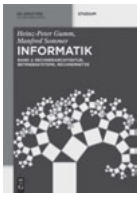
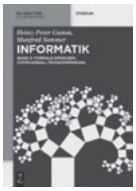
Heinz-Peter Gumm, Manfred Sommer
Informatik
Band 1: Programmierung, Algorithmen und Datenstrukturen
De Gruyter Studium

Weitere empfehlenswerte Titel

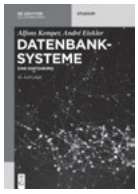


Informatik, Band 2: Rechnerarchitektur, Betriebssysteme, Rechnernetze
H.P. Gumm, M. Sommer, 2017
ISBN 978-3-11-044235-9, e-ISBN 978-3-11-044236-6,
e-ISBN (EPUB) 978-3-11-043442-2



Informatik, Band 3: Formale Sprachen, Compilerbau, Berechenbarkeit und Verifikation
H.P. Gumm, M. Sommer, 2018
ISBN 978-3-11-044238-0, e-ISBN 978-3-11-044239-7,
e-ISBN (EPUB) 978-3-11-043405-7

Rechnerorganisation und Rechnerentwurf, 5. Auflage
D. Patterson, J.L. Hennessy, 2016
ISBN 978-3-11-044605-0, e-ISBN 978-3-11-044606-7,
e-ISBN (EPUB) 978-3-11-044612-8



Datenbanksysteme, 10. Auflage
A. Kemper, 2015
ISBN 978-3-11-044375-2

IT-Sicherheit, 9. Auflage
C. Eckert, 2014
ISBN 978-3-486-77848-9, e-ISBN 978-3-486-85916-4,
e-ISBN (EPUB) 978-3-11-039910-3

Heinz-Peter Gumm, Manfred Sommer

Informatik

Band 1: Programmierung, Algorithmen und Datenstrukturen

DE GRUYTER
OLDENBOURG

Autoren
Prof. Dr. Heinz-Peter Gumm
Philipps-Universität Marburg
Fachbereich Mathematik
und Informatik
Hans-Meerwein-Straße
35032 Marburg
gumm@mathematik.uni-marburg.de

Prof. Dr. Manfred Sommer
Elsenhöhe 4B
35037 Marburg
manfred.sommer@gmail.com

ISBN 978-3-11-044227-4
e-ISBN (PDF) 978-3-11-044226-7
e-ISBN (EPUB) 978-3-11-044231-1

Library of Congress Cataloging-in-Publication Data
A CIP catalog record for this book has been applied for at the Library of Congress.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Druck und Bindung: CPI books GmbH, Leck
♾ Gedruckt auf säurefreiem Papier
Printed in Germany

www.degruyter.com

Teil I

Programmierung, Algorithmen und Datenstrukturen

Inhalt

I Programmierung, Algorithmen und Datenstrukturen

Vorwort —— ix

1 Grundlagen —— 1
1.1 Was ist „Informatik“? —— **1**
1.2 Information und Daten —— **4**
1.3 Informationsdarstellung —— **12**
1.4 Zahlendarstellungen —— **19**
1.5 Geschichtliche Entwicklung —— **39**
1.6 Aufbau von Computersystemen —— **48**
1.7 Speicher- und Anzeigemedien —— **58**
1.8 Von der Hardware zum Betriebssystem —— **63**

2 Grundlagen der Programmierung —— 75
2.1 Probleme und Algorithmen —— **75**
2.2 Programmiersprachen —— **83**
2.3 Daten und Operationen —— **90**
2.4 Typen, Variablen und Terme —— **100**
2.5 Anweisungen und Kontrollstrukturen —— **104**
2.6 Strukturiertes Programmieren —— **115**
2.7 Kollektionen und Iterationen —— **122**
2.8 Funktionales Programmieren in Python —— **135**
2.9 Objektorientiertes Programmieren —— **153**
2.10 Module und Bibliotheken —— **161**
2.11 Korrektheit —— **163**

3 Einige Python Projekte —— 173
3.1 Turtle-Grafik —— **173**
3.2 Aktienkurse aus Webseiten „kratzen“ —— **178**
3.3 Edbebenkarten —— **181**

3.4 Messen und Steuern — **185**
3.5 Zusammenfassung — **201**

4 Die Programmiersprache Java — 203
4.1 Die lexikalischen Elemente von Java — **205**
4.2 Datentypen und Methoden — **212**
4.3 Ausführbare Java-Programme — **228**
4.4 Ausdrücke und Anweisungen — **234**
4.5 Klassen und Objekte — **248**
4.6 Fehler und Ausnahmen — **269**
4.7 Dateien: Ein- und Ausgabe — **277**
4.8 Threads — **282**
4.9 Lambdas, Ströme und Funktionale — **288**
4.10 Grafische Benutzeroberflächen mit dem AWT — **298**

5 Algorithmen und Datenstrukturen — 315
5.1 Suchalgorithmen — **317**
5.2 Einfache Sortierverfahren — **326**
5.3 Schnelle Sortieralgorithmen — **340**
5.4 Abstrakte Datenstrukturen — **361**
5.5 Stacks — **363**
5.6 Queues, Puffer, Warteschlangen — **371**
5.7 Container Datentypen — **375**
5.8 Bäume — **389**
5.9 Graphen — **419**
5.10 Zeichenketten — **436**

Literatur — 441

Stichwortverzeichnis — 445

Vorwort

Dieses dreiteilige Buch versteht sich als allgemeine Einführung in den Umgang mit Computern. Es ist damit gleichermaßen geeignet für Leser, die sich einen Überblick über das Thema Informatik verschaffen wollen, als auch für solche, die in das Thema einsteigen und mit Computern professionell arbeiten wollen. In erster Linie richtet es sich an Studenten, die Informatik im Haupt- oder Nebenfach studieren. Es ist gedacht als Begleitlektüre für die Vorlesungen des Grundstudiums und zur Einführung in die weiteren Themen der Informatik.

Der vorliegende erste Teil des Buches ist den Grundlagen der Programmierung gewidmet, insbesondere den Themen Programmierung, Algorithmen und Datenstrukturen. Im geplanten zweiten Band werden die Themen Rechnerarchitektur, Betriebssysteme, Rechnernetze und das Internet behandelt und der dritte Band wird sich mit der Theoretischen Informatik befassen.

Dieser Band beginnt mit einer Einführung in allgemeine Themen der Informatik. Dazu gehören grundlegende Fragen wie die Informationsdarstellung durch Bits und Bytes, die Darstellung und die Arithmetik von ganzen Zahlen und Dezimalzahlen. Nach einem kurzen Einschub zur geschichtlichen Entwicklung erklären wir den prinzipiellen Aufbau von Computersystemen von ihrer Architektur bis zu Betriebs- und Bediensystemen.

In vielen Bereichen, auch an Schulen und Universitäten, wird mittlerweile *Python* als erste Programmiersprache eingesetzt. Python ist leicht zu erlernen und unkompliziert, ist aber dennoch praxistauglich und vielfältig einsetzbar. Daher widmen wir Kapitel 2 einer Einführung in die Grundlagen der Programmierung anhand dieser Sprache. Im Folgekapitel stellen wir beispielhaft noch einige kleine Programmierprojekte in Python vor, die die Alltagstauglichkeit dieser Sprache unterstreichen. Beispielhaft demonstrieren wir die Turtlegraphik, die automatisierte Gewinnung von Informationen aus dem Internet und deren Darstellung mit Hilfe von Google-Karten, sowie das Messen und Steuern von Sensoren und Alarmgeräten basierend auf der Python-Schnittstelle des aktuellen Kleinrechners Raspberry.

Im folgenden Kapitel wenden wir uns der statisch typisierten objektorientierten Sprache *Java* zu, die für größere Programmierprojekte und für die Zusammenarbeit von Teams von Programmierern geeignete Unterstützung bietet. Diese Sprache ist für solche Anforderungen de fakto zum Industriestandard geworden. Nach einer umfas-

senden Einführung in *Java* werden auch die in der aktuellen Version 8 eingeführten Neuerungen, Lambdas und Ströme, ausführlich behandelt.

Das letzte Kapitel ist den grundlegenden Datenstrukturen der Informatik, wie Stacks, Listen, Bäumen und Graphen gewidmet, sowie den wichtigsten Algorithmen, die mit diesen Datenstrukturen arbeiten. Durch geeignete Wahl von Datenstrukturen läßt sich die Komplexität von Programmieraufgaben beurteilen und beherrschen. Statt einer rein theoretischen Diskussion zeigen wir auch, wie die diskutierten Datenstrukturen und ihre Algorithmen konkret in Java implementiert werden können.

Die Beispielprogramme zu diesem Buch, Errata, etc. werden wir auf der Webseite *www.informatikbuch.de* bereitstellen.

Marburg an der Lahn, im Juli 2016

Heinz-Peter Gumm
Manfred Sommer

Kapitel 1

Grundlagen

In diesem Kapitel werden wir wichtige Themen der Informatik in einer ersten Übersicht darstellen. Zunächst beschäftigen wir uns mit dem Begriff Informatik, dann mit fundamentalen Grundbegriffen wie z.B. Bits und Bytes. Danach behandeln wir die Frage, wie Texte, logische Werte und Zahlen in Computern gespeichert werden. Wir erklären den Aufbau eines PCs und das Zusammenwirken von Hardware, Controllern, Treibern und Betriebssystem bis zur benutzerfreundlichen Anwendungssoftware. Viele der hier eingeführten Begriffe werden in den späteren Kapiteln noch eingehender behandelt. Daher dient dieses Kapitel als erster Überblick und als Grundsteinlegung für die folgenden.

1.1 Was ist „Informatik“?

Der Begriff Informatik leitet sich von dem Begriff Information her. Er entstand in den 60er Jahren. Informatik ist die Wissenschaft von der maschinellen Informationsverarbeitung. Die englische Bezeichnung für Informatik ist *Computer Science*, also die Wissenschaft, die sich mit Rechnern beschäftigt. Wenn auch die beiden Begriffe verschiedene Blickrichtungen andeuten, bezeichnen sie dennoch das Gleiche. Die Spannweite der Disziplin Informatik ist sehr breit, und demzufolge ist das Gebiet in mehrere Teilgebiete untergliedert.

1.1.1 Technische Informatik

Die *Technische Informatik* beschäftigt sich vorwiegend mit der Konstruktion von Rechnern, Speicherchips, schnellen Prozessoren oder Parallelprozessoren, aber auch mit dem Aufbau von Peripheriegeräten wie Festplatten, Druckern und Bildschirmen. Die Grenzen zwischen der Technischen Informatik und der Elektrotechnik sind fließend. An einigen Universitäten gibt es den Studiengang Datentechnik, der gerade diesen Grenzbereich zwischen Elektrotechnik und Informatik zum Gegenstand hat.

Man kann vereinfachend sagen, dass die Technische Informatik für die Bereitstellung der Gerätschaften, der so genannten Hardware, zuständig ist, welche die Grundlage jeder maschinellen Informationsverarbeitung darstellt. Naturgemäß muss die Technische Informatik aber auch die beabsichtigten Anwendungsgebiete der Hardware im Auge haben. Insbesondere muss sie die Anforderungen der Programme berücksichtigen, die durch diese Hardware ausgeführt werden sollen. Es ist ein Unterschied, ob ein Rechner extrem viele Daten in begrenzter Zeit verarbeiten soll, wie etwa bei der Wettervorhersage oder bei der Steuerung einer Raumfähre, oder ob er im kommerziellen oder im häuslichen Bereich eingesetzt wird, wo es mehr auf die Unterstützung intuitiver Benutzerführung, die Präsentation von Grafiken, Text oder Sound ankommt.

1.1.2 Praktische Informatik

Die *Praktische Informatik* beschäftigt sich im weitesten Sinne mit den Programmen, die einen Rechner steuern. Im Gegensatz zur Hardware sind solche Programme leicht veränderbar, man spricht daher auch von Software. Es ist ein weiter Schritt von den recht primitiven Operationen, die die Hardware eines Rechners ausführen kann, bis zu den Anwendungsprogrammen, wie etwa Textverarbeitungssystemen, Spielen und Grafiksystemen, mit denen ein Anwender umgeht. Die Brücke zwischen der Hardware und der Anwendungssoftware zu schlagen, ist die Aufgabe der Praktischen Informatik.

Ein klassisches Gebiet der Praktischen Informatik ist der Compilerbau. Ein Compiler übersetzt Programme, die in einer technisch-intuitiven Notation, einer so genannten Programmiersprache, formuliert sind, in die stark von den technischen Besonderheiten der Maschine geprägte Notation der Maschinensprache. Es gibt viele populäre Programmiersprachen, darunter BASIC, Cobol, Fortran, Pascal, C, C++, C#, Java, Scala, JavaScript, PHP, Python, Perl, LISP, ML und PROLOG. Programme, die in solchen Hochsprachen formuliert sind, können nach der Übersetzung durch einen Compiler auf den verschiedensten Maschinen ausgeführt werden oder, wie es im Informatik-Slang heißt, laufen. Ein Programm in Maschinensprache läuft dagegen immer nur auf dem Maschinentyp, für den es geschrieben wurde.

1.1.3 Theoretische Informatik

Die *Theoretische Informatik* beschäftigt sich mit den abstrakten mathematischen und logischen Grundlagen aller Teilgebiete der Informatik. Theorie und Praxis sind in der Informatik enger verwoben, als in vielen anderen Disziplinen, theoretische Erkenntnisse sind schneller und direkter einsetzbar. Durch die theoretischen Arbeiten auf dem Gebiet der formalen Sprachen und der Automatentheorie zum Beispiel hat man das Gebiet des Compilerbaus heute sehr gut im Griff. In Anlehnung an die theoretischen Erkenntnisse sind praktische Werkzeuge entstanden. Diese sind selbst wie-

der Programme, mit denen ein großer Teil des Compilerbaus automatisiert werden kann. Bevor eine solche Theorie existierte, musste man mit einem Aufwand von ca. 25 *Bearbeiter-Jahren* (Anzahl der Bearbeiter * Arbeitszeit = 25) für die Konstruktion eines einfachen Compilers rechnen, heute erledigen Studenten eine vergleichbare Aufgabe im Rahmen eines Praktikums.

Neben den Beiträgen, die die Theoretische Informatik zur Entwicklung des Gebietes leistet, ist die Kenntnis der theoretischen Strukturen eine wichtige Schulung für jeden, der komplexe Systeme entwirft. Gut durchdachte, theoretisch abgesicherte Entwürfe erweisen sich auch für hochkomplexe Software als sicher und erweiterbar. Software-Systeme, die im Hauruck-Verfahren entstehen, stoßen immer bald an die Grenze, ab der sie nicht mehr weiterentwickelt werden können. Die Entwicklung von Software sollte sich an der Ökonomie der Theoriebildung in der Mathematik orientieren – möglichst wenige Annahmen, möglichst keine Ausnahmen. Ein wichtiger Grund etwa, warum die Konstruktion von Fortran-Compilern so kompliziert ist, liegt darin, dass dieses Prinzip bei der Definition der Programmiersprache nicht angewendet wurde. Jeder Sonder- oder Ausnahmefall, jede zusätzliche Regel macht nicht nur dem Konstrukteur des Compilers das Leben schwer, sondern auch den vielen Fortran-Programmierern.

1.1.4 Angewandte Informatik

Die *Angewandte Informatik* beschäftigt sich mit dem Einsatz von Rechnern in den verschiedensten Bereichen unseres Lebens. Da in den letzten Jahren die Hardware eines Rechners für jeden erschwinglich geworden ist, gibt es auch keinen Bereich mehr, der der Computeranwendung verschlossen ist. Einerseits gilt es, spezialisierte Programme für bestimmte Aufgaben zu erstellen, andererseits müssen Programme und Konzepte entworfen werden, die in vielfältigen Umgebungen einsetzbar sein sollen. Beispiele für solche universell einsetzbaren Systeme sind etwa Textverarbeitungssysteme oder Tabellenkalkulationssysteme (engl. *spreadsheet*). Angewandte Informatik nutzt heute jeder, der im Internet die aktuelle Tageszeitung liest, seine E-Mail erledigt, Bankgeschäfte tätigt, chattet, sich in sozialen Netzwerken tummelt, Musik herunterlädt, oder nur Filme anschaut.

Auch die Angewandte Informatik ist nicht isoliert von den anderen Gebieten denkbar. Es gilt schließlich, sowohl neue Möglichkeiten der Hardware als auch im Zusammenspiel von Theoretischer und Praktischer Informatik entstandene Werkzeuge einer sinnvollen Anwendung zuzuführen. Als Beispiel mögen die Smartphones, Organizer und Tablet-PCs dienen, die im Wesentlichen aus einem Flüssigkristall-Bildschirm bestehen, den man mit einem Griffel oder einfach mit Fingergesten bedienen kann. Die Angewandte Informatik muss die Einsatzmöglichkeiten solcher Geräte, etwa in der mobilen Lagerhaltung, auf der Baustelle oder als vielseitiger, „intelligenter“ Terminkalender entwickeln. Die Hardware wurde von der Technischen Informatik konstru-

iert, die Softwaregrundlagen, etwa zur Handschrifterkennung, von der Praktischen Informatik aufgrund der Ergebnisse der Theoretischen Informatik gewonnen.

Wenn man im deutschsprachigen Raum auch diese Einteilung der Informatik vornimmt, so ist es klar, dass die einzelnen Gebiete nicht isoliert und ihre jeweiligen Grenzen nicht wohldefiniert sind. Die Technische Informatik überlappt sich stark mit der Praktischen Informatik, jene wieder mit der Theoretischen Informatik. Auch die Grenzen zwischen der Praktischen und der Angewandten Informatik sind fließend. Gleichgültig in welchem Bereich man später einmal arbeiten möchte, muss man auch die wichtigsten Methoden der Nachbargebiete kennen lernen, um die Möglichkeiten seines Gebietes entfalten und entwickeln zu lernen, aber auch um die Grenzen abschätzen zu können.

Neben der in diese vier Bereiche eingeteilten Informatik haben viele Anwendungsgebiete ihre eigenen Informatik-Ableger eingerichtet. So spricht man zum Beispiel von der *Medizinischen Informatik*, der *Wirtschaftsinformatik*, der *Medieninformatik*, der *Bio-Informatik*, der *Linguistischen Informatik*, der *Juristischen Informatik* oder der *Chemie-Informatik*. Für einige dieser Bereiche gibt es an Fachhochschulen und Universitäten bereits Studiengänge. Insbesondere geht es darum, fundierte Kenntnisse über das angestrebte Anwendungsgebiet mit grundlegenden Kenntnissen informatischer Methoden zu verbinden.

1.2 Information und Daten

Was tut eigentlich ein Computer? Diese Frage scheint leicht beantwortbar zu sein, indem wir einfach eine Fülle von Anwendungen aufzählen. Computer berechnen Wettervorhersagen, steuern Raumfähren, spielen Schach, machen Musik und erzeugen verblüffende Effekte in Kinofilmen. Sicher liegt hier aber nicht die Antwort auf die gestellte Frage, denn wir wollen natürlich wissen, *wie* Computer das machen. Um dies zu erklären, müssen wir uns zunächst einigen, in welcher Tiefe wir anfangen sollen. Bei der Erklärung des Schachprogramms wollen wir vielleicht wissen:

- Wie wird das Schachspiel des Computers bedient?
- Wie ist das Schachprogramm aufgebaut?
- Wie sind die Informationen über den Spielstand im Hauptspeicher des Rechners gespeichert und wie werden sie verändert?
- Wie sind die Nullen und Einsen in den einzelnen Speicherzellen organisiert und wie werden sie verändert?
- Welche elektrischen Signale beeinflussen die Transistoren und Widerstände, aus denen Speicherzellen und Prozessor aufgebaut sind?

Wir müssen uns auf *eine* solche mögliche Erklärungsebene festlegen. Da es hier um Informatik geht, also um die Verarbeitung von Informationen, beginnen wir auf der Ebene der Nullen und Einsen, denn dies ist die niedrigste Ebene der *Informationsver-*

arbeitung. Wir beschäftigen uns also zunächst damit, wie Informationen im Rechner durch Nullen und Einsen repräsentiert werden können. Die so repräsentierten Informationen nennen wir *Daten*. Die *Repräsentation* muss derart gewählt werden, dass man aus den Daten auch wieder die repräsentierte Information zurückgewinnen kann. Diesen Prozess der Interpretation von Daten als Information nennt man auch *Abstraktion*.

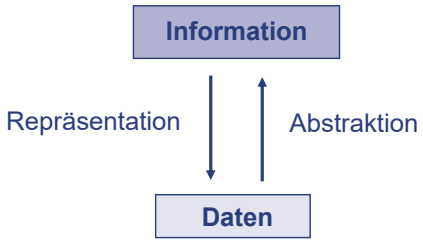


Abb. 1.1. Information und Daten

1.2.1 Bits

Ein *Bit* ist die kleinstmögliche Einheit der Information. Ein Bit ist die Informationsmenge in einer Antwort auf eine Frage, die zwei Möglichkeiten zulässt

- ja oder nein,
- wahr oder falsch,
- schwarz oder weiß,
- hell oder dunkel,
- groß oder klein,
- stark oder schwach,
- links oder rechts.

Zu einer solchen Frage lässt sich immer eine Codierung der Antwort festlegen. Da es zwei mögliche Antworten gibt, reicht ein Code mit zwei Zeichen, ein so genannter *binärer Code*. Man benutzt dazu die Zeichen

0 und 1.

Eine solche Codierung ist deswegen nötig, weil die Information technisch dargestellt werden muss. Man bedient sich dabei etwa elektrischer Ladungen

0 = ungeladen,
1 = geladen,

oder elektrischer Spannungen

0 = 0 Volt,
1 = 5 Volt,
oder Magnetisierungen
0 = unmagnetisiert,
1 = magnetisiert.
So kann man etwa die Antwort auf die Frage
Welche Farbe hat der Springer auf F3?
im Endeffekt dadurch repräsentieren bzw. auffinden, indem man prüft,
- ob ein Kondensator eine bestimmte Ladung besitzt,
- ob an einem Widerstand eine bestimmte Spannung anliegt oder
- ob eine bestimmte Stelle auf einer Magnetscheibe magnetisiert ist.

Da es uns im Moment aber nicht auf die genaue technische Realisierung ankommt, wollen wir die Übersetzung physikalischer Größen in Informationseinheiten bereits voraussetzen und nur von den beiden möglichen Elementarinformationen 0 und 1 ausgehen. Mit $\bar{0} = 1$ und $\bar{1} = 0$ bezeichnet man die jeweils komplementären Bits.

1.2.2 Bitfolgen

Lässt eine Frage mehrere Antworten zu, so enthält die Beantwortung der Frage mehr als ein Bit an Information. Die Frage etwa, aus welcher Himmelsrichtung, Nord, Süd, Ost oder West, der Wind weht, lässt 4 mögliche Antworten zu. Der Informationsgehalt in der Beantwortung der Frage ist aber nur 2 Bit, denn man kann die ursprüngliche Frage in zwei andere Fragen verwandeln, die jeweils nur zwei Antworten zulassen:
1. Weht der Wind aus einer der Richtungen Nord oder Ost (ja/nein)?
2. Weht der Wind aus einer der Richtungen Ost oder West (ja/nein)?

Eine mögliche Antwort, etwa *ja* auf die erste Frage und *nein* auf die zweite Frage, lässt sich durch die beiden Bits

```
1 0
```

repräsentieren. Die Bitfolge 10 besagt also diesmal, dass der Wind aus Norden weht. Ähnlich repräsentieren die Bitfolgen

```
0 0 = Süd
0 1 = West
1 0 = Nord
1 1 = Ost.
```

Offensichtlich gibt es genau 4 mögliche Folgen von 2 Bit. Mit 2 Bit können wir also Fragen beantworten, die 4 mögliche Antworten zulassen. Lassen wir auf dieselbe Frage

(Woher weht der Wind?) auch noch die Zwischenrichtungen *Südost, Nordwest, Nordost* und *Südwest* zu, so gibt es 4 weitere mögliche Antworten, also insgesamt 8. Mit einem zusätzlichen Bit, also mit insgesamt 3 Bits, können wir alle 8 möglichen Antworten darstellen. Die möglichen Folgen aus 3 Bits sind

```
000, 001, 010, 011, 100, 101, 110, 111.
```

und die möglichen Antworten auf die Frage nach der Windrichtung sind

```
Süd, West, Nord, Ost, Südost, Nordwest, Nordost, Südwest.
```

Jede beliebige eindeutige Zuordnung der Himmelsrichtungen zu diesen Bitfolgen können wir als Codierung von Windrichtungen hernehmen, zum Beispiel

```
000 = Süd
100 = Südost
001 = West
101 = Nordwest
010 = Nord
110 = Nordost
011 = Ost
111 = Südwest
```

Offensichtlich verdoppelt jedes zusätzliche Bit die Anzahl der möglichen Bitfolgen, so dass gilt

Es gibt genau 2^n verschiedene Bitfolgen der Länge n.

1.2.3 Hexziffern

Ein Rechner ist viel besser als ein Mensch in der Lage, mit Kolonnen von Bits umzugehen. Für den Menschen wird eine lange Folge von Nullen und Einsen bald unübersichtlich. Es wird etwas einfacher, wenn wir lange Bitfolgen in Gruppen zu 4 Bits anordnen. Aus einer Bitfolge wie `01001111011000010110110001101100` wird dann

```
0100 1111 0110 0001 0110 1100 0110 1100
```

Eine Gruppe von 4 Bits nennt man auch *Halb-Byte* oder *Nibble*. Da nur $2^4 = 16$ verschiedene Nibbles möglich sind, bietet es sich an, jedem einen Namen zu geben. Wir wählen dazu die Ziffern „0“ bis „9“ und zusätzlich die Zeichen „A“ bis „F“. Jedem Halb-Byte ordnet man auf natürliche Weise eine dieser so genannten *Hexziffern* zu

```
0000=0 0100=4 1000=8 1100=C
0001=1 0101=5 1001=9 1101=D
```

```
0010=2 0110=6 1010=A 1110=E
0011=3 0111=7 1011=B 1111=F.
```

Damit lässt sich die obige Bitfolge kompakter als Folge von Hexziffern darstellen:

```
4 F 6 1 6 C 6 C.
```

Die Rückübersetzung in eine Bitfolge ist ebenso einfach, wir müssen nur jede Hexziffer durch das entsprechende Halb-Byte ersetzen.

So wie sich eine Folge von Dezimalziffern als Zahl im Dezimalsystem deuten lässt, können wir eine Folge von Hexziffern auch als eine Zahl im Sechzehner- oder Hexadezimal-System auffassen. Den Zahlenwert einer Folge von Hexziffern erhalten wir, indem wir jede Ziffer entsprechend ihrer Ziffernposition mit der zugehörigen Potenz der Basiszahl 16 multiplizieren und die Ergebnisse aufsummieren. Ähnlich wie die Dezimalzahl 327 für den Zahlenwert

$$3 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

steht, repräsentiert z.B. die Hexzahl 1AF3 den Zahlenwert 6899, denn

$$1 \times 16^3 + A \times 16^2 + F \times 16^1 + 3 \times 16^0 = 1 \times 4096 + 10 \times 256 + 15 \times 16 + 3 = 6899.$$

Die Umwandlung einer Dezimalzahl in eine Hexzahl mit dem gleichen Zahlenwert ist etwas schwieriger, wir werden darauf eingehen, wenn wir die verschiedenen Zahldarstellungen behandeln. Da man das Hex-System vorwiegend verwendet, um lange Bitfolgen kompakter darzustellen, ist eine solche Umwandlung selten nötig.

Die Hex-Darstellung wird von Assembler-Programmierern meist der Dezimaldarstellung vorgezogen. Daher findet man oft auch die ASCII-Tabelle (siehe Abb. 1.2), welche eine Zuordnung der 256 möglichen Bytes (s.u.) zu den Zeichen der Tastatur und anderen Sonderzeichen festlegt, in Hex-Darstellung. Für das ASCII-Zeichen 'o' (das kleine „Oh“, nicht zu verwechseln mit der Ziffer „0“) hat man dann den Eintrag 6F, was der Dezimalzahl $6 \times 16 + 15 = 111$ entspricht. Umgekehrt findet man zu dem 97-sten ASCII-Zeichen, dem kleinen „a“, die Hex-Darstellung 61, denn $6 \times 16 + 1 = 97$.

Allein aus der Ziffernfolge „61“ ist nicht ersichtlich, ob diese als Hexadezimalzahl oder als Dezimalzahl aufzufassen ist. Wenn eine Verwechslung nicht ausgeschlossen ist, hängt man zur Kennzeichnung von Hexzahlen ein kleines „h“ an, also 61h. Gelegentlich benutzt man die *Basiszahl* des Zahlensystems auch als unteren Index, wie in der folgenden Gleichung

$$97_{10} = 61_{16} = 01100001_2.$$

Programmiersprachen verlangen oft, Hexzahlen durch Voranstellen von **0x** zu kennzeichnen. Beispielsweise gibt man in Java die Hexzahl 61h als 0x61 an.

1.2.4 Bytes und Worte

Wenn ein Rechner Daten liest oder schreibt, wenn er mit Daten operiert, gibt er sich nie mit einzelnen Bits ab. Dies wäre im Endeffekt viel zu langsam. Stattdessen arbeitet er immer nur mit Gruppen von Bits, entweder mit 8 Bits, 16 Bits, 32 Bits oder 64 Bits. Man spricht dann von 8-Bit-Rechnern, 16-Bit-Rechnern, 32-Bit-Rechnern oder 64-Bit-Rechnern. In Wahrheit gibt es aber auch Mischformen – Rechner, die etwa intern mit 32-Bit-Blöcken rechnen, aber immer nur Blöcke zu 64 Bits lesen oder schreiben. Stets jedoch ist die Länge eines Bitblocks ein Vielfaches von 8. Eine Gruppe von 8 Bits nennt man ein *Byte*. Ein Byte besteht infolgedessen aus zwei Nibbles, man kann es also durch zwei Hex-Ziffern darstellen. Es gibt daher $16^2 = 256$ verschiedene Bytes von 0000 0000 bis 1111 1111. In Hexzahlen ausgedrückt erstreckt sich dieser Bereich von 00h bis FFh, dezimal von 0 bis 255.

Für eine Gruppe von 2, 4 oder 8 Bytes sind auch die Begriffe Wort, Doppelwort und Quadwort im Gebrauch, allerdings ist die Verwendung dieser Begriffe uneinheitlich. Bei einem 16-Bit Rechner bezeichnet man eine 16-Bit Größe als Wort, ein Byte ist dann ein Halbwort. Bei einem 32-Bit Rechner steht „Wort“ auch für eine Gruppe von 4 Bytes.

1.2.5 Dateien

Eine *Datei* ist eine beliebig lange Folge von Bytes. Dateien werden meist auf Festplatten, USB-Sticks oder anderen Datenträgern gespeichert. Jede Information, mit der ein Rechner umgeht, Texte, Zahlen, Musik, Bilder, Programme, muss sich auf irgendeine Weise als Folge von Bytes repräsentieren lassen und kann daher als Datei gespeichert werden.

Hat man nur den Inhalt einer Datei vorliegen, so kann man nicht entscheiden, welche Art von Information die enthaltenen Bytes repräsentieren sollen. Diese zusätzliche Information versucht man durch einen geeigneten Dateinamen auszudrücken. Insbesondere hat es sich eingebürgert, die Dateinamen aus zwei Teilen zusammenzusetzen, einem Namen und einer Erweiterung. Diese beiden Namensbestandteile werden durch einen Punkt getrennt. Beispielsweise besteht die Datei mit Namen „FoxyLady.wav“ aus dem Namen „FoxyLady“ und der Erweiterung „wav“. Die Endung „wav“ soll andeuten, dass es sich um eine unkomprimierte Musikdatei handelt, die mit einer entsprechenden Software abgespielt werden kann. Nicht alle Betriebssysteme verwenden diese Konventionen. In UNIX ist es z.B. üblich den Typ der Datei in den ersten Inhalts-Bytes zu kennzeichnen.

1.2.6 Datei- und Speichergrößen

Unter der *Größe einer Datei* versteht man die Anzahl der darin enthaltenen Bytes. Man verwendet dafür die Einheit B. Eine Datei der Größe 245B enthält also 245 Byte. In einigen Fällen wird die Abkürzung B auch für ein Bit verwendet, so dass wir es vorziehen,

bei Verwechslungsgefahr die Einheiten als Byte oder als Bit auszuschreiben. Dateien von wenigen hundert Byte sind äußerst selten, meist bewegen sich die Dateigrößen in Bereichen von Tausenden, Millionen oder gar Milliarden von Bytes. Es bietet sich an, dafür die von Gewichts- oder Längenmaßen gewohnten Präfixe kilo- (für tausend) und mega- (für million) zu verwenden. Andererseits ist es günstig, beim Umgang mit binären Größen auch die Faktoren durch Zweierpotenzen 2, 4, 8, 16, ... auszudrücken. Da trifft es sich gut, dass die Zahl 1000 sehr nahe bei einer Zweierpotenz liegt, nämlich

$$2^{10} = 1024.$$

Daher stehen in vielen Bereichen der Informatik das Präfix *kilo* für 1024 und das Präfix *mega* für

$$2^{20} = 1024 \times 1024 = 1048576.$$

Die Abkürzungen für die in der Informatik benutzten Größenfaktoren sind daher

$$\begin{aligned}
k &= 2^{10} \qquad (k = kilo) \\
M &= 1024 \times 1024 = 2^{20} \qquad (M = mega) \\
G &= 1024 \times 1024 \times 1024 = 2^{30} \qquad (G = giga) \\
T &= 1024 \times 1024 \times 1024 \times 1024 = 2^{40} \qquad (T = tera) \\
P &= 1024 \times 1024 \times 1024 \times 1024 \times 1024 = 2^{50} \qquad (P = peta) \\
E &= 1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024 = 2^{60} \qquad (E = exa)
\end{aligned}$$

Die obigen Maßeinheiten haben sich auch für die Angabe der Größe des Hauptspeichers und anderer Speichermedien eingebürgert. Allerdings verwenden Hersteller von Festplatten, DVDs, und Blu-ray-Discs meist G (Giga) für den Faktor 10^9 statt für 2^{30}. So kann es sein, dass der Rechner auf einer 500 GByte Festplatte nur 465 GByte Speicherplatz erkennt. Dies liegt daran, dass folgendes gilt:

$$500 \times 10^9 \approx 465 \times 2^{30}$$

Die IEC (International Electrotechnical Commission) schlug bereits 1996 vor, die in der Informatik benutzten Größenfaktoren, die auf den Zweierpotenzen basieren, mit einem kleinen „i“ zu kennzeichnen, also ki, Mi, Gi, etc. und die Präfixe k, M, G für die Zehnerpotenzen zu reservieren. Dann hätte obige Festplatte 500 GB, aber 465 GiB (ausgesprochen: Gibibyte). Dieser Vorschlag hat sich aber bisher nicht durchgesetzt.

Anhaltspunkte für gängige Größenordnungen von Dateien und Geräten sind:

$\sim 200\,B$	eine kurze Textnotiz
$\sim 4\,kB$	dafür benötigter Platz auf der Festplatte
$\sim 100\,kB$	formatierter Brief, Excel Datei, pdf-Dokument ohne Bildern
$\sim 4\,MB$	Musiktitel im mp3-Format
$\sim 40\,MB$	Musiktitel im wav-Format
$\sim 700\,MB$	CD-ROM Kapazität
$\sim 4\,GB$	DVD
$\sim 16\,GB$	PC Hauptspeicher – bis 64 GB
$25\,GB$	Blu-ray Disc (dual-layer – 50 GB)
$32\,GB$	USB-Stick, SD-Karten – bis 128 GB
$256\,GB$	Halbleiterspeichermedien (SSD = Solid State Disk) – bis 4 TB
$2\,TB$	Festplatten – bis 16 TB

Natürlich hängt die genaue Größe einer Datei von ihrem Inhalt, im Falle von Bild- oder Audiodateien auch von der Spieldauer und dem verwendeten Aufzeichnungsverfahren ab. Durch geeignete Kompressionsverfahren lässt sich ohne merkliche Qualitätsverluste die Dateigröße erheblich reduzieren. So kann man z.B. mithilfe des MP3-Codierungsverfahrens einen Musiktitel von 40 MB Größe auf ca. 4 MB komprimieren. Dadurch ist es möglich, auf einer einzigen CD den Inhalt von 10 – 12 herkömmlichen Musik-CDs zu speichern.

1.2.7 Längen- und Zeiteinheiten

Für Längen- und Zeitangaben werden auch in der Informatik dezimale Einheiten benutzt. So ist z.B. ein 2,6 GHz Prozessor mit $2,6 \times 10^9 = 2600000000$ Hertz (Schwingungen pro Sekunde) getaktet. Ein Takt dauert also $1/(2,6 \times 10^9) = 0,38 \times 10^{-9}$ sec, das sind 0,38 ns. Das Präfix *n* steht hierbei für *nano*, also den Faktor 10^{-9}. Die anderen Faktoren kleiner als 1 sind:

$$
\begin{aligned}
m &= 1/1000 = 10^{-3} \quad (m = milli) \\
\mu &= 1/1000000 = 10^{-6} \quad (\mu = mikro) \\
n &= 1/1000000000 = 10^{-9} \quad (n = nano) \\
p &= 1/1000000000000 = 10^{-12} \quad (p = pico) \\
f &= 1/1000000000000000 = 10^{-15} \quad (f = femto)
\end{aligned}
$$

Für Längenangaben wird neben den metrischen Maßen eine im Amerikanischen immer noch weit verbreitete Einheit verwendet. Für amerikanische Längenmaße hat sich nicht einmal das Dezimalsystem durchgesetzt.

1" = 1 in = 1 inch = 1 Zoll = 2,54 cm = 25,4 mm.

1.3 Informationsdarstellung

Als *Daten* bezeichnen wir die Folgen von Nullen und Einsen, die irgendwelche Informationen repräsentieren. In diesem Abschnitt werden wir die Repräsentation von Texten, logischen Werten, Zahlen und Programmen durch Daten erläutern.

1.3.1 Text

Um *Texte* in einem Rechner darzustellen, codiert man Alphabet und Satzzeichen in Bitfolgen. Mit einem Alphabet von 26 Kleinbuchstaben, ebenso vielen Großbuchstaben, einigen Satzzeichen wie etwa Punkt, Komma und Semikolon und Spezialzeichen wie „+“, „&“ und „%“ hat eine normale Tastatur eine Auswahl von knapp hundert Zeichen. Die Information, wo ein Zeilenumbruch stattfinden oder wo ein Text eingerückt werden soll, codiert man ebenfalls durch spezielle Zeichen. Solche *Sonderzeichen*, dazu gehören das CR-Zeichen (von englisch *carriage return* = Wagenrücklauf) und das Tabulatorzeichen *Tab*, werden nie ausgedruckt, sie haben beim Ausdrucken lediglich die entsprechende steuernde Wirkung. Sie heißen daher auch Steuerzeichen oder nicht-druckbare Zeichen.

1.3.2 ASCII-Code

Auf jeden Fall kommt man für die Darstellung aller Zeichen mit 7 Bits aus, das ergibt $2^7 = 128$ verschiedene Möglichkeiten. Man muss also nur eine Tabelle erstellen, mit der jedem Zeichen ein solcher Bitcode zugeordnet wird. Dazu nummeriert man die 128 gewählten Zeichen einfach durch und stellt die Nummer durch 7 Bit binär dar.

Die heute fast ausschließlich verwendete Nummerierung ist die so genannte ASCII-Codierung. ASCII steht für „*American Standard Code for Information Interchange*“. Sie berücksichtigt einige Systematiken, insbesondere sind Ziffern, Großbuchstaben und Kleinbuchstaben in natürlicher Reihenfolge durchnummeriert.

Die in Abbildung 1.2 dargestellte Tabelle zeigt alle Zeichen mit ASCII-Codes zwischen 0 und 127, das sind hexadezimal 00h bis 7Fh. Die Ziffern 0 bis 9 haben die Codes 30h bis 39h, die Großbuchstaben bzw. die Kleinbuchstaben haben die Nummern 41h-5Ah bzw. 61h-7Ah.

Die Zeichen mit Nummern 21h bis 7Eh nennt man die druckbaren Zeichen. Zu ihnen gehören neben Ziffern und Buchstaben auch die verschiedenen Sonderzeichen:

```
! " # $ % & ’ ( ) * + , - . / = ; < = > ? @ [ \ ] ^_ ‘ { | } ~.
```

Abb. 1.2. ASCII-Tabelle

Code	...0	...1	...2	...3	...4	...5	...6	...7	...8	...9	...A	...B	...C	...D	...E	...F
0...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1...	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2...	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3...	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4...	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5...	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6...	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7...	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ASCII 20h repräsentiert das Leerzeichen SP (engl.: *space*). CR und LF (ASCII 0Dh und 0Ah) stehen für carriage return (Wagenrücklauf) und line feed (Zeilenvorschub). Hier handelt es sich um sogenannte Steuerzeichen, mit denen man früher Fernschreiber, also elektrisch angesteuerte Schreibmaschinen steuerte. In diese Kategorie gehören auch BS (backspace ASCII 08h), HT (horizontal tab = Tabulator) und FF (form feed = neue Seite). Diese Zeichen werden bei der Repräsentation von Texten auch heute noch in dieser Bedeutung verwendet. Allerdings verwendet man für eine neue Zeile unter Linux und Mac OS X nur das Zeichen LF (ASCII 0A), während Windows die Kombination CR LF erwartet.

Da der ASCII-Code zur Datenübertragung (*information interchange*) konzipiert wurde, vereinbarte man weitere Zeichen zur Steuerung einer Datenübertragung: STX und ETX (start/end of text), ENQ (enquire), ACK (acknowledge), NAK (negative acknowledge), EOT (end of transmission). Mit BEL (bell = Glocke) konnte man den Empfänger alarmieren.

Standardtastaturen haben je eine Taste für die meisten druckbaren Zeichen. Allerdings sind viele Tasten mehrfach belegt. Großbuchstaben und viele Satzzeichen erreicht man nur durch gleichzeitiges Drücken der Shift-Taste. Durch Kombinationen mit Ctrl, der Control-Taste, (auf deutschen Tastaturen oft mit Strg (Steuerung) bezeichnet) kann man viele Steuerzeichen eingeben. Zum Beispiel entspricht ASCII 07h (das Klingelzeichen) der Kombination Ctrl-G und ASCII 08h (backspace) ist Ctrl-H. Einige dieser Codes können in Editoren oder auf der Kommandozeile direkt benutzt werden. Die gebräuchlichen Tastaturen spendieren den wichtigsten Steuerzeichen eine eigene Taste. Dazu gehören u.a. Ctrl-I (Tabulator), Ctrl-H (Backspace), Ctrl-[(Escape = ASCII 1Bh) und Ctrl-M (Return).

Die 128 ASCII-Zeichen entsprechen den Bytes 0000 0000 bis 0111 1111, d.h. den Hex-Zahlen 00 bis 7F. Eine Datei, die nur ASCII-Zeichen enthält, also Bytes, deren ers-

tes Bit 0 ist, nennt man ASCII-Datei. Oft versteht man unter einer ASCII-Datei auch einfach eine Textdatei, selbst wenn Codes aus einer ASCII-Erweiterung verwendet werden.

1.3.3 ASCII-Erweiterungen

Bei der ASCII-Codierung werden nur die letzten 7 Bits eines Byte genutzt. Das erste Bit verwendete man früher als Kontrollbit für die Datenübertragung. Es wurde auf 0 oder 1 gesetzt, je nachdem ob die Anzahl der 1-en an den übrigen 7 Bitpositionen gerade (*even*) oder ungerade (*odd*) war. Die Anzahl der 1-en in dem so aufgefüllten Byte wurde dadurch immer gerade (*even parity*). Wenn nun bei der Übertragung ein kleiner Fehler auftrat, d.h. wenn in dem übertragenen Byte genau ein Bit verfälscht wurde, so konnte der Empfänger dies daran erkennen, dass die Anzahl der 1-en ungerade war.

Bei der Verwendung des ASCII-Codes zur Speicherung von Texten und auch als Folge der verbesserten Qualität der Datenübertragung wurde dieses Kontrollbit überflüssig. Daher lag es nahe, nun alle 8 Bit zur Zeichenkodierung zu verwenden. Somit ergab sich ein weiterer verfügbarer Bereich von ASCII 128 bis ASCII 255. Der von IBM ab 1980 entwickelte PC benutzte diese zusätzlichen Codes zur Darstellung von sprachspezifischen Zeichen wie z.B. „ä“ (ASCII 132), „ö“ (ASCII 148) „ü“ (ASCII 129) und einigen Sonderzeichen anderer Sprachen, darüber hinaus auch für Zeichen, mit denen man einfache grafische Darstellungen wie Rahmen und Schraffuren zusammensetzen kann. Diese Zeichen können über die numerische Tastatur eingegeben werden. Dazu muss diese aktiviert sein (dies geschieht durch die Taste „Num“), danach kann bei gedrückter „Alt“-Taste der dreistellige ASCII-Code eingegeben werden.

Leider ist auch die Auswahl der sprachspezifischen Sonderzeichen eher zufällig und bei weitem nicht ausreichend für die vielfältigen Symbole fremder Schriften. Daher wurden von der International Organization for Standardization (ISO) verschiedene andere Optionen für die Nutzung der ASCII-Codes 128-255 als sog. ASCII-Erweiterungen normiert. In Europa ist die ASCII-Erweiterung ISO Latin-1 verbreitet, die durch die Norm ISO 8859-1 beschrieben wird.

Einige Rechner, insbesondere wenn sie unter UNIX betrieben werden, benutzen nur die genormten ASCII-Zeichen von 0 bis 127. Auf solchen Rechnern sind daher Umlaute nicht so einfach darstellbar. Die Verwendung von Zeichen einer ASCII-Erweiterung beim Austausch von Daten, E-Mails oder Programmtexten ist ebenfalls problematisch. Benutzt der Empfänger zur Darstellung nicht die gleiche ASCII-Erweiterung, so findet er statt der schönen Sonderzeichen irgendwelche eigenartigen Symbole oder Kontrollzeichen in seinem Text. Schlimmstenfalls geht auch von jedem Byte das erste Bit verloren. Einen Ausweg bietet hier die Umcodierung der Datei in eine ASCII-Datei (z.B. mit dem Programm „uuencode“) vor der Übertragung und eine Dekodierung beim Empfänger (mittels „uudecode“). Viele E-Mail Programme führen solche Umkodierungen automatisch aus.

1.3.4 Unicode und UCS

Wegen der Problematik der ASCII-Erweiterungen bei der weltweiten Datenübertragung entstand in den letzten Jahren ein neuer Standard, der versucht, sämtliche relevanten Zeichen aus den unterschiedlichsten Kulturkreisen in einem universellen Code zusammenzufassen. Dieser neue Zeichensatz heißt *Unicode* und verwendet eine 16-Bit-Codierung, kennt also maximal 65536 Zeichen. Landesspezifische Zeichen, wie z.B. ö, ß, æ, ç oder Ã gehören ebenso selbstverständlich zum Unicode-Zeichensatz wie kyrillische, arabische, japanische und tibetische Schriftzeichen. Die ersten 128 Unicode-Zeichen sind identisch mit dem ASCII-Code, die nächsten 128 mit dem ISO Latin-1 Code.

Unicode wurde vom Unicode-Konsortium (www.unicode.org) definiert. Dieses arbeitet ständig an neuen Versionen und Erweiterungen dieses Zeichensatzes. Die Arbeit des Unicode-Konsortium wurde von der ISO (www.iso.ch) aufgegriffen. Unter der Norm ISO-10646 wurde Unicode als Universal Character Set (UCS) international standardisiert. Beide Gremien bemühen sich darum, ihre jeweiligen Definitionen zu synchronisieren, um unterschiedliche Codierungen zu vermeiden. ISO geht allerdings in der grundlegenden Definition von UCS noch einen Schritt weiter als Unicode. Es werden sowohl eine 16-Bit-Codierung (UCS-2) als auch eine 31-Bit-Codierung (UCS-4) festgelegt. Die Codes von UCS-2 werden als *basic multilingual plane* (BMP) bezeichnet, beinhalten alle bisher definierten Codes und stimmen mit Unicode überein. Codes, die UCS-4 ausnutzen sind für potenzielle zukünftige Erweiterungen vorgesehen.

Die Einführung von Unicode bzw. UCS-2 und UCS-4 führt zu beträchtlichen Kompatibilitätsproblemen, ganz abgesehen davon, dass der Umfang von derart codierten Textdateien wächst. Es ist daher schon frühzeitig der Wunsch nach einer kompakteren Codierung artikuliert worden, die kompatibel mit der historischen 7-Bit ASCII Codierung ist und die den neueren Erweiterungen Rechnung trägt. Eine solche Codierung, mit dem Namen UTF-8, wurde auch tatsächlich in den 90er Jahren eingeführt. Sie wurde von der ISO unter dem Anhang R zur Norm ISO-10646 festgeschrieben und auch von den Internetgremien als RFC2279 standardisiert.

1.3.5 UTF-8

Die Bezeichnung UTF ist eine Abkürzung von UCS Transformation Format. Dadurch wird betont, dass es sich lediglich um eine andere Codierung von UCS bzw. Unicode handelt.

UTF-8 ist eine Mehrbyte-Codierung. 7-Bit ASCII-Zeichen werden mit einem Byte codiert, alle anderen verwenden zwischen 2 und 6 Bytes. Die Idee ist, dass häufig benutzte Zeichen mit einem Byte codiert werden, seltenere mit mehreren Bytes. Die Kodierung erfolgt nach den folgenden Prinzipien:

– Jedes mit 0 beginnende Byte ist ein Standard 7-Bit ASCII Zeichen.

– Jedes mit 1 beginnende Byte gehört zu einem aus mehreren Bytes bestehenden UTF-8 Code. Besteht ein UTF-8 Code aus $n \geq 2$ Bytes, so beginnt das erste (Start–) Byte mit n vielen 1-en, und jedes der $n - 1$ Folgebytes mit der Bitfolge 10.

Der erste Punkt garantiert, dass Teile eines Mehrbyte UTF-8 Zeichens nicht als 7-Bit-ASCII Zeichen missdeutet werden können. Der zweite Punkt erlaubt es, Wortgrenzen in einer UTF-8 codierten Datei leicht zu erkennen, was ein einfaches Wiederaufsetzen bei einem Übertragungsfehler ermöglicht. Auch einfache syntaktische Korrektheitstests sind möglich.

UTF-8 kann die verschiedenen UCS-Codes auf einfache Weise repräsentieren:

– 1-Byte-Codes haben die Form `0xxx xxxx` und ermöglichen die Verwendung von 7 (mit x gekennzeichneten) Bits und damit die Codierung von allen 7-Bit ASCII Codes.
– 2-Byte-Codes haben die Form `110x xxxx 10xx xxxx` und ermöglichen die Codierung aller 11-Bit UCS-2 Codes.
– 3-Byte-Codes haben die Form `1110 xxxx 10xx xxxx 10xx xxxx`. Mit den 16 noch verfügbaren Bits können alle 16-Bit UCS-2 Codes dargestellt werden.
– 4-Byte-Codes der Form `1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx` ermöglichen die Verwendung von 21 Bits zur Codierung aller 21-Bit UCS-4 Codes.
– 5-Byte-Codes können alle 26-Bit UCS-4 Codes darstellen. Sie haben die Form:
  `1111 10xx 10xx xxxx 10xx xxxx 10xx xxxx 10xx xxxx`
– 6-Byte-Codes ermöglichen die Codierung des kompletten 31-Bit UCS-4 Codes:
  `1111 110x 10xx xxxx 10xx xxxx 10xx xxxx 10xx xxxx 10xx xxxx`

UTF-8 codierte Dateien sind also kompatibel zur 7-Bit ASCII Vergangenheit und verlängern den Umfang von Dateien aus dem amerikanischen und europäischen Bereich gar nicht oder nur unwesentlich. Diese Eigenschaften haben dazu geführt, dass diese Codierungsmethode der de facto Standard bei der Verwendung von Unicode geworden ist. Bei den Webseiten des Internets wird UTF-8 immer häufiger verwendet – alternativ dazu können in HTML-Dateien Sonderzeichen, also z.B. Umlaute wie „ä“ durch so genannte Entities als „ä“ umschrieben werden. Neben UTF-8 gibt es noch andere Transformations-Codierungen wie UTF-2, UTF-7 und UTF-16, die allerdings nur geringe Bedeutung erlangt haben.

Java erlaubte als erste der weit verbreiteten Sprachen die Verwendung beliebiger Unicode Zeichen für Bezeichnernamen und in Zeichenketten. Somit können in jedem Kulturkreis die vertrauten Namen für Variablen und Methoden vergeben werden. Allerdings sollte das Betriebssystem mit den verwendeten Zeichen auch umgehen können, insbesondere wenn Sonderzeichen in Dateinamen oder Klassennamen vorkommen.

1.3.6 Zeichenketten

Zur Codierung eines fortlaufenden Textes fügt man einfach die Codes der einzelnen Zeichen aneinander. Eine Folge von Textzeichen heißt auch *Zeichenkette* (engl. string). Der Text „`Hallo Welt`“ wird also durch die Zeichenfolge

```
H, a, l, l, o, , W, e, l, t
```

repräsentiert. Jedes dieser Zeichen, einschließlich des Leerzeichens “␣“, ersetzen wir durch seine Nummer in der ASCII-Tabelle und erhalten:

```
072 097 108 108 111 032 087 101 108 116
```

Alternativ können wir die ASCII-Nummern auch hexadezimal schreiben, also:

```
48 61 6C 6C 6F 20 57 65 6C 74
```

Daraus können wir unmittelbar auch die Repräsentation durch eine Bitfolge entnehmen:

```
01001000 01100001 01101100 01101100 01101111
00100000 01010111 01100101 01101100 01110100.
```

Viele Programmiersprachen markieren das Ende einer Zeichenkette mit dem ASCII-Zeichen NUL = 00h. Man spricht dann von nullterminierten Strings.

Es soll hier nicht unerwähnt bleiben, dass im Bereich der Großrechner noch eine andere als die besprochene ASCII-Codierung in Gebrauch ist. Es handelt sich um den so genannten EBCDI-Code (*extended binary coded decimal interchange*). Mit dem Rückgang der Großrechner verliert diese Codierung aber zunehmend an Bedeutung.

1.3.7 Logische Werte und logische Verknüpfungen

Logische Werte sind die *Wahrheitswerte Wahr* und *Falsch* (engl. *true* und *false*). Je nach Kontext kann man sie durch die Buchstaben T und F oder durch 1 und 0 abkürzen. Auf diesen logischen Werten sind die *booleschen Verknüpfungen* NOT (Negation oder Komplement), AND (Konjunktion), OR (Disjunktion) und XOR (exklusives OR) durch die folgenden Verknüpfungstafeln festgelegt.

Die AND-Verknüpfung zweier Argumente ist also nur dann T, wenn beide Argumente T sind. Die OR-Verknüpfung zweier Argumente ist nur dann F, wenn beide Argumente F sind. Die XOR-Verknüpfung zweier Argumente ist genau dann T, wenn beide Argumente verschieden sind. So gilt z.B. `T XOR F = T`, denn in der XOR-Tabelle findet sich in der Zeile neben T und der Spalte unter F der Eintrag T.

NOT	
F	T
T	F

AND	F	T
F	F	F
T	F	T

OR	F	T
F	F	T
T	T	T

XOR	F	T
F	F	T
T	T	F

Abb. 1.3. Logische Verknüpfungen

Da es nur zwei Wahrheitswerte gibt, könnte man diese durch die beiden möglichen Werte eines Bits darstellen, z.B. durch `F = 0` und `T = 1`. Da aber ein Byte die kleinste Einheit ist, mit der ein Computer operiert, spendiert man meist ein ganzes Byte für einen Wahrheitswert. Eine gängige Codierung ist `F = 0000 0000` und `T = 1111 1111`.

Man kann beliebige Bitketten auch als Folgen logischer Werte interpretieren. Die logischen Verknüpfungen sind für diese Bitketten als bitweise Verknüpfung der entsprechenden Kettenelemente definiert. So berechnet man z.B. das *bitweise Komplement*

`NOT` 01110110 = $\bar{0}\bar{1}\bar{1}\bar{1}\bar{0}\bar{1}\bar{1}\bar{0}$ = 10001001

oder die bitweise Konjunktion

01110110 `AND` 11101011 = 01100010.

1.3.8 Programme

Programme, also Folgen von Anweisungen, die einen Rechner veranlassen, bestimmte Dinge zu tun, sind im Hauptspeicher des Rechners oder auf einem externen Medium gespeichert. Auch für die Instruktionen eines Programms benutzt man eine vorher festgelegte Codierung, die jedem Befehl eine bestimmte Bitfolge zuordnet. Wenn Programme erstellt werden, sind sie noch als Text formuliert, erst ein Compiler übersetzt diesen Text in eine Reihe von Befehlen, die der Rechner versteht, die so genannten Maschinenbefehle. So repräsentiert z.B. die Bytefolge „03 D8“ den PC-Maschinenbefehl „ADD BX, AX“, welcher den Inhalt des Registers AX zu dem Inhalt von BX addiert.

1.3.9 Bilder und Musikstücke

Auch Bilder und Musikstücke können als Daten in einem Computer verarbeitet und gespeichert werden. Ein Bild wird dazu in eine Folge von *Rasterpunkten* aufgelöst. Jeden dieser Rasterpunkte kann man durch ein Bit, ein Byte oder mehrere Bytes codieren, je nachdem, ob das Bild ein- oder mehrfarbig ist. Eine Folge solcher Codes für Rasterpunkte repräsentiert dann ein Bild. Eine Konvention wie ein Bild (allgemei-

ner auch ein Film oder ein Musikstück) in eine Folge von Bytes übersetzt und in einer Datei gespeichert wird, heißt ein Format. Es gibt viele Standardformate für die Speicherung von Bildern, darunter solche, die jeden einzelnen Bildpunkt mit gleichem Aufwand speichern (dies nennt man eine Bitmap) bis zu anderen, die das gespeicherte Bild noch komprimieren. Dazu gehören das gif- und das jpeg-Format. Offiziell heißt dieses, von der *joint photographic expert group* (abgekürzt: *jpeg*) definierte Format jfif, doch die Bezeichnung jpeg ist gebräuchlicher. Das letztere Verfahren erreicht sehr hohe Kompressionsraten auf Kosten der Detailgenauigkeit – es ist verlustbehaftet, man kann die Pixel des Originalbildes i.A. also nicht wieder exakt zurückgewinnen.

Bei Musikstücken muss das analoge Tonsignal zunächst digital codiert werden. Man kann sich das so vorstellen, dass die vorliegende Schwingung viele tausend mal pro Sekunde abgetastet wird. Die gemessene Amplitude wird jeweils als Binärzahl notiert und in der Datei gespeichert. Mit der *mp3-Codierung*, die gezielt akustische Informationen unterdrückt, welche die meisten Menschen ohnehin nicht wahrnehmen, können die ursprünglichen wav-Dateien auf ungefähr ein Zehntel ihrer ursprünglichen Größe in mp3-Dateien komprimiert werden.

1.4 Zahlendarstellungen

Wie alle bisher diskutierten Informationen werden auch Zahlen durch Bitfolgen dargestellt. Wenn eine Zahl wie z.B. „4711“ mitten in einem Text vorkommt, etwa in einem Werbeslogan, so wird sie, wie der Rest des Textes, als Folge ihrer ASCII-Ziffernzeichen gespeichert, d.h. als Folge der ASCII Zeichen für „4“, „7“, „1“ und „1“. Dies wären hier die ASCII-Codes mit den Nummern 34h, 37h, 31h, 31h. Eine solche Darstellung ist aber für Zahlen, mit denen man arithmetische Operationen durchführen möchte, unpraktisch und verschwendet unnötig Platz.

Man kann Zahlen viel effizienter durch eine umkehrbar eindeutige (eins-zu-eins) Zuordnung zwischen Bitfolgen und Zahlen kodieren. Wenn wir nur Bitfolgen einer festen Länge N betrachten, können wir damit 2^N viele Zahlen darstellen. Gebräuchlich sind $N = 8, 16, 32$ oder 64. Man repräsentiert durch die Bitfolgen der Länge N dann

- die natürlichen Zahlen von 0 bis $2^N - 1$, oder
- die ganzen Zahlen zwischen -2^{N-1} und $2^{N-1}-1$, oder
- ein Intervall der reellen Zahlen mit begrenzter Genauigkeit.

1.4.1 Binärdarstellung

Will man nur positive ganze Zahlen (natürliche Zahlen) darstellen, so kann man mit N Bits den Bereich der Zahlen von 0 bis 2^{N-1}, das sind 2^N viele, überdecken. Die Zuordnung der Bitfolgen zu den natürlichen Zahlen geschieht so, dass die Bitfolge der *Binärdarstellung* der darzustellenden Zahl entspricht. Die natürlichen Zahlen nennt man in

der Informatik auch *vorzeichenlose Zahlen*, und die Binärdarstellung heißt demzufolge auch vorzeichenlose Darstellung.

Um die Idee der Binärdarstellung zu verstehen, führen wir uns noch einmal das gebräuchliche Dezimalsystem (Zehnersystem) vor Augen. Die einzelnen Ziffern einer Dezimalzahl stellen bekanntlich die Koeffizienten von Zehnerpotenzen dar, wie beispielsweise in

$$\begin{aligned} 4711 &= 4 \times 1000 + 7 \times 100 + 1 \times 10 + 1 \times 1 \\ &= 4 \times 10^3 + 7 \times 10^2 + 1 \times 10^1 + 1 \times 10^0 \end{aligned}$$

Für das Binärsystem (Zweiersystem) hat man anstelle der Ziffern 0 ... 9 nur die beiden Ziffern 0 und 1 zur Verfügung, daher stellen die einzelnen Ziffern einer Binärzahl die Koeffizienten der Potenzen von 2 dar. Die Bitfolge 1101 hat daher den Zahlenwert:

$$\begin{aligned} (1101)_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 13 \end{aligned}$$

Dies können wir durch die Gleichung $(1101)_2 = (13)_{10}$ ausdrücken, wobei der Index angibt, in welchem Zahlensystem die Ziffernfolge interpretiert werden soll. Der Index entfällt, wenn das Zahlensystem aus dem Kontext klar ist.

Mit drei Binärziffern können wir die Zahlenwerte von 0 bis 7 darstellen:

```
000 = 0, 001 = 1, 010 = 2, 011 = 3,
100 = 4, 101 = 5, 110 = 6, 111 = 7.
```

Mit 4 Bits können wir analog die 16 Zahlen von 0 bis 15 erfassen, mit 8 Bits die 256 Zahlen von 0 bis 255, mit 16 Bits die Zahlen von 0 bis 65535 und mit 32 Bits die Zahlen von 0 bis 4 294 967 295.

1.4.2 Das Oktalsystem und das Hexadezimalsystem

Neben dem Dezimalsystem und dem Binärsystem sind in der Informatik noch das *Oktalsystem* und das *Hexadezimalsystem* in Gebrauch. Das Oktalsystem stellt Zahlen zur Basis 8 dar. Es verwendet daher nur die Ziffern 0 ... 7. Bei einer mehrstelligen Zahl im Oktalsystem ist jede Ziffer d_i der Koeffizient der zugehörigen Potenz 8^i, also:

$$(d_n d_{n-1} \ldots d_0)_8 = d_n \times 8^n + d_{n-1} \times 8^{n-1} + \ldots + d_0 \times 8^0 \, .$$

Beispielsweise gilt: $(4711)_8 = 4 \times 8^3 + 7 \times 8^2 + 1 \times 8^1 + 1 \times 8^0 = (2505)_{10}$.

Ähnlich verhält es sich mit dem Hexadezimalsystem, dem System zur Basis 16. Die 16 Hexziffern 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F drücken den Koeffizienten derjenigen

Potenz von 16 aus, der ihrer Position entspricht. Beispielsweise stellt die Hexadezimalzahl 2C73 die Dezimalzahl 11379 dar, es gilt nämlich:

$$(2C73)_{16} = 2 \times 16^3 + 12 \times 16^2 + 7 \times 16^1 + 3 \times 16^0 = (11379)_{10}\,.$$

Prinzipiell könnte man jede beliebige positive Zahl, sogar 7 oder 13, als Basiszahl nehmen. Ein Vorteil des Oktal- und des Hexadezimalsystems ist, dass man zwischen dem Binärsystem und dem Oktal- bzw. dem Hexadezimalsystem ganz einfach umrechnen kann. Wenn wir von einer Binärzahl ausgehen, brauchen wir lediglich, von rechts beginnend, die Ziffern zu Vierergruppen zusammenzufassen und jeder dieser Vierergruppen die entsprechende Hexziffer zuzuordnen. Die resultierende Folge von Hexziffern ist dann die Hexadezimaldarstellung der Binärzahl. So entsteht z.B. aus der Binärzahl 10110001110011 die Hexadezimalzahl 2C73, denn

$$(10110001110011)_2 = (10\ 1100\ 0111\ 0011)_2 = (2C73)_{16}.$$

Gruppieren wir jeweils drei Ziffern einer Binärzahl und ordnen jeder Dreiergruppe die entsprechende Oktalziffer zu, so erhalten wir die Oktaldarstellung. Mit dem Beispiel von eben:

$$(10110001110011)_2 = (010\ 110\ 001\ 110\ 011)_2 = (26163)_8\,.$$

Warum dies so einfach funktioniert, lässt sich leicht plausibel machen. Sei dazu eine Binärzahl $b_7b_6b_5b_4b_3b_2b_1b_0$ gegeben. Wir füllen zunächst links mit Nullen auf, bis die Anzahl der Stellen ein Vielfaches von 3 ist. Von rechts nach links fassen wir nun jeweils drei Summanden zu einer Dreiergruppe zusammen und klammern Potenzen von $2^3 = 8$ aus. Danach summiert das Überbleibsel jeder Dreiergruppe zu einer Zahl zwischen 0 und 7, die wir als Oktalziffer repräsentieren können. Wir zeigen das am Beispiel 9-stelliger Binärzahlen:

$$\begin{aligned}
(b_8b_7b_6\ldots b_2b_1b_0)_2 &= b_8 \times 2^8 + b_7 \times 2^7 + b_6 \times 2^6 + \ldots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 \\
&= (b_8{\times}2^2 + b_7{\times}2^1 + b_6{\times}2^0){\times}2^6 + \ldots + (b_2{\times}2^2 + b_1{\times}2^1 + b_0{\times}2^0){\times}2^0 \\
&= (\quad z_2 \quad){\times}8^2 + \ldots + (\quad z_0 \quad){\times}8^0 \\
&= (z_2z_1z_0)_8\,.
\end{aligned}$$

Genauso zeigt sich, dass die beschriebene Umwandlung zwischen Binärzahlen und Hexadezimalzahlen korrekt ist. Dabei muss man jeweils 4 Ziffern zusammenfassen und wachsende Potenzen von 16 ausklammern. Die Umwandlung, etwa vom Oktal- in das Hexadezimalsystem oder umgekehrt, erfolgt am einfachsten über den Umweg des Binärsystems. Als Beispiel:

$$(4711)_8 = (100\ 111\ 001\ 001)_2 = (1001\ 1100\ 1001)_2 = (9C9)_{16}.$$

1.4.3 Umwandlung in das Dezimalsystem

Die Umwandlung einer Binär-, Oktal- oder Hexzahl in das Dezimalsystem ist einfach, allerdings treten viele Multiplikationen auf, wenn man alle Potenzen der Basiszahl berechnet und mit den Ziffern multipliziert. Eine einfachere Methode beruht auf der Idee, aus den höherwertigen Stellen immer wieder die Basiszahl d auszuklammern. Wir zeigen dies nur am Beispiel der oben berechneten Oktalzahl:

$$\begin{aligned}(4711)_8 &= 4\times 8^3 + 7\times 8^2 + 1\times 8^1 + 1\\ &= (4\times 8^2 + 7\times 8^1 + 1)\times 8 + 1\\ &= ((4\times 8 + 7)\times 8 + 1)\times 8 + 1\end{aligned}$$

Einen solchen Ausdruck können wir von innen nach außen ausrechnen. Auf dieser Erkenntnis basiert die folgende Berechnungsmethode, die als *Horner-Schema* bekannt ist. Er erlaubt uns eine einfache Berechnung mit einem Taschenrechner. Die linke Spalte zeigt die Tastenfolge, die rechte die jeweilige Anzeige:

Tastenfolge :		*Anzeige* :
4	(=)	4
×8 + 7	(=)	39
×8 + 1	(=)	313
×8 + 1	(=)	2505

Das funktioniert für alle Zahlensysteme. Wir demonstrieren die Methode, indem wir erneut $(2C73)_{16}$ berechnen. Normale Taschenrechner erwarten natürlich die Eingabe „12“ statt „C“:

2	(=)	2
×16 + 12	(=)	44
×16 + 7	(=)	711
×16 + 3	(=)	11379

1.4.4 Umwandlung in das Binär-, Oktal- oder Hexadezimalsystem

Der etwas schwierigeren Umwandlung von Dezimalzahlen in andere Zahlensysteme, wollen wir eine, auch für spätere Kapitel wichtige zahlentheoretische Beobachtung vorwegschicken:

Wenn man eine natürliche Zahl z durch eine andere natürliche Zahl $d \neq 0$ exakt teilt, erhält man einen Quotienten q und einen Rest r. Beispielsweise gilt $39/8 = 4\ Rest\ 7$. Die Probe ergibt: $39 = 4\times 8 + 7$. Der Rest $r$ist immer kleiner als der Divisor d.

In der Informatik bezeichnet man die Operation des Dividierens ohne Rest mit *div* und die Operation, die den Divisionsrest ermittelt, mit *mod*. Im Beispiel haben wir also $39\ div\ 8 = 4$ und $39\ mod\ 8 = 7$. Der Rest ist immer kleiner als der Divisor, also und die Probe muss die ursprüngliche Zahl ergeben, also:

$$z = (z\ div\ d) \times d + (z\ mod\ d) \text{ wobei } 0 \leq (z\ mod\ d) < d.$$

Für eine Zahl z im Dezimalsystem ist $(z\ mod\ 10)$ die letzte Ziffer und $(z\ div\ 10)$ erhält man durch Streichen der letzten Ziffer:

$$4711\ mod\ 10 = 1 \text{ und } 4711\ div\ 10 = 471.$$

Dies gilt analog für alle Zahlensysteme. Für eine beliebige Binärzahl z rechnen wir es nach:

$$\begin{aligned} z &= (b_n b_{n-1} \ldots b_1 b_0)_2 \\ &= b_n \times 2^n + b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0 \\ &= (b_n \times 2^{n-1} + b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^0) \times 2 + b_0 \times 2^0 \\ &= (\qquad\qquad b_n b_{n-1} \ldots b_1 \qquad\qquad)_2 \times 2 + b_0 \end{aligned}$$

Diese Darstellung zeigt, dass für eine beliebige Zahl $z = (b_n b_{n-1} \ldots b_1 b_0)_2$ gilt

$$\begin{aligned} z\ div\ 2 &= (b_n b_{n-1} \ldots b_1)_2 \\ z\ mod\ 2 &= b_0. \end{aligned}$$

Suchen wir die Binärdarstellung einer Zahl z, so erhalten wir also die letzte Ziffer b_0 als Rest $z\ mod\ 2$. Die Folge der ersten Ziffern $b_n b_{n-1} \ldots b_1$ ist die Binärdarstellung von $z\ div\ 2$. Als nächstes erhalten wir b_1 als letzte Ziffer von $z\ div\ 2$, etc. So finden wir also die Binärdarstellung einer beliebigen natürlichen Zahl z. Fortgesetztes Teilen durch 2 liefert nacheinander die Ziffern der Darstellung von z im Zweiersystem als Reste. Die Binärziffern entstehen dabei von rechts nach links. Wir zeigen dies beispielhaft für die Zahl $2017_{10} = 11111100001_2$:

Alles, was über Binärzahlen gesagt wurde, gilt sinngemäß auch in anderen Zahlensystemen. Verwandeln wir z.B. die dezimale Zahl 2017 in das Oktalsystem, so liefert fortgesetztes Teilen durch die Basiszahl 8 die Reste 1, 4, 7, 3

$$\begin{aligned} 2017 &= 8 \times 252 + 1 \quad \rightarrow \text{Ziffer: } 1 \\ 252 &= 8 \times 31 + 4 \quad \rightarrow \text{Ziffer: } 4 \\ 31 &= 8 \times 3 + 7 \quad \rightarrow \text{Ziffer: } 7 \\ 3 &= 8 \times 0 + 3 \quad \rightarrow \text{Ziffer: } 3 \end{aligned}$$

Wir lesen an den Resten die Oktaldarstellung $(3741)_8$ ab. Ganz entsprechend erhalten wir die Hexadezimaldarstellung durch fortgesetztes Teilen durch 16. Die mögli-

Tab. 1.1. Binärzahlentwicklung von 2017

z	*z div* 2	*z mod* 2
2017	1008	1
1008	504	0
504	252	0
252	126	0
126	63	0
63	31	1
31	15	1
15	7	1
7	3	1
3	1	1
1	0	1

chen Reste 0 bis 15 stellen wir jetzt durch die Hex-Ziffern 0 ... 9, A ... F dar und erhalten

$$\begin{aligned} 2017 &= 16{\times}126 + 1 \;\rightarrow \text{Ziffer: } 1 \\ 126 &= 16{\times}7 + 14 \;\rightarrow \text{Ziffer: } E \\ 7 &= 16{\times}0 + 7 \;\rightarrow \text{Ziffer: } 7 \end{aligned}$$

also die Hex-Darstellung 7E1. Zur Sicherheit überprüfen wir, ob beide Male tatsächlich der gleiche Zahlenwert erhalten wurde, indem wir vom Oktalsystem in das Binär- und von dort ins Hexadezimalsystem umrechnen:

$(3741)_8 = (011\ 111\ 100\ 001)_2 = (0111\ 1110\ 0001)_2 = (7E1)_{16}$

1.4.5 Arithmetische Operationen

Zwei aus mehreren Ziffern bestehende Binärzahlen werden addiert, wie man es analog auch von der Addition von Dezimalzahlen gewohnt ist. Ein an einer Ziffernposition entstehender *Übertrag* (engl.: *carry*) wird zur nächsthöheren Ziffernposition addiert. Ein Übertrag entsteht immer, wenn bei der Addition zweier Ziffern ein Wert entsteht, der größer oder gleich dem Basiswert ist. Bei Binärziffern ist dies bereits der Fall, wenn wir 1+1 addieren. Es entsteht ein Übertrag von 1 in die nächste Ziffernposition.

Abb.1.4 zeigt ein Beispiel für die schriftliche Addition zweier natürlicher Zahlen in den besprochenen Zahlensystemen, dem Binär-, Oktal- und Hexadezimalsystem. Es werden jedes Mal die gleichen Zahlenwerte addiert und die Probe zeigt, dass das Ergebnis immer das gleiche ist.

Auch Subtraktion, Multiplikation und Division verlaufen in anderen Zahlensystemen analog wie im Dezimalsystem. Wir wollen dies hier nicht weiter vertiefen, sondern stattdessen die Frage ansprechen, was geschieht, wenn durch eine arithmetische Ope-

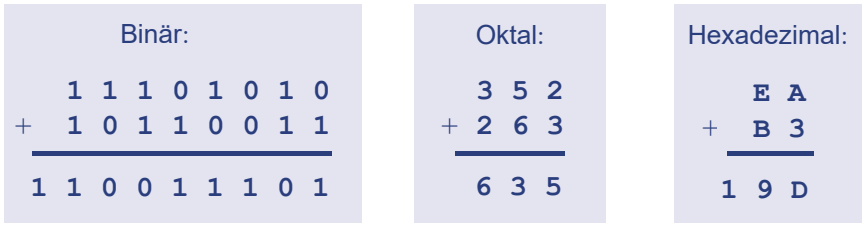


Abb. 1.4. Addition in verschiedenen Zahlensystemen

ration ein Zahlenbereich überschritten wird. Bei der Addition ist dies z.B. daran zu erkennen, dass in der höchsten Ziffernposition ein Übertrag auftritt. Hätte man im oberen Beispiel Zahlen durch Bytes, repräsentiert, so würde die erste Stelle des Ergebnisses der Addition nicht mehr in das Byte passen. Der vorderste Übertrag würde also unter den Tisch fallen. Der wahre Wert (110011101)2 = 413 würde von dem gefundenen Wert $(10011101)_2 = 157$ um $2^8 = 256$ abweichen. Die Differenz entspricht gerade dem Stellenwert des verschwundenen Übertrags.

Wenn ein Ergebnis einer arithmetischen Operation nicht in das gewählte Zahlenformat passt, rechnet der Prozessor klaglos mit dem abgeschnittenen falschen Ergebnis weiter. Der Programmierer ist selber dafür verantwortlich, dass das gewählte Zahlenformat so groß ist, dass kein Überlauf entstehen kann. Alternativ kann er auch direkt nach einer Operation prüfen, ob ein Überlauf stattgefunden hat. Zu diesem Zweck signalisiert der Prozessor immer in einem Status Bit, dem sogenannten Carry Flag, ob die Operation korrekt verlaufen ist. Bei der obigen Operation hätte der Prozessor das sogenannte *Carry Flag C* gesetzt. Bei einer anschließenden korrekt verlaufenen arithmetischen Operation wird es wieder zurückgesetzt.

1.4.6 Darstellung ganzer Zahlen

Als *ganze Zahlen* bezeichnet man die natürlichen Zahlen unter Hinzunahme der *negativen Zahlen*, also

```
... -9 -8 -7 -6 -5 -4 -3 -2 -1 0 +1 +2 +3 +4 +5 +6 +7 ...
```

Für die Kenntnis einer ganzen Zahl ist also nicht nur der absolute Zahlenwert nötig, sondern auch noch das Vorzeichen, „+“ oder „-“. Für diese zwei Möglichkeiten benötigen wir ein weiteres Bit an Information. Zunächst bietet sich eine Darstellung an, in der das erste Bit das Vorzeichen repräsentiert (0 für „+“) und (1 für „-“) und der Rest den Absolutwert. Diese Darstellung nennt man die Vorzeichendarstellung. Stehen z.B. 4 Bit für die Zahldarstellung zur Verfügung und wird ein Bit für das Vorzeichen verwendet, so bleiben noch 3 Bit für den Absolutwert. Mit 4 Bit kann man also die Zahlen von –7 bis +7 darstellen. Wir erhalten:

```
0000 = + 0   0100 = + 4   1000 = - 0   1100 = - 4
0001 = + 1   0101 = + 5   1001 = - 1   1101 = - 5
0010 = + 2   0110 = + 6   1010 = - 2   1110 = - 6
0011 = + 3   0111 = + 7   1011 = - 3   1111 = - 7
```

Bei näherem Hinsehen hat diese Darstellung aber eine Reihe von Nachteilen. Erstens erkennt man, dass die Zahl 0 durch zwei verschiedene Bitfolgen dargestellt ist, durch 0000 und durch 1000, also +0 und −0. Zweitens ist auch das Rechnen kompliziert geworden. Um zwei Zahlen zu addieren, kann man nicht mehr die beiden Summanden übereinander schreiben und schriftlich addieren.

$$\begin{array}{rrcl} & -2 & = & 1010 \\ + & +5 & = & 0101 \\ \hline & +3 & \neq & 1111 \end{array}$$

Es ließe sich durchaus eine Methode angeben, mit der man solche Bitfolgen korrekt addieren kann. Es gibt aber eine bessere Darstellung von ganzen Zahlen, die alle genannten Probleme vermeidet und die wir im nächsten Abschnitt besprechen wollen.

1.4.7 Die Zweierkomplementdarstellung

Die *Zweierkomplementdarstellung* ist die gebräuchliche interne Repräsentation ganzer positiver und negativer Zahlen. Sie kommt auf sehr einfache Weise zu Stande. Wir erläutern sie zunächst für den Fall $N = 4$.

Mit 4 Bits kann man einen Bereich von $2^4 = 16$ ganzen Zahlen abdecken. Den Bereich kann man frei wählen, also z.B. die 16 Zahlen von −8 bis +7. Man zählt nun von 0 beginnend aufwärts, bis man die obere Grenze +7 erreicht, anschließend fährt man an der unteren Grenze −8 fort und zählt aufwärts, bis man die Zahl −1 erreicht hat.

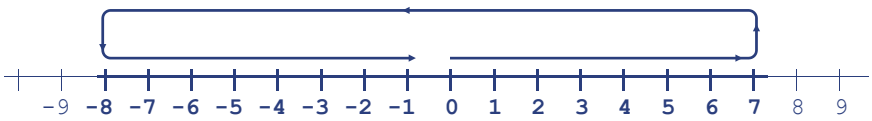


Abb. 1.5. Zweierkomplementdarstellung

Auf diese Weise erhält man folgende Zuordnung von Bitfolgen zu ganzen Zahlen:

```
0000 = +0   0100 = +4   1000 = -8   1100 = -4
0001 = +1   0101 = +5   1001 = -7   1101 = -3
```

```
0010 = +2   0110 = +6   1010 = −6   1110 = −2
0011 = +3   0111 = +7   1011 = −5   1111 = −1
```

Hier erkennt man, wieso der Bereich von −8 bis +7 gewählt wurde und nicht etwa bis +8. Bei dem bei 0 beginnenden Hochzählen wird bei der achten Bitfolge zum ersten Mal das erste Bit zu 1. Springt man also ab der achten Bitfolge in den negativen Bereich, so hat man die folgende Eigenschaft:

Bei den Zweierkomplementzahlen stellt das erste Bit das Vorzeichen dar.

Den so einer Bitfolge $b_n \ldots b_0$ zugeordneten Zahlenwert wollen wir mit $(b_n \ldots b_0)_z$ bezeichnen, also z.B. $(0111)_z = +7$ und $(1000)_z = -8$. Verfolgen wir die Bitfolgen und die zugeordneten Zahlenwerte, so sehen wir, dass diese zunächst anwachsen bis 2^{n-1} erreicht ist, alsdann fallen sie plötzlich um 2^n und wachsen dann wieder an bis zu $(1 \ldots 1)_z = -1$.

Für die positiven Zahlen von 0 bis 2^{n-1} stimmen daher Zweierkomplementdarstellung und Binärdarstellung überein, d.h.: $(0b_{n-1} \ldots b_0)_z = (0b_{n-1} \ldots b_0)_2$. Für die negativen Werte gilt hingegen$(1b_{n-1} \ldots b_0)_z = -2^n + (0b_{n-1} \ldots b_0)_2$. Das erste Bit der Zweierkomplementdarstellung steht also für -2^n, die übrigen Bits haben ihre normale Bedeutung daher gilt:

$$\begin{aligned}(b_n b_{n-1} \ldots b_0)_z &= b_n \times (-2^n) + b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \\ &= b_n \times (-2^n) + (b_{n-1} \ldots b_0)_2\end{aligned}$$

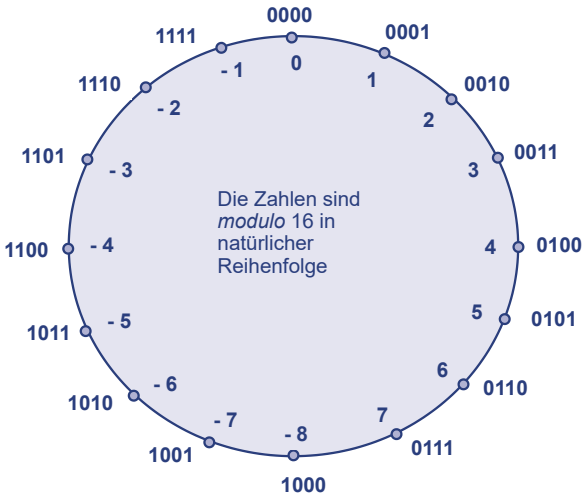


Abb. 1.6. Zweierkomplementdarstellung in einem Zahlenkreis

Aufgrund der bekannten geometrischen Formel

$$1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$$

folgt sofort, dass die nur aus 1-en bestehende Zweierkomplementzahl stets den Wert –1 repräsentiert, denn

$$\begin{aligned}(111\ldots11)_z &= (-2^n) + 2^{n-1} + \ldots + 2^1 + 2^0 \\ &= -2^n + (2^n - 1) \\ &= -1\end{aligned}$$

Diese Eigenschaft gibt den Zweierkomplementzahlen ihren Namen: Addiert man eine beliebige Ziffernfolge $b_n b_{n-1} \ldots b_1 b_0$ zu ihrem bitweisen Komplement (siehe S. 18), so erhält man stets 11...11, also die Repräsentation von –1. Mit dieser Beobachtung können wir zu einer beliebigen Zweikomplementzahl ihr Negatives berechnen:

Bilde das bitweise Komplement und addiere 1.

Beispielsweise erhält man die Zweierkomplementdarstellung von –6, indem man zuerst die Binärdarstellung von +6, also 0110 bildet, davon das bitweise Komplement 1001, und zum Schluss 1 addiert. Diese Operation nennt man *Zweierkomplement*.

$$-6 = (\bar{0}\bar{1}\bar{1}\bar{0})_z + 1 = (1001)_z + 1 = (1010)_z$$

Erneutes Bilden des Zweierkomplements liefert die Ausgangszahl zurück:

$$(\bar{1}\bar{0}\bar{1}\bar{0})_z + 1 = (0101)_z + 1 = (0110)_z = 6$$

Da das Negative einer Zahl so einfach zu bilden ist, führt man die Subtraktion auf die Negation mit anschließender Addition zurück.

1.4.8 Rechnen mit Zweierkomplementzahlen

Die Addition der Zweierkomplementzahlen $a = (a_n a_{n-1} \ldots a_0)_z$ und $b = (b_n b_{n-1} \ldots b_0)_z$ funktioniert wie die Addition von vorzeichenlosen Zahlen. Für die niedrigen Bitpositionen, $n-1$ bis 0 ist das klar. Sie haben in beiden Repräsentationen dieselbe Interpretation. Ob das Ergebnis gültig ist, kann nur von den höchsten Bits a_n und b_n und einem eventuellen Übertrag c von dem $(n-1)$-ten in das n-te Bit abhängen. Da dieser den Wert $c \times 2^n$ repräsentiert, a_n und b_n dagegen die Werte $a_n \times (-2^n)$ bzw. $b_n \times (-2^n)$, ist das Ergebnis genau dann gültig, wenn $(a_n + b_n - c) \in \{0, 1\}$ gilt. Ist dies nicht der Fall, so setzt der Prozessor das *Overflow-Flag*.

Was den Prozess der Addition angeht, spielt es keine Rolle, ob eine Bitfolge eine Binärzahl darstellt oder eine Zweierkomplementzahl. Der Prozessor benötigt also kein gesondertes Addierwerk für Zweierkomplementzahlen. Ob das Ergebnis jedoch gültig

ist, das hängt sehr wohl davon ab, ob die addierten Bitfolgen als Binärzahlen oder als Zweierkomplementzahlen verstanden werden sollen. Als Summe von Binärzahlen ist die Addition genau dann gültig, wenn kein Übertrag C aus der n-ten Bitposition entsteht, wenn also $(a_n + b_n + c) \in \{0, 1\}$ gilt, und als Summe von Zweierkomplementzahlen ist es gültig, falls $(a_n + b_n - c) \in \{0, 1\}$ ist. Das ist aber gleichbedeutend mit $(a_n + b_n + c) \in \{2c, 2c + 1\}$, d.h. $(a_n + b_n + c)\ div\ 2 = c$, also $C = c$.

Im Beispiel der Abbildung 1.4 werden zwei 8-Bit Binärzahlen addiert. Als Addition vorzeichenbehafteter Zahlen ist das Ergebnis nicht gültig, da es einen Übertrag C aus der höchsten Stelle gab. Als Addition von 8-Bit Zweierkomplementzahlen war das Ergebnis korrekt, es gab einen Übertrag c in die höchste Bitposition und einen Übertrag C aus der höchsten Bitposition, so dass c = C ist, das Overflow Bit also nicht gesetzt wird. In der Tat wurden die Zahlen –22 und –77 addiert mit dem korrekten Ergebnis –99.

1.4.9 Formatwechsel

Wechseln wir jetzt den Zahlenbereich und betrachten 8-Bit große Zweierkomplementzahlen so erhalten wir die Zuordnung

```
0000 0000 =   0   1000 0000 = - 128
0000 0001 = +1    1000 0001 = - 127
... ...
0000 1000 = +8    1000 1000 = - 120
0000 1001 = +9    1000 1001 = - 119
... ...
0111 1110 = +126   1111 1110 = -2
0111 1111 = +127   1111 1111 = -1
```

Wandelt man also Zweierkomplementzahlen in ein größeres Format um, also z.B. von 4-Bit nach 8-Bit, so muss man bei positiven Zahlen vorne mit Nullen auffüllen, bei negativen aber mit Einsen. Beispielsweise ist $-7 = (1001)_z$ als 4-Bit Zahl, jedoch $(11111001)_z$ als 8-Bit-Zahl und $(1111111111111001)_z$ als 16-Bit Zahl. In beiden Fällen entsteht durch Addition von 6 die Zahl

$$-1 = (1111)_z = (1111\ 1111)_z = (1111\ 1111\ 1111\ 1111)_z$$

Es gibt noch eine einfachere Berechnung der Zweierkomplementdarstellung einer negativen Zahl. Dazu erinnern wir uns an die Gleichung

$$z = (z\ div\ d) \times d + (z\ mod\ d)$$

mit $0 \leq (z\ mod\ d) < d$. Damit konnten wir bequem die Binärdarstellung einer positiven Zahl finden. Das funktioniert auch für negative Zahlen, wenn wir *div* und *mod* geeignet wählen.

Wir definieren auch auf negativen Zahlen $z\ div\ d$ als „*exakt teilen und abrunden*“, also

$$z\ div\ d := \lfloor z/d \rfloor .$$

Aus der Gleichung folgt dann $z\ mod\ d$ als Rest

$$z\ mod\ d := z - d * \lfloor z/d \rfloor .$$

Beispielsweise ist $-13 = (-7) \times 2 + 1$, also $-13\ div\ 2 = -7$ und $-13\ mod\ 2 = 1$. Im Falle $d = 2$ kann $z\ mod\ 2$ nur die Werte 0 oder 1 annehmen, und liefert, auch wenn z negativ ist, die letzte Ziffer der binären Darstellung von z. Die vorderen Ziffern werden analog aus $z\ div\ 2$ bestimmt. Wir zeigen dies am Beispiel $z = -13$.

Tab. 1.2. Binärentwicklung einer negativen Zahl

z	*z div* 2	*z mod* 2
-13	-7	1
-7	-4	1
-4	-2	0
-2	-1	0
-1	-1	1
-1	-1	1
...	...	...

Die Entwicklung einer negativen Zahl führt immer auf -1 und produziert dann nur noch Einsen. Somit hat also -13 die Binärdarstellung ...110011. So wie positive Zahlen vorne mit beliebig vielen Nullen aufgefüllt werden, beginnen negative Zahlen mit beliebig vielen Einsen und werden entsprechend dem gewählten Zahlenformat abgeschnitten. –13 lautet also als 1-Byte Zahl 1111 0011 und als 2-Byte Zahl 1111 1111 1111 0011.

Bedauerlicherweise gibt es in den gängigen Programmiersprachen ein großes Durcheinander was die Implementierung von *div* und *mod* für negative Zahlen angeht. *C* überlässt dies ganz dem Compilerbauer, *Ada* implementiert mehrere Varianten (*mod* und *rem*). Java besitzt die Operatoren / und %, die für positive z und d unserem div und mod entsprechen. Für negatives z kann aber $z\ mod\ d$ negativ werden und z/d ist exaktes Teilen und betragsmäßig abrunden. In Java gilt zwar immer noch

$$z = (z/d) \times d + (z\ \%\ d)$$

aber nur mit $0 \le |\, z\ \%\ d\,| < d$.

1.4.10 Standardformate ganzer Zahlen

Prinzipiell kann man beliebige Zahlenformate vereinbaren, in der Praxis werden fast ausschließlich Zahlenformate mit 8, 16, 32 oder 64 Bits eingesetzt. In den meisten Programmiersprachen gibt es vordefinierte ganzzahlige Datentypen mit unterschiedlichen Wertebereichen. Je größer das Format ist, desto größer ist der erfasste Zahlenbereich. Die folgende Tabelle zeigt Zahlenformate und ihre Namen, wie sie in der Programmiersprachen Java und in einigen anderen Sprachen vordefiniert sind. Durch die Wahl eines geeigneten Formats muss der Programmierer dafür sorgen, dass der Bereich nicht überschritten wird. Dennoch auftretende Überschreitungen führen häufig zu scheinbar gültigen Ergebnissen: So hat in Java die Addition 127+5 im Bereich byte das Ergebnis –124!

Tab. 1.3. Zahlenformate

Bereich	Format	Java Datentyp
-128 ... 127	8 Bit	byte
-32768 ... 32767	16 Bit	short
$-2^{31} \ldots 2^{31}-1$	32 Bit	int
$-2^{63} \ldots 2^{63}-1$	64 Bit	long

1.4.11 Gleitpunktzahlen: Reelle Zahlen

Für wissenschaftliche und technische Anwendung muss man in einem Rechner auch mit Kommazahlen (Mathematiker sagen: *reelle Zahlen*) wie z.B. 3.1415 oder 0.33 rechnen können. Statt des deutschen Kommas wird im Englischen (und daher in fast allen Programmiersprachen) der Dezimal*punkt* geschrieben. Wir werden das auch hier so halten.

Reelle Zahlen entstehen entweder, wenn man ganze Zahlen teilt (man nennt sie dann rational) oder als Grenzwert eines Limesprozesses. Rationale Zahlen sind zum Beispiel $3/4 = 0.75$ oder $1/3 = 0.33333333333333333333\ldots$

Bei *rationalen Zahlen* bricht die Folge der Nachkommastellen ab, oder sie wird periodisch: $123/12 = 10.25$ aber $123/13 = 9.461538461538461538461538461538\ldots$

Nicht rationale Zahlen heißen irrational. Dazu gehören u.a. $\sqrt{2} = 1.4142135\ldots$ oder die Kreiszahl $\pi = 3.141592653589793238\ldots$.

Schon die Babylonier wussten, wie man Quadratwurzeln immer genauer berechnen konnte und den Griechen war klar, dass man den Umfang eines Kreises durch den Umfang des regelmäßigen n-Ecks approximieren und so π beliebig genau bestimmen kann, indem man n wachsen lässt. Heute sind mehr als die ersten 12 Billionen Stellen von π bekannt.

In den letzten Beispielen würde eine exakte Darstellung unendlich viele Stellen erfordern, daher kann man von einem Rechner bestenfalls erwarten, sie näherungsweise darzustellen. Ein wie auch immer gestaltetes Zahlenformat für reelle Zahlen muss dazu einen großen Bereich von Zahlen abdecken. Astronomen benötigen sehr große Zahlen, Physiker und Chemiker sehr kleine. Ein- und dasselbe Zahlenformat muss also u.a. folgende Werte darstellen können:

$$\begin{aligned}
Sonnenmasse &= 1989100000000000000000000000000\ kg \\
1\ Lichtjahr &= 9460730472580.0\ km \\
abs.Nullpunkt &= -273.15\ ^\circ C \\
el.Feldkonst. &= 0.00000000000885418781762\ Fm^{-1} \\
Planck-Konst. &= 0.00000000000000000000000000000000001054571628\ Js
\end{aligned}$$

Solche Zahlen kann man in technisch-wissenschaftlicher Darstellung kompakter darstellen:

$$\begin{aligned}
Sonnenmasse &= 1.9891 \times 10^{30}\ kg \\
1\ Lichtjahr &= 9.46073047258 \times 10^{12}\ km \\
abs.Nullpunkt &= -273.15\ ^\circ C \\
elektr.Feldkonst. &= 8.85418781762 \times 10^{-12}\ Fm^{-1} \\
Planck-Konst. &= 1.054571628 \times 10^{-34}\ Js
\end{aligned}$$

Der Exponent von 10 ist eine positive oder negative Zahl und besagt anschaulich, wie weit nach rechts bzw. nach links das Komma zu schieben ist. Die Darstellung ist nicht eindeutig, wir könnten den absoluten Nullpunkt auf mehrere Weisen schreiben, z.B. als

$$-2731.5 \times 10^{-1} = -273.15 \times 10^{0} = -27.315 \times 10^{1} = -2.7315 \times 10^{2} = -0.27315 \times 10^{3}$$

Normieren wir die Darstellung, so dass die erste Ziffer ungleich 0 direkt vor dem Komma zu stehen kommt, so erhalten wir für -2731.5 die *normierte Darstellung* -2.7315×10^{2}.

Alle anderen Beispiele sind schon normiert. Die Folge der Ziffern ohne führende Nullen, im obigen Beispiel 27315 nennt man *Mantisse*, die Potenz der Basiszahl, im Beispiel 2, heißt *Exponent*. Für eine derart normierte Zahl müssen wir also nur 3 Bestimmungsstücke speichern:

$$(Vorzeichen, Exponent, Mantisse).$$

Wir benötigen eine Stelle für das Vorzeichen, den Rest des Speicherplatzes müssen wir zwischen Exponent und Mantisse aufteilen. An den obigen Beispielen sieht

man, dass der Exponent relativ klein gewählt werden kann. Die Länge der Mantisse entspricht der Messgenauigkeit der Zahl. Angenommen, wir reservieren zwei Ziffern für den Exponenten und sieben Ziffern für die Mantisse, so könnten wir die obigen physikalischen Konstanten in 10 Dezimalziffern speichern:

Tab. 1.4. Wissenschaftliche Darstellung der physik. Konstanten

Vorzeichen	Exponent	Mantisse
0	30	19891
0	12	9460730
1	2	27315
0	-12	8854187
0	-34	1054571

Selbstverständlich können nicht alle Zahlen exakt gespeichert werden, nicht einmal die Zahl 1/3, denn sie erhielte die Darstellung

$$(0,\ 0,\ 3333333).$$

Eine kleine Schwierigkeit verbirgt sich noch in dem für den Exponenten zusätzlich benötigten Vorzeichen. Wir umgehen das Problem aber einfach, indem wir zum Exponenten einen festen Verschiebewert (engl.: bias, gelegentlich auch deutsch: Exzess) addieren, der das Ergebnis in einen positiven Bereich schiebt. Im obigen Beispiel wäre z.B. 49 ein guter bias. Statt 30 würden wir also eine 79 im Exponenten speichern und statt -32 eine 17. Unser Format wäre also durch folgende Parameter gekennzeichnet:

Mantisse: 7 Stellen, **Exponent**: 2 Stellen, **bias**: 49.

1.4.12 Binäre Gleitkommazahlen

Nun übertragen wir das System auf die Binärdarstellung, mit der Computer rechnen. Jede reelle Zahl kann auch im Binärsystem als Kommazahl dargestellt werden. Sei also

$$b_n b_{n-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-m}$$

eine binäre Kommazahl. Sie repräsentiert den Zahlenwert

$$(b_n \ldots b_0 . b_{-1} b_{-2} \ldots b_{-m})_2 = b_n \times 2^n + \ldots + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \ldots + b_{-m} \times 2^{-m}$$

Beispielsweise gilt

$$(10.101)_2 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 2 + 1/2 + 1/8 = 2.625$$

Für die umgekehrte Umwandlung einer (positiven) Dezimalzahl in eine binäre Kommazahl zerlegen wir die Dezimalzahl in den ganzzahligen Anteil und den Teil nach dem Komma. Diese können wir getrennt nach binär umwandeln und wieder zusammensetzen. Wie wir ganze Zahlen in Binärzahlen umwandeln, wissen wir schon, wir konzentrieren uns also auf den Nachkommateil. Sei $0.d_1 \ldots d_k$ die Dezimalzahl und $0.b_1 \ldots b_n$ die gesuchte binäre Kommazahl mit

$$(0.d_1 \ldots d_k)_{10} = (0.b_1 \ldots b_n)_2$$

Multiplizieren wir beide Seiten mit 2, so wandert in der binären Kommazahl rechts das Komma um eins nach rechts, während links eine neue Dezimalzahl $e.e_1 \ldots e_k$ entsteht. Es muss dann gelten:

$$(e.e_1 \ldots e_k)_{10} = (b_1.b_2 \ldots b_n)_2$$

Weil die Werte vor und nach dem Komma jeweils übereinstimmen müssen, folgt weiter $b_1 = e$ und $(0.e_1 \ldots e_k)_{10} = (0.b_2 \ldots b_n)_2$.

Wir erhalten also die erste Binärziffer b_1, indem wir die Dezimalzahl mit 2 multiplizieren und von dem Ergebnis die Ziffer vor dem Komma nehmen. Aus dem Wert nach dem Komma berechnen wir genauso die restlichen Binärziffern. Wir demonstrieren dies am Beispiel der Dezimalzahl 45.6875. Für die Stellen vor dem Komma erhalten wir $(45)_{10} = (101101)_2$. Für die Nachkommastellen $0,6875$ rechnen wir:

$$\begin{aligned} 2 \times 0,6875 &= 1,375 && \rightarrow \text{Ziffer: } 1 \\ 2 \times 0,375 &= 0,75 && \rightarrow \text{Ziffer: } 0 \\ 2 \times 0,75 &= 1,5 && \rightarrow \text{Ziffer: } 1 \\ 2 \times 0,5 &= 1 && \rightarrow \text{Ziffer: } 1 \end{aligned}$$

Es folgt:

$$45.6875 = (101101.1011)_2$$

Für einige der oben betrachteten Dezimalzahlen erhalten wir so:

$$\begin{aligned} 1\ Lichtjahr &= (10001001101010111111011110101010100010 0)_2\ km \\ abs.Nullpunkt &= -(100010001.0010011001100110011001100110011\ldots)_2\ °C \\ \pi &= (11.00100100001111111011010101000100010 0001\ldots)_2 \end{aligned}$$

In technisch wissenschaftlicher Binärnotation ist die normierte Darstellung daher

$$\begin{aligned} 1\ Lichtjahr &= (1.00010011010101111110111010101010001)_2 \times 2^{45}\ km \\ abs.Nullpunkt &= -(1.00010001001001100110011001100110011\ldots)_2 \times 2^8\ °C \\ \pi &= (1.10010010000111111011010101000100010000 1\ldots)_2 \times 2^1 \end{aligned}$$

Weil es sich um Binärzahlen handelt, steht nach der Normierung vor dem Dezimalpunkt nur eine 1. Es besteht eigentlich kein Grund, diese 1 zu speichern, da sie immer da ist. Infolgedessen speichert man nur die restlichen Ziffern der Mantisse. Die nicht gespeicherte 1 wird auch als *hidden bit* bezeichnet.

Zusammenfassend stellt also eine normierte binäre Gleitpunktzahl mit Vorzeichen V, Mantisse $m_1 \ldots m_n$ und Exponent E den folgenden Zahlenwert dar:

$$(-1)^V \times (1 + m_1 \times 2^{-1} + \ldots + m_n \times 2^{-n}) \times 2^E$$

Bei der Auswahl einer Darstellung von Gleitpunktzahlen in einem festen Bitformat, etwa durch 32 Bit oder durch 64 Bit, muss man sich entscheiden, wie viele Bits für die Mantisse und wie viele für den Exponenten reserviert werden sollen.

Die Berufsvereinigung *IEEE* (*Institute of Electrical and Electronics Engineers*) hat zwei Formate standardisiert. Eines verwendet 32 Bit und heißt *single precision* oder *short real*, das zweite verwendet 64 Bit und heißt *double precision* oder auch *long real*.

short real:	Vorz.: 1 Bit,	Exp.: 8 Bit,	Mantisse: 23 Bit,	bias 127
long real :	Vorz.: 1 Bit,	Exp.: 11 Bit,	Mantisse: 52 Bit,	bias 1023.

Allgemein gilt: Werden für die Darstellung des Exponenten e Bits verwendet, so wählt man als bias $2^{e-1}-1$. Als single precision Gleitkommazahl wird beispielsweise der absolute Nullpunkt -273.15 durch folgende Bitfolge gespeichert:

```
1 10000111 00010001001001100110011
```

- Das erste Bit ist das Vorzeichen, das anzeigt dass die Zahl negativ ist.
- Dann folgt der Exponent mit bias 127, also also $127 + 8 = 135 = (10000111)_2$.
- Die Mantisse (ohne das hidden Bit) ist 00010001001001100110011.

1.4.13 Rechenungenauigkeiten

Die Zahl 0 hat aufgrund der Annahme des hidden Bits keine exakte Repräsentation. Die Bitfolge, die nur aus Nullen besteht, repräsentiert die folgende Zahl:

$$\begin{aligned} Vorzeichen &: 0 \\ Exponent &: -127 \\ Mantisse &: 00000000000000000000000 \end{aligned}$$

Da zur Mantisse noch das hidden Bit hinzukommt, erhalten wir 1.0×2^{-127}. Dies ist die betragsmäßig kleinste darstellbaren Zahl. Sie wird mit 0 identifiziert, ebenso ihr Negatives, -1.0×2^{-127}. Es gibt also +0 und −0.

Die Zahlen $1.0 \times 2^{+127}$ und $-1.0 \times 2^{+127}$ werden in vielen Programmiersprachen als *Infinity* (unendlich) bzw. *−Infinity* bezeichnet. Alle anderen Zahlen mit Exponent 128 mit *NaN* (not a number), denn ihr Inverses wäre nicht darstellbar.

Bereits beim Umrechnen dezimaler Gleitpunktzahlen in binäre Gleitpunktzahlen treten Rundungsfehler auf, weil wir die Anzahl der Stellen für die Mantisse beschränken. So lässt sich beispielsweise die dezimale Zahl 0.1 nicht exakt durch eine binäre Gleitpunktzahl darstellen. Wir erhalten nämlich:

2×0.1	=	0.2	→ Ziffer: 0
2×0.2	=	0.4	→ Ziffer: 0
2×0.4	=	0.8	→ Ziffer: 0
2×0.8	=	1.6	→ Ziffer: 1
2×0.6	=	1.2	→ Ziffer: 1
2×0.2	=	0.4	→ Ziffer: 0
...	*etc.*	...	

Die binäre Gleitpunktdarstellung ist hier periodisch:

$$(0.1)_{10} = (0.0001100110011001100\ldots)_2.$$

Brechen wir die Entwicklung ab, so wird die Darstellung ungenau. Im kurzen Gleitpunktformat hat die Dezimalzahl 0.1 die binäre Darstellung:

```
0 011 1101 1100 1100 1100 1100 1100 1101.
```

Nach dem Vorzeichen 0 folgen die 8 Bits 01111011 des Exponenten, welche binär den Wert 123 darstellen. Davon muss $127 = 2^7 - 1$ subtrahiert werden, um den tatsächlichen Exponenten −4 zu erhalten. Es folgen die Bits der Mantisse. Erstaunlich ist hier das letzte Bit. Es sollte eigentlich 0 sein. In Wirklichkeit rechnet die CPU intern mit einem 80 Bit Gleitpunktformat und rundet das Ergebnis, wenn es als float (short real) gespeichert wird. So wird ...0011 beim Verkürzen zu ...01 aufgerundet.

Zusätzliche Ungenauigkeiten entstehen bei algebraischen Rechenoperationen. Addiert man 0.1 zu 0.2, so erhält man in Java:

$$0.1 + 0.2 = 0.30000000000000004$$

Die Ungenauigkeit an der siebzehnten Nachkommastelle ist meist unerheblich, in einem Taschenrechner würde man sie nicht bemerken, weil sie in der Anzeige nicht mehr sichtbar ist.