *oder* $\vee$. Die Rolle des Komplements spielen das Mengenkomplement $\overline{X}$ bzw. die Negation $\neg x$. Die folgende Tabelle fasst die Gemeinsamkeitem noch einmal zusammen:

| | Mengenalgebra | Aussagenlogik | Boolsche Algebra |
|---|---|---|---|
| allg. Element | Teilmenge von $G$ | Aussage | Boolsche Variable |
| | $X \subset G$ | $x, y, \ldots$ | $a, b, \ldots$ |
| Nullelement | $\emptyset$ | 0 | 0 |
| Einselement | $G$ | 1 | 1 |
| Komplement | $\overline{X}$ | $\neg x$ | $\overline{a}$ |
| und | $X \cap Y$ | $x \wedge y$ | $a \odot b$ |
| oder | $X \cup Y$ | $x \vee y$ | $a \oplus b$ |

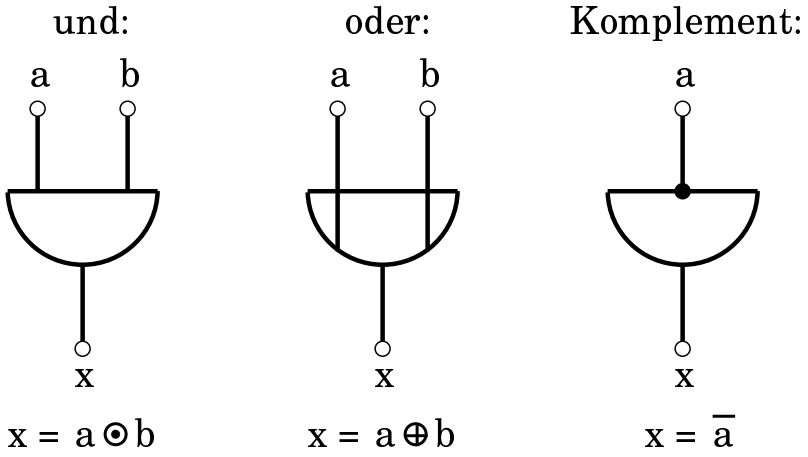
*Alle* denkbaren Operationen, die man mit in binärer Form gespeicherten Informationen ausführen kann, lassen sich durch die oben aufgelisteten elementaren Verknüpfungen der Booleschen Algebra darstellen. Ein digitaler Computer, der die binäre Darstellung von Informationen (Daten) benutzt, lässt sich also mit den Rechenregeln der Booleschen Algebra, niedergelegt in den Gl.(1.2) - (1.6), konstruieren.

Man kann sogar noch mit Hilfe der Komplementbildung die Verknüpfung $\odot$ durch $\oplus$ oder umgekehrt ausdrücken, denn aus den Eigenschaften Gl.(1.2) - (1.6) folgen die Komplementformeln

$$\begin{aligned} a \odot b &= \overline{\overline{a} \oplus \overline{b}} \\ a \oplus b &= \overline{\overline{a} \odot \overline{b}}. \end{aligned} \tag{1.7}$$

Damit sind *alle* Funktionen eines digitalen Computers, der die binäre Darstellung verwendet, durch das Komplement $\overline{a}$ und eine der beiden Verknüpfungen $\odot$ oder $\oplus$ für Boolesche Variable darstellbar.

Unabhängig von der technischen Realisierung der logischen Funktionen haben sich für ihre graphische Darstellung einige Schaltsymbole, die sogenannten *Gatter*, eingebürgert, die wir hier kurz vorstellen wollen.




Zur Abkürzung der Symbolik wird eine Komplementbildung an einem Ein- oder Ausgang eines $\odot$ oder $\oplus$ Gatters durch einen ausgefüllten Kreis angedeutet.

### 1.2.3 Der Computer lernt rechnen.

Jedes Bit in der binären Darstellung von Informationen ist eine Boolesche Variable, die nur zwei Werte annehmen kann, nämlich 0 und 1. Die Auswertung der Booleschen Verknüpfungen und des Komplements ist dann gemäß Gl.(1.3) und (1.4) vollständig durch eine kurze Tabelle gegeben, die man auch als *Wahrheitstabelle* bezeichnet, weil sie gleichzeitig die Wahrheitswerte von gemäß der Booleschen Algebra verknüpften Aussagen wiedergibt.

| $a$ | $b$ | $a \odot b$ | $a \oplus b$ | $\overline{a}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Die Funktionen unseres Computers lassen sich im Prinzip durch solche Wahrheitstafeln darstellen, wenn diese auch sehr groß werden würden. Wir wollen an wenigen Beispielen exemplarisch zeigen, wie mit Hilfe der Booleschen Algebra die Funktionen eines Computers aufgebaut werden können.

Unser Computer soll sicher in der Lage sein, die Summe zweier ganzer Zahlen zu bestimmen, addieren wir also erst einmal zwei einstellige Binärzahlen, d.h., zwei Bit. Die Aufgabe ist schnell gelöst:

| $x$ | $y$ | Übertrag $c$ | Summe $s$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Die Summe zweier Bits ist höchstens 2, in Binärdarstellung 10, benötigt also zwei Bit. Nach den Rechenregeln von Adam Riese bezeichen wir das fürende Bit als *Übertrag*, das hintere Bit ist die *Summe*. Es muss nun möglich sein, die oben angegebene Wahrheitstabelle aus $a$ und $b$ durch die beiden Verknüpfungen und das Komplement der Booleschen Algebra zu erzeugen. In der Tat, die Darstellung

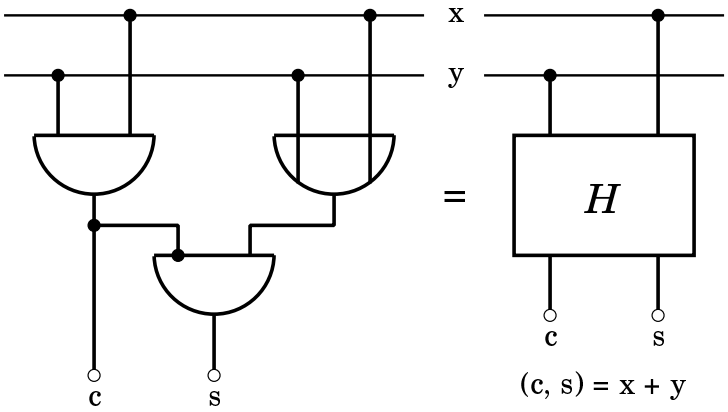
$$\begin{aligned} c &= x \odot y \\ s &= (\overline{x} \odot y) \oplus (x \odot \overline{y}) \end{aligned} \tag{1.8}$$

für Summe und Übertrag erfüllt diese Anforderung. Wir formen unser Ergebnis nach den Regeln Gl.(1.2) - (1.6) noch etwas um

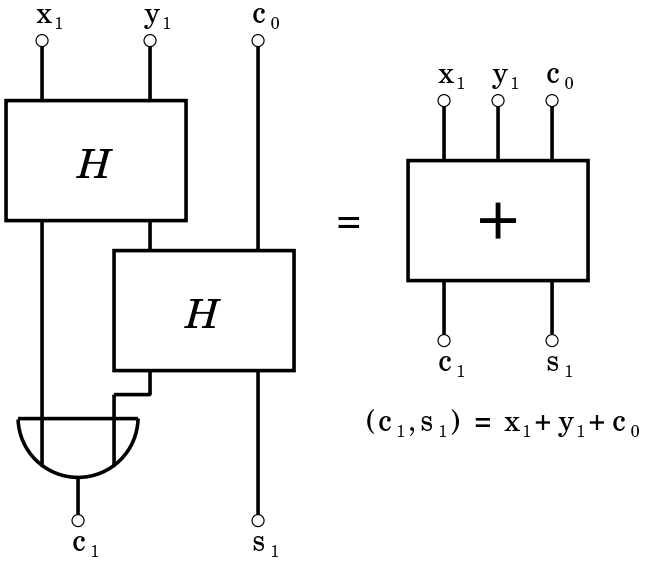
$$\begin{aligned} c &= x \odot y = \overline{\overline{x} \oplus \overline{y}} \\ s &= (\overline{x} \odot y) \oplus (x \odot \overline{y}) \\ &= \overline{x} \odot (x \oplus y) \oplus \overline{y} \odot (x \oplus y) \\ &= (\overline{x} \oplus \overline{y}) \odot (x \oplus y) \\ &= \overline{c} \odot (x \oplus y). \end{aligned} \tag{1.9}$$

Diese kleine Rechnung gestattet es, den mit $\odot$ bestimmten Übertrag $c$ für die Berechnung der Summe $s$ zu benutzen, was die Zahl der logischen Operationen reduziert. Die 150 Jahre alte Boolesche Algebra erlaubt nicht nur die Konstruktion eines digitalen Computers, sie reduziert auch noch die Kosten für dessen Bau, indem sie die Zahl der Gatter in der endgültigen Schaltung reduziert. Wir haben

mit Gl.(1.9) den sogenannten *Halbaddierer* konstruiert, und wir werden ihn im Folgenden durch ein eigenes Symbol $\boxed{H}$ darstellen. Das Symbol $c$ für den Übertrag stammt übrigens aus dem englischen Wort *carry*. Der Halbaddierer besitzt den unten dargestellten Aufbau, das Ersatzsymbol ist rechts daneben gezeigt.



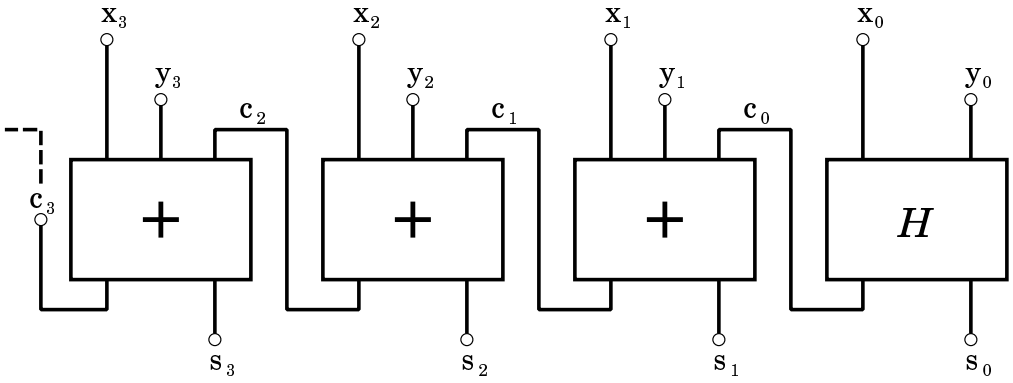

Nach den Regeln der Addition für mehrstellige Zahlen wird für die nächste Stelle der Summe auch der Übertrag aus der ersten Summation benötigt. Im Binärsystem reicht dazu der Halbaddierer nicht mehr aus, hier muss nämlich die Summe von *drei* Bit gebildet werden. Das kann man durch die Hintereinanderschaltung zweier Halbaddierer erreichen, wobei der zweite das Summenbit des ersten und das dritte Bit addiert. Die folgende Graphik zeigt das Schaltbild.




Für die Addition der beiden Überträge reicht ein $\oplus$ (*oder*) Gatter, weil mindestens einer der beiden Überträge Null ist. Es sei dem geneigten Leser überlassen, sich davon selbst zu überzeugen. Die Summe dreier Bit ist höchstens 3 (= 11 in

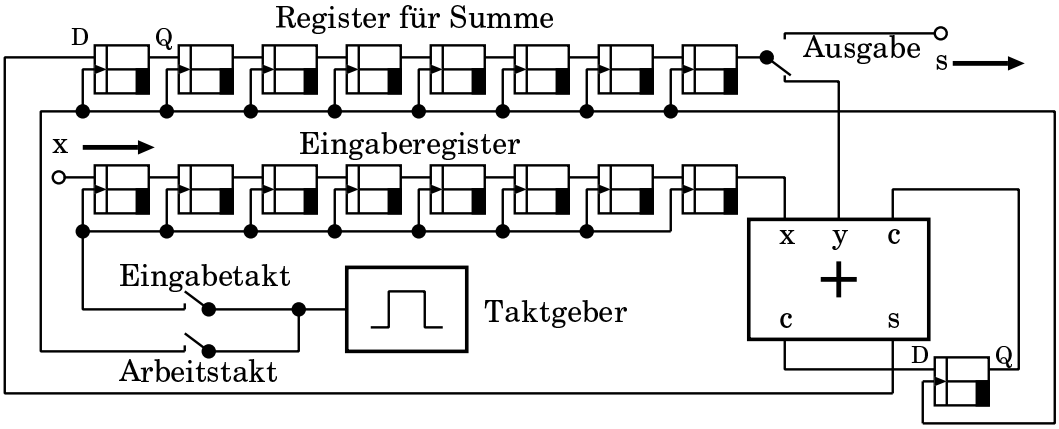
Binärsystem), also die größte zweistellige Binärzahl. Weil dieser erweiterte Addierer auch den Übertrag aus der vorherigen Stelle verarbeiten kann, hat er den Namen *Volladdierer* bekommen und wird ebenfalls mit einem eigenen Symbol $\boxed{+}$ dargestellt.

Wir sind jetzt in der Lage, die Früchte unserer Bemühungen zu ernten, denn die beiden Bausteine eines Addierwerks für beliebig lange Binärzahlen liegen jetzt vor uns. Die letzte Stelle der beiden Summanden verarbeitet ein Halbaddierer, für jede weitere Stelle zeichnet jeweils ein Volladdierer verantwortlich, der den Übertrag von der vorherigen Stelle direkt eingespeist bekommt. Wegen der charakteristischen Gestalt der Verbindungen zwischen den einzelnen Addierern hat die folgende Schaltung den Namen *Paddeladdierer* bekommen.




Der Paddeladdierer arbeitet *gleichzeitig* auf allen Bits der Summanden. Will man die Anzahl der Stellen erhöhen, dann muss das Addierwerk um weitere Volladdierer erweitert werden.

Ein *serieller* Addierer entält nur einen einzigen Volladdierer, dem über *Schieberegister* die Bits der Summanden *nacheinander* eingespeist werden. Ein Schieberegister besteht aus einer Reihe sogenannter *D-Flip-Flops*. Ein D-Flip-Flop ist eine Speicherzelle für ein Bit, das über einen Taktgeber aus dem Eingang D des Flip-Flop ausgelesen und in den Ausgang Q geschrieben wird. In einem Schieberegister ist der Ausgang Q eines D-Flip-Flop mit dem Eingang D des nächsten Flip-Flop verbunden, sodass die gespeicherte Bitfolge bei jedem Takt um eine Stelle weiterrückt. Auf diese Weise werden die Summanden Bit für Bit durch den Addierer geschoben und gleichzeitig rückt die Summe Bit für Bit in ein weiteres Register. Ein einzelnes D-Flip-Flop speichert den anfallenden Übertrag von einem Arbeitstakt für den nächsten. Die folgende Grafik zeigt das Prinzipschaltbild eines seriellen 8-Bit Addierers.




Man kann auch, wie in dem Prinzipschaltbild, eines der Eingaberergister gleichzeitig als Ausgaberegister verwenden. Der Vorteil des seriellen Addierers liegt darin, dass für eine Erhöhung der Zahl der Stellen nur die Register verlängert werden müssen. Bei unveränderter Taktfrequenz dauert eine Addition dann allerdings entsprechend länger. Für die Eingabe eines neuen Summanden x kann man den Arbeitstaktgeber abtrennen. Wir wollen die Digitaltechnik hier aber nicht weiter vertiefen, denn die technischen Details liegen außerhalb der Ziele dieses Kurses. Der serielle Addierer zeigt aber schon einige der technischen Merkmale von heutigen Mikroprozessoren.

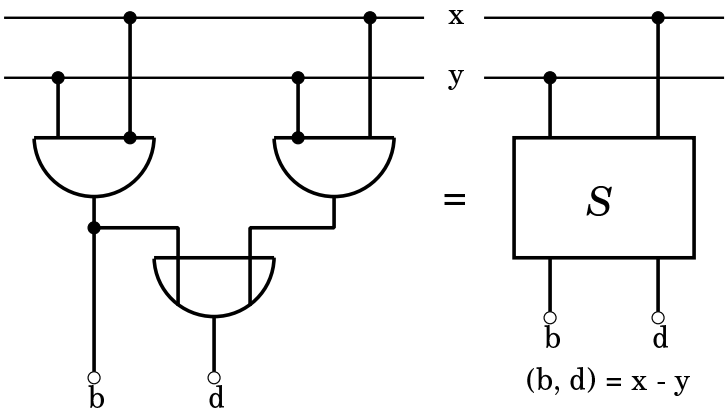
Ein Computer, der etwas auf sich hält, sollte auch subtrahieren können. Die Darstellung negativer ganzer Zahlen im Binärsystem folgt dem weiter oben schon besprochenen Muster. Wie für die Addition bestimmt man zuerst die Differenz zweier einstelliger Binärzahlen über die Wahrheitstabelle

| $x$ | $y$ | "Bürgschaft" $b$ | Differenz $d$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

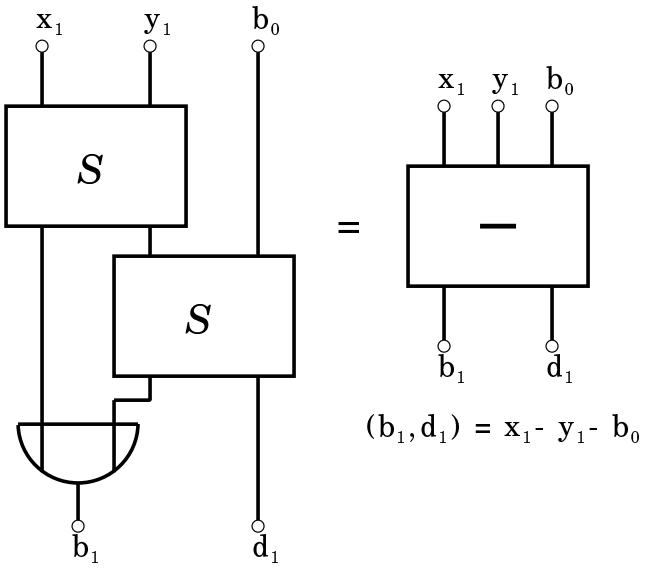
wo $b$ diesmal das von der nächsten Stelle "geborgte" (engl. *borrow*) Bit anzeigt. Nach den Regeln der Booleschen Algebra erzeugen die einfachen Ausdrücke

$$\begin{aligned} b &= \overline{x} \odot y \\ d &= b \oplus (x \odot \overline{y}) \end{aligned} \tag{1.10}$$

die richtigen Werte für die Differenz und das geborgte Bit. Anders als Gl.(1.8) für der Halbaddierer ist Gl.(1.10) bereits optimal vereinfacht, was der Vergleich mit Gl.(1.9) schon zeigt. Mit Gl.(1.10) erhalten wir das Schaltbild




das den sogenannten *Halbsubtrahierer* $\boxed{S}$ realisiert. Für die Bildung der Differenz mehrstelliger Binärzahlen muss man ab der zweiten Stelle die hinterlassene "Bürgschaft" der jeweils vorherigen Subtraktion übernehmen, d.h., das geborgte Bit muss von der Differenz der gerade verarbeiteten Stellen auch noch abgezogen werden. Eine Schaltung, die das leistet, wird naheliegenderweise als *Vollsubtrahierer* $\boxed{-}$ bezeichnet, und ihren Aufbau aus dem Halbsubtrahierer kann man beihnahe erraten. Sie ist die präzise Analogie des Volladdierers, wie die folgende Grafik zeigt.




Wie die beiden Überträge innerhalb des Volladdierers sind auch die beiden geborgten Bits innerhalb des Vollsubtrahierers niemals gleichzeitig gleich eins. Die Konstruktion eines Subtrahierwerks ist damit klar, man folgt entweder dem Muster des Paddeladdierers oder dem des seriellen Addierers mit Schieberegistern als Zwischenspeicher, es sind lediglich die Addierer durch die entsprechenden Subtrahierer zu ersetzen. Die Prinzipschaltbilder bieten also nichts Neues mehr, wir wollen daher hier auf sie verzichten.

Die Multiplikation mit der Basis $B$ der Zahlendarstellung ist immer besonders einfach, man schiebt die Ziffernfolge um eine Stelle nach links und schreibt in die letzte freiwerdende Stelle eine Null. Genau das tut ein Schieberegister mit einer Binärzahl bei einem Arbeitstakt, wenn der Dateneingang des letzten D-Flip-Flops auf Null gehalten wird. Für die Multiplikation mit $2^n$ muss man also nur die Bitfolge der zu multiplizierenden Zahl um $n$ Stellen nach links verschieben und die freiwerdenden Stellen mit Nullen belegen. Aus diesen Schiebeoperationen und nachfolgenden Additionen kann man also frei nach Adam Riese einen Multiplizierer aufbauen. Ein kleines Beispiel zeigt das Prinzip für 4 Bit lange Binärzahlen

| 1 | 0 | 1 | 1 | · | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| + | | | 0 | 0 | 0 | 0 | 0 | 0 |
| + | | | | 1 | 1 | 1 | 0 | 0 |
| + | | | | | 1 | 1 | 1 | 0 |
| = | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |

oder anders ausgedrückt, $11 \cdot 14 = 154$. Weitere technische Details wollen wir uns hier ersparen.

Dem Computer fehlt noch die sehr wichtige Fähigkeit, Zahlenwerte vergleichen zu können. Es sind mehrere technische Lösungen denkbar, wir wollen uns hier mit der Feststellung begnügen, dass die Bildung der Differenz die Lösung dieses Problems mitliefert. Ein gesetztes Vorzeichenbit (das führende Bit der Bitfolge) zeigt eine negative Differenz an, also war das erste Argument echt kleiner als das zweite. Gleichheit zweier Zahlen ist gleichbedeutend mit bitweiser Gleichheit, und die wird z.B. durch das Ergebnis 0 im Summenbit eines Halbaddierers angezeigt. Damit haben wir die technischen Lösungen für Vergleiche ebenfalls schon in der Hand, um Optimierungsfragen wollen wir uns hier nicht kümmern. Eigentlich hat der binäre Computer das Rechnen gar nicht mehr lernen müssen, die Boolesche Algebra hat ihm diese Fähigkeit schon in die Wiege gelegt.

### 1.2.4 Der Computer und seine "Gedanken"

Auf der Basis der logischen Schaltungen, die dem Computer das Rechnen, also die Verarbeitung von Daten erlauben, ist auch die Steuerung des Computers realisierbar. Die elementaren bitweisen Operationen, mit denen wir Addition oder Multiplikation realisieren können, sind in modernen Mikroprozessoren der Inhalt kleiner Programme, dem sogenannten *Mikrocode*, die hinter den Maschinenbefehlen für Addition und Multiplikation zweier Zahlen stecken. Programmierbarkeit von Funktionen bedeutet letztlich, dass die logischen Verbindungen zwischen den Registern, dem Rechenwerk und dem Taktgeber nicht fest "verdrahtet" sind, sondern durch Anweisungen, die selbst wieder nichts anderes als Bitmuster sind, von außen geknüpft und wieder gelöst werden können. In dem Prinzipschaltbild des

seriellen Addierers ist diese Möglichkeit bereits angedeutet. Die drei Schalter *Eingabetakt*, *Arbeitstakt* und *Ausgabe* aktivieren die drei Funktionen des Addierers: 1. Lade einen Wert x in das Eingaberegister, 2. Addiere x zum Inhalt des Registers für die Summe und 3. Schreibe das Ergebnis in den Ausgang s. Der *Mikrocode* für eine Multiplikation zweier Zahlen mit Hilfe des seriellen Addierers besteht im Wesentlichen aus einer abwechselnden Ausführung der Funktionen 1 und 2 des Addierers bis alle Stellen des Multiplikanden abgearbeitet sind, also bis zur Länge des Registers, dann wird Funktion 3 ausgeführt.

Die innersten "Gedanken" des Computers sind also die Anweisungen des Mikrocodes für eine Multiplikation oder eine andere Operation. Wie oben schon vorweggenommen ist eine Operation mit zwei Zahlen in einem *Maschinenbefehl* zusammengefasst, der die Abarbeitung des dazu gehörenden Mikrocodes auslöst. Auf der Maschinenebene kann der Mensch als Anwender des Computers, d.h., als *Programmierer* frühestens eingreifen. Eine Rechnung, die er ausgeführt haben möchte, muss er hier in der *Sprache* der Maschinenbefehle als Anweisungsfolge, d.h., als *Maschinenprogramm* formulieren. Dazu gehören wie oben schon angedeutet Operationen mit den Daten (Zahlen) und Operationen zum Lesen und Schreiben von Daten aus dem Speicher in die Register oder umgekehrt. Jede seiner Anweisungen ist dabei selbst wieder ein kleines Programm aus dem Mikrocode. Die Programmierung in Maschinensprache, auch *Assembler* genannt, stellt dem Programmierer zwar die volle Leistung seines Computers zur Verfügung, aber dafür ist er auch zu jedem Zeitpunkt für das Verhalten des Computers verantwortlich, ein Fehler in seinem Programm hat unmittelbare Folgen für die gesamte Anlage. Ausserdem ist er für die Übersetzung seiner Rechenergebnisse in eine vom Menschen wieder lesbare Form zuständig und deren Übertragung z.B. auf einen Drucker oder in eine Datei auf einem Band- oder Festplattenlaufwerk. All dies muss ein Teil des Maschinenprogramms sein, wenn man die Rechenergebnisse nicht nur erzeugen, sondern auch selbst zu Gesicht bekommen will. Vieles davon, wie die Steuerung von Druckern oder Festplattenlaufwerken, sollte der Computer aber ständig und vor allem selbstständig ausführen können, weil diese Geräte erstens immer wieder benötigt werden und zweitens natürlich auf immer wieder die gleiche Weise gesteuert werden müssen. Das erzeugt natürlich ein starkes Verlangen nach einer neuen, über der Maschinensprache liegenden Ebene der Kommunikation mit dem Computer, innerhalb derer die in Maschinensprache so aufwendigen Funktionen, wie das Ausdrucken von Rechenergebnissen auf einem Drucker durch einen einfachen Befehl zu bewerkstelligen sind, so einfach wie der Befehl für die Multiplikation zweier Zahlen in Maschinensprache statt im Mikrocode. Es sind zwei Dinge, die sich an dieser Stelle aufdrängen. Erstens liegt es sehr nahe, die Programmierung und Steuerung eines Computers *schichtweise* aufzubauen. Zweitens möchte man die Steuerung des Computers und evt. angeschlossener Geräte am liebsten dem Computer selbst überlassen.

Die unterste Schicht der Programmierung bildet der Mikrocode des Prozessors. Die nächste Schicht ist die Maschinensprache, in der z.B. die Programme

für die Steuerung der Peripheriegeräte, die sogenannten *Treiber* vorliegen. Auf der gleichen Ebene befindet sich das Programm, das den Computer verwaltet, steuert und auf Anfrage über die Treiber Zugriff auf die Geräte wie Festplattenlaufwerk, Drucker oder Bildschirm gewährt, das sogenannte *Betriebssystem*. Das Betriebssystem bildet gleichzeitig die nächst höhere Ebene für den Programmierer, auf der er bequem Daten ausdrucken, Dateien anlegen und mit Daten füllen, Daten auf dem Bildschirm darstellen, und eigene Programme erstellen und ausführen lassen kann. Das geschieht durch Eingabe von Kommandos auf der sogennanten *Kommandozeile*, die das Betriebssystem als Plattform für die Kommunikation mit dem Computer, als *Schnittstelle* für den Benutzer, zur Verfügung stellt. Moderne Betriebssysteme halten eine noch höhere Ebene bereit, die man als graphische Benutzeroberfläche bezeichnet. Die bekanntesten unter diesen sind die Betriebssysteme der Windows Familie und das X-Windows System für das Betriebssystem UNIX. Damit ist die Ebene der Anwenderprogramme erreicht. Noch eine Stufe höher führen spezielle Anwenderprogramme, die manchmal als *Emulatoren* bezeichnet werden und den Computer als ein ganz neues System mit einer neuen Programmiersprache, als eine sogenannte *virtuelle Maschine*, erscheinen lassen. Emulatoren können zum Beispiel einen bequemen Zugriff auf die Elemente der graphischen Oberfläche ermöglichen, wie das etwa bei der Programmiersprache JAVA der Fall ist. Das JAVA System *emuliert* eine Art Standardrechner, der unabhängig von den tatsächlich vorhandenen technischen Komponenten des Computers immer auf die gleiche Weise programmiert werden kann. Ein auf diesem System entwickeltes Programm ist auf jedem anderern Computer ausführbar, der die virtuelle Maschine zur Verfügung stellt. Damit ist das Prinzip der Programmierung quasi umgekehrt, statt einer maschinenspezifischen Sprache auf Prozessorniveau eine universelle überall einsetzbare Sprache auf einer Ebene oberhalb gewöhnlicher Anwenderprogramme. In der Praxis freilich ist die Lage nicht ganz so ideal, denn die Standards solcher Sprachen müssen auch umgesetzt, d.h., von kommerziellen Softwareanbietern akzeptiert und in ihren Produkten realisiert werden. Manchmal schaffen herstellerspezifische Abweichungen von den Standards aber neue Probleme für den Programmierer. Frei verfügbare Alternativen werden manchmal von Privatleuten mit zwangsläufig beschränkten Mitteln gepflegt, sodass nicht immer alles realisiert ist. Für kommerzielle Projekte werden in der Regel Lizenzen für die Benutzung kommerzieller Softwarepakete erworben, weil man damit bei Bedarf auf Unterstützungsleistungen des Herstellers zurückgreifen kann. Frei verfügbare Software enthält diesen Service im Allgemeinen nicht. Außerdem muss man sich als Anwender damit abfinden, dass *fehlerfreie* Softwarepakete, seien sie frei oder kommerziell verfügbar, ab einer gewissen Größe praktisch nicht mehr existieren.

Was also denkt sich nun ein Computer? Wie wir gesehen haben, sind die Gedanken vielschichtig, aber es sind natürlich nicht die eigenen. Ohne die gedankliche Vorarbeit eines Programmierers ist jeder Computer nichts als eine leere Hülle. Auf welchem Niveau der Programmierer auch immer mit dem Computer umgeht,

letztlich gilt immer:

*Der Computer tut genau das, was man ihm sagt.*

# Kapitel 2

# Geschichte und Aufbau des Computers

Computer können so aufgebaut sein, dass sie entweder mit kontinuierlichen Spannungswerten arbeiten oder aber allein mit den zwei möglichen Zuständen „Spannung“ (typischerweise 5 V) oder „keine Spannung“. Während man im ersten Fall von Analogrechnern spricht, die v.a. in der Regelungstechnik eingesetzt werden, bezeichnet man die zweite Möglichkeit als digital. Dieses Skript wird ausschließlich von Digitalcomputern handeln.

## 2.1 Geschichtliche Entwicklung

Bereits vor Beginn des 20. Jahrhunderts wurde mit mechanischen Rechenmaschinen gearbeitet. Diese waren jedoch nicht programmierbar und auf Grund ihrer mechanischen Konstruktion für heutige Begriffe extrem langsam. Erst durch Nutzung der Elektrizität wurden hier wesentliche Fortschritte erzielt.

**1944** Conrad Zuse und IBM entwickeln unabhängig voneinander durch Lochstreifen programmierbare Relaisrechner (Taktfrequenz ca. 200 Hz).

**1945** ENIAC, ein Röhrenrechner mit etwa 8000 Hz Taktfrequenz.

Parallel zu dieser Entwicklung gab es eine zweite auf dem Gebiet der Mathematik:

**1931** Gödel entwickelt den Begriff des Algorithmus.

**1936** Turing entwirft einen Zellularautomaten („Turing-Maschine“), welcher allgemeine Rechnungen durchführen kann.

**1951** von Neuman: Konzept des Universalrechners

Die Röhrentechnik der ENIAC, die immer noch lediglich eine programmierbare Rechenmaschine war, ermöglichte die praktische Umsetzung der Theorie von

Neumans. Das Ergebnis war 1951 die UNIVAC I, der erste real existierende Universalrechner.

## 2.2 Aufbau und Funktionsweise von Universalrechnern

Ein Universalrechner besteht im Wesentlichen aus zwei Komponenten, dem Prozessor (*Central Processing unit*, kurz *CPU*) und dem (Arbeits-)Speicher. Der Prozessor wiederum gliedert sich auf in Leitwerk und Rechenwerk. Diese grundsätzliche Struktur zeigt Abbildung 2.1, die Details werden in den nächsten Unterabschnitten erläutert.

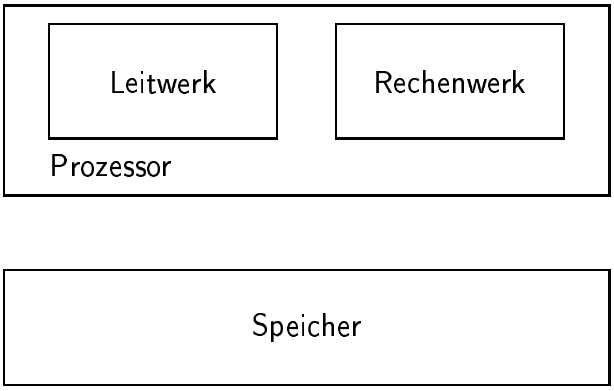



Abbildung 2.1: Aufbau eines Universalrechners

### 2.2.1 Rechenwerk

Das Rechenwerk setzt sich aus zahlreichen einzelnen Schaltungen zusammen, die festverdrahtet jeweils bestimmte arithmetische oder logische Operationen durchführen. Die Daten, auf die diese Operationen angewandt werden, werden in internen Speicherzellen mit höchster Zugriffsgeschwindigkeit, sog. Registern, zwischengespeichert und in ebensolche ausgegeben.

Die Datenein- und ausgabe in die Register des Rechenwerks erfolgt über einen Datenbus (Strang paralleler Leitungen zur gleichzeitigen Übertragung aller Bits eines Datenwortes). Die Information darüber, welche der möglichen Operationen jeweils ausgeführt werden soll, wird über einen (i. d. R. schmaleren) Kontrollbus vom Dekoder des Leitwerks übermittelt. Dies ist in Abbildung 2.2 dargestellt.

Beispiele für die Aufgaben der Schaltungen des Rechenwerks sind:

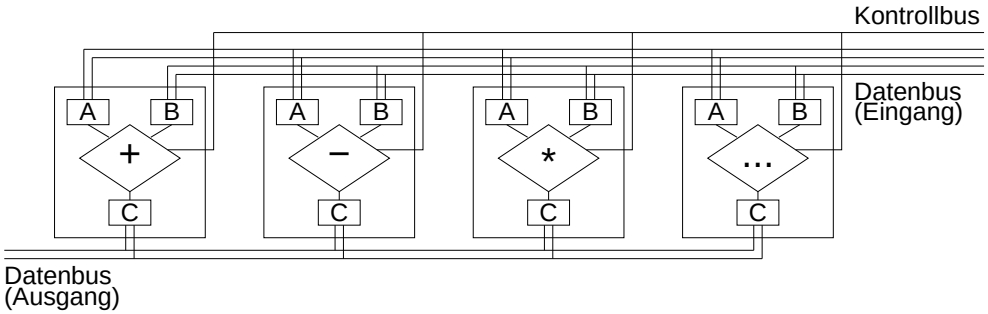



Abbildung 2.2: Aufbau des Rechenwerks

- arithmetische Operationen wie Addition oder Division,
- logische Operationen, z.B. bitweises UND oder bitweise Negation,
- Verschieben der Bitwerte in einem Wort (*Bitshift*),
- Zählen der Bits mit Wert 1 in einem Wort.

Das Rechenwerk wird oft in zwei Einheiten unterteilt, die *ALU* (*Arithmetic Logical Unit*), die arithmetische Operationen ganzer Zahlen sowie logische Operationen durchführt, und die *FPU* (*Floating Point Unit*), die arithmetische Operationen mit Gleitkommazahlen ausführt. Letztere sind ungleich komplexer (und zeitaufwändiger) als vergleichbare Ganzzahl-Operationen.

### 2.2.2 Leitwerk

Das eben beschriebene Rechenwerk führt die Operationen aus, die von einem Computer erwartet werden, verrichtet also die eigentliche „Arbeit“. Zur Steuerung benötigt es jedoch eine weitere Einheit, die je nach Erfordernissen die einzelnen Schaltungen aktiviert, ihnen die zu bearbeitenden Informationen bereitstellt und dafür verantwortlich ist, dass der Computer Programme, also Aufzählungen einzelner Anweisungen, abarbeiten kann. Für alle diese Aufgaben ist das Leitwerk zuständig. Zum besseren Verständnis sollen im Folgenden die typischen Abläufe im Leitwerk während der Ausführung eines Programms dargestellt werden. Diese können in Abbildung 2.3 verfolgt werden.

Jedes Programm besteht aus einer Liste von Maschinenbefehlen, von denen wiederum jeder aus einer bestimmten Zahl von Bits (einem Befehlswort) besteht. Damit diese überhaupt ausgeführt werden können, müssen sie zuerst vom Betriebssystem in den Arbeitsspeicher (RAM) geladen worden sein. Obwohl dieser bereits sehr schnell lesbar ist, wird die Geschwindigkeit weiter dadurch erhöht, dass das Leitwerk die Befehle nicht direkt aus dem RAM liest, sondern die Programmteile, die gerade benötigt werden, in einen kleineren und schnelleren Zwischenspeicher, den Cache, kopiert und aus diesem heraus abarbeitet.

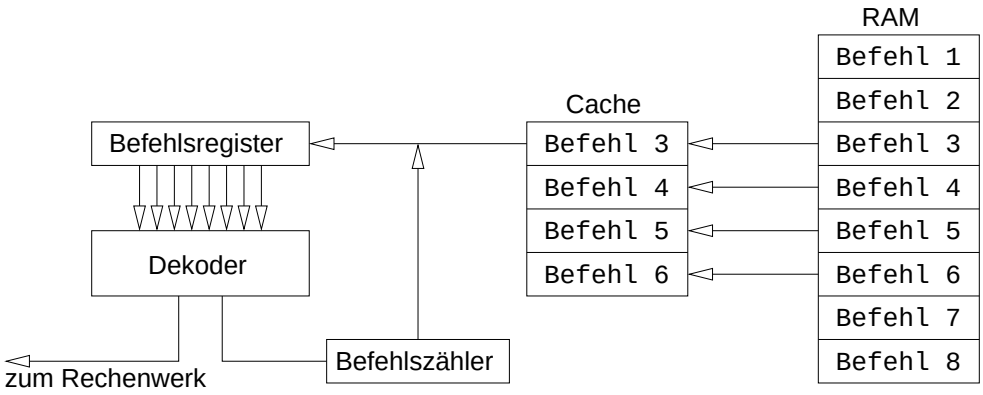



Abbildung 2.3: Aufbau des Leitwerks

Welcher Befehl gerade aktuell ist, d. h., gelesen werden soll, steht in einem sog. Befehlszähler. Der gerade aktuelle Befehl wird nun aus dem Cache in ein spezielles Befehlsregister gelesen. Der unmittelbar an dieses angeschlossene Dekoder interpretiert das Bitmuster des Befehls entsprechend seiner Bedeutung: Einzelne Gruppen von Bits kodieren die auszuführende Operation im Rechenwerk, andere geben die Adresse an, woher die Eingabedaten für diese Operationen zu beziehen sind, oder stellen selbst einen zu verwendenden Datenwert dar. Andere Bits kontrollieren das Leitwerk selbst, wie z. B. Programmschleifen oder Interrupts. Der Dekoder sendet je nach Befehlswort diese Daten über einen Datenbus an das Rechenwerk und gibt diesem über einen Kontrollbus bekannt, welche Operation damit ausgeführt werden soll.

Anschließend muss der Befehlszähler die Adresse des gerade dekodierten Befehlswortes um eins erhöhen, sodass als nächstes der unmittelbar folgende Befehl aus dem Cache gelesen und dekodiert wird. Damit bestimmte Befehlssequenzen wiederholt ausgeführt werden können, gibt es auch Befehle, die nicht an das Rechenwerk weitergegeben werden, sondern im Leitwerk selbst abgearbeitet den Befehlszähler verändern.

### 2.2.3 Takt

Um den Prozessor sinnvoll nutzen zu können, ist es notwendig, ihm einen bestimmten Takt zu geben, in dem die einzelnen Befehle ausgeführt werden. Je schneller man diesen Arbeitstakt wählt, je höher also die Taktfrequenz ist, desto schneller arbeitet der Prozessor. Hier sind allerdings durch die elektronischen Schaltungen Grenzen gesetzt, da die einzelnen Bauteile gewisse Zeiten benötigen, um sich auf eine Spannungsänderung von 0 auf 1 (oder umgekehrt) einzustellen und zudem die Wärmeentwicklung mit zunehmender Taktfrequenz steigt.

Die Taktfrequenz $f$ gibt die Anzahl der Taktimpulse pro Zeiteinheit an. Den Kehrwert davon bezeichnet man als *cycle time* $t_{cy}$, er gibt die Zeitdauer zwischen

zwei Taktimpulsen an.

$$t_{cy} = \frac{1}{f} \tag{2.1}$$

Die Taktfrequenz heutiger Prozessoren bewegt sich im Bereich von einigen Gigahertz (GHz), die zugehörigen Taktdauern $t_{cy}$ betragen damit Nanosekunden (ns).

Beispiele für den typischen Zeitbedarf verschiedener Operationen sind

- logische Operationen: 1 $t_{cy}$,
- bitweises Verschieben: 2 $t_{cy}$,
- Ganzzahladdition: 3–4 $t_{cy}$,
- Ganzzahlmultiplikation: 7 $t_{cy}$,
- einfache Gleitkommaoperationen: 10–14 $t_{cy}$.

Oft ist es von Interesse, die Geschwindigkeit eines Computers quantitativ zu beschreiben oder aber mit der eines anderen zu vergleichen. Dafür gibt es verschiedene Möglichkeiten:

### Taktzeit $t_{cy}$

Der einfachste Vergleichsmaßstab ist die Dauer eines Taktzyklus bzw. die Taktfrequenz. Dies ist zugleich aber auch der am wenigsten aussagekräftige, da keinerlei Angabe über die Anzahl an Taktzyklen gemacht wird, die die verschiedenen Operationen zur Ausführung benötigen.

### *Floating Point Operations per Second* (kurz *Flop*)

Die Bewertung eines Prozessors in Flop gibt an, wie viele Gleitkommaoperationen dieser pro Sekunde durchführen kann. Zur Bestimmung verwendet man ein Testprogramm, das typischerweise aus 30% Gleitkommamultiplikationen und 70% Gleitkommaadditionen besteht.

Dieser Maßstab gibt zwar im Gegensatz zur Taktfrequenz die tatsächliche (Gleitkomma-)„Rechenleistung“ einer Maschine an, für den Praxisnutzen besteht jedoch die Einschränkung, dass reale Programme abhängig vom Einsatzzweck zu unterschiedlich hohem Anteil aus Gleitkommaoperationen bestehen, in keinem Fall aber zu 100%.

Beispiele für Rechenleistung gemessen in Flop sind:

- Großrechner „T3E“ in München: 1,2 TFlop,
- PC: weniger als 200 MFlop.

***Million Instructions per Second* (kurz *Mips*)**

Ähnlich wie Flop ist die Einheit Mips definiert. Hier wird jedoch nicht die Anzahl der Gleitkommaoperationen pro Sekunde gezählt, sondern die beliebiger Operationen. Da Computerhersteller ein verständliches Interesse daran haben, ihre Produkte als möglichst leistungsfähig darzustellen, wird als Referenzoperation i.d.R. die „billigste“ Anweisung des Befehlssatzes gewählt, also diejenige, die am wenigsten Taktzyklen benötigt. Typischerweise handelt es sich dabei um eine logische Operation mit einer Dauer von 1 $t_{cy}$.

***LINPACK***

Eine Alternative zu den vorher genannten eher theoretischen Maßstäben ist es, in einem Testprogramm typische Arbeitssituationen möglichst realistisch nachzubilden und dann die Geschwindigkeit zu bestimmen, die ein Computer unter diesen Bedingungen hat. Mit Hilfe dieser Methode kann man relativ genau feststellen, welcher Computer für eine bestimmte Art von Aufgaben in der Praxis am besten geeignet ist. Als Beispiel eines Testprogramms sei hier LINPACK genannt, das die Anforderungen der gängigsten Programme aus dem Gebiet der linearen Algebra an einen Computer nachstellt.

### 2.2.4 Multiprozessorarchitektur

Damit umfangreiche Prozesse von Computern zügig bearbeitet werden können, arbeiten in vielen Rechnern mehrere Prozessoren gleichzeitig. Hierbei gibt es zwei unterschiedliche Ansätze der Arbeitsteilung zwischen den Prozessoren, die Vektor- und die Parallelarchitekturen.

- Im Bezug auf Vektorarchitekturen wird auch von Pipelining gesprochen, da wie an einem Fließband mehrere verschiedene Teiloperationen an unterschiedlichen Daten direkt hintereinander ausgeführt werden. Ohne Pipelining würde vor Beginn mit dem Zweiten, die Verarbeitung des ersten Datensatzes abgeschlossen. Der Vorteil der Vektorarchitekturen besteht darin, dass mehr Verbindungskabel gleichzeitig genutzt werden. Die gesteigerte Rechnerleistung wird in Abbildung 2.4 am Beispiel von Additionen deutlich. Derartige Vektorarchitekturen finden Anwendung in Großrechnern wie NEC und YMP.

- Bei den parallelen Rechnerarchitekturen wird zwischen *distributed* und *sharedmemory* unterschieden. Im ersten Fall besitzt jeder Prozessor einen ihm zugeordneten Speicher und führt, wie in Abbildung 2.5 gezeigt, mit diesem die ihm zugeordneten Operationen aus. Der Großrechner $T3E$ arbeitet nach diesem Prinzip. Es wird ineffizient, sobald ein Prozessor oft auf die Daten eines fremden Speichers zugreifen muss.

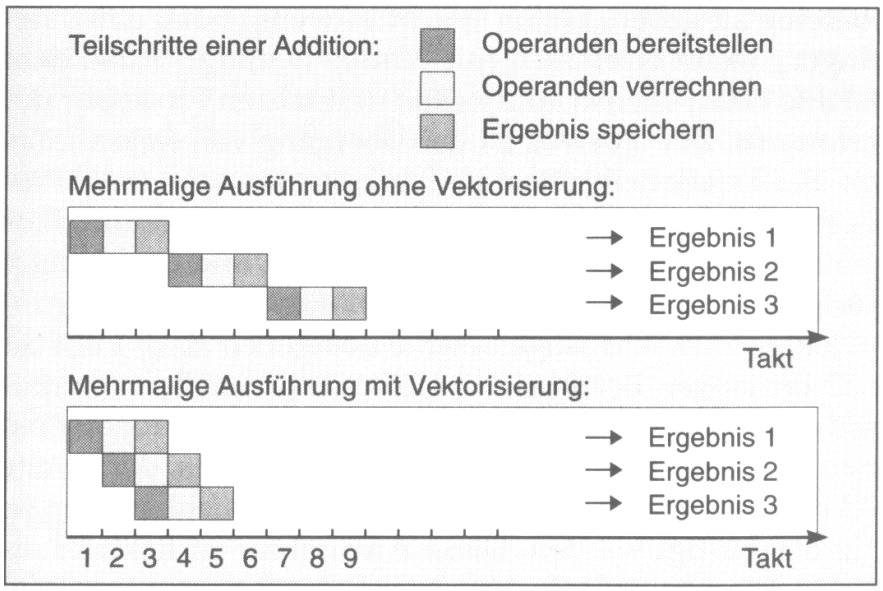



Abbildung 2.4: Schematische Darstellung des Fließbandprinzips ([1], S. 65)

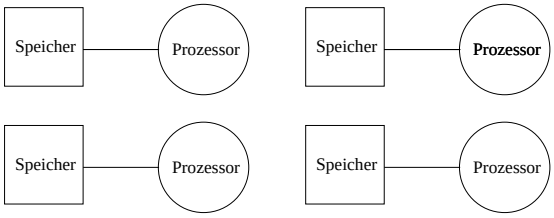



Abbildung 2.5: Schematischer Aufbau eines parallel Rechners mit zerteiltem Speicher

In Abbildung 2.6 ist der Aufbau von Parallelarchitekturen nach dem Prinzip des shared memory dargestellt. Alle Prozesoren sind um einen großen gemeinsamen Massenspeicher herum angeordnet und greifen auf diesen zu. Hier kommt es zu Verzögerungen, sobald mehrere Prozessoren auf den gleichen Speicherplatz zugreifen müssen.

Außerdem unterscheidet man zwei Arten der Parallelisierung: MIMD und SIMD. Im ersten Fall wird von jedem Prozessor eine andere Arbeit verrichtet, bei der nicht die gleichen Daten benötigt werden. Im anderen Fall, dem SIMD wird auch von farming gesprochen, da jeder Prozessor zwar die gleiche Arbeit verrichtet, aber jeweils mit unterschiedlichen Datensätzen. Dies ist selbstverständlich nicht bei allen Problemen anwendbar, führt aber zu guten Rechnerauslastungen.

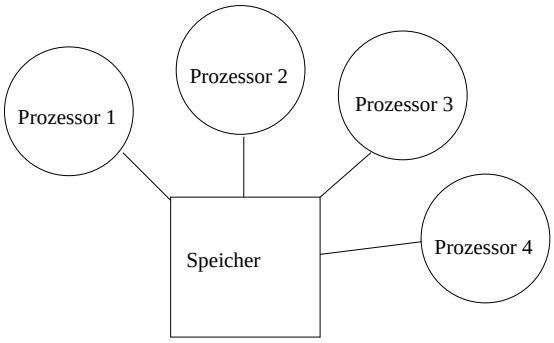



Abbildung 2.6: Schematischer Aufbau eines parallel Rechners einheitlichem Speicher

### 2.2.5 Speicher

Im Speicher werden Daten und Programme als Dateien gespeichert. Diesen Dateien sind Namen, Header (Dateitypen wie z.B. txt, ps, exe) und Attribute wie das Erstellungsdatum zugeordnet. Diese Informationen werden in UNIX mit dem Befehl `ls -l` angezeigt.

#### Interne Speicher

Grundsätzlich gibt es zwei Arten von Speichern in einem Rechner, den ROM (read only memory) und den RAM (random access memory):

- Der ROM kann nur gelesen werden, auf ihm können keine Dateien gespeichert werden. Seine Speicherkapazität liegt üblicherweise im kB-Bereich, was für das Boot-Programm auch völlig ausreichend ist. ROM hat im Allgemeinen sehr kurze Zugriffszeiten.

- Der RAM kann sowohl gelesen, als auch beschrieben werden. Er zeichnet sich durch Auslesezeiten für Dateien im Bereich von 10-40ns aus. Aufgebaut ist der RAM aus Kondensatoren, deren Ladungszustände verändert werden können. Da diese allerdings schnell ihre Ladung und damit die gespeicherte Information verlieren, müssen sie mehrmals pro Sekunde neu aufgeladen werden. Nach Abschalten des Computers gehen folglich alle im RAM gespeicherten Dateien verloren.

#### Aufbau und Funktionsweise interner Speicher

Die kleinste Einheit elektronischer Daten sind die Bits, sie enthalten die Informationen 'Strom' oder 'kein Strom'. Die Zustände werden oft mit 1 und 0 angegeben.

Acht dieser einfachen Schalter werden im Speicher zu einem Byte zusammengefasst. Die nächst größere Einheit ist ein Wort, das wiederum vier oder acht Byte umfasst. Jedem dieser Wörter, die zusammen einen Speicherplatz bilden, wird im Rechner eine eigene Adresse zugeordnet.
Wenn ein Programm auf bestimmte Daten zugreift, so sendet es zuerst über einen aus vielen parallelen Leitungen bestehenden Adressbus die Adresse des benötigten Wortes binärcodiert an den Multiplexer. Dieser verbindet den Datenbus anhand der Adresse mit dem Wort, in welchem die benötigten Informationen in Bits gespeichert sind. Im Falle des Random Access Memory kommt außerdem vom Leitwerk die Anweisung in den Speicher, ob Daten eingelesen oder ausgegeben werden sollen. Daraufhin verlassen oder betreten die Daten in Form von Strom oder fehlendem Strom das betreffende Wort im Speicher über den Datenbus.

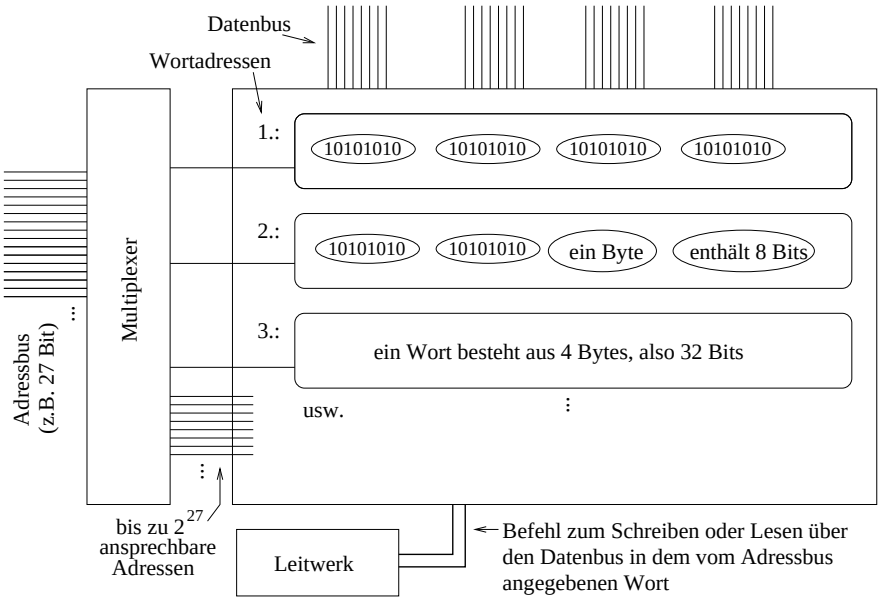



Abbildung 2.7: Schematischer Aufbau eines Speichers

### Massenspeicher

Um Daten auch nach dem Abschalten des Rechners erhalten zu können, werden sogenannte Massenspeicher angeschlossen. Diese können sowohl gelesen als auch beschrieben werden, sind stromunabhängig und besitzen wesentlich höhere Speicherkapazitäten als ROM und RAM. Allerdings erfolgt der Zugriff auf die Daten in Massenspeichern bedeutend langsamer. Trotzdem finden selbst Magnetbänder heute noch Verwendung bei der Langzeitspeicherung von Daten. Die Bilder des Weltraumteleskopes Hubble werden beispielsweise auf Magnetbändern festgehalten. Die unterschiedlichen Eigenschaften einiger Speichermedien werden

in Tabelle 2.1 deutlich, wobei berücksichtigt werden muss, dass besonders die Speicherkapazitäten von Festplatten laufend erhöht werden.

| Massenspeicher: | Festplatten | CD | Diskette | Magnetband |
|---|---|---|---|---|
| Zugriffszeit: | 10ms | 200ms | 200ms | Minuten |
| Speicherkapazität: | 10GB | 700MB | 2MB | 10GB |
| Übertragungsgeschw.: | 10MB/sec | 1MB/sec | 100kB/sec | 3MB/sec |

Tabelle 2.1: Vergleich einiger Massenspeicher

### 2.2.6 Speicherung und Erstellung von Binärcodes

Damit Dateien von verschiedenen Rechnern gelesen und bearbeitet werden können wurde ein internationaler Speicherstandart, der IEEE festgelegt.

- Der Binärcode von ausführbaren Dateien, den executables oder Programmen besteht aus folgenden Elementen, die zusammen jeweils 32 oder 64 Bit Speicherplatz benötigen:
  - Interrupt, unterbricht den Befehl, falls Fehler auftreten. Es gibt verschiedene starke Interupts, die schwächere besiegen können. Bei dem Größten spricht man vom Crash, wohingegen der Kleinste das Unterbrechen einer Programmschleife bewirkt.
  - Operationsbits, welche die Befehle zum Ausführen von Operationen enthalten.
  - Adresse, gibt die nötigen Anweisungen um die benötigten Daten oder Befehle zu finden.
- Der Binärcode von Gleitkommazahlen setzt sich aus folgenden Teilen zusammen:
  - Vorzeichen, es wird vom ersten Bit festgelegt.
  - Mantisse (Ziffernfolge), sie wird von den folgenden Bits bestimmt.
  - 10er Potenz, sie wird durch die letzten Bits gegeben und bestimmt die Stellung des Kommas.

Da Binärcodes nicht direkt programmiert werden können, nimmt man sich Programmiersprachen zur Hilfe, die von speziellen Programmen in Maschinensprache übersetzt werden. Die üblichen Schritte zur Erstellung eines Binärcodes sind in Tabelle 2.2 zusammengefaßt. Ausführbare Dateien, executables oder exe-files genannt, sind normalerweise in Assembler und/oder Binärcodes geschrieben.

| | | |
|---|---|---|
| Basic, Pascal, C | Datei in beliebiger Programmiersprache geschrieben | a+b=c |
| ⇓ | passender Compiler übersetzt die Sprache und optimiert dabei bereits möglichst die Abläufe auf den vorliegenden Prozessor | ⇓ |
| Assembler | Datei ist bereits prozessorabhängig, kann aber noch programmiert werden, sie besteht aus einer Programmierhilfe (=mnemonic) und einer Zeichenkette | fetch a +Zeichenkette<br>fetch b +Zeichenkette<br>add ab +Zeichenkette<br>store c +Zeichenkette |
| ⇓ | im CORE schreibt der Interpreter die Assembler-Datei in Binärcodes um | ⇓ |
| Binärcode | die ausführbare Datei ist absolut prozessorabhängig und im allgemeinen nicht kompatibel | 0100101101... |
| ⇒ Decoder | liest die Datei bei Aufruf, um sie auszuführen | |

Tabelle 2.2: Vorgehensweise zur Erstellung von Computerprogrammen

### 2.2.7 Aufbau von Mikrochips

Ein Chip besteht aus mehreren Millionen Schaltern, diese Schalter sind aus Halbleitern wie Silizium oder Germanium hergestellte Transistoren, welche auf kleinstem Raum miteinander verbunden sind. Abbildung 2.8 zeigt den Querschnitt eines Transistors, der den Stromfluss beim anlegen negativer Spannung am Gatter unterbricht. Hergestellt werden sie durch Ätzen der gewünschten Strukturen in spezielle, hochreine Siliziummonokristalle. Die feinen Strukturen werden, wie in Abbildung 2.9 gezeigt, durch das stellenweise Belichten eines lichtempfindlichen Lackes auf die Kristallschichten übertragen. Diese extrem aufwendige Herstellung kann weltweit nur von etwa 5 Firmen in Japan und den USA in besonderen Cleanrooms bewerkstelligt werden. Die Grenze zur Verkleinerung der Chips liegt im Nanometer-Bereich und wird durch das unkontrollierte Tunneln von Elektronen durch die Transistoren gesetzt.

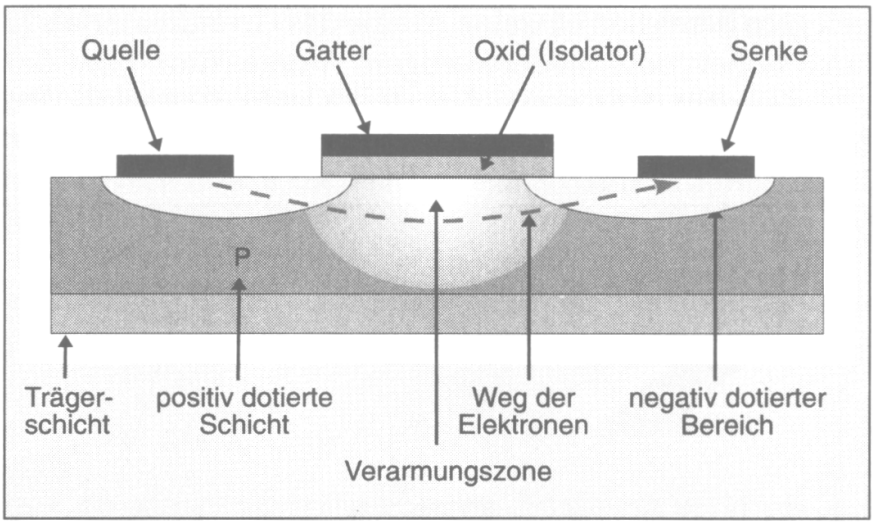



Abbildung 2.8: Schematischer Aufbau eines Metalloxid-Feldeffekt-Transistors (MOSFET). Wenn am Gatter keine Spannung anliegt, können Elektronen von der Quelle zur Senke durch die Verarmungszone passieren. Wenn negative Spannung anliegt, können sie nicht passieren. ([1], S. 45)

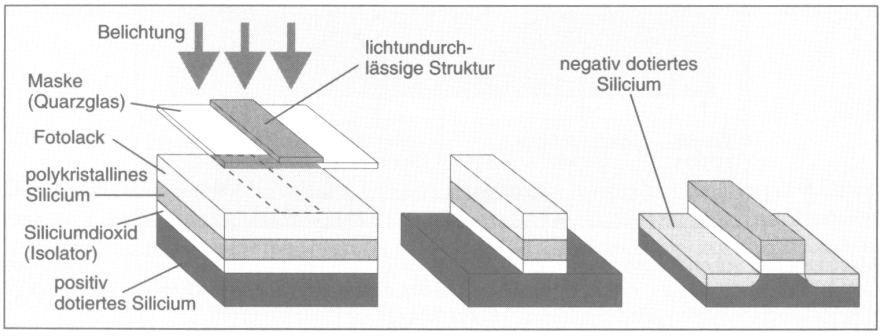



Abbildung 2.9: Die Herstellung von Schaltkreisen auf einem Wafer (dünne Scheibe des Silicium-Einkristalls) erfordert mehrere Belichtungs- und Ätzvorgänge. Der belichtete Fotolack wird in diesem Beispiel weggeätzt, der unbelichtete Fotolack widersteht dem Ätzen. ([1], S. 46)

# Kapitel 3

# Einführung in das UNIX-Betriebssystem

Beim Start eines Computers werden zunächst nur die im ROM gespeicherten Befehle, das Boot-Programm, ausgeführt. Diese überprüfen das System und greifen daraufhin auf den Massenspeicher zu. Von dort wird das installierte Betriebssystem in den Arbeitsspeicher (RAM) geladen. Daraufhin übernimmt dieses die Kontrolle über den Computer. Dabei erfüllt das Betriebssystem drei wichtige Aufgaben, die Verwaltung von Massenspeichern und Peripherie, sowie die Ausführung von Befehlen.

## 3.1 Arbeitsweise von Betriebssystemen

Die frühen Betriebssysteme konnten nur nach dem batch-Prinzip arbeiten, das heißt, dass der Prozessor alle Programme nacheinander abarbeitet. Da der Benutzer hierbei mitunter sehr lange bis zum Ende eines Prozesses warten mußte, wurde eine interaktive Prozessorsteuerung entwickelt. Dies bedeutet, dass jeder neuen Eingabe eine Priorität zugeordnet wird. Um dennoch nicht alle anderen Prozesse stoppen zu müssen, werden die Aufgaben stückweise in Zeitscheiben, den *time slices* bearbeitet. Befehle mit höheren Prioritäten werden dabei verstärkt ausgeführt. Dabei entsteht für den Benutzer der Eindruck, es liefen mehrere Programme gleichzeitig ab, obwohl der Prozessor immer nur eines bearbeiten kann. Dieses Prinzip des time sharing ermöglicht folgende Anwendungen, die im batch-Modus nicht möglich sind:

- Multiuser (MU), das bedeutet, es können mehrere Personen gleichzeitig an einem Rechner arbeiten oder eine Person kann mit mehreren Bildschirmen arbeiten. Bei dem Betriebssystem UNIX ist es möglich, sich von einem kleinen Rechner, dem Arbeitsplatz, in einen großen Rechner, den Server, einzuwählen. Dabei übernimmt der Arbeitsplatz nur die Ein- und Ausgabe von Daten, während der Server mit seiner höheren Rechenleistung die

Befehle ausführt.

- Multitasking (MT) findet statt, wenn ein Benutzer gleichzeitig an mehreren Programmen arbeiten kann, obwohl der Prozessor immer nur ein Programm ausführt. Ein im Hintergrund laufender Batchjob wird in UNIX mit dem Befehl `name &` aufgerufen, z.B. `netscape &`. Dabei behält das interaktive Befehlsfenster (x-term) die erste Priorität, aber im Hintergrund läuft Netscape als Batchjob. In Wirklicheit laufen wesentlich mehr Prozesse gleichzeitig, wie zum Beispiel die Mausbewegung.

Ein gutes Betriebssystem zeichnet sich sowohl durch Effizienz als auch durch Benutzerfreundlichkeit aus. Bekannte Betriebssysteme sind Windows, dessen Vorgänger DOS und UNIX. Während erst die neuesten Windows-Betriebssysteme MT und MU fähig sind, ist dies bei UNIX traditonell vorhanden. Der Grund hierfür liegt darin, dass UNIX die Kontrolle über das gesammte System immer behält und lediglich Aufgaben an einzelne Programme verteilt. Windows hingegen übergibt die Kontrolle an das laufende Programm, was laufende Batchjobs blockiert und bei einzelnen Programmfehlern leichter zum Absturz des gesamten Systems führen kann.

## 3.2 Einordnung des Betriebssystems in Hard- und Software

Das Betriebssystem besteht wie in Abbildung 3.1 gezeigt aus zwei Teilen, der SHELL und dem CORE. Es stellt die Verbindung zwischen den beliebig veränderbaren Applications und der Hardware dar. Die SHELL wendet die Befehle der Programme an und kann durch hinzufügen von SHELL-Skripten vom Benutzer verändert werden. Der CORE ist der feste, für den Benutzer unzugängliche Teil des Betriebssystems, der für die Verwaltung der Hardware, also beispielsweise der Speicherplätze, zuständig ist.

## 3.3 Dateisystem

### 3.3.1 Verzeichnisstruktur

Das Dateisystem besteht aus allen Dateien oder Verzeichnissen, die für einen Rechner sichtbar sind. Es ist in einer hierarchischen Baumstruktur angeordnet, die ihren Ursprung stets in dem sog. *root-Verzeichnis* `/` hat. Eine Auswahl typischer Verzeichnisse und ihre Anordnung ist in Abbildung 3.2 dargestellt.

Diese Struktur muss nichts mit der tatsächlichen physikalischen Speicheranordnung zu tun haben. Einzelne Verzeichnisse können sich auf unterschiedlichen

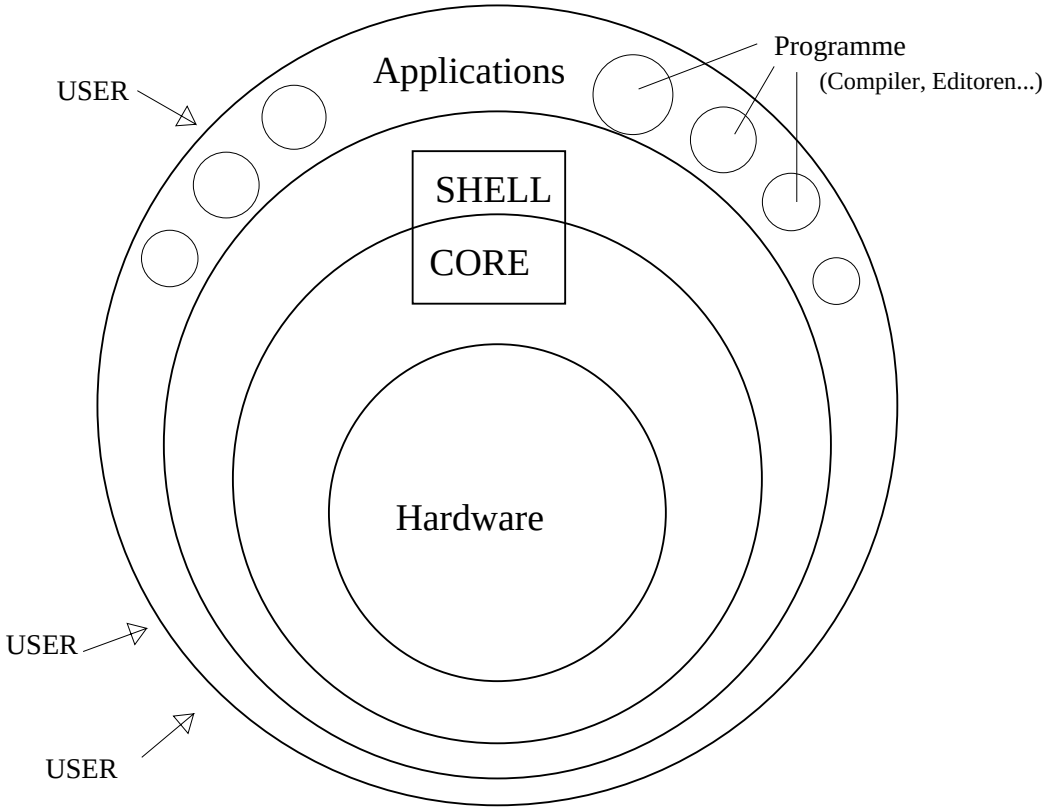



Abbildung 3.1: Schematische Darstellung des Betriebssystems als Übersetzer der von verschiedenen Benutzern dur ch unterschiedliche Programme angeforderten Hardwareprozesse

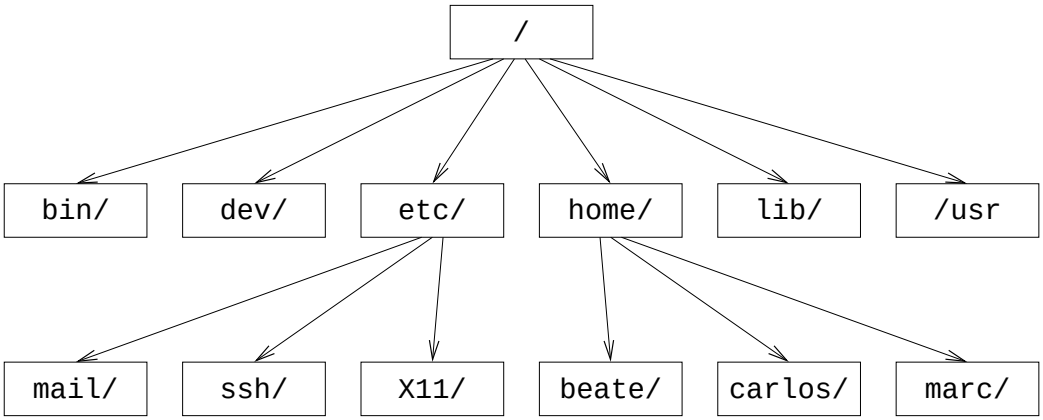



Abbildung 3.2: Typische Einträge einer UNIX-Verzeichnisstruktur. `bin/` enthält ausführbare Programme, `dev/` Gerätedateien (siehe 3.3.3), `etc/` Einstellungen für verschiedene Komponenten, `home/` persönliche Daten der einzelnen Benutzer, `lib/` Programmbibliotheken und `usr/` Anwendungsprogramme.

Festplatten, Partitionen der selben Platte oder sogar auf einem anderen Computer, der über eine Netzwerkverbindung zugänglich ist, befinden. Auch wechselbare Datenträger wie Disketten oder CD-ROMs werden bei Gebrauch an bestimmten Stellen in diesen Baum eingebunden. Allgemein spricht man beim Einbinden von Teilverzeichnisbäumen vom *Mounten*. Häufig befindet sich z. B. der Inhalt des `home`-Verzeichnisses auf einer gesonderten Festplattenpartition. Diese wird dann beim Systemstart an der Stelle `/home` *gemountet* (vgl. Abbildung 3.3). Ebenso wie das Partitionieren von Festplatten ist das Mounten eine Aufgabe des Systemadministrators.

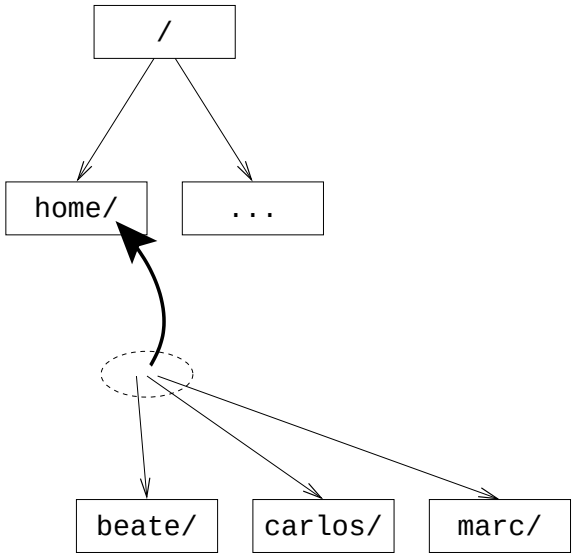



Abbildung 3.3: Die Verzeichnisse der Benutzer werden an der Stelle `/home/` in den Verzeichnisbaum gemountet.

Will man in ein untergeordnetes Verzeichnis wechseln, so gibt man hierfür den Befehl `cd` *unterverzeichnis* ein. Jedes Verzeichnis enthält die beiden Einträge „`.`“ und „`..`“. „`.`“ steht für das Verzeichnis selbst, „`..`“ für das übergeordnete Verzeichnis. Somit kann man mittels `cd ..` eine Ebene nach oben wechseln. Absolute Pfadangaben sind möglich, indem man sich auf das root-Verzeichnis `/` bezieht, z. B. `cd /etc/mail`. Selbstverständlich können mehrere Angaben in einem Kommando kombiniert werden: `cd ../../home/beate`

### 3.3.2 Zugriffsrechte

Unter UNIX gehört jeder Benutzer üblicherweise der Gruppe `users` an. Benutzer- und Gruppenname der Person, die eine Datei erstellt hat, spiegeln sich auch im Dateisystem wieder. Die vollständigen Eintragungen zu einer Datei zeigt der Befehl `ls -l` an. Für die Datei `liebe.tex` könnte dies folgendermaßen aussehen:

```
-rw-r-----   1 marc users    9430 Jan 26 11:19 liebe.tex
```
An dieser Stelle sind nur die Einträge `-rw-r-----`, `marc` und `users` von Interesse: Die Datei wurde vom Benutzer Marc, der der Gruppe `users` angehört, erstellt und trägt die Dateiattribute (auch *Zugriffsbits* genannt) `-rw-r-----`. Diese Zugriffsbits sind folgendermaßen zu lesen: Die erste Position (hier `-`) bestimmt den Dateityp, ein Strich steht für eine gewöhnliche Datei. Die nächsten 3 Stellen (`rw-`) bestimmen die Rechte des Eigentümers, also Marc: Er darf die Datei lesen (`r`), schreiben (`w`), aber nicht ausführen (`-`), was bei einer Datendatei auch wenig sinnvoll wäre. Für eine ausführbare Datei müsste statt dem `-` ein `x` stehen, also z. B. `-rwx------`. Die beiden verbleibenden Zeichentripel `r--` und `---` stehen für die Rechte zunächst der Angehörigen der Gruppe `users` und dann aller anderen Benutzer. Wie unschwer zu erkennen ist, dürfen die „Gruppenkollegen" von Marc die Datei lesen, aber nicht verändern, die restlichen Benutzer dürfen nicht einmal das. Eine Ausnahme bildet der Systemadministrator `root`, der unabhängig von Zugriffsbits alle Dateien lesen und beschreiben darf.

Eigentümer und Gruppenzugehörigkeit einer Datei können mit dem Kommando `chown` *user*`:`*group* *datei* verändert werden, z. B.
`chown carlos:users baum.eps`.

Zum Ändern der Zugriffsbits dient der Befehl `chmod` *wer aktion bits datei*. *wer* setzt sich aus einer beliebigen Kombination der Buchstaben `u` für „user", `g` für „group", `o` für „others" und `a` für „all" zusammen, bestimmt also, für wen die Rechte geändert werden sollen. *aktion* ist entweder `+` für setzen oder `-` für löschen der in *bits* genannten Zugriffsbits `r` („read"), `w` („write"), `x` („execute") oder einer Kombination aus diesen. Wichtig ist, dass die Zeichenfolge *wer aktion bits* kein Leerzeichen enthält. Das folgende Beispiel erlaubt nun allen Mitgliedern der Gruppe `users` den Schreibzugriff auf die Datei `liebe.tex`:
`chmod g+w liebe.tex`

### 3.3.3 Dateitypen

Wie bereits erwähnt steht für gewöhnliche Dateien an erster Stelle der Zugriffsbits ein `-`. Die wichtigsten „nicht gewöhnlichen" Dateien und ihre Kennbuchstaben sind Verzeichnisse (`d`) und Links (`l`; siehe 3.4.3). Außerdem werden unter UNIX Hardwarekomponenten wie Massenspeicher oder Eingabegeräte als sog. *Gerätedateien* im Verzeichnis `/dev/` aufgeführt. Ihr Kennbuchstabe ist `b` oder `c`.

### 3.3.4 Versteckte Dateien

Dateinamen dürfen unter UNIX beliebig viele Punkte enthalten. Dateien, deren Name mit einem Punkt beginnt, können als „versteckte Dateien" betrachtet werden. Die meisten Befehle ignorieren sie standardmäßig. Z. B. zeigt `ls` üblicherweise nur nicht versteckte Dateien an. Erst mit der Option `-a` als `ls -a` werden auch versteckte Dateien angezeigt.

| sh | ksh | csh | **Bedeutung** |
|---|---|---|---|
| `$` | `$` | `%` | Prompt |
| | `alias x=y` | `alias x y` | `x` steht für `y` |
| `set x y` | `set x y` | `set x=y` | Wertzuweisung `x` := `y` |
| `$?` | `$?` | `$status` | exit-Status |
| `$#` | `$#` | `$#argv` | Anzahl der Argumente |
| `pwd` | `pwd` | `dirs` | „print working directory“ |
| `read` | `read` | `$<` | Lese vom Terminal |
| `for ... do` | `for ... do` | `foreach` | for-Schleife |
| `while ... do` | `while ... do` | `while` | while-Schleife |
| `done` | `done` | `end` | Schleifenende |
| `if[$i -eq 5]` | `if ((i==5))` | `if ($i == 5)` | Abfrage, ob `i` = 5 |
| `fi` | `fi` | `endif` | Ende eines `if` |

Tabelle 3.1: Vergleich der Implementierung wichtiger Konzepte in der Bourne- (`sh`), Korn- (`ksh`) und C-Shell (`csh`)

Aus diesem Grund beginnen die Namen von Systemdateien oder Konfigurationsdateien, die bei normaler Benutzung nicht geändert werden müssen und sollen, oft mit einem Punkt.

## 3.4 Shell-Kommandos

### 3.4.1 Vorbemerkung

Mit dem Begriff Shell bezeichnet man Teile des Betriebssystems (vgl. Abbildung 3.1), die die Schnittstelle zwischen Betriebssystemkern und Anwendungen darstellen. Wenn hier von *der* Shell die Rede ist, so ist damit nur *ein* Programm aus diesem Teil des Betriebssystems gemeint, und zwar der Befehlsinterpreter für Textkommandos.

Es gibt verschiedene Shells für UNIX, z. B. die Bourne Shell `sh` oder die unter Linux häufige Bourne Again Shell `bash`. In der grundlegenden Bedienung unterscheiden sich alle diese Shells kaum. Wenn man jedoch intensiver mit einer Shell arbeitet, stößt man auf individuelle Besonderheiten. Soweit nicht anders angegeben beziehen sich alle hier gegebenen Informationen auf die C-Shell `csh`. Tabelle 3.1 vergleicht Bourne-, Korn- und C-Shell unter verschiedenen Gesichtspunkten; die dort erwähnten Konzepte werden weiter unten erläutert, soweit sie nicht über den Rahmen dieser Einführung hinausgehen.

Es ist nicht immer einfach, zwischen UNIX-Befehlen, die normale Programme, also ausführbare Dateien, sind, und Shell-Befehlen, die von der Shell selbst interpretiert werden, zu unterscheiden, da eine Shell bestimmte UNIX-Befehle ab-

| MS-DOS | UNIX | Beschreibung |
|---|---|---|
| `copy` | `cp` | copy file(s) |
| `ren` | `move` | move (=rename) |
| `del` | `rm` | remove |
| `mkdir` | `mkdir` | make directory |
| `rmdir` | `rmdir` | remove directory |
| `dir` | `ls` | list files |
| `type` | `cat` | zeige Dateiinhalt |
| `more` | `more` | zeige Dateiinhalt seitenweise |
| `print` | `lp` | line-print |

Tabelle 3.2: Wichtige UNIX-Befehle und ihre Pendants aus MS-DOS

fangen und intern bearbeiten kann. UNIX-Befehle an sich sind jedoch unabhängig von einem bestimmten Kommandointerpreter.

Es ist einfach, bei der Arbeit mit UNIX zu einer anderen Shell zu wechseln, sofern diese installiert ist. Dazu gibt man lediglich ihren (abgekürzten) Namen ein, z. B. `ksh` für die Korn-Shell. Um wieder zur vorherigen Shell zurückzukehren, muss man sich aus der neuen Shell mit `exit` abmelden.

### 3.4.2 Allgemeines

Obwohl es mittlerweile auch für UNIX grafische Oberflächen gibt, ist es üblich, mit Textkommandos zu arbeiten. So erreicht man eine höhere Flexibilität und ist, ein gewisses Maß an Übung vorausgesetzt—schneller als mit grafischer Benutzerführung und überhaupt erst in der Lage, die Möglichkeiten des Systems vollständig auszuschöpfen. Wichtige UNIX-Befehle sind grundsätzlich mit den Befehlen von MS-DOS vergleichbar, jedoch ungleich leistungsfähiger und flexibler. Eine Übersicht einiger UNIX- und MS-DOS-Befehle bietet Tabelle 3.2.

Die allgemein übliche Syntax von UNIX-Kommandos sieht folgendermaßen aus:

*command -option(en) argument1 argument2 ...*

Natürlich können Optionen wie auch Argumente entfallen, wenn das Kommando keine weiteren Informationen benötigt. Es folgen einige Beispiele, die Befehle werden in den folgenden Abschnitten erklärt:

```
cp liebe.tex liebe1.tex
ls -a subdir
grep -i wolf adressen.txt
```

Die hier gebotenen Befehlserläuterungen können nur ein grober Überblick

sein. Eine ausführliche Anleitung zu jedem Kommando *command* erhält man durch Eingabe des Befehls `man` *command*.

### 3.4.3 Dateiverwaltung

Zum Kopieren von Dateien oder Verzeichnissen wird das Kommando `cp` eingesetzt; Beispiel:
`cp quelle.txt subdir/ziel.txt`.

Soll die Datei dagegen verschoben, die Quelle also nach dem Kopieren gelöscht werden, so bedient man sich des Befehls `mv`. Diesen verwendet man auch zum Umbenennen von Dateien, z. B.
`mv alt neu`

Oft wird nicht tatsächlich eine Kopie einer Datei benötigt, sondern man möchte sie lediglich an zwei verschiedenen Orten im Dateisystem aufgeführt haben. Für das Erstellen sog. *Links* ist das Kommando `ln` zuständig.
`ln liebe.tex liebe1.tex`
erstellt mit `liebe1.tex` einen zweiten Namen für die bereits existierende Datei `liebe.tex`. Beide Namen sind später völlig gleichberechtigt und damit wie andere Dateien bearbeitbar, verweisen jedoch auf den selben Speicherbereich. Wenn man einen der beiden Namen löscht, so bleiben die Daten unter dem zweiten Namen erhalten, von Änderungen einer Datei ist auch die andere betroffen. So erstellte Links bezeichnet man als *hard links*, alternativ dazu gibt es auch *soft* oder *symbolic links*. Sie werden ebenfalls mit `ln`, jedoch unter Angabe der Option `-s` erstellt, also z. B. mit
`ln -s liebe.tex liebe1.tex`.
Nun ist liebe1.tex ein soft link, d. h. lediglich ein Verweis auf die Ausgangsdatei `liebe.tex`. Wenn man den Link löscht, hat das keine weiteren Folgen; wird jedoch die Ausgangsdatei gelöscht, so bleibt der Link zwar bestehen, ist aber unbrauchbar, da er auf eine nicht mehr existierende Datei verweist. Die meisten Kommandos folgen bei der Angabe eines soft links als Argument automatisch dem Verweis und arbeiten mit der entsprechenden Datei, wie wenn man sie direkt angegeben hätte.

Links spielen eine wichtige Rolle in der internen Struktur des Betriebssystems. Unter anderem gibt es Programme, die sich in Abhängigkeit des Namens, unter dem sie aufgerufen wurden, unterschiedlich verhalten. Dass ein und das selbe Programm unter verschiedenen Namen aufgerufen werden kann, lässt sich mittels Links besonders effizient realisieren.

Bereits mehrmals erwähnt wurde das Kommando `ls`. Ohne Argument aufgerufen zeigt `ls` die Namen aller Dateien und Verzeichnisse im aktuellen Verzeichnis, die nicht mit einem Punkt beginnen, an. Wenn auch versteckte Dateien oder Verzeichnissen aufgelistet werden sollen, so muss die Option `-a` verwendet werden. Für ausführlichere Informationen ist die Option `-l` erforderlich. `ls -l` zeigt für jeden Eintrag wie folgt aussehende Zeilen an:

| | |
|---|---|
| `*` | passt auf jeden String (inkl. Sonderzeichen) |
| `?` | passt auf genau ein beliebiges Zeichen |
| `[ab7]` | passt auf eines der Zeichen a, b oder 7 |
| `[A-Z0-9]` | passt auf einen Großbuchstaben oder eine Ziffer |
| `[^0]` | passt auf alle Zeichen außer 0 |
| `[^A-Z]` | passt auf alle Zeichen außer Großbuchstaben |

Tabelle 3.3: Häufig verwendete Wildcards

```
drwxr-xr-x   2 marc  users    4096 Mär  2 16:23 backup
-rw-r-----   1 marc  users    9430 Jan 26 11:19 liebe.tex
```

Die erste Spalte enthält die Zugriffsbits. Man beachte, dass für Verzeichnisse (erkennbar am führenden `d`) das `x`-Bit eine andere Bedeutung hat: Statt Ausführen erlaubt es das Wechseln in das Verzeichnis mit `cd`.

In der zweiten Spalte steht die Anzahl der Verweise (hard links) auf die Datei. Für Verzeichnisse steht hier mindestens die Zahl 2, da zusätzlich zum eigentlichen Verzeichnis *dir* immer auch der Eintrag *dir*`/.` auf die selbe Speicherstelle verweist.

Die restlichen Spalten sind selbsterklärend: Sie informieren über Benutzer- und Gruppenzugehörigkeit, Größe, Erstellungs- bzw. Veränderungsdatum- und zeit und den Namen des Eintrags.

Will man ausführliche Informationen über ein Verzeichnis erhalten, so ist der Aufruf von `ls -l` *dir* keine gute Idee. Der Grund ist, dass `ls` standardmäßig den Inhalt eines Verzeichnisses anzeigt, nicht das Verzeichnis selbst. Will man nur den Verzeichniseintrag mit seinen Attributen sehen, so gibt man zusätzlich die Option `-d` an: `ls -ld` *dir*

### 3.4.4 Wildcards

Bestimmte Zeichen in Kommandozeilen können für ganze Gruppen von Zeichenfolgen stehen. Diese Zeichen bezeichnet man als *Wildcards* oder Jokerzeichen. Tabelle 3.3 zeigt die häufigsten dieser Symbole. Zeichenfolgen mit Wildcards werden von der Shell vor Ausführung des eigentlichen Kommandos durch eine Liste aller Verzeichniseinträge ersetzt, die auf die jeweilige Zeichenfolge „passen“. D. h. `*` wird durch alle Einträge ersetzt, `?` durch alle, die aus nur einem Zeichen bestehen, usw. Durch Kombination verschiedener Wildcards und fester Namensbestandteile lassen sich fast beliebige Dateien für bestimmte Befehle herausgreifen. Es folgen einige Beispiele:

```
cp *.jpg /scratch
```

kopiert alle Dateien mit der Endung `.jpg` in das Verzeichnis `/scratch`.

```
mv *beate?* beate
```

verschiebt alle Dateien und Verzeichnisse, deren Name „beate“ enthält, aber mindestens um ein einzelnes Zeichen länger ist als die Zeichenfolge „beate“ allein, in

das Verzeichnis `beate`. Die Angabe des `?` ist sinnvoll, um die Fehlermeldung zu verhindern, die beim Versuch `beate` selbst nach `beate` zu verschieben, entstehen würde; `*` kann also für beliebige, aber auch für *kein* Zeichen stehen.

```
rm adressen/[a-c]*
```

entfernt alle Dateien aus dem Verzeichnis Adressen, die mit einem a, b oder c beginnen.

**Achtung:** Dateinamen sind in UNIX grundsätzlich *case-sensitive*, d. h. Namen, die sich nur in Groß- bzw. Kleinschreibung unterscheiden, stehen dennoch für verschiedene Dateien!

### 3.4.5 Arbeiten mit Textdateien

Textdateien spielen unter UNIX eine wichtige Rolle. Selbstverständlich gibt es also eine ganze Menge von Befehlen, die sich damit befassen, sie auf dem Bildschirm anzuzeigen oder ihren Inhalt zu bearbeiten.

Der einfachste dieser Befehle ist `cat` *datei*, er druckt den Inhalt von *datei* Zeile für Zeile auf dem Bildschirm aus. Passt der Text nicht vollständig auf den Bildschirm, so läuft einfach der gesamte Dateiinhalt mit maximaler Geschwindigkeit durch, es wird nicht angehalten, um auf eine Benutzereingabe zu warten. Der Vorteil dieses Verhaltens ist, dass man bei Angabe mehr als eines Arguments mehrere Dateien lückenlos hintereinander anzeigen kann.

Zum Lesen längerer Dateien komfortabler ist `more` *datei*. `more` zeigt Text seitenweise an, wobei es nach jeder Seite auf einen Tastendruck wartet, bevor es die nächste Seite zeigt.

Noch wesentlich praktischer als `more` ist `less`. `less` *datei* ermöglicht es nämlich dem Benutzer, sich völlig willkürlich seiten- oder zeilenweise vorwärts wie rückwärts im Text zu bewegen. Darüberhinaus bietet es bei Tippen des `/` eine Suchfunktion an. `less` kann wie `more` mit `q` beendet werden.

Bei Systemprotokollen, die stets nur am Ende erweitert werden, oder wenn es darum geht, lediglich einen kurzen Blick in eine Datei zu werfen, kann es genügen, ausschließlich Beginn oder Ende des Inhalts zu sehen. Dafür gibt es die Kommandos `head` *datei* und `tail` *datei*. Sie zeigen nur die ersten bzw. letzten 10 Zeilen einer Textdatei an.

Im Gegensatz zu den bisher genannten Befehlen, die den Inhalt einer Datei ausgeben, gibt der `echo`-Befehl beliebigen Text aus, den der Anwender als Argument übergeben muss. Z. B. druckt der Befehl

```
echo "Es ist 10:36 Uhr."
```

den Text „Es ist 10:36 Uhr.“ (ohne Anführungszeichen) auf dem Bildschirm aus. Die Anführungszeichen sollte man als Vorsichtsmaßname immer hinzufügen, da für die Shell bestimmte Sonderzeichen, wenn sie nicht in Anführungszeichen eingeschlossen sind, spezielle Bedeutung haben und so zu unerwarteten Ergebnissen führen könnten.

Der Befehl `wc` zählt die Zeilen, Wörter und Zeichen einer Datei. Mit den Optionen `l`, `w` und `m` kann man die Bearbeitung auf das Zählen von (in dieser Reihenfolge) Zeilen, Wörtern oder Zeichen beschränken. Beispielsweise gibt der folgende Aufruf nur die Anzahl der Zeilen und Wörter einer Datei aus:

```
wc -lw liebe.tex
```

`sort` gibt die Zeilen einer Datei sortiert aus. Zu beachten ist, dass die standardmäßige Sortierreihenfolge nicht wie erwartet die alphabetische ist, sondern sich nach den Zeichencodes, also den Bytewerten jedes einzelnen Zeichens, richtet. Um nach anderen Kriterien zu sortieren, kann man eine der zahlreichen Optionen angeben, die mittels `man sort` abgerufen werden können. Z. B. gibt

```
sort -f index
```

den Inhalt der Datei `index` ohne Berücksichtigung von Groß- und Kleinschreibung zeilenweise sortiert aus.

Das Vergleichen von zwei Dateien ist mit dem Befehl `diff` möglich. Er erwartet als Argumente die Namen der beiden Dateien. Standardmäßig angezeigt werden alle Zeilen, die Unterschiede enthalten bzw. nur in einer Datei vorkommen. Beim Vergleich wird das erste Argument als Ausgangsdatei betrachtet, von der aus gesehen nach Veränderungen in der zweiten Datei gesucht und angezeigt werden. Das folgende Beispiel gibt an, was sich seit Erstellung der Sicherungskopie `main.c~` in der Datei `main.c` geändert hat:

```
diff main.c~ main.c
```

Oft will man Dateien nach bestimmten Wörtern durchsuchen. Dies geschieht mit dem Kommando `grep`. Das erste Argument ist das Suchmuster, im einfachsten Fall ein beliebiges Wort, alle weiteren Argumente geben die Dateien an, in denen gesucht werden soll; es ist also möglich, mit einem Aufruf mehrere Dateien zu durchsuchen. Ausgegeben werden alle Zeilen, die das Suchwort enthalten, bei mehreren Dateien wird auch der jeweilige Dateiname mit angegeben. Mit der Option `-l` zeigt `grep` lediglich die Namen aller Dateien an, in denen es fündig wurde, aber keine Ausschnitte aus deren Inhalt. Beispiel:

```
grep -l Carlos *.tex *.html
```

gibt die Namen aller TEX- und HTML-Dateien aus, die „Carlos“ enthalten.

`grep` bietet zahllose Optionen wie z. B. `-i`, um die Suche ohne Berücksichtigung von Groß- und Kleinschreibung durchzuführen. Als Suchmuster kann auch eine sog. *regular expression* angegeben werden, das ist eine Zeichenfolge, die bestimmte Symbole enthält und damit auf verschiedene Zeichenketten passt. Regular expressions können u. a. die in Tabelle 3.3 gezeigten eckigen Klammern enthalten, um Gruppen von Zeichen an bestimmten Positionen zu erlauben oder zu verbieten, stellen aber noch weitaus komplexere Möglichkeiten bereit, auf die hier nicht eingegangen werden kann. Zu beachten ist, dass die Zeichen `*` und `?` hier eine andere Bedeutung als in Wildcards besitzen. Ein Beispiel für eine einfache Benutzung von regular expressions mit `grep` ist

```
grep -i "[^aeiou][0-9]" artikel,
```

das in `artikel` alle Kombinationen aus einem Zeichen, das kein Vokal ist, und

einer Ziffer findet. Die Anführungszeichen erfüllen hier den Zweck, die Ausdrücke mit eckigen Klammern vor der Shell zu schützen, die sie sonst als Wildcards betrachten und durch die Menge aller Dateinamen, die auf das Suchmuster passen, ersetzen würde.

### 3.4.6 Umleiten von Aus- und Eingabe

Fast alle hier besprochenen Befehle lesen bzw. schreiben von einer sog. *Standardein-* bzw. *ausgabe*. Die Standardeingabe ist üblicherweise mit der Tastatur verbunden, die Standardausgabe mit dem Bildschirm. Für bestimmte Aktionen ist es sinnvoll, diese Schnittstellen umzuleiten, sodass ein Programm seine Eingaben ohne Zutun des Benutzers aus einer Datei liest und seine Ausgaben in eine andere Datei geschrieben werden.

Das Umleiten der Eingabe aus einer Datei erfolgt mittels des <-Zeichens, also durch *command* `<` *datei*. Viele Kommandos wie `cat` lesen von der Standardeingabe, also normalerweise der Tastatur, wenn keine Datei als Argument angegeben wird. Also ist die Umleitung `cat < liebe.tex` äquivalent zu `cat liebe.tex`.

Zum Umleiten der Ausgabe schreibt man einfach `>` *datei* hinter ein Kommando. *datei* wird daraufhin neu erstellt bzw. falls bereits vorhanden überschrieben und mit der Ausgabe des Kommandos gefüllt. Beispielsweise kann man den Inhalt eines Verzeichnisses mit dem folgenden Kommando in eine Datei ausdrucken:

```
ls /home/carlos > ls-home-carlos.txt
```

In Zusammenhang mit der Ausgabeumleitung bietet `echo` eine einfache Möglichkeit, (kurze) Textdateien zu erstellen:

```
echo Hallo Welt! > hello
```

erzeugt die Datei `hello` mit dem Text „Hallo Welt!“ als Inhalt.

Auch `cat` erhält nun eine höchst sinnvolle Anwendung, das Hintereinanderhängen von Textdateien:

```
cat liebe.txt kinder.txt > stories.txt
```

Soll die Ausgabedatei nicht überschrieben werden, sondern neuer Text lediglich am Ende angehängt werden, so schreibt man zur Umleitung `>>`, z. B.
`ls /home/carlos >> letzter-stand.txt`.

Wie man Dateien als Tastatureingaben betrachten kann, so geht das auch umgekehrt: Man kann (und muss manchmal sogar) bei Programmen, die auf Eingaben von der Tastatur warten, diese mit einem *Dateiendezeichen* (EOF=„end of file“) abschließen, bzw. das Programm damit zum Beenden veranlassen. Dieses Zeichen ist nicht direkt auf der Tastatur zu finden, sondern wird durch `Ctrl`+`D` ausgedrückt. Damit kann man sich beispielsweise von einer Shell abmelden.

### 3.4.7 Kommandoverkettung

Die wahre Mächtigkeit der UNIX-Shells erschließt sich erst, wenn man Befehle nicht nur einzeln benutzt, sondern sie miteinander verkettet, um leistungsfähigere

| | |
|---|---|
| *cmd1* `&` *cmd2* | Führe *cmd1* im Hintergrund aus, *cmd2* im Vordergrund |
| *cmd1* `;` *cmd2* | Führe erst *cmd1* aus, danach *cmd2* |
| *cmd1* `&&` *cmd2* | AND; führe *cmd2* nur aus, wenn *cmd1* fehlerfrei terminiert |
| *cmd1* `||` *cmd2* | OR; führe *cmd2* nur aus, wenn *cmd1* mit Fehler terminiert |
| *cmd1* `|` *cmd2* | Bilde eine Pipe aus *cmd1* und *cmd2* |
| *cmd1* `` `cmd2` `` | Benutze die Ausgabe von *cmd2* als Argument für *cmd1* |
| `( , )` | Zum Gruppieren von Kommandos |

Tabelle 3.4: Unter UNIX gibt verschiedene Möglichkeiten, Shellkommandos zu verketten. Zur Ausführung im Hintergrund siehe Abschnitt 3.5.1.

Kommandos zur Lösung individueller Probleme zu erzeugen. Tabelle 3.4 zeigt die Verkettungsmöglichkeiten, die im folgenden Text erläutert werden sollen. Jeder der Platzhalter *cmd1* und *cmd2* steht dabei für ein beliebiges vollständiges Shell-Kommando mit ggf. Optionen, Argumenten sowie Ein-/Ausgabeumleitung.

Ein einfaches Semikolon (`;`) beendet ein Kommando, damit können mehrere Befehle in einer Zeile geschrieben werden, z. B.
`latex liebe.tex; dvips liebe.dvi; gv liebe.ps`.

Zwei `&` (kaufmännisches Und oder Ampersand) führen dazu, dass das zweite Kommando nur dann ausgeführt wird, wenn das erste erfolgreich terminiert, d. h. nicht mit einem Fehler vorzeitig abbricht. Die folgende Befehlszeile wäre deutlich sinnvoller als das vorherige Beispiel, da die Kommandos `dvips` und `gv` auf der korrekten Ausführung von `latex` bzw. `dvips` aufbauen (mehr zu LaTeX an anderer Stelle im Skript):
`latex liebe.tex && dvips liebe.dvi && gv liebe.ps`

Ähnlich verhält sich der Operator `||`: Hier wird das zweite Kommando nur ausgeführt, wenn das erste nicht funktioniert hat, was nützlich z. B. für die Ausgabe von Fehlermeldungen ist:
`rm /var/spool/mail/carlos || echo "Fehler: Keine Berechtigung?"`

Die wohl am häufigsten verwendete Verkettungsoperation ist das sog. *Pipen* (*cmd1* `|` *cmd2*). Hier laufen beide Kommandos synchron, zusätzlich wird die Ausgabe von *cmd1* umgeleitet und als Eingabe für *cmd2* verwendet. Das ist am einfachsten an einem Beispiel zu verstehen:
`ls | wc -l`
zählt die Dateien im aktuellen Verzeichnis. `ls` liefert als Ausgabe eine Liste mit den Dateinamen, wobei in jeder Zeile ein Name steht. Diese Liste wird jedoch nicht auf den Bildschirm ausgegeben, sondern als Eingabe für `wc -l` verwendet, das die Zeilen der Liste, also die Dateien zählt. Erst die Ausgabe von `wc` wird auf dem Bildschirm angezeigt. Das selbe Ergebnis hätte man auch durch folgende Befehlsfolge erreichen können:

```
ls > /tmp/temp
```

```
wc -l /tmp/temp
rm /tmp/temp
```

Das ist natürlich umständlicher wegen der Benutzung einer temporären Datei, die zusätzlich in einem anderen Verzeichnis stehen muss, um nicht mitgezählt zu werden. Außerdem hat diese Variante den Nachteil, dass die Befehle nicht gleichzeitig ausgeführt werden, sondern nacheinander.

Durch den Einsatz einer Pipe kann auch `grep` optimiert werden. Will man nach allen Zeilen einer Datei suchen, die sowohl das Schlüsselwort „Müller“ als auch „Frank“ enthalten, kann man zwei Aufrufe von `grep` mit einer Pipe verbinden:

```
grep Müller adressen | grep Frank
```

Die letzte Verkettungsform (*cmd1* ‘*cmd2*‘) schließlich führt erst *cmd2* aus und übergibt dessen Ausgabe als Argument an *cmd1*, fügt sie also an Stelle des in Anführungszeichen eingeschlossenen Ausdrucks in die Kommandozeile ein. Es müssen unbedingt die „richtigen“ Anführungszeichen benutzt werden, das sind die, die auf der Tastatur i. d. R. als Strich von links oben nach rechts unten dargestellt werden. Das folgende Beispiel führt dazu, dass die Zeilen aller Dateien des aktuellen Verzeichnisses gezählt werden:

```
wc -l ‘ls‘
```

Natürlich hätte man dafür viel einfacher `wc -l *` schreiben können.

Bei Verkettung von mehr als zwei Befehlen kann es vorkommen, dass die Shell diese in unerwünschter Reihenfolge durchführt. Um dies zu vermeiden, können mehrere (verkette) Befehle mit einfachen Klammern zu einer Einheit gruppiert werden. Die Fehlermeldung im folgenden Beispiel besteht aus zwei Zeilen. Ohne die Klammern würde die zweite Zeile unabhängig vom Ergebnis des Kopierkommandos angezeigt. Durch die Gruppierung werden beide `echo`-Kommandos als ein einzelner Befehl betrachtet, dessen Ausführung nun vom `&&`-Operator abhängt.

```
cp *.tex backup/tex && ( echo "Passt." ; echo "Glück gehabt..." )
```

### 3.4.8 Variablen

Jede Shell bietet die Möglichkeit, Variablen anzulegen. In der C-Shell geschieht dies mit

```
set hallo="Guten Tag!".
```

Damit wird der Variablen `hallo` die Zeichenkette „Guten Tag!“ zugewiesen. Auf die so angelegte Variable kann man sich in nachfolgenden Befehlen durch Voranstellen eines `$` beziehen:

```
echo $hallo
```

führt zur Ausgabe von „Guten Tag!“ (ohne Anführungszeichen). Auch die Shell selbst legt eine große Anzahl von Variablen an, um Daten für sich selbst oder den Benutzer zugänglich bzw. auch veränderbar zu machen. Tabelle 3.5 nennt dafür Beispiele. Diese Variablen kann jeder Benutzer nach persönlichem Geschmack

| | |
|---|---|
| `argv` | Liste der Argumente zum gegenwärtigen Kommando (default: `()`) |
| `echo` | Falls gesetzt: Zeige jede Zeile nochmal, ehe sie ausgeführt wird |
| `history` | Länge der History-Liste |
| `home` | Home directory des Benutzers (als Abkürzung für `$home` kann auch `~` geschrieben werden) |
| `ignoreeof` | Ignoriere EOF vom Terminal (verhindert versehentliches Ausloggen durch `Ctrl+D`) |
| `noclobber` | Verhindere Output redirection in bereits bestehende Datei |
| `path` | Liste von Pfadnamen, in denen Kommandos nach Dateien suchen (default: `(. /usr/ucb /usr/sbin)`) |
| `prompt` | String, der als Prompt dienen soll (default: `%`) |
| `status` | Exit status des letzten Kommandos (`0` für success, sonst failure) |
| `term` | Terminaltyp (z. B. `vt100`) |
| `user` | Login Name des Users |
| `$$` | Prozessnummer der gegenwärtigen Shell |
| `$<` | Lies eine Zeile von Standard-Input |

Tabelle 3.5: Eine Auswahl der von der C-Shell gesetzten bzw. ausgewerteten Variablen

initialisieren, indem er in seinem Home-Verzeichnis die entsprechenden Zuweisungen in eine Datei namens `.profile` einträgt. Diese wird beim Einloggen eines Benutzer von der Shell ausgewertet. Ähnliches hat es mit der Datei `.xsession` auf sich, die Befehle, die beim Start der grafischen Oberfläche (*X Window System*) ausgeführt werden sollen, enthält.

Die beiden letzten Einträge in Tabelle 3.5 nehmen eine Sonderstellung ein, da sie keine „richtigen“ Variablen sind, sondern von der Shell selbst zur Verfügung gestellt werden und nur gelesen werden können.

Zusätzlich zum einfachen `$` gibt es eine Reihe weiterer Wege, sich auf Variablen zu beziehen. Diese in Tabelle 3.6 gezeigten Funktionen spiegeln die Eigenschaft der C-Shell, Variablen als Listen oder Felder zu betrachten, wieder.

### 3.4.9 Programmierung von Shell-Skripts

Shell-Skripts sind einfache Textdateien, die eine Liste von Shell-Kommandos enthalten. Setzt man ihr `x`-Zugriffsbit, so kann man sie wie Befehle des Betriebssystems durch Eingabe ihres Namens ausführen. Tatsächlich sind manche UNIX-Befehle auch nichts anderes als solche Skripte.

| | |
|---|---|
| `${`*var*`}` | Der Wert der Variablen *var* (die Klammern sind hier optional) |
| `${`*var*`[`*i*`]}` | Wähle ein Wort oder eine Liste von Worten aus der Liste *var*. *i* kann eine Zahl *n* sein, ein Bereich *m*`-`*n*, `-`*n*, *m*`-`, oder `*`. |
| `${#`*var*`}` | Die Anzahl aller Worte in *var* |
| `${?`*var*`}` | 1 falls *var* gesetzt, sonst 0 |

Tabelle 3.6: Weitere Möglichkeiten für den Zugriff auf Variablen.

Manche Systeme sind aus Sicherheitsgründen so konfiguriert, dass ausführbare Dateien im aktuellen Verzeichnis nur mit expliziter Pfadangabe gestartet werden können. Dazu stellt man dem Namen der Skript-Datei im aktuellen Verzeichnis einfach ein `./` voran, also z. B.

```
./mein-skript
```

statt

```
mein-skript.
```

Üblicherweise werden Shell-Skripts in einer neuen Shell gestartet. Dies lässt sich dadurch veranschaulichen, dass ein `exit` in einer Skript-Datei nur das Skript beendet, nicht jedoch die aufrufende Shell. Wenn es aus irgendeinem Grund nötig ist, dass die Kommandos aus einer Datei in der aufrufenden Shell selbst ausgeführt werden (also wie wenn man sie selbst eingegeben hätte), so kann man dies durch das Kommando `source` *skript* erreichen. `source` kann durch einen einfachen Punkt abgekürzt werden:

```
. abmelde-skript
```

Die UNIX-Shells bieten eine einfache, für viele Aufgaben jedoch vollkommen ausreichende Programmiersprache. Dafür stellen sie auch Abfragebefehle wie `if` und `case` sowie Schleifenanweisungen (z. B. `for` und `while`) zur Verfügung. Diese zu erklären würde jedoch entschieden über das Ziel dieser Vorlesung hinausreichen. Bereits mit den hier erlernten Fähigkeiten kann man aber sinnvolle Skripts schreiben.

Die folgenden beiden Kommandos erstellen ein Shell-Skript `fc`, das die Dateien im aktuellen Verzeichnis zählt:

```
echo "ls | wc -l" > fc
chmod a+x fc
```

**Achtung:** Die `bash` definiert bereits ein eigenes Kommando `fc` mit anderer Bedeutung, hier sollte man einen anderen Namen wählen.

| | |
|---|---|
| `!!` | Das vorige Kommando |
| `!`*n* | Kommando Nummer *n* |
| `!-`*n* | Das *n*-letzte Kommando (`!!` = `!-1`) |
| `!`*string* | Das letzte Kommando, das mit *string* anfängt |
| `!?`*string*`?` | Das letzte Kommando, das *string* enthält |
| `!*` | Die Liste der Argumente des letzten Kommandos |
| `^`*alt*`^`*neu*`^` | Wiederhole das letzte Kommando, aber ersetze alle *alt* durch *neu* |

Tabelle 3.7: Abkürzungen zur Wiederholung der letzten Kommandos

### 3.4.10 Vereinfachungen durch die Shell

Um die Eingabe von Befehls- und Dateinamen zu erleichtern, bietet die Shell automatische Ergänzung bereits teilweise eingegebener Namen bei Druck auf die `TAB`-Taste an. Möchte man beispielsweise den relativ langen Befehl
`emacs tex/skript/main.tex`
eingeben, so kann es genügen, die folgenden Tasten zu drücken: `e, m, TAB, t, e, TAB, s, TAB, m, a, i, TAB, t, e, x`. Voraussetzung dafür ist, dass die jeweils eingegebenen Namensanfänge eindeutig sind, es also beispielsweise kein weiteres Programm namens `emacs-old` oder keine Datei `maigloeckchen.eps` im Verzeichnis `tex/skript` gibt.

Sind Dateinamen nicht vollständig eindeutig ergänzbar, so werden sie zumindest so lange ergänzt, wie eine Ergänzung allen Alternativen gemein ist. Dann ist es durch *zweimaligen* Druck auf die `TAB`-Taste möglich, eine Liste aller nun passenden Namen anzeigen zu lassen. Dies ist praktisch, wenn man sich nur noch an den Anfang eines Dateinamens erinnern kann.

Außerdem speichert die Shell automatisch die zuletzt eingegebenen Befehle jedes Benutzers in einer Datei `~/.history`. Den Inhalt dieser Datei kann man durch Eingabe des Befehls `history` sichtbar machen. Ein Aufruf von `history` *n* mit einer natürlichen Zahl *n* als Argument zeigt nur die *n* letzten Befehle an. Auf die in `.history` gespeicherten Befehle kann man sich mittels der in Tabelle 3.7 genannten Abkürzungen beziehen. Beispielsweise ruft
`!l`
das Kommando
`!ls -ld /home/carlos`
auf, wenn dies die letzte mit einem „l“ beginnende Eingabe war.

Zusätzlich ist es möglich, durch das Shell-Kommando `alias` *abk bef* die Zeichenkette *abk* als Abkürzung für den Befehl *bef* zu definieren, um sich unnötige Tipparbeit bei langen oder argumentreichen und häufig benutzten Kommandos zu ersparen. Das folgende Beispiel definiert `re` als Alias für das Löschen von tem-

porären Dateien im Verzeichnis `/var/tmp/mail/ernst`:
`alias re "rm -rf /var/tmp/mail/ernst/*"`
Die Optionen `r` und `f` bei `rm` bewirken, dass Unterverzeichnisse rekursiv gelöscht werden (*recursive*) und nie nachgefragt werden soll, auch dann nicht, wenn Probleme auftreten (*force*). Sie machen den `rm`-Befehl für unerfahrene Anwender sehr gefährlich.

## 3.5 Jobverwaltung

### 3.5.1 Die Jobverwaltung der Shell

Wie bereits oben erwähnt, können unter UNIX mehrere Programme als sog. *Prozesse* oder *Jobs* gleichzeitig laufen. Dabei hat jede Shell einen Vordergrundprozess, mit dem der Benutzer im Shell-Fenster interagieren kann, und beliebig viele Hintergrundprozesse, auf die der Benutzer nicht direkt zugreifen kann, sofern sie kein eigenes Programmfenster öffnen.

Standardmäßig werden Programme im Vordergrund ausgeführt, durch Eingabe eines `&` am Ende eines Kommandos kann eine Ausführung im Hintergrund erreicht werden, sodass weiter mit der Shell gearbeitet werden kann, während das Programm läuft. Hat man ein Programm bereits gestartet, so kann man im ansonsten blockierten Shell-Fenster durch Drücken der Tastenkombination `Ctrl`+`Z` dieses anhalten. In der jetzt wieder frei gewordenen Shell können beliebige Kommandos eingegeben werden, während das angehaltene Programm weiter existiert, aber nicht arbeitet. Will man dessen Arbeit nun wieder fortsetzen, so geschieht dies mit dem Befehl `fg` zur Weiterausführung im Vordergrund bzw. `bg` zur Fortführung im Hintergrund, also wie wenn man das Programm mit `&` aufgerufen hätte.

Durch Aufruf des Shell-Kommandos `jobs` wird eine Liste aller von der aktuellen Shell gestarteten Prozesse angezeigt:

```
[1]  - Running                 gv
[2]  + Angehalten              xfig
[3]    Running                 emacs
```

Jeder dieser Prozesse erhält eine fortlaufende Nummer, auf die man sich durch Voranstellen eines `%` beziehen kann. Um im Beispiel `xfig` im Hintergrund fortzuführen, kann man
`bg %2`
eintippen.

### 3.5.2 Systemweite Prozessverwaltung

Mit dem UNIX-Befehl `ps` werden alle Prozesse angezeigt, die auf einem System zur Zeit laufen. Dazu ist die Option `-e` erforderlich. Zusätzlich bewirkt die Op-

tion `-f` eine ausführlichere Anzeige. Abbildung 3.4 zeigt eine typische Reaktion auf die Eingabe von `ps -ef`. Wie unschwer zu erkennen ist, laufen bereits ohne die Arbeit eines Benutzers zahllose vom System selbst gestartete Prozesse, sog. *daemons*. Sie sind für das automatische Ausführen von Befehlen (`crond`), für die Entgegennahme und Zustellung von E-Mails (`sendmail`) und viele andere Dinge zuständig. Für den täglichen Einsatz sinnvoller ist der Aufruf von `ps` ohne Optionen, das nur die vom Benutzer gestarteten Prozesse anzeigt, und z. B. die folgende Ausgabe liefern könnte:

```
  PID TTY          TIME CMD
 1273 pts/1    00:00:00 bash
 1280 pts/1    00:00:00 gv
 1281 pts/1    00:00:00 ps
```

Im Gegensatz zur Jobverwaltung der Shell mit ihrer eigenen Numerierung für jede Shell wird hier die systemweite Identifikationsnummer *PID* jedes Prozesses angezeigt. Sie wird jedoch seltener benötigt, da die Shell ihre eigenen Nummern in die PID umwandelt, wenn man ein `%` davorstellt.

Der Befehl `kill` *pid* kann verwendet werden, um den Prozess mit der PID *pid* zu beenden. Um `gv` aus dem Beispiel zu beenden, schreibt man also:
`kill 1280`
oder
`kill %1`
wenn 1 die Jobnummer von `gv` ist.

`kill` kann allgemein verwendet werden, um sog. *Interrupts* an einen Prozess zu senden. Wenn man keinen Interrupt explizit angibt, wird der Interrupt `15` namens `SIGTERM` versandt, der den Prozess auffordert, sich zu beenden. Prozesse, die nicht richtig funktionieren („abgestürzt sind“) müssen gewaltsam beendet werden, hierzu benutzt man den Interrupt `9`, der für `SIGKILL` steht. Die allgemeine Syntax für das Versenden von Interrupts lautet `kill` *-intnr pid*. Sollte `gv` also wg. eines Programmfehlers nicht auf `SIGTERM` reagieren, so ist zu schreiben:
`kill -9 1280`
Die Nummern *intnr* aller verfügbaren Interrupts kann man der Ausgabe des Kommandos `kill -l` entnehmen. Ein Beispiel dafür zeigt Abbildung 3.5. Die meisten dieser Interrupts wird man nie benötigen, eine Auswahl der wichtigsten ist deshalb in Tabelle 3.8 aufgeführt.

Wie aus der Tabelle hervorgeht, kann man, um im Vordergrund laufende Prozesse zu beenden, auch `Ctrl`+`C` drücken. Im Unterschied zu `Ctrl`+`Z` wird der Prozess dadurch jedoch nicht nur angehalten, sondern tatsächlich beendet.

Es kann vorkommen, dass man bei Benutzung der Shell versehentlich die Tastenkombination `Ctrl`+`S` drückt. Hierauf ignoriert die Shell weitere Tastatureingaben. Um weiterarbeiten zu können, drückt man einfach `Ctrl`+`Q`.

```
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:16 init
    2 ?        SW     0:00 [keventd]
    3 ?        SWN    0:00 [ksoftirqd_CPU0]
    4 ?        SWN    0:00 [ksoftirqd_CPU1]
    5 ?        SW     0:00 [kswapd]
    6 ?        SW     0:00 [bdflush]
    7 ?        SW     0:14 [kupdated]
   10 ?        SW     0:00 [khubd]
  138 ?        SW     0:22 [kjournald]
  468 ?        S      0:02 syslogd -m 0
  473 ?        S      0:00 klogd -2
  493 ?        S      0:03 portmap
  521 ?        S      0:00 rpc.statd
  637 ?        SL     0:26 ntpd -U ntp
  664 ?        S      0:06 ypbind
  743 ?        S      0:00 /usr/sbin/automount /auto.all file /etc/auto.all
  745 ?        S      0:00 /usr/sbin/automount /auto.linux file /etc/auto.linux
  777 ?        S      0:00 xinetd -stayalive -reuse -pidfile /var/run/xinetd.pid
  824 ?        S      0:02 rwhod
  866 ?        S      0:00 gpm -t ps/2 -m /dev/mouse
  884 ?        S      0:01 crond
  932 ?        S      0:00 xfs -droppriv -daemon
  949 ?        S      0:03 /usr/sbin/sshd
  967 ?        S      0:00 /usr/sbin/atd
  974 tty1     S      0:00 /sbin/mingetty tty1
  975 tty2     S      0:00 /sbin/mingetty tty2
  976 tty3     S      0:00 /sbin/mingetty tty3
  977 tty4     S      0:00 /sbin/mingetty tty4
  978 tty5     S      0:00 /sbin/mingetty tty5
  979 tty6     S      0:00 /sbin/mingetty tty6
 2658 ?        SW     0:29 [rpciod]
 2659 ?        SW     0:00 [lockd]
24900 ?        S      0:40 /usr/sbin/sshd
27244 ?        S      0:00 sendmail : accepting connections
27281 ?        S      0:00 lpd Waiting
```

Abbildung 3.4: `ps -ef` zeigt eine ausführliche Liste aller auf einem System laufenden Prozesse an.

| **Nummer** | **Name** | **Bedeutung** |
|---|---|---|
| 2 | SIGINT | Unterbrechung wie mit **Ctrl**+**C** |
| 9 | SIGKILL | Sofortiges Beenden |
| 15 | SIGTERM | Aufforderung zum Beenden |
| 17 | SIGSTOP | Prozess anhalten, wie **Ctrl**+**Z** |
| 19 | SIGCONT | Angehaltenen Prozess fortsetzen |

Tabelle 3.8: Die wichtigsten Interrupts und ihre Nummern

```
 1) SIGHUP     2) SIGINT     3) SIGQUIT    4) SIGILL
 5) SIGTRAP    6) SIGABRT    7) SIGBUS     8) SIGFPE
 9) SIGKILL   10) SIGBUS    11) SIGSEGV   12) SIGSYS
13) SIGPIPE   14) SIGALRM   15) SIGTERM   16) SIGURG
17) SIGSTOP   18) SIGTSTP   19) SIGCONT   20) SIGCHLD
21) SIGTTIN   22) SIGTTOU   23) SIGIO     24) SIGXCPU
25) SIGXFSZ   26) SIGVTALRM 27) SIGPROF   28) SIGWINCH
29) SIGINFO   30) SIGUSR1   31) SIGUSR2
```

Abbildung 3.5: Das Kommando `kill -l` liefert eine Liste aller auf dem System bekannten Interrupts.

### 3.5.3 Ändern der Priorität

Die Priorität jedes Prozesses legt fest, wie viel Prozessorzeit er im Verhältnis zu anderen Prozessen erhält. Prozesse mit hoher Priorität laufen deshalb meist schneller und lassen sich weniger durch die Existenz anderer Prozesse stören als Prozesse mit niedriger Priorität. Die Priorität ist eine Zahl zwischen -20 (höchste Priorität) und 19 (niedrigste priorität). Nur dem Systemadministrator `root` ist es erlaubt, Prioritäten kleiner 0 zu setzen.

Ein guter Weg, um Prioritäten zu beobachten, ist der Aufruf des Kommandos `top`. Dieser Befehl zeigt die momentan laufenden Prozesse interaktiv nach Resourcenverbrauch geordnet an. Die Prioritäten werden dabei in der Spalte `PRI` ausgedruckt. Um `top` wieder zu verlassen drückt man die Taste `Q`, Hilfe zu den zahlreichen weiteren Funktionen erhält man durch Tippen von `H`.

Um einen Befehl mit einer anderen als der Standardpriorität zu starten, startet man ihn mit dem Kommando `nice` *pri cmd arg*. Dabei ist *pri* die Priorität, die durch eine Zahl zwischen `+0` und `+19` (für `root` `-20` bis `+19`) mit vorangestelltem Vorzeichen dargestellt wird. *cmd* und *arg* stehen für das eigentlich auszuführende Kommando mit seinen Argumenten. Beispiel:

```
nice +15 latex liebe.tex
```

Die Priorität eines bereits gestarteten Prozesses lässt sich durch `renice` ändern. Normale Benutzer dürfen die Werte dabei nur erhöhen, also die Priorität verringern; damit soll verhindert werden, dass Herabstufungen von Seiten des Systemadministrators wieder rückgängig gemacht werden können. Die Syntax von `renice` ist die selbe wie die von `nice` mit dem Unterschied, dass statt des zu startenden Kommandos die *PID* des Prozesses steht, z. B. also:

```
renice +9 1280
```

Das Zurücksetzen von Prioritäten dient vor allem dem Zweck, Prozesse, die langwierige Berechnungen durchführen, davon abzuhalten, die Maschine für Benutzer weitgehend zu blockieren. Wenn eine Berechnung sowieso Tage in An-

spruch nimmt, ist es akzeptabel, sie nur nachts oder während Arbeitspausen laufen zu lassen, während keine interaktiven Benutzerprozesse mit höherer Priorität tätig sind.

In solchen Fällen muss zusätzlich dafür gesorgt werden, dass ein Prozess auch nach dem Abmelden des Benutzers, der ihn gestartet hat, weiterläuft. Normalerweise würde er nämlich spätestens zu diesem Zeitpunkt vom System beendet. Dazu dient das Kommando `nohup`. Es leitet zusätzlich—da ja kein Terminal zur Verfügung steht—die Ausgabe des Programms in die Datei `nohup.out` um. `nohup` kann folgendermaßen angewendet werden, um einen aufwändigen Job „über Nacht“ laufen zu lassen:

```
nohup latex grosse-liebe.tex &
```

## 3.6 Netzwerk

In aller Regel sind UNIX-Computer in eine Netzwerkstruktur integriert. In dieser Netzwerkstruktur gibt es spezialisierte Rechner für verschiedene Aufgaben, z. B. Fileserver mit großen Hochleistungsfestplatten, die Daten für alle Anwender zur Verfügung stellen oder sichern, oder Mailserver zur Weiterleitung von E-Mails. Es reicht aus, wenn innerhalb eines solchen Netzwerks nur einer oder wenige Geräte wie Drucker oder Scanner angeschlossen sind, auf die jedoch von allen Arbeitsplatzrechnern aus zugegriffen werden kann. Die Schnittstelle zum Internet wird aus Sicherheitsgründen durch einen speziell konfigurierten Rechner, eine sog. *firewall* abgeschirmt, sodass genau festgelegt werden kann, welche Daten nach außen gelangen und welche Verbindungen von außen in das lokale Netz aufgebaut werden dürfen.

Innerhalb einer solchen Struktur besitzt jedes Gerät eine sog. *IP-Nummer*, diese könnte z. B. folgendermaßen aussehen:

```
129.69.120.13
```

Die vier Zahlen bilden eine hierarchische Gliederung, die von links nach rechts zu lesen ist. `129` bildet eine lokale Festlegung, `69` steht beispielsweise für die Universität Stuttgart, `120` für ein bestimmtes Institut und `13` für einen bestimmten Rechner innerhalb dieses Institutes.

Weil solche Adressen nicht besonders einprägsam sind, gibt es eine weitere Möglichkeit, Rechner in einem Netzwerk zu benennen:

```
hase.ica1.uni-stuttgart.de
```

Solche Namen als Adressen sind genau andersherum zu lesen, hier ist `hase` der Rechnername, `ica1` das Institut, `uni-stuttgart` die *Domäne* und `de` die nationale Festlegung.

Wenn man selbst an einem wenig leistungsstarken Rechner innerhalb eines Netzwerkes sitzt und mehr Rechenleistung für ein Programm benötigt, so kann man sich über das Netzwerk auf einem schnelleren Computer anmelden und dort arbeiten. Dies ist mit dem Kommando `rlogin` *adresse* möglich, wobei *adresse* die

IP-Nummer oder der Name des gewünschten Rechners ist. Ähnliches geschieht durch den Befehl `telnet`.

Auf diese Weise können mehrere Benutzer gleichzeitg am selben Computer eingeloggt sein und arbeiten. Diese können mittels `write` *user* Nachrichten untereinander austauschen. *user* steht für den Namen eines anderen am selben Gerät angemeldeten Benutzers, der die Nachricht erhalten soll. Nach Eingabe des Kommandos können beliebig viele Textzeilen geschrieben werden, die jeweils beim Sprung in eine neue Zeile mit der `Return`-Taste übertragen und in der Shell des Adressaten angezeigt werden. Das Schreiben der Nachricht wird beendet, indem man ein Dateiende simuliert, was durch `Ctrl`+`D` geschieht. Tippt man statt `write` *user* den Befehl `wall`, so wird die Nachricht an alle momentan angemeldeten Benutzer versandt.

Jeder kann selbst bestimmen, ob er derartige Nachrichten lesen möchte. Dafür ist das Kommando `mesg` zuständig. Je nachdem, ob man als Argument `y` oder `n` angibt, werden Nachrichten angezeigt oder nicht.

Um Mitteilungen an Benutzer, die auf anderen Rechnern angemeldet sind, zu verschicken, existiert der Befehl `talk`.

Alle diese Befehle funktionieren nur, wenn der Empfänger der Nachricht gerade eingeloggt ist. Mittels des Programms `mail` können Nachrichten als herkömmliche E-Mails versandt werden. Als Argument erwartet `mail` nicht den Benutzernamen, sondern die E-Mail-Adresse des Adressaten (was innerhalb eines lokalen Netzwerks allerdings oft identisch ist). Wiederum wird die Eingabe des Textes durch ein `Ctrl`+`D` abgeschlossen. In der Regel wird man zum Schreiben von Mails jedoch auf „gewöhnliche“ Mailprogramme wie Mutt oder Netscape zurückgreifen, die u. a. Funktionen zum Archivieren von Nachrichten bieten.

Möchte man erfahren, wer momentan im Netzwerk angemeldet ist, so tippt man den Befehl `rwho`. Mittels `finger` *user* lassen sich weitere Informationen über den Benutzer *user* in Erfahrung bringen. `finger` ohne Argument zeigt eine Liste aller Benutzer des Netzwerks an. Um zu ermitteln, wie lange die Rechner eines Netzwerks bereits ohne Unterbrechung laufen, kann man den Befehl `ruptime` benutzen. Analog zu `rwho` und `ruptime` gibt es die Kommandos `who` und `uptime` zur Abfrage der Benutzer des lokalen Systems bzw. dessen „*uptime*“.

Es ist auch möglich, vorausgesetzt man kennt das entsprechende Passwort, sich unter dem Namen eines anderen Benutzers anzumelden. Hierfür schreibt man z. B.

`su marc`.

Nach der Aufforderung, Marcs Kennwort einzugeben, kann man im selben Shell-Fenster unter dem Namen `marc` arbeiten und so z. B. Marcs persönliche Dateien betrachten und löschen.

Gibt man zusätzlich einen Strich als Argument an, also

`su - marc`,

so wird die Aktion behandelt, als ob man sich direkt als `marc` eingeloggt hätte, d. h. die persönlichen Skriptdateien von Marc wie z. B. `.profile` werden abgear-

beitet, sodass im Gegensatz zum vorherigen Aufruf jetzt auch alle Shell-Variablen und Aliase so definiert sind, wie Marc sie für sich konfiguriert hat.

In beiden Fällen kehrt man durch Eingabe von `exit` oder kurz `Ctrl`+`D` wieder in die eigene Shell bzw. unter seine eigene Benutzerkennung zurück.

Um Dateien zwischen Rechnern auszutauschen, steht das Programm `ftp` zur Verfügung. Mit `ftp` *benutzername*`@`*rechnername* baut man eine Verbindung zu *rechnername* auf und kann dann mittels spezieller ftp-Kommandos Dateien hoch- oder herunterladen.

`ftp` sowie die oben genannten Kommandos `telnet` und `rlogin` haben den Nachteil, dass bei ihnen während der Anmeldung das Kennwort des Benutzers über das Netzwerk offen übertragen wird. Modernere Standards ermöglichen eine Anmeldung mittels verschlüsselter Passwörter, die erheblich sicherer ist. Diese Anmeldemethode wird von den Programmen `ssh` als Ersatz für `telnet` und `sftp` als Ersatz für `ftp` bei vergleichbarer Benutzung unterstützt. Wann immer möglich sollten diese Programme vorgezogen werden.

# Kapitel 4

# Textverarbeitung

Zur Textverarbeitung stehen spezifische und allgemein nutzbare Programme zur Verfügung. Word von Windows beispielsweise ist auf das Drucken von Texten spezialisiert. Der geschriebene Text wird am Bildschirm laufend so gezeigt, wie er gedruckt wird. Das Programm ist sehr leicht zu bedienen, da sich die Symbole von selbst erklären. Allerdings wird die Bearbeitung des Textes über die Buttons immer zeitaufwändiger, je mehr Unterteilungen und Sonderzeichen im Text verwendet werden. Im Gegensatz dazu stehen verschiedene Texteditoren, bei denen am Bildschirm nur eine Schriftart in einer bestimmten Größe angezeigt wird. Diese lassen sich aber wesentlich allgemeiner einsetzen, da neben Texten auch Programme, Shellscripte und sogar Musikdateien geschrieben werden können.

## 4.1 Verschiedene Editoren

Grundsätzlich lassen sich Editoren je nach ihrer Benutzeroberfläche in die drei Gruppen, Zeilen-, Bildschirm- und graphische Editoren einteilen:

- Zeileneditoren wie 'ed', bei denen immer nur eine Textzeile angezeigt wird und bearbeitet werden kann. Die ersten Editoren arbeiteten nach diesem Prinzip, das aber heute kaum noch Verwendung findet.

- Bildschirmeditoren wie 'Vi' und 'Emacs', bei denen eine ganze Seite Text sichtbar ist und auch bearbeitet werden kann. Der erste ist ein sehr allgemeiner Editor, der fast überall intstalliert ist und mit dem Befehl 'ESC+w+q' beendet wird. Von 'Emacs' gibt es verschiedene Ausführungen wie die Freeware-Version 'Gnuemacs' oder die benutzerfreundlich bunt gestaltete Version 'Xemacs'. Die Bedienung von Emacs wird später noch genauer erklärt.

- Graphische Editoren ermöglichen das intensive Arbeiten mit der Maus. Ein Beispiel hierfür ist 'Nedit', der im Internet unter *http://www.ftp.nlal.gov* heruntergeladen werden kann.

## 4.2 Funktionsweise von LaTex

Um einen im Editor verfassten Text anzusehen und auszudrucken wird ein Programm benötigt, das die Layoutbefehle des Quelltextes erkennen und die Datei in eine Graphik übersetzten kann. Besonders häufig wird hierfür das Programm "LaTex" verwendet, das die Methoden von Programmiersprachen benutzt, um Textdokumente zu erstellen. Die Erstellung der druckbaren Textdatei 'liebe.ps' aus einer im Editor Emacs geschriebenen Latex-Datei läuft folgendermaßen ab:

- Der im Editor erstellte Quelltext mit Latex-Befehlen wird unter dem Namen 'liebe.tex' abgespeichert.
- Mit dem Befehl 'latex liebe.tex' im X-term wird die Datei interpretiert und als 'liebe.dvi' ausgegeben.
- Die DVI-Datei wird mit dem Befehl 'dvips liebe.dvi' in eine Postscriptdatei umgewandelt.
- Die entstandene Datei 'liebe.ps' kann jetzt mit dem Befehl 'lp liebe.ps' gedruckt oder mittels 'gv liebe.ps' am Bildschirm angezeigt werden.

Wenn die Datei 'liebe.tex' folgenden Quelltext enthält,

```
\documentclass{report}
\usepackage{a4}
\usepackage{german}

\begin{document}
\paragraph{Hallo Welt,\\}
ich liebe Dich!
\end{document}
```

so würde die nach obigem Muster erzeugte Datei 'liebe.ps' am oberen linken Rand einer DIN-A4 Seite folgenden Schriftzug enthalten:

**Hallo Welt,**
ich liebe Dich!

## 4.3 Zur Arbeit im Editor Emacs

Die Steuerbefehle werden bei den meisten Emacs-Versionen nicht durch Klicken auf Symbole, sondern durch die Eingabe von Tastenkomandos erteilt. Eine auch ohne Vorkenntnisse leicht verständliche Erklärung der Kommandos ist das Emacs-online-Tutorial, erreichbar mittels der Tastenkombination `Ctrl` + `H`, `T`. Wichtige

Tastenkombinationen für den Umgang mit mehreren Buffern, den Fenstern in welchen gearbeitet wird, sind in Tabelle 4.1 genannt. Tabelle 4.2 listet wichtige Emacs-Befehle im Umgang mit Dateien auf. In Tabelle 4.3 sind schließlich noch Kommandos zum Markieren und Kopieren von Textpassagen aufgeführt. Alle drei Tabellen benutzen folgende Abkürzungen:

`C`: `Ctrl-` oder `Strg`-Taste (und zwar die linke)
`M`: `ALT-` oder `Meta`-Taste (und zwar die linke - nicht Alt Gr)
`RET`: Return-Taste

| Tastenkombination oder: `M-x` + Kommandoname | LISP-Kommando | Beschreibung |
|---|---|---|
| `C-x C-c` | exit | Beendet Emacs |
| `C-x b` | switch-to-buffer | Auf einen anderen Buffer umschalten |
| `C-x C-b` | list-buffers | Halbiert das aktuelle Fenster und zeigt alle verfügbaren Buffer im neuen Fenster an. |
| `C-x k` | kill-buffer | Einen Buffer schließen |
| `C-x C-k` | save-buffer | Einen Buffer speichern |
| `C-x s` | save-some-buffer | Frägt bei jedem Buffer nach, der verändert wurde, ob er gespeichert werden soll. Eingabe von "!" bedeutet "Ja für alle" |

Tabelle 4.1: Tastenkombinationen für den Umgang mit verschiedenen Buffern im Emacs

| Tastenkombination oder: `M-x` + Kommandoname | LISP-Kommando | Beschreibung |
|---|---|---|
| `C-x C-f` | find-file | Öffnet eine Datei.<br>Im Minibuffer wird bereits ein Vorschlag für einen Pfad angegeben. Die Tab-Taste bringt eine Auflistung aller Dateien unter diesem Pfad. |
| `C-x C-s` | save-buffer | Speichert den gerade aktiven Buffer (also den, in dem sich der Cursor befindet) |
| `C-x C-w` | write-file | Speichert den aktuellen Buffer unter einem anderen Namen |

Tabelle 4.2: Tastenkombinationen zum Öffnen und Schließen von Dateien im Emacs

| Tastenkombination oder: `M-x` + Kommandoname | LISP-Kommando | Beschreibung |
|---|---|---|
| `C-d` oder `Entf` | delete-char | Löscht das nächste Zeichen |
| Backspace | delete-backward-char | Löscht das vorhergehende Zeichen |
| `M-d` oder `ESC-d` | kill-word | Löscht das nächste Wort |
| `M`-Backspace oder `ESC`-Backspace | backward-kill-word | Löscht das vorhergehende Wort |
| `C-k` | kill-line | Löscht eine Zeile |
| `C`-Leertaste | set-mark-command | Setzt den Beginn einer Markierung. Zum Markieren eines Textbereiches muss der Beginn markiert werden. Das Ende der Markierung liegt immer am Cursor. Eine Markierung ist immer so lange aktiv, bis entweder `C-g` oder ein Kopier- oder Lösch-Befehl ausgeführt wurde. Ist die Markierung nicht sichtbar (nicht hinterlegt), dann gibt man "`M-x` transient-mark-mode" ein und die Markierung wird sichtbar. |
| `C-w` | kill-region | Löscht die Markierung und legt sie ins Clipoard (enspricht `C-x` unter Fensters) |
| `M-w` | kill-ring-save | Kopiert die Markierung (entspricht `C-c` unter Fensters) |
| `C-y` | yank | Fügt die letzte kopierte / gelöschte Markierung an der aktuellen Position ein. Zu den einzufügenden Regionen gehören z.B. mit `C-w` oder `C-k` gelöschte Bereiche. |
| `M-y` | yank-pop | Fügt eine frühere Markierung ein. Emacs speichert alle ins Clipboard kopierten Regionen intern und wenn man an einer Stelle eine Region einfügen will, die man nicht beim letzten Kopieren / Löschen ins Clipboard gebracht hat, so gibt man `C-y` für die letzte Kopie und dann `M-y` für die vorletzte ein. |

Tabelle 4.3: Tastenkombinationen zum Löschen, Markieren und Einfügen von Text im Emacs

# Kapitel 5

# Meßdatenvisualisierung am Computer

## 5.1 Allgemeines

### 5.1.1 Zielsetzung und Mittel

Computervisualisierung kann einer Vielzahl von Zwecken dienen. Eine Auswahl davon ist hier aufgelistet:

- Sie hilft durch grafische Darstellung beim Auswerten und Durchsuchen von Daten.
- Sie verhilft zu neuen Einsichten, z. B. Röntgenaufnahmen.
- Sie verbessert das Verständnis von Prozessen, Konzepten, Informationen, z. B. Flussdiagramme.
- Sie stellt sonst Unsichtbares dar, z. B. Atommodelle.
- Sie dient als Medium zur Kommunikation, z. B. in der Lehre.

Bei der Ausgabe unterscheidet man zwischen zwei- und dreidimensionalen Medien. Wichtige Ausgabemedien sind der Bildschirm oder der Ausdruck auf Papier. Andere Möglichkeiten sind z. B. Ausbelichtung auf Film (für zwei Dimensionen) und Hologramme, Stereoprojektionen und Modelle, die sowohl am Computer simuliert als auch real hergestellt werden können (in drei Dimensionen).

In der Darstellung bedient man sich verschiedener Techniken wie Kennzeichnung durch Farbe, Symbole oder Texturen, geeignete Skalierung oder auch Animationen, um aussagekräftige Grafiken zu erhalten.

### 5.1.2 Arbeitsschritte bei der Visualisierung

In Abbildung 5.1 ist der typische Arbeitsablauf vom Erzeugen der darzustellenden Daten bis zum fertigen Ergebnis gezeigt. Dies ist ein Beispiel für den Einsatz von Computergrafik zur Visualisierung eines Prozesses.

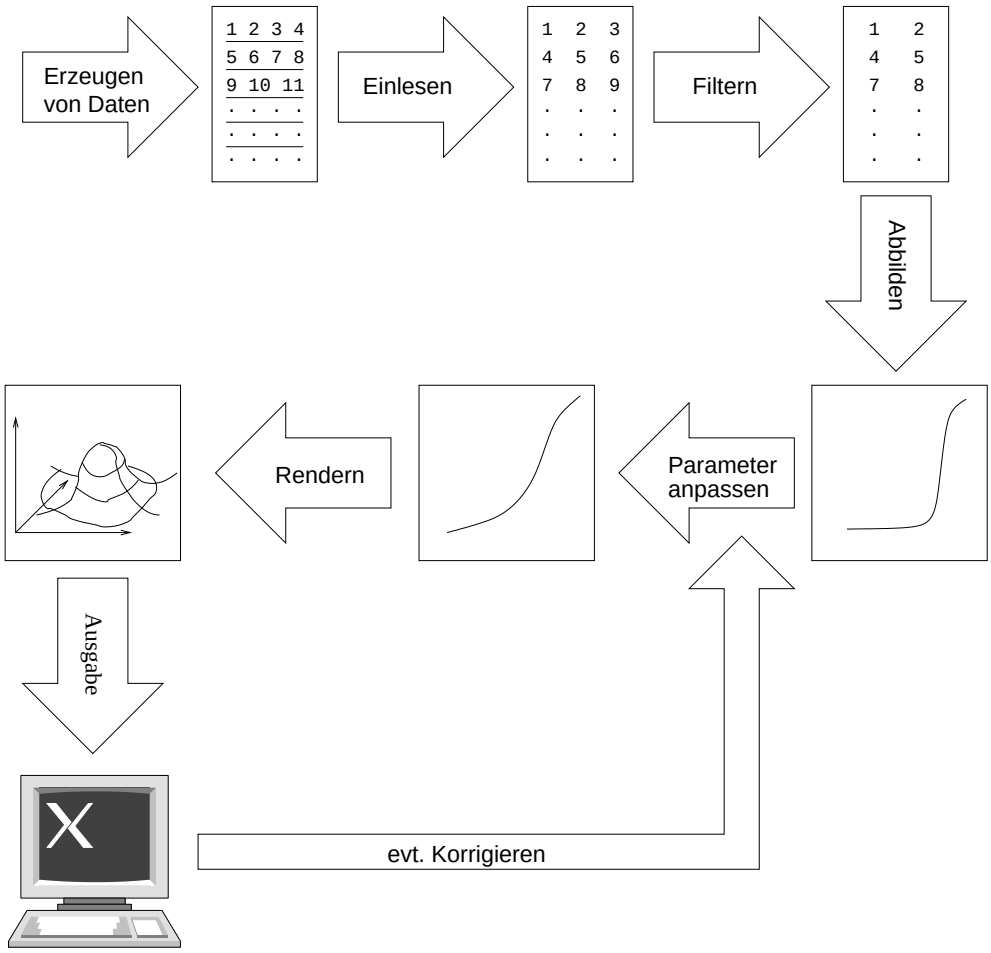



Abbildung 5.1: Arbeitsschritte bei der Erzeugung von Computergrafik

Zuerst müssen die Daten, die z. B. von einer Computersimulation errechnet oder in einer Messanordnung bestimmt wurden, in den Computer bzw. in das Visualisierungsprogramm eingelesen werden. Oft sind diese Daten sehr umfangreich, sodass das Herausfiltern bestimmter Aspekte nötig ist. Die in den Daten enthaltene Information wird als nächstes in einem geeigneten Datenmodell abgebildet. Meist muss man zahlreiche Parameter anpassen, um den eigentlichen Sachverhalt erst darstellbar zu machen. Ist man damit fertig, kann man von der Software ein mehr oder weniger aufwändiges Bild rendern lassen, das Ziel des gesamten Arbeitsablaufes. In der Praxis wird man bei Betrachtung der Ausgabe

allerdings fast immer Verbesserungsmöglichkeiten finden, was eine Korrektur der Einstellungen und erneutes Rendern zur Folge hat.

### 5.1.3 Datengewinnung und Techniken der Datenvisualisierung

Die Erzeugung der Daten erfolgt i. d. R. durch eine der folgenden Methoden:

- Messung an einem Versuchsaufbau (häufig automatisiert)
- Computersimulation oder -modellierung
- bildgebende Verfahren (Filmaufnahmen, Computertomographie, Scanner, ...)

Diese Daten müssen nun bearbeitet werden, um sinnvoll dargestellt werden zu können. Wichtig ist dabei, dass sie nur aufbereitet, nicht jedoch verfälscht werden dürfen. Aufbereitung kann nötig sein, um

- bei der Messung entstandenes Hintergrundrauschen zu entfernen.
- zusätzliche Daten zwischen diskreten Messpunkten zu interpolieren.
- die verschiedene Datensätze mittels Normierung vergleichbar zu machen.
- die eigentlich wichtigen Phänomene freizustellen.

Es ist oft sinnvoll, die eigenen Formate auf ein generisches Datenmodell abzubilden. Diese legen oft bereits eine bestimmte Visualisierungstechnik nahe. Ausgewählte Datenmodelle sind z. B.:

**Nominales Datenmodell** Die Daten repräsentieren verschiedene Objekte, Elemente einer Gruppe, usw., z. B. einzelne Bundesländer. In der Visualisierung bietet sich eine Kodierung durch bestimmte Farben oder Symbole für jeden Datenpunkt an.

**Geordnetes Datenmodell** Im geordneten Datenmodell liegen die Daten in einer bestimmten Reihenfolge zueinander vor. Diese kann z. B. eine alphabetische Reihenfolge sein, oder sich daraus ergeben, dass die Werte monoton steigen, ohne dass jedoch wie beim quantitativen Datenmodell (s. u.) zwingend Informationen über die absoluten Werte vorhanden sein müssten. Zur Darstellung bietet sich eine Skala oder aber eine Kodierung durch Heligkeit, Größe oder Farbton an.

**Quantitatives Datenmodell** Jeder Datenpunkt stellt einen bestimmten quantitativen Wert dar, z. B. 2.34 oder 5.7. Diese Werte lassen sich leicht durch Positionen innerhalb eines Koordinatensystems oder Längen darstellen.

Für eine gute Grafik ist es ist wichtig, sich für die jeweils anschaulichste Visualisierungsform zu entscheiden, um dem Verwendungszweck der Grafik sowie der gewünschten Aussage bestmöglich gerecht zu werden. Als Beispiele seien hier genannt:

**Punkt („Scatter Plot“)** Bei einem Scatterplot wird jedes Datenelement lediglich durch einen Punkt in einem Koordinatensystem dargestellt. Die so entstehenden Punkte werden nicht durch Linien zu einem Graphen verbunden. Das wäre auch selten sinnvoll, da man Scatter Plots meist für Daten anwendet, bei denen es nicht um den Verlauf einer Funktion in Abhängigkeit einer Veränderlichen geht, sondern um das Verhältnis verschiedener Messgrößen zueinander und deren Häufigkeitsverteilung bei einer endlichen Zahl von Messungen.

Abbildung 5.2 als Beispiel für einen Scatter Plot stellt die Abmessungen von Blättern in Relation zueinander dar. Aus den verschiedenen Häufungspunkten kann man nun auf unterschiedliche Baumarten schließen oder eine Durchschnittsgröße ermitteln.

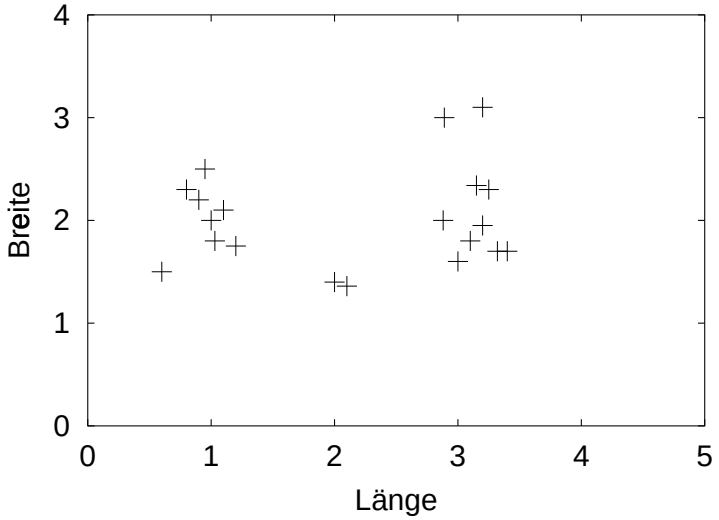



Abbildung 5.2: Die Größe verschiedener Blätter, aufgetragen in einem Scatter Plot

**Skalar** Dies ist die häufigste Form der Visualisierung. Jedes Datenelement steht für einen numerischen Wert, der an seiner Position ablesbar ist. Hiermit kann man z. B. auch Daten wie die Wahlergebnisse im Verlauf der letzten Jahre o. ä. verdeutlichen.

**Vektor / Tensor** Eine allgemeingültige Beschreibung dieser Visualisierungsvariante ist schwer möglich. Brauchbarkeit und Bedeutung ergeben sich un-

mittelbar aus der jeweiligen Anwendung. Sinnvoll wäre es beispielsweise, zur Darstellung eines Wirbels in einem Fluss die Ortskoordinaten in einem gewöhnlichen zwei- oder dreidimensionalen Koordinatensystem aufzutragen und die Strömungsgeschwindigkeit an ausgewählten Orten durch Pfeile mit zu dieser proportionalen Länge einzuzeichnen.

**Kontinuierlich** Eine kontinuierliche Darstellung in Form eines Graphen ist natürlich nur möglich, wenn man entweder eine mathematische Relation der Koordinaten zueinander oder aber ausreichend viele und dicht positionierte Messpunkte hat. In beiden Fällen hilft eine kontinuierliche Darstellung bei der mathematischen Betrachtung der Daten.

## 5.2 Auswahl der Visualisierungssoftware

### 5.2.1 Vektor- und Bitmap-Grafik

Um Grafiken am Computer darzustellen und zu speichern, gibt es im Wesentlichen zwei Möglichkeiten: Vektor- und Bitmapgrafik. Dies soll zunächst am Beispiel eines Kreises verdeutlicht werden.

Um einen Kreis eindeutig zu beschreiben benötigt man laut Mathematik die Koordinaten des Mittelpunkts und den Radius. Im Zweidimensionalen also z. B. das Zahlentripel (5, 4, 1.5) verbunden mit der Information, dass die ersten beiden Zahlen als x- und y-Koordinate des Mittelpunkts und die dritte Zahl als Radius zu interpretieren ist. Ein Programm könnte das als Zeichenkette „`circle(5, 4, 1.5)`“ in einer Datei speichern und, wenn es diese zu einem späteren Zeitpunkt einliest, daraus wieder exakt dieselbe Abbildung generieren. Für komplexere Grafiken kann man leicht weitere Codes entwickeln, die auf die selbe Weise Punkte, Linien, Pfeile und auch Beschriftungselemente in einer parametrisierten Form darstellen können. Diese Form der Grafikbeschreibung bezeichnet man als *Vektorgrafik*.

Die zweite Möglichkeit, die Bitmap-Grafik genannt wird, besteht darin, die Zeichnung in ein feines rechteckiges Raster zu zerlegen um dann für jedes Feld dieses Rasters (ein sog. *Pixel*) eine Zahl zu speichern, die die in diesem Feld überwiegende Farbe kodiert. Will man nun einen Kreis beschreiben, so sieht dies (bei entsprechend grobem Raster, also geringer Auflösung) folgendermaßen aus:

Es ist offensichtlich, dass für das Beispiel des Kreises Vektorgrafik bei geringerem Speichervolumen höhere Bildqualität (nämlich einen tatsächlich kreisrunden Kreis im Gegensatz zu einer groben Annäherung) bietet. Darüberhinaus kann man Vektorgrafiken für den Druck ohne Qualitätseinbußen beliebig skalieren, wohingegen bei Bitmaps durch Vergrößerung die Rasterstruktur unschön sichtbar gemacht wird.

Dennoch hat Bitmap-Grafik ihre Berechtigung, und zwar dann, wenn es um das Speichern hochkomplexer Grafiken mit vielen Details und zahllosen Farbab-

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |

Abbildung 5.3: Die Bitmap eines Kreises, wobei 1 für die Vorder- und 0 für die Hintergrundfarbe steht.

stufungen wie z. B. Fotos geht. In solchen Fällen wäre die Darstellung mittels tausender einzelner „Zeichenbefehle“ wie `circle` aus dem obigen Beispiel ungleich aufwändiger als mit einer Bitmap, selbst wenn man berücksichtigt, dass für befriedigende Druckausgabe möglicherweise eine sehr hohe Auflösung mit resultierenden Dateigrößen im Bereich mehrerer 10 Megabyte erforderlich ist.

Beispiele für Vektorgrafikformate sind (Encapsulated) Postscript (PS / EPS), PDF, CGM oder WMF. Die meisten dieser Formate sind trotz ihres Vektorgrafikcharakters in der Lage, Bitmap-Grafiken einzubetten, was z. B. für das Einfügen von Fotos als Illustration zu einem Text nützlich ist. Häufig verwendete Formate für Bitmaps sind TIFF, GIF und sein „Nachfolger“ PNG, JPEG und PBM.

### 5.2.2 Einführung in `gnuplot`

Um aufbereitete und gefilterte Daten aus einer Datei in der gewünschten Weise darstellen zu können gibt es verschiedene Grafikprogramme. Eines davon heißt gnuplot und wird in diesem Kapitel vorgestellt. Aufgerufen wird es von einem UNIX-Xterm aus mit dem Befehl `gnuplot`, daraufhin erscheint eine gnuplot-Befehlseingabe. Der Wechsel zurück zur UNIX-Eingabe und das Schließen von gnuplot erfolgt durch die Eingabe von `exit`. Eine allgemeine Einführung und eine Liste der genauer erläuterten Themen in englischer Sprache kann in gnuplot durch `help` aufgerufen werden. Im Folgenden werden einige gnuplot-Befehle genannt und kurz erläutert.

**Darstellen der Messwerte aus einer Datei:**

- `plot "Dateiname"`
  Zeigt in einem extra Fenster am Bildschirm 2 Koordinatenachsen mit automatischer Skalierung, die Messwerte sind darin als Punkte aufgetragen; Voraussetzung: in der Datei "Dateiname" sind die Messwerte in senkrechten Spalten gespeichert, die 1. Spalte wird zur X-Achse, die 2. zur Y-Achse, weitere Spalten bleiben unberücksichtigt.

- `plot "Dateiname1", plot "Dateiname2"`
  Zwei Messpunktreihen aus zwei verschiedenen Dateien werden in einem Koordinatensystem dargestellt. Es ist auch möglich noch mehr Kurven gemeinsam darzustellen.

- `plot "Dateiname" with lines`
  Die Messpunkte werden zu einer durchgehenden Linie verbunden, die Kurzschreibweise hierfür ist: `w l`

- `plot "Dateiname" w lp`
  Die einzelnen Messpunkte werden auf der durchgehenden Linie als Punkte markiert.

- `plot "Dateiname" w err`
  Die Werte der 3. Spalte in der Datei "Dateiname" werden senkrecht über und unter den Messpunkten als Fehlerbalken aufgetragen. Die Endung `w xerr` erzeugt Fehlerbalken in X-Richtung.

- `replot`
  Dieser Befehl wiederholt den letzten `plot`-Befehl. Dies ist immer dann nötig, wenn Beschriftungen (z.B. `xlabel`) hinzugefügt oder Einstellungen (z.B. `xrange`) verändert wurden.

**Beschriftungen:**

- `plot "Dateiname" title "Meßkurve"`
  Am rechten oberen Rand des Koordinatensystems erscheint der Titel "Meßkurve" mit einer Farbzuordnung. Ohne diesen Zusatz erscheint an der gleichen Stelle immer der Dateiname.

- `set xlabel "Zeit"`
  Bei allen folgenden Darstellungen wird die X-Achse mit "Zeit" beschriftet. Danach `replot` nicht vergessen.

- `set ylabel "Dichte /Symbol r"`
  Diese Anweisung ordnet der Y-Achse den Ausdruck "Dichte $\rho$" zu. Sonderzeichen können allerdings nicht im Xterm, sondern nur z.B. in eps-Dateien ausgegeben werden.

- `set label "I=1" at 1000,0.96`
  Der Ausdruck `I=1` wird am Punkt (1000/0,96) ins Koordinatensystem eingefügt.

**Einstellung der X- und Y-Achse:**

- `set log x`
  Die Werte der X-Achse werden in den folgenden Plots logarithmisch aufgetragen. Entsprechend mit `set log y` für die Y-Achse.

- `set xrange [0:1]`
  Die X-Achse erstreckt sich in folgenden Darstellungen von 0 bis 1. Entsprechend für die Y-Achse.

- `set xtics 1,100,1e12`
  Die Skalierung der X-Achse beginnt nach diesem Befehl immer mit 1, der erste Schritt geht bis 100, dannach geht es äquidistant weiter bis $10^{12}$. Für die Y-Achse lautet der Befehl: `set ytics`

**Speichern der Grafik als druckbare Datei:**

- `set terminal postscript`
  Die folgenden Ausgaben werden im Postscript-Format erstellt.

- `set output "Dateiname.ps"`
  Die erzeugten Plots werden ab jetzt als Datei unter dem Namen "Dateiname.ps" gespeichert. Gespeichert wird die zuletzt dargestellte Kurve wieder mit dem `replot`-Befehl

- `set terminal postscript eps`
  Dies ist die Formateinstellung zur Erzeugung von eps-Dateien.

- `set terminal x11`
  Das Terminal wird auf Bildschirmausgabe gesetzt.

**Erstellen einer Fitfunktion:**

- `f(x)=a-b/(1+c*log(1+x/d))`
  In gnuplot können auch Funktionen definiert, berechnet und geplottet werden. Im Bespiel wird die Funktion `f(x)` definiert.

- `fit f(x) "Dateiname" via a,b,c,d`
  Wenn bestimmte Parameter, hier a,b,c und d, einer die Daten in "Dateiname" beschreibenden Funktion `f(x)` unklar sind, so berechnet gnuplot mit diesem Befehl die Parameter, mit denen die maximale Übereinstimmung zwischen `f(x)` und den unter "Dateiname" gespeicherten Daten erreicht wird. Die Ergebnisse der einzelnen Parameter werden ausgegeben. Dannach kann `f(x)` in den Plot geschrieben werden, wobei die Werte der Variablen automatisch durch die berechneten Werte ersetzt werden.

**Erzeugen von Multiplots:**

- `set multiplot`
  Am Anfang der Eingabezeile wird daraufhin das Prompt `gnuplot>` durch `multiplot>` ersetzt. Es kann ein Multiplot erstellt werden, das heißt es werden aufeinander folgende Plots in dem selben Bild dargestellt.

- `set size 0.4,0.4`
  Die Abmessungen eines Fensters, in dem ein zusätzlicher Plot mit eigenem Koordinatensystem erzeugt werden soll, werden festgelegt. Das Fenster wird später in das zuvor, in der Multiplotumgebung geplottete Koordinatensystem eingefügt.

- `set origin 0.55,0.15`
  Die linke untere Ecke des Fensters im bestehenden Bild wird angegeben.

- `set nolabel`
  Möglicherweise für den ersten Plot festgelegte Beschriftungen werden abgeschaltet. Daraufhin können die Eigenschaften im Fenster neu definiert werden. Eingefügt wird die Kurve mit dem Befehl Befehl: `plot "Dateiname"`

- `set nomultiplot`
  Diese Anweisung wechselt aus der Multiplot- in die normale Gnuplot-Umgebung.

## 5.3 Weitere Software

### 5.3.1 `xmgr`

Als Alternative zu `gnuplot` sei hier `xmgr` erwähnt. Dieses Programm bietet ähnliche Funktionen wie `gnuplot`, der Unterschied besteht in der Bedienung. Während `gnuplot` über eine Kommandozeile gesteuert wird, hat `xmgr` eine grafische Benutzeroberfläche mit Menü, Werkzeugleiste und natürlich Mausbenutzung. Allerdings ist es nicht immer einfach, die gerade benötigte Funktion in den verschachtelten Menüs zu finden! Dafür kann man mit `xmgr` relativ einfach Histogramme zeichnen. Das Programm kann man von der folgenden FTP-Adresse beziehen:
*ftp://plasma-gate.weizmann.ac.il/pub/xmgr4*

### 5.3.2 `xfig`

`xfig` („Facility for Interactive Generation of figures under X11“) stellt ein leistungsfähiges Vektorgrafik-Programm mit einer mehr oder weniger intuitiven Benutzeroberfläche im Stil bekannter Windows-Grafikprogramme dar. Damit eignet

es sich gut für Skizzen und Illustrationen. Besonders nützlich ist die Exportfunktion ins `eps`-Format (Encapsulated Postscript) und die verfügbaren Clipart-Bibliotheken zu Themenbereichen wie Informatik, Elektrotechnik oder Mechanik. Ein erstes Bild von `xfig` kann man sich in Abbildung 5.4 machen. Das Programm ist frei erhältlich unter *http://www.xfig.org*.

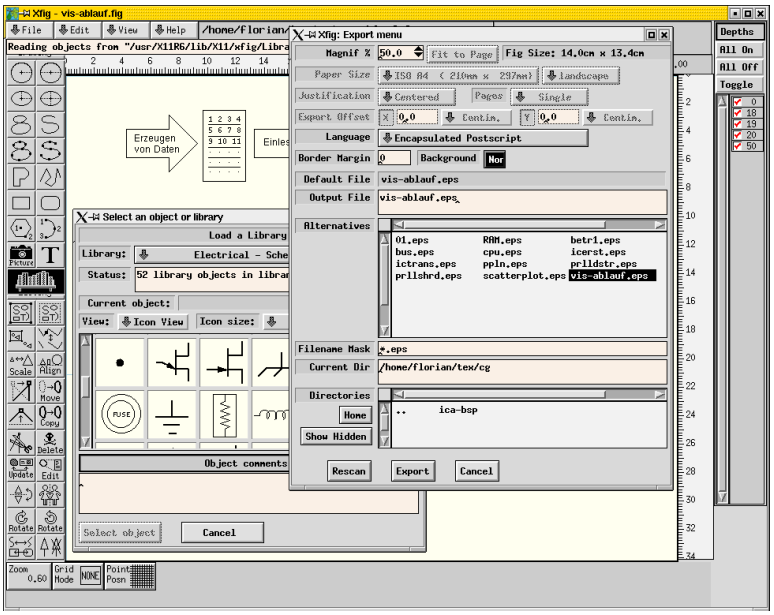



Abbildung 5.4: Die Benutzeroberfläche von `xfig`

### 5.3.3 ImageMagick

ImageMagick ist ein Programmpaket zur Arbeit mit Grafikdateien. Es stellt u. a. die beiden folgenden Kommandos bereit:

`display`

`display` ist in der Lage, Dateien in (fast) jedem beliebigen Grafikformat zu lesen und auf dem Bildschirm anzuzeigen. Das Öffnen der Datei `bild.png` erfolgt durch den Aufruf `display bild.png`. Es werden viele Bitmap- und einige Vektorgrafikformate, z. B. auch TrueType-Fonts unterstützt. Ist die Datei einmal geöffnet, kann sie mittels (einfacher) Bildbearbeitungsfunktionen verändert und ggf. in einem anderen Format gespeichert werden.

`convert`

`convert` wandelt Dateien in ein anderes Format um. Die Konvertierung einer Datei `bild.png` im `png`-Format in eine JPEG-kodierte Datei `web.jpg` ist mittels

des folgenden Befehles möglich: `convert bild.png web.jpg`.

Unter *http://www.imagemagick.org* findet man weitere Informationen sowie ImageMagick selbst.

### 5.3.4 `gimp`

Bei `gimp` („GNU Image Manipulation Program“) handelt es sich ähnlich wie bei `xfig` um ein Bildbearbeitungsprogramm mit grafischer Oberfläche. Obwohl es Ex- und Import einiger Vektorgrafikformate unterstützt, ist `gimp` ein reines Bitmapgrafikprogramm. Trotz der für Einsteiger zunächst gewöhnungsbedürftigen Bedienung hat sich `gimp` mit seinen vielen oft sehr komplexen Bearbeitungsfunktionen zu einem in vielen Fällen gleichwertigen Ersatz für Programme wie Adobe PhotoShop oder Corel PhotoPaint unter Windows entwickelt.

## 5.4 Grafikprogrammierung

Möchte man eigene Programme schreiben, die Computergrafik einsetzen, scheint es der naheliegendste Weg zu sein, die Grafik-Hardware direkt anzusteuern, um das gewünschte Ergebnis auf dem Monitor darzustellen. Was unter DOS auf Kosten der Systemsicherheit problemlos möglich ist, kann in einer Multi-User-Umgebung wie UNIX jedoch nicht erlaubt sein. Hier ist allein der Superuser berechtigt, direkt auf die Hardware zuzugreifen. Für gewöhnliche Benutzer gibt es dagegen spezielle Bibliotheken, die sichere Schnittstellen zur Grafikhardware darstellen.

Dieses Konzept bringt außer erhöhter Sicherheit auch direkte Vorteile für den Programmierer mit sich: Da die meisten Programme mit Grafikausgabe ähnliche Funktionen benötigen, wäre es reine Zeitverschwendung, wenn jeder Programmierer seine eigene Version grundsätzlich gleicher Ausgabefunktionen schreiben müsste. Stattdessen ist es effizienter, eine Grafikbibliothek zu benutzen. Dies gilt insbesondere unter Betrachtung der Portabilität. Da es unzählige verschiedene Grafikadapter mit selten übereinstimmender Programmierschnittstelle gibt, ist es vernünftig, die Teile eines Programmes, die sich mit direkter Hardwareansteuerung beschäftigen und die folglich auf beinahe jedem System anders aussehen müssen, auszukoppeln und sich stattdessen auf die Ansteuerung einer überall definierten gemeinsamen Schnittstelle zu beschränken.

Eine solche Schnittstelle gibt es auf UNIX-Systemen in Form des *X Window Systems*. Unter Ansteuerung der zu Grunde liegenden Bibliothek *xlib* ist es jedem Benutzer erlaubt, beliebige Grafikausgaben zu erzeugen. Der Nachteil dieser Methode ist der immer noch immense Programmieraufwand. Dies hat zwei Gründe: Zum einen sind die Funktionen der xlib sehr allgemein gehalten und beinhalten anders, als der Leser vielleicht erwartet, keinerlei Funktionen, um wie auch immer geartete Steuerelemente (sog. *Widgets*, Kurzform f. „Window Gadget“, z. B. Text-

eingabefelder, Menüleisten, Fensterrahmen, usw.) zu zeichnen. Die Erzeugung der grafischen Benutzeroberfläche aus einfachsten Zeichenbefehlen obliegt also allein dem Programmierer. Zum anderen darf der ungeheure Verwaltungsaufwand, den ein Programm mit grafischer Oberfläche gegenüber einem reinen Textprogramm zusätzlich erfordert, nicht vernachlässigt werden. Dieser entsteht dadurch, dass die grafische Oberfläche ständig und unvorhersehbar sog. „Ereignissen“ ausgesetzt ist, also Dingen wie Tastendrücken, Mausklicks, dem Verschieben von Fenstern, usw.. In einer *EventList* werden diese Ereignisse von der xlib zwar aufgezeichnet, um die jeweilige Reaktion des Programms auf jedes nur erdenkliche Ereignis muss sich der Programmierer aber selbst kümmern. Er muss also z. B. dafür sorgen, dass Tastendrücke an die richtige Stelle weitergeleitet werden, dass Mausklicks auf Menüs zu den gewünschten Aktionen führen und darüberhinaus möglicherweise das Aussehen des gerade angeklickten Menüs verändern (z. B. durch 3D-Effekte wie das „Eindrücken“ von Knöpfen) und dass zuvor verdeckte Teile des Fensters neu gezeichnet werden, wenn sie durch das Verschieben anderer Programmfenster sichtbar geworden sind.

Um die Grafikprogrammierung um lästige und für das eigentliche Programm unwesentliche Verwaltungsaufgaben und das eigene Zeichnen von Steuerelementen zu erleichtern, wurden verschiedene auf xlib aufbauende Bibliotheken entwickelt, die sowohl für Programmierer mit geringem Aufwand einsetzbare als auch für die späteren Anwender einfach und einheitlich zu bedienende grafische Oberflächen bereitstellen. Diese Bibliotheken sind Schnittstellen zwischen dem Programmierer und der xlib. Sie stellen eine Vielzahl bereits vordefinierter Steuerelemente bereit, die der Programmierer einfach in seinen Programmen einsetzen und ggf. anpassen kann. Außerdem nehmen sie ihm den durch Ereignisse auftretenden Verwaltungsaufwand größtenteils ab, so dass er sich auf die eigentliche Aufgabe seines Programms konzentrieren kann.

Die wohl meist eingesetzten Bibliotheken dieser Art sind Trolltechs *Qt*, auf der die v. a. bei Linux-Anwendern beliebte Oberfläche KDE aufbaut, *gtk*, das die Grundlage für den nicht minder beliebten *Gnome*-Desktop darstellt, und *Tk*. Wenn nicht wirklich schwerwiegende Gründe dagegen sprechen, sollte man Grafikprogrammierung nicht unterhalb der Ebene derartiger Bibliotheken betreiben.

## 5.5 3D-Grafik

Wenn von 3D-Grafik am Computer die Rede ist, so meint man damit meist die Projektion einer virtuellen Welt aus 3D-Objekten auf eine zweidimensionale Ausgabefläche, z. B. einen Bildschirm. Denn ein wirklich räumlicher Bildeindruck ist mit der heute üblichen Hardware nicht realisierbar.

Zur Erzeugung dieser Projektionsgrafik wendet der Computer das in Abbildung 5.5 gezeigte Prinzip des Raytracing an. Dabei hat man sich den Bildschirm als „Fenster“ in die darzustellende Welt vorzustellen. Jedem einzelnen Bildpunkt

(*Pixel*) wird nun ein Lichtstrahl zugeordnet, der aus der virtuellen Welt kommend den jeweiligen Bildpunkt trifft und für seine spätere Farbe verantwortlich ist. Um diese Farbe zu bestimmen, verfolgt der Computer den Lichtstrahl so lange zurück, bis er auf eine Reflexion an einem der darzustellenden Objekte trifft. Unter Berücksichtigung der Form und der angenommenen Oberflächenbeschaffenheit kann nun der Strahl weiter zurückverfolgt werden, wobei er entweder im Nichts verläuft, auf ein weiteres Objekt oder aber eine Lichtquelle trifft.

Strahlen, die ins Nichts laufen, können auch kein Licht führen, bedingen also eine schwarze Einfärbung des Bildpunkts. Lässt sich der Strahl dagegen auf eine Lichtquelle zurückführen, so kann man aus der Lichtfarbe und der Oberflächenfarbe des reflektierenden Objekts die Farbe des Bildpunkts ermitteln. Bei Strahlen, die mehrfach auf Objekte treffen, ist es eine Frage der gewünschten Genauigkeit der Ergebnisse und natürlich des beabsichtigten Rechenaufwands, wie viele Reflexionen man verfolgt bzw. ab welcher Zahl von Reflexionen man den Strahl vernachlässigt.

In Abhängigkeit davon, wie exakt die zu Grunde liegenden optischen Gesetzmäßigkeiten und die Oberflächenbeschaffenheiten der darzustellenden Objekte berücksichtigt werden, kann Raytracing für komplexe Grafiken leicht zu Bearbeitungszeiten im Stundenbereich führen.

Ein unter UNIX sehr beliebter Raytracer ist `povray`. Die Definition der zu rendernden Objekte erfolgt über Textdateien in einem an die Programmiersprache C erinnernden Format. Außerdem ist eine komfortable grafische Oberfläche unter KDE in Entwicklung, der `kpovmodeler`. Beide Programme sind im Internet frei verfügbar.

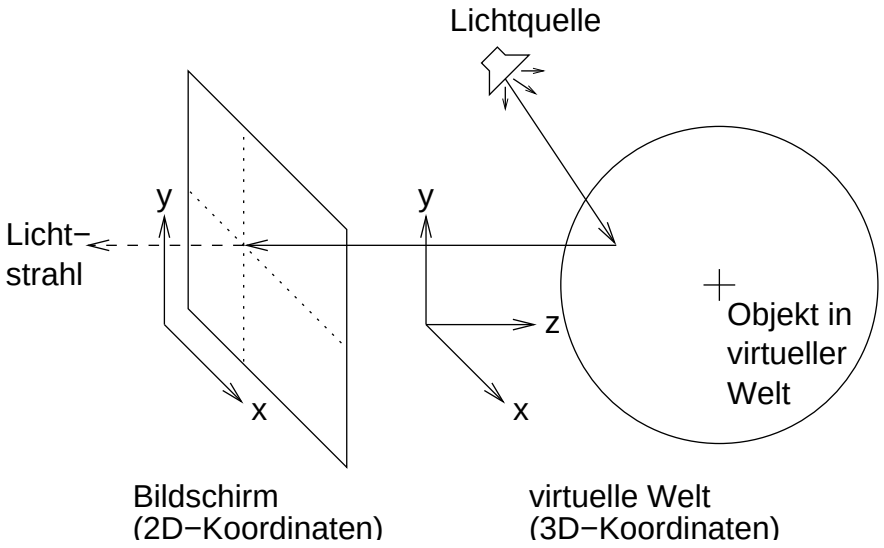



Abbildung 5.5: Grundprinzip des Raytracing

Für Animationen benötigt man häufig 3D-Grafik in Echtzeit, d. h. im Ideal-

fall mindestens 25 Bilder pro Sekunde. Um 3D-Grafik schnell zu erzeugen, stehen verschiedene Bibliotheken als Schnittstellen zur 3D-Hardware oder als deren Ersatz (Emulation) zur Verfügung. Eine weit verbreitete Schnittstelle ist *OpenGL*. Sie hat allerdings die Einschränkung, dass alle Objekte aus Dreiecken zusammengesetzt bzw. angenähert werden müssen. In Fällen, in denen das zu aufwändig ist, kann die auf OpenGL aufbauenede *glu*-Bibliothek verwendet werden, die Vereinfachungen z. B. in Form automatisch aus Dreiecken erzeugter Kugeln bietet. Praktisch ist auch das *OpenGL-Widget*, das von Qt bereitgestellt wird: Mit seiner Hilfe kann man 3D-Animationen in die Oberfläche von mit Qt erstellten Programmen integrieren.

# Kapitel 6

# Programmierung in C

Es gibt verschiedene Programmiersprachen, in denen Anweisungen an einen Computer geschrieben werden können. Diese Quelltexte von Programmen werden dann von einem Compiler, also einem für die Sprache spezifischen Übersetzungsprogramm, in die vom Rechner verständliche Maschinensprache übersetzt.
Da sich Programmiersprachen genau wie gesprochene Sprachen ständig verändern gibt es verschiedene Versionen der meisten Sprachen. Eine Weiterentwicklung von `C`, die selbst wieder eine eigene Sprache darstellt ist `C++`. Die älteste Sprache heißt "Fortran", das Besondere an ihr ist, dass eine Kommission festlegt, welche Änderungen durchgeführt werden. Sie entwickelt sich also stetig fort und bewahrt dennoch ihre einheitliche Form. Daneben gibt es noch viele weitere Sprachen wie Pascal und Basic.
Die meisten Programmiersprachen ähneln sich in ihrem Aufbau, im folgenden wird aber nur noch die Sprache `C` behandelt, in der beispielsweise auch das Betriebssystem UNIX geschrieben wurde.

## 6.1 Compilieren eines C-Programmes

Das Computerprogramm wird mit der Sprache `C` in einem Editor geschrieben und als Datei mit der Endung `.c` abgespeichert. Im Xterm des Betriebssystems wird daraufhin der Befehl `cc Programm.c` aufgerufen. Der installierte C-Compiler liest jetzt den Quelltext des Programmes, lädt dabei die angeforderten Bibliotheken in einen Zwischenspeicher und verlinkt sie mit den entsprechenden Befehlen. Diesen ersten Schritt führt ein sogenannter Präcompiler durch. Im zweiten Schritt liest der Compiler das Programm erneut und übersetzt es in Binärcodes. Dabei fügt er an den richtigen Stellen die benötigten Unterprogramme aus den angeforderten Bibliotheken ein. Die Übersetztung, also das ausführbare Programm wird als Datei mit dem Namen `a.out` gespeichert und kann mit dem Kommando `./a.out` im Xterm aufgerufen werden. Um dem Programm beim Compilieren einen anderen Namen zuzuordnen steht der Befehl `cc Programm.c -o Name`

zur Verfügung.
Falls der Quelltext Programmierfehler enthält, die der Compiler nicht versteht, so wird der Vorgang gestoppt und die Zeile, in welcher der Fehler aufgetreten ist, mit einer entsprechenden Fehlermeldung angegeben.
Zur Übersetzung von C-Programmen stehen mehrere verschiedene Compiler zur Auswahl, die auf bestimmten Computern und für spezielle Anforderungen einen möglichst schnellen Programmablauf gewährleisten sollen. Meist fallen diese Unterschiede nicht ins Gewicht, trotzdem kann es passieren, dass Programme mit manchen Compilern auf einigen Rechnern nicht einwandfrei funktionieren.

## 6.2 Allgemeiner Programmaufbau

Ein C-Programm besteht aus einem Programmkopf, sowie dem eigentlichen Programm. Alle Anweisungen im Programmkopf beginnen mit `#` und enden mit einem einfachen Zeilenumbruch, ohne Strichpunkt. Hier werden Bibliotheken aufgerufen und Konstanten definiert. Nach dem Ausdruck `main()` steht der Befehlsteil des Programmes in geschweiften Klammern. Diese Zeile beginnt wie alle folgenden bereits ohne `#`. Innerhalb der geschweiften Klammern endet jede Zeile mit einem Strichpunkt. Zu Beginn werden globale Variable definiert, dann folgen die eigentlichen Anweisungen. Abgeschlossen wird das Programm mit dem Schließen der geschweiften Klammer. /* Kommentare, die zum Verständnis des Programmes im Quelltext gegeben werden, vom Compiler aber nicht beachtet werden sollen, sind wie dieser Satz einzurahmen.*/

## 6.3 Der Aufruf von Bibliotheken und Unterprogrammen

Neben den Standardbefehlen in `C` gibt es einige Befehle, die vom Compiler nur übersetzt werden können, wenn ihnen bestimmte Programme vom Präcompiler vorangestellt werden. Diese Unterprogramme sind in sogenannten Systembibliotheken gespeichert und stehen dem Präcompiler zur Verfügung. Damit dieser weiß welche Bibliotheken in dem Programm benötigt werden, müssen diese am Beginn des Quelltextes mit dem Befehl `#include <stdio.h>` aufgerufen werden. Das `#` am Anfang einer Zeile kennzeichnet eine Präcompileranweisung, das `include` ist der Befehl zum Einbinden einer Bibliothek und zwischen den Klammern steht der Name der angeforderten Bibliothek. Im Quelltext sind Befehle, welche auf Programme aus den Bibliotheken angewiesen sind, nicht speziell gekennzeichnet, der Präcompiler erkennt sie von selbst. Einige Bibliotheken sind in Tabelle 6.1 aufgelistet und Tabelle 6.2 weist auf einige Inhalte hin. Üblicherweise enthalten sie etwa je 15 Programme.

| | | | |
|---|---|---|---|
| <assert.h> | <inttypes.h> | <signal.h> | <stdlib.h> |
| <complex.h> | <iso646.h> | <stdarg.h> | <string.h> |
| <ctype.h> | <limits.h> | <stdbool.h> | <tgmath.h> |
| <errno.h> | <locale.h> | <stddef.h> | <time.h> |
| <fenv.h> | <math.h> | <stdint.h> | <wchar.h> |
| <float.h> | <setjmp.h> | <stdio.h> | <wctype.h> |

Tabelle 6.1: Liste einiger in C verfügbarer Bibliotheken

| Name der Bibliothek | Inhalte der Bibliotheken |
|---|---|
| `stdio.h` | Standard Ein- und Ausgabe. |
| `math.h` | Mathematische Operatoren wie sin und cos |
| `float.h` | Schnellere Bearbeitung von Floatingpoint Operationen |
| `time.h` | Ermöglicht Zeitmessungen |
| `complex.h` | Rechnen mit komplexen Zahlen |

Tabelle 6.2: Inhalte der wichtigsten Bibliotheken

Es ist auch möglich selbst Unterprogramme zu schreiben, die später mit einem einzigen Befehl aufgerufen werden können. Der einfachste Fall ist die Festlegung einer Konstanten, die im Programmverlauf wiederholt eingelesen werden soll. Der Befehl `#define pi 3.141592` ist aufgrund des `#` wieder eine Präcompileranweisung und weist dem Ausdruck `pi` für das folgende Programm den Wert `3,141592` zu. Der Präcompiler ersetzt also vor dem Compilen alle `pi` im Quelltext durch den Zahlenwert.
Ein Unterprogramm, welches zwei beim Aufruf gegebene Zahlen quadriert und addiert würde folgendermaßen definiert:

```
#define Quadsum(a,b) (a*a + b*b)
```

Der Präcompiler würde in diesem Fall im Quelltext den Aufruf des Unterprogrammes mit dem Befehl `Quadsum(3,5)` durch den Ausdruck `(3*3 + 5*5)` ersetzten, was bei häufigem Aufruf dieses Unterprogrammes eine erhebliche Programmierhilfe darstellt. Wenn das Unterprogramm aus mehreren einzelnen Befehlen besteht, so werden diese in geschweiften Klammern zusammengefasst.

## 6.4 Variable in C

Nach der Anweisung `main(){` beginnt der eigentliche Programmteil. Doch vor den ersten Befehlen werden üblicherweise noch alle globalen Variablen deklariert.

Die Anweisung `int Variable;` beispielsweise definiert eine Variable vom Typ `int`, welche den Namen `Var1` trägt. Der Befehl `Var1 = 23;` weist schließlich der Integervariablen den Wert der ganzen Zahl 23 zu. Verschiedene in `C` verfügbare Arten von Variablen sind in Tabelle 6.3 dargestellt.

| Variablentyp: | Speicherart: | Speicherumfang: |
|---|---|---|
| `char` | beliebige Zeichenkette | 8 Bit |
| `int` | ganze Zahlen | |
| `float` | Gleitkommazahlen | 32 Bit |
| `double` | lange Gleitkommazahlen | 64 Bit |
| `long double` | sehr lange Gleitkommazahlen | 256 Bit |

Tabelle 6.3: Variablentypen und ihre Bedeutung

Mit dem Befehl `enum boolean {true,false}` wird ein eigener Variablentyp mit dem Namen `boolean` definiert, der nur die Werte `true` (=0) und `false` (=1) annehmen kann.

## 6.5 Daten Ein- und Ausgabe

Der Befehl `printf("Hallo Welt");` gibt am Bildschirm den Text `Hallo Welt` aus. Tabelle 6.4 nennt einige wichtige Befehle für die Textausgabe mit `C`. Die Ausgabe eines Variablenwertes wird durch ein Prozentzeichen angekündigt, es folgt eine Angabe, wie viele Stellen vor und nach dem Komma berücksichtigt werden sollen und das in Tabelle 6.5 gegebene Ausgabeformat. Der Name der Variablen wird erst nach Schließen der Anführungszeichen durch ein Komma getrennt angegeben. Sind den Variablen m, x, y und s die Werte 13; 17,4; 3,2; und 3 zugewiesen, so bewirkt der Befehl
`printf("M=%6dnX=%3.5f; Y=%4.7 e %s",m,x,y,s);`
die Bildschirmausgabe des folgenden Textes, wobei die Unterstriche für den Raum stehen, den die Zahlen einnehmen, obwohl sie selbst nicht so lang sind und an dessen Stellen Leerzeichen gedruckt werden.

```
M=____13
X=_17.4____; Y=___3.2______ e 3
```

Das Einlesen von Daten über die Tastatur erfolgt mit dem Befehl
`scanf(Einleseformat, Variable);`
wobei sich die Formatangabe wieder aus einem der in Tabelle 6.5 gegebenen Buchstaben und dem vorangestellten Prozentzeichen zusammensetzt. Auch stehen beide in Anführungszeichen und sind durch ein Komma von der folgenden Variablen getrennt. Der Variablen geht beim Einlesen von Daten allerdings ein

| Zeichenfolge in Formatangabe | Wirkung |
|---|---|
| `\n` | In der Ausgabe wird eine neue Zeile begonnen (*new line*). |
| `\t` | Es wird ein Tabulatorsprung bewirkt (Tabulator ist voreingestellt). |
| `\b` | Die Ausgabeposition wird um 1 Zeichen zurückgesetzt (*backspace*). |
| `\"`<br>`\\`<br>`%%` | Es wird { das Anführungszeichen (`"`), der invertierte Schrägstrich (`\`), das Prozentzeichen (`%`) } in den Ausgabetext übernommen. |

Tabelle 6.4: Befehle zur Textpositionierung und Zeichenausgabe, sie sind Teil der Formatangabe und stehen zwischen den Anführungszeichen.

`&`-Zeichen voran. Der Befehl `scanf("%d",&Name);` läßt das Programm also an der betreffenden Stelle je nach Deklaration der Variablen `Name` beispielsweise auf die Eingabe einer ganzen Zahl über die Tastatur warten.

## 6.6 Mathematische Operationen und ihre Abkürzungen

Die Grundrechenarten Addieren, Subtrahieren, Multiplizieren, Teilen sowie das Klammernsetzten werden vom `C`-Compiler verstanden. Kompliziertere Operationen wie Wurzelziehen und das Berechnen von Sinuswerten können nach Aufruf der Bibliothek `math.h` ebenfalls mit einfachen Befehlen durchgeführt werden. Tabelle 6.6 nennt die in `math.h` definierten Operationen und deren Funktionsweise. Tabelle 6.7 zeigt einige der in `C`-üblichen Abkürzungen einfacher Rechenoperationen. Zum besseren Verständnis dieser Kurzschreibweisen sind in Tabelle 6.8 Beispiele aufgeführt.

## 6.7 Vergleichende Operationen

In `C` stehen verschiedene Operationen zum Vergleich von Variablen zur Verfügung. In Tabelle 6.9 sind ihre mathematischen Entsprechungen sowie die möglichen Ergebnisse aufgelistet.

| Format-Code | Ausgabe |
|---|---|
| `s` | String-Variable oder -Konstante |
| `c` | einzelnes Zeichen vom Typ `char`. |
| `d`, `ld`, `u` | Ausgabe einer ganzen Zahl: Typ `char`, `short` oder `int` / `long` / `unsigned` |
| `o` | Ausgabe in oktaler Form. |
| `x` | Ausgabe in hexadezimaler Form. |
| `e`, `f`, `g` | Ausgabe einer `float`- oder `double`-Variablen: in Gleitkommadarstellung / in Festkommadarstellung / in der kürzeren Ausgabe nach `e` od. `f`. |

Tabelle 6.5: Zeichen zur Festlegung des Ausgabeformates einer Variablen nach dem Muster
`printf("% Zahl Zeichen",Variablenname);`

## 6.8 Logische Operationen

Zur Verknüpfung vergleichender Operationen untereinander werden die in Tabelle 6.10 erklärten logischen Operationen angewandt. Zu deren besseren Verständnis sind in Tabelle 6.11 Beispiele genannt.

Die bitweise Bearbeitung von Speicherplätzen hat zwar keine praktische Bedeutung, ermöglicht aber eine wesentlich bessere Rechnerauslastung. Die bitweise Negation einer Zahl `k=ñ;` und das Verschieben von Bits werden in Tabelle 6.12 verständlich. Der Befehl `n=1«3` setzt den Wert von `n` auf 1 und verschiebt dann den Bit um drei Stellen nach links, so dass `n` den Wert 8 annimmt.

## 6.9 Kontrollstrukturen in C

Diese Strukuren geben an ob, wann und wie lange bestimmte Programmteile ablaufen sollen. Meist hängen diese Festlegungen vom Inhalt spezieller Variablen ab.

- IF-Anweisung: Falls eine genannte Bedingung (hier: b ist genau gleich 1) erfüllt wird, soll ein erstes Unterprogramm durchlaufen werden, andernfalls ein zweites. Beispiel:
  `if(b==1){...1.Programm;}`
  `else{...2.Programm;}`

- WHILE-Schleife: Ein Programmabschnitt wird solange wiederholt, bis eine gegebene Bedingung nicht mehr zutrifft. Eine mögliche Anwendung hierfür ist eine mit Ja oder Nein zu beantwortende Frage:
```
char c;
c=0;
while(c!='y' && c!='n'){
printf("Bitte antworten Sie mit 'y' oder 'n':");
scanf("%d",&c);
```
} Die Abfrage wird solange wiederholt, bis eine der Bedingungen `'c` ungleich `y'` oder `'c` ungleich `n'` nicht mehr erfüllt ist. Um sicher zu stellen, dass die Schleife mindestens einmal durchlaufen wird, ist es auch möglich eine DO-WHILE-Schleife zu benutzen:
`do {..`Unterprogramm`..} while(Bedingung);`

- FOR-Schleife: Sie eignet sich wie die WHILE-Schleife zur Wiederholung eines Programmabschnittes. Das Besondere ist, dass hier mit einer lokalen Variablen gearbeitet werden kann:
`for(int i=0; i<20; i++) {..`Unterprogramm`..}`
Die Art der FOR-Schleife wird durch die drei Teile innerhalb der runden Klammern beschrieben, die Definition einer Variablen, die Formulierung der Bedingung sowie einem Rechenschritt, der bei jedem Durchlauf der Schleife einmal ausgeführt wird.

- BREAK: Der Aufruf des Break-Befehls bewirkt, dass die Schleife, welche das Programm gerade durchläuft, verlassen wird. Das Programm überspringt alle Befehle und Überprüfungen in der Schleife und fährt mit der Bearbeitung der folgenden Anweisungen fort. Um dies an eine Bedingung zu knüpfen wird meist eine IF-Anweisung voran gestellt.
```
double a[20];
double sum_sqrt=0;
for(int i=0; i<20; i++)
{ if(a[i]<0) break;
sum_sqrt+=sqrt(a[i]);}
```

- GOTO-Anweisung: Das Kommando erlaubt einen Sprung im Programm an eine beliebige andere Stelle.
```
if(a>0) goto Sprungziel;
...
Sprungziel;
```
Im Beispiel werden also alle Programmschritte zwischen der IF-Anweisung und dem Wort "`Sprungziel`" nicht ausgeführt, sofern der Variablen `a` ein Wert größer 0 zugeordnet ist. Trotz der unkomplizierten Anwendung dieses Befehls ist darauf zu achten, dass er bei wiederholter Verwendung schnell zu einer unübersichtlichten Programmgestaltung führen kann.

## 6.10 Vektoren in C

Vektoren werden wie einfache Variablen mit einem Variablentyp und dem Namen des Vektors definiert. Allerdings folgt dem Namen in eckigen Klammern eine ganze Zahl, welche die Größe des Vektors angibt.
`double k[3]={1.5,2.0};`
Im Beispiel werden dem dreidimensionalen Vektor nur zwei Werte zugewiesen, in diesem Fall wird die dritte Komponente automatisch gleich Null gesetzt. Im Programmverlauf können nicht allen Werten des Vektors `k` in einem Schritt neue Zahlen zugeordnet werden. Der Befehl `k={5.2,3.0,1.5}` wäre also nicht zulässig. Es ist allerdings möglich elementweise neue Werte zuzuordnen, etwa die erste Zahl des oben genannten Vektors auf 5,2 heraufzusetzten mit dem Befehl:
`k[0]=5.2;`

Zeichenketten werden in C als Felder vom Typ `char` gespeichert, wobei ein Feld mit 20 Elementen nur maximal 19 Zeichen enthalten darf, da die Kette intern immer von einem zusätzlich angehängten Nullzeichen abgeschlossen wird.
Beispielsweise `char S[5]="Name";`

Matrizen oder mehrdimensionale Arrays werden analog zu den Vektoren, allerdings mit zwei konstanten Angaben zur Größe definiert. Zuerst wird dabei die Anzahl der Zeilen und dann die der Spalten angegeben. Wie bei den Vektoren ist es auch hier nur möglich einzelnen Elementen neue Werte zuzuweisen. Matrizen werden zeilenweise befüllt, wie folgendes Beispiel zeigt. Die beschriebene Matrize `a` ist in Tabelle 6.13 dargestellt:
`double a[3][4]={{1,3,5,7.56},{6},{9,4.002,0,2}};`

## 6.11 Zeiger (Pointer)

Bei der Variablendeklaration wird jeder Variablen ein ihrem Format entsprechender Speicherplatz zugeordnet. Zeiger dienen dazu, dass Daten nicht mehrfach kopiert und auf anderen Speicherplätzen abgelegt werden müssen, da das Umordnen von Zeigern den selben Zweck, nur schneller, erfüllt. Es ist beispielsweise möglich eine Namensliste alphabetisch zu ordnen, ohne die Adressen aufwändig kopieren zu müssen, indem eine geordnete Liste von Zeigern erstellt wird, welche lediglich die Speicheradressen der entsprechenden Namen enthalten. Ein auf den Speicherplatz der Variablen `i` vom Typ `int` zeigender Pointer mit Namen `p_i` läßt sich mit dem Befehl `int *p_i=&i;` erzeugen (vgl. Abb. 6.1). Der Befehl sucht den Speicherplatz der Variablen `i` und weist `p_i` deren Adresse zu, da dieser Variablen der Adress-Operator `&` vorangestellt ist. Um auf den gespeicherten Wert zurückgreifen zu können steht der Zugriffs-Operator `*` zur Verfügung. Folglich setzt der Befehl `*p_i=23;` den Wert der Variablen `i` auf 23 fest, der Zeiger selbst weist unverändert auf die Adresse von `i`.

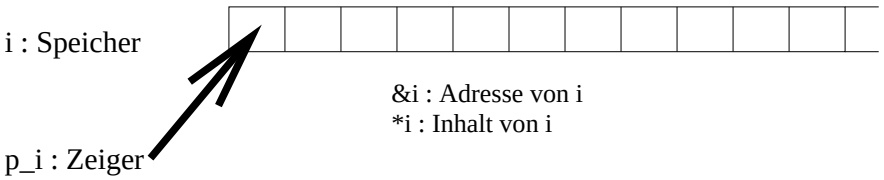



Abbildung 6.1: Schematische Darstellung zur Funktionsweise von Zeigern

| Aufruf | Bedeutung / Hinweise |
|---|---|
| `abs(w)` | $\|w\|$, Typ von `w` beliebig, falls `abs` Makro, sonst `int` |
| `acos(x)`<br>`asin(x)`<br>`atan(x)` | } Umkehrfunktionen zu `cos`, `sin`, `tan`. |
| `ceil(x)` | Liefert kleinste ganze Zahl, die größer oder gleich `x` ist. |
| `cos(x)` | `x` im Bogenmaß. |
| `cosh(x)` | Berechnung von $\frac{e^x+e^{-x}}{2}$. |
| `exp(x)` | $e^x$ |
| `fabs(x)` | $\|x\|$ als Funktion. |
| `floor(x)` | Es wird $[x]$ berechnet, d. h., die größte ganze Zahl, die kleiner der gleich `x` ist. |
| `fmax(x1,x2)` | Maximum zweier Zahlen. |
| `fmin(x1,x2)` | Minimum zweier Zahlen. |
| `hypot(x1,x2)` | Berechnet $\sqrt{x_1^2+x_2^2}$. |
| `log(x)` | $\ln x$ (Umkehrfunktion zu $e^x$). |
| `log10(x)` | $\log x$ (Logarithmus zur Basis 10). |
| `max(n1,n2)` | Maximum zweier Werte } Parameter und Ergeb- |
| `min(n1,n2)` | Minimum zweier Werte } nis vom Typ `int`. |
| `pow(x1,x2)` | Berechnet $(x_1)^{x_2}$. |
| `rand()` | Ziehen einer Zufallszahl; Ergebnis vom Typ `int`, aus Intervall $[0, 32767]$. |
| `randl()` | Ziehen einer Zufallszahl; Ergebnis vom Typ `double`, aus Intervall $[0, 1)$. |
| `sin(x)` | `x` im Bogenmaß. |
| `sinh(x)` | Berechnet $\frac{e^x-e^{-x}}{2}$. |
| `sqrt(x)` | $\sqrt{x}$ |
| `srand(p)` | Starten des Zufallszahlengenerators (für `rand` und `randl` mit Primzahl p; kein Typ für srand anzugeben). |
| `tan(x)` | `x` im Bogenmaß. |
| `tanh(x)` | Berechnet wird $\frac{e^x-e^{-x}}{e^x+e^{-x}}$. |

Tabelle 6.6: Kommandos zum Aufruf mathematischer Funktionen in der Bibliothek "math.h". Die Parameter `x`, `x1`, `x2` sind vom Typ `double`, die Parameter `n`, `n1`, `n2` vom Typ `int`. Die Funktionen besitzen den Typ `double`, soweit nicht anders angegeben.

| Rechenoperation: | Alternativ: | Abkürzung: |
|---|---|---|
| `n = n - 1;` | `n-1;` | `n-;` |
| `n = n + 1;` | `n+1;` oder `n+=1;` | `n++;` |
| `n = 2 * n;` | | `n*=2;` |
| `z = n; n = n + 1;` | | `z=n++;` |
| `n = n + 1; z=n;` | | `z=++n;` |

Tabelle 6.7: Abkürzende Schreibweisen einfacher Rechenoperationen in `C`, `n` und `z` seien Variablen vom Typ `int`.

| Programmzeilen: | Wert von z: | Wert von n: |
|---|---|---|
| `int n,z;  n=2;` | - | 2 |
| `z=n++` | 2 | 3 |
| `z=++n` | 4 | 4 |
| `z=n--` | 4 | 3 |
| `z=--n` | 2 | 2 |

Tabelle 6.8: Beispiele der mathematischen Abkürzungen in `C`

| mathem. Zeichen | Zeichen in C | Priorität | Anwendungsform | Ergebnis mit Typ `int` |
|---|---|---|---|---|
| $<$ | `<` | 6 | `z = a < b` | $z = \begin{cases} 1 & \text{falls} \quad a < b \\ 0 & \text{sonst} \end{cases}$ |
| $\leq$ | `<=` | | `z = a <= b` | $z = \begin{cases} 1 & \text{falls} \quad a \leq b \\ 0 & \text{sonst} \end{cases}$ |
| $\geq$ | `>=` | | `z = a >= b` | $z = \begin{cases} 1 & \text{falls} \quad a \geq b \\ 0 & \text{sonst} \end{cases}$ |
| $>$ | `>` | | `z = a > b` | $z = \begin{cases} 1 & \text{falls} \quad a > b \\ 0 & \text{sonst} \end{cases}$ |
| $=$ | `==` | 7 | `z = a == b` | $z = \begin{cases} 1 & \text{falls} \quad a = b \\ 0 & \text{sonst} \end{cases}$ |
| $\neq$ | `!=` | | `z = a != b` | $z = \begin{cases} 1 & \text{falls} \quad a \neq b \\ 0 & \text{sonst} \end{cases}$ |

Tabelle 6.9: Vergleichende mathematische Zeichen, ihre Entsprechungen in `C`, sowie deren Ergebnisausgabe.

| math. Zeichen | Zeichen in C | Prio-rität | Bedeutung | Anwen-dungsform | Ergebnis mit Typ `int` |
|---|---|---|---|---|---|
| $\neg$ | `!` | 2 | Verneinung | `z = !r` | $z = \begin{cases} 1 & \text{falls } r = 0 \\ 0 & \text{sonst} \end{cases}$ |
| $\wedge$ | `&` | 8 | UND | bitweise Verknüpfung der Operanden; nur für `char`, `short`, `int`, `long` und `unsigned` | |
| | `&&` | 11 | | `z = r && s` | $z = \begin{cases} 1 & \text{falls } r, s \neq 0 \\ 0 & \text{sonst} \end{cases}$ |
| $\vee$ | `^` | 9 | exklusives ODER | bitweise Verknüpfung der Operanden; nur für `char`, `short`, `int`, `long` und `unsigned` | |
| | `\|` | 10 | ODER | | |
| | `\|\|` | 12 | ODER | `z = r \|\| s` | $z = \begin{cases} 1 & \text{falls } r \neq 0 \text{ oder } s \neq 0 \\ 0 & \text{sonst} \end{cases}$ |

Tabelle 6.10: Logisch verknüpfende mathematische Zeichen, ihre Entsprechungen in `C`, sowie deren Ergebnisausgabe.

| `C`-Befehl: | Variablen-werte | Binärdarstellung ..64\|32\|16\|8\|4\|2\|1\| |
|---|---|---|
| `k=n&m;` | $n = 40$ | ..00101000 |
| | $m = 15$ | ..00001111 |
| | $\rightarrow k = 8$ | ..00001000 |

Bitweise Kombination:

| `&` | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| `k=n\|m;` | $n = 40$ | ..00101000 |
|---|---|---|
| | $m = 15$ | ..00001111 |
| | $\rightarrow k = 47$ | ..00101111 |

| `\|` | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| `k=n^m;` | $n = 40$ | ..00101000 |
|---|---|---|
| | $m = 15$ | ..00001111 |
| | $\rightarrow k = 39$ | ..00100111 |

| `^` | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Tabelle 6.11: Beispiele zur logischen Verknüpfung in `C`

| C-Befehl: | Variablen-werte | Binärdarstellung ..64\|32\|16\|8\|4\|2\|1\| |
|---|---|---|
| `k=~m;` | $n = 40$ | ..00101000 |
| | $\rightarrow k =?$ | ..11010111 |
| `n=1<<3;` | $n = 1$ | ..00000001 |
| | $\rightarrow n = 8$ | ..00001000 |
| `n\|=1<<5;` | $n = 8$ | ..00001000 |
| | $1 << 5 = 32$ | ..00100000 |
| | $\rightarrow n = 40$ | ..00101000 |
| `n<<=2;` | $n = 40$ | ..00101000 |
| | $\rightarrow n = 160$ | ..10100000 |
| `n&=~(1<<5);` | $1 = 1$ | ..00000001 |
| | $\rightarrow (1 << 5) = 32$ | ..00100000 |
| | $\rightarrow$ ~(32) =? | ..11011111 |
| | $n = 160$ | ..10100000 |
| | $\rightarrow k = 128$ | ..10000000 |

Bitweise Kombination:

| ~ | 0 | 1 |
|---|---|---|
| k | 1 | 0 |

Verschiebung: $0 \leftarrow 1$

| \| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Tabelle 6.12: Beispiele zur bitweisen Negation und Bitverschiebung

| Position | `a[..][1]` | `a[..][2]` | `a[..][3]` | `a[..][4]` |
|---|---|---|---|---|
| `a[1][..]` | 1 | 3 | 5 | 7.56 |
| `a[2][..]` | 6 | 0 | 0 | 0 |
| `a[3][..]` | 9 | 4.002 | 0 | 2 |

Tabelle 6.13: Darstellung der im Beispiel definierten Matrize `a`

# Literaturverzeichnis

[1] Manfred Precht, Nikolaus Meier, Joachim Kleinlein. *EDV-Grundwissen, Eine Einführung in Theorie und Praxis der modernen EDV* 4. Auflage, Addison-Wesley-Longman, Bonn, 1997