Friedrich Steimann

Einsen und Nullen: Grundlagen der Digitalisierung

Kompaktkurs in vier Einheiten 1♂

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung s
Übersetzung und des Nachdrucks, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Forn
Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung e
scher Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Dieser Text verwendet die Schriftarten Frutiger LT Com 45 Light, Roboto Mono und Fira Math.

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **i**

Vorwort ..... **i**

Zum Inhalt ii

Zur Form ii

Danksagungen iv

Kurseinheit 1: Zeichenspiele **1**

1 Zeichen **1**

1.1 Zeichen in Zeit und Raum 3

1.2 Bedeutung von Zeichen 4

2 Zahlen **4**

2.1 Stellenwertsysteme 5

2.2 Die Länge von Zahlen 7

2.3 Das Dualsystem 9

2.4 Rechnen mit Dualzahlen 10

2.4.1 Addition 10

2.4.2 Multiplikation 13

2.4.3 Subtraktion und negative Dualzahlen 14

2.4.4 Division 15

2.4.5 Vergleich von zwei Zahlen 16

2.5 Andere Stellenwertsysteme 16

2.6 Andere als ganze Zahlen 17

2.6.1 Rationale Zahlen 17

2.6.2 Dezimalbrüche und Festkommazahlen 17

2.6.3 Gleitkommazahlen 18

2.6.4 Zahlen, die keine sind 19

2.7 Inkrement, Dekrement und die zeitliche Dimension 19

3 Wahrheitswerte **21**

3.1 Rechnen mit Wahrheitswerten: logische Operatoren 23

3.2 Aussagenlogik 26

3.2.1 Erfüllbarkeit und Allgemeingültigkeit 28
3.2.2 Logische Schlussfolgerung 29
3.2.3 Boolesche Algebra 33
3.3 Andere Logiken 34
3.3.1 Mehrwertige Logik 34
3.3.2 Prädikatenlogik 36
3.4 Künstliche Intelligenz 37

4 Mehr Zeichen und Zeichenketten 39
4.1 Codierung von Zeichen 39
4.2 Zeichenketten 41
4.3 Die Länge von Zeichencodes und von Zeichenketten 42
4.4 Operationen auf Zeichenketten 43
4.5 Manipulation von Zeichenketten 44

5 Grafiken und Bilder 44

6 Signale 46

7 Kompression 47

8 Verschlüsselung 48

9 Programme 49

10 Segen und Fluch des Binären 53

Kurseinheit 2: Computer 55

11 Hardware 56
11.1 Gatter 57
11.2 Zeit, Synchronisation und Takt 60
11.2.1 Taktung von Gattern 61
11.2.2 Sequentielle Logik und Automaten 63
11.2.3 Grenzen der Taktfrequenz 64
11.3 Flüchtiger Speicher 66
11.4 Bus 68
11.4.1 Speicheradressierung 68
11.4.2 Paralleler vs. serieller Bus 69

11.4.3 Busbreiten 70
11.5 Prozessor 70
11.6 Ein- und Ausgabe 72
11.6.1 Controller 72
11.6.2 Festspeicher 73
11.6.3 A/D- und D/A-Wandler 74
11.7 Computerarchitektur 75
11.8 Der Computer als Universalmaschine 76
11.9 Zusammenfassung 76

12 Betriebssystem 77
12.1 Systemaufrufe 78
12.2 Hochfahren 78
12.3 Gerätetreiber 79
12.4 Speicherverwaltung 80
12.5 Dateisystem 81
12.6 Programmverwaltung und Multitasking 83
12.7 Klicki-Bunti 85
12.8 Herunterfahren 86

13 Internet 87
13.1 Rechnerkommunikation über Internet 88
13.2 Personenkommunikation über Internet 89
13.3 Das Web 89
13.4 Der Webbrowser als Betriebssystem 91
13.5 Das Internet als rechtsarmer Raum 91

14 Cloud 93

15 Sicherheit 94
15.1 Quellen der Unsicherheit 95
15.2 Ungenügender staatlicher Druck 96

Kurseinheit 3: Programmierung 97

16 Algorithmus 97

17 Spezifikation 101

17.1 Spezifikation als Teil eines Softwareentwicklungsprozesses 102
17.2 Formale Spezifikation 103
17.3 Formaler Beweis und Testen der Korrektheit: Verifikation und Falsifikation .. 104
17.4 Algorithmen und Korrektheit heute 105

18 Programm 106
18.1 Programme als Texte 106
18.2 Vom Algorithmus zum Programm 109
18.3 Korrektheit von Programmen 112
18.3.1 Verifikation von Programmen 112
18.3.2 Falsifikation von Programmen: Testen 114
18.3.3 Defensive Programmierung 114
18.3.4 Design by Contract 114
18.4 Vom Problem zum Programm: funktionale Dekomposition 115
18.5 Ausführung von Programmen auf einem Computer 117

19 Programmierparadigmen 118
19.1 Imperative Programmierung 118
19.2 Funktionale Programmierung 121
19.3 Logische Programmierung 123
19.4 Objektorientierte Programmierung 126
19.5 Prototypenbasierte Programmierung 127

20 Programmiersprachen 128
20.1 Syntax und Semantik von Programmiersprachen 128
20.2 Faktoren beim Programmiersprachenentwurf 129
20.3 Verbreitung von Programmiersprachen 130
20.4 Klassifikation von Programmiersprachen 131
20.5 Abstraktion und essenzielle vs. akzidentelle Komplexität 134

21 Programmierwerkzeuge 135
21.1 Debugger 136
21.2 Testwerkzeuge 136
21.3 Versionskontrolle 137
21.4 Automatische Programmierung 138
21.5 Integrierte Entwicklungsumgebungen 138

22 Zum Beispiel Scratch 139
22.1 Das Programmiermodell von Scratch 140
22.1.1 Objekte 140
22.1.2 Ereignisgesteuerte, strukturierte Skripte 140
22.1.3 Zustand und Variablen 142
22.2 Arbeiten mit Listen 144
22.3 Programmieren in Scratch 147
22.4 Entwicklung eines Programmierstils 147

23 Anwendungsprogramme 148
23.1 Anwendungsprogramme mit Webschnittstelle 149
23.2 Apps 150

24 Zum Beispiel Visual Basic for Applications (VBA) 151

25 Eingebettete Software 154

26 Softwarehaftung und Stand der Technik 155

Kurseinheit 4: Daten 157

27 Datentypen und -strukturen 158
27.1 Typ und Instanz 159
27.2 Typkonstruktoren 160
27.2.1 Datenverbünde (Records) 160
27.2.2 Rekursive Datentypen und Zeiger 161
27.2.3 Felder (Arrays) 162
27.2.4 Weitere Typkonstruktoren 163
27.3 Abstrakte Datentypen 164
27.4 Wichtige Datentypen 165
27.4.1 Listen 165
27.4.2 Bäume 166
27.4.3 Assoziativspeicher 167

28 Datenmodellierung 168
28.1 Schema 169
28.2 Datenmodelle 171
28.2.1 Das Entity-Relationship-Modell 171

28.2.2 Das relationale Datenmodell 171
28.2.3 Ältere Datenmodelle 172
28.2.4 Semantische Datenmodelle 173
28.3 Ontologien 174
28.4 Begriffssysteme 175
28.5 Datenmodellierung mit Grammatiken 176
28.6 Zeitliche Dimension 176

29 Schwach strukturierte Daten 177
29.1 Attribut/Wert-Paare 177
29.2 Auszeichnung 179

30 Datenbanken 181
30.1 Datenbanksprachen 181
30.2 Datenbankprogramme 183
30.3 Transaktionssicherheit 184

31 Big Data 186

32 Datenvisualisierung 188
32.1 Visualisierung zum Zweck der Theoriebildung 188
32.2 Visualisierung zum Zweck der Information 189
32.3 Visualisierung multidimensionaler Daten 189

Nachwort 191

Verzeichnis der Weblinks im Rand 193

Index 197

Omnibus ex nihilo ducendis sufficit unum.
(Um alles aus nichts herzuleiten genügt eines.)

Gottfried Wilhelm Leibniz (* 1. Juli 1646, † 14. November 1716)

Vorwort

Digitalisierung ist eine von den technischen Neuerungen, die von vielen genutzt, von wenigen gestaltet und von niemandem aufgehalten werden. Da Digitalisierung unseren Alltag zunehmend durchdringt, kann man zwar versuchen, sie zu ignorieren, sich ihr damit aber nicht entziehen. Warum sich ihr also nicht stellen und sich ein eigenes Bild von ihr machen? Weil es mühsam ist und am Ende gar nicht notwendig?

Dass Digitalisierung in unserem Alltag so durchschlägt, liegt schlicht daran, dass

- sich viele unserer alltäglichen Anliegen als *Leibnizsche* „**Spiele mit Zeichen**" auffassen lassen und dass
- für diese Spiele zwei verschiedene Zeichen ausreichen.

Man könnte natürlich auch mehr verschiedene Zeichen verwenden, aber die Beschränkung auf nur zwei verschiedene Zeichen macht die technische Umsetzung der Zeichenspiele einfacher (was auch schon Leibniz erkannt hatte) und so ist es bei zwei geblieben. Als Zeichen haben sich 1 und 0 eingebürgert, die die meisten Menschen als Ziffern („Buchstaben von Zahlen") kennen; die Universalität der Digitalisierung speist sich jedoch daraus, dass die Bedeutung der beiden Zeichen allein durch ihren Gebrauch bestimmt wird, also das, was man — oder ein *Computer* — damit macht. Und auch wenn „Computer" übersetzt „*Rechner*" heißt, so ist Rechnen doch nur ein Teil dessen, was Computer tun.

Nun hilft die Einsicht, dass Computer nur Einsen und Nullen verarbeiten, beim Verständnis der Digitalisierung nicht weiter als die Einsicht, dass der Kölner Dom aus Steinen und Mörtel gebaut wurde, beim Verständnis vom Bauen: Man staunt vielleicht noch mehr als ohnehin schon, weiß aber noch immer nicht, wie es geht. Und so befasst sich dieser Kurs nicht nur mit Einsen und Nullen und den Zeichenspielen, die damit etwas machen, sondern auch mit den vielen Ebenen, die von der Verarbeitung von Einsen und Nullen abstrahieren und die es damit leichter machen, Aufgaben per Digitalisierung zu lösen. Der Kurs gerät damit zu einem Streifzug durch die Informatik, der viele Dinge anreißt, ohne sie erschöpfend zu behandeln. Dabei ist sein Ziel, seine Leser auf eine — hoffentlich die richtige — Spur zu setzen, wenn sie einmal selbst ein Digitalisierungsprojekt starten; allen anderen möchte er dabei helfen, die Natur der Digitalisierung besser zu verstehen, um ihre Deutung nicht anderen überlassen zu müssen.

Zum Inhalt

Der Kurs ist in vier Einheiten gegliedert. Die erste Kurseinheit „Zeichenspiele“ ist eher theoretischer Natur und befasst sich mit der Repräsentation und Verarbeitung beliebiger Daten mit nur zwei verschiedenen Zeichen. Dieser Teil des Kurses ist losgelöst von irgendeiner technischen Realisierung der Digitalisierung: Zur Veranschaulichung und zum Ausprobieren werden lediglich Papier, Bleistift und Radiergummi benötigt.

Die zweite Kurseinheit „Computer“ steht im Gegensatz dazu: Hier geht es um die Repräsentation und Verarbeitung von Zeichen in einer Maschine, eben dem Computer. Dabei ist die sog. Hardware nur ein Teil eines Computers; der andere ist sein Betriebssystem, das von den Details der Hardware abstrahiert und für seine Benutzer und Programmierer gleichermaßen eine über Hardwarevarianten hinweg standardisierte „logische Maschine“ zur Verfügung stellt. Mit Computern unauflösbar verbunden sind auch das Internet sowie die Verteilung von Computerdienstleistungen in der sog. Cloud. Eine kurze Betrachtung der Sicherheitsprobleme, die Computern innewohnen, schließt diese Kurseinheit ab.

Die dritte Kurseinheit „Programmierung“ befasst sich mit der Abrichtung von Computern für bestimmte Aufgaben. Da heute praktisch alle digitalen Prozesse programmiert sind, kommt der Programmierung eine zentrale Bedeutung zu. Neben der die eigentliche Programmierung vorbereitenden Befassung mit Algorithmen und Spezifikationen spielen Programmierparadigmen, -sprachen und -werkzeuge in dieser Kurseinheit eine wichtige Rolle.

Mit der vierten Kurseinheit „Daten“ schließt sich ein Kreis: Einerseits sind Zeichen und Daten eng verwandt, andererseits gibt es starke Wechselwirkungen zwischen der Programmierung und der Strukturierung von Daten. Letztere lässt sehr viele Freiräume und in vielen Fällen ist es sinnvoll, die Strukturierung nicht von den Programmen, die die Daten verarbeiten, bestimmen zu lassen, sondern sie als eigenständige Aufgabe zu betrachten, schon weil Daten so von einzelnen Programmen unabhängig und vielen verschiedenen zugänglich werden. Diese Sicht auf Daten schlägt sich in der Existenz von Datenmodellierung und Datenbanken nieder. Mit Big Data und Datenvisualisierung wird in dieser Kurseinheit auch noch die Betrachtung von Daten als dem „neuen Öl“ unserer Gesellschaft kurz beleuchtet.

Zur Form

Index

Der Kurstext ist mit einem klassischen Schlagwortverzeichnis ausgestattet. Dieses soll drei Zwecken dienen: der Verwendung des Kurstextes als Nachschlagwerk, der Herstellung von inhaltlichen Zusammenhängen zwischen verschiedenen Teilen des Kurstextes und dem Wiederauffinden von irgendwo einmal Gelesenem und nicht mehr so genau Erinnertem. Um die Nutzung des Indexes zu erleichtern, sind alle Vorkommen indizierter Begriffe (Schlagwörter) im Text hervorgehoben, und zwar kursiv für Anführungen oder Verwendungen von Begriffen und fett für Definitionen. Wenn Sie also über einen kursiv gesetzten Begriff stolpern, dann kann es sich lohnen, ihn anzuklicken oder ihn im Index nachzuschlagen und dort nach seiner Definition zu suchen. Allerdings werden nicht alle

angeführten Begriffe auch definiert und die Definitionen fallen hier und da eher schmal aus — der Fluss war mir wichtiger als die Abgeschlossenheit oder, anders ausgedrückt, ich wollte eher einen Kurstext als ein Nachschlagwerk schreiben. Fehlt eine Definition, kann der Index immer noch dabei helfen, sich die Bedeutung eines Begriffs aus anderen Kontexten zu erschließen. Da ist es gut zu wissen, dass in der elektronischen Version die Seitenzahlen im Index mit den Vorkommen der indizierten Begriffe im Text verlinkt sind.

Apropos Nachschlagwerk: Der Kurstext enthält zahlreiche Verweise auf Wikipedia-Artikel, die sich hinter den Wikipedia-Kugeln im Rand verbergen. Die Namen und URLs der Artikel müssen in papiergebundenen Ausgaben dieses Kurses im entsprechenden Verzeichnis an seinem Ende („Verzeichnis der Weblinks im Rand") nachgeschlagen werden; Android-Nutzer können auch den Barcode im Seitenfuß mittels dazugehöriger experimenteller *App* einscannen und bekommen so alle Links der gescannten Seite sowie die dazugehörigen Indexeinträge angezeigt (*Papier/Digital-Brücke*). Zu den Wikipedia-Artikeln selbst kann man natürlich vieles sagen; ich sage: Wer etwas Falsches entdeckt, möge es korrigieren.

Barrierearmut

Barrierefreiheit ist eine Illusion. Ich habe mich gleichwohl bemüht, einen barrierearmen Kurstext zu erstellen. Die oben bereits erwähnte Android-App (die mittels nebenstehendem QR-Code heruntergeladen werden kann) bietet zu diesem Zweck eine seitenweise Vorlesefunktion, die der von PDF-Readern zumindest teilweise überlegen ist.

Genderisierung

Die Genderisierung eines Textes wie diesem ist eine interessante Angelegenheit, zeigt sie doch auf, wie sehr das (grammatische) Geschlecht in unserer Sprache verwurzelt ist. Wem das noch nicht klar sein sollte, der möge diesen Satz in Betracht ziehen: Fängt er noch scheinbar unverfänglich mit „wem" an, so offenbart das anschließende „der" des zweiten Halbsatzes, dass mit „wem" wohl ein männliches Wesen gemeint sein muss. Dem generischen Feminin verpflichtet habe ich daher in der Vergangenheit stets „Wem …, die möge …" geschrieben, aber das generische Feminin verursacht nicht nur diskriminatorische Kollateralschäden (wenn ich nun etwa von Anfängerinnen und Kindern anstelle von Anfängern und Kindern schreibe), sondern hat zu einigen teils deutlichen Beschwerden wegen verminderter Lesbarkeit (nicht wenige davon von Frauen) meiner sowieso schon recht verwinkelten Sätze geführt. Nun könnte man hier auf Gewöhnung setzen („die Gästin" klingt ja auch von Mal zu Mal natürlicher und bei Kurgästin stolpert wohl niemand mehr — ach ja, ist niemand eigentlich Mann oder Frau oder tatsächlich niemand?) oder aber gänzlich umformulieren, doch weil ich etwas Neues ausprobieren wollte, bin ich einen dritten Weg gegangen: Ich habe einen parametrisierten Text erstellt, aus dem sich zwei Varianten generieren lassen, eine mit generischem Feminin und eine mit generischem Maskulin (die vorliegende). Damit die beiden Texte sich inhaltlich nicht unterscheiden (was zu neuen Diskriminierungen führen könnte), habe ich auf Umformulierungen weitgehend verzichtet, weswegen es hier und da — insbesondere in der femininen Version — in den Ohren mancher nach wie vor holpern mag. Trotz des Verzichts auf größere Anpassungen können sich aus genderisierungsbedingten Unterschieden unterschiedliche Zeilen- und damit auch Seitenumbrüche ergeben, wobei letztere zu in den Seitenzahlen differierenden Inhaltsverzeichnissen und Indexen führen können. Da Seitenzahlen aber sowieso eher ein Auslaufmodell sind, habe ich mich entschieden, diesen Kollateralschaden hinzunehmen.

Danksagungen

Ich bedanke mich herzlich bei meinen Kollegen Jörg Desel, Jörg Keller, Christoph Beierle und Bernhard Rumpe für die (teilweise) Durchsicht dieses Textes und bei meinem Kollegen Hubertus Busche für das Recherchieren des Beleges dafür, dass Leibniz auch wörtlich schon von Zeichenspielen („jeu de caractères“) schrieb.

Mein größter Dank aber gilt meinem Informatiklehrer Klaus König — ich wüsste nicht, wie man das Fach besser hätte vermitteln können. Und so enthält dieser Kurs viele bewusste Anleihen aus seinem Informatikunterricht der Jahre 1981–1984 am Otto-Hahn-Gymnasium in Springe.

Kurseinheit 1: Zeichenspiele

When I use a word, … it means just what I choose it to mean — neither more nor less.

Lewis Carroll (* 27. Januar 1832 , † 14. Januar 1898)
Through the Looking Glass

Die Bedeutung eines Wortes ist sein Gebrauch in der Sprache.

Ludwig Wittgenstein (* 26. April 1889, † 29. April 1951)
Philosophische Untersuchungen

Die erste Kurseinheit unterscheidet sich von den nachfolgenden dadurch, dass ihre Inhalte von der Existenz von Computern weitgehend unabhängig sind. Gleichwohl bleibt sie nicht vollkommen abstrakt — alle Zeichenspiele, die vorgestellt werden, können Sie sich auf einem Blatt Papier mittels Bleistift und Radiergummi durchgeführt denken. Dabei soll das Papier kariert und stets hoch und breit genug für den jeweiligen Zweck sein.

1 Zeichen

Zeichen und Glyphen

Zeichen sind Elemente der Kommunikation. Sie sind nicht an eine bestimmte Form gebunden: So gibt es beispielsweise grafische, akustische, optische und elektrische Formen von Zeichen, und dasselbe Zeichen kann in mehr als einer Form vorkommen. In einem geschriebenen Text wie diesem hier haben wir es ausschließlich mit grafischen Zeichen (Buchstaben, Satzzeichen etc.) zu tun, die dazu zumeist die Gestalt von sog. **Glyphen** annehmen (prominente Ausnahme: das *Leerzeichen*, das die Gestalt eines Zwischenraums hat). Wenn ich im folgenden Zeichen notiere, um darüber zu schreiben, dann verwende ich dafür zwar (notwendigerweise) Glyphen, meine aber in der Regel Zeichen ganz allgemein.

WIKIPEDIA

Notierung von Zeichen mit Zeichen

Da die Schrift selbst aus (grafischen) Zeichen besteht, bringt Zeichen zu notieren und darüber zu schreiben dann ein Problem mit sich, wenn alle Zeichen aus demselben Vorrat stammen: Es vermischt sich dabei nämlich zwangsläufig die

Objektebene, also die Ebene der Zeichen, über die man schreibt, mit der **Metaebene**, also der Ebene, in der man mithilfe von Zeichen schreibt. Dies kann zum Verlust der Eindeutigkeit führen. Ich werde daher in dieser Kurseinheit die beiden Ebenen voneinander trennen und tue dies, nicht ohne didaktische Hintergedanken, mithilfe von Zeichen (und nicht, was auch leicht möglich gewesen wäre, mithilfe verschiedener *Fonts*, also verschiedener Glyphen für dieselben Zeichen). Da in der Regel die allermeisten Zeichen eines Textes, in dem es um Zeichen geht, auf der Metaebene angesiedelt sind (so z. B. alle Zeichen dieses Absatzes bis hierher), verwende ich für die Zeichen, die auf der Objektebene stehen, eine spezielle Notation: Ich setze sie in schweizerische Anführungszeichen, also z. B. «1» und «0».

Anführungszeichen

Der letzte Satz des vorangegangenen Absatzes löst übrigens auf seine eigene Weise ein vertracktes Problem: Er führt schweizerische Anführungszeichen an, ohne sie dabei zu verwenden — dies tut er erst im nachgeschobenen Beispiel, das buchstäblich der Illustration dessen, was schweizerische Anführungszeichen sind, dient. Dabei wird es dem Leser überlassen, den Zusammenhang zwischen dem Namen „schweizerische Anführungszeichen“[1] und den so bezeichneten Zeichen, die erst im Beispiel verwendet werden, zu erkennen. Folgende alternative Formulierungen verdeutlichen das so gelöste Problem:

- „Beispielsweise schreibe ich «1» und «0» für die Zeichen 1 und 0.“ Dieser Satz vermischt in seinem hinteren Teil Objekt- und Metaebene (das Problem, das ja gerade gelöst werden soll), es sei denn, man versteht das vorangestellte Wort „Zeichen“ als die Ankündigung eines vorübergehenden Ebenenwechsels, der genau dem entspricht, der durch die schweizerischen Anführungszeichen eingeleitet und beendet wird.
- „Ich setze sie in schweizerische Anführungszeichen « und ».“ Dieser Satz kommt ohne ein Beispiel aus und führt die schweizerischen Anführungszeichen direkt an, wobei die Apposition „schweizerische Anführungszeichen“ nicht nur der Benennung dient, sondern auch einen vorübergehenden Ebenwechsel einleiten muss, damit der Satz das zu lösende Problem nicht selbst darstellt.
- „Ich setze sie in ««» und «»».“ Dieser Satz benutzt keine Apposition für die Anzeige des Ebenenwechsels, sondern die angeführten Anführungszeichen selbst. Dies beinhaltet aber ein klassisches Henne-Ei-Problem (oder *Münchhausen-Trilemma*), da hierbei Zeichen unter Verwendung derselben Zeichen eingeführt werden, man also den Satz nicht verstehen kann, wenn man seinen Inhalt nicht schon kennt.[2]

---

[1] Man beachte, dass die deutschen Anführungszeichen ebenfalls einen Ebenenwechsel anzeigen: hier den von der Verwendung von Namen zu deren Nennung, weiter unten den von Sätzen über Sätze zu den Sätzen, über die geschrieben wird. Die Bedeutung dieser Anführungszeichen habe ich dabei als vereinbart angenommen.

[2] Es stellt sich hierbei die Frage, ob es nicht ausreicht, zu wissen, dass in einem Satz ein Selbstbezug vorkommt (z. B. weil man verstanden hat, dass am Anfang ein Selbstbezug stehen muss, es ohne also gar nicht geht), um den Selbstbezug zu identifizieren und seinen Elementen eine Bedeutung zuzuweisen. Wenn es dann für die Bedeutung keine sinnvolle Alternative gibt, ist der Selbstbezug kein Problem.

Auch wenn der durch ein vorangestelltes Wort wie „Zeichen“ angekündigte vorübergehende Ebenenwechsel seinen sprachlichen Charme hat, verwende ich in dieser Kurseinheit (aber nur in dieser) die schweizerischen Anführungszeichen, nicht zuletzt, weil sie mir erlauben, das *Leerzeichen* auch ohne Glyphe als Zeichen aufzuschreiben: « ».

Sonderzeichen und reservierte Wörter

Übrigens: Beide Möglichkeiten zur Kennzeichnung eines Ebenenwechsels, die Verwendung bestimmter Wörter oder bestimmter Zeichen, verbrennen diese für deren normalen Gebrauch, da ein normaler Gebrauch nicht von ihrer Sonderfunktion zu unterscheiden wäre. Die Verwendung für diesen einen Zweck macht sie also zu *reservierten Wörtern* bzw. *Sonderzeichen* und reduziert den für den normalen Gebrauch verfügbaren Wort- bzw. Zeichenvorrat entsprechend. Daraus folgt aber unmittelbar, dass man in einem System mit nur zwei verschiedenen Zeichen auf die Zusammenfassung einer Folge von mehreren Zeichen in Einheiten, die als Ganzes zu verstehen sind, zurückgreifen muss, wenn man einen Ebenenwechsel kenntlich machen können möchte. Dies ist Gegenstand von Kapitel 4, „Mehr Zeichen und Zeichenketten“.

Digitalisierung ist Sophisterei

Wenn Ihnen die vorangegangenen Betrachtungen als haarspalterisch und verzichtbar erscheinen, dann sollten Sie bedenken, wie es wohl ist, wenn man nur Zeichen hat und auf keine Gewohnheiten, Erfahrungen oder externe, reale Bezugspunkte zurückgreifen kann, um einen Sachverhalt zu vermitteln. Wie würden Sie einer außerirdischen Intelligenz erklären, was Zeichen sind, welche Sie kennen und was sie bedeuten? Bei der Übertragung von Zeichen und was wir damit machen wollen (den Zeichenspielen) auf Computer, die zwar irdisch, aber nicht intelligent sind, müssen wir uns an eine — möglichst einfache — Systematik halten. Sich also mit so kleinen Details wie der Anführung von Zeichen zu befassen mag dem einen oder anderen mühsam erscheinen, aber es hat auch einen ganz besonderen Reiz, nämlich den, zu versuchen, sich mit minimalen Voraussetzungen in die Lage zu versetzen, die ganze Welt systematisch zu beschreiben.

1.1 Zeichen in Zeit und Raum

Sobald man es mit mehr als einem Vorkommen von Zeichen zu tun hat, stellt sich die Frage nach deren Anordnung. In der Schrift werden Zeichen[3] in linearer Folge im (zunächst eindimensionalen) Raum angeordnet, wobei solche *Zeichenfolgen* aus Platzgründen in Zeilen (zweite Dimension) und Seiten (dritte Dimension) umgebrochen werden. Zeichenfolgen, die keinem gesprochenen Text entsprechen (der ja, der Natur des Sprechens folgend, stets eindimensional ist), werden auch gern tabellarisch dargestellt, wobei dann nicht nur ihre Anordnung neben-, sondern auch untereinander bedeutungstragend ist (bei der Addition von Zahlen zum Beispiel). Akustische und elektrische Zeichenfolgen sind von Natur aus eindimensional; diese Folgen erstrecken sich jedoch nicht im Raum, sondern in der *Zeit*. Wie wir

[3] Hier und im folgenden verwende ich das Wort „Zeichen“ auch, wenn ich „Vorkommen von Zeichen“ meine. Dies entspricht dem allgemeinen Sprachgebrauch und soll nicht zum Gegenstand einer weiteren Sophisterei gemacht werden.

noch sehen werden, ist die Konvertierung zwischen den verschiedenen Formen und Dimensionen von Zeichenfolgen ein zentraler Bestandteil der Digitalisierung.

1.2 Bedeutung von Zeichen

Das Wesen von Zeichen ist, dass man sie unterscheiden kann. Werden sie in der Kommunikation benutzt, muss man ihnen zusätzlich eine Bedeutung zuordnen.

Im Kontext der Digitalisierung besondere Prominenz erlangt haben die Zeichen «1» und «0». Sie können für vieles stehen, so beispielsweise für

- die Zustände *An* und *Aus* (oder *Hoch* und *Niedrig*, oder irgendein Paar von komplementären Zuständen),
- die Wahrheitswerte *Wahr* und *Falsch* und
- die Zahlen Eins und Null.

Dabei ist die Zuordnung der Zeichen zu ihrer jeweiligen Bedeutung willkürlich und lediglich durch ihren Gebrauch festgelegt. Den meisten Menschen dürfte die Interpretation von «1» und «0» als Zahlen Eins und Null die geläufigste sein und tatsächlich spielen Zahlen bei der Digitalisierung eine wichtige Rolle, aber die Grundlage der Digitalisierung, wie wir sie heute kennen, sind die beiden ersten Interpretationen: Die erste spiegelt die technische Umsetzung der Digitalisierung wider und die zweite gehört zur Logik, die dahintersteht. Zahlen kommen erst danach und stehen auf einer Stufe mit allen anderen Inhalten der Digitalisierung. Dennoch will ich hier mit den Zahlen beginnen; Wahrheitswerte folgen in Kapitel 3 und (elektrische) Zustände in Kurseinheit 2.

2 Zahlen

Da sie zum Zählen benutzt werden, entwickelt jedes Kind sehr schnell ein intuitives Verständnis für die natürlichen Zahlen.[4] So steht „Null“ für keines, „Eins“ für eines, „Zwei“ für zwei usw. Andere Zahlen (also z. B. negative, gebrochene oder reelle Zahlen) werden mittels Rechenoperationen aus den natürlichen Zahlen abgeleitet (also etwa „minus Eins“ als Null minus Eins, „Einhalb“ als Eins geteilt durch Zwei usw.). Dabei sind die Zahlen für sich genommen vollkommen abstrakt; erst indem man sie etwa der Länge, dem Gewicht oder der Anzahl von etwas zuordnet, bekommen sie einen Bezug zur Realität. Dadurch erhalten auch

[4] Man kann darüber streiten, ob natürliche Zahlen wirklich natürlich sind. Ihre Bedeutung setzt den Begriff der *Entität* voraus, also den von etwas Abgrenzbarem (in einer Welt nur aus Wasser und Luft wären auch natürliche Zahlen nicht natürlich). Man könnte aber behaupten, dass auch Entität nur ein mentales, von unserer Wahrnehmung unserer Umwelt geprägtes Konstrukt ist.

die (für sich genommen ebenfalls vollkommen abstrakten) Rechenregeln ihre praktische Bedeutung. So entspricht etwa die Addition der Aneinanderreihung von Längen, dem gemeinsamen Abwiegen und dem Hinzuzählen von etwas. Interessanterweise fällt vielen Menschen das abstrakte Hantieren mit Zahlen leichter als das Lösen der berühmt-berüchtigten „Textaufgaben“ mit ihrem Realitätsbezug, was vermutlich daran liegt, dass bei den abstrakten Rechenaufgaben klarer ist, welche Rechnungen durchzuführen sind, und diese Rechnungen eingeübten Zeichenspielen entsprechen. Bei Computern zumindest ist das so: Sie beherrschen nur Zeichenspiele.

Während wir in der gesprochenen Sprache Zahlen Namen gegeben haben (eben „Null“, „Eins“ usw.), schreiben wir sie meistens als Folge von speziellen Zeichen auf. In den Anfängen der Menschheit wurden dafür vielleicht Striche verwendet, also etwa «I» für Eins, «I»«I» für Zwei usw. (mit dem Problem, dass man Null schlecht schreiben konnte); für größere Zahlen später andere Zeichen (wie bei den Römern «V» für Fünf, «X» für Zehn usw.), wobei dann einfaches Abzählen schon nicht mehr genügte, um eine geschriebene Zahl in eine gesprochene Zahl (ihren Namen) oder ihre Bedeutung umzusetzen. In der Tat ist ein Nachteil der römischen Zahlenschrift, dass sie sich für Zeichenspiele nicht besonders eignet.

2.1 Stellenwertsysteme

Heute notieren fast alle Menschen Zahlen mithilfe von **Stellenwertsystemen**. Ein Stellenwertsystem beruht auf einer mit ihm verbundenen Anzahl verschiedener Zeichen, **Ziffern** genannt, und eben **Stellen**. Das **Zehnersystem**, das auch **Dezimalsystem** genannt wird, hat zehn Ziffern, «0»–«9», die, wenn wir sie als Zahlen interpretieren, Null bis Neun bedeuten.[5] Zahlen größer als Neun werden als Aneinanderreihung, oder Folge, von Ziffern geschrieben, also etwa «1»«0» für Zehn, «1»«1» für Elf, «1»«2» für Zwölf und «1»«3» für Dreizehn.[6] Für welche Zahl eine solche Folge von Ziffern steht, wird durch das verwendete Stellenwertsystem festgelegt; ich habe hier stillschweigend das Dezimalsystem unterstellt (und werde dies auch weiter so tun, wenn ich auf nichts anderes hinweise).

Die Stellen einer Zahl in üblicher Schreibweise werden von rechts nach links gezählt. So hat **10** zwei Stellen[7], von denen die erste mit «0» und die zweite mit «1» besetzt ist. Dabei hat

---

[5] Englisch heißt Ziffer übrigens digit, von lat. digitus für Finger, derer wir ja zehn haben. Die Stellen einer Zahl heißen places (wobei eine *n*-stellige Zahl trotzdem *n*-digit number heißt).

[6] Am Wechsel der Nomenklatur am Übergang von „Zwölf“ (Eigenname) und „Dreizehn“ (zusammengesetzt) kann man ablesen, dass das Dezimalsystem erst nach unserer Sprache in unseren Kulturkreis Einzug gehalten hat (sonst würde **11** wohl „Einzehn“ heißen).

[7] Ab hier schreibe ich Zahlen auch auf der Objektebene (als Gegenstand von Zeichenspielen) nicht mehr als Folge in Anführungszeichen gesetzter Zeichen, sondern wie üblich als einfache Ziffernfolge, verwende dafür aber (sofern sie auf der Objektebene stehen) einen etwas anderen Font (also beispielsweise **10** anstatt 10 oder gar «1»«0»). Für andere Zeichenfolgen werden später auch andere Schreibweisen eingeführt.

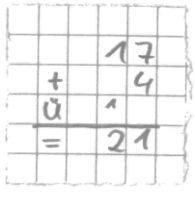
jede Stelle einen Wert, den **Stellenwert**, der sich aus ihrer Position ergibt: Im Dezimalsystem hat die erste Stelle (ganz rechts stehend) den Stellenwert Eins (= 10^0), die zweite den Stellenwert Zehn (= 10^1), die dritte den Stellenwert Hundert (= 10^2) sowie allgemein die *n*-te Stelle den Stellenwert 10^{n-1}. Die Zahl, für die **10** steht, errechnet sich als Eins mal Zehn plus Null mal Eins (gleich Zehn), **11** als Eins mal Zehn plus Eins mal Eins usw.; **2768** steht für Zwei mal Tausend plus Sieben mal Hundert plus Sechs mal Zehn plus Acht mal Eins. Im *Zwölfersystem* dagegen hat die erste Stelle die Wertigkeit Eins (12^0), die zweite Zwölf (12^1), die dritte Einhundertvierundvierzig (12^2) usw.

Kommazahlen

Wir zählen also die Stellen einer geschriebenen ganzen Zahl von ihrem rechten Ende her und verbinden mit jeder Stelle einen Stellenwert, der sich im Dezimalsystem als Zehnerpotenz der Position der Stelle minus Eins ergibt. Wenn man die Stellen hinter dem Ende einer ganzen Zahl nach rechts fortsetzt, ergeben sich nach diesem System für die Stellen negative Exponenten, also ein Zehntel (10^{-1}), ein Hundertstel (10^{-2}) usw. Dabei trennt man den gebrochenen (also nicht ganzen) Teil der Zahl vom ganzen durch ein Komma. So steht **0,1** für Null mal Zehn hoch Null plus Eins mal Zehn hoch minus Eins (wobei Zehn hoch minus Eins dieselbe Zahl wie Eins durch Zehn hoch Eins bezeichnet). Mehr dazu in den Abschnitten 2.6.2 und 2.6.3.

Addition als Zeichenspiel

Ein großer Vorteil von Stellenwertsystemen wie dem Dezimalsystem ist es, dass es die Mechanisierung des Rechnens als einfache *Zeichenspiele* erlaubt. So lässt sich beispielsweise die Addition zweier Zahlen bewerkstelligen, indem man die beiden Summanden am Komma ausgerichtet (oder rechtsbündig, wenn kein Komma darin vorkommt) untereinander aufschreibt und dann stellenweise von rechts nach links ihre Ziffern zur Summe addiert (genauso, wie Sie es in der Schule gelernt haben):



Dazu muss man nur die Summe von je zwei Ziffern „im Kopf rechnen“ und ggf. noch den **Übertrag** hinzuzählen — alles andere ist ein reines Zeichenspiel.

Multiplikation als Zeichenspiel

Bei der Multiplikation kann man ausnutzen, dass man für die Multiplikation mit **10** nur eine «0» an den Multiplikanden anhängen muss (oder, anders ausgedrückt, alle Ziffern des Multiplikanden um eine Stelle nach links verschieben und die freigewordene Einerstelle mit «0» auffüllen muss) und dass man die Multiplikation zweier Zahlen auf eine Kombination von Additionen und Multiplikationen mit **10** zurückführen kann: $12 \cdot 3 = 10 \cdot 3 + 2 \cdot 3 = 30 + 3 + 3$. So ist auch die Multiplikation im Dezimalsystem nur ein — wenn auch etwas komplexeres — Zeichenspiel.

Beschränkung von Stellenwertsystemen

Mit einem Stellenwertsystem allein kann man nicht alle Zahlen darstellen. So lässt sich beispielsweise schon ein Drittel nicht als Dezimalzahl (d. h.,

als eine Zahl im Dezimalsystem) hinschreiben, weil man dafür eine unendliche Anzahl von Nachkommastellen bräuchte. Für diese Zahlen braucht man ergänzende Schreibweisen, also etwa 1/3 (die Bruchschreibweise). Diese Schreibweisen geben eine Rechenoperation an, mittels derer man Zahlen aus anderen Zahlen erhält, für die es schon eine Schreibweise gibt. Auch für solche Zahlendarstellungen existieren Zeichenspiele, die einem erlauben, mit ihnen zu rechnen (bei Brüchen etwa die Regeln der Bruchrechnung). Für manche Zahlen, wie etwa die Kreiszahl Pi oder die Eulersche Zahl, sind die definierenden Rechenoperationen Approximationen und so unhandlich, dass man für diese Zahlen *Sonderzeichen* eingeführt hat, wie «π» und «e».[8] Zahlen, die sich eigentlich nicht in einem Stellenwertsystem darstellen lassen, werden es, der einfacheren Zeichenspiele wegen, häufig dennoch — sie, wie auch die Rechenergebnisse, sind dann allerdings nur näherungsweise korrekt.[9]

2.2 Die Länge von Zahlen

Die Länge und damit auch die Größe oder Kleinheit von Zahlen, die mit einem Stellenwertsystem dargestellt werden können, ist theoretisch unbegrenzt. Beim tatsächlichen Aufschreiben von Zahlen kann man aber (aus naheliegenden praktischen Gründen) immer nur endlich viele Stellen verwenden. Eine gängige Konvention ist dabei, dass man vor dem Komma am Anfang der Zahl stehende („führende") und nach dem Komma am Ende der Zahl stehende („nachfolgende") Nullen weglässt. Statt 0815 schreibt man also 815 und statt 2,0 nur 2. Dies ist zulässig, weil diese Nullen zum Zahlenwert nichts beitragen und, anders als eingeschlossene Nullen, nicht zum Abzählen der Stellen (Bestimmung der Position einer Ziffer) benötigt werden — davor oder danach „kommt nichts mehr", also kann man aufhören. Das gilt allerdings nur, wenn eine Zahl alleine dasteht, also insbesondere nicht mehrere Zahlen direkt hintereinander aufgeschrieben werden.

In der Digitalisierung passiert aber genau das: Wenn alles in Zeichenfolgen konvertiert wird, dann ergibt sich beim Aufschreiben die Frage, wo die eine Zahl aufhört und die nächste beginnt. Um das zu beantworten gibt es mindestens drei Möglichkeiten:

1. Man legt sich auf eine einheitliche Stellenzahl für alle Zahlen fest (vor und nach dem Komma) und kann so durch Abzählen der Stellen bestimmen, wo eine aufhört und die nächste beginnt und wo in der Zahl das Komma steht. Dies hat den Vorteil, dass

---

[8] Für wieder andere Zahlen gibt es nicht einmal Approximationen. Das folgt daraus, dass es überabzählbar viele Zahlen gibt, aber nur abzählbar viele Approximationsausdrücke. Letzteres folgt wiederum daraus, dass Approximationsausdrücke stets endlich sind und es ein Verfahren gibt, mit dem man alle aufzählen kann (auch wenn es unendlich viele sind und das Verfahren entsprechend niemals endet).

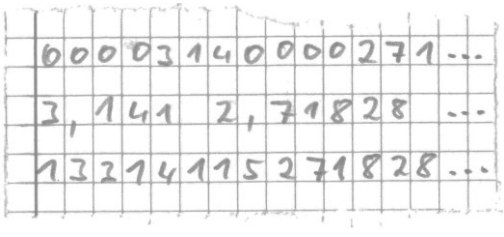
[9] Dass viele Berechnungen, die von einem Computer durchgeführt werden, nur näherungsweise richtig sind, kann man als eine Beschränkung von Computern ansehen. Allerdings kann man mit Computern Rechnungen auch symbolisch durchführen und dann sind die Ergebnisse auch genau. Solche symbolischen Rechnungen dauern jedoch in aller Regel wesentlich länger, so dass man sich meistens mit den numerischen zufriedengibt.

man Zeichen nur für die Ziffern der Zahl benötigt und den Nachteil, dass die Länge von Zahlen beschränkt wird und kurze Zahlen nicht mehr kurz, sondern genau so lang wie alle anderen sind (die freien Stellen werden mit führenden und nachfolgenden Nullen aufgefüllt).

2. Man lässt variable Stellenzahlen zu und führt neben den Ziffern noch ein Trennzeichen, das das Ende einer Zahl bzw. den Anfang der nächsten bezeichnet, sowie ein Zeichen für das Komma ein. Ein solches Trennzeichen kann beispielsweise das *Leerzeichen* sein, das einen Abstand herstellt.

3. Man stellt jede Zahl als ein Paar von Zahlen dar, von denen die erste die Länge der zweiten angibt. Allerdings unterliegt die erste Zahl (eine natürliche) dabei dem Problem, das das Zahlenpaar gerade lösen soll: Ihre Länge muss mit einem der drei hier genannten Verfahren bestimmt werden (wobei für den Anfang einer solchen Rekursion das dritte nicht in Frage kommt). Für die Festlegung der Position des Kommas braucht man dann auch noch eine dritte Zahl.

Die drei Möglichkeiten auf zwei Zahlen angewendet ergeben das folgende Bild:

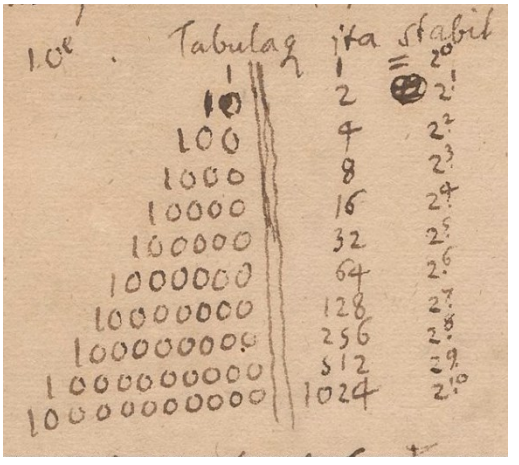


Man beachte, dass man keine der drei Schreibweisen korrekt lesen kann, wenn man nicht die jeweilige Konvention zur Festlegung der Länge von Zahlen kennt. Man beachte weiterhin, dass jeweils festgelegt sein muss, wo die erste Zahl beginnt (oben durch die vertikale Randlinie gekennzeichnet).

Alle drei Möglichkeiten kommen zur Anwendung, nicht nur bei der Darstellung von Zahlen, sondern auch bei der Codierung von Zeichen und Zeichenketten (oder Texten; s. Kapitel 4). Wo nötig werde ich im folgenden daher dazusagen, auf welche der Arten die Länge festgelegt wird. Eine vierte Möglichkeit, die Einhaltung der sog. **Fano-Bedingung**, kann in Stellenwertsystemen nicht zur Anwendung kommen: Sie würde bestimmen, dass keine Zahl der Anfang einer anderen Zahl ist, was zwar für Telefonnummern gilt (weder 1 noch 11 ist eine gültige Telefonnummer, wenn 110 eine ist, und keine andere Telefonnummer fängt dann mit 110 an, so dass man beim Lesen von 110 weiß, dass danach die nächste Telefonnummer kommen muss), aber eben nicht für Zahlen, wenn sie mithilfe eine Stellenwertsystems notiert sind.

WIKIPEDIA

2.3 Das Dualsystem



Originalschrift Leibniz' aus dem Jahr 1697.
Quelle: Gottfried Wilhelm Leibniz Bibliothek, Hannover

Es gibt 10 Arten von Menschen:
die, die das Dualsystem verstanden haben, und die, die nicht.

unbekannt

Das Stellenwertsystem, das mit den wenigsten Ziffern auskommt, ist das **Zweiersystem**, auch als **Dualsystem** bekannt. Als Ziffern des Dualsystems werden in der Regel «0» und «1» verwendet, die ja auch im Dezimalsystem verwendet werden. Die Stellenwerte der Stellen links vom Komma einer Dualzahl ergeben sich aus den Zweierpotenzen Eins (2^0), Zwei (2^1), Vier (2^2), Acht (2^3), Sechzehn (2^4) usw., Zahlen, die im Dualsystem (und ohne führende Nullen) als **1**, **10**, **100**, **1000** usw. aufgeschrieben werden. Da dies auch Zahlen im Dezimalsystem sind (wobei sie hier den Zahlen Eins, Zehn, Hundert, Tausend usw. entsprechen), kann man einer Ziffernfolge ohne Angabe des Zahlensystems keine Zahl zuordnen. Genau das ist die Grundlage obigen Witzes zu den 10 Arten von Menschen: Die meisten Menschen werden 10 als Zehn lesen — nur wer das Dualsystem verstanden hat erkennt, dass hier 10 als Zwei gelesen werden muss.

Der besondere Reiz des Dualsystems ist, dass es mit zwei Ziffern alle Zahlen ausdrücken kann, die man auch mit Stellenwertsystemen mit mehr Ziffern ausdrücken kann. Neben dem eher Esoterischen eines Reizes hat dies den ganz praktischen Vorteil, dass man bei der Mechanisierung des Rechnens mit Dualzahlen nur zwei verschiedene Zustände unterscheiden muss, was u. a. die Gefahr der Verwechselung von Zeichen (etwa durch technische Störeinflüsse) verringert. Dies hatte auch schon *Leibniz* für die Konstruktion seiner ersten fehlerfrei funktionierenden Rechenmaschine ausgenutzt; seine vorausgegangenen Versuche, eine solche auf Basis des Dezimalsystems zu bauen, scheiterten nämlich am bisweilen fehlerhaften Zehnerübertrag an einer der höheren Stellen (ein mechanisches Problem). Allerdings braucht man, wie man leicht erkennen kann, im Dualsystem für die Darstellung einer Zahl größer

Eins mehr Stellen als in jedem anderen Stellenwertsystem. Da die Behandlung einzelner Stellen jedoch nicht von der Anzahl der Stellen abhängt, ist das aber kein Problem — man braucht dafür nur *mehr vom Selben*.

Bits und Bitfolgen

Die Stellen von Zahlen im Dualsystem heißen englisch „binary digits", zusammengefasst zu „bits" oder, deutsch, „**Bits**". Eine Dualzahl mit vier Bits hat demnach vier Stellen. Im Gebrauch hat sich die Bedeutung von Bits von Zahlen losgelöst: So bezeichnet „*n* Bits" eine Folge von *n* Stellen, an denen «1» oder «0» stehen kann, und zwar unabhängig davon, ob diese **Bitfolge** eine Dualzahl oder etwas anderes repräsentiert. Eine Folge von acht Bit wird zu einem **Byte** zusammengefasst; allgemeiner bezeichnet man Zusammenfassungen einer Folge von mehreren Bits als „**Wort**", wobei die Anzahl der Bits, anders als beim Byte, beim Wort nicht normiert ist (s. dazu auch Abschnitt 11.4.3 in Kurseinheit 2). Der Vollständigkeit halber sei noch erwähnt, dass Bit auch die Größe *Informationsgehalt* bezeichnet (deren Einheit *bit* ist).

WIKIPEDIA

WIKIPEDIA

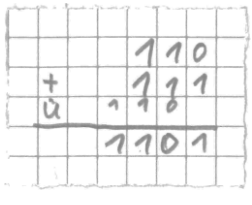
2.4 Rechnen mit Dualzahlen

Kurs

Solange Zahlen nur dastehen (auf dem Papier oder im Speicher eines Computers), haben sie keine Bedeutung — ihre Ziffern könnten für alles Mögliche stehen. Ihre Bedeutung ergibt sich daraus, was man (oder ein Computer) damit macht, nämlich rechnen. Wie bereits oben nahegelegt ist dieses Rechnen nicht notwendigerweise ein intellektueller Akt, sondern kann als ein Spiel mit Zeichen aufgefasst werden.

2.4.1 Addition

Wie aber addiert man zwei Zahlen im Dualsystem? Genauso wie im Dezimalsystem! Konkret schreibt man für die Berechnung einer Summe die beiden Summanden komma- bzw. rechtsbündig untereinander auf und addiert dann deren Stellen einzeln, von rechts nach links, wobei man ab der zweiten Stelle noch den *Übertrag* aus der Addition der vorigen Stelle berücksichtigen muss. Die Addition **110 + 111** zum Beispiel ergibt



(Man beachte, dass zur korrekten Darstellung des Ergebnisses vier Stellen benötigt werden, obwohl die beiden Summanden nur dreistellig sind.) Dabei ist die Addition im Dualsystem einfacher als die im Dezimalsystem, da man nur wissen muss, was die elementaren Additionen **0 + 0**, **0 + 1**, **1 + 0** und **1 + 1** ergeben. Dafür muss man nicht einmal im Kopf rechnen, sondern kann die Ergebnisse der nachfolgenden *Additionstabelle* entnehmen:

1. Summand	2. Summand	Summe
0	0	00
0	1	01
1	0	01
1	1	10

Wenn man beispielsweise wissen will, wie viel **1 + 0** ergibt, sucht man die Zeile, in der unter „1. Summand“ «1» und unter „2. Summand“ «0» steht (die dritte) und liest dann in derselben Zeile rechts das Ergebnis («0»«1») ab. Hierbei ist die zweite (linke) Stelle der Summe der *Übertrag*, der bei der Addition der zweiten Stellen der beiden Summanden berücksichtigt werden muss. Für die zweite Stelle kommt deshalb die Additionstabelle

1. Summand	2. Summand	Übertrag	Summe
0	0	0	00
0	1	0	01
1	0	0	01
1	1	0	10
0	0	1	01
0	1	1	10
1	0	1	10
1	1	1	11

zur Anwendung, die neben den Stellen der beiden Summanden noch den Übertrag aus einer vorherigen Addition berücksichtigt. Man beachte, dass die Summe dreier einstelliger Zahlen im Dualsystem nicht größer als **11** (Drei) sein kann, so dass man für die Summe mit einer Stelle und dem Übertrag in die nächste Stelle auskommt. Insbesondere hat der Übertrag bei der stellenweisen Addition von zwei Zahlen immer nur eine Stelle, so dass sich dieselbe Tabelle auch für die Addition der dritten und aller weiteren Stellen verwenden lässt. Man kommt also, um zwei Dualzahlen beliebiger Länge addieren zu können, mit nur zwei Tabellen aus (wobei die erste auch noch von der zweiten subsumiert wird — man muss hierfür nur den Summanden „Übertrag“ konstant auf «0» setzen). Regelmäßigkeiten dieser Art machen eine wesentliche Grundlage für die rasante Entwicklung der Digitalisierung aus: Viele komplexe Aufgaben lassen sich auf die stumpfe Wiederholung einfacher Aufgaben zurückführen. Für *Hardware*, die die Zeichenspiele umsetzt (s. Kapitel 11 in Kurseinheit 2), heißt das: Um sie leistungsfähiger zu machen genügt es häufig, einfach nur *mehr vom Selben* hinzuzufügen.

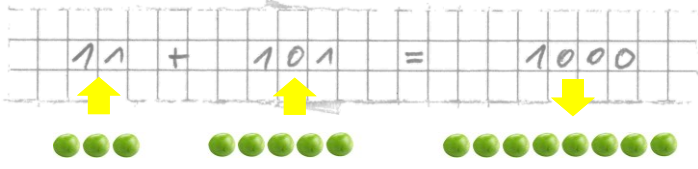
Überlauf

Dem aufmerksamen Leser wird aufgefallen sein, dass bei der Addition der dritten Stelle im obigen Beispiel ein Übertrag in die vierte Stelle auftritt, an der aber bei den

beiden Summanden gar nichts steht. Streng genommen kann man also auch keine der beiden obigen Tabellen für die Bestimmung der vierten Ziffer verwenden. Wenn wir allerdings die Länge (Stellenzahl) von Zahlen nicht auf drei begrenzen wollen (vgl. Abschnitt 2.2), dann können wir hier einfach führende Nullen annehmen, so dass wir der zweiten Tabelle das korrekte Ergebnis **1** (**= 0 + 0 + 1**) entnehmen können. Wenn wir hingegen die Zahl der Stellen der Zahlen auf drei begrenzen, dann reicht dies für die Addition von **110** und **111** nicht aus: Wenn es keine vierte Stelle gibt, kann man auch den Übertrag in die vierte Stelle nicht verarbeiten. Während das bei einem Übertrag von «0» kein Problem ist (man kann ihn einfach ignorieren), kennzeichnet ein Übertrag von «1» nach der dritten ziffernweisen Addition einen sog. **Überlauf**; er zeigt an, dass das nur dreistellige Ergebnis (im obigen Beispiel: **101**) nicht korrekt wäre. Überläufe sind ein fundamentales Problem des Rechnens mit endlicher Stellenzahl; obwohl es so offensichtlich ist, führt es immer wieder zu folgenschweren Fehlleistungen (deren spektakulärste wohl der gescheiterte Erstflug der Ariane 5 im Jahr 1996 war).

Addition als reines Zeichenspiel

Bemerkenswert an obiger Darstellung der Addition von Dualzahlen ist, dass sie sich als ein *reines Zeichenspiel* auffassen lässt (oder wo wurde oben „gerechnet"?), sie also nach feststehenden Regeln vollständig mechanisch ablaufen kann. Insbesondere funktioniert sie, ohne dass eine ausführende Maschine die Bedeutung von natürlichen Zahlen (als Anzahl von etwas) kennen oder gar Striche abzählen müsste (wie Kinder zu addieren lernen, was ja auch funktionieren würde und obendrein der natürlichen Bedeutung von Zahlen näherkäme). Trotzdem ist das Ergebnis dieses Zeichenspiels stets richtig und lässt sich bedenkenlos auf reale Sachverhalte übertragen:



Und so funktioniert das immer beim Einsatz von Computern: Ein Sachverhalt wird als Zeichenfolge codiert, auf der Zeichenspiele ausgeführt werden, deren Ergebnis dann für den Sachverhalt eine (idealerweise: die richtige) Bedeutung hat.

Voraussetzung ist natürlich, dass sich jemand mal die Mühe gemacht hat, die Gültigkeit des Zeichenspiels zu beweisen. Beispiele für solch einen Beweis, allerdings auf dem Gebiet der Logik, finden Sie in Abschnitt 3.2.2. Hier sei noch erwähnt, dass viele mathematische Beweise selbst die Form eines Zeichenspiels haben. Ein besonders schönes Beispiel hierfür ist der Diagonalisierungsbeweis für die Existenz überabzählbar vieler Zahlen.

WIKIPEDIA

Das Zeichenspiel zur Addition wird in seiner konkreten Ausführung übrigens dadurch erleichtert, dass wir die Zahlen (Summanden, Übertrag und Summe) untereinander aufschreiben (und damit die zweite Dimension des Papiers bemühen); es ließe sich auch für eine eindimensionale Darstellung formulieren, würde dadurch aber bedeutend umständlicher,

insbesondere, wenn man nicht verlangt, dass alle Zahlen dieselbe Stellenzahl haben (s. Abschnitt 2.2). Und so sind die Stellen in einem Computer, an denen eine Eins oder eine Null stehen kann, auch zweidimensional organisiert (s. Abschnitt 11.4.1 in Kurseinheit 2).

2.4.2 Multiplikation

Die Multiplikation zweier ganzer Zahlen im Dualsystem kann man, genau wie im Dezimalsystem, als wiederholte Addition auffassen: So ist beispielsweise $11 \cdot 10 = 10 + 10 + 10$. Wenn man also ganze Zahlen addieren kann, kann man sie auch multiplizieren — man muss dazu lediglich die eine Zahl, den Multiplikanden, so oft zum Ergebnis der vorherigen Addition (bzw. zu **0** am Anfang der Wiederholung) addieren, wie es die andere Zahl, der Multiplikator, vorgibt. Dieses Zeichenspiel, das die Multiplikation auf die Addition (die ja ebenfalls als ein Zeichenspiel erledigt werden kann) zurückführt, funktioniert natürlich nur, wenn der Multiplikator eine ganze Zahl ist.

Multiplikation mit Zweierpotenzen

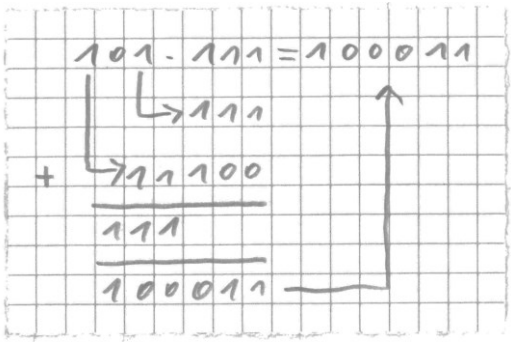
Multiplikation als iterative Addition ist zwar einfach, dauert aber u. U. (wenn dafür viele Additionen ausgeführt werden müssen) recht lange. Das Verfahren lässt sich beschleunigen, indem man ausnutzt, dass die Multiplikation mit Potenzen von Zwei im Dualsystem der Multiplikation mit Potenzen von Zehn im Dezimalsystem entspricht: für eine Multiplikation mit Zwei (**10** im Dualsystem) hängt man eine Null an, für eine mit Vier (**100**) zwei usw. Dieses Anhängen einer Null entspricht einer Linksverschiebung aller Stellen, die eine Zahl darstellen, und dem Eintragen von «0» an der freiwerdenden, ganz rechten Stelle. So wird aus **11** (Drei) durch Multiplikation mit **10** (Zwei) **110** (Sechs), durch Multiplikation mit **100** (Vier) **1100** (Zwölf) usw. Die offensichtliche Beschränkung ist, dass damit nur die Multiplikation mit Zweierpotenzen abgedeckt wird.

Multiplikation mit beliebigen ganzen Zahlen

Für die Multiplikation mit beliebigen ganzen Zahlen lässt sich jedoch das Distributivgesetz der Arithmetik ausnutzen. Es besagt u. a., dass für beliebige Zahlen a, b und c stets gilt: $(a + b) \cdot c = a \cdot c + b \cdot c$. Damit ist beispielsweise

$$101 \cdot 111 = (100 + 1) \cdot 111 = 100 \cdot 111 + 1 \cdot 111 = 11100 + 111$$

Die Multiplikation von **111** (sieben) mit **101** (fünf) lässt sich also auf die Multiplikation von **111** mit **100** (vier) und mit **1** (eins; also mit zwei Zweierpotenzen) und eine Addition der beiden so erhaltenen Produkte zurückführen. Diese Zurückführung ist Grundlage eines Zeichenspiels zur (schriftlichen) Multiplikation:



Dass dieses Zeichenspiel den richtigen Wert liefert folgt aus der Gültigkeit der schriftlichen Multiplikation mit Zweierpotenzen (durch Anhängen von Nullen), der Gültigkeit der schriftlichen Addition und der Gültigkeit des Distributivgesetzes[10]. Allgemein lässt sich die Multiplikation mit einer beliebigen *n*-stelligen ganzen Zahl auf maximal *n* - 1 Multiplikationen mit Zweierpotenzen ungleich **1** und maximal *n* - 1 Additionen zurückführen (maximal deswegen, weil sich Multiplikationen und Additionen für Stellen des Multiplikators, an denen «0» steht, sparen lassen). Wem das immer noch zu viele Additionen sind, der kann auch auf eine Tabelle zurückgreifen, in der die Produkte gespeichert sind:

Multiplikation		Multiplikand				
		0000	0001	0010	0011	..
Multiplikator	0000	0000	0000	0000	0000	..
	0001	0000	0001	0010	0011	..
	0010	0000	0010	0100	0110	..
	0011	0000	0011	0110	1001	..
	…	…	…	…	…	..

Diese *Multiplikationstabelle* entspricht einem großen Einmaleins für Dualzahlen und das „Rechnen" durch Nachschlagen dem, was Menschen machen, wenn sie das auswendig gelernte Einmaleins anwenden. Allerdings wird die gesparte Rechenzeit hier mit dem für die Tabelle benötigten Platz erkauft. Dieser sog. *Time-memory* (oder auch *Space-time*) *trade-off* ist bei vielen Digitalisierungsproblemen zu treffen.

2.4.3 Subtraktion und negative Dualzahlen

Mit der Einführung der Subtraktion verlässt man den Raum der natürlichen Zahlen, da man für die Darstellung eines Teils der Ergebnisse auch negative Zahlen benötigt. Negative Zahlen lassen sich aber mit den bisher beschriebenen Mitteln eines Stellenwertsystems gar nicht darstellen — es gibt weder negative Ziffern noch negative Stellen. Die erste Herausforderung ist also die Darstellung negativer Zahlen; die zweite ist dann, damit zu rechnen.

---

[10] wobei die Anwendung des Distributivgesetzes (bzw. die damit einhergehende Termumformung) selbst als ein Zeichenspiel aufgefasst werden kann, dass jedoch nur der Begründung der schriftlichen Multiplikation dient und bei dieser selbst gar nicht mehr angewendet werden muss

In der üblichen Schreibweise für Zahlen verwendet man zur Kennzeichnung negativer Zahlen das negative Vorzeichen «-», also beispielsweise **–1** für minus Eins. Nun ist das Vorzeichen ein Zeichen und wenn man nur zwei Zeichen zur Verfügung hat, muss man sich etwas einfallen lassen. Man nennt einen solchen Einfall, wenn er verabredet ist, eine Konvention.

Vorzeichen

Die Konvention, die der Verwendung eines Vorzeichens am nächsten kommt, ist die Einführung eines **Vorzeichenbits**. Dieses Bit hätte dann beispielsweise den Wert **0** für eine positive Zahl und den Wert **1** für eine negative Zahl. Es stünde außerhalb des Stellenwertsystems und würde einer Zahl zugeordnet (eine Zahl wäre also immer ein Paar bestehend aus einer positiven Zahl und einem Vorzeichenbit, das aus einer positiven ggf. eine negative Zahl macht). Dieses Vorzeichenbit wäre dann bei Additionen und Multiplikationen zu berücksichtigen, was aber eine Anpassung der Addier- und Multiplizierverfahren bedeuten würde.

Stattdessen verwendet man meist die sog. **Komplementdarstellung**, nach der man für eine negative Zahl alle Bits der entsprechenden positiven Zahl invertiert (also aus «1» «0» macht und umgekehrt) und, aus rechnerischen Gründen, noch die Zahl **1** addiert. Die Zahl Drei würde demnach, bei Verwendung von 4 Bit, wie gehabt als **0011**, die Zahl minus Drei als **1101** (**= 1100 + 0001**) dargestellt. Man erkennt hier negative Zahlen an der «1» an der ganz linken Stelle (wodurch dieses Bit den Charakter eines Vorzeichenbits hat); die Zahl **–0** gibt es demnach nicht (**1111 + 0001 = 0000** mit Übertrag **1**). Der (wenig offensichtliche) Vorteil dieser Darstellung ist, dass man so dargestellte negative Zahlen nach demselben Verfahren wie positive addieren kann und dabei das richtige Ergebnis herauskommt: **0011 + 1101 = 0000** (wobei der Übertrag, hier **1**, unter den Tisch fällt).

Komplementdarstellung

Die Behandlung negativer Zahlen müssen Sie sich nicht merken — wie in anderen Fällen auch handelt es sich hier um die Ausnutzung bestimmter Gesetzmäßigkeiten mit dem Ziel, mit nur zwei Zeichen auf möglichst einfache Weise so viel wie eben möglich machen zu können. Wichtig ist, dass Sie erkennen, dass einmal mehr nur mit Zeichen nach bestimmten Regeln gespielt werden muss, dass also keinerlei mathematische Fähigkeiten (was auch immer sonst das sein mag) zum Rechnen benötigt werden.

2.4.4 Division

Die Division $a \div b$ lässt sich, als Umkehrung der Multiplikation, mit ähnlichen Mitteln durchführen: per wiederholter Subtraktion (wie oft steckt b in a, wobei hier die Anzahl der Subtraktionen zur Bestimmung des Ergebnisses gezählt werden muss), per Rechtsverschiebung der Stellen von a (entsprechend der Division durch Zwei) und per Nachschlagen in einer *Divisionstabelle*.[11] Ähnlich wie bei der Subtraktion den Bereich der natürlichen (positiven) kann man hier jedoch den Bereich der ganzen Zahlen verlassen, nämlich wenn a sich nicht

[11] Eine fehlerhafte Divisionstabelle war übrigens auch Ursache des sog. FDIV-Bugs der Intel-Pentium-Prozessoren, der in den 1990er Jahren für reichlich Spott sorgte („intel inside — can't divide").

ganz durch ***b*** teilen lässt, also entweder ein Rest oder das Ergebnis als Bruch (rationale Zahl; s. Abschnitt 2.6) ausgewiesen werden muss.

2.4.5 Vergleich von zwei Zahlen

Zwei Zahlen können gleich sein; sind sie das nicht, ist eine größer als die andere. Wie aber bestimmt man, ob zwei Zahlen gleich sind oder eine Zahl größer ist als eine andere? In der Grundschule lernt man dazu ein einfaches Zeichenspiel: Man vergleicht, beginnend mit der höchsten Stelle, stellenweise die Ziffern der beiden Zahlen — ist eine Ziffer größer als die andere (ggf. eine führende Null), ist deren Zahl größer, sind sie gleich, macht man mit der nächsthöheren Stelle genauso weiter. Auf Dualzahlen angewendet ist dieses Verfahren besonders einfach, da man hier nur die erste Stelle (von links gesehen) finden muss, an der bei der einen Zahl eine 1 und der anderen eine 0 steht. Allerdings funktioniert dies nur bei einer Beschränkung auf positive Zahlen in Festkommadarstellung (s. Abschnitt 2.6.2).

Ein anderes Verfahren, das immer funktioniert, ist, die eine der zu vergleichenden Zahlen von der anderen zu subtrahieren. Ist das Ergebnis 0, sind die Zahlen gleich, ist es negativ, war die zweite Zahl größer als die erste, ist es positiv, ist es umgekehrt. Voraussetzung ist natürlich, dass man ein zuverlässig funktionierendes Zeichenspiel für die Subtraktion zur Verfügung hat.

2.5 Andere Stellenwertsysteme

Grundsätzlich eignet sich jede ganze Zahl größer als **1** als Basis eines Stellenwertsystems. Allerdings steigt mit der Basis die Anzahl der Zeichen, die man als Ziffern benötigt. Wenn man bedenkt, dass wir mit weniger als 30 verschiedenen Zeichen (Buchstaben) für unsere Schrift auskommen (und dafür lange Wörter in Kauf nehmen), scheint es keinen praktischen Grund zu geben, höhere Basen als Zehn für unser Stellenwertsystem zu bevorzugen.[12]

Im Kontext der Digitalisierung hat dennoch ein weiteres Stellenwertsystem eine gewisse Bedeutung erlangt: das **Sechzehnersystem**, auch **Hexadezimalsystem** genannt. Dies liegt daran, dass eine Stelle im Sechzehnersystem genau vier Stellen im Dualsystem entspricht: Zu jeder Ziffer des Sechzehnersystems (mit den Werten Null bis Fünfzehn, dargestellt durch «0»–«9» und die Buchstaben «A»–«F» für die übrigen sechs Ziffern) gehört genau eine vierstellige Dualzahl (**0000**–**1111**). Man kann also leicht Dualzahlen in **Hexadezimalzahlen** umwandeln und der Wert eines Byte, eine achtstellige Dualzahl, lässt sich abkürzend als eine zweistellige Hexadezimalzahl darstellen. Allerdings wird Ihnen die Hexadezimalschreibweise von Zahlen nur selten begegnen; tatsächlich werden selbst die 4 · 8 Bit = 32 Bit einer Internetadresse seltener als Folge von vier zweistelligen Hexadezimalzahlen als von vier bis zu dreistelligen Dezimalzahlen dargestellt (also etwa 255.0.0.13 statt FF.00.00.0D).

---

[12] Allerdings hat das *Zwölfersystem* in der Praxis einige Vorteile gegenüber dem Dezimalsystem. Wenn Rechenkünste in der Evolution eine Rolle gespielt hätten, hätten wir wohl zwölf Finger.

2.6 Andere als ganze Zahlen

2.6.1 Rationale Zahlen

Nach den ganzen Zahlen kommen die rationalen, die als Brüche von zwei ganzen Zahlen (oder als Paar zweier ganzer Zahlen, dem Zähler und dem Nenner) dargestellt werden können. Beim Rechnen mit Brüchen sind die Regeln der Bruchrechnung zu befolgen, die jedoch auch nur Regeln eines Spiels mit Zeichen sind (das nächste Level sozusagen). Sofern die Anzahl der Stellen von Zähler und Nenner unbegrenzt sind und sofern man lediglich die vier Grundrechenarten (Addition, Subtraktion, Multiplikation und Division) braucht, sind der Genauigkeit des Rechnens mit Brüchen keine Grenzen gesetzt. Allerdings werden Zähler und Nenner, wenn man nicht regelmäßig kürzt, schnell sehr groß, so dass man lange Zahlen benötigt (s. Abschnitt 2.2 für die damit einhergehenden Probleme). Das Kürzen ist zwar auch als Zeichenspiel recht leicht durchzuführen (nach dem sog. *euklidschen Algorithmus*), aber relativ aufwendig, so dass man in aller Regel, unter Aufgabe der Genauigkeit, auf Gleitkommazahlen (Abschnitt 2.6.3) ausweicht. Dennoch sollte man die Verwendung von Brüchen in Betracht ziehen, wenn Genauigkeit verlangt ist und nur die vier Grundrechenarten benötigt werden.

WIKIPEDIA

2.6.2 Dezimalbrüche und Festkommazahlen

Wenn man sich auf die Anzahl der Nachkommastellen festlegt (sog. **Festkommazahlen**), können Kommazahlen genau wie Ganzzahlen behandelt werden. Im Dezimalsystem spricht man dann auch von **Dezimalbrüchen**: Eine Zahl mit zwei Nachkommastellen kann als die Ganzzahl, die man durch Streichung des Kommas erhält (nur das Komma, nicht die Nachkommastellen!) geteilt durch 100 darstellen. So ist beispielsweise

$$3{,}33 = \frac{333}{100}$$

Dezimalbrüche lassen sich leicht addieren und subtrahieren, wenn ihr Nenner gleich ist (also wenn sie als Festkommazahlen die gleiche Anzahl von Nachkommastellen haben); bei der Multiplikation und der Division erhöht sich der Nenner jedoch in aller Regel („Zähler mal Zähler, Nenner mal Nenner“ bzw. „Zähler mal Nenner, Nenner mal Zähler“ — zwei weitere Zeichenspiele) und damit auch die Zahl der Nachkommastellen. Um hier die Nachkommastellenzahl konstant zu halten, muss ggf. gerundet werden.

Dezimalbrüche im Dualsystem

Bei der Inbetrachtnahme von Dezimalbrüchen muss man bedenken, dass die Nachkommastellen im Dualsystem eine andere Wertigkeit haben: So steht **0,1** für $0 \cdot 2^0 + 1 \cdot 2^{-1}$, also für ein Halbes und nicht für ein Zehntel. Tatsächlich ist die Darstellung der Dezimalzahl **0,1** im Dualsystem periodisch, hat also eine unendliche Anzahl von Nachkommastellen (so wie beispielsweise ein Drittel im Dezimalsystem; s. Abschnitt 2.1). Man kann daher Dezimalbrüche im Dualsystem i. Allg. nur mit begrenzter Genauigkeit darstellen.

binär codierte Dezimalzahlen

Nun gibt es aber durchaus Anwendungen, in denen Genauigkeit auch nach dem Komma eine große Rolle spielt. Bestes Beispiel hierfür sind *Registrierkassen*, eine der ersten weiten Verbreitungen von Computern. Um die oben beschriebenen Probleme der Darstellung von Dezimalzahlen zu vermeiden, hat man die sog. **binär codierten Dezimalzahlen** (engl. binary coded decimals, **BCD**) eingeführt, in denen Dezimalzahlen ziffernweise codiert werden, also für jede Stelle einer Dezimalzahl vier binäre Stellen reserviert werden. Die BCD-Repräsentation von Zahlen wird heute jedoch kaum mehr herangezogen (außer in Taschenrechnern vielleicht); stattdessen werden, wo Dezimalbrüche verlangt werden, Gleitkommazahlen mit der Basis Zehn verwendet.

internationales Durcheinander

Übrigens: Das deutsche Dezimalkomma ist im englischen Sprachraum ein Dezimalpunkt, und der deutsche Dezimalpunkt (zum Trennen von Tausendern verwendet) ist im Englischen das Komma. Zwar versuchen *Betriebssysteme* (Kapitel 12 in Kurseinheit 2), diese Unterschiede auszugleichen, indem sie eine automatische Anpassung der Ein- und Ausgabe von Zahlen an den eingestellten Sprachraum anbieten, doch machen *Programmiersprachen* (Kapitel 20 in Kurseinheit 3) bei der Darstellung von (konstanten) Zahlen in Programmen diese Unterscheidung in aller Regel nicht und Verwechselungen von Dezimalpunkt und -komma sind eine nicht zu vernachlässigende Fehlerquelle der Programmierung und Nutzung von Computern.

2.6.3 Gleitkommazahlen

Gleitkommazahlen (auch **Fließkommazahlen** genannt) erlauben die Darstellung von sehr großen und sehr kleinen Zahlen mit einer relativ kleinen, festen Anzahl von Stellen. Ist bei langen Zahlen die Zahl der zu ihrer Darstellung zur Verfügung stehenden Stellen erschöpft, werden die Stellen großer Zahlen so lange nach rechts und die Stellen kleiner Zahlen so lange nach links verschoben, bis sie genau eine Stelle vor dem Komma haben; die Nachkommastellen werden dann entsprechend der zur Verfügung stehenden Stellenzahl abgeschnitten (ggf. nach einer Rundung). Dabei wird die Anzahl der durchgeführten Verschiebungen gezählt und der so entstandenen Zahl beigeordnet. Sie kennen solche Zahlenpaare vielleicht von Ihrem Taschenrechner: **1,0E9** (mit «E» als Trennzeichen) beispielsweise bedeutet hier **1.000.000.000** (eine Eins mit neun Nullen) und **1,1E-2** bedeutet **0,011**. Der Preis ist natürlich ein Genauigkeitsfehler: **1.000.000.000,011** lässt sich als Gleitkommazahl nicht exakt darstellen, wenn man weniger als 13 Stellen zur Verfügung hat. Dieser Fehler vergrößert sich beim Rechnen mit Gleitkommazahlen und die mangelnde Berücksichtigung von Rundungsfehlern bei der Gleitkommaarithmetik ist ein häufiger Programmierfehler (ein konkretes Beispiel hierfür finden Sie in Abschnitt 22.1.3 in Kurseinheit 3). Gleichwohl ist die Gleichkommaarithmetik wichtiger Bestandteil gerade des wissenschaftlichen Rechnens und

wird daher auf jedem heutigen Computer per *Hardware* unterstützt. Man sollte jedoch immer die Möglichkeit in Betracht ziehen, dass ein berechnetes Ergebnis zu ungenau ist, um verwertbar zu sein.[13]

2.6.4 Zahlen, die keine sind

Manchmal werden zu den Zahlen aus solche hinzugerechnet, die definitionsgemäß keine sind. Dazu zählen minus und plus Unendlich sowie das Ergebnis undefinierter Rechenoperationen wie die Division durch **0** (manchmal *NaN*, für *Not a Number*, genannt). Erstere können in Zeichenspielen Verwendung finden, in denen zwei Zahlen verglichen werden sollen (jede echte Zahl ist größer als minus Unendlich und kleiner als plus Unendlich); letztere erlaubt, Rechnungen fortzusetzen, die eigentlich abgebrochen werden müssten (eben weil eine undefinierte Rechenoperation darin vorkommt, man also wohl einen Fehler gemacht hat). Man sollte sich davor hüten, zu glauben, dass mit Einführung dieser „Sonderzahlen" die dahinterstehenden Probleme aus der Welt geschafft wurden — das Auftreten von NaN als Ergebnis einer Rechnung verlangt immer eine **Ausnahmebehandlung**, also eine Festlegung darauf, was in diesem Fall zu tun ist.

2.7 Inkrement, Dekrement und die zeitliche Dimension

Zwei Rechenoperationen, die in der Digitalisierung eine besondere Rolle spielen, sind die Addition und Subtraktion von **1**, auch als **Inkrement** und **Dekrement** bezeichnet. Sie werden zum Zählen, genauer zum Ab- und Aufzählen, verwendet, und geben damit den Zahlen die Bedeutung von *Kardinalia* und *Ordinalia*. Entsprechend können beide Operationen auch nur auf ganzen Zahlen angewendet werden.

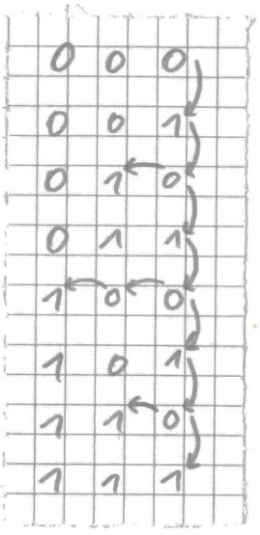
Dabei ist es günstig, dass sich das Zählen im Dualsystem als besonders einfaches Zeichenspiel realisieren lässt: Man wechselt dazu an jeder Stelle einfach nur zwischen «0» und «1» hin und her, und zwar

- an der ersten Stelle immer für die jeweils nächste Zahl und
- an der zweiten sowie allen weiteren Stellen immer dann, wenn die rechte Nachbarstelle auf «0» wechselt.

---

[13] Besonders gemein ist, dass man den Zahlen allein nicht ansieht, wie ungenau sie sind — das hängt nämlich auch von ihrer Herkunft ab. So liegt der Fehler bei Ergebnissen von Rechenoperationen nicht nur in den weggelassenen Ziffern, sondern „wandert" durch Rechenoperationen auch in die dargestellten. So ist eine Gleitkommazahl mit fünf Nachkommastellen eben nicht unbedingt „auf fünf Stellen genau", wenn sie das Ergebnis einer Berechnung ist. Als Faustregel kann dienen, dass eine Zahl immer ungenauer wird, je mehr Schritte ihre Berechnung benötigte. Das (mechanisierte) Rechnen mit Gleitkommazahlen ist eine Wissenschaft für sich, in der immer noch aktiv geforscht wird.

Die ersten sieben Schritte dieses Spiels sind nachfolgend dargestellt (wobei die Zahlen die Horizontale und die Zählfolge die Vertikale belegen):



Beim achten Schritt würde die erste Stelle wieder auf «0» wechseln und mit ihr die zweite und auch die dritte Stelle. Sofern keine vierte Stelle vorgesehen ist, beginnt das Zählen damit wieder bei **000**. Den Wechsel der dritten Stelle auf «0» kann man dann auch als *Überlauf* beim Zählen interpretieren (vgl. Abschnitt 2.4.1). Man beachte, dass das Zählen so nicht nur vollkommen mechanisch, sondern auch ohne die explizite Addition von **1** (ohne Verwendung des in Abschnitt 2.4.1 dargestellten Zeichenspiels zur Addition) funktioniert. So ähnlich arbeiten übrigens auch mechanische Zähler im Dezimalsystem, wie man sie beispielsweise als Kilometerzähler kennt — diese können auch nicht rechnen (und kennen im übrigen keinen Überlauf).

Selbsttest: Spezifizieren Sie das Zeichenspiel für das Dekrement.

Zähler und Behälter

Inkrement und Dekrement geben aber Zahlen nicht nur eine besondere Bedeutung (wenn man eine Zahl inkrementiert oder dekrementiert, dann verwendet man sie wohl zum Ab- oder Aufzählen) — sie führen auch das Konzept des **Zählers** ein. Ein Zähler ist keine Zahl, sondern ein **Behälter** für eine Zahl, dessen Inhalt, eben eine (ganze) Zahl, durch Inkrement und Dekrement verändert wird. Mathematisch kann man diesen Behälter am ehesten mit einer *Variable* vergleichen, die ja auch für eine Zahl steht; es gibt aber fundamentale Unterschiede zwischen dem Konzept eines Behälters und dem einer mathematischen Variable, so dass man diesen Vergleich mit Vorsicht genießen muss. Insbesondere steht ein Behälter im Allgemeinen sowie ein Zähler im Besonderen für zeitliche Veränderbarkeit seines Inhaltes bzw. Wertes, während eine mathematische Variable für *einen*, wenn auch unbekannten, Wert steht (weswegen sie auch *Unbekannte* genannt wird) und unterschiedliche Werte alternative Werte sind, ohne dass die Zeit hier irgendeine Rolle spielen würde. Wenn also Z eine mathematische Variable ist, die einen Zähler repräsentieren soll, dann wird das Inkrement mathematisch korrekt durch die Gleichung $Z' = Z + 1$ wiedergegeben, wobei Z' hier eine andere Variable ist, die für den neuen Wert des Zählers steht. Tatsächlich würde man einen Zähler mathematisch durch eine Folge von (indizierten) Variablen

beschreiben: $Z_{n+1} = Z_n + 1$. Die Zeit ist kein Modus der Mathematik — von Computern hingegen schon (vgl. Abschnitte 11.2.2 und 18.2 in Kurseinheit 2).

zeitliche Dimension

Tatsächlich hält mit der Betrachtung von Behältern die *zeitliche Dimension* Einzug in unsere Zeichenspiele: Während wir bei den vorherigen Zeichenspielen lediglich Zeit benötigten, um sie durchzuführen, die Zeit aber keinerlei Einfluss auf das Ergebnis hatte, gibt es jetzt ein **Vorher** und ein **Nachher**: Der Behälter hat vorher einen anderen Inhalt als nachher. Auf einem Blatt Papier brauchen wir dazu erstmals das Radiergummi:

An den Stellen, an denen wir den alten Zählerstand notiert hatten, steht nach dem Inkrement bzw. Dekrement der neue Zählerstand. Ist dieser Zählerstand Teil einer Berechnung gewesen, unterliegt auch diese Berechnung der Zeit: Erneut durchgeführt liefert sie ein anderes Ergebnis. Wenn das aus mathematischer Sicht auch nach einem Problem aussehen mag, so ist es doch Grundlage aller Computer (und auch vieler *Algorithmen*, die auf ihnen ausgeführt werden).

Größe von Behältern

Übrigens: So, wie Zahlen eine Länge haben (Abschnitt 2.2), haben Behälter eine Größe, die bestimmt, wie lang die Zahlen, die sie aufnehmen können, höchstens sein dürfen. Dies ist schon deswegen notwendig, weil man mehrere Behälter haben kann, die, wenn sie hintereinander angeordnet sind, sich nicht überlappen dürfen, selbst dann nicht, wenn die Länge einer darin gespeicherten Zahl (etwa beim Zählen) wächst. Über die Größe eines Behälters hinauszuschreiben ist ein schwerwiegender Fehler, der u. a. dazu führt, dass Zeichenspiele nicht korrekt funktionieren (s. dazu auch Abschnitt 27.2.3 in Kurseinheit 4)

Identifikation von Behältern

Man kann sich einen Behälter wie eine (reservierte) Folge von Kästchen (Karos) auf einem karierten Blatt Papier vorstellen, in denen geschrieben und radiert werden darf und die durch die Position auf dem Blatt Papier (und nicht ihren Inhalt) identifiziert werden. Die Position eines Behälters kann durch ein Koordinatensystem oder eine einfache Nummerierung aller Kästchen bestimmt werden; man kann den Behältern aber auch Namen geben. Die Namen können dann verwendet werden, um in Rechnungen auf den Inhalt der Behälter zuzugreifen, ganz so, wie die Namen von Variablen in einer Gleichung für ihren Wert stehen.

3 Wahrheitswerte

Was die Zahlen für die Mathematik sind, sind *Wahr* und *Falsch* für die **Logik**: Werte, mit denen man nach bestimmten Regeln „rechnen“ kann. Genau wie Zahlen sind die **Wahrheitswerte** *Wahr* und *Falsch* für sich genommen vollkommen abstrakt — der Bezug zu

WIKIPEDIA

Konkretem kann hergestellt werden, indem man sie *Aussagen* zuordnet, die dadurch als wahr oder falsch ausgezeichnet werden. Die „Rechenregeln" für *Wahr* und *Falsch* werden davon, welche Aussagen wahr oder falsch sind, nicht beeinflusst — genau wie bei den Zahlen sind die Rechenregeln von einer konkreten Verwendung unabhängig. Ihre Bedeutung für die Praxis liegt darin, dass ihre Anwendung auf reale Sachverhalte einen Sinn ergibt: Die Anwendung der Regeln der Logik in der Praxis wird von Menschen i. Allg. genauso wenig angezweifelt wie die Anwendung der Regeln der Arithmetik („unlogische" Argumente werden in aller Regel verworfen).[14]

Ähnlich wie bei Zahlen entwickeln Menschen schon sehr früh ein intuitives Verständnis von Logik. Dieses Verständnis ist aber insofern naiv, als es nicht vor **Antinomien**, also inneren Widersprüchen, die man mit dem mitgebrachten Logikverständnis nicht auflösen kann, schützt. Eine Auflösung dieser Widersprüche, die allgemein akzeptiert wird, wurde erst Anfang des letzten Jahrhunderts gefunden, nachdem sich Philosophen und Mathematiker schon über zwei Jahrtausende damit gequält hatten. Sie beruht u. a. auf einer strengen Trennung von **Objekt-** und **Metasprache**, die jedoch für Menschen, die sich vor allem der natürlichen Sprache bedienen, um ihre Gedanken zu ordnen, schwierig ist — unsere natürliche Sprache ist *abgeschlossen*, d. h., sie kennt keine verschiedenen Spracheebenen[15] und man kann alles in ihr sagen.

WIKIPEDIA

Objekt- und Metasprache

Ein klassisches Beispiel für eine Antinomie, oder widersprüchliche Aussage, der kein Wahrheitswert zugeordnet werden kann, ist der einfache Satz

Dieser Satz ist falsch.

Wäre dieser Satz tatsächlich falsch, dann müsste sein Gegenteil wahr, er also seinem Inhalt gemäß nicht falsch, sondern wahr sein; wäre er hingegen tatsächlich wahr, müsste er seinem Inhalt gemäß falsch sein. Da sich dies nicht auflösen lässt, ist der Satz als *unsinnig* abzutun — ihm kann keine Bedeutung zugeordnet werden. Mit der strikten Trennung von Objekt- und Metasprache können nun Antinomien einer solchen Bauart ausgeschlossen werden, indem man nämlich verlangt, dass nur in der Metasprache Aussagen über in der

---

[14] *Leibniz* hatte sich schon nicht nur mit der Mathematik, also den Zahlen, den Rechenoperationen und deren Gesetzmäßigkeiten beschäftigt, sondern auch mit der Sprache, also den Wörtern, den Sätzen und den Gesetzen der Logik. So hatte er unter anderem die Idee, Wörtern Zahlen zuzuordnen und so die logische Argumentation auf Rechenoperationen zurückzuführen, für die er eine Rechenmaschine gebaut hatte, so dass er die Argumente würde ausrechnen und so die unter Philosophen üblichen endlosen Diskussionen würde abkürzen können (sein berühmtes „Calculemus!"). Leider hatte seine (mechanische) Rechenmaschine Probleme beim Zehnerübertrag, so dass er schnell noch eine auf dem Dualsystem basierende in Auftrag gab, die einfacher zu konstruieren war. Tatsächlich wird heute vielerorts, nicht nur unter Philosophen, nicht mehr gestritten, sondern das Smartphone gezückt, um einen Faktencheck durchzuführen. Dieser bemüht ebenfalls das Dualsystem, nutzt dieses aber ganz anders, als Leibniz es sich erträumt hatte.

[15] es sei denn, man führt sie über bestimmte Konventionen wie beispielsweise den Gebrauch von Anführungszeichen, wie er in Kapitel 1 diskutiert wird, nachträglich ein

Objektsprache formulierte Aussagen getroffen werden dürfen. Demnach müsste obiger Satz ein Satz der Metasprache sein und sich „dieser“ auf einen (anderen) Satz in der Objektsprache beziehen; welcher genau gemeint ist, wäre dann dem Kontext zu entnehmen.

Syntax und Semantik

Heute wird die (mathematische) Logik praktisch immer als eine formale Sprache verstanden, deren Definition sich aus der ihrer **Syntax** und der ihrer **Semantik** zusammensetzt. Dabei bestimmt die Syntax die Form der *Ausdrücke* (oder Sätze) der Sprache und die Semantik deren *Bedeutung*. Diese Unterscheidung, die auch für die Definition von Programmiersprachen zentral ist (s. Abschnitt 20.1 in Kurseinheit 3), erlaubt in der Logik die Abtrennung eines förmlichen, auf Zeichenspielen beruhenden *(syntaktischen) Schlussfolgerungsmechanismus* von einem inhaltlichem *(semantischen) Schlussfolgerungsbegriff*. Diese Trennung erlaubt es wiederum, sich bei der Beantwortung logischer Fragestellungen ganz auf Zeichenspiele verlegen zu können und somit die Knoten im Hirn, die die Beantwortung dieser Fragestellungen auf der inhaltlichen, logischen Ebene nur allzu leicht verursachen kann, zu vermeiden. Ich werde diese Trennung hier jedoch nur andeuten, zum einen, weil ich die Analogie zur Behandlung von Zahlen (Kapitel 2) herausstellen will (bei der ich auf Syntax und Semantik gar nicht eingegangen bin), zum anderen, weil es der Zeichenspiele in der Digitalisierung viele gibt und ich denen der (mathematischen) Logik nicht unverhältnismäßig viel Platz geben will.

Kurs

Zeichen für *Wahr* und *Falsch*

Genau wie Zahlen werden die Wahrheitswerte *Wahr* und *Falsch* mithilfe von Zeichen aufgeschrieben. Da es nur zwei Wahrheitswerte gibt (im Gegensatz zu unendlich vielen Zahlen), ist dafür aber kein Stellenwertsystem notwendig — jedes Zeichen entspricht für sich einem Wahrheitswert. Dabei werden zur Darstellung der Wahrheitswerte häufig «1» und «0» für *Wahr* und *Falsch* verwendet; ihre Bedeutung, insbesondere, dass es sich dabei *nicht* um Ziffern einer Dualzahl, sondern um Wahrheitswerte handeln soll, muss dann aus dem Kontext oder ihrer Verwendung geschlossen werden. Um nicht unnötig Verwirrung zu stiften (und aus pädagogischen Gründen; s. Kapitel 10) verwende ich hier «W» und «F» als Zeichen für *Wahr* für *Falsch* (in tabellarischen Darstellungen auch ohne die Anführungszeichen); für die Werte (die Bedeutung der Zeichen) bleibe ich bei den Wörtern „Wahr“ und „Falsch“, setze sie aber — zur Kennzeichnung als Werte — kursiv.

3.1 Rechnen mit Wahrheitswerten: logische Operatoren

Um mit Wahrheitswerten zu „rechnen“ verwendet man *logische Verknüpfungen*, oder *Operatoren* (entsprechend den arithmetischen Operatoren +, – usw.). Die klassischen logischen Operatoren heißen „Und“ (die **Konjunktion**), „Oder“ (die **Disjunktion**) und „Nicht“ (die **Negation**); sie werden heute meistens mit den Zeichen «∧» (für Und), «∨» (für Oder) und «¬» (für Nicht) notiert. Ihre Bedeutung, oder *Semantik*, wird durch die folgende, **Wahrheitstabelle** oder **Wahrheitstafel** genannte Tabelle festgelegt:

WIKIPEDIA

Operanden		Ergebnisse der Operationen		
α	β	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\neg\alpha$
F	F	F	F	W
F	W	F	W	W
W	F	F	W	F
W	W	W	W	F

Man beachte, dass es sich dabei eigentlich um drei Wahrheitstafeln handelt (eine für jeden Operator), die jedoch aus anschaulichen Gründen zu einer zusammengefasst sind. In der Tabelle stehen α und β als Platzhalter für die Operanden der Operatoren ∧, ∨ und ¬, die jeweils *Wahr* («W») oder *Falsch* («F») sein können.[16] Wenn man beispielsweise wissen möchte, was *Falsch* ∧ *Wahr* ergibt, dann sucht man die Zeile, in der unter α «F» und unter β «W» steht, und liest den Wert in derselben Zeile der Spalte $\alpha \wedge \beta$ ab (hier: «F» für *Falsch*). Man beachte, dass der Operator ∨ nicht immer der umgangssprachlichen Bedeutung von Oder entspricht, die manchmal auch die von „entweder oder" sein kann (das sog. **exklusive Oder**, das *Falsch* ergibt, wenn beide Operanden *Wahr* sind, im Gegensatz zum **inklusiven Oder** der Logik, bei dem auch beide Operanden *Wahr* sein dürfen, um *Wahr* zu ergeben). Man beachte ferner die Analogie von Wahrheitstafeln und den Additionstabellen aus Abschnitt 2.4.1: Die Verknüpfung von Wahrheitswerten mit logischen Operatoren lässt sich wie die Addition von zwei Ziffern rein mechanisch, nämlich durch Nachschlagen in einer Tabelle bewerkstelligen.

logische Ausdrücke und ihr Wahrheitswert

Genau wie mit den arithmetischen lassen sich auch mit den logischen Operatoren längere Ausdrücke bilden. So sind beispielsweise $\neg W \vee F$ oder $W \wedge F \vee \neg W \wedge \neg F$ logische Ausdrücke, in denen mehr als ein Operator vorkommt. Ihr Wert lässt sich durch wiederholtes Nachschlagen in den Wahrheitstafeln für ∧, ∨ und ¬ bestimmen, wenn man weiß, dass „¬ vor ∧ geht und ∧ vor ∨" (so wie in der Arithmetik „Punktrechnung vor Strichrechnung geht"). Für $\neg W \vee F$ liest man also zunächst den Wert von $\neg W$, F, in der Wahrheitstafel von ¬ ab und dann den von $F \vee F$ in der Tafel von ∨. Wünscht man sich eine andere Bindung der Operatoren, muss man (genau wie bei *arithmetischen Ausdrücken*) Klammern setzen, also beispielsweise $\neg(W \vee F)$. Man beachte, dass die Auswertung logischer Ausdrücke nichts mit logischem Denken zu tun hat — es ist einmal mehr ein reines Zeichenspiel.

Implikation und Äquivalenz

Während es von den einstelligen logischen Operatoren mit der Negation nur einen interessanten (von insgesamt vier möglichen) gibt, gibt es von den zweistelligen mehr (hier sind 16 möglich). Neben dem bereits erwähnten exklusiven Oder finden vor allem die **Implikation**, häufig mit «→» notiert und als „… impliziert …", „aus … folgt …" oder „wenn …, dann …" gelesen, und die **Äquivalenz**, mit «↔» notiert

[16] Hier und im folgenden schreibe ich ∧, ∨ und ¬, wenn die logischen Operatoren Und, Oder und Nicht und nicht die sie bezeichnenden Zeichen («∧», «∨» und «¬») gemeint sind.

und als „wenn und nur wenn …, dann …“ oder „… genau dann, wenn …“ gelesen, Verwendung. Ihre Bedeutung lässt sich der folgenden Wahrheitstafel entnehmen:

Operanden		Ergebnisse	
α	β	$\alpha \to \beta$	$\alpha \leftrightarrow \beta$
F	F	W	W
F	W	W	F
W	F	F	F
W	W	W	W

Man beachte, dass $\to$ und $\leftrightarrow$ Operatoren sind, die auf einer Ebene mit $\wedge$, $\vee$ und $\neg$ stehen. Dies ist insofern wichtig zu verstehen, als die Begriffe (nicht die Zeichen!) der Implikation und der Äquivalenz mit einer anderen Bedeutung auch auf der *Metaebene*, bei der Rechtfertigung der Zeichenspiele der Logik, vorkommen und dort sogar eine zentrale Rolle spielen. Um damit einhergehende Verwirrungen zu vermeiden kann es helfen, zu wissen, dass $\to$ (zu besseren Unterscheidung auch *materiale Implikation*[17] genannt) und $\leftrightarrow$ (*materiale Äquivalenz*) auch durch eine Kombination von $\wedge$, $\vee$ und $\neg$ ersetzt werden können. Es gilt nämlich

$$\alpha \to \beta \quad \Leftrightarrow \quad \neg\alpha \vee \beta$$

und

$$\alpha \leftrightarrow \beta \quad \Leftrightarrow \quad (\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$$

wobei hier «$\Leftrightarrow$» die Äquivalenz auf der Metaebene, auch **logische** oder **semantische Äquivalenz** genannt, bezeichnet und bedeutet: Egal, ob α und β *Wahr* oder *Falsch* sind — auf der linken und der rechten Seite kommt stets das gleiche heraus, sie sind gleichbedeutend. Und in der Tat ergibt der Vergleich der linken und der rechten Seite der ersten obigen Äquivalenz in einer Wahrheitstafel

α	β	$\neg\alpha$	$\neg\alpha \vee \beta$	$\alpha \to \beta$
F	F	W	W	W
F	W	W	W	W
W	F	F	F	F
W	W	F	W	W

Man kann also auch $\neg\alpha \vee \beta$ anstelle von $\alpha \to \beta$ schreiben — am Ergebnis ändert das nichts.

[17] Die Bezeichnung „material“ ist, wie viele andere Namenswahlen in der Logik, nicht selbstverständlich, sondern nur aus der Geschichte der Logik heraus erklärbar. Ohne Kenntnis dieser Geschichte muss man die Namen einfach hinnehmen, was Erwachsenen zugegebenermaßen schwerfallen mag.

Selbsttest: Stellen Sie die vergleichende Wahrheitstafel auch für die oben behauptete Äquivalenz von $\alpha \leftrightarrow \beta$ und $(\alpha \wedge \beta) \vee (\neg\alpha \wedge \neg\beta)$ auf.

Quell der Wahrheit

Wie aber kommt es zu den Wahrheitstafeln und der damit einhergehenden Bedeutungsfestlegung der Operatoren? Zunächst einmal ist festzuhalten, dass es für zweistellige logische Operatoren (Operatoren mit zwei Operanden) und eine Logik mit nur zwei Werten (*Wahr* und *Falsch*) überhaupt nur 16 verschiedene Wahrheitstafeln geben kann — die Möglichkeiten wachsen also nicht in den Himmel. Die Wahrheitstafeln für ∧, ∨ und ¬ sind so ausgesucht, dass sie der umgangssprachlichen Bedeutung von Und, Oder und Nicht weitgehend entsprechen und so erlauben, alltäglichen Sachverhalten mithilfe der Logik eine Form zu geben. Die Fassung der Semantik von → und ↔ durch Wahrheitstafeln erscheint, gemessen an der der umgangssprachlichen Bedeutung von Implikation und Äquivalenz, erstaunlich schlicht; wie wir noch sehen werden, ist sie aber vollkommen angemessen. Grundsätzlich bleibt jedoch anzumerken, dass die Auswahl der logischen Operatoren und ihre Belegung mit Wahrheitstafeln einer gewissen Willkür unterliegt (so war Oder früher einmal als exklusives Oder definiert) und dass das, was hier so dogmatisch erscheint, das Ergebnis einer weit über hundertjährigen Reise ist. Die mathematische Logik ist, genau wie die Mathematik selbst, gewachsen.

Selbsttest: Stellen Sie Wahrheitstafeln für alle 16 möglichen zweistelligen Operatoren auf. Was fällt Ihnen auf? Können Sie alle Operatoren mithilfe von ∧, ∨ und ¬ nachbilden?

3.2 Aussagenlogik

Wikipedia

Die **Aussagenlogik** ist eine einfache Form der Logik, bei der es, wie der Name nahelegt, um die *Wahrheit* (den Wahrheitswert) von **Aussagen** geht. Dabei entsprechen einfache Sätze wie „Die Sonne scheint." oder „Es ist warm." **elementaren Aussagen**, die jeweils für sich wahr oder falsch sein können. Mittels logischer Operatoren, wie Sie sie im vorherigen Abschnitt kennengelernt haben, lassen sich aus elementaren Aussagen **zusammengesetzten Aussagen** bilden. Die Aussagenlogik ist damit einer einfachen, auf die Formulierung logischer Zusammenhänge abzielenden *natürlichen Sprache* nachempfunden.

Analogie zur angewandten Mathematik

Ob eine elementare Aussage wahr oder falsch ist, ist zunächst nicht Gegenstand der Aussagenlogik — vielmehr regelt sie, wie sich der Wahrheitswert von zusammengesetzten Aussagen aus den Wahrheitswerten ihrer elementaren Aussagen bestimmt. Die Zuordnung von Wahrheitswerten zu Aussagen entspricht dabei in etwa der Zuordnung von Zahlen zu Eigenschaften (wie beispielsweise dem Gewicht oder der Länge) von Objekten eines interessierenden Sachverhalts: Die zugeordneten Werte können mit Zeichenspielen verarbeitet und das Ergebnis wieder auf den Sachverhalt rückübertragen werden (vgl. dazu Kapitel 2). Elementare Aussagen werden daher in der Aussagenlogik in der Regel durch Buchstaben («A», «B» usw.) repräsentiert, die vom Inhalt der Aussage abstrahieren und die man sich wie *Variablen* für Wahrheitswerte vorstellen kann (ähnlich wie man die Variablen l, V usw. für die Länge, Volumen usw. von

etwas verwendet). Im Folgenden setze ich die Buchstaben, die in aussagenlogischen Ausdrücken für elementare Aussagen stehen, kursiv (und lasse die schweizerischen Anführungsstriche weg).

Aus elementaren Aussagen lassen sich wie gesagt mithilfe logischer Operatoren zusammengesetzte Aussagen bilden. So ist z. B.

„die Sonne scheint“ ∧ „es ist warm“[18]

eine zusammengesetzte Aussage. Sie ist nach der Wahrheitstafel für ∧ dann und nur dann wahr, wenn beide Einzelaussagen wahr sind. Bei der Betrachtung dieser zusammengesetzten Aussage ist es unerheblich, ob die Sonne wirklich scheint oder ob es wirklich warm ist — aussagenlogisch betrachtet hätte man genauso gut

$$A \wedge B$$

schreiben können, denn der Realitätsbezug der beiden obigen meteorologischen Feststellungen existiert nur für den menschlichen Leser und ist für die Aussagenlogik als Zeichenspiel ohne Bedeutung. Gleichwohl kann man sich durch einen Realitätsbezug davon überzeugen, dass die Regeln, nach denen logische Zeichenspiele vollzogen werden, einen praktischen Wert haben (genauso wie die der Arithmetik; vgl. Abschnitt 2.4). So ist beispielsweise die zusammengesetzte Aussage

¬ („die Sonne scheint“ ∧ „es ist warm“)

dann und nur dann wahr, wenn „die Sonne scheint“ oder „es ist warm“ falsch sind[19] (logisch, oder etwa nicht?) — es gilt nämlich

¬ („die Sonne scheint“ ∧ „es ist warm“) ⇔ ¬ „die Sonne scheint“ ∨ ¬ „es ist warm“

oder, abstrakter,

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$$

wie man sich anhand eines Vergleichs der für die linke und die rechte Seite dieser *semantischen Äquivalenz* aufgestellten Wahrheitstafeln vergewissern kann (ja, es ist tatsächlich logisch!). Man kann also in einer Aussage $\neg(A \wedge B)$ und $\neg A \vee \neg B$ per Zeichenspiel gegeneinander austauschen, ohne dass sich am Wahrheitswert der Aussage etwas ändert — erwiesenermaßen!

---

[18] Hier und im folgenden schreibe ich, der besseren Lesbarkeit wegen, auch ganze Sätze als Phrasen, wenn sie Aussagen im aussagenlogischen Sinn darstellen sollen.

[19] Die Verwendung des ungrammatisch anmutenden Plural „sind“ an dieser Stelle soll klarstellen, dass mit dem das Subjekt bildenden „oder“ das *inklusive Oder* gemeint ist.

Selbsttest: Überlegen Sie sich, welche Äquivalenz für zusammengesetzte Aussagen der Form $\neg(A \vee B)$ gilt. Hat Ihnen Ihr mitgebrachtes Logikverständnis dabei geholfen?

3.2.1 Erfüllbarkeit und Allgemeingültigkeit

Der Wahrheitswert aussagenlogischer Ausdrücke leitet sich also systematisch aus den Wahrheitswerten der darin vorkommenden elementaren Aussagen ab. Man kann daher die Frage stellen, welche Wahrheitswerte die elementaren Aussagen eines Ausdrucks haben müssen, damit der Ausdruck insgesamt wahr ist, und ob es überhaupt solche Konstellationen gibt. Letztere Frage nennt man auch die Frage der **Erfüllbarkeit**; dass sie gerechtfertigt ist, kann man daran erkennen, dass schon der einfache Ausdruck $A \wedge \neg A$ nicht erfüllbar ist (sowohl *Wahr* ∧ *Falsch* als auch *Falsch* ∧ *Wahr* ergeben *Falsch*). Unerfüllbare Aussagen nennt man auch **Kontradiktionen**; ihre Falschheit ist inhärent, d. h., sie wohnt ihnen inne und hängt nicht von den Wahrheitswerten der darin enthaltenen elementaren Aussagen ab.

Erfüllbarkeitsproblem der Aussagenlogik

Die Frage der Erfüllbarkeit aussagenlogischer Ausdrücke ist ein viel allgemeineres Problem, als man es angesichts seiner Schlichtheit vermuten würde. Das ergibt sich daraus, dass sich viele andere Probleme auf ein **Erfüllbarkeitsproblem der Aussagenlogik**, auch **SAT-Problem** genannt, zurückführen lassen — hat man das Erfüllbarkeitsproblem gelöst, hat man zugleich das Ursprungsproblem gelöst. Man ist daher sehr an Verfahren interessiert, die Erfüllbarkeitsprobleme mit möglichst wenig Aufwand lösen können. Die Suche nach solchen Verfahren hält unvermindert an; man vermutet jedoch, dass es kein Verfahren gibt, das beliebige Erfüllbarkeitsprobleme entscheidend schneller lösen kann als alle bisher bekannten Verfahren.

WIKIPEDIA

Ursache des Problems

Dabei ist die Frage der Erfüllbarkeit aussagenlogischer Ausdrücke im Grunde ganz einfach mithilfe ihrer Wahrheitstafeln zu beantworten: Ergibt sich in mindestens einer Zeile der Wahrheitstafel eines Ausdrucks *Wahr* als sein Wert, ist er erfüllbar, andernfalls nicht. Allerdings wächst die Größe der Wahrheitstafeln mit der Anzahl der elementaren Aussagen, die in den Ausdrücken vorkommen, exponentiell: Für n elementare Aussagen hat eine Wahrheitstafel 2^n Zeilen. Für viele Problemstellungen, die man als Erfüllbarkeitsprobleme ausdrücken kann, ist die Zahl der elementaren Aussagen aber so groß, dass sich die dazugehörigen Wahrheitstafeln nicht mehr vollständig aufschreiben lassen, schon weil die Zeit dafür nicht reicht. Nur wenn man viel Glück hat, findet sich eine Kombination von *Wahr* und *Falsch*, die *Wahr* ergibt, schon am Anfang einer Tafel; sonst ist man darauf angewiesen, bestimmte Regelmäßigkeiten im Ausdruck zu erkennen und daraus Probierreihenfolgen abzuleiten, die mit einer gewissen Wahrscheinlichkeit früher zum Ziel führen.

allgemeingültige Aussagen

Das Gegenteil einer unerfüllbaren Aussage, eine **allgemeingültige Aussage** (in der Logik auch **Tautologie** genannt), liegt vor, wenn eine Aussage immer, also unter allen möglichen Zuordnungen von Wahrheitswerten zu den elementaren Aussagen, die darin vorkommen, wahr ist. So ist beispielsweise $A \vee \neg A$ allgemeingültig

(sowohl *Wahr* ∨ *Falsch* als auch *Falsch* ∨ *Wahr* ergeben *Wahr*). Die Wahrheit ist allgemeingültigen Aussagen also, genau wie die Falschheit unerfüllbaren, inhärent. Allgemeingültigkeit kann man, genau wie die (Un)Erfüllbarkeit, mithilfe einer Wahrheitstafel feststellen; die Probleme sind allerdings ebenfalls dieselben.

Allgemeingültigkeit und semantische Äquivalenz

Es sind übrigens zwei aussagenlogische Ausdrücke α und β genau dann *semantisch äquivalent* (geschrieben als $\alpha \Leftrightarrow \beta$) und damit gegeneinander austauschbar, wenn $\alpha \leftrightarrow \beta$ **allgemeingültig** (eine Tautologie) ist.

Selbsttest: Überprüfen Sie diese Behauptung anhand aller bisher angeführten semantischen Äquivalenzen.

3.2.2 Logische Schlussfolgerung

Die Logik ganz allgemein (und somit auch die Aussagenlogik) dient der Argumentation „mit formaler Strenge". Sie ist die formale Grundlage dessen, was man umgangssprachlich mit „Das ist doch logisch!" meint. Insbesondere möchte man mithilfe der Logik aus Fakten den Wahrheitsgehalt (oder -wert) von Behauptungen (Thesen) ableiten können und so aus bestehendem Wissen neues Wissen gewinnen. Dies geschieht mithilfe der **semantischen** oder **logischen Schlussfolgerung**; sie erlaubt es, aus der Wahrheit von Aussagen auf die anderer zu schließen.

materiale vs. logische Implikation

Man könnte nun versucht sein, die (materiale) *Implikation* → (die ja auch als „wenn dann" gelesen wird; s. o.) mit der logischen Schlussfolgerung, die ebenfalls eine Form der Implikation darstellt, gleichzusetzen. Tatsächlich legt eine Aussage wie „wenn es warm ist, bin ich draußen" eine Schlussfolgerung nahe, insbesondere dann, wenn man noch die Aussage „es ist warm" hinzunimmt. Jedoch ist in der Sprache der Aussagenlogik mit den beiden obigen Aussagen gar nicht gesagt, ob es warm ist oder ob die Implikation gilt — beide Aussagen könnten ja (wie jede Aussage, die keine Tautologie ist) auch falsch sein (wobei sich letzteres daraus ergibt, dass es zwar warm ist, ich aber nicht draußen bin; vgl. die Wahrheitstafel von → in Abschnitt 3.1). Eine Aussage hinzuschreiben heißt noch nicht, dass sie wahr ist. Das ist ganz so wie bei Gericht.

Axiome

Nun kann man aus nichts nichts schlussfolgern und so müssen am Anfang Fakten stehen, also Aussagen, deren Wahrheit entweder feststeht oder vorausgesetzt wird. So könnte im obigen Beispiel „es ist warm" oder ¬„es ist warm" (aber nur eines von beiden!) Fakt sein. Eine unbedingt wahre aussagenlogische, elementare oder zusammengesetzte Aussage nennt man **Axiom**; ihre Wahrheit muss entweder in ihr selbst liegen (die Aussage muss *allgemeingültig* oder eine *Tautologie* sein; s. o.) oder extern begründet werden (z. B. mit Fakten oder Gesetzmäßigkeiten der Domäne, über die man Aussagen machen möchte und in der man die Gesetze der Logik anwenden will). Dabei unterliegt die Festlegung der Axiome einer gewissen Freiheit — so gibt es beispielsweise verschiedene, konkurrierende Axiomatisierungen der Mengenlehre. Allerdings sollte die Menge der Axiome so gewählt sein, dass man mithilfe der Schlussfolgerung alle Fragen beantworten kann, die für

eine gegebene Aufgabenstellung von Bedeutung sind; zudem muss sie in sich *widerspruchsfrei* sein, da sich sonst beliebiges aus ihr schlussfolgern lässt („ex contradictione sequitur quodlibet“).

Schlussfolgerungsregeln

Hat man eine Menge von Axiomen, dann kann man mithilfe von **Schlussfolgerungsregeln** (auch **Schluss-** oder **Ableitungsregeln** genannt) von weiteren elementaren oder zusammengesetzten Aussagen feststellen, dass sie wahr sein müssen. Eine Schlussfolgerungsregel besteht dabei aus einer Anzahl von Aussagen, die die Voraussetzung der Schlussfolgerung sind und die auch **Prämissen** genannt werden, und einer Aussage, die das Gefolgerte darstellt und die auch **Konklusion** genannt wird. Zwei einfache Schlussfolgerungsregeln sind beispielsweise

$$\frac{\alpha \wedge \beta}{\alpha} \qquad \text{und} \qquad \frac{\alpha \wedge \beta}{\beta}$$

wobei hier jeweils die Prämisse über dem Strich und die Konklusion unter dem Strich steht und α und β als Platzhalter, oder *Metavariablen*, für beliebige Aussagen stehen. Die Platzhalter machen den Schemacharakter der Regeln deutlich: Bei ihrer Anwendung auf einen konkreten Fall werden sie durch konkrete Aussagen ersetzt. Wenn also beispielsweise „es ist warm“ ∧ „ich bin draußen“ gilt, dann kann man mithilfe der Regeln „es ist warm“ bzw. „ich bin draußen“ folgern. Die beiden Regeln sind dadurch logisch gerechtfertigt, dass in der Wahrheitstafel von ∧ da, wo für $\alpha \wedge \beta$ *Wahr* steht, auch für α bzw. β *Wahr* steht (dass also, wenn $\alpha \wedge \beta$ wahr sein soll, auch α bzw. β wahr sein muss):

α	β	$\alpha \wedge \beta$
~~F~~	~~F~~	~~F~~
~~F~~	~~W~~	~~F~~
~~W~~	~~F~~	~~F~~
W	W	W

Die Wahrheitstafel wird hier also gewissermaßen rückwärts gelesen (es wird vom Ergebnis auf die Operanden geschlossen).

Übrigens: Die Wahl der Notation für die Schlussfolgerungsregeln macht die Unterscheidung zwischen *Objektsprache* (Aussagenlogik) und *Metasprache* (Schlussfolgerungsregel) auch auf dem Papier deutlich: Die gezielte Nutzung der zweiten Dimension deutet an, dass mit der Schlussfolgerungsregel etwas über die Aussagenlogik ausgesagt wird, was nicht in einer Ebene mit den Aussagen, die in der Sprache der Aussagenlogik formuliert sind, steht. Das ist bei alternativen, linearen (einzeiligen) Schreibweisen (die beispielsweise «⇒» oder «⊢» anstelle des Schlussfolgerungsstrichs verwenden) nicht so offensichtlich.

Modus ponens

Die wohl bekannteste Schlussfolgerungsregel hat die Form

$$\frac{\alpha \qquad \alpha \rightarrow \beta}{\beta}$$

Sie wird **Modus ponens** genannt und besagt, dass wenn α wahr und auch $\alpha \rightarrow \beta$ wahr ist, dass dann β folgt, also ebenfalls wahr sein muss.[20] Mit der Anwendung des Modus ponens wird also der Zusammenhang vollzogen, der mit $\alpha \rightarrow \beta$ auf der objektsprachlichen Ebene behauptet und durch die vorausgesetzte Wahrheit dieser Implikation festgestellt wurde. Wenn also beispielsweise „es ist warm" ein Axiom ist (das Gegenteil also gar nicht in Betracht gezogen werden muss) und „wenn es warm ist, bin ich draußen" ebenfalls ein Axiom ist (man also die Möglichkeit, dass die Implikation nicht wahr ist, ausschließen kann), dann kann man folgern, dass auch „ich bin draußen" nur wahr sein kann. Dies ergibt sich unmittelbar aus der Wahrheitstafel von $\rightarrow$: Wenn man die Zeilen, in denen für α oder für $\alpha \rightarrow \beta$ *Falsch* steht, streicht, dann bleibt nur noch eine Zeile, und in der steht für β *Wahr*:

α	β	$\alpha \rightarrow \beta$
~~F~~	~~F~~	~~W~~
~~F~~	~~W~~	~~W~~
~~W~~	~~F~~	~~F~~
W	W	W

Wenn man hingegen $\neg\alpha$ zum Axiom erhebt (also festlegt, dass α den Wert *Falsch* hat), müssen anstelle der ersten beiden die letzten beiden Zeilen gestrichen werden, so dass sich für den Wert von β nichts ableiten lässt — er kann *Wahr* oder *Falsch* sein. Man sieht also, dass die Wahrheitstafel von $\rightarrow$, der materialen Implikation, genau die Schlussfolgerungen erlaubt, die man erwarten würde (vgl. Abschnitt 3.1). Dabei gilt der Modus ponens genauso, wenn man darin $\alpha \rightarrow \beta$ durch $\neg\alpha \vee \beta$ ersetzt.

Modus tollens

Während aus der Wahrheit von $\neg\alpha$ und von $\alpha \rightarrow \beta$ die Wahrheit von $\neg\beta$ zu folgern ein Trugschluss ist (dies gilt nur für $\neg\alpha$ und $\alpha \leftrightarrow \beta$), ist der sog. **Modus tollens** sehr wohl eine gültige Schlussfolgerungsregel

$$\frac{\alpha \rightarrow \beta \qquad \neg\beta}{\neg\alpha}$$

wie man sich anhand einer weiteren Betrachtung der Wahrheitstafel von $\rightarrow$ überzeugen kann. So kann man beispielsweise aus dem Wissen „wenn die Sonne scheint und es warm ist, dann ist Sommer" und „es ist nicht Sommer" das Wissen „es ist nicht warm oder die Sonne scheint nicht" schlussfolgern.

Selbsttest: Betrachten Sie die beiden Aussagen „nie ess ich Essig" und „ess ich Essig, ess ich Essig zum Salat". Können sie beide gleichzeitig wahr sein oder sind sie widersprüchlich? Was können Sie daraus ableiten, wenn Sie sie als Axiome auffassen?

[20] Man beachte, dass hier die metasprachliche Konjunktion implizit ist — ihr Zeichen ist der Abstand zwischen α und $\alpha \rightarrow \beta$.

Schlussfolgerungsketten als Zeichenspiel

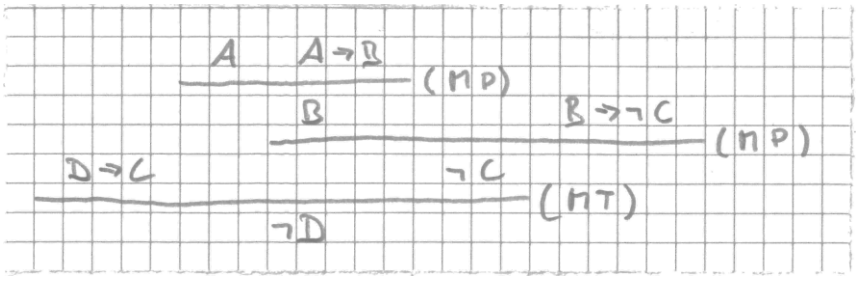
Die Mächtigkeit der logischen Schlussfolgerung beruht darauf, dass man abgeleitete wahre Aussagen wie Axiome als Prämissen für weitere Schlussfolgerungen verwenden darf und so **Schlussfolgerungsketten** bilden kann. Dies wird im folgenden Beispiel, in Form eines Zeichenspiels auf Papier ausgeführt, ausgenutzt. Nehmen Sie dazu

A	„es ist warm“
B	„ich bin draußen“
C	„ich höre das Telefon“
D	„ich habe Bereitschaftsdienst“

als elementare Aussagen sowie

$A \to B$	„wenn es warm ist, bin ich draußen“
$B \to \neg C$	„wenn ich draußen bin, höre ich das Telefon nicht“
$D \to C$	„wenn ich Bereitschaftsdienst habe, höre ich das Telefon“
A	„es ist warm“

als Axiome an. Es soll also u. a. A („es ist warm“) wahr sein, aber über den Wahrheitswert der atomaren Aussagen B, C und D ist zumindest direkt nichts gesagt. Aus den Axiomen lässt sich aber der Reihe nach die Wahrheit von B („ich bin draußen“), $\neg C$ („ich höre das Telefon nicht“) und $\neg D$ („ich habe keinen Bereitschaftsdienst“) wie folgt per Zeichenspiel schlussfolgern (wobei hier „(MP)“ die Anwendung des Modus ponens und „(MT)“ die Anwendung des Modus tollens kennzeichnet):



Es sind also erwiesenermaßen B wahr sowie C und D falsch.

Ableitung wahrer zusammengesetzter Aussagen

Lässt man das Axiom A weg, kann aus den übrigen Axiomen immer noch die Wahrheit von $A \to \neg C$, $B \to \neg D$ und $A \to \neg D$ geschlossen werden. Dies folgt aus der *semantischen Äquivalenz* von $\alpha \to \beta$ und $\neg\beta \to \neg\alpha$ sowie der mehrmaligen Anwendung der **Kettenschlussregel**

$$\frac{\alpha \to \beta \qquad \beta \to \gamma}{\alpha \to \gamma}$$

Die abgeleiteten Implikationen sagen zwar nichts über den Wahrheitsgehalt der elementaren Aussagen aus, stellen aber dennoch einen Erkenntnisgewinn dar. Wenn Sie nun finden,

dass an der gewonnenen Implikation „wenn es warm ist, habe ich keinen Bereitschaftsdienst“ ($A \rightarrow \neg D$) inhaltlich etwas nicht stimmen kann (durch Herstellung eines Realitätsbezuges), dann bedeutet dies nicht, dass Sie die Zeichenspiele der Logik anzweifeln sollten, sondern dass Sie die Axiome inhaltlich infrage stellen müssen. Das ist kein Tabubruch, sondern gute wissenschaftliche Praxis — Axiome können auch mal falsch sein (dann aber hoffentlich nur irrtümlich).

Selbsttest: Eine Schlussfolgerungsregel mit Prämissen $\alpha_1 \ldots \alpha_n$ und Konklusion β ist genau dann logisch gerechtfertigt, wenn $\alpha_1 \wedge \ldots \wedge \alpha_n \rightarrow \beta$ *allgemeingültig*, oder eine *Tautologie*, ist. Überprüfen Sie diese Behauptung anhand der obigen Schlussfolgerungsregeln.

3.2.3 Boolesche Algebra

Im Zusammenhang mit der Digitalisierung taucht häufig der nach dem englischen Mathematiker *George Boole* benannte Begriff der **booleschen Algebra** auf, wenn es um *Wahr* und *Falsch* sowie *Und*, *Oder* und *Nicht* geht. Dabei handelt es sich um eine mathematische Struktur, die von der *Aussagenlogik* und der *Mengenlehre* abstrahiert. Dazu führt sie das sog. **Null-** und das **Einselement** ein, die anstelle des Wahrheitswerts *Falsch* und der leeren Menge bzw. des Wahrheitswerts *Wahr* und des Universums stehen, und ersetzt die logischen Operatoren Und, Oder und Nicht sowie die Mengenoperatoren Durchschnitt, Vereinigung und Komplement jeweils durch *abstrakte Operationen*, deren Bedeutung durch eine Reihe von Axiomen festgelegt ist. Dabei gelten für eine boolesche Algebra mit nur zwei Werten, dem Null- und dem Einselement, die Wahrheitstafeln von Abschnitt 3.1 entsprechend. Dermaßen von einer konkreten Bedeutung befreit lassen sich die Gesetze der booleschen Algebra auch auf andere Bereiche übertragen, wobei im Kontext der Digitalisierung vor allem die *Schaltalgebra* (s. Abschnitt 11.1 in Kurseinheit 2) eine wichtige Rolle spielt.

Rechnen mit boolescher Algebra

Mit der Darstellung des Null- und Einselements durch die beiden Ziffern des *Dualsystems*, «0» und «1», lassen sich die beiden Additionstabellen aus Abschnitt 2.4.1 als *boolesche Ausdrücke* darstellen: Wenn S_1 eine Ziffer des ersten Summanden und S_2 eine Ziffer des zweiten Summanden an gleicher Stelle sowie S die Ziffer der Summe und $\ddot{U}$ die des erhaltenen Übertrags an der Stelle darstellt, so gilt für eine Addition ohne Berücksichtigung eines vorliegenden Übertrags aus einer vorausgegangenen Addition (also für die erste Tabelle, den sog. **Halbaddierer**)

$$S = \neg (S_1 \leftrightarrow S_2)$$
$$\ddot{U} = S_1 \wedge S_2$$

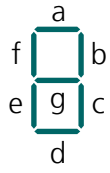
(wobei ich hier für die booleschen Operatoren die in der Logik verwendeten Zeichen verwendet habe; s. Abschnitt 3.1). Wer das nicht glaubt, der möge bitte die Wahrheitstafeln für die beiden obigen Ausdrücke aufstellen und neben die erste Tabelle aus Abschnitt 2.4.1 legen. Man beachte jedoch, wie durch diese Darstellung die Bedeutung der Zeichen verwischt: Stehen S_1, S_2, S und $\ddot{U}$ (bzw. «0» und «1») für Wahrheitswerte oder Ziffern einer

Zahl? Dieser Verlust an Eindeutigkeit und was er bedeutet wird in Kapitel 10 wieder aufgegriffen.

Selbsttest: Formulieren Sie die zweite der Additionstabellen aus Abschnitt 2.4.1, den sog. **Volladdierer**, mithilfe von booleschen Ausdrücken.

Die Addition lässt sich also wie eine logische Verknüpfung als boolescher Ausdruck darstellen. Für die Praxis bedeutet dies, dass man für die Addition von Zahlen und die logische Verknüpfung von Aussagen keine verschiedenen Regeln für Zeichenspiele braucht — es reichen die der booleschen Algebra. Voraussetzung ist allerdings, dass man für Ziffern und Wahrheitswerte dieselben Zeichen verwendet.

Selbsttest: Überlegen Sie sich, wie Sie mithilfe boolescher Ausdrücke die sieben Segmente einer Siebensegmentanzeige (nebenstehend mit „a“ bis „g“ bezeichnet) so ansteuern können, dass sie eine Stelle einer binär codierten Dezimalzahl (s. Abschnitt 2.6.2) anzeigt. Wie codieren Sie An (leuchtend) und Aus (dunkel)?



3.3 Andere Logiken

Die Aussagenlogik ist eine sehr einfache Logik. Entsprechend umständlich ist es, komplexere Sachverhalte in ihr auszudrücken. Obwohl die grundsätzliche Beschränkung der Aussagenlogik eher theoretischer Natur ist (es lässt sich das meiste, was man in Bezug auf die reale Welt aussagen möchte, in Aussagenlogik ausdrücken), werden ihr in der Praxis häufig andere Logiken vorgezogen.

3.3.1 Mehrwertige Logik

In der klassischen Logik gilt der Grundsatz vom ausgeschlossenen Dritten („tertium non datur“): Eine Aussage ist entweder wahr oder falsch. In der Praxis reicht das jedoch oftmals nicht aus.

Unbekannt als Wahrheitswert

Zum einen muss man regelmäßig in der Lage sein, mit *unsicherer Information* umzugehen: Ob eine Aussage wahr oder falsch ist kann schlicht unbekannt sein. Gleichwohl kann man in solchen Situationen immer noch Aussagen treffen, z. B. wenn der Wahrheitswert einer elementaren Aussage irrelevant für den Wahrheitswert einer zusammengesetzten Aussage ist. So ist $A \vee B$ *Wahr*, wenn A *Wahr* ist, auch wenn der Wahrheitswert von B unbekannt ist. Der Wahrheitswert von $A \wedge B$ ist hingegen selbst unbekannt, wenn der von A *Wahr* und der von B unbekannt ist. *Unbekannt* kann man also als einen dritten Wahrheitswert auffassen und alle Wahrheitstafeln entsprechend erweitern (wobei «?» für *Unbekannt* stehe):

Operanden		Ergebnisse der Operationen		
α	β	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\neg\alpha$
F	F	F	F	W
F	W	F	W	W
F	?	F	?	W
W	F	F	W	F
W	W	W	W	F
W	?	?	W	F
?	F	F	?	?
?	W	?	W	?
?	?	?	?	?

Solche **dreiwertige Logiken** sind für die Digitalisierungspraxis sehr wichtig: Sie kommen beispielsweise in *Datenbanksystemen* (s. Kapitel 30) zum Einsatz (wo *Unbekannt* nicht nur Wahrheitswert, sondern auch Zahl, Text usw. sein kann), die ohne die Möglichkeit, Unwissenheit abzubilden, schlicht unbrauchbar wären.

Fuzzylogik

Des Weiteren ist die Welt nicht schwarzweiß. Selbst wenn man Subjektivität außen vor lässt, ist der Wahrheitswert einer Aussage „Es ist warm." bei einer Temperatur von 15°C nicht eindeutig *Wahr* oder *Falsch*, solange „warm" nicht scharf (durch die Angabe einer Grenztemperatur) definiert ist. In solchen Fällen kann es hilfreich sein, die Alternative von *Wahr* und *Falsch* durch ein Kontinuum von (unendlich vielen) Wahrheitswerten zu ersetzen. So wird bei der sog. **Fuzzylogik** *Wahr* durch die Zahl **1** und *Falsch* durch die Zahl **0** repräsentiert sowie alle Wahrheitswerte dazwischen durch Werte aus dem Intervall **(0, 1)**. Wahrheit wird dadurch *relativ*: Eine Aussage kann wahrer sein als eine andere. Außerdem können, wegen der Unendlichkeit des Wertevorrats, Operationen auf Wahrheitswerten (wie $\wedge$, $\vee$ oder $\neg$) nicht mehr durch Tabellen definiert werden — stattdessen kommen mathematische Operatoren (wie das Minimum zweier Werte für $\wedge$ oder das Maximum für $\vee$) zum Einsatz. Man beachte, dass „unscharfe" (fuzzy) Wahrheitswerte von Wahrscheinlichkeiten strikt abzugrenzen sind; so haben auch die Axiome der *Wahrscheinlichkeitstheorie* keine Bedeutung für die Fuzzylogik. Dennoch werden Wahrscheinlichkeitstheorie und Fuzzylogik manchmal (auch im Kontext der sog. *künstlichen Intelligenz*; s. Abschnitt 3.4) als Alternativen zueinander angesehen; eine solche Sichtweise zeugt jedoch von einer nur oberflächlichen Betrachtung der Problemstellung.

WIKIPEDIA

Systeme, die auf Fuzzylogik basieren, insbesondere die sog. *Fuzzy controller*, gehören zusammen mit den sog. *neuronalen Netzen* in die Gruppe der sog. *universellen Approximatoren*: Beide sind in der Lage, beliebige Funktionen beliebig genau anzunähern. Allerdings sind Fuzzy-Systeme (zumindest traditionelle) regelbasiert, weswegen ihre Entscheidungen logisch nachvollziehbar sind. Dies ist für neuronale Netze, die angelernt und nicht programmiert werden, in der Regel nicht der Fall.

3.3.2 Prädikatenlogik

Die Geschichte der Logik ist unauflösbar mit der Betrachtung *natürlicher Sprache* und ihrer Verwendung im (philosophischen) Diskurs verbunden. Zumindest den europäischen Sprachen ist dabei gemein, dass die allermeisten einfachen Sätze dieselbe einfache Struktur aufweisen: Ein *Prädikat* trifft eine Aussage über ein oder mehrere Objekte. Aussagen, die auch in der **Prädikatenlogik** (wie in der *Aussagenlogik*) wahr oder falsch sein können, haben also selbst eine Struktur; es sind ausdrücklich Aussagen über Objekte. Der Inhalt des Satzes „Sokrates ist Mensch." etwa wird demnach durch das (einstellige) Prädikat ***mensch***, angewendet auf das Objekt ***Sokrates***, dargestellt: ***mensch***(***Sokrates***).[21]

Da es der Objekte viele (sogar unendlich viele!) geben und somit auf dem Papier schlecht jedes Prädikat auf jedes Objekt angewendet werden kann, erlaubt die Prädikatenlogik Schreibweisen für unendliche *Konjunktionen* und *Disjunktionen*. Die (nicht atomare) Aussage „Alle Menschen sind sterblich." etwa wird in der Prädikatenlogik als

$$\forall x\colon mensch(x) \rightarrow sterblich(x)$$

notiert, was als „Für alle *x* gilt: wenn *x* Mensch ist, dann ist *x* sterblich." gelesen wird. In einer Welt mit nur drei Objekten ***Sokrates***, ***Chairephon*** und ***Hermogenes*** entspräche dies der Konjunktion

$$\begin{aligned} &(mensch(Sokrates) \rightarrow sterblich(Sokrates)) \\ \wedge\, &(mensch(Chairephon) \rightarrow sterblich(Chairephon)) \\ \wedge\, &(mensch(Hermogenes) \rightarrow sterblich(Hermogenes)) \end{aligned}$$

In einer Welt mit unendlich vielen Objekten entspräche es einer unendlichen Konjunktion (die man natürlich nicht explizit aufschreiben kann). Entsprechendes gilt für die Disjunktion, wobei hier $\exists x\colon \ldots$ (gelesen als „es existiert ein *x*, für das gilt: …") an die Stelle von $\forall x\colon \ldots$ tritt. Man nennt $\forall$ und $\exists$ auch **Quantoren** und ***x*** eine durch einen Quantor *gebundene* (oder *quantifizierte*) *Variable*.

---

[21] Die Schreibweise erinnert an die der mathematischen *Funktionsanwendung* $f(x)$ und tatsächlich kann man ein (einstelliges) Prädikat auch als eine (einstellige) Funktion auffassen, die ihr Argument, ein Objekt, auf einen Wahrheitswert abbildet. Allerdings unterscheidet die Prädikatenlogik Prädikate von Funktionen; letztere werden zwar ebenfalls auf Objekte angewendet, bleiben aber stets *uninterpretiert* (ihre Anwendung ordnet in der Prädikatenlogik Objekten also keinen Wert — auch keinen Wahrheitswert — zu). S. dazu auch Abschnitt 19.3 in Kurseinheit 3.

Wie man schon an den Beispielen leicht erkennt, ist Prädikatenlogik *ausdrucksstärker* als Aussagenlogik. Diese Ausdrucksstärke hat jedoch auch ihren Preis: Genau wie in natürlicher Sprache ist es damit möglich, Paradoxien wie die vom lügenden Kreter Epimenides („Epimenides der Kreter sagte: Alle Kreter sind Lügner.“) auszudrücken, wenn man nämlich zulässt, dass Aussagen (Prädikate) auch über Aussagen (Prädikate) gemacht werden können. In der sog. *Prädikatenlogik erster Stufe* verbietet man dies daher. Trotz dieser Einschränkung musste man erkennen, dass in der Prädikatenlogik erster Stufe Aussagen formuliert werden können, deren Gültigkeit oder Erfüllbarkeit sich nicht durch die Anwendung von *Schlussfolgerungsregeln* ableiten lässt (die mangelnde *Entscheidbarkeit* der Prädikatenlogik). Deswegen verwendet man zum Schlussfolgern gern noch weiter eingeschränkte Formen der Prädikatenlogik, so etwa *monadische Prädikatenlogik* oder *Beschreibungslogik*.

WIKIPEDIA

WIKIPEDIA

WIKIPEDIA

3.4 Künstliche Intelligenz

Anders als das automatische Rechnen verspricht das automatische Schlussfolgern, also das Ableiten von neuem Wissen (im Sinne *allgemeingültiger Aussagen*) aus vorhandenem Wissen (Abschnitt 3.2.2), Computer **künstlich intelligent** zu machen. Allerdings hat sich schnell herausgestellt, dass das bloße Schlussfolgern so wenig zielgerichtet ist, dass nur höchst selten, und dann auch eher nur zufällig, etwas Sinnvolles dabei herauskommt (je nach Axiomensatz kann es unendlich viele ableitbare Aussagen geben, aber die wenigsten davon sind nützlich oder auch nur interessant). Gerade wenn die *Schlussfolgerungsketten* lang sind, liegt die Herausforderung darin, zielgerichtet vorzugehen, was voraussetzt, dass man das Ziel, also was man schlussfolgern möchte, zumindest ungefähr weiß. Das ist auch bei menschlichen Intelligenzleistungen nicht viel anders.

Theorembeweiser

Etwas vielversprechender erscheint daher der Versuch, von bestimmten Aussagen automatisch festzustellen, ob sie sich aus vorhandenem Wissen ableiten lassen. Sog. *Theorembeweiser*, die z. B. in der Mathematik eingesetzt werden, können Menschen mittlerweile dabei helfen, solche Schlussfolgerungsketten zu finden; der vollständig automatische Beweis gelingt aber noch eher selten.

A year spent in artificial intelligence is enough to make one believe in God.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

Die künstlich intelligenten Systeme, die heute von sich reden machen (wie z. B. IBM Watson) funktionieren daher anders. Sie versuchen, einen möglichst großen Korpus von bestehendem nützlichem Wissen zusammenzutragen und auf dieser Basis mit eher wenig Intelligenz Fragen zu beantworten. Ein gutes Beispiel hierfür war noch bis vor kurzem das Schachspiel: Ein konkurrenzfähiger Schachcomputer kennt sehr viele gespielte Schachpartien und weiß, wie sie ausgegangen sind. Er kann daher viele Stellungen daraufhin beurteilen, ob sie in der

WIKIPEDIA

Vergangenheit zu Sieg oder Niederlage geführt haben, und kann damit bei jedem anstehenden Zug die möglichen Alternativen bewerten. Erst in den Endspielen, wenn nur noch sehr wenige Figuren auf dem Feld stehen, können Spiele bis zum Ende „durchgerechnet", also alle möglichen Folgen von Spielzügen untersucht und so die gefunden werden, die zu einem Sieg führen. Für einen menschlichen Beobachter des Spiels mag diese Leistung wahnsinnig intelligent erscheinen, doch handelt es sich tatsächlich nur um das Ergebnis einer zwar sehr umfassenden, jedoch stumpfen statistischen Analyse bzw., beim Endspiel, um das Ergebnis einer erschöpfenden Suche.

Statistik statt Intelligenz

Viele andere intelligent erscheinende Leistungen von Computern, wie etwa die *Spracherkennung* oder die automatische *Übersetzung natürlichsprachlicher Texte*, basieren heute auf der statistischen Analyse großer Mengen von Fallbeispielen, wobei diese Mengen genau den Ausschnitt der Welt abbilden, in dem der Computer intelligent erscheinen soll. Dabei sind die Leistungen in der Regel umso besser, je enger der Kontext eingeschränkt werden kann. So lässt sich beispielsweise eine Bedienungsanleitung für einen Fotoapparat leichter automatisch übersetzen als ein Stück Weltliteratur, die beim Lesen und somit auch beim Übersetzen eine breite Bildung sowie allgemeine Lebenserfahrung voraussetzt.[22]

unüberwachtes Lernen als Idealfall

Für Spiele wie das Schachspiel oder Go konnte kürzlich demonstriert werden, dass selbstlernende Verfahren deutlich effektiver sein können (sowohl was den getriebenen Aufwand als auch was das Ergebnis angeht) als solche, die auf der Analyse einer großen Menge realer Fallbeispiele basieren. Dabei wird allerdings ausgenutzt, dass die Domäne (das Spiel) durch einen bekannten Satz von Regeln vollständig beschrieben ist und sich das Ergebnis (Sieg oder Niederlage) ohne Hilfe von außen feststellen lässt. So können Computer gegen sich selbst spielen und dabei lernen, ohne jemals einen Abgleich mit der Außenwelt vornehmen zu müssen. Diese Bedingungen sind jedoch jenseits der Welt von Spielen kaum anzutreffen und insofern ist zumindest nicht offensichtlich, welche Bedeutung ein Fortschritt beim Spielen für allgemeinere Fragestellungen der künstlichen Intelligenz hat.

Link

Verkauf künstlich künstlicher Intelligenz

Dass die Leistungen der Künstlichen Intelligenz noch recht beschränkt sind (Stand 2018), erkennt man auch daran, dass man im Internet für Aufgaben, deren Erledigung Intelligenz erfordert, Menschen buchen kann, die sich der Aufgaben annehmen. Dieser (nach dem ersten „Schachcomputer", der ebenfalls nicht künstlich war, erstaunlich diskriminierend anmutend) auch *Mechanical turk* genannte Dienst ist offenbar derzeit noch kosteneffektiver als ein Computer. Auch gibt es Überlegungen, Probleme, deren Lösung Intelligenz erfordert, in Computerspiele zu verpacken und so das menschliche Problemlösungspotential für andere Zwecke als die Unterhaltung zu nutzen (*Gamification*; s. dazu auch Kapitel 31 in Kurseinheit 4). Die Verwendung von Menschen als sog. Content-Moderatoren in „sozialen Netzwerken" schließlich kommt

---

[22] Man mag daraus ableiten, dass Spezialisierung zumindest mittelfristig keine Beschäftigungsgarantie ist, nämlich wenn sich Spezialisten einfacher durch künstliche Intelligenz ersetzen lassen als Generalisten.

einem Offenbarungseid von Unternehmen gleich, die sich ansonsten nur allzu gern mit ihrer Vorreiterschaft in künstlicher Intelligenz schmücken — wenn ein Schaden für das Kerngeschäft droht, verlässt man sich doch lieber auf das menschliche Urteil.

künstliche Wesen und der Turing-Test

Seit ein paar Jahren kann man einen Bedeutungswechsel des Begriffs „künstliche Intelligenz" beobachten: War mit „der künstlichen Intelligenz" früher zumeist eine Disziplin (Teilgebiet der Informatik) gemeint, so ist heute immer öfter von „einer künstlichen Intelligenz" die Rede, wobei damit eine Art Persona oder Avatar (also ein künstliches Wesen) gemeint ist. Historisch müsste eine solche künstliche Intelligenz, deren Idee schon einer der Urväter der Informatik, *Alan Turing*, in den Raum stellte, allerdings den nach ihm benannten **Turing-Test** bestehen, der verlangt, dass ein Mensch anhand einer Konversation per Tastatur und Bildschirm mit einem unbekannten Gegenüber nicht zuverlässig sagen kann, ob es sich bei dem Gegenüber um einen Menschen oder „eine künstliche Intelligenz" handelt.[23] Damit wird der Begriff der Intelligenz freilich auf den Inhalt zwischenmenschlicher Kommunikation reduziert, eine Definition, die auf ganz wesentliche Intelligenzleistungen des Menschen (wie etwa Erfindungen oder das Planen komplexer Handlungen oder — nicht zuletzt — die Schaffung eines formalen Wahrheitsbegriffs) nicht zutrifft. Trotzdem hat der Turing-Test als Maß für den Fortschritt in der Forschung zur künstlichen Intelligenz bis heute Bestand und wird in Form von Wettbewerben regelmäßig durchgeführt.

WIKIPEDIA

4 Mehr Zeichen und Zeichenketten

Dass zwei verschiedene Zeichen ausreichen, um Zahlen und Wahrheitswerte zu repräsentieren, bedeutet nicht, dass wir Menschen uns so ausdrücken möchten — nicht umsonst verwenden wir das Dezimalsystem und schreiben Wahr und Falsch nicht als «1» und «0». Außerdem sind Zahlen und Wahrheitswerte nicht der einzige Gegenstand der Digitalisierung — *Schriftzeichen*, also Buchstaben und Satzzeichen, und aus Schriftzeichen zusammengesetzte Texte gehören mindestens ebenfalls dazu.[24]

4.1 Codierung von Zeichen

Codierung und Code

Anstatt nun den Zeichenvorrat, der in Zeichenspielen verarbeitet werden kann, entsprechend zu erweitern (was auf dem Papier kein Problem wäre, aber bei der technischen Umsetzung erheblichen Aufwand bedeuten würde), bedient man sich einer eineindeutigen (d. h., umkehrbaren) Zuordnung der gewünschten Zeichen[25] zu *Bitfolgen*,

---

[23] Bezeichnenderweise geht es dabei ausdrücklich nicht darum, ob das Gegenüber den Wahrheitsgehalt von Aussagen zu bestimmen in der Lage ist, also etwa die Gesetze der Logik anwenden kann.

[24] Schriftzeichen heißen im Englischen übrigens „*character*".

[25] Gemeint sind hier tatsächlich Zeichen, nicht *Glyphen* (vgl. Kapitel 1); Glyphen werden ebenfalls als Folgen von Einsen und Nullen codiert (s. Kapitel 5), sind aber keine Zeichen.

also Folgen von «1» und «0», die, der besseren Zugänglichkeit für uns Menschen wegen, in der Regel als eine Zuordnung von Zeichen zu Zahlen (nämlich den Zahlen, die durch die Folgen von «1» und «0» ebenfalls repräsentiert werden) dargestellt werden. Eine solche Zuordnung wird gemeinhin **Codierung** und die einem zu codierenden Zeichen zugeordnete Bitfolge (ggf. als Zahl gelesen) **Zeichencode** oder auch nur **Code** genannt. Doch Achtung: Man verwechsele diese Verwendungen der Wörter Codierung und Code bitte nie mit *Verschlüsselung* oder *Chiffrierung* — es handelt sich bei Codes nicht um Geheimcodes!

Eine der bekanntesten Codierungen ist mit dem **American Standard Code for Information Interchange** (**ASCII**) gegeben, der 128 Zeichen den Dezimalzahlen **0**–**127** gegenüberstellt (eigentlich: siebenstelligen Dualzahlen — bei ASCII handelt es sich um einen 7-bit-Code). Dabei entsprechen die Buchstaben «A»–«Z» den Dezimalzahlen **65**–**90** und «a»–«z» **97**–**122**. Die Ziffern «0»–«9» des Dezimalsystems werden durch die Dezimalzahlen **48**–**57** codiert; dabei ist zwischen den dargestellten Zeichen «0»–«9» und den zur Darstellung des Zeichencodes verwendeten Zeichen «0»–«9» (bzw. «0» und «1» bei Darstellung des Zeichencodes als Bitfolge) zu unterscheiden — erstere sind extern (keine Zeichen eines Zeichenspiels) und müssen zur Verarbeitung (durch Zeichenspiele) codiert werden, letztere sind intern und dienen der Codierung.[26]

druckbare und nicht druckbare Zeichen

Nicht alle von den 128 Zeichen des ASCII sind druckbar; zu den nicht druckbaren zählen ein Signalton (ASCII-Code **7**), der Rückschritt (ASCII-Code **8**), der *Tabulator* (ASCII-Code **9**) sowie der *Wagenrücklauf* (ASCII-Code **13**; englisch „Carriage return" oder kurz „Return" genannt — daher der Name der Return-Taste!) und der *Zeilenvorschub* (ASCII-Code **10**; englisch „Line feed" genannt). An diesen nicht druckbaren Zeichen erkennt man gut, woher der ASCII rührt: von der Ansteuerung elektrischer Schreibmaschinen.[27]

Escape-Sequenzen

Auch wenn 128 Zeichen zunächst üppig bemessen erscheinen, sind es doch zu wenige, um alle Wünsche zu erfüllen. Um die Anzahl der Binärstellen (Bits) für die Zeichencodierung nicht erhöhen zu müssen (das achte Bit wurde ursprünglich als Prüfziffer, das sog. *Paritätsbit*, verwendet), hat man sog. **Escape-Sequenzen** eingeführt. Dabei folgt auf ein *Sonderzeichen*, das Escape-Zeichen (das zu den nicht druckbaren, d. h. glyphenlosen, Zeichen zählt; ASCII-Code **27**), ein oder mehrere weitere Zeichen (Länge der Folge nach der *Fano-Bedingung* bestimmt; s. Abschnitt 2.2), denen dadurch eine von ihrer eigentlichen

---

[26] Aus der Notwendigkeit der Codierung und der Interpretation der Codes als Zahlen rührt das Missverständnis, Computer könnten nur Zahlen verarbeiten — Computer können nur Folgen von «1» und «0» verarbeiten, die als Zahlen oder eben auch als Zeichen interpretiert werden können. Für weitere Interpretationen s. u.

[27] Streng genommen können Wagenrücklauf und Zeilenvorschub zumindest auf einer Schreibmaschine getrennt voneinander vorgenommen werden, auch wenn ein Wagenrücklauf ohne Zeilenvorschub bewirkt, dass dieselbe Zeile erneut beschrieben wird. Es ist übrigens ein großes Ärgernis, dass manche *Betriebssysteme* als Zeichen für eine neue Zeile Carriage return + Line feed (also zwei Zeichen) verwenden und andere nur Line feed. Dies führt immer wieder dazu, dass mehrzeilige Texte (mit „harten" Zeilenumbrüchen) in nur einer Zeile oder mit zusätzlichen Leerzeilen angezeigt werden.

Bedeutung abweichende zugeordnet wird. So steht beispielsweise die Folge «Escape»«M» (im ASCII dezimal 27 77) für einen umgekehrten Zeilenvorschub.

Escape-Sequenzen für Sonderzeichen

Escape-Sequenzen gibt es in anderen Codierungen auch für druckbare Zeichen; als Escape-Zeichen wird dann gern der Rückstrich (Backslash) «\» (ASCII-Code 92) verwendet. So sieht beispielsweise das Drucksatzprogramm Tex die Zeichenfolge «\»«"»«a» für den Umlaut «ä» vor. Übrigens: Wenn das Escape-Zeichen ein druckbares ist, es aber eine Escape-Sequenz einleitet, braucht man auch eine solche Sequenz, um es selbst zu drucken. Im Fall von «\» als Escape-Zeichen wird dann häufig «\»«\» verwendet. Man vergleiche dies mit dem eingangs Kapitel 1 geschilderten Dilemma des Schreibens über Zeichen mit Zeichen.

Im Folgenden werde ich Zeichenfolgen zu ihrer Darstellung im Text in doppelte gerade Anführungszeichen setzten, also etwa "ab" für «a»«b». Diese Anführungsstriche kann ich in solchen Zeichenfolgen naturgemäß selbst nicht verwenden, ohne einen Escape-Mechanismus zu bemühen — üblich ist hier die Verwendung eines Escape-Zeichen, also etwa "a\"b" für die Zeichenfolge «a»«"»«b». Die im Microsoft-Umfeld stattdessen anzutreffende Konvention, doppelte Anführungszeichen innerhalb einer Zeichenfolge doppelt anzuführen, also etwa "a""b" für «a»«"»«b», ist dagegen etwas gewöhnungsbedürftig.

4.2 Zeichenketten

Eine zusammenhängende Folge von Zeichen, oder *Zeichenfolge*, wird auch **Zeichenkette** (engl. **String**) genannt. Zeichenketten werden üblicherweise zur Darstellung (Codierung) von Wörtern, Sätzen und ganzen Texten verwendet; sie können aber auch der Steuerung von Geräten (wie beispielsweise einem Drucker) dienen. Man kann Zeichenketten als *Werte* (wie Zahlen, Wahrheitswerte oder Zeichen) auffassen, auf denen Zeichenspiele definiert sind (Abschnitt 4.4), oder als *Objekte*, die der Veränderung über die Zeit unterliegen (Abschnitt 4.5). Diese Unterscheidung ist wichtig und den Unterschied nicht zu kennen immer wieder Anlass zur Verwirrung.

Wörter in Zeichenketten

Für viele Problemstellungen ist es wesentlich, eine Zeichenkette, die einen Text darstellt, in kleinere Einheiten wie Sätze und Wörter unterteilen zu können. Zu diesem Zweck werden bestimmte Zeichen der Zeichenkette (wie das *Leerzeichen* oder auch *Tabulator*, *Wagenrücklauf* und *Zeilenvorschub*, zusammen manchmal als *White spaces* bezeichnet) als Trennzeichen betrachtet. Man beachte, dass auch nach einer solchen Unterteilung die Wörter eines Textes i. Allg. immer noch als Zeichenketten codiert werden, obwohl die Wörter einer Sprache selbst aufzählbar sind und man insofern für jedes Wort einen eigenen „Wortcode“ vorsehen könnte. Dies ist jedoch für die reine Darstellung von Texten (inkl. ihrer Verarbeitung wie beispielsweise in einem Textverarbeitungsprogramm) nicht sinnvoll. Etwas anderes ist es, wenn man den Wörtern eine Bedeutung zuordnen möchte, wie das beispielsweise für die Interpretation von Texten (inkl. deren Übersetzung) notwendig ist.

Die schriftliche Darstellung einer Zeichenkette als eine Zeichenfolge nennt man auch *Literal* oder, genauer, **Zeichenkettenliteral** (oder **Stringliteral**). Um sie von anderen Werten und auch von Operatoren abzugrenzen, verwenden Zeichenkettenliterale in der Regel Anführungsstriche als Begrenzungszeichen (s. o.). Analog nennt man die Darstellung einer Zahl als Zeichenfolge ein Zahlliteral; es unterscheidet sich von einem Zeichenkettenliteral in der Regel dadurch, dass es nicht in Begrenzungszeichen gesetzt wird.

4.3 Die Länge von Zeichencodes und von Zeichenketten

ASCII ist ein Code mit fester Stellenlänge (sieben Bit, wobei in der Praxis zumeist Erweiterungen auf acht Bit, ein Byte, verwendet werden). Eine Zeichenkette als eine Folge von Zeichen aus dem ASCII-Zeichensatz lässt sich somit als eine Folge von Bits darstellen, ohne dass die Bitfolgen, die jeweils ein Zeichen repräsentieren, dafür voneinander abgesetzt werden müssten (vgl. dazu Abschnitt 2.2 zu Zahlenfolgen).

Da die meisten natürlichen Sprachen aber mehr oder andere Zeichen haben, als es der ASCII vorsieht (und da es insgesamt deutlich mehr Schriftzeichen gibt, als sich mit sieben oder acht Bit darstellen lassen), werden heute andere Zeichensätze verwendet (mit aktuell bis über 130.000 verschiedenen Zeichen im *Unicode-Standard*). Da man aber nicht jedes Zeichen mit 17 oder mehr Bits codieren möchte, verwendet man kompaktere, kaskadierte Codierungen, die für die 128 Zeichen des ASCII mit acht Bit und für viele anderen Zeichen mit 16 Bit auskommen. Dabei zeigt das achte Bit des ersten Byte an, ob das Zeichen mit acht oder mit mindestens 16 Bit codiert wird (wobei eine Codierung mit 24 Bit dann in den zweiten 8 Bit codiert ist). Da die Länge eines Zeichencodes im Code so selbst festgelegt ist, ist auch hier klar, wo ein Zeichen aufhört und wo das nächste beginnt (sofern man nur die Stelle des ersten Codes kennt). Die derzeit am häufigsten verwendete Codierung des Unicode-Standards nennt sich *UTF-8*; da es aber auch andere Codierungen gibt, muss man immer mit angeben, welche verwendet wurde (die Verwendung einer anderen Codierung führt zu einer anderen Darstellung derselben Zeichenkette).

Neben der Länge der Codierung einzelner Zeichen muss auch die Länge von Zeichenketten irgendwie festgehalten werden (genau wie bei Zahlen; s. Abschnitt 2.2). Sofern eine Zeichenkette alle Zeichen enthalten kann, kommt eine Längenfestlegung durch ein Ende- oder Trennzeichen nicht in Betracht (da es ja Teil der Zeichenkette sein und somit an beliebiger Stelle stehen kann); sie muss dann durch die Paarung der Zeichenkette mit ihrer Länge, einer Zahl, erfolgen. Alternativ kann auch ein Code, dem kein Zeichen zugeordnet ist, zur Trennung verwendet werden. In der Praxis wird dafür gern **0** genommen; man spricht dann von **null-terminierten Zeichenketten**.

platzsparende Codierungen

Zeichenketten können sehr lang werden, insbesondere, wenn sie Texte darstellen. Da in Texten nicht alle Zeichen gleich häufig vorkommen, verschwenden alle auf fester Stellenzahl basierenden Zeichencodierungen (einschließlich Codierungen mit variabler Bytezahl wie UTF-8) Bits. Zeichencodierungen mit variabler Stellenzahl sind da sparsamer, da sie für häufig gebrauchte Zeichen weniger Stellen verwenden als für selten gebrauchte. Dafür müssen sie aber die Länge der Darstellung eines jeden Zeichens (der dafür verwendeten Bitfolge) bzw. die Trennung zwischen zwei Zeichen ebenfalls codieren. Der *Morsecode*, der anstelle von «1» und «0» Kurz und Lang verwendet, braucht dafür die Pause (vergleichbar mit einem Leerzeichen) als Trennzeichen; eine Codierung, die der *Fano-Bedingung* genügt, ist ebenfalls denkbar (z. B. die *Huffman-Codierung*). In der Praxis verwendet man jedoch aus verschiedenen Gründen lieber Zeichencodierungen mit fester Stellenzahl (wie ASCII) oder, als Kompromiss, Codierungen mit variabler Bytezahl; der (verbleibenden) Verschwendung, wenn sie denn wehtut, begegnet man durch *Kompression* (s. Kapitel 7).

WIKIPEDIA

WIKIPEDIA

4.4 Operationen auf Zeichenketten

Genau wie Zahlen lassen sich also Zeichen und Zeichenketten als Folgen von Einsen und Nullen darstellen. Ob eine solche Bitfolge als Zahl oder Zeichenkette (Text) interpretiert wird, hängt zum einen vom Betrachter ab und zum anderen davon, was man damit tut. Mit Zahlen kann man rechnen; mit Zeichenketten könnte man dies auch, aber das Ergebnis erscheint wenig sinnvoll — was ergäbe beispielsweise die Addition von "Haus" und "arbeiten"?

Für Zeichenketten gibt es andere *Operationen*, die für die Digitalisierung nicht weniger von Bedeutung sind. Eine der einfachsten ist die **Verkettung** (Aneinanderreihung oder **Konkatenation**) von zwei Zeichenketten zu einer, die so lang ist wie die beide anderen zusammen. So wird aus "Haus"*"arbeiten" (wobei * hier für die Verkettung steht) "Hausarbeiten". Weitere Operationen sind das Einfügen eines Zeichens oder einer Zeichenkette in eine andere Zeichenkette an einer bestimmten Stelle, das Löschen von einem oder mehreren Zeichen ab einer Position innerhalb einer Zeichenkette oder die Bestimmung der Position des Vorkommens eines Zeichens oder einer Zeichenkette innerhalb einer anderen Zeichenkette. Auch kann man Zeichenketten vergleichen: Zwei Zeichenketten sind gleich, wenn sie gleich lang und ihre Zeichen stellenweise gleich sind. Ob eine „größer" als eine andere ist, ist eine Frage des Kontextes — je nach Aufgabenstellung können hier verschiedene Definitionen zur Anwendung kommen. Benötigt wird der „Größenvergleich" typischerweise bei der Sortierung mehrerer Zeichenketten (beispielsweise der Einträge in einem Telefonbuch).

Zahlen als Zeichenketten

Eine besondere Operation auf Zeichenketten ist die Umwandlung von einer als Zeichenkette vorliegenden Dezimalzahl in die binäre Repräsentation einer Zahl, mit der man nach den Regeln aus Abschnitt 2.4 rechnen kann. Es wird nämlich beispielsweise die Zeichenfolge "10" im ASCII-Code als Bitfolgenpaar **0110001 0110000** (oder als Paar von Dezimalzahlen **49 48**) dargestellt, zum Rechnen müsste sie aber (bei einer festen Zahlenlänge von acht Bit) als **00001010** im *Dualsystem* oder als **00010000** im *BCD-*

Format (s. Abschnitt 2.6.2) dargestellt werden. Um von der Darstellung als Zeichenfolge zur Darstellung im BCD-Format zu gelangen, kann man zunächst von der Codierung der beiden Zeichen durch die beiden Dualzahlen **00110001** und **00110000** jeweils **00110000** (**48**) subtrahieren und erhält dann **00000001** bzw. **00000000**; die Stellen der ersten Dualzahl verschiebt man dann um vier Stellen nach links (wobei die vier leeren Stellen rechts mit «0» besetzt werden) und addiert das Ergebnis (**00010000**) zur zweiten (**00000000**) hinzu, so dass man wie gewünscht **00010000** erhält. Die Konvertierung einer Zeichenfolge, die eine Zahl im Dezimalsystem darstellt, in eine gleichwertige Dualzahl ist dagegen aufwendiger; trotzdem handelt es bei beiden Verfahren lediglich um Zeichenspiele.

Selbsttest: Konvertieren Sie nach obigem System die Darstellung der Dezimalzahl **42** als Zeichenfolge "42" in eine BCD-Zahl.

4.5 Manipulation von Zeichenketten

Aneinanderhängen und Einfügen von Zeichenketten kann man als Operationen auffassen, die, analog zu den arithmetischen Operationen Addition, Subtraktion etc. (Abschnitt 2.4), zwei Werte entgegennehmen und einen dritten liefern. Genauso gut kann man sie aber auch als Operationen auffassen, die, analog zu Inkrement und Dekrement (Abschnitt 2.7), den Inhalt eines *Behälters*, oder ein *Objekt*, abändern: Die Zeichenkette ändert sich gewissermaßen über die Zeit, wie es beispielsweise bei einer Textverarbeitung geschieht. Da sich bei solchen Operationen jedoch die Länge der Zeichenkette ändert, muss u. U. (je nach Organisation) beim Spiel mit den Zeichen einiges auf dem Papier verschoben (ausradiert und neu geschrieben) werden (dies umso mehr, wenn der Platz im Behälter nicht ausreicht, um eine längere Zeichenkette aufzunehmen). Mit der platz- und zeitsparenden Umsetzung von Operationen auf und Manipulation von Zeichenketten, wie auch der Suche in Zeichenketten, haben sich schon Generationen von Programmierern ausgiebig befasst.

5 Grafiken und Bilder

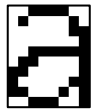
«1» und «0» können nicht nur für Eins und Null stehen, sondern auch für Schwarz und Weiß oder Hell und Dunkel (oder umgekehrt). Die Zeichenfolge "1010101110111010101" etwa entspricht bei solch einer Interpretation z. B. dem Streifen

wenn man jedem Zeichen der Folge ein Quadrat zuordnet und die Quadrate nebeneinander anordnet.

Wenn man einen Streifen wie den obigen in gleichen Abständen umbricht, entsteht ein in Zeilen und Spalten gegliedertes Rechteck. Die Kästchen, die die Fläche dieses Rechtecks bilden und die schwarz oder weiß sein können, sind dann die Punkte, oder Elemente, einer

gerasterten Grafik oder eines gerasterten Bildes. Sie werden englisch *Picture elements* oder kurz *Pixels* sowie deutsch **Pixel** genannt. So entsteht beispielsweise aus dem 80 Pixel umfassenden Streifen

durch Umbruch nach jedem achten Pixel das acht Pixel breite und zehn Pixel hohe Rechteck



Solche gerasterten Grafiken oder Bilder, wegen ihrer Darstellung als umgebrochene Bitfolgen auch **Bitmaps** genannt, werden u. a. (und wie das obige Beispiel nahelegt) verwendet, um codierte Schriftzeichen (Kapitel 4) in Form von *Glyphen* auf einem (ebenfalls gerasterten) Papier oder Bildschirm darzustellen. Dazu wird, im einfachsten Fall, jede Glyphe als ein Tripel bestehend aus Höhe, Breite und Muster codiert, wobei Höhe und Breite als (binär codierte) ganze Zahlen und das Muster als eine Bitfolge entsprechend dem obigen Streifen in einer Tabelle hinterlegt werden.[28] Die Darstellung eines durch einen Code bestimmten Zeichens an einer bestimmten Position der Anzeige (wiederum bestimmt durch Spalte und Zeile, oder x- und y-Koordinate) erfolgt dann, indem zunächst über den Zeichencode die Codierung der Glyphe als Bitfolge in der Tabelle nachgeschlagen und dann die Bitfolge an die gegebene Position kopiert wird. Das dabei zu lösende Problem ist, dass die Stellen auf dem Ausgabemedium (Blatt Papier oder Bildschirm), die das Bitmuster der Glyphe aufnehmen sollen, anders als die der Glyphe selbst nicht alle nebeneinander liegen, so dass die Glyphe abschnittsweise in Zeilen des Mediums kopiert werden muss. Dieser Kopiervorgang, der auch für andere Aufgaben benötigt und der auch *Bit Block Image Transfer* genannt wird, ist eines der am häufigsten von einem Computer durchgeführten Zeichenspiele. Man beachte, dass er kaum etwas mit Rechnen zu tun hat (und die Bezeichnung „Rechner" daher eine etwas unglückliche Verkürzung ist).

graue und farbige Pixel

Auf vielen Ausgabemedien können Pixel nicht nur schwarz oder weiß sein, sondern auch grau oder farbig. Ein Bildpunkt wird dann nicht durch ein Bit repräsentiert, sondern durch eine Bitfolge, die die Erscheinung des Bildpunktes bestimmt. Gängige Codierungen verwenden 8 Bit für 256 Grauwerte oder 24 Bit für 16.777.216 Farben, gewöhnlich in Rot-, Grün- und Blauwerte (RGB) à 8 Bit aufgeteilt (wobei ein Pixel dann aus drei unterschiedlich farbigen Punkten besteht, die allerdings so dicht beieinanderliegen, dass sie normalerweise als ein Punkt wahrgenommen werden). Um den Inhalt eines Farbbildschirms mit voller HD-Auflösung (1080 Zeilen und 1920 Spalten, entsprechend gut 2 Millionen Pixeln) zu repräsentieren, braucht man knapp 50 Millionen Bits, was

[28] In textbasierten (d. h., nicht grafikfähigen) Anzeigen (wie z. B. den sog. *Character terminals*) haben alle Glyphen dieselbe Höhe und Breite und die Anzeige ist in eine feste Anzahl von Zeilen und Spalten aufgeteilt, an deren Kreuzungspunkten (Zellen) jeweils eine Glyphe erscheinen kann.

mehr als 6 Megabyte entspricht. Eine DIN-A4 Seite mit 300 dpi (Dots, oder Pixel, per inch) benötigt schon mehr als 200 Millionen Bits (knapp 25 Megabyte).

Vektorgrafiken

Als Muster codierte Grafiken haben den Nachteil, dass sie bei Vergrößerung schnell „pixelig“ werden, man also die Bildelemente wie beim oben abgebildeten „a“ als solche wahrnimmt. Das gilt auch für Glyphen, deren Größe man zur Darstellung unterschiedlicher Schriftgrößen variieren können möchte. Anstatt nun für jede Größe ein eigenes Muster zu hinterlegen (oder aus einem Muster größere zu berechnen, was aber zu „Pixeligkeit“ oder anderen Artefakten führt; so ist z. B. obiges Muster die 12,5-fache Vergrößerung des weniger pixelig erscheinenden „a“), kann man die Muster auch geometrisch beschreiben. Aus diesen geometrischen Beschreibungen werden dann die für eine Anzeige benötigten Bitmuster fallweise berechnet. Dies benötigt naturgemäß mehr Zeit, führt aber stets zu glatt erscheinenden Formen. Während diese sog. **Vektorgrafiken** gut für *Glyphen* (sie liegen den sog. *Typ-1-* und den *TrueType-Fonts* zugrunde) und Diagramme geeignet sind, eignen Sie sich jedoch kaum für Bilder (Gemälde, Fotografien oder Filme).

6 Signale

Viele Signale lassen sich direkt mithilfe von Zeichen oder Zahlen codieren, so z. B. die Signale im Schienen- und Straßenverkehr oder die Fehlermeldungen eines Computerprogramms (der berühmt-berüchtigte Fehler 404 von Webservern beispielsweise). Etwas anders ist das mit Signalen wie Schall-, Licht- oder elektrischen Wellen — hier handelt es sich um einen *Signalstrom*, der in der kontinuierlichen Veränderung einer wahrnehmbaren oder messbaren physikalischen Größe, also etwa in sich veränderndem Luftdruck oder in der schwankenden Amplitude elektromagnetischer Wellen, codiert ist. Auch diese Signale werden von Computern verarbeitet und müssen zu diesem Zweck in Zeichenfolgen umgewandelt, oder *umcodiert*, werden.

Diskretisierung

Dies geschieht mithilfe der sog. **Diskretisierung**. Dabei wird ein Messwert für einen Augenblick eingefroren und in eine Zahl umgewandelt, die diesen Messwert repräsentiert. Dabei wird der Messwert umso genauer wiedergeben, je mehr Stellen die für seinen Wert vorgesehene Zahl hat: Eine Diskretisierung mit nur einem Bit kann nur zwei Messwerte (etwa Licht an oder aus; allgemeiner, ob der Messwert über einem gegebenen Schwellwert liegt oder nicht) wiedergeben, eine Diskretisierung mit acht Bit hingegen schon 256 usw. Höhere Auflösungen (entsprechend höheren Bitzahlen) führen dabei nicht unbedingt zu einem besseren Ergebnis, da die Messung nicht nur selbst den Messwert geringfügig verfälscht, sondern auch zufälligen Einflüssen (dem sog. Rauschen) unterliegt, die man in aller Regel nicht mit codieren möchte. So reichen für die Codierung von Schallwellen fürs Telefonieren schon acht Bit und für Audio-CDs werden die Schallsignale mit einer Auflösung von 16 Bit diskretisiert.

Zeitreihen

Wie das obige Beispiel vom Ton (Telefonieren und CDs) schon nahelegt, reicht es bei der Codierung von Signalströmen nicht aus, lediglich einen Momentanwert in

eine Zahl zu überführen — die eigentliche Information steckt hier in der Änderung des Wertes über die Zeit. Genau wie der Messwert selbst erscheint uns die Zeit aber kontinuierlich und Digitalisierung bedingt, dass auch die zeitliche Dimension diskretisiert, also in eine Folge von Zeitpunkten aufgeteilt wird. Die Diskretisierung wird deswegen mit einer vorgegebenen Frequenz wiederholt, und diese Wiederholung, **Abtastung** genannt, führt zu einer *zeitlichen Folge* von Zahlen, auch *Zeitreihe* genannt, die zum Zweck der Aufzeichnung in eine räumliche Zahlenfolge konvertiert wird.

Wahl der Abtastrate

Analog zur Frage nach der Auflösung der Messgröße ergibt sich somit die Frage nach der Auflösung der Zeit: Mit welcher Frequenz soll man ein Signal abtasten? Während die Antwort auf diese Frage vom Verwendungszweck abhängt, gibt es doch für einen bestimmten Zweck, nämlich die originalgetreue Reproduktion des Signals aus der Abtastfolge, einen fundamentalen Satz: Dafür genügt nämlich eine Abtastrate, die etwas mehr als doppelt so hoch ist wie der höchste in dem Signal vorkommende Frequenzanteil (das **Abtasttheorem**). Wenn also beispielsweise die höchste Frequenz in einem Signal bei 10 kHz liegt, dann reicht es, das Signal etwas mehr als 20.000-mal in der Sekunde abzutasten, um das Originalsignal aus der Folge der Messwerte zu rekonstruieren. Auch wenn eine solche Bandbegrenzung auf Seiten der Signalquelle nicht vorliegt, so kann doch eine Bandbegrenzung der signalverarbeitenden Stelle ausgenutzt werden, um Signale in einer für die Stelle vom Original nicht zu unterscheidenden Qualität zu reproduzieren: So beträgt die Abtastrate der Audiosignale auf CDs 44.1000 Hz, was bei einer Grenze des menschlichen Hörvermögens von ca. 22 kHz ausreicht, damit Menschen die Reproduktion der Signale nicht vom Original unterscheiden können. Übrigens: Bei einer Auflösung der Diskretisierung von 16 Bit und einer Codierung unseres Alphabets mit 5 Bit entspricht eine Sekunde Tonsignal von einer CD knapp 120.000 Anschlägen oder knapp 30 Schreibmaschinenseiten. Man kann sich also leicht vorstellen, dass die Kapazität unseres Gehirns nicht ausreicht, um selbst das bandbegrenzte Audiosignal in all seinen Details vollständig wahrzunehmen.

WIKIPEDIA

digitale Signalverarbeitung

Als Bitfolgen codiert lassen sich Signalströme mittels Zeichenspielen verarbeiten. Diese **digitale Signalverarbeitung** kann als einer der zentralen Beiträge zur Digitalisierung verstanden werden, wird aber in diesem Kurs nicht weiter behandelt.

7 Kompression

verlustbehaftete Kompression

Während das *Abtasttheorem* aussagt, wie dicht man ein Signal abtasten muss, um es technisch exakt rekonstruieren zu können, gelten für Signale, die ausschließlich für den Konsum durch Menschen bestimmt sind, andere Regeln: Hier geht es nicht um objektiv, sondern um subjektiv nicht vom Original unterscheidbare Reproduktion von Signalen. Dabei lässt sich ausnutzen, dass unsere Nervenzellen den originalen Signalstrom selbst dann nicht vollständig verarbeiten können, wenn er bandbegrenzt ist. Wenn man die Mechanismen der Verarbeitung kennt, kann man sie ausnutzen, um den Inhalt der Signale zu reduzieren, ohne dass deren Konsum davon wesentlich beeinträchtigt

wird. Zeichenfolgen, die den originalen Signalverlauf repräsentieren, werden so vor ihrer Speicherung **komprimiert** und vor einer späteren Wiedergabe **expandiert**.

Die wohl bekanntesten Kompressionsformate für mediale Daten (Zeichenfolgen) sind *JPEG* (für stehende Bilder), *MPEG* (für bewegte Bilder) und *MP3* (für Ton). Sie sind allesamt *verlustbehaftet*, d. h., die Originale lassen sich daraus nicht exakt rekonstruieren. Übrigens: Bei „JPEG" und „MPEG" handelt es sich eigentlich um die Namen von Standardisierungsgremien, der Joint Photographic Experts Group und der Moving Picture Experts Group; „MP3" ist eigentlich die Bezeichnung des dritten Standards, der für die Kompression des Tons von MPEG-komprimierten Filmen entwickelt wurde, der aber ein prominentes Eigenleben entwickelt hat.

verlustfreie Kompression

Jenseits des Medienkonsums sind Kompressionsverluste in der Regel nicht hinnehmbar. Die *verlustfreie Kompression* erlaubt daher die exakte Reproduktion der originalen Zeichenfolgen aus den komprimierten. Sie nutzt bestimmte Regelmäßigkeiten in den Zeichenfolgen wie beispielsweise eine (generell beobachtbare) Ungleichverteilung der relativen Häufigkeit von Zeichen (der auch das *Morsealphabet* zugrunde liegt) oder immer wiederkehrende Muster (wie beispielsweise lange Folgen von Nullen oder Einsen, die man dann durch ihre als Zahl codierte Länge ersetzt; beispielsweise bei der Fax-Codierung verwendet). Rein zufällige Folgen gleichverteilter Zeichen lassen sich dagegen überhaupt nicht komprimieren. Interessanterweise bringt die Anwendung verlustfreier Kompressionsverfahren auf Zeichenfolgen, die den Ton von gesprochener Sprache oder von Musik codieren, nur wenig (*FLAC* beispielsweise komprimiert in etwa auf die Hälfte der originalen Größe) — es lassen sich darin kaum Regelmäßigkeiten erkennen, was darauf hindeutet, dass die zur Verfügung stehende Kapazität zur Informationsübertragung von gesprochener Sprache und insbesondere Musik grundsätzlich sehr gut ausgenutzt wird. Geschriebene und als Zeichenfolge mit fester Zeichencodelänge codierte Texte hingegen lassen sich in der Regel stark komprimieren.

8 Verschlüsselung

Eng verwandt mit der *Kompression* ist die **Verschlüsselung** von Zeichenfolgen, auch **Chiffrierung** genannt. Dabei wird aus einer vorgegebenen, zu verschlüsselnden Zeichenfolge mithilfe eines Schlüssels eine neue Zeichenfolge erzeugt, aus der sich, wiederum mithilfe eines Schlüssels, die ursprüngliche Zeichenfolge rekonstruieren lässt. Einfache Verschlüsselungen tauschen Zeichen nach einem, durch die Schlüssel mitbestimmten Schema gegen andere aus oder ändern ihre Reihenfolge. Nach einem solchen Prinzip funktionierte beispielsweise die Verschlüsselungsmaschine *Enigma*; wie man leicht erkennt, handelt es sich auch dabei nur um Zeichenspiele.

ideale Verschlüsselung

Bei einer idealen Verschlüsselung ist es unmöglich, die originale Zeichenfolge aus der verschlüsselten Zeichenfolge ohne Kenntnis des Schlüssels zu rekonstruieren oder den Schlüssel zu erraten, d. h., aus der verschlüsselten Zeichenfolge

Hinweise auf den Schlüssel abzuleiten.[29] Dabei ist interessanterweise gerade die Ungleichverteilung der Häufigkeit von Zeichen in Zeichenfolgen (die ja die Grundlage vieler Kompressionsverfahren darstellt) der Schlüssel zum Erraten des Schlüssels. Bei der Verschlüsselung versucht man daher, diese Ungleichverteilung zu beseitigen — die verschlüsselte Zeichenfolge sollte von einer zufälligen möglichst nicht zu unterscheiden sein.

symmetrische vs. asymmetrische Verschlüsselung

Ein Problem der gewöhnlichen, **symmetrischen Verschlüsselung** ist, dass beiden Seiten, der verschlüsselnden und der entschlüsselnden, der Schlüssel bekannt sein muss. Der Schlüssel muss dazu selbst übermittelt werden, was, wenn diese Übermittlung die Verschlüsselung nicht gefährden soll, dasselbe Problem beinhaltet: Die Übermittlung muss geheim, d. h. verschlüsselt, erfolgen. Bei der sog. **asymmetrischen Verschlüsselung** publiziert der Empfänger einer Nachricht einen, seinen, öffentlichen Schlüssel, den jeder Sender zur Verschlüsselung von Nachrichten an ihn (und nur an ihn) verwenden kann; dieser Schlüssel taugt jedoch nicht zur Entschlüsselung der Nachricht — dazu ist der zum öffentlichen Schlüssel passende private Schlüssel notwendig, den nur der Empfänger kennt.

ungelöstes Problem

Verschlüsselung ist ein eigenes Forschungsgebiet, das auf Anleihen aus der Zahlen- und der Informationstheorie basiert. Sie ist eine der Grundlagen der sog. *digitalen Transformation*, wird dabei aber von den meisten Menschen nicht besonders ernstgenommen. So bleiben z. B. die meisten der heute verschickten Emails unverschlüsselt (Motto: „Ich habe nichts zu verbergen!“). Endschuldigend könnte man anführen, dass kein bis heute verwendetes Verfahren zur Verschlüsselung wirklich sicher ist — so ist es z. B. schwierig, sicherzustellen, dass ein (beispielsweise auf einer Webseite) publizierter öffentlicher Schlüssel einer Person auch wirklich zu der Person gehört (eine Frage der *Authentizität*; vgl. dazu auch Kapitel 15 in Kurseinheit 2). So betrachtet man Verschlüsselungen allgemein als sicher, wenn der Aufwand, sie zu knacken, in keinem Verhältnis zum Nutzen steht. Nach dieser Definition können allerdings sichere Verschlüsselungen schnell unsicher werden, nämlich wenn der Nutzen ihrer Überwindung ansteigt.[30]

9 Programme

Um Zeichenspiele durchzuführen braucht es Spieler. Spätestens mit der nächsten Kurseinheit sollte endgültig klar werden, dass diese Rolle im Kontext der Digitalisierung Maschinen zugedacht ist: den Computern.

---

[29] Blind erraten kann man einen Schlüssel natürlich immer, aber bei einem idealen Verschlüsselungsverfahren bräuchte man dazu schon so viel Glück, dass man nicht ernsthaft darauf setzen kann, auch wenn man alle verfügbaren Computer gleichzeitig zum Raten einsetzt.

[30] Gleiches gilt übrigens auch für die (ebenfalls, aber nicht nur auf Verschlüsselung basierende) Blockchain-Technik: Auch hier ist die Sicherheit nur relativ.

Nun ist es nicht sinnvoll, in eine Maschine alle nützlichen Zeichenspiele fest einzubauen — das geht vielleicht noch bei einem Taschenrechner, aber für jeden anderen Zweck eine andere Maschine mit anderen eingebauten Zeichenspielen herzustellen wäre eine ziemlich teure Angelegenheit. Da ist es günstiger, eine Maschine für verschiedene Zeichenspiele programmierbar zu machen. Eine solche Maschine nennt man **Computer**.

Programm

Die sich unmittelbar ergebende Frage ist dann allerdings, was ein **Programm** ist und wie es ein Computer in ein Zeichenspiel umsetzen kann. Die Antwort ist genauso einfach wie genial: Ein Programm ist selbst eine Zeichenfolge und ein Zeichenspiel daraus abzuleiten nichts weiter als ein (anderes) Zeichenspiel. *Ein einziges fest eingebautes Zeichenspiel, nämlich das, das Programme ausführt, reicht also aus, um alle Zeichenspiele, die durch die Programme beschrieben werden können*[31]*, umzusetzen*. Aus dieser Universalität ergibt sich der Erfolg der Computer — man braucht einfach nichts anderes!

Multiplikation als Programm

Dazu ein einfaches Beispiel. Ein einfaches Programm, das zwei Zahlen mittels wiederholter Addition multipliziert (wie in Abschnitt 2.4.2 nahegelegt), sieht im **Maschinencode** eines bestimmten *Prozessors*, nämlich des *6502-Prozessors*, etwa so aus:

Stelle	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E
Inhalt	A9	00	A6	0C	65	0D	CA	D0	FB	85	0E	60	?	?	?
Mnemo	LDA	#00	LDX	0C	ADC	0D	DEX	BNE	-03	STA	0E	RTS			

Hierbei enthält die Zeile *Stelle* die Nummern von (aufeinanderfolgenden) 8-Bit *Behältern* (Speicherzellen in Byte-Größe) und die Zeile *Inhalt* den Inhalt der Behälter als zweistellige *Hexadezimalzahl*. Diese Zahlen (richtiger: die sie repräsentierenden Bitfolgen) codieren die *Operationen*, oder **Maschinenbefehle**, mittels sog. *Operationscodes* (beim 6502 ein Byte lang) und, sofern vorhanden, ihre Operanden (die hier alle ebenfalls ein Byte lang sind). Die letzten drei Spalten nehmen der Reihe nach den Multiplikator (Behälter 0C), den Multiplikanden (Behälter 0D) und das Produkt (Behälter 0E) auf, wobei Multiplikator und Multiplikand Eingaben und das Produkt Ausgabe des Programms sein sollen. Die dritte Zeile der Tabelle enthält die sog. *Mnemonics*, das sind leicht(er) merkbare Zeichenfolgen, die von Programmierern anstelle der Operationscodes aus der mittleren Zeile verwendet werden können. Dabei kann ein Mnemonic für verschiedene Operationscodes stehen; welcher gemeint ist, hängt dann von den Operanden ab (s. u.). Die Summe der Operationscodes eines Prozessors bezeichnet man auch als seine **Maschinensprache**.

die Spiele der Operationscodes

Das durch das obige Programm beschriebene Zeichenspiel lässt sich nachvollziehen, wenn man weiß, was die Operationscodes bzw. Mnemonics bedeuten, d. h., für welche Zeichenspiele sie selbst stehen. Die erste Operation des Programms, LDA (für LoaD Accumulator) lädt ihren Operanden, das nachfolgende Byte, in einen Behälter, der *Akkumulator* genannt wird und der u. a. dem Aufaddieren von Zahlen

[31] und das sind alle, die überhaupt umgesetzt werden können, so zumindest die *Church-Turing-These*

dient. Der Operand #00 gibt hier an, dass die Zahl **0** in den Akkumulator geladen werden soll; dabei bedeutet das vorangestellte Hash-Zeichen, dass **0** als Wert und nicht als **Adresse** (Nummer) eines Behälters interpretiert werden soll. Dies ist bei der nachfolgenden Operation, LDX, (für LoaD X-register), anders: Sie lädt den Inhalt des Behälters mit der nachfolgenden Nummer (hier 0C, der Behälter für den Multiplikator) an eine Stelle, die X-Register genannt wird. Die nächste Operation, ADC (für Add with *Carry*, wobei Carry das englische Wort für *Übertrag* ist), addiert den Wert des Behälters, der durch den Inhalt OD des auf den der Operation folgenden Behälters adressiert ist (der Behälter für den Multiplikanden) zum Wert des Akkumulators. DEX (für *Decrement* X-Register) tut genau das; der Inhalt des X-Registers wird um **1** verringert (vgl. Abschnitt 2.7; dort wird auch auf den Begriff des Behälters in Zeichenspielen eingegangen). Die nächste Operation, BNE (für Branch on Not Equal, oder verzweige wenn ungleich Null[32]) hat es in sich: Wenn die vorangegangene Operation ein anderes Ergebnis als Null hat, dann setzt das Programm an der Stelle fort, die drei Stellen zurück liegt, also bei der Stelle 04. Addition des Multiplikanden und Dekrement werden also so oft wiederholt (in einer sog. *Schleife*), wie der Inhalt von Behälter 0C (der Multiplikator) dies vorgibt. Ist dies geschehen, wird der Inhalt des Akkumulators mittels der Operation STA (für STore Accumulator) in den Behälter, der durch ihren Operanden adressiert wird (Behälter 0E, für das Produkt), geschrieben. Die letzte Operation, RTS (ReTurn from Subroutine) beendet das Zeichenspiel.

Unterprogramm

Bezeichnenderweise heißt die das Zeichenspiel beendende (oder abschließende) Operation RTS und nicht END, STP oder HLT. Das hat die folgende Bewandtnis. Alle durch die Operationscodes bzw. Mnemonics des obigen Programms angesprochenen Zeichenspiele (LDA, ADC, DEX etc.) sind in einem 6502-Prozessor fest eingebaut (wie das technisch vollzogen wird ist Gegenstand von Kurseinheit 2, Kapitel 11). Das obige Programm beschreibt aber selbst ein (zusammengesetztes) Zeichenspiel, nämlich das, das die Multiplikation zweier Zahlen umsetzt. Damit dieses Zeichenspiel ähnlich wie ein eingebautes von wieder anderen Zeichenspielen verwendet werden kann, kann es von anderen Programmen mittels einer besonderen Operation, JSR (für Jump to SubRoutine), aufgerufen werden. Operand dieser Operation ist die Adresse (Nummer) des Behälters, an der das Programm beginnt; zum Aufruf des obigen, an Stelle 00 beginnenden Programms wäre also JSR 00 zu verwenden.[33] Per Ausführung von RTS wird dann das aufrufende Programm an der Stelle, die auf den Programmaufruf folgt, fortgesetzt. Da das aufgerufene Programm so zu einem

---

[32] Equal heißt natürlich nicht Null; die Verwendung von Equal leitet sich daraus ab, dass zwei Zahlen genau dann gleich sind, wenn ihre Differenz Null ist (vgl. Abschnitt 2.4.5). In der Informatik wimmelt es vor Fehlbenennungen, die allerdings meistens historisch erklärt werden können.

[33] Eigentlich müsste da JSR 0000 stehen, weil die Adressen von Sprüngen und Unterprogrammaufrufen beim 6502-Prozessor immer 16 Bit, entsprechend 4 Stellen im Hexadezimalsystem, haben müssen. Die Adressierung von Behältern im obigen Programm bezieht sich auf die sog. Zero page des 6502, bei der die höheren 8 Bit immer «0» sind. Es gibt für die Operationen, die die Zero-page-Adressierung verwenden, eigene Operationscodes; da sie nur ein Byte für die Adressierung der Operanden benötigen, sind die Befehle kürzer und können schneller ausgeführt werden (sie benötigen weniger *Takte*).

Teil des aufrufenden wird, nennt man es auch **Unterprogramm**. Dabei können Unterprogramme, genau wie die fest eingebauten Operationen, von beliebigen Programmen, auch von anderen Unterprogrammen und sogar von sich selbst, verwendet werden.

es geht auch besser

Das obige Unterprogramm (als das wir es nunmehr entlarvt haben) ist in mehrfacher Hinsicht unbefriedigend. Zum einen ignoriert es einen möglichen *Überlauf* (und allgemein den *Übertrag*) und kann somit nur kleine Zahlen korrekt multiplizieren (selbst dafür wäre ein Zurücksetzen des *Carry-Bits* vor der ersten Addition notwendig). Zum anderen müssen die Operanden der Multiplikation an im Programm festgeschriebenen Stellen hinterlegt sein (hier 0C und 0D), und auch das Ergebnis steht hinterher an einer festgeschriebenen Stelle (0E). Vor und nach einem Unterprogrammaufruf müssen die Ein- und Ausgaben an diese festen Stellen geschrieben bzw. aus ihnen gelesen werden. Dies ist bei den fest eingebauten Operationen anders: Die Operanden sind hier immer entweder impliziter Teil der Operation (wie z. B. der Akkumulator oder das X-Register) oder Teil des Aufrufs. Prozessoren sehen daher Mechanismen vor, wie man das für Unterprogrammaufrufe ähnlich gestalten kann; dies zu betrachten würde jedoch an dieser Stelle zu weit gehen (und ist beim 6502 auch nicht sonderlich ausgeprägt).

Zusammenfassung

Wenig überraschend ist Programmieren selbst nicht mehr als ein Zeichenspiel: Operationen wie etwa der Addition oder der Multiplikation werden Zeichen zugeordnet, deren Bedeutung entweder primitiv (also im Computer fest eingebaut) oder programmiert, also als Zeichenspiel definiert ist. Wie man sich leicht vorstellen kann, verlangt die Programmierung einen Anfang: Der Computer muss einen Satz von fest eingebauten Operationen (elementaren Schritten eines jeden Zeichenspiels) vorsehen, der hinreicht, um alle denkbaren Zeichenspiele als eine Folge solcher Operationen (Schritte, ggf. mit *Verzweigungen* und *Schleifen*) darzustellen. In der Theorie genügt dazu bereits eine sehr kleine Menge von primitiven Operationen (vielleicht haben Sie schon einmal von der *Turingmaschine* gehört); in der Praxis werden jedoch — aus Effizienzgründen — andere Operationen eingebaut, manchmal auch solche, die sich auf bereits eingebaute zurückführen lassen. Die Einführung von Unterprogrammen und deren Aufruf erlaubt dabei, den Satz von eingebauten Operationen beliebig um programmierte zu erweitern.

WIKIPEDIA

Dies soll an dieser Stelle genügen; Kapitel 11 in Kurseinheit 2 befasst sich mit der Umsetzung von Zeichenspielen in *Hardware* allgemein und Kurseinheit 3 ist schließlich ganz der Programmierung gewidmet, wenn auch auf einem wesentlich *höheren Abstraktionsniveau*. Gleichwohl ist die Programmierung in Maschinencode eine der unmittelbarsten Erfahrungen, die man mit Computern machen kann und die man sich, genügend Zeit und Interesse vorausgesetzt, unbedingt geben sollte. Am besten eigenen sich dafür ältere Computer, die für den Hausgebrauch (Spiele!) entwickelt wurden, da man bei diesen noch einen direkten

Kurs

Zugriff auf Ein- und Ausgabegeräte (Tastatur und Bildschirm) hat und somit die rohe Kraft (Geschwindigkeit) der Hardware erleben kann.[34]

10 Segen und Fluch des Binären

Binnensicht

Wie wir gesehen haben, reichen zwei verschiedene Zeichen aus, um viele unterschiedliche Dinge darzustellen. Wie wir weiterhin gesehen haben, wird die Bedeutung der Zeichen *innerhalb eines Systems* allein durch ihre Verwendung festgelegt — manche Vorkommen der Zeichen dienen der Darstellung von Zahlen, andere der Darstellung von Wahrheitswerten und wieder andere der Darstellung von Texten, Ton oder Bildern. Zu jeder dieser Verwendungen gehören Zeichenspiele, die festlegen, was man mit den Zeichen sinnvoll machen kann: Rechnen, Schlussfolgern, Text- oder Signalverarbeitung. Dabei sind die Spiele selbst nicht wählerisch, was die *von außen* bestimmte oder gedachte Verwendung der Zeichen angeht: So lassen sich beispielsweise Bitfolgen, die Zahlen darstellen sollen, den Zeichenspielen der Textverarbeitung unterwerfen und anders herum. Das kann manchmal sehr praktisch sein und insgesamt wird die Universalität des Binären (der Zweiwertigkeit) wohl von den meisten als Segen begriffen.

Außensicht

Andererseits haben Bitfolgen eben auch eine Bedeutung, die *außerhalb des Systems* liegt, in dem mit ihnen gespielt wird, und es ist diese Bedeutung, die Digitalisierung erst nützlich macht. Ein Name beispielsweise ist nicht nur ein Text, sondern auch der Name von etwas. Sollte zu der Zeichenfolge, die den Namen repräsentiert, eine Zahl addiert werden (wodurch dem Namen durch ebenseine Verwendung in einer Addition zumindest vorübergehend die Bedeutung einer Zahl beigemessen würde), dann ist dies in den meisten Fällen wohl sinnlos und somit ein Fehler. Dieser Fehler fällt jedoch innerhalb des Systems nicht weiter auf — erst wenn dem Ergebnis der Addition außerhalb des Systems eine Bedeutung beigemessen werden soll, kann man feststellen, dass hier „etwas nicht stimmt". Je nachdem, wieviel es kostet, den Fehler festzustellen und zu korrigieren (oder, im schlimmeren Fall, wie viel der Fehler kostet, wenn er nicht bemerkt und korrigiert wird), ist die Universalität der Zweiwertigkeit, an der es liegt, dass solche Fehler überhaupt vorkommen können, auch ein Fluch.

Typisierung

Um solche Fehler zu verhindern, müsste man sicherstellen, dass allen Zeichenfolgen auch von außerhalb, also nicht nur durch ihre Verwendung innerhalb des Systems, eine Bedeutung explizit zugeordnet ist. Diese externe Bedeutung könnte dann intern verwendet werden, um bei jeder Anwendung eines Zeichenspiels (und der damit erfolgten internen Bedeutungszuordnung) zu prüfen, ob dies zur externen Bedeutung passt — wenn nicht, ist ein Fehler aufgetreten. Eine Möglichkeit, die externe Bedeutung zu codieren, wäre natürlich, für Zahlen, Texte etc. jeweils andere Zeichen zu verwenden (wie wir Menschen es

---

[34] So erlaubt beispielsweise ein so alter Prozessor wie der 6502 mit einer Taktfrequenz von nur 1 MHz, die Helligkeit des Kathodenstrahls eines als Monitor angeschlossenen PAL-Fernsehers während eines einzelnen Zeilendurchlaufs mehrfach zu ändern.

ja schließlich auch weitgehend tun) — damit wären allerdings sämtliche Vorteile einer universellen Zweiwertigkeit dahin. Eine andere wäre, jeder Zeichenfolge eine zweite beizuordnen, die ihren **Typ** und damit ihre mögliche Verwendung angibt; vor der Anwendung eines Zeichenspiels müsste dann jeweils geprüft werden, ob der Typ zum Spiel passt. So könnte man beispielsweise verhindern, dass eine Zahl zu einem Text addiert wird. Dieser Ansatz verlangt allerdings ein bestimmtes Maß an *Redundanz*, da die Bedeutung einer Zeichenfolge auch intern zweimal festgelegt wird — einmal durch ihren Typ und einmal durch ihre Verwendung in einem Zeichenspiel. Werden alle Zeichenspiele immer korrekt angewendet, ist diese Redundanz überflüssig, weswegen der damit verbundene Aufwand verschwendet erscheint. Auf der anderen Seite werden sich Fehler in einem System mit einer solchen Universalität nicht ohne Aufwand finden lassen — ohne Redundanz kann ein Fehler nicht ohne Hilfe von außen festgestellt werden. Insofern ist nicht die Notwendigkeit eines Aufwands das Problem, sondern dass der Aufwand gescheut wird.

Sicherheit

Wie wir noch sehen werden, sind auch viele *Sicherheitsprobleme*, unter denen heutige Computersysteme leiden, Folge der Universalität von Bitfolgen. So können auch heute immer noch nicht Daten und Programme in einem Computer zuverlässig voneinander getrennt werden, wodurch sich Schadprogramme als Daten getarnt leicht verbreiten können. Dies zu verhindern wäre mit erheblichem technischem Aufwand verbunden und würde verlangen, dass wir alle unsere Computersysteme gegen neue austauschen. Dafür ist der Leidensdruck jedoch noch nicht groß genug.

Kurseinheit 2: Computer

On two occasions I have been asked, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage (* 26. Dezember 1791, † 18. Oktober 1871)

Garbage in, garbage out.

unbekannt

Es muss in der 11. oder 12. Klasse gewesen sein, als mein Deutschlehrer mich fragte, ob ich denn wüsste, dass ein Computer nur aus Einsen und Nullen bestünde. Er war Deutschlehrer und deswegen musste ich davon ausgehen, dass er nicht ungenau formuliert hatte, sondern genau das meinte: Ein Computer besteht aus Einsen und Nullen. Aber selbst wenn er gemeint hätte, dass ein Computer nur Einsen und Nullen verarbeiten kann (was für damalige wie für heutige Computer im Wesentlichen zutrifft), so hatte die Frage doch etwas Unterstellendes: Auch mir müsse klar sein, dass ein Computer ein äußerst beschränktes Gerät sei. Dies aber offenbart ein grundlegendes Missverständnis: Nur Einsen und Nullen verarbeiten zu können ist keine wesentliche Einschränkung, so dass aus der Annahme, dass diese Eigenschaft einem Computer zugeschrieben werden kann, nichts Wesentliches folgt.

In der ersten Kurseinheit haben Sie gesehen, was man mit nur zwei Zeichen, 1 und 0, und darauf definierten Zeichenspielen alles machen kann. [35] In dieser Kurseinheit geht es darum, wie diese beiden Zeichen und die Zeichenspiele in Hardware umgesetzt werden (Kapitel 11) und die Hardware durch die Paarung mit Betriebssystemen einer breiten Verwendung zugeführt wird (Kapitel 12). Betrachtungen von Internet (Kapitel 13), Cloud (Kapitel 14) und der Sicherheit von Computern (Kapitel 15) schließen diese Kurseinheit ab.

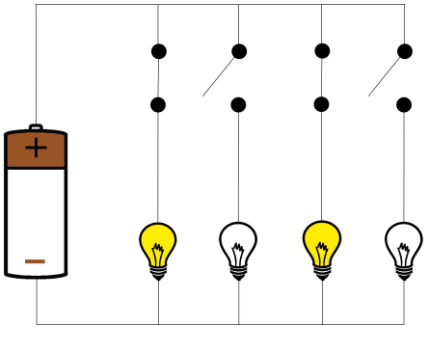
[35] Ab sofort verwende ich keine Anführungszeichen mehr, um Zeichen zu kennzeichnen.

11 Hardware

Bislang haben wir Zeichen ja auf Papier notiert, aber für eine technische Umsetzung ist das unpraktisch. Wenn wir Bleistift und Papier durch ein technisches Gerät, eine **Hardware**, ersetzen wollen, müssen wir zunächst einmal klären, wie wir die Zeichen verorten wollen und wie sie geschrieben und gelesen werden können. Dazu stellen wir uns vor, dass wir die Stellen der Hardware, an denen die Zeichen 1 und 0 stehen können (und die den Karos oder Kästchen auf dem Papier aus Kurseinheit 1 entsprechen), mit Messpunkten verbinden. An diesen Messpunkten sollen zwei verschiedene Zustände ablesbar sein, die wir im folgenden auch *An* (für 1) und *Aus* (für 0) nennen (vgl. Abschnitt 1.2 in Kurseinheit 1).

Der einfachen Anschauung wegen stellen wir uns die Hardware als ein elektromechanisches Gerät vor, das aus einer Stromquelle (Batterie), Leitungen (Drähten), Lampen (Glühbirnen) und Schaltern besteht. Von den Schaltern unterscheiden wir zwei Arten: solche, die von dem Bediener des Geräts zum Zweck der Eingabe betätigt werden (Lichtschalter), und solche, die durch einen Strom geschaltet werden (*Relais*). Die Lampen sind unsere Ausgaben, die uns anzeigen, ob an einer Stelle 1 (*An*) oder 0 (*Aus*) vorliegt. Lampen sollen dabei mit ihrem einen Anschluss immer mit dem Minuspol der Stromquelle verbunden sein; der andere Anschluss sei über einen oder mehrere Schalter mit dem Pluspol der Stromquelle verbunden. Ist der oder sind die Schalter offen, liegt auch an diesem Anschluss der Minuspol an, nämlich über den Glühfaden der Lampe (der elektrisch einen Widerstand darstellt); ist er oder sind sie geschlossen, leuchtet die Lampe. Die Lampe zeigt somit immer den Zustand an diesem Anschluss an.

Die Ein- und Ausgabe einer vierstelligen Dualzahl lässt sich demnach wie folgt in Hardware umsetzen:



Hierbei zeigen die Lampen den Zustand der Schalter an (gelb bzw. grau bedeutet *An* oder 1, weiß bedeutet *Aus* oder 0): Ist eine Lampe aus, ist der mit ihr verbundene Schalter offen, ist sie an, ist er geschlossen. Ein geschlossener Schalter gibt demnach eine 1 ein, ein geöffneter eine 0, und die obige Kombination der Schalterstellungen bedeutet, als Ziffern einer Zahl interpretiert, die *Dualzahl* 1010 (6 im Dezimalsystem; s. Kapitel 2). Man sieht hier unmittelbar, was der Vorteil einer solchen technischen Realisierung gegenüber dem Schreiben auf einem Blatt Papier ist: Man kann durch Betätigen der Schalter an denselben vier Stellen (Messpunkten) nacheinander jede beliebige vierstellige Dualzahl einstellen, ohne zu einem

Radierer greifen zu müssen. In einem Abbild einer Schaltung wie dem obigen wird dabei natürlich immer nur ein möglicher Zustand der Schaltung angezeigt.

Spannung statt Strom

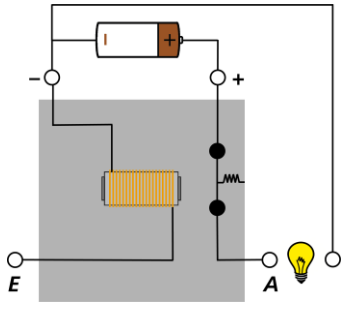
Tatsächlich versucht man, Stromfluss in Computern so weit wie möglich zu vermeiden (Energieverbrauch!) und arbeitet lieber mit Spannungen als mit Strömen. Der digitale Zustand wird entsprechend durch eine hohe Spannung (engl. high, mit H abgekürzt) für *An* oder durch eine niedrige Spannung (L für low) für *Aus* angegeben. Wir bleiben hier jedoch bei dem anschaulichen Bild von Schaltern, Lampen und Strömen.

11.1 Gatter

Mithilfe obiger technischer Bausteine lassen sich nun auf einfache Weise logische Operationen realisieren. Nehmen wir an, wir wollen den *Halbaddierer* aus Abschnitt 3.2.3 bauen. Wie wir gesehen haben, lässt sich dieser durch die beiden Gleichungen

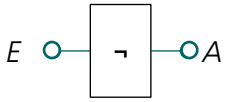
$$S = (S_1 \wedge \neg S_2) \vee (\neg S_1 \wedge S_2)$$
$$\ddot{U} = S_1 \wedge S_2$$

darstellen, wobei hier S_1 und S_2 für Eingabestellen, an denen jeweils wahlweise *An* oder *Aus* anliegen kann, und S und $\ddot{U}$ für Ausgabestellen, an denen dann *An* oder *Aus* gemessen werden kann, stehen. Auch wenn sich ein solcher Halbaddierer geschickter konstruieren lässt, wollen wir aus Gründen der Systematik die obigen Gleichungen genauso in Hardware umsetzen und konstruieren zu diesem Zweck zunächst Bausteine, **Gatter** genannt, für die logischen Operatoren ∧, ∨ und ¬. Das einfachste ist das Gatter für ¬, die *Negation*:



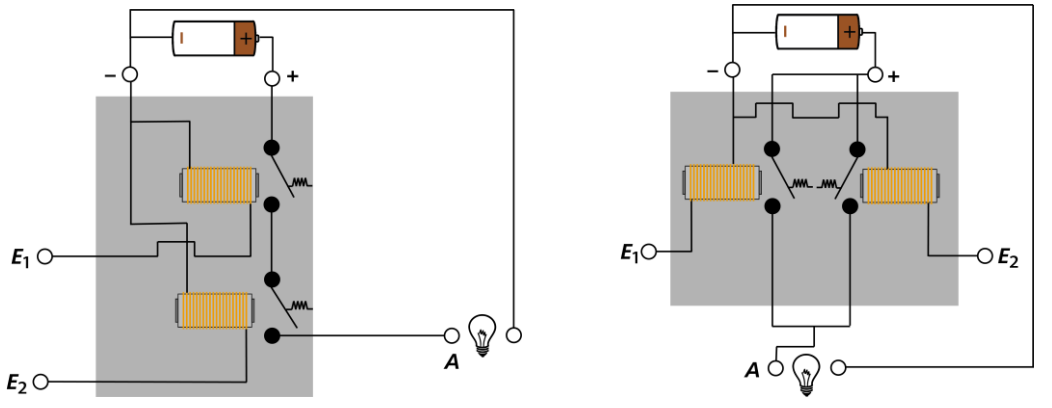
Hier öffnet sich der Schalter des *Relais* (durch die Magnetkraft, die in der Spule erzeugt wird), wenn der mit *E* bezeichneten Eingang des Gatters (das durch die graue Fläche vom Rest der Schaltung abgegrenzt sei) mit dem Pluspol verbunden ist — die Verbindung des mit *A* bezeichneten Ausgangs des Gatters mit dem Pluspol wird damit unterbrochen, so dass von dort kein Strom nach *A* fließen kann, die Lampe also ausgeht. Diese einfache Hardware, die eine Eingabe von 1 oder 0 negiert (in der Schaltungstechnik sagt man auch *invertiert*), wird **Nicht-Gatter** genannt; sie soll im folgenden als elementarer Baustein betrachtet und in *Blockschaltbildern* durch das *Schaltsymbol*

WIKIPEDIA



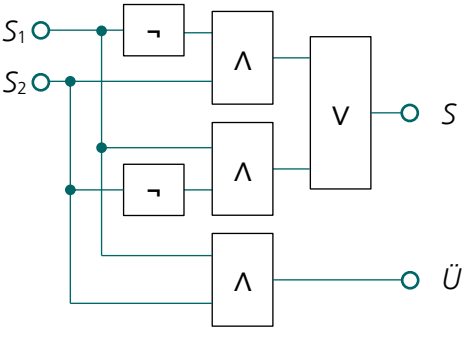
dargestellt werden (wobei in der Schaltungstechnik gern andere Zeichen als das logische ¬ verwendet werden). Man spricht dann auch von **Schaltalgebra**, einer zur *Aussagenlogik* analogen Ausprägung der *booleschen Algebra* (vgl. Abschnitt 3.2.3 in Kurseinheit 1). Man beachte, dass bei Schaltsymbolen nur Ein- und Ausgang ausgewiesen sind; Plus- und Minusanschluss, die in der technischen Umsetzung ebenfalls benötigt werden, sind für das Verständnis eines aus Schaltsymbolen zusammengesetzten Blockschaltbildes nicht notwendig und werden deshalb weggelassen.

Mit Relais lassen sich auch leicht Gatter für die logischen Operatoren ∧ (*Konjunktion*) und ∨ (*Disjunktion*), die sog. **Und-** und **Oder-Gatter**, bauen:



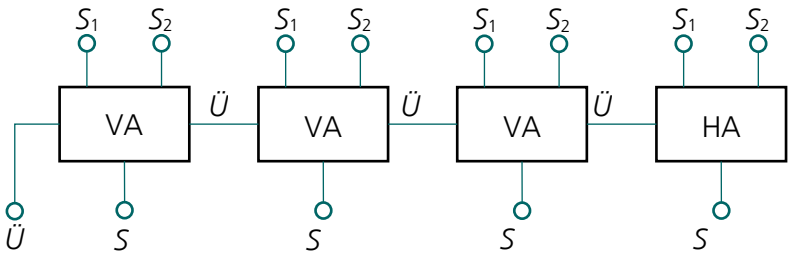
Das Und-Gatter (links) wird durch eine Serienschaltung von zwei Relais, das Oder-Gatter (rechts) durch eine Parallelschaltung umgesetzt. Genau wie beim Nicht-Gatter betrachten wir fortan beide als elementare Bauteile und notieren

Damit lässt sich dann ein Halbaddierer zusammenstellen, der genau der obigen Gleichung entspricht und diese somit in Hardware umsetzt:



Auf ähnliche Weise lässt sich dann auch ein *Volladdierer* zusammenstellen. Beide können wieder als elementare Bausteine aufgefasst werden und werden in Blockschaltbildern durch

entsprechende Schaltsymbole repräsentiert. Das Blockschaltbild für eine Schaltung mit einem Halbaddierer (HA) und drei Volladdierern (VA), die zwei vierstellige Dualzahlen addieren kann, sieht etwa so aus:



Hierbei signalisiert der letzte, ganz linke *Übertrag* (*Ü*) einen *Überlauf* (s. Abschnitt 2.4.1). Tatsächlich sehen alle als Hardware realisierten Addierer eine feste Stellenzahl vor, so dass immer ein Überlauf auftreten kann; für Zahlen, deren Darstellung mehr Stellen benötigen würde, müssen entweder mehrere Addierer miteinander verbunden oder mit demselben Addierer mehrere Additionen zeitlich nacheinander durchgeführt werden, wobei bei den nachfolgenden Additionen ein Überlauf aus der vorherigen Addition als Übertrag zu berücksichtigen ist.[36] Mit der zeitlich nacheinander ausgeführten Addition auf demselben Addierer können Zahlen beliebiger Stellenzahl addiert werden; sie erfolgt in der Regel programmgesteuert (s. Abschnitt 11.5).

Modularität von Hardware

Am Beispiel des obigen vierstelligen Addierers erkennt man sehr schön, wie sich durch das wiederholte Zusammensetzen von Komponenten standardisierte Bauteile entwerfen lassen, die komplexe Operationen, oder *Zeichenspiele*, durchführen können. Man kann diese Bauteile verwenden und auch zu noch komplexeren zusammensetzen, ohne ihren inneren Aufbau kennen zu müssen: Es reicht zu wissen, welches Zeichenspiel sie umsetzen. Diese Eigenschaft von Hardware, auch *Modularität* genannt, macht sie so außerordentlich erfolgreich.

Vereinheitlichung der Gatter

In der Praxis verwendet man übrigens schon längst keine *Relais* mehr, um Gatter und damit Computer zu bauen, sondern *Transistoren*, die dazu noch nicht einmal mehr als eigenständige Bauteile ausgeführt, sondern auf *Halbleiterplatten* aufgedampft werden. Um den Herstellungsprozess zu vereinfachen, werden auch keine verschiedenen Gatter aufgedampft, sondern nur Gatter einer Sorte, nämlich sog. NAND- oder NOR-Gatter. Dabei steht NAND für ein negiertes Und und NOR für ein negiertes Oder; mit nur einem von beiden lassen sich alle anderen logischen Operatoren (und damit alle Gattertypen) ersetzen. So gilt nämlich beispielsweise

$$\begin{aligned} \neg A &\Leftrightarrow \neg(A \wedge A) \\ A \wedge B &\Leftrightarrow \neg(\neg(A \wedge B) \wedge \neg(A \wedge B)) \\ A \vee B &\Leftrightarrow \neg(\neg(A \wedge A) \wedge \neg(B \wedge B)) \end{aligned}$$

[36] Deswegen verwenden mehrstellige Addierer, anders als oben dargestellt, auch an der ersten Stelle einen Voll- und keinen Halbaddierer.

Neben den Gattern für die logischen Operatoren gibt es auch Gatter zum Speichern von Zuständen, mit denen auch die Multiplikation mit und die Division durch Zwei realisiert werden können. Diese zu bauen verlangt jedoch zunächst ein Betrachtung der *zeitlichen Dimension von Hardware*.

11.2 Zeit, Synchronisation und Takt

Zeit

Elektrische Signale wie der Wechsel von *An* zu *Aus* oder umgekehrt breiten sich nicht augenblicklich aus — sie haben sog. *Laufzeiten*. Bis ein solcher Wechsel von der Quelle bis zum Ziel (von einem Ende einer Leitung bis zum anderen) gelangt ist, vergeht also eine gewisse Zeit. Hinzu kommt die Zeit, die elektrische Schalter (wie Relais; aber auch Transistoren) zum Schalten benötigen. Dies kann schon bei obigem vierstelligen Addierer zu kurzzeitig falschen Ergebnissen führen, wie die folgende Betrachtung zeigt.

Angenommen, an allen Eingängen des obigen Addierers liegt zunächst 0 an. An allen Ausgängen liegt dann, nachdem die Addition durchgeführt wurde, ebenfalls 0 an. Nun wechseln alle Eingänge für den ersten Summanden gleichzeitig auf 1, die für den zweiten Summanden bleiben aber auf 0 (entsprechend 1111 + 0000). Die Überträge bleiben dabei zunächst ebenso auf 0. Wenn wir annehmen, dass der Strom durch alle Addierer genau gleich lang, also eine bestimmte Zeiteinheit braucht, dann liegt an den Ausgängen um diese Zeiteinheit später das Ergebnis 1111 an. Die Überträge bleiben auch nach dieser Zeit weiter auf 0. So weit so gut.

Synchronisation

Wenn aber nun nur die niedrigststellige Ziffer (ganz rechts) des zweiten Summanden auf 1 wechselt (entsprechend 1111 + 0001), dann steht eine Zeiteinheit später an der Ausgabe die Summe 1110, was aber nicht der korrekten Summe (0000 + Überlauf, entsprechend 10000) entspricht. Das liegt daran, dass der Übertrag des ersten, ganzen rechten Addierers in seinen Nachbarn zu diesem Zeitpunkt gerade erst von 0 auf 1 wechselt, er bei der Addition der zweiten Stelle also noch nicht berücksichtigt werden konnte. Dies geschieht erst, wenn der Signalwechsel durch den Addierer der zweiten Stelle durchgelaufen ist, also genau eine Zeiteinheit später. Dann wechselt die zweite Stelle der Summe ebenfalls auf 0. Leider ist das Ergebnis 1100 immer noch nicht korrekt, da nun der Übertrag von der zweiten auf die dritte Stelle zu 1 wechselt, was zu berücksichtigen wiederum eine Zeiteinheit benötigt usw., bis das Ergebnis (inkl. Überlauf) schließlich nach fünf Zeiteinheiten korrekt anliegt. Man beachte, dass es umso länger dauert, je mehr Stellen „gleichzeitig“ addiert werden.

Das obige Beispiel ist so konstruiert, dass man mit wenig Aufwand erkennen kann, was das Problem ist. Es lassen sich andere Beispiele konstruieren, bei denen das Problem andere Formen annimmt. Wenn man z. B. 1111 + 0101 addiert, dann wandern zunächst die Überträge von der ersten und der dritten Stelle (wie immer von rechts gezählt) gleichzeitig um eine Stelle nach links; eine Zeiteinheit später aber läuft der Übertrag von der zweiten auf die dritte Stelle auf, weswegen sich hier auch die Summe noch einmal ändert. Allgemein

können die Signallaufzeiten dazu führen, dass sich Zustände eine Zeitlang scheinbar chaotisch ändern; allerdings müssen sie sich zu einem bestimmten Zeitpunkt stabilisiert haben, da sonst nie ein Ergebnis ablesbar wäre und somit ein Fehler in der Schaltung vorläge.

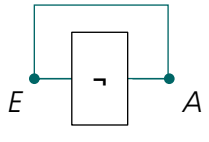
Takt

Hardware braucht also (genau wie ein Mensch) ihre Zeit, um Zeichenspiele durchzuführen. Deswegen teilt sie die Zeit in eine Folge von Zeitpunkten auf, zwischen denen eine Zeitspanne liegt, in der die Zustände als ungültig anzusehen sind. Jede Schaltung hat damit Zeit, innerhalb der Zeitspanne in einen stabilen Zustand zu gelangen — erst dann wird das Ergebnis erwartet. Die Folge der Zeitpunkte wird durch einen **Takt** vorgegeben, der im Wesentlichen ein Signal ist, das mit einer bestimmten Frequenz von 0 auf 1 und wieder zurück wechselt. Man spricht hier von der **Taktfrequenz** einer Schaltung; sie wird in Hertz (Wechsel pro Sekunde) gemessen. Damit die Zeitpunkte auch wirklich Punkte (und keine Zeitspannen oder Phasen) sind, werden sie auf den Wechsel des Taktsignals (von 0 zu 1 oder von 1 zu 0; eine sog. Taktflanke) gelegt.

Die Idee des Takts ist, von der kontinuierlichen Dimension der Zeit, der die Physik aller Gatter unterliegt, zugunsten einer diskreten *Folge von Zuständen* zu abstrahieren, in der es nur ein *Vorher* und ein *Nachher* und keine Dauer mehr gibt. In diesem abstrakten Modell überführen alle Gatter ihre Eingänge in ihre Ausgänge in einem Schritt. Im Fall unseres vierstelligen Addierers wäre die Taktfrequenz also so zu wählen, dass die Zeit auf jeden Fall ausreicht, um in der Zeitpanne, die zwischen zwei Taktschlägen liegt (dem Kehrwert der Taktfrequenz) eine stabile Ausgabe zu erhalten. Allerdings reicht das nicht aus, wenn wir den Überlauf unseres Addierers mit dem Eingang eines weiteren vierstelligen Addierers verbinden wollen, um so einen achtstelligen Addierer zu bauen: Hier müsste entweder die Taktzeit verdoppelt (also die Taktfrequenz halbiert) oder zwei Takte abgewartet werden, bis ein stabiles Ergebnis vorliegt.

11.2.1 Taktung von Gattern

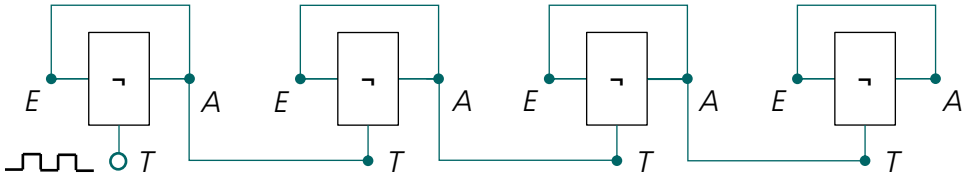
Die Einführung eines Taktes allein reicht nicht aus, um beliebige Zeichenspiele in Hardware umzusetzen. Dies zeigt das einfache Beispiel von einem Nicht-Gatter, dessen Ausgang in seinen Eingang zurückgekoppelt wird:



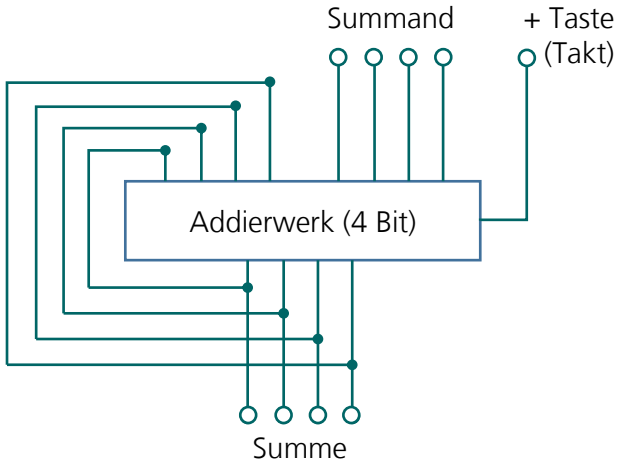
Mit einem Relais (das beim Schalten klackt) ausgeführt würde diese Schaltung einen prima Summer abgeben, der zu den durch einen Takt vorgegebenen Zeitpunkten zufällige Zustände an seinem Ein- und Ausgang hätte. Um das zu verhindern, verlangen Schaltungen mit Rückkopplungsschleifen (auch *Schaltwerke* genannt, im Gegensatz zu *Schaltnetzen*, die keine Rückkopplungen aufweisen), dass alle Gatter einen zusätzlichen Takteingang haben und sicherstellen, dass die Ausgabe des Gatters erst mit dem nächsten Taktschlag auf seinen Ausgang geschaltet wird und erst zu diesem Zeitpunkt die Eingabe von seinem Eingang

entgegengenommen wird. Auf diese Weise oszillieren Ein- und Ausgang der obigen Schaltung zwar immer noch zwischen *An* und *Aus*, aber jetzt genau im Takt, so dass der Zustand zu jedem Taktschlag durch den vorherigen eindeutig bestimmt (und nicht mehr zufällig) ist.

Mithilfe getakteter Gatter lassen sich Zeichenspiele wie beispielsweise das Zählen aus Abschnitt 2.7 umsetzen. Hierzu reicht es bereits, mehrere wie oben rückgekoppelte, jedoch zusätzlich getaktete Nicht-Gatter hintereinanderzuschalten, und zwar so, dass der Ausgang des ersten als Takt des zweiten dient usw.:



Diese Schaltung zählt dann die Takte, die am ersten Gatter an dessen Anschluss *T* anliegen (wobei hier die Taktschläge die Rolle der Zählimpulse, also der zu zählenden Ereignisse, spielen). Eine alternative Schaltung, die sogar in beliebigen ganzzahligen Schritten zählen kann, nimmt ein Addierwerk wie das obige, koppelt die Ausgänge (Summe) an die Eingänge, die den einen der zwei Summanden darstellen, zurück und belegt die Eingänge, die den anderen Summanden darstellen, mit der ganzen Zahl, um die bei jedem Takt (oder Zählimpuls) erhöht werden soll.



Eine solche Schaltung realisiert übrigens eine einfache *Registrierkasse*, die bei jedem Taktschlag (ausgelöst beispielsweise über eine Addiere-Taste) die als nicht rückgekoppelter Summand anliegende (über Schalter eingegebene) Zahl zur Gesamtsumme hinzuzählt.

11.2.2 Sequentielle Logik und Automaten

Mit Schaltungen mit Rückkopplungen und der Taktung von Gattern verliert man den unmittelbaren Bezug zur *Logik* (Abschnitt 3.2) und der *booleschen Algebra* (Abschnitt 3.2.3). Wenn man nämlich beispielsweise ein (ungetaktetes) Nicht-Gatter mit Eingang ***E*** und Ausgang ***A*** durch die Gleichung ***A*** = ¬***E*** beschreibt, dann ergibt die Rückkopplung des Ausgangs

in den Eingang, beschrieben durch $A = E$, einen logischen Widerspruch: $E = \neg E$. Dieser Widerspruch entspricht dem undefinierten (oder zufälligen) Zustand einer entsprechenden Schaltung vor Einführung eines Taktes.

sequentielle Logik

Mit der Einführung der zeitlichen Dimension in Form des Taktes lässt sich dieser Widerspruch jedoch auflösen: Danach hat nämlich jede logische Variable (wie A oder E) zu jedem Zeitpunkt (oder Taktschlag) t einen eigenen Wert und die obige Rückkopplung würde durch die Gleichungen $A_{t+1} = \neg E_t$ und $E_t = A_t$ beschrieben. Die Einsetzung ergibt dann $E_{t+1} = \neg E_t$, was keinen Widerspruch darstellt, sondern den zu beobachtenden Wechsel des Zustands am Eingang des Gatters mit jedem nächsten Taktschlag. Da sie eine Sequenz von Zuständen beschreibt, wird diese Form der Logik auch **sequentielle Logik** genannt. Sie wird von der **kombinatorischen Logik** unterschieden, die keine zeitliche Dimension kennt (und in der Rückkopplungen nicht gestattet sind).

Automaten

Die Einführung getakteter Gatter und einer sequentiellen Logik bedeutet, dass der aktuelle Zustand einer Schaltung nicht mehr vom Eingang allein, sondern auch von ihrem vorherigen Zustand abhängt. In der Informatik spricht man dabei von Zustandsautomaten, genauer von **deterministischen endlichen (Zustands-)Automaten** (engl. deterministic finite state machines). Sie sind endlich, weil sie nur eine endliche Anzahl verschiedener Zustände haben, und deterministisch, weil der Folgezustand ausschließlich vom aktuellen Zustand und den (aktuellen) Eingaben abhängt. Dagegen können **nichtdeterministische Automaten** ihren Zustand auch ohne Eingaben ändern, wobei der Folgezustand nicht vorhersehbar (eben nichtdeterministisch, also zufällig) ist. Wenn man mehrere kleine Automaten zu einem größeren zusammenfügt (indem man mehrere Gatter mit Rückkopplungsschleifen zu einem größeren Bauteil zusammensetzt), dann erhält man wiederum einen Automaten, dessen Zustand das Produkt der Zustände seiner Bestandteile ist.

WIKIPEDIA

Gedächtnis

Das Bemerkenswerte an Automaten (kleinen wie großen) ist, dass sie ein *Gedächtnis* haben: Sie können auf gleiche Eingaben unterschiedlich reagieren, wobei die genaue Reaktion davon abhängt, was vorher war oder, richtiger, was sich die Automaten davon gemerkt haben. Der aktuelle Zustand eines Automaten ist dabei die Folge eines Startzustands sowie aller vorherigen Eingaben und stellt damit eine Art Zusammenfassung dieser Eingaben dar. Was der Automat sich genau merken kann wird durch den Aufbau des Automaten festgelegt. So gibt es beispielsweise Automaten, die Ereignisse zählen (s. o.; sie merken sich den aktuellen Zählerstand) oder auch Automaten, die sich einfach nur Eingaben merken (Speicher).

Jeder Computer kann als ein deterministischer endlicher Automat begriffen werden. Allerdings ist die Menge der möglichen Zustände eines Computers so groß, dass diese Betrachtungsweise Computer nicht verniedlicht: Eine Schaltung mit 320 wie oben in Reihe geschalteten, rückgekoppelten und getakteten Nicht-Gattern (entsprechend einem einfachen Computer mit 40 Byte Speicher) hat bereits mehr Zustände, als es Atome im Universum gibt ($2^{320} > 10^{85}$). Dies ist jedoch kein Verdienst des Computers, sondern schlicht den Gesetzen der *Kombinatorik* geschuldet.

11.2.3 Grenzen der Taktfrequenz

Die Grenzen der Taktfrequenz einer Hardware werden nicht nur durch die Schaltzeiten der einzelnen Gatter bestimmt, sondern auch durch die Laufzeiten der Signale, einschließlich des Taktsignals selbst. Dies kann sich nämlich (abhängig vom Leitermaterial) maximal mit Lichtgeschwindigkeit ausbreiten, so dass es bei einer Taktfrequenz von beispielsweise 4 GHz zwischen zwei Taktschlägen höchstens 7,5 cm weit kommen kann (die tatsächlichen Werte liegen deutlich darunter, bei ca. 5 cm). Dies bedeutet aber, dass es an verschiedenen Stellen einer größeren Schaltung trotz Taktes keine Gleichzeitigkeit gibt, sofern nicht die Leitung, die den Takt überträgt, zu allen Stellen gleich lang ist, was allerdings sehr aufwendige Wegführungen verlangen würde. Erschwerend kommt hinzu, dass die physikalischen Grenzen der Miniaturisierung von Hardware inzwischen erreicht sind, so dass sich eine Steigerung der Leistungsfähigkeit nur noch über eine Vergrößerung der Fläche, auf der die Schaltungen untergebracht sind, erreichen lässt. Wegen der dadurch größer werdenden Distanzen und entsprechend länger werdenden Signallaufzeiten entstehen verschiedene „Zeitzonen" in einer Schaltung: Während an manchen Stellen schon der nächste Takt geschlagen hat, ist an anderen der vorherige noch gar nicht angekommen. Man ist daher dazu übergegangen, die Hardware zu parallelisieren, also in mehrere entkoppelte Einheiten aufzuteilen, die weitgehend unabhängig voneinander arbeiten können. Wie Sie sehen, sind digitale Schaltungen höchst *analoge*[37] Angelegenheiten und es ist einiges an Tüftelei notwendig, um Digitalität zu simulieren.

Eine weitere Grenze der Taktfrequenz liegt bei der heute verwendeten Halbleitertechnologie (*CMOS*, für Complementary Metal Oxyde Semiconductor) übrigens darin, dass der Stromverbrauch wesentlich am *Zustandswechsel* (von 0 zu 1 oder umgekehrt) hängt: Zum einen müssen für einen Zustandswechsel elektrische Kapazitäten ge- bzw. entladen werden, zum anderen fließt bei jedem Zustandswechsel für einen winzigen Augenblick ein Kurzschlussstrom. Da Zustandswechsel an den Takt gebunden sind, erhöht sich mit der Taktfrequenz auch der Stromfluss. Die Höhe der Taktfrequenz wird also auch durch die Menge des verfügbaren Stroms begrenzt, was insbesondere bei batteriebetriebenen Geräten zu bedenken ist. Zudem wird, da der Strom vollständig in Wärme ungesetzt wird, die Schaltung durch höhere Taktfrequenzen heißer (die Leistungsdichten heutiger hochintegrierter Schaltungen liegen teilweise deutlich höher als bei Herdplatten), was zu Alterungsprozessen bis hin zu Auflösungserscheinungen führt. Höhere Taktfrequenz bedeutet also nicht nur logisch das gleiche Ergebnis in weniger Zeit, sondern auch physikalisch höhere Leistung (wie wenn ein Motor schneller dreht) und damit höheren Energieverbrauch. Moderne Computer senken daher ihre Taktfrequenz(en) ab, wenn gerade nur wenig Leistung benötigt wird.

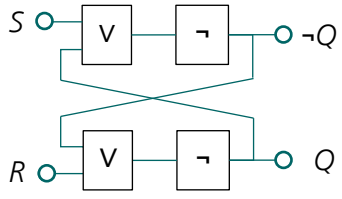
---

[37] „Analog" bezeichnet hier die stufenlose, kontinuierliche Veränderlichkeit physikalischer Größen, die im Gegensatz zur stufigen, diskreten Veränderung im Digitalen steht.

11.3 Flüchtiger Speicher

Flipflops

Die obige Idealisierung von getakteten Gattern verlangt für ihre technische Umsetzung, dass man Zustände (*An* oder *Aus*, 1 oder 0) an den Eingängen eine Zeit lang halten, oder speichern, kann, so dass nach Ablauf der Zeit, beim nächsten Taktschlag, am Ausgang ein stabiler Zustand anliegt. Eine solcher Speicher wird durch ein sog. **Flipflop**[38] realisiert, das man in seiner einfachsten Variante durch zwei über Nicht-Gatter rückgekoppelte Oder-Gatter umsetzen kann:



Hierbei wird durch ein kurzeitiges Anlegen von 1 an *S* (für setzen) an *Q* solange eine 1 gehalten, bis eine 1 an *R* (für rücksetzen) angelegt wird, woraufhin *Q* zu 0 wechselt. Sind *S* und *R* beide 0, bleibt *Q* in seinem aktuellen Zustand; sind beide 1, fängt die Schaltung, die nicht getaktet ist, an zu oszillieren. Ein solches Flipflop erfüllt in einem getakteten Baustein seinen Zweck, wenn es die 1 an *S* oder *R* nur im Augenblick des Taktschlages erhält (und beide Eingange ansonsten auf 0 liegen). Auf die technische Umsetzung dieser Taktung gehe ich hier nicht weiter ein, will aber nicht unerwähnt lassen, dass solche Flipflops aufgrund ihrer Funktion, Eingänge zu „verriegeln", auch *Latches* genannt werden.

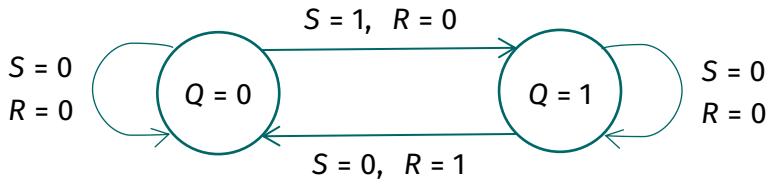
Flipflops als Automaten

Ein getaktetes Flipflop ist, genau wie ein getaktetes rückgekoppeltes Nicht-Gatter, ein *deterministischer endlicher Automat*. Es lässt sich durch die folgende **Zustandsübergangstabelle** beschreiben:

Q_t	S_t	R_t	Q_{t+1}
0	0	0	0
0	0	1	0
0	1	0	1
1	0	0	1
1	0	1	0
1	1	0	1

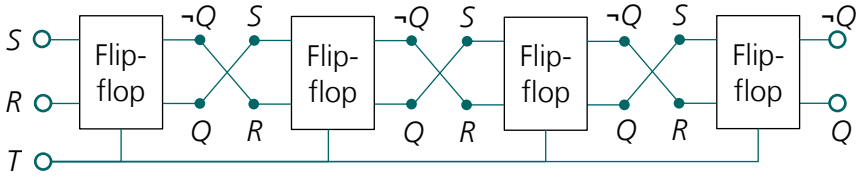
In der Informatik werden endliche Automaten auch gern als **Zustandsübergangsdiagramme** dargestellt, in denen Zustände durch Kreise und *Zustandswechsel* durch Pfeile repräsentiert werden, wobei letztere mit den die Wechsel auslösenden Ereignissen beschriftet sind. Das Zustandsübergangsdiagramm für ein getaktetes Flipflop sieht wie folgt aus:

[38] Zu Deutsch heißt Flipflop auch *bistabile Kippstufe*, was zwar ernsthafter klingt, aber nicht unbedingt eingängiger ist.



Hierbei wird die zeitliche Dimension nicht wie in obiger Tabelle über einen Index ***t***, sondern über die Pfeile, die zum Folgezustand zeigen, dargestellt.

Übrigens: Indem man mehrere getaktete Flipflops hintereinanderschaltet, kann man eine Bitfolge durch sie hindurchwandern lassen:



Solche Schaltungen werden u. a. dazu verwendet, um eine in den Flipflops gespeicherte Zahl durch Links- oder Rechtsverschiebung der Stellen mit Zwei zu multiplizieren bzw. durch Zwei zu teilen (s. Abschnitte 2.4.2 und 2.4.4 in Kurseinheit 1). Die Anzahl der Verschiebungen um eine Stelle ist gleich der Anzahl der Takte (bzw. Taktflanken) an *T*.

Flipflops als Speicher

Flipflops haben nicht nur als Bestandteil von Schaltungen, die anderen Zwecken dienen, sondern auch alleinstehend eine wichtige Funktion: Jedes von ihnen kann genau 1 Bit (also eine 1 oder eine 0; s. Abschnitt 2.3 in Kurseinheit 1) speichern. Genügend Flipflops vorausgesetzt kann man also beliebig viele Bits und damit alles speichern, was sich als Bitfolge darstellen lässt. Einen solchen Speicher, der allgemeinen Zwecken (wie der Programm- und Datenspeicherung) zur Verfügung steht und der nicht an bestimmte Schaltungen gebunden ist, stellt der **Hauptspeicher** eines Computers dar.

Dass ein Flipflop seinen Zustand über die Zeit halten kann, soll nicht darüber hinwegtäuschen, dass der Zustand weg ist, wenn der Strom abgeschaltet wird. Man spricht daher von einem **flüchtigen Speicher**. Flüchtige Speicher sind für bestimmte Anwendungen (wie z. B. Taschenrechner) vollkommen ausreichend; für andere hingegen nicht. Man braucht dann auch *nichtflüchtige*, oder *permanente*, *Speicher* (auch *Festspeicher* genannt; mehr dazu in Abschnitt 11.6.2).

DRAM und SRAM

Wie obige Abbildung nahelegt, braucht ein Flipflop für die Speicherung eines einzigen Bits eine ganze Menge Bauelemente. Dies macht insbesondere große Speicher teuer. Deswegen verwendet man auch einfachere Speicherformen, die im Wesentlichen aus einem Kondensator bestehen, der gezielt ge- und entladen werden und dessen Ladungszustand man auslesen kann. Aufgrund der nichtidealen Trennung des Kondensators nach dem Laden verliert dieser jedoch kontinuierlich seine Ladung, so dass er in regelmäßigen Abständen (ungefähr 15 mal pro Sekunde) aufgefrischt werden muss. Man spricht

deswegen auch von dynamischem Speicher (oder **DRAM**, für Dynamic Random Access Memory; mehr zu *Random Access Memory* in Abschnitt 11.4.1). Auf Flipflops basierende Speicher nennt man im Gegensatz dazu **statischer Speicher** oder **SRAM** (mit S für static; nicht zu verwechseln mit den in Abschnitt 11.6.2 behandelten *Festspeichern*).

11.4 Bus

Wie wir oben gesehen haben, können Flipflops in getaktete Schaltungen eingebaut werden, um Zustände zu halten oder einzufrieren (die sog. *Latches*). Wie an der gleichen Stelle bereits bemerkt, können sie auch alleinstehend verwendet werden, um Zustände zu speichern. Dabei stellt sich natürlich die Frage, wie solche Speicher in eine Schaltung eingebettet werden können, ohne dass sie fest mit bestimmten Ein- und Ausgängen anderer Bausteine verbunden werden — das wäre zumindest dann erforderlich, wenn die Speicherzellen universell eingesetzt werden, also mal dies und mal das speichern können sollen. Eine ganz ähnliche Problemstellung kennen Sie aus der Festnetztelefonie: Hier sind ja auch zwei Telefongeräte nicht permanent und ausschließlich miteinander verbunden, sondern die Verbindung wird „wahlweise" (durch Wählen einer Nummer) hergestellt. Dazu ist es in der Regel notwendig, dass zwei Telefone, das anrufende und das angerufene, auf eine gemeinsame Leitung aufgeschaltet werden. Diese Leitung wird durch die beiden Anschlüsse für die Zeitdauer der Verbindung belegt, d. h., sie kann während dieser Zeit nicht für andere Verbindungen verwendet werden.

Eine solche gemeinsame, d. h. von vielen Komponenten gemeinsame genutzte Leitung nennt man in Computern **Bus**. Entsprechend der Natur digitaler Schaltungen werden auf einem solchen Bus nur *An* oder *Aus* (bzw. 1 oder 0) übertragen. An ihm sind alle Bauteile angeschlossen, die Daten austauschen können sollen, also Einsen und Nullen produzieren oder konsumieren. Neben den Daten wird auch Steuerungsinformation auf einem Bus übertragen, u. a., ob er gerade frei ist oder wer ihn gerade belegt.

Aufgrund seiner verschiedenen Funktionen teilt man den Bus logisch mindestens in einen **Adressbus**, einen **Datenbus** und einen **Steuerbus** auf. Die verschiedenen logischen Busse können auch physikalisch getrennt (als unabhängige Leitungen) ausgeführt werden, was die Schaltungen einfacher macht, aber mehr Platz für die Leitungen benötigt. (Werden Busleitungen für mehrere Zwecke genutzt, braucht man zusätzliche Schaltungen, die die Zwecke erkennen und auseinanderhalten.)

11.4.1 Speicheradressierung

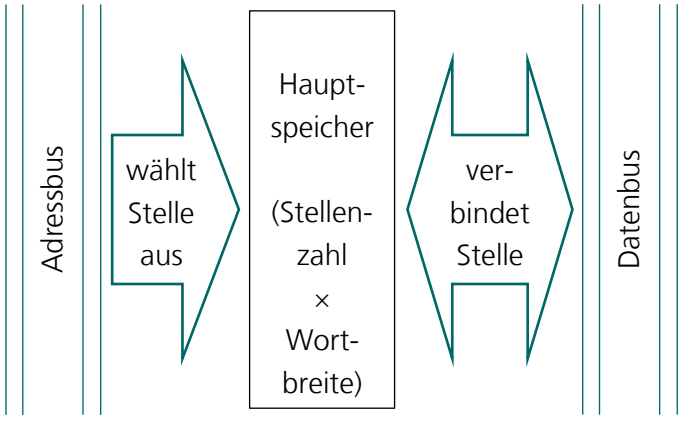
Um den *Hauptspeicher* eines Computers einer allgemeinen Verwendung zuzuführen, ist er an den Bus angeschlossen. Um über den Bus Speicherzellen mit Werten (1 oder 0) belegen und auslesen zu können, müssen sie (wie beim Telefonieren Telefonapparate) eindeutig adressiert werden können. Die Speicherzellen sind zu diesem Zweck durchnummeriert. Die Verbindung zu einer Speicherzelle wird hergestellt, indem zunächst ihre Nummer, **Speicheradresse** genannt, auf den Adressbus gegeben wird; die so adressierte Speicherzelle

wird dann mit dem Datenbus verbunden. Ob ihr Zustand dabei gelesen oder geschrieben werden soll, wird über den Steuerbus übertragen.

Der Hauptspeicher bietet über den Bus einen sog. **wahlfreien Zugriff** (engl. random access, daher **Random Access Memory** oder **RAM**), d. h., man kann auf seine einzelnen Speicherstellen in beliebiger Reihenfolge zugreifen. Dies unterscheidet ihn nicht unbedingt von anderen Speichermedien (und insbesondere nicht von einem **Nur-Lese-Speicher**, engl. **Read-Only Memory** oder **ROM**); dennoch hat sich die Bezeichnung RAM für Hauptspeicher eingebürgert.

Größe der Speicherstellen

Wie in Kurseinheit 1 dargelegt, werden die von einem Computer zu verarbeitenden Daten als Bitfolgen codiert. Es ist daher nicht sinnvoll, einzelne Bits als Speicherstellen in einem Hauptspeicher einzeln zu adressieren. Stattdessen werden immer mehrere Bits zu einer Speicherstelle zusammengefasst und jeweils mit einer *Adresse* versehen. Wie viele Bits das jeweils sind wird durch die Hardware festgelegt; neben dem üblichen *Byte* (8 Bit) sind auch andere *Wortbreiten* möglich. Man kann sich den Hauptspeicher also als eine zweidimensionale Tabelle vorstellen, deren eine Dimension durch die Wortbreite und deren andere durch die Anzahl der Speicherstellen bestimmt ist und deren Zellen jeweils eine Eins oder eine Null aufnehmen. Um an ein einzelnes Bit zu kommen, muss immer ein ganzes Wort gelesen bzw. geschrieben werden.



11.4.2 Paralleler vs. serieller Bus

Mit der Anzahl der zu adressierenden Elemente eines Computers (Speicherzellen, aber auch andere Elemente; s. u.) wächst auch die Länge der Adressen: Reichen für 10 Elemente noch 4 Bit, braucht man für 1000 schon 10 usw. (allgemein für n Elemente $\log_2 n$ Bit). Wenn man diese Bits für die Adressierung der Elemente nicht zeitlich nacheinander auf den Adressbus geben will, braucht man parallele Leitungen. Ein Bus mit parallelen Leitungen wird **paralleler Bus** genannt; einen mit nur einer Leitung (oder jedenfalls weniger Leitungen, als zur parallelen Übertragung aller Bits notwendig sind), nennt man **seriell**. Ein prominentes Beispiel für einen seriellen Bus ist der *Universal Serial Bus* (*USB*); er wird allerdings nur zur Verbindung verschiedener Geräte (Computer und anschließbare Peripherie) und nicht innerhalb eines Computers genutzt.

WIKIPEDIA

11.4.3 Busbreiten

Die Anzahl der Leitungen, die von einem Bus für die parallele Übertragung verwendet werden, nennt man die **Breite des Busses**. In der Regel bestimmt die Breite des *Adressbusses* die Größe des maximal adressierbaren Hauptspeichers: bei 32 Bit sind dies 4.294.967.296 (mehr als vier Milliarden) *Adressen*. Die Breite des *Datenbusses* bestimmt hingegen die Anzahl der gleichzeitig aus dem Speicher geholten oder in den Speicher geschriebenen Bits. Übliche Breiten sind hier 4, 8, 16, 32 und 64 bit; bei Breiten größer als 8 Bit werden also immer mehrere Bytes auf einmal aus dem Speicher geholt oder in ihn geschrieben. Ein Geschwindigkeitsvorteil ergibt sich aus höheren Datenbusbreiten nur dann, wenn man auch an längeren (zusammenhängenden) Bitfolgen interessiert ist (was die vergleichbar bescheidenen Geschwindigkeitszuwächse beim Übergang von 32- auf 64-Bit-Computer erklärt).

Die verschiedenen innerhalb eines Computers zusammenarbeitenden Komponenten (Bus, Speicher, Prozessor) können verschiedene Wortbreiten haben. So haben z. B. Prozessoren intern häufig größere Wortbreiten als die angeschlossenen Geräte. Die Wortbreiten werden dann durch entsprechende Schaltungen am Bus angepasst. So kann beispielsweise ein 64-Bit-Wort aus dem Speicher in Form von zwei aufeinanderfolgenden 32-Bit-Wörtern geholt werden. Der Vorteil höherer interner Wortbreiten ist, das bestimmte Operationen nicht nacheinander ausgeführt werden müssen. So lassen sich mit einem 64-Bit-Prozessor Zahlen mit 64 Bit in einem Schritt addieren; ein 32-Bit-Prozessor braucht dafür zwei Additionen (vgl. Abschnitt 11.1).

11.5 Prozessor

Wenn man nun die Möglichkeit hat, verschiedene Komponenten einer Hardware über ein Bussystem (bestehend aus Adress-, Daten- und Steuerbus) miteinander zu verbinden, stellt sich die Frage, welche Komponente dies tut. Zwar könnte man das einer Selbstorganisation überlassen, aber in praktisch allen heute verwendeten Computern spielt der **Prozessor** eines Computers dabei eine herausgehobene Rolle. Er wird übrigens häufig (aufgrund seiner im Vergleich zu den Anfängen der Computerei kleinen Bauform) auch **Mikroprozessor** oder (aufgrund seiner zentralen Funktion und aufgrund der Existenz anderer Prozessor genannter Bauteile im selben Computer; s. Abschnitt 11.6) **Central Processing Unit** (**CPU**) genannt.

Ein Prozessor ist ein extrem komplexes Schaltwerk, das (wie alle anderen Komponenten einer Hardware) aus *Gattern* aufgebaut ist. Er verfügt über eigene Speicherblöcke, **Register** genannt, die sowohl an den *Datenbus* als auch an die sog. **arithmetisch-logische Einheit** (engl. arithmetic logic unit, ALU) angeschlossen sind, die regelmäßig zumindest ein Addierwerk wie das aus Abschnitt 11.1 sowie Implementierungen der logischen Operatoren umfasst. Um die notwendigen Verbindungen zwischen Registern und der ALU herstellen zu können, verfügt der Prozessor über einen internen Bus.

Die Verwendung beider Busse erfolgt durch das sog. **Steuerwerk** eines Prozessors; der Einfachheit halber wird im folgenden aber nur die Verwendung des Busses, der den Prozessor mit der übrigen Hardware eines Computers verbindet, skizziert. Das Steuerwerk besitzt mindestens zwei eigene Register, das **Befehlsregister** und den **Programmzähler**. Das Befehlsregister wird mit einem *Maschinenbefehl* aus dem *Hauptspeicher* des Computers geladen, indem der Inhalt des Programmzählers auf den Adressbus gegeben und der Inhalt der adressierten Speicherzelle über den Datenbus aus dem Speicher geholt wird. Anschließend wird der Programmzähler um 1 erhöht und, abhängig vom zuvor geladenen Befehl, eine Schaltung in Gang gesetzt, die den Befehl umsetzt. Lautete der Befehl beispielsweise „inkrementiere den Wert von Register 1", dann wird Register 1 mit der ALU verbunden und dort die Funktion „addiere 1" auf dem Register ausgeführt. Lautet der Befehl dagegen „lade den Inhalt der Speicherzelle, deren *Adresse* an der Stelle des Programmzählers steht, in Register 1", dann wird zunächst wieder der Programmzähler auf den Adressbus gegeben, sodann der adressierte Speicherwert in ein spezielles Register, das **Adressregister** geladen, danach das Adressregister auf den Adressbus gegeben und dann der Inhalt der so adressierten Speicherzelle in Register 1 geladen. Diese sog. *indirekte Adressierung* einer Speicherstelle dient der *Trennung von Programm und Datenspeicher*: Anstatt den in Register 1 zu ladenden Wert direkt hinter dem Befehl im Programm zu speichern, wird er an der Adresse gespeichert, die hinter dem Befehl gespeichert ist (vgl. Kapitel 9 in Kurseinheit 1). Dies ist insbesondere dann von Bedeutung, wenn der Wert auch geändert und wieder in den Speicher zurückgeschrieben werden soll — ohne indirekte Adressierung müssten Programme in den Programmspeicher schreiben, was, sagen wir mal, verpönt ist.

Indirektion

Die **Indirektion** ist eine Art Rad der Informatik, etwas, das in der Natur nicht vorkommt, ohne das aber der Fortschritt undenkbar ist:

We can solve any problem by introducing an extra level of indirection.

David Wheeler (* 9. Februar 1927, † 13. Dezember 2004)

Dabei sieht man einem Wert im Speicher nicht an, ob er eine Adresse ist (also der indirekten Adressierung dient) oder ein anderes Datum — einmal mehr wird die Bedeutung einer Folge von Einsen und Nullen allein durch den Gebrauch festgelegt.

Während die indirekte Speicheradressierung die Trennung von Programm- und Datenspeicher erlaubt, ist sie durch den oben beschriebenen Aufbau eines Prozessors nicht vorgegeben. Vielmehr wurde es als einer der Meilensteile der Entwicklung von Computern angesehen, dass hardwareseitig nicht zwischen Programm und Daten unterschieden wird: So sind die geladenen Befehle genauso als Folgen von Einsen und Nullen codiert wie Daten jedweder Art (Zahlen, Texte etc.) und Befehle wie Daten können im gleichen Speicher abgelegt werden. Dies erlaubt nicht nur einen einfacheren Aufbau der Hardware, sondern auch größere Flexibilität bei ihrer Verwendung. Zugleich handelt man sich dadurch aber auch erheb-

liche Sicherheitsprobleme ein (s. Kapitel 10 in Kurseinheit 1 sowie Kapitel 15 in dieser Kurseinheit), von denen man zu der Zeit, als die Weichen für heutige Computerarchitekturen gestellt wurden, freilich noch keine Vorstellung hatte.

11.6 Ein- und Ausgabe

Prozessor und Hauptspeicher eines Computers dienen der programmgesteuerten Datenverarbeitung. Damit die Daten in einen Computer hinein und auch wieder heraus kommen, benötigt ein Computer zusätzlich Geräte für die Ein- und Ausgabe. Im einfachsten Fall sind das Schalter und Lampen, wie in diesem Kurstext bisher zu diesem Zweck verwendet (und wie immer noch in so manchem Spielfilm, in dem es Computer zu bestaunen gilt, dargestellt). Für die allgemeine Praxis ist das aber zu wenig.

Prozessorein- und -ausgabe

Prozessoren können selbst Anschlüsse zur Ein- und Ausgabe vorsehen. Dazu besitzen sie dann spezielle Register, die mit Leitungen verbunden sind, über die die Bits des Registers von außen gesetzt (z. B. über Schalter) oder angezeigt (z. B. über Leuchtdioden) werden können. Alternativ kann ein Teil des Hauptspeichers durch solche Register überlagert werden, die dann weder Teil des Prozessors noch Teil des Hauptspeichers sind, aber wie der Hauptspeicher über den Bus adressiert, gelesen und geschrieben werden können (sog. *Memory-mapped I/O*). Zuletzt kann ein Prozessor über einen speziellen Bus verfügen, mit dem Ein- und Ausgabegeräte unabhängig vom Hauptspeicher angesprochen werden können. In allen Fällen läuft die Ein- und Ausgabe über den Prozessor.

WIKIPEDIA

Direct Memory Access

Im Gegensatz dazu ist es auch möglich, dass Ein- und Ausgabegeräte den Bus eines Computers übernehmen und selbst Daten in den Hauptspeicher schreiben oder aus ihm lesen (der sog. **Direct Memory Access**, **DMA**). Wenn der Bus dabei geschickt genutzt wird (z. B. durch zeitlichen Versatz), kann viel Zeit gespart werden, weil der Prozessor die Ein- und Ausgabe nicht selbst vornehmen muss und stattdessen andere Dinge tun kann. Dies setzt allerdings voraus, dass die Eingabegeräte selbst mindestens über ein Steuerwerk verfügen, das den Bus bedienen kann; in der Regel haben sie sogar einen eigenen *Prozessor*.

11.6.1 Controller

Viele Ein- und Ausgabegeräte sind so komplex, dass sie selbst als (Spezial-)Computer angesehen werden können. Dazu zählen beispielsweise die sog. Grafik- und Soundkarten (Karten deswegen, weil sie wie Karten in dafür vorgesehene Steckplätze, die sie mit dem Bus des Computers verbinden, eingesteckt werden können). Diese Geräte verfügen teilweise über sehr leistungsfähige Prozessoren, die in Sachen Komplexität einer CPU um nichts nachstehen, so z. B. die sog. **Graphical Processing Units** (**GPU**s). Für einfachere Ein- und Ausgabegeräte (wie beispielsweise Tastaturen und Mäuse) sind auch die Prozessoren einfacher; sie werden dann häufig nur **Controller** genannt.

WIKIPEDIA

11.6.2 Festspeicher

Eine große Gruppe der Ein-/Ausgabegeräte sind die **Festspeicher**, auch *persistente* oder *nichtflüchtige Speicher* genannt. Sie werden, anders als der *Hauptspeicher* eines Computers, zu den Ein-/Ausgabegeräten gezählt, weil ihr Zustand das Ausschalten des Computers überdauert und in Festspeicher ausgegebene Daten zu einem beliebigen späteren Zeitpunkt wieder als Eingabe verwendet werden können. Außerdem lassen sich Festspeicher in der Regel leichter austauschen als Hauptspeicher, da sie loser mit dem Computer gekoppelt sind (und beim Austausch nicht ihren Inhalt verlieren). Festspeicher verfügen alle über ihren eigenen *Controller*; viele von diesen sind *DMA-fähig*.

Magnetspeicher

Festspeicher verwenden zur Speicherung ein Medium, auch **Datenträger** genannt, das seinen Zustand ohne Energiezufuhr halten kann. Immer noch weit verbreitet sind Magnetspeicher in Form von Magnetbändern (wie sie auch zur *analogen* Audio- und Videoaufzeichnung verwendet werden) und von rotierenden Magnetscheiben (harten, fest verbauten Platten, den sog. **Hard disks**, oder weichen, entnehmbaren **Floppy disks**, die allerdings praktisch ausgestorben sind). Bänder haben den Vorteil, dass auf ihnen riesige Datenmengen kostengünstig gespeichert und archiviert werden können, aber auch den Nachteil, dass die Daten wieder auszulesen insbesondere dann unverhältnismäßig lang dauern kann, wenn man nur einen bestimmten Teil der Daten auf einem Band benötigt (sog. **sequentieller**, im Gegensatz zu *wahlfreiem*, **Zugriff** auf Daten). Festplatten (mit ihrem wahlfreien Zugriff) sind daher heute selbst für unvorstellbar große Datenmengen das Medium der Wahl.[39]

optische Datenträger

Eine Alternative zu Magnetspeichern sind sog. **optische Datenträger**, die in der Regel auch als rotierende Scheiben beschrieben und gelesen werden. Allerdings sind diese vergleichsweise langsam und können in der Regel nur einmal (wenn überhaupt, dann nur nach einer vollständigen Löschung nochmals) beschrieben werden, wobei hier die Schreibvorgänge entweder alle am Stück oder zumindest in größeren zusammenhängenden Abschnitten erfolgen müssen (anders als bei Magnetplatten, die sehr kleinteilig beschrieben werden können). Dafür sind die Medien (CD, DVD oder Bluray) im Vergleich zu Magnetplatten sehr billig und müssen nicht in gleichem Maße vor Staub und Beschädigung geschützt werden (auch wenn die Speicherdichte deutlich geringer ist als bei Magnetplatten; sie liegt bei ca. einem Achtzigstel).

Flash-Speicher

Seit einigen Jahren wird den Festplatten von sogenannten **SSD**s (für **Solid State Drives**, wobei Drive — für Laufwerk — hier im wörtlichen Sinne gar nicht mehr zutrifft) der Rang abgelaufen, zumindest was die Festspeicher in Laptops angeht (in Handys und Tablets sind sie schon immer verbaut worden). Dabei handelt es sich um sog. **Flash-Speicher**. Datenspeicherung auf einem gelöschten Flash-Speicher funktioniert, indem eine elektrische Ladung über eine Überspannung in einen ansonsten nahezu ideal isolierten Elektronenspeicher (Kondensator) eingebracht wird, aus dem der Ladungszustand ohne

---

[39] Google fügte 2016 angeblich täglich ein Petabyte (das sind 10^{15} Bytes oder 1024 Terabytes) seiner Speicherkapazität hinzu, das meiste davon für YouTube-Videos.

Stromfluss aus dem Speicher wieder ausgelesen werden kann. Vor einem erneuten Beschreiben muss ein Flash-Speicher erst gelöscht werden, was wiederum eine Überspannung (umgekehrter Polarität) erfordert, die allerdings nicht speicherzellenweise angelegt wird, so dass immer größere Bereiche eines Flash-Speichers auf einmal gelöscht werden müssen (anders als bei einem Magnetspeicher, bei dem einfach die magnetische Polarität einer Speicherstelle geändert wird). Selbst wenn nur ein Bit zurückgesetzt (gelöscht, also auf 0 gesetzt) werden soll, muss daher zunächst der ganze Sektor ausgelesen und zwischengespeichert (gepuffert), dann der Sektor gelöscht und dann der Inhalt des Zwischenspeichers (Puffers) mit dem zurückgesetzten Bit wieder in den Sektor geschrieben werden (wobei man aus Geschwindigkeitsgründen auch einen schon oder noch leeren Sektor nehmen und das Löschen des alten parallel oder verzögert vornehmen kann). Dadurch sind Flash-Speicher beim Ändern von Daten in der Regel deutlich langsamer als beispielsweise Magnetplatten (nicht aber unbedingt beim Schreiben von neuen Daten in einen noch leeren oder schon gelöschten Speicherbereich). Man erkennt, dass Flash-Speicher ursprünglich für Fotoapparate gedacht waren.

Read-Only Memory

Vorläufer der Flash-Speicher sind sogenannte *EPROMs* (für Erasable Programmable *Read-Only Memory*), die durch Bestrahlung mit UV-Licht gelöscht werden können, und *EEPROMs* (für elektrisch löschbare EPROMs). In beiden Fällen ist aber das Löschen ein eher seltener Vorgang, der beispielsweise im Rahmen eines Updates durchgeführt wird. So wird in EEPROMs ein Teil des Betriebssystems, das sogenannte *BIOS* (für basic input output system; s. Kapitel 12) eines Computers, gespeichert. Vorstufen dazu sind *PROMs*, bei denen ein Bit durch Wegschmelzen einer Leiterbahn gespeichert wird (und die nicht wieder gelöscht werden können), und *ROMs*, bei denen der Zustand der Bits bereits bei der Herstellung festgelegt wird. PROMs und ROMs kommen fast nur noch in Geräten vor, auf die keine Updates aufgespielt werden können oder sollen. Der C64 beispielsweise hatte sein gesamtes Betriebssystem und seinen *BASIC-Interpreter* auf ROM gespeichert.

Nicht zuletzt kann auch Papier als Festspeicher verwendet werden, z. B. indem man es mit einem Drucker bedruckt. Mittels *optischer Zeichenerkennung* (engl. optical character recogniton, *OCR*) kann dieser Speicher auch wieder eingelesen werden. Auch wenn man diese Form der Speicherung belächeln mag, hat sie doch den Vorteil, dass die auf Papier gespeicherten Daten selbst bei vollständigem Energieverlust (oder bei Verlust elektronsicher Geräte) Menschen als Informationsquelle dienen können. Eine solche halten Sie vielleicht gerade in Ihren Händen.

11.6.3 A/D- und D/A-Wandler

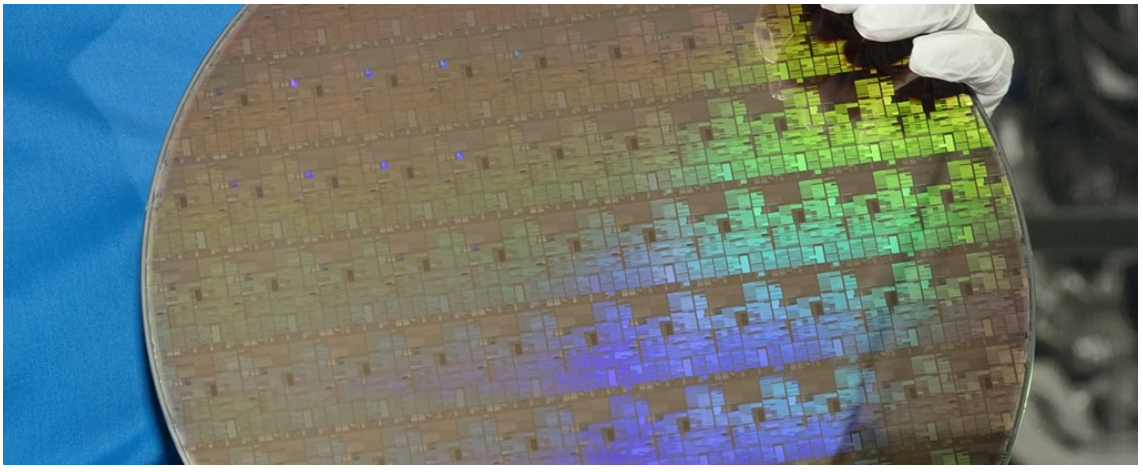
Während viele Ein- und Ausgabegeräte direkt digitale Signale liefern bzw. konsumieren, benötigen viele Anwendungen der Digitalisierung Schnittstellen zur *analogen* Welt. An dieser Schnittstelle kommen beispielsweise Mikrofone, Kameras, Lautsprecher und Bildschirme zum Einsatz.

Mit sog. **Analog/Digital-Wandlern** (oder kurz **A/D-Wandlern**) wird ein analoges Signal *diskretisiert* (vgl. Abschnitt 6 in Kurseinheit 1). Dafür ist ein *Takt* erforderlich sowie eine Schaltung, die mit den Taktschlägen einen analogen Momentanwert, der entweder direkt als Höhe einer analogen Spannung vorliegt oder durch eine analoge (elektronische) Schaltung in eine solche überführt wird, in einen diskreten Wert, eine Dualzahl, konvertiert. Die resultierende Zahlenfolge wird dann entweder in einem Pufferspeicher des Geräts zwischengespeichert, an ein Register des Computers übergeben oder (über *DMA*) in den *Hauptspeicher* des Computers geschrieben.

Bei der sog. **Digital/Analog-Wandlung** geschieht dann genau das Umgekehrte: Eine Folge von diskreten Zahlenwerten wird in einem vorgegebenen Takt in analoge Spannungen umgesetzt. Analoge Schaltungen und Geräte sorgen dann für die notwendige Glättung (eine Art Interpolation zur Entfernung der Stufen), Verstärkung und Umsetzung in beispielsweise Schall- oder Lichtwellen. Sound- und Grafikkarten, die solche Digital/Analog-Wandlungen vornehmen, verfügen dafür in der Regel über große Pufferspeicher, die vom Computer kontinuierlich beschrieben werden müssen.

11.7 Computerarchitektur

Die hier erfolgte Beschreibung der Hardware eines Computers beginnt recht kleinteilig, wird dann aber immer grobzügiger. So unternehme ich erst gar keinen Versuch, auf Einzelheiten der Realisierung eines Bussystems einzugehen oder die einzelnen Komponenten eines Prozessors als Schaltwerke darzustellen, ja selbst die Realisierung eines getakteten Nicht-Gatters (Abschnitt 11.2.1) wird nicht gezeigt. Das liegt zum einen daran, dass die technische Realisierung dieser Schaltungen an der jeweils verwendeten Technologie (Halbleiter statt Relais, wobei es viele verschiedene Halbleitertechnologien gibt) hängt, zum anderen daran, dass die Architektur heutiger Computer, die sog. **Von-Neumann-Architektur**, vermutlich nicht optimal ist, also nicht ausgeschlossen werden kann, dass sich die Architektur in Zukunft wesentlich ändert. Die gewählte Darstellung sollte jedoch ausreichen, zu vermitteln, dass die Hardware eines Computers ein kumulatives Menschenwerk ist, das hierarchisch und so aus Bauteilen (von denen jedes für sich eine bestimmte, wohldefinierte Funktion erfüllt) zusammengesetzt ist, dass man am Ende eine Maschine erhält, die beliebige Zeichenspiele ausführen kann. Man kann einen Computer als Ganzes nur verstehen, wenn man darauf vertraut, dass die jeweiligen Bauteile auf der Ebene, die man gerade betrachtet, ihre Funktion erfüllen; betrachtet man einen ganzen Computer auf der Ebene der *Gatter*, sieht man gar nichts.



Siliziumscheibe (Wafer) mit aufgedampften Gattern, bestehend aus 30 Milliarden Transistoren pro Chip (Quelle: www.sciencealert.com/new-computer-chips-can-fit-30-million-transistors-on-your-fingertip)

11.8 Der Computer als Universalmaschine

So wie Einsen und Nullen dazu ausreichen, jede Information mit hinreichender Genauigkeit darzustellen, so reichen Computer dazu aus, alles zu tun, was man mit Einsen und Nullen und den damit repräsentierten Daten machen kann. Die Grenzen liegen

- theoretisch in den Grenzen der *Berechenbarkeit* (ein theoretisches Konstrukt, das zwar *Leibniz*' Calculemus, also den Traum von einem automatischen Verfahren, das jede Frage beantworten kann, beendet hat, dessen Bedeutung für die Praxis aber eher umstritten ist) sowie
- praktisch in der endlichen Speicherkapazität (*endlicher Automat*!) und der Rechendauer, die je nach Problem so groß werden kann, dass auf eine Antwort zu warten nicht sinnvoll ist, egal wie schnell Computer noch werden.

Da Computer nichts Materielles produzieren und auch keine Materie verändern können, sind sie wohl nur mit dem Gehirn vergleichbar, das freilich seinesgleichen sucht und dem „Elektronenhirn" in Sachen Universalität um einiges voraus ist. So kann man von keinem heute bekannten Computer behaupten, er verstünde, was er tut, geschweige denn, warum. Es wird noch ein Weile dauern, bis der erste Computer (und nicht sein Erbauer oder Programmierer) vor Gericht erscheinen muss.

11.9 Zusammenfassung

Wie Sie gesehen haben ist Hardware zwar technisch anspruchsvoll (zu dem Grad, dass einen heutigen Computer aus einzelnen Gattern von Grund auf neu zu bauen aussichtslos erscheint — aktuelle CPUs haben über eine Milliarde Transistoren!), aber sehr regelmäßig aufgebaut. Insbesondere die stufenweise Komponierbarkeit von Bausteinen zu größeren

Einheiten, den sog. *Modulen* (Schalter zu Gattern, Gatter zu Addierern, Addierer zu Rechenwerken usw.), deren Verbindung durch elektrische Leitungen hergestellt werden, die nur zwei Zustände übertragen, sowie die Skalierung durch das simple Prinzip *mehr vom Selben* erlauben es, extrem komplexe Hardware weitgehend fehlerfrei zu entwerfen und zu bauen. Das gleiche (oder auch nur ähnliches) kann man von *Software* leider nicht behaupten.

Hier könnte ein Kurstext über die Grundlagen der Digitalisierung zu Ende sein: Mit der Codierung von Zahlen, Wahrheitswerten, Texten und Signalen als Zeichenfolgen, genauer als *Folgen von Einsen und Nullen*, und mit der Umsetzung von *Zeichenspielen*, oder Operationen auf diesen Zeichenfolgen, durch Hardware hat man alles, was man für die Digitalisierung in ihren mannigfaltigen Erscheinungsformen braucht. Allerdings hätte der Digitalisierungsfortschritt niemals seine atemberaubende Geschwindigkeit erreicht, wenn jedes neue Problem in Form eines Zeichenspiels mit Einsen und Nullen gelöst werden müsste. Stattdessen wurden über die Jahre immer weitere Aufbauten geschaffen, die es einem erlauben, eine Problemlösung auf immer *höheren Ebenen der Abstraktion* zu beschreiben, ohne dabei die Universalität oder die Umsetzbarkeit mit der gegebenen Hardware zu verlieren. Und so befasst sich der ganze Rest des Kurstextes denn auch nur mit diesen Aufbauten.

Sie sind übrigens allesamt als Software realisiert. Das heißt, sie sind programmiert.

12 Betriebssystem

Kurs

Ein Betriebssystem ist eine Sammlung von Programmen, die zum Betrieb einer Computerhardware notwendig sind. Betriebssysteme sind immer an eine bestimmte Hardware angepasst und werden in der Regel mit der Hardware zusammen verkauft. Manche Betriebssysteme (z. B. Unix) sind dennoch hardwareübergreifend verfügbar, was nicht zuletzt durch eine Vereinheitlichung von Hardwarearchitekturen (speziell der PC-Architektur, der sich im Jahr 2006 auch Apple angeschlossen hat) begünstigt wurde. Da moderne Betriebssysteme zudem in Schichten aufgebaut sind, muss für eine Übertragung (*Portierung* genannt) auf einen anderen Computertyp in der Regel nur ein kleiner Teil, darunter der sog. *Betriebssystemkern* sowie die *Gerätetreiber*, angepasst werden.

Man beachte, dass für die Verwendung von Computern das Betriebssystem allein nicht ausreicht — nur mit dem Betriebssystem kann man keine Anwendungsprobleme lösen. Dafür werden zusätzlich sog. *Anwendungsprogramme* benötigt, wie sie in Kurseinheit 3, Kapitel 23 behandelt werden. Anwendungsprogramme profitieren vom Betriebssystem, indem sie sich nicht um Details wie die der Ein- und Ausgabe auf den verschiedenen zu einem Computer gehörenden oder daran angeschlossenen Geräten kümmern müssen. Auch lassen sich Anwendungsprogramme leicht von einem Computertyp auf einen anderen übertragen, wenn das Betriebssystem für sie eine einheitliche, hardwareunabhängige *Schnittstelle* bereitstellt. Voraussetzung dafür ist jedoch, dass auf beiden Computertypen dasselbe,

oder zumindest ein funktionsgleiches, Betriebssystem läuft. Sog. *eingebettete Software* (Kapitel 25) kommt hingegen häufig ohne (oder mit einem deutlich reduzierten) Betriebssystem aus.

12.1 Systemaufrufe

Damit ein Anwendungsprogramm die Funktionalität eines Betriebssystems nutzen kann, muss es seine Funktionen aufrufen. Dies geschieht mittels sog. **Systemaufrufe**. Systemaufrufe kann man sich zunächst wie *Unterprogrammaufrufe* vorstellen (s. dazu auch Abschnitt 9 in Kurseinheit 1); bei den meisten Computern (Hardware plus Betriebssystem) geht der Prozessor bei Systemaufrufen aber zusätzlich in einen sog. **privilegierten Modus** über, in dem er Operationen und Ressourcenzugriffe durchführen kann, die einem Anwendungsprogramm aus Sicherheitsgründen nicht erlaubt werden sollen. Typische Systemfunktionen führen das Lesen und Schreiben von bzw. in Dateien durch (Abschnitt 12.5) oder stellen Hauptspeicher zur Verfügung und schützen ihn für die exklusive Nutzung durch ein Programm (Abschnitt 12.4).

Um die Systemfunktionen, die durch Systemaufrufe ausgeführt werden sollen, auch tatsächlich aufzurufen, bieten manche Betriebssysteme sog. **Sprungtabellen**, das sind festgelegte Speicherbereiche, an der die tatsächlichen *Speicheradressen* der Unterprogramme hinterlegt sind. Diese *Indirektion* dient der Versionsunabhängigkeit (verschiedene Versionen eines Betriebssystems können die Systemfunktionen an verschiedenen Stellen speichern, ohne dass ein Anwendungsprogramm etwas davon wissen muss). Unabhängig davon rechnet man die Systemfunktionen eines Betriebssystems zu seinem sog. **Application Programming Interface** (API), also einer *Schnittstelle*, die Anwendungsprogramme verwenden können, um das Betriebssystem zu nutzen.

12.2 Hochfahren

Wenn Sie heute Ihren persönlichen Computer einschalten, dann dauert es eine Weile, bis er Sie in seiner schönen bunten Welt mit Desktop etc. willkommen heißt. Diese Weile veranschaulicht die Distanz zwischen der Hardware (als nur Einsen und Nullen verarbeitendes Gerät) und der Bedienoberfläche Ihres Computers, die das Betriebssystem für Sie (und für die von Ihnen genutzten Anwendungsprogramme) überbrückt.

Boot strapping

Da das Betriebssystem eines Computers auf einem Festspeicher des Computers gespeichert ist und Festspeicher exklusiv vom Betriebssystem verwaltet werden (s. Abschnitt 12.5), muss ein kleiner Teil des Betriebssystems ohne das Betriebssystem ausgeführt werden können. Dieser für das Starten des Computers unverzichtbare Teil, bei PCs als **Basic Input/Output System** (**BIOS**) bekannt, ist ein Programm, das beim Starten des Computers als erstes ausgeführt wird. Es kann dafür beispielsweise in einem nichtflüchtigen Teil des Hauptspeichers (einem *ROM*, das an dieser Stelle den flüchtigen Speicher ersetzt oder temporär überlagert) abgelegt sein, dessen erste *Speicheradresse* dem (fest eingestellten)

Wert des *Befehlszählers* der CPU beim Einschalten entspricht. Mit der Ausführung dieses Programms beginnt dann die sog. **Boot-Sequenz**, in der nach und nach weitere Teile des Betriebssystems geladen und gestartet werden. Die Boot-Sequenz heißt so, weil sie ein Henne-Ei-Problem löst und sich zu diesem Zweck wie einst *Münchhausen* am eigenen Schopf aus dem Sumpf zieht oder, wie es im Englischen heißt, sich an den eigenen Stiefelschlaufen (bootstraps) hochzieht (woraus dann im englischen Computersprech über den Umweg „to bootstrap" das Verb „to boot", zu Deutsch „booten", geworden ist). „Schopfsequenz" wäre eine mindestens ebenso selbsterklärende Wortwahl gewesen.

Durch den schichtenweisen Start des Betriebssystems und dessen Laden von einem (austauschbaren) Speichermedium ist es möglich, dieselbe Hardware mit verschiedenen Betriebssystemen zu betreiben. Wenn das Betriebssystem aber erst einmal gestartet wurde, muss man es erst wieder beenden (*herunterfahren*; s. Abschnitt 12.8), bevor man es wechseln kann, es sei denn, man startet ein zweites Betriebssystem innerhalb einer *virtuellen Maschine* (das ist ein Programm, das einen weiteren Computer auf dem Computer simuliert, auf dem es läuft).

12.3 Gerätetreiber

Technischer Fortschritt und technische Vielfalt bedingen, dass sich in einem Computer verbaute und an ihn angeschlossene externe Geräte selbst derselben Klasse technisch unterscheiden und deswegen verschieden angesteuert werden müssen. Während man Maus, Tastatur und Bildschirm mittlerweile in der Regel problemlos gegen andere Modelle auswechseln kann, so unterscheiden sich beispielsweise verschiedene Druckermodelle erheblich in dem, wie sie vom Computer angesprochen werden müssen, damit sie ihren Dienst erfüllen. Da das Betriebssystem des Computers diese Unterschiede nicht kennen kann (die anschließbaren Geräte sind dem Hersteller des Betriebssystems gar nicht alle bekannt), muss ein logischer Adapter zwischen Betriebssystem und Gerät her. Einen solchen Adapter nennt man **Gerätetreiber** oder, verkürzt, einfach nur **Treiber** (engl. *driver*).

Gerätetreiber sind *Software* und werden grundsätzlich mit den Geräten ausgeliefert (auf einem Datenträger oder als Download über das Internet). Wenn die Geräte den sog. *Plug-and-play*-Modus unterstützen, dann geben sie sich dem Betriebssystem gegenüber beim ersten Anschließen zu erkennen (über eine eindeutige Gerätekennung). Das Betriebssystem hat dann entweder den passenden Treiber schon verfügbar (in seiner Treiberdatenbank) oder sucht einen solchen auf dem mitgelieferten Datenträger oder im Internet. Leider ist es heute gängige Praxis, zunächst mangelhafte Treiber mit den Geräten auszuliefern (es ist scheinbarer schwieriger, einen Treiber für ein Gerät zu entwickeln als ein Gerät selbst — auch das sagt einiges über das Verhältnis von Hard- und Software aus); in solchen Fällen hilft (manchmal) ein Treiberupdate oder das Rückfallen auf einen sog. generischen Treiber (falls verfügbar), der zwar nicht alle, aber doch zumindest die Grundfunktionen des Geräts (die bei allen Geräten einer Klasse vorhanden sein sollten) unterstützt.

12.4 Speicherverwaltung

Wie in Abschnitt 11.5 erläutert wird der *Hauptspeicher* eines Computers grundsätzlich für Programme und Daten gleichermaßen genutzt. Laufen[40] auf einem Computer mehrere Programme gleichzeitig, dann teilen auch die sich alle denselben Speicher. Dies erlaubt nicht zuletzt eine maximale Nutzung des verfügbaren Hauptspeichers, der trotz heutiger Hauptspeichergrößen immer noch eine knappe Ressource ist (und das vermutlich auch immer sein wird).

Speicherschutz

Voraussetzung für ein problemloses Nebeneinander von Programmen ist jedoch, dass kein Programm Speicher eines anderen Programms verwendet. Das Betriebssystem weist zu diesem Zweck jedem Programm seinen eigenen Speicher exklusiv zu. Um die exklusive Verwendung durchzusetzen, also effektiv zu verhindern, dass ein Programm die Speicherzuweisung des Betriebssystems umgeht, ist jedoch eine Hardwareunterstützung notwendig, die bei jeder *Speicheradressierung* prüft, ob die *Adresse* im zugewiesenen Bereich liegt. Das Vorkommen einer sog. **Speicherschutzverletzung** wird in der Regel durch sofortigen *Programmabbruch* (vom Nutzer als *Absturz* wahrgenommen) quittiert.

Speicherfreigabe

Eine andere Voraussetzung für eine problemlose Speicherverwaltung ist, dass das Betriebssystem den von einem Programm belegten Speicher nach seiner Beendigung wieder freigibt und somit anderen Programmen zur Verfügung stellt. Ist das nicht der Fall, spricht man von **Speicherlecks**. Speicherlecks sind Zeichen der Umgehung der Speicherverwaltung des Betriebssystems in einem Programm (oder Zeichen eines mangelhaften Betriebssystems).

Speicher-fragmentierung

Durch fortgesetztes Starten und Beenden von Programmen wird der Hauptspeicher **fragmentiert**, d. h., die freien Speicherbereiche sind über den Hauptspeicher verteilt. In solchen Fällen hilft eine *indirekte Adressierung* des Hauptspeichers, bei der den Programmen ein zusammenhängender Speicherbereich vorgespielt wird, dieser aber in Wirklichkeit fragmentiert ist. Eine solche indirekte Adressierung, die am besten durch Hardware umgesetzt wird, erlaubt es auch, den Speicher eines Programms während seines Ablaufs zu vergrößern. Können die Speicheranforderungen während der *Laufzeit* eines Programms nicht (mehr) bedient werden, kommt es zu einem *Out-of-memory-Fehler*, dessen Folge in der Regel ebenfalls ein sofortiger *Programmabbruch* ist. Bevor ein Programm also mehr Speicher vom Betriebssystem anfordert, sollte ersteres prüfen, ob letzteres noch freien Speicher verfügbar hat.

virtueller Speicher

Die physikalische Begrenzung der Größe des Hauptspeichers kann umgangen werden, indem man den Speicher **virtualisiert**. Dazu wird ein größerer logischer

---

[40] Im Informatikjargon „laufen“ Programme, nachdem sie „gestartet“ wurden. Wenn sie nicht mehr laufen, sind sie entweder „beendet“ (planmäßig) oder „abgestürzt“ (unplanmäßig). Letzteres ist von „hängen“ zu unterscheiden, was bedeutet, dass ein Programm nicht aufhört zu laufen, obwohl es das eigentlich sollte. Sowohl beim Absturz als auch beim Hängen handelt es sich nicht um ein Unglück oder gar höhere Gewalt, sondern man darf von einem Programmierfehler ausgehen.

Speicheradressraum auf den kleineren physikalischen Adressraum abgebildet, so dass mehrere logische Speicherblöcke — auch *Speicherseiten* oder kurz *Seiten* genannt — demselben physikalischen Speicherblock entsprechen. Um die logischen Speicherblöcke physikalisch zu trennen, werden sie auf einen Festspeicher ausgelagert, von dem einzelne Speicherblöcke bei Bedarf in den Hauptspeicher geladen und, falls der Speicher auch beschrieben wird, danach wieder in den Festspeicher ausgelagert werden. Bei einem Speicherzugriff wird also zunächst geprüft, ob der (*logischen*) *Speicheradresse* unmittelbar eine (*physikalische*) Adresse im Hauptspeicher zugeordnet werden kann (weil der logische Speicher sich auch physikalisch gerade da befindet); ist das der Fall, kann der Zugriff mit der physikalischen Adresse erfolgen. Ist das nicht der Fall, muss zunächst ein freier (gerade nicht benutzter) Speicherblock im Hauptspeicher gefunden werden, in den dann der der logischen Speicheradresse entsprechende Speicherblock aus dem Festspeicher geladen wird; erst danach kann die logische Adresse in eine physikalische umgesetzt werden. Wie man sich leicht vorstellen kann, benötigen das Laden und Speichern von Speicherblöcken von bzw. in einem Festspeicher Zeit — sobald der physikalische Hauptspeicher nicht mehr ausreicht, um alle Anforderungen der laufenden Programme zu bedienen, müssen Speicherblöcke aus- und eingelagert werden, was den Computer langsam macht.

Idealerweise würde die gesamte Hauptspeicherverwendung eines Computers strikt unter die Kontrolle des Betriebssystems gestellt. Dies wird jedoch aus Effizienzgründen (die Kontrolle belegt Kapazitäten des Computers) bisweilen unterlassen. Bei angeschlossenen Festspeichern ist das jedoch anders.

12.5 Dateisystem

Genauso wie den Hauptspeicher eines Computers kann man auch seine angeschlossenen Festspeicher als eine nummerierte Reihe von Speicherzellen betrachten, die sich einzeln direkt adressieren lassen. Allerdings verbietet das Betriebssystem in der Regel, Festspeicher so auszulesen oder zu beschreiben — aus verschiedenen Gründen.

- Ein Grund ist, dass Festspeicher über die Zeit verschleißen und so einzelne Bereiche nicht mehr zuverlässig gelesen oder beschrieben werden können. Das Betriebssystem erkennt dies daran, dass die Zahl der gescheiterten Lese- oder Schreibversuche in diesen Bereichen langsam ansteigt (am Anfang lassen sich diese Fehler noch durch wiederholtes Lesen oder Schreiben korrigieren). Diese Bereiche werden dann, nachdem ihr Inhalt in andere, freie Bereiche kopiert wurde, gesperrt und fortan nicht mehr verwendet. Die Kapazität des Festspeichers sinkt dadurch zum einen; zum andern kann sich niemand darauf verlassen, dass die Daten dauerhaft an der Stelle stehen, an die sie ursprünglich einmal geschrieben wurden.
- Ein anderer Grund ist, dass sich verschiedenen Anwendungsprogramme (s. Kapitel 16) einen Festspeicher teilen. Diese Programme müssten sich nicht nur untereinander einigen, welches seine Daten wo speichert, sondern sich auch noch über die Ablage von gemeinsam genutzten Daten verständigen.

- Ein dritter Grund ist, dass nicht immer von Anfang an feststeht, wie viel Festspeicher ein Programm braucht. Anstatt ein maximales Kontingent zu reservieren, das zum einen vielleicht gar nicht ausgeschöpft wird, das sich zum anderen vielleicht irgendwann doch als zu klein herausstellt, ist es sinnvoll, den belegten Festspeicherbereich dynamisch wachsen und schrumpfen zu lassen.

All das hat dazu geführt, dass Festspeicher vollständig vom Betriebssystem verwaltet werden. Alle Lese- und Schreibvorgänge von Programmen erfolgen damit nicht direkt, sondern als Aufträge an das Betriebssystem. Das Betriebssystem führt Buch darüber, wo die Daten liegen, und sorgt dafür, dass die Speicherplatzbedürfnisse einzelner Programme im Rahmen der Gesamtkapazität der Festspeicher bedient werden.[41] Dafür führt es die Abstraktion **Datei** (engl. **File**) ein.[42]

Datei

Eine Datei ist eine benannte Folge von nicht notwendigerweise zusammenhängenden Speicherzellen, die in aller Regel auf einem Festspeicher angesiedelt sind. Die *Adressen* der einer Datei zugeordneten Speicherbereiche werden in einer sog. **Dateizuordnungstabelle** (engl. **File Allocation Table**, oder **FAT**) hinterlegt. Der Zugriff auf die in einer Datei gespeicherten Daten erfolgt ausschließlich über das Betriebssystem unter Angabe eines sog. **File handle** (einer Art *Zeiger* auf die Datei) sowie ggf. einer relativen Position der zu lesenden oder zu schreibenden Daten innerhalb der Datei. Relative Positionen werden vom Anwendungsprogramm verwaltet; das Betriebssystem rechnet sie in absolute Positionen auf dem Festspeicher um und verwehrt ggf. Lese-und Schreibzugriffe jenseits des Endes der Datei. Dateien können nach festgelegten Regeln auch wachsen; bei sog. **sequentiellen Dateien**, bei denen Daten immer nur der Reihe nach gelesen bzw. geschrieben werden können (Bandspeicher!), können z. B. neue Daten am Ende angefügt werden. U. a. das Wachsen von Dateien führt auf Dauer zu einer *Fragmentierung des Festspeichers*, die sich u. U. nachteilig auf die Zugriffsgeschwindigkeit auswirkt.

Verzeichnisse

Bei Unix-artigen Betriebssystemen und auch bei Windows enthält jeder Festspeicher eine baumartige Struktur von **Dateiverzeichnissen** und **Unterverzeichnissen** (engl. **Directories** und **Subdirectories**), in denen alle auf dem Speicher vorhandenen Dateien aufgelistet sind. Dabei sind Verzeichnisse und Unterverzeichnisse selbst nur (besondere) Dateien, genau wie Programme, die ebenfalls als Dateien gespeichert werden, die allerdings den Sonderstatus haben, ausführbar zu sein. Darüber hinaus wissen diese Betriebssysteme aber nichts über den Inhalt, die Verwendung oder die Verwendbarkeit der Dateien — es sind immer nur Folgen von Bytes.[43]

---

[41] Wie zentral diese Funktion ist, kann man schon daran ablesen, dass das erste PC-Betriebssystem von IBM noch PC DOS (DOS für Disk Operating System) hieß.

[42] Datei ist eigentlich ein Kunstwort (zusammengefügt aus Daten und Kartei) — wenn es heute nicht mehr so klingt, dann sagt das einiges über unsere Sozialisierung mit Computern und deren Betriebssystemen aus.

[43] Bei Dateien hat sich *Byte* als Verarbeitungsgröße gehalten, anders als bei Prozessoren, wo das *Wort* die Bitzahl festlegt (wobei die Wortgröße von der jeweiligen Hardware abhängt).

Neutralität des Dateisystems

Diese Neutralität des Dateisystems gegenüber dem Inhalt der Dateien wurde einst als große Errungenschaft gefeiert. Ich bin mir da aber nicht so sicher. So hängen viele Dateien doch an konkreten Anwendungsprogrammen (auch, was die Lesbarkeit angeht, denn viele Anwendungsprogramme verwenden proprietäre, sog. *Binärformate*, um ihre Daten zu speichern) und der Umstand, dass zumindest das Betriebssystem einen nicht daran hindert, Dateien mit einem Programm zu lesen oder zu schreiben, das gar nicht dafür vorgesehen war, ist nicht zuletzt ein *Sicherheitsproblem*. In Windows hängen stattdessen die zum Lesen einer Datei gedachten Anwendungen über die Dateinamenerweiterung, die Zeichen nach dem letzten Punkt im Dateinamen, an den Dateien, aber da sowohl die Erweiterung eines Dateinamens als auch die Zuordnung zu Anwendungsprogrammen vom Benutzer beliebig geändert werden können, stellt das keinen wirksamen Schutz dar (standardmäßig verbirgt Windows — vielleicht deswegen — mittlerweile die Dateinamenerweiterung in Dateiverzeichnissen, was die Zuordnung von Datei zu Anwendungsprogramm als eine interne erscheinen lässt).

Öffnen und Schließen von Dateien

Dateien müssen vor dem Lesen oder Schreiben geöffnet und danach wieder geschlossen werden. Dies hat zum einen die Funktion, dem Betriebssystem die Kontrolle darüber zu ermöglichen, dass Dateien exklusiv und nur zu einem bestimmten Zweck verwendet werden (z. B. nur zum Lesen oder nicht zum gleichzeitigen Schreiben), zum anderen werden mit den Dateien sog. *Pufferspeicher* im Hauptspeicher des Computers oder dem Controller des Festspeichers reserviert, die Teile des Inhalts der Datei für schnelleren Zugriff replizieren und die daher mit dem Inhalt der Datei zu bestimmten Zeitpunkten abgeglichen werden sowie bei einer Änderung spätestens beim Schließen der Datei wieder in den Festspeicher zurückgeschrieben werden müssen. Das Öffnen und Schließen von Dateien erfolgt ausschließlich durch das Betriebssystem, wird aber von Anwendungsprogrammen angestoßen.

Dateibesitz

Auf Computern, die von *mehreren genutzt* werden (nacheinander oder gleichzeitig; der sog. *Persönliche Computer*, der *PC*, gehört definitionsgemäß eigentlich nicht dazu), haben Dateien zudem einen Besitzer. Lesender und schreibender Zugriff auf eine sowie Ausführung von einer Datei können dann dem Benutzer oder einer Gruppe von Benutzern vorbehalten oder auch allen erlaubt sein. Die ersten beiden Möglichkeiten verlangen natürlich, dass sich der Benutzer dem Betriebssystem gegenüber ausweist; in der Regel geschieht dies bei der sog. **Anmeldung** oder dem **Einloggen** durch Angabe von Name und Passwort. Einmal angelegt können die Schreib-, Lese- und Ausführungsrechte an Dateien nur von dem Besitzer oder einem sog. **Superuser** (oder **Administrator**), der sog. *Root-Rechte* besitzt, geändert werden. U. a. deswegen sind Angriffe auf Root-Rechte so gefährlich.

12.6 Programmverwaltung und Multitasking

Um *Anwendungsprogramme* (Abschnitt 23) auf einem Computer zur Ausführung zu bringen (ein Vorgang, der auch *Programmaufruf* genannt wird), müssen sie zunächst in den Hauptspeicher geladen werden. Dies wird vom Betriebssystem vorgenommen. Sodann wird

dem Programm ein Speicherbereich für seine Daten zugewiesen und das Programm gestartet (ebenfalls vom Betriebssystem). Die Größe des zugewiesenen Speicherbereichs kann dabei ein Parameter des Programmaufrufs sein. Nach (planmäßiger oder unplanmäßiger) Beendigung muss das Betriebssystem dafür Sorge tragen, dass alle vom Programm belegten Ressourcen (dazu zählen neben dem Hauptspeicher u. a. auch belegte *File handle*) wieder freigegeben werden, da ansonsten ein *Ressourcenleck* auftritt. Dazu zählt auch, dass geöffnete Dateien wieder geschlossen werden, wenn dies vom Anwendungsprogramm nicht schon getan wurde (was insbesondere bei Programmabstürzen regelmäßig der Fall ist). Nicht zuletzt muss ein Betriebssystem auch in der Lage sein, ein Programm zwangsweise zu beenden, spätestens wenn dies nicht mehr aus dem Programm selbst heraus möglich ist (weil es sich „aufgehängt" hat; vgl. Fußnote 40).

Wenn auf einem Computer mehrere Programme gleichzeitig laufen (was, außer bei *eingebetteten Systemen*, wie sie in Abschnitt 25 beschrieben werden, praktisch immer der Fall ist), dann befinden sich diese Programme in unmittelbarer Konkurrenz um die Ressourcen des Computers. Für den Hauptspeicher wurde das schon in Abschnitt 12.4 thematisiert.

Wenn die Anzahl der auf einem Computer gleichzeitig laufenden Programme die Anzahl der Prozessoren, die Programme ausführen können (die CPUs; Abschnitt 11.5) übersteigt, dann ergibt sich auch eine Konkurrenz um Prozessoren (und das mit der Gleichzeitigkeit hat sich erledigt — ein Prozessor kann sich nicht teilen). Stattdessen sorgt dann das Betriebssystem dafür, dass jedem laufenden Programm — in diesem Kontext auch **Prozess** genannt — sog. **Zeitscheiben** zur Verfügung gestellt werden, während derer es einen Prozessor für sich hat. Das bedingt allerdings, dass ein laufendes Programm jederzeit (mittels einer besonderen Vorkehrung, dem sog. **Interrupt**) unterbrochen werden kann, um ein anderes auf „seinem" Prozessor laufen zu lassen. Damit das unterbrochene Programm später nahtlos fortgesetzt werden kann, ist es notwendig, dass der aktuelle Prozessorzustand, bestehend aus dem Inhalt all seiner Register (inkl. Programmzähler), zwischengespeichert wird. Diese sog. **Prozessumschaltung** ist dabei selbst ein Programm, das jedoch nicht wie andere unterbrochen werden kann.

WIKIPEDIA

Multitasking

Damit ein Prozessor möglichst gut ausgenutzt und die Gesamtlaufzeit aller gleichzeitig ausgeführten Programme minimiert werden kann, muss das Betriebssystem erkennen können, was ein Programm gerade tut. Wartet es beispielsweise gerade auf Eingaben (was sich am Aufruf entsprechender Betriebssystemfunktionen erkennen lässt), kann es unterbrochen werden, ohne dass dies zu einer unliebsamen Verzögerung seiner Ausführung führt. Diese Beobachtung laufender Programme kostet jedoch selbst Zeit und so ist die geschickte Umsetzung des Zeitscheibenverfahrens, auch **Multitasking** genannt, ein wesentliches Qualitätsmerkmal eines Betriebssystems. Dabei hat das Multitasking auch die Aufgabe, die Ausführung der Programme auf mehrere Prozessoren zu verteilen (die meisten aktuellen *Mikroprozessoren* verfügen über *mehrere Prozessorkerne*, die logisch unabhängige Prozessoren repräsentieren).

WIKIPEDIA

Time sharing und Terminals

Vom Multitasking abzugrenzen, aber dennoch mit ihm verwandt, ist das sog. **Time sharing**, bei dem *mehrere Benutzer gleichzeitig* an einem

Computer arbeiten. Dies bedingt jedoch die Existenz separater Ein- und Ausgabegeräte für jeden Benutzer (Tastatur und Bildschirm, auch als **Terminal** oder **Konsole** zusammengefasst). Solche Terminallösungen findet man heute immer noch im Bereich der Großrechner (bei Banken, Versicherungen und Behörden), wobei die Terminals inzwischen meistens durch Arbeitsplatzrechner (PCs) mit einer *Terminalemulation* abgelöst wurden.

Multitasking und Fenster

Das Multitasking hat insofern ein umgekehrtes Problem zu lösen, als hier mehrere Programme gleichzeitig mit nur einem Bildschirm und nur einer Tastatur zu bedienen sind. Dies geschah bis zum Aufkommen von Mobilgeräten mit ihren zu kleinen Bildschirmen praktisch ausschließlich durch **Fenster** (die aufgrund ihrer Dominanz bei der Wahrnehmung eines Betriebssystems sogar namensgebend für ein solches sind). Dabei kann ein Fenster als ein zu einem Programm gehöriger, logischer Bildschirm betrachtet werden, der einen in Größe und Position meist veränderlichen Teil des physikalischen Bildschirms einnimmt und der andere Fenster überlappen oder sogar ganz überdecken kann. Die Tastatur als Eingabegerät ist dabei immer dem Fenster (richtiger: dem Programm, das in dem Fenster läuft) zugeordnet, das den *Fokus* hat, wobei der Fokus einem Fenster vom Bediener des Computers in der Regel über Tastatur und Maus gegeben wird. Dieser Fenstermetapher der Computerbenutzung wird aktuell aber durch eine Art virtueller Bildschirme Konkurrenz gemacht, wobei immer nur der Bildschirm des Programms, das gerade den Fokus hat, angezeigt wird, der dafür aber den ganzen (physikalischen) Bildschirm für sich alleine hat.

12.7 Klicki-Bunti

grafische Benutzungsschnittstelle

Mit dem Aufkommen von Fenstern ging auch der Übergang von zeilen- und zeichenbasierter Bildschirmausgabe (mit festen, durch die Hardware vorgegebenen *Zeichensätzen* und *Glyphen*) zu pixelbasierten grafischen Anzeigen einher. Damit wurden auch der Gestaltung der Benutzungsschnittstelle von Programmen ganz neue Möglichkeiten eröffnet. Um die sog. **grafische Benutzungsschnittstelle** (engl. **Graphical User Interface**, **GUI**) zu vereinheitlichen und ihre Programmierung zu vereinfachen, wurden ihre Elemente (Buttons, Eingabefelder etc.) zu Teilen des Betriebssystems, das damit auch das Verteilen von Ereignissen (wie Tastendrücke und Mausklicks) auf die einzelnen Bestandteile des GUI eines Programms übernahm. Damit wurden die GUI-Elemente aber selbst Betriebssystemressourcen, d. h., zu einem Gut, dessen Verfügbarkeit und Nutzung gewissen Einschränkungen unterliegt. Insbesondere ergibt sich das Problem, dass Programme die von ihren GUIs verwendeten Elemente spätestens mit Beendigung wieder freigeben müssen, genauso, wie sie es mit Hauptspeicher (Abschnitt 12.4) und File handles (Abschnitt 12.5) tun sollten. Das gelingt jedoch nicht in allen Betriebssystemen gleich gut, so dass das eine oder andere Betriebssystem in regelmäßigen Abständen einen Neustart verlangt, weil ihm die Ressourcen ausgegangen sind. Betriebssysteme können dafür über den aktuellen Ressourcenverbrauch Auskunft geben — so zeigt mein Rechner aktuell

Verwendung	Geschwindigkeit	
9%	0,53 GHz	
Prozesse	Threads	Handles
211	2438	106417
Betriebszeit		
4:23:42:10		

(und ich plane, nach nur knapp fünf Tagen Nutzung, schon mal einen Neustart ein).

Kundenbindung durch Oberflächliches

Die starke Verbreitung von GUIs als Quasi-Standards der Interaktion mit Programmen hat leider auch zu einem etwas oberflächlichen Wettbewerb der Betriebssystemhersteller um die Gunst der Käufer geführt, der eine starke Bindung von *Anwendungsprogrammen* an Betriebssysteme zur Folge hat: Da sich mit den unterschiedlichen Oberflächen auch das *Application Programming Interface* (API) der GUIs der verschiedenen Betriebssysteme erheblich voneinander unterscheidet, lassen sich Anwendungsprogramme, die das API nutzen, nur mit viel Aufwand von einem Betriebssystem auf ein anderes portieren. Da viele Softwarehersteller diesen Aufwand scheuen, müssen für das gleiche Anwendungsproblem auf verschiedenen Computern u. U. verschiedene Anwendungsprogramme eingesetzt werden und, was schwerer wiegt, die Verfügbarkeit eines benötigten Anwendungsprogramms für nur ein Betriebssystem führt zu einer Bindung, die gegen andere Bindungen (darunter auch solche, die die Sicherheit eines Systems betreffen) abgewogen werden muss. Wenn man bedenkt, zu welcher explosionsartigen Entwicklung die Übernahme der von IBM eingeführten PC-Architektur durch andere Hersteller auf dem Gebiet der Hardware geführt hat, scheint diese ihre Oberfläche betreffende, auf Abgrenzung ausgerichtete Konkurrenz der Betriebssysteme für die Entwicklung von Software eher kontraproduktiv.

12.8 Herunterfahren

So wie einzelne Programme ordnungsgemäß beendet werden sollten (damit sie alle begonnen Vorgänge auch abschließen und ihre Ressourcen zurückgeben), soll auch ein Betriebssystem beendet werden, bevor der Computer ausgeschaltet wird. Dieses Beenden, das selbst ein recht komplexer Vorgang ist, nennt man **Herunterfahren**. Das Betriebssystem sendet dabei zunächst Aufforderungen an alle noch laufenden Programme, sich zu beenden; wenn sie dies nicht innerhalb einer vorgesehenen Zeitspanne tun, bricht es sie ab. Danach speichert es seinen Zustand, sofern dieser für den Neustart relevant ist, und beendet alle seine eigenen noch laufenden Prozesse. Am Ende zeigt es eine Nachricht an, dass der Computer jetzt ausgeschaltet werden kann (mit einer primitiven Ausgabefunktion, denn alle höheren Funktionen sollten zu diesem Zeitpunkt schon nicht mehr zur Verfügung stehen), oder aber, wenn es das kann, es schaltet ihn selbst ab.

13 Internet

Das **Internet** ermöglicht einen weltweiten Datenaustausch zwischen den damit verbundenen Computern. Es besteht aus einer technischen Infrastruktur (Signalleitungen, die auch kabellos ausgeführt sein können) sowie einer in Schichten geordneten Menge von *Kommunikationsprotokollen*[44], die zum Teil in Hardware (die Protokolle der unteren Schichten), zum Teil in Software (die der oberen Schichten, von denen wiederum die unteren Teil des Betriebssystems sein können) umgesetzt sind.

Internetprotokolle

Auf der obersten Schicht, der sog. *Anwendungsschicht*, finden sich die Protokolle, für die man das Internet vor allem kennt: *Email* (aufgeteilt in *Simple Mail Transfer Protocol*, *SMTP*, für das Versenden sowie *Post Office Protocol*, *POP*, und *Internet Message Access Protocol*, *IMAP*, für das Abrufen von Mails), *File Transfer Protocol* (*FTP*), *Hypertext Transfer Protocol* (*HTTP*) und noch einige mehr. Eine Schicht darunter finden sich Protokolle für den *Datentransport* (*Transmission Control Protocol*, *TCP*, oder *User Datagram Protocol*, *UDP*) und darunter wiederum das eigentliche Internet mit dem *Internet Protocol* (*IP*), das die Verbindung zwischen den Computern per *Vermittlung* herstellt. Dabei ist der wesentliche Unterschied zum Telefonieren (bei dem ja auch zwischen den Teilnehmern vermittelt werden muss) der, dass die Datenströme in Pakete aufgeteilt werden, die einzeln vermittelt werden, und dass die Vermittlung — anhand sog. *IP-Adressen* anstelle von Telefonnummern — weitgehend dezentral erfolgt. Der Legende nach wollte das amerikanische Verteidigungsministerium, ein Finanzier des frühen Internets, damit eine hohe Ausfallsicherheit bei Atomangriffen erzielen; sicher ist hingegen, dass dadurch die Internetverbindung zwischen Rechnern hochflexibel ist, sich also die Netzstruktur (Menge der verbundenen Rechner und der verfügbaren Leitungen) ohne Aufwand jederzeit ändern lässt. Darunter folgen dann die Protokolle für die physikalische *Verbindung*, also den Datenaustausch beispielsweise über ein *Wide Area Network* (*WAN*) oder ein *Local Area Network* (*LAN*, wozu auch ein *Wireless LAN*, *WLAN*, zählt).

IP-Adressen

Zur Kommunikation über das Internet bekommt jeder teilnehmende Rechner eine Adresse, die sog. **IP-Adresse**, zugeordnet. Die IP-Adresse im Internet Protocol Version 4 (IPv4) ist 32 Bit lang (und wird häufig als Quartett von Dezimalzahlen, von denen jede ein Byte der Adresse repräsentiert, dargestellt, also etwa als 132.176.114.181 für die IP-Adresse des Webservers der Fernuni). Damit lassen sich allerdings nur ca. 4 Milliarden Rechner mit einer global eindeutigen Adresse versehen. Im IPv6 stehen daher 128 Bit für IP-Adressen zur Verfügung.

Ports

Neben der IP-Adresse, die die Rechner identifiziert, regelt die sog. **Portnummer** (mit 16 Bit) die Lenkung der Datenströme zwischen zwei Teilnehmern. So werden

---

[44] In der Informatik bezeichnet man mit **Protokoll** einen standardisierten Ablauf eines wechselseitigen Datenaustauschs. Ein Protokoll gibt mit seinen Regeln Reihenfolge, Form und Bedeutung der Nachrichten des Austauschs vor.

über verschiedene Portnummern zum einen verschiedene gleichzeitige Verbindungen zwischen denselben Teilnehmern unterschieden und zum anderen bestimmte Dienste an bestimmte, feststehende Portnummern gebunden. So warten Webserver beispielsweise standardmäßig auf Port 80 auf Anfragen. Diese feste Bindung von Portnummern an bestimmte Dienste bzw. Protokolle ist auch Grundlage der Drosselung bzw. Unterbindung des Datenverkehrs durch sog. *Firewalls*. Logisch kann man Portnummern auch als Kanalnummern (und die mit den Portnummern versehenen Datenströme als Kanäle, entsprechend etwa denen beim Fernsehen) begreifen.

13.1 Rechnerkommunikation über Internet

Über das Internet miteinander verbundene Rechner können miteinander kommunizieren. Während Internetknotenrechner dies im Dienste anderer tun, nutzen andere Rechner das Internet beispielsweise für den Austausch von Dateien (über *FTP*) oder zum entfernten Programm- oder Prozeduraufruf (engl. *Remote Procedure Call*, *RPC*). Besondere Bedeutung haben dabei die sog. *Client/Server-Systeme*, bei denen mehrere verteilte Rechner (die *Clients*) an einem Zentralrechner (dem *Server*) hängen, auf den sie gemeinsam zugreifen, z. B. um eine zentrale *Datenbank* (Kapitel 30) zu nutzen.

Mittlerweile hat das Internet auch als Medium für die Verteilung von *Software* (und deren Updates) Datenträgern wie CDs oder DVDs den Rang abgelaufen (mit dem Ergebnis, dass die Auslieferung fehlerbehafteter Software für den Hersteller günstiger wird — man kann ja jederzeit schnell nachbessern). Aber auch die Anbindung von *Terminals* (*Konsolen* oder *Terminalemulationen* über das *Teletype Network*, *Telnet*, bis hin zu sog. *Remote Desktops*) an mehr oder weniger entfernt stehende Rechner läuft über das Internet. Schließlich hängt die Virtualisierung von Speicher- und Rechenkapazitäten durch die sog. *Cloud* (Kapitel 14) am Internet.

HTTP als Universalprotokoll

Auch wenn für die Rechnerkommunikation eine ganze Reihe von dedizierten Protokollen zur Verfügung stehen (s. o.), verwenden viele aktuelle Formen dafür das *Hypertext Transfer Protocol* (*HTTP*); s. Abschnitt 13.3), selbst wenn es eigentlich gar nicht passt. Der Grund hierfür ist häufig ein ganz simpler: Praktisch alle Knotenrechner, die vor unbefugter Kommunikation schützen sollen, indem sie Kanäle sperren, sind so konfiguriert, dass sie HTTP (an Port 80; s. o.) durchlassen. Leider fehlen dem HTTP bestimmte Eigenschaften (es ist z. B. *zustandslos*, d. h., es unterstützt keine Sitzungen mit *Authentifizierung* wie etwa *Telnet* oder *FTP*), so dass zur Erreichung dieser Eigenschaften fernab der Standardisierung des Internets improvisiert wird. Praktisch alle diese Improvisationen bringen jedoch Sicherheitsprobleme mit sich.

13.2 Personenkommunikation über Internet

Es ist eines der Phänomene des Internets, dass es, als Rechnernetz entworfen, schon immer wesentlich als Medium der Kommunikation zwischen Menschen rezipiert wurde. Zwei klassische Protokolle der Personenkommunikation über das Internet sind *Email* und *Newsgroups* (das *Network News Transfer Protocol*, *NNTP*). Heute ist zudem die *IP-Telefonie* (mit ihrem *Session Initiation Protocol*, *SIP*) nicht nur weit verbreitet, sondern im Begriff, die klassische Telefonie abzulösen. Während Email und Newsgroups für den Nachrichtenaustausch einen Server benötigen, den sich Privatpersonen normalerweise nicht selbst leisten, ist die Internettelefonie grundsätzlich anbieterunabhängig; lediglich für die Vermittlung von Anrufen wird die Anmeldung an einem SIP-Server benötigt. Ähnliches gilt für standardisierte Chat-Protokolle wie das *Internet Relay Chat* (*IRC*).

proprietäre Protokolle

Während alle obengenannten Formen der Personenkommunikation über die Umsetzung offener Standards realisiert werden und ihre Nutzung den Charakter eines *Gemeingebrauchs* hat (entsprechend beispielsweise der Nutzung öffentlicher Straßen), kann man dasselbe für konkurrierende Angebote privater Anbieter (wie aktuell etwa Facebook, Twitter, WhatsApp oder Skype) mit ihren *proprietären Protokollen* nicht sagen. Auch wenn alle diese Dienste auf dem Internet basieren (und so die öffentlich zugängliche Infrastruktur für ihre privatwirtschaftlichen Zwecke nutzen), so satteln sie doch ihre eigenen Protokolle obendrauf, so dass die Kommunikation über einen solchen Dienst in den Händen (und unter Kontrolle) des Dienstanbieters bleibt und ein Anbieterwechsel immer disruptiv ist (der sog. Lock-in-Effekt).[45] Damit ändern kommerzielle Kommunikationsdienste den öffentlichen Charakter des Internets erheblich. Insofern ist es nachvollziehbar, wenn Politiker versuchen, die Kommunikationsplattformen privater Anbieter als öffentlichen Raum zu deklarieren, um so zumindest der darüber stattfindenden Kommunikation Herr zu werden, doch scheint dies aussichtslos, da schon das Internet (als Basis dieser Dienste), obwohl um einiges öffentlicher, kein öffentlicher Raum ist (s. dazu auch Abschnitt 13.5).

13.3 Das Web

Der Begriff des **World Wide Web** (**WWW**, oder kurz **Web**), der im Wesentlichen für die barrierearme Abrufbarkeit von miteinander verknüpften Dokumenten im Internet steht, wird heute in der Öffentlichkeit oft als synonym für das Internet selbst verwendet. Tatsächlich war das Web Motor für die massive Verbreitung des Internets, insbesondere im privaten Bereich. Zugleich ist es aber auch immer mehr zu einem Wirtschafts- und Industriestandard

---

[45] Diese Kommunikationsdienste unterliegen übrigens nicht wie etwa Telefonie oder SMS der Regulierung durch das *Telekommunikationsgesetz*. Insofern ist es ein Unding, wenn öffentlich-rechtliche Anstalten dazu auffordern, mit ihnen über private Dienste in Kontakt zu treten, schon allein deswegen, weil diese Kommunikation nicht dem *Telekommunikationsgeheimnis* unterliegt.

geworden, was aus Informatiksicht ein Problem darstellt, weil sich die Anforderungen von Wirtschaft und Privatpersonen an das Web zum Teil erheblich voneinander unterscheiden.

Die technische Infrastruktur des Web besteht im Wesentlichen aus vier Komponenten:

- den **Webservern**, die die Dokumente halten und sie auf Abruf zur Verfügung stellen,
- den **Webbrowsern** (heute verkürzt häufig nur noch *Browser* genannt, obwohl es auch andere Browser gibt, z. B. Filebrowser), die zum Abrufen und Anzeigen der Dokumente benötigt werden,
- dem **Hypertext Transfer Protocol** (**HTTP**), das für die Übertragung der Dokumentanfragen von Browsern zu Servern sowie umgekehrt die Übertragung der Dokumente zu den Browsern steht, und
- der **Hypertext Markup Language** (**HTML**), die der Auszeichnung von Text als Hypertext (sowie der Einbettung anderer Inhalte in einen Text) dient.

Dabei ist ein **Hypertext** ein Text, dessen Elemente mit Verknüpfungen zu anderen Texten, den sog. **Hyperlinks** oder kurz **Links**, verbunden sein können, die ohne viel Aufwand verfolgt werden können. Man spricht bei der Verfolgung solcher Links auch von **Navigation**.

ungesteuerte Fortentwicklung des World Wide Web

Bestanden die über das Web übertragenen Inhalte anfangs nur aus Texten, Grafiken und einfachen Formularen (wobei mit letzteren bereits Inhalte vom Browser zum Server übertragen werden konnten), so kamen schon bald sog. *ausführbarer Inhalt* (*Java Applets*) sowie Audio und Video hinzu. Mit der weiteren Verwendung von Webservern und -browsern für das Betreiben von *Anwendungsprogrammen* (s. Kapitel 23) wurden nicht nur immer mehr Informationen vom Browser (bzw. seinem Benutzer) zum Server übertragen, sondern waren auch immer mehr Elemente einer *grafischen Bedienoberfläche* (*GUI*; s. Abschnitt 12.7) als vom Server zum Browser zu übertragender Inhalt gefragt, was aufgrund des dafür nicht gedachten, aber verwendeten HTTP mittlerweile zu einem kaum noch überschaubaren Wildwuchs an Quasi-Standards geführt hat, der zu allem Übel, getrieben von einer skurrilen Mischung aus Industrie und Privatpersonen, auch noch weitgehend unkontrolliert ständig weiter wächst, ohne dass man das Ergebnis als gut bezeichnen könnte. Man kann wohl sagen, dass Wissenschaft praktisch keinen Einfluss (oder falls doch, dann jedenfalls keinen ausreichenden) auf die Entwicklung der Standards des World Wide Web hat, was ich angesichts seiner mittlerweile erreichten Bedeutung für die Gesellschaft für bedenklich halte.[46] Letztlich ist jedoch das Grundproblem, dass das Web praktisch uneingeschränkt für Dinge verwendet wird, für die es nicht konzipiert wurde — ganz so, als würde man Autos und Schiffe auf Eisenbahnschienen fahren lassen. Es wäre längst an der Zeit, für die heutige Nutzung des Web dedizierte *Protokolle*

---

[46] Das führt unmittelbar zur Frage, wie so im Kontext des World Wide Web der *Stand von Wissenschaft und Technik* zur Anwendung kommen soll, wie er beispielsweise in Schadensersatzprozessen herangezogen werden müsste. S. dazu auch Kapitel 26 in Kurseinheit 3.

zu vereinbaren und zu verwenden. Das Web von heute würde dann das, was es im Grunde längst ist: eine *weltweite Altlast*.

13.4 Der Webbrowser als Betriebssystem

Mit der Ausweitung des Webs zu einer allgemeinen Infrastruktur für die Benutzung von Programmen, die nicht auf einem Endgerät (PC oder Mobilgerät) installiert sind, sondern auf zentralen Servern laufen, und der zunehmenden Verschiebung von Programmen und Daten in die Cloud (s. Kapitel 14) stellt sich die Frage, ob man überhaupt noch ein eigenständiges Betriebssystem für die Ausführung von Programmen braucht oder ob der Webbrowser nicht das einzige benötigte Programm auf einem Endgerät ist. Diese Frage hat beispielsweise Google mit seinem Chromebook positiv beantwortet.

Sandboxing ade

Damit wird aber die Kontrolle über die Hardware eines Computers vollständig an einen Webbrowser abgegeben. Was für manche Horrorszenarien sind (wie z. B. die Möglichkeit, dass Webseiten Dateien auf einem Computer speichern oder die die Kamera oder das Mikrofon eines Computers einschalten) wird damit zur Voraussetzung: Wenn alles in einem Browser läuft, dann muss der Browser Zugriff auf die Hardware eines Computers haben, damit sie für *Anwendungsprogramme* (s. Kapitel 23 in Kurseinheit 3) genutzt werden kann. Die Idee des sog. *Sandboxing*, mit dem Browser sicherer gemacht werden sollen (richtiger: das Computer vor unerwünschten, durch Webseiten verursachten Aktivitäten schützen soll), wird damit konterkariert.

Link

Konfigurationshölle

Ein weiteres Problem von Browsern als Betriebssysteme resultiert aus der Vielzahl der verschiedenen Browser und der Notwendigkeit von Browser-Erweiterungen (Plugins genannt), die für bestimmte Dienste benötigt werden: Funktioniert das Zusammenspiel in irgendeiner Konstellation einmal nicht, ist guter Rat meist teuer. Dauerhaft zuverlässig funktionierende Systeme, deren aktuelle Konfiguration man nötigenfalls sogar einfrieren kann (u. a. durch Verweigerung von Updates), scheinen damit in weite Ferne gerückt.

13.5 Das Internet als rechtsarmer Raum

grenzenlose Freiheit von Meinungen

Als die Freedom of Speech 1791 Eingang in die amerikanische Bill of Rights fand, musste man sich noch auf einen öffentlichen Platz stellen und laut rufen, um seine Meinung vor Publikum kundzutun (von schriftlichen Verlautbarungen einmal abgesehen, aber eine Vervielfältigung der Schrift war damals mit erheblichem Aufwand verbunden und längst nicht jedem zugänglich). Mit dem Aufkommen des Sprechfunks Anfang des 20. Jahrhunderts wurde es dann möglich, die persönliche Meinung weiter zu verbreiten, aber es dauerte nicht lange, bis das Funken vom Rundfunk (dem Radio und später dem Fernsehen) verdrängt wurde. Radio und Fernsehen wurden Teil der öffentlichen Infrastruktur, wobei der Konsum von Inhalten allen zugedacht, deren Produktion jedoch vergleichsweise wenigen vorbehalten war (die sog. offenen Kanäle, oder Public Access Channels in den USA, sind in ihrer Ausbreitung lokal begrenzt). Gleichzeitig entwickelten

sich gesellschaftliche Normen, die dafür sorgten, dass jemand Verantwortung für das Gesendete übernehmen musste. Auch wenn die freie Meinungsäußerung dadurch nicht eingeschränkt wurde, hat doch nicht jeder einen Anspruch darauf, dass seine Meinung über Kanäle, die von anderen betrieben werden, verbreitet wird (und das Betreiben eigener Kanäle unterliegt einer Lizensierung). Dies scheint beim Internet anders zu sein.

Geschäfte mit nicht Geschäftsfähigen ganz ungeniert

Heute verbreiten private Internetunternehmen unter dem Label „soziale Medien" Meinungen, für die sie keine Verantwortung übernehmen wollen, bis in den hintersten Winkel der Erde. Damit sich das für sie lohnt, gehen sie Geschäftsbeziehungen selbst mit Minderjährigen ein, ohne dass Erziehungsberechtigte dem effektiv widersprechen könnten.[47] Die Minderjährigen werden damit Inhalten ausgesetzt und somit beeinflusst (Wünsche werden geweckt, Normen werden gesetzt), ohne dass ein allgemeiner Jugendschutz greifen würde (von individuellen, Erziehungsberechtigten zuzugestehenden Schützbedürfnissen einmal ganz abgesehen). Als wäre das nicht genug haben Erziehungsberechtigte auch keinerlei Kontrolle darüber, wie die gesammelten Daten ihrer Schutzbefohlenen verwendet werden — auch Minderjährige werden so zur Ware von Unternehmen, deren Wert vor allem im Besitz persönlicher Daten besteht.

fehlende Güterabwägung

In der aktuellen Aufregung um das Internet (das für viele Menschen mit der Digitalisierung gleichbedeutend ist) und seine Bedeutung für die wirtschaftliche Zukunft ist nicht erkennbar, wie die freie Meinungsäußerung, die das Internet in nie dagewesenem Ausmaß fördert, gegen andere Normen abgewogen werden soll: „Anything goes" ist das Motto — jeder Versuch, das einzuschränken, versinkt im Geschrei von Zensur derer, die diese Freiheit weiter unter allen Umständen auskosten wollen. Dabei ist Anonymität bei der Meinungsäußerung zumindest in Deutschland kein Grundrecht. Aber wo es keine Landesgrenzen gibt, gibt es auch keinen Rechtsraum. Wenn aber Normen nicht weltweit durchgesetzt werden können oder sollen, dann muss man ein „nationales Internet" als Alternative zumindest in Erwägung ziehen.[48]

automatisierte Meinungsmache

Interessanterweise wird die Diskussion aktuell (Herbst 2017) in einem anderen Zusammenhang ganz neu geführt, nachdem nämlich der Verdacht aufgekommen ist, ausländische Geheimdienste hätten sich, u. U. sogar unter Nutzung sog.

---

[47] Selbst wenn Erziehungsberechtigte formal widersprechen können, geht dieser Widerspruch ins Leere, wenn sie die Anmeldedaten ihrer Schutzbefohlenen nicht kennen — deren wahre Identität bei der Anmeldung hätte schließlich allenfalls durch eine Behörde oder durch einen persönlichen Besuch von Mitarbeitern des Unternehmens bei einem gesetzlichen Vertreter festgestellt werden können. Von entsprechenden Anstrengungen der Unternehmen ist mir zumindest nichts bekannt.

[48] Tatsächlich ist auch schon die weltweite Verbreitung von Fernsehsendungen und ihre normierende Wirkung für viele Kulturen ein Problem. So beklagte sich ein australischer Kollege vor einiger Zeit mir gegenüber, dass seine Kinder durch das Fernsehen zunehmend amerikanischer würden. Wie mag es da erst Kulturen ergehen, die einen nicht-westlichen Lebensstil pflegen möchten? Und wie kann eine Kultur sich gegen eine solche „externe Normierung" wehren, wenn die (noch einigermaßen kontrollierbare) Verbreitung „unserer" (der westlichen) „Art zu leben" über Sendeanstalten durch die weitgehend unkontrollierbare Verbreitung durch das Internet ersetzt wird?

Chatbots, der freien Meinungsäußerung bedient, um die Präsidentschaftswahl in den USA zu beeinflussen. Abgesehen davon, dass hier Nationalitäten und Landesgrenzen auf einmal doch eine Rolle spielen sollen, wird die interessante Frage aufgeworfen, ob ein Chatbot eine Meinung vortragen kann, deren Verbreitung unter die freie Meinungsäußerung fällt. Vielleicht wird im Verlauf der Diskussion dieser Frage auch juristisch geklärt werden, ob ein Chatbot selbst eine Meinung hat oder wem sie stattdessen zugeschrieben werden muss oder ob Meinungen auch unabhängig von einer sie äußernden Entität solche sind.

Mittlerweile, im Mai 2018, ist eine Datenschutzgrundverordnung in der EU inkraftgetreten; einige der oben beklagten Missstände sind damit hoffentlich abgestellt. Bis dahin war allerdings schon eine Generation der grenzenlosen Freiheit von Meinungen und ihren kommerziellen Blüten ausgesetzt. Es bleibt die Frage, ob man nicht mit geltenden Gesetzen schon viel früher etwas hätte machen können, oder ob die Sorge schwerer wog, etwas abzuwürgen, das man nicht genau versteht (nämlich dieses Internet), und so im weltweiten Wettbewerb um die besten Startplätze für die Zukunft ins Hintertreffen zu geraten.

14 Cloud

Jeder, der digital fotografiert oder filmt, stellt sich irgendwann einmal die Frage, wie er seine Bilder für die Nachwelt aufhebt. Digitale Langzeitspeichermedien wie CDs, DVDs oder Blu-ray-Disks sind zwar nicht teuer, müssen sich in Sachen Zuverlässigkeit aber erst noch beweisen und wer weiß schon, ob es die dazugehörigen Lesegeräte in 30 Jahren noch gibt. Eine mögliche Lösung ist, die Daten einem Unternehmen zur Aufbewahrung zu überlassen. Da man nicht genau weiß, wo das Unternehmen die Daten speichert und wie, spricht man hier passenderweise vom „Speichern in der Cloud". Das Wesensmerkmal der Cloud ist nämlich das Diffuse: Man weiß nur, dass sie da ist und wer sie betreibt — der Rest muss einen nicht kümmern.

Dieser populäre Begriff der Cloud (mit dem auch das Abrufen von digitalen Inhalten, die andere produziert haben, über sog. Streaming-Dienste verbunden ist) leitet sich vom weiter gefassten Begriff des **Cloud computing** ab, also dem Rechnen (und nicht nur dem Speichern) an einem nicht genau bekannten Ort. Über die Cloud wird heute allgemein die Nutzung von Hardware, Rechen- und Speicherleistung oder auch nur einzelner Programme angeboten, wobei sich die Kosten in der Regel an der tatsächlichen Nutzung der Dienste und nicht an der Bereitstellung orientieren. Das macht die Cloud-Nutzung für so manches Unternehmen attraktiv.

WIKIPEDIA

Das Betreiben eigener Computer oder gar Rechenzentren ist nämlich für Unternehmen mit nicht zu unterschätzenden Risiken verbunden, zum einen, weil immer mehr Geschäftsprozesse digitalisiert werden und somit die Abhängigkeit von unterbrechungsfrei funktionierenden Computersystemen steigt, zum anderen, weil kurze Hard- und Softwareinnovationszyklen nicht nur die Kosten für Betrieb, Instandhaltung und Wartung in die Höhe treiben,

sondern auch noch ständige Weiterqualifikation des Personals erfordern. Immer mehr Betriebe lagern daher Prozesse an externe Rechenzentren, sog. IT-Dienstleister, aus. Mit der zunehmenden Verfügbarkeit schneller Datenleitungen (Internet!) gibt es technisch tatsächlich kaum noch Gründe, eigene Computer für andere Zwecke als die des klassischen *Terminals* (zur Ein- und Ausgabe) zu betreiben. Hinzu kommt, dass große Internetunternehmen über riesige Speicher- und Rechenkapazitäten verfügen, die für Spitzenlasten (wie beispielsweise im Versandhandel die Weihnachtszeit) ausgelegt sind, ansonsten aber weitgehend brachliegen, also günstig angeboten werden können.

Wenn man beides zusammentut, dann liegt es nahe, dass die großen Internetunternehmen ihre freien Kapazitäten an andere Unternehmen verkaufen. Dazu werden Computer „virtualisiert", d. h., man mietet scheinbar einen Computer eines bestimmten Typs, bekommt aber in Wirklichkeit nur Rechenzeit auf einem oder mehreren Computern (u. U. eines anderen Typs), die dem Nutzer (per entsprechender Software, einer *virtuellen Maschine*) einen Computer vorspielen. Für den Nutzer ist das freilich egal, denn das gemietete Produkt verhält sich funktional (d. h. hier abgesehen von der Rechengeschwindigkeit, die stark variieren kann) genauso wie ein realer Computer, der bei ihm auf dem Schreibtisch oder im Keller steht. Die rechtlichen Probleme, die sich aus der Übertragung der Datenverarbeitung auf andere ergeben, insbesondere, wenn diese in anderen Ländern oder gar auf anderen Kontinenten (anderen Rechtsräumen) angesiedelt sind, liegen auf der Hand.

15 Sicherheit

Unter **Sicherheit** versteht man i. Allg. das Fehlen von Risiko. Ein Flugzeug etwa gilt als sicher, wenn seine technische Beschaffenheit so ist, dass es nicht abstürzt; eine Bank gilt als sicher, wenn die ihr zur Aufbewahrung überlassenen Werte nicht abhandenkommen. Bei Computern unterscheidet man technisch zwischen zwei Arten von Sicherheit: dem Ausschluss von Fehlfunktionen und dem Ausschluss unerlaubten Zugriffs (auf den Computer oder die Daten, die er verarbeitet). Im Englischen nennt man erstes **Safety** und zweites **Security**; im Deutschen gibt es diese sprachliche Unterscheidung nicht, man denkt aber bei Sicherheit eher an Security (und sagt auch **Computersicherheit** dazu). Praktisch alle Computer sind aber auf beide Arten unsicher und beide Arten teilen sich Quellen der Unsicherheit (was die fehlende sprachliche Unterscheidung im Deutschen rechtfertigt).

Kurs

Im Zusammenhang mit der zunehmenden Verbreitung und Abhängigkeit vom Internet wird der Begriff der Computersicherheit mit verschiedenen Schutzzielen verbunden:

- *Vertraulichkeit*, also dass Information nur denen zugänglich ist, für die sie bestimmt ist;
- *Integrität*, also dass Inhalte nicht unautorisiert abgeändert werden können;
- *Authentizität*, also dass die Quelle von Informationen gesichert ist; und

- *Verfügbarkeit*, also dass ein angebotener Dienst nicht von Dritten blockiert werden kann.

Bezeichnenderweise werden alle vier Schutzziele im Internet, so wie es heute hauptsächlich genutzt wird, nicht erreicht. So ist beispielsweise bei nicht digital signierter, über SMTP verschickter Email nicht gesichert, dass sie von der Person oder Einrichtung stammt, die im Absenderfeld angegeben ist (SMTP verlangt keine *Authentifizierung* für den Email-Versand).

15.1 Quellen der Unsicherheit

Eine klassische Quelle von Unsicherheit ist die mangelnde Absicherung von Eingaben: So können unsinnige oder inkonsistente Eingaben zu unsinnigem oder gar fehlerhaftem Verhalten führen, bis hin zu *Programmabstürzen*. Genauso gut können sie aber die Datensicherheit aushebeln: Sog. *Code injection* schiebt einem Eingaben verarbeitenden Programm Schadcode unter, indem es ausnutzt, dass die Eingaben als (Teile von) *Befehlen* interpretiert werden und die Befehle nicht umfassend genug auf Zulässigkeit geprüft werden.

Eine andere Quelle von Unsicherheit ergibt sich aus der Universalität des Binären: Wie in Kurseinheit 1 gezeigt werden mit Einsen und Nullen die verschiedenartigsten Daten repräsentiert. Wie in Kapitel 10 bereits diskutiert bedeutet dies allerdings auch, dass man Daten dadurch beliebig um- oder fehlinterpretieren kann. Das gilt genau so auch für Computer: Zunächst hindert einen Computer nämlich nichts daran, ein Byte, das eigentlich eine Zahl oder ein Zeichen eines Textes darstellen soll, als Befehl seiner *Maschinensprache* zu interpretieren. Wenn dies absichtlich passiert, dann mag das zweckmäßig sein (wobei der gute vom bösen Zweck hier kaum zu unterscheiden ist); wenn dies aus Versehen passiert, dann wegen eines Fehlers, der, wenn er hätte entdeckt werden können, auch hätte entdeckt werden sollen.

Gefahr erkannt, Gefahr gebannt, so könnte man meinen, denn man könnte ja für jedes Byte irgendwo notieren, für was es steht: für eine Zahl, ein Zeichen, oder was auch immer. Vor seiner Verwendung für einen bestimmten Zweck könnte man prüfen, ob sein Inhalt dem Zweck entspricht und andernfalls warnen. Man könnte nicht nur, man kann sogar, aber macht es in aller Regel dann doch nicht, weil dieses Notieren und Prüfen auf Hardwareebene erhebliche Ressourcen verschlingen würde, worunter auch die fehlerlose Verarbeitung von Daten zu leiden hätte: Computer würden langsamer und teurer. Anders als beispielsweise beim Automobilbau ist aber die Sicherheit von Computerhardware für den Gesetzgeber kaum ein Thema und so nimmt man es als gegeben hin, dass Hardware immer schneller und billiger wird, dabei aber unsicher bleibt.

Auf Softwareebene sieht die Sache etwas anders aus, wie wir noch sehen werden: Hier können Daten *typisiert* und somit bestimmten Verwendungszwecken vorbehalten werden, ohne dass sich das negativ auf die Laufzeit oder den Speicherverbrauch eines Programms auswirkt (Kurseinheit 3, Abschnitt 20.4). Allerdings gibt es auch hier keinen Konsens, wie

viel Sicherheit es denn sein darf, denn die Unterscheidung von Daten anhand ihrer Bedeutung kostet bei der Programmierung Zeit und Flexibilität. Solange derjenige das Geschäft macht, der am schnellsten liefert, hat es die Sicherheit schwer.

Seit der finanzielle Druck durch Schadsoftware aber zugenommen hat, denkt man auch auf Hardwareebene vermehrt darüber nach, wie man bestimmte Sicherheitslücken schließen kann. So erfordert die Ausführung bestimmter Teile eines Betriebssystems schon seit geraumer Zeit einen speziellen Modus, der verhindern soll, dass andere Programmteile Betriebssystemfunktionen übernehmen können (den *privilegierten Modus*, Abschnitt 12.1), und moderne Prozessoren erweitern ihren Befehlssatz um eine Instruktion, deren einzige Aufgabe es ist, als Sprungziel zu dienen — alle Sprünge auf andere Instruktionen sind dann ungültig und bedingen einen sofortigen *Programmabbruch*. Trotz solcher Maßnahmen bleibt die Gefahrenabwehr aber zu lückenhaft.

15.2 Ungenügender staatlicher Druck

Es ist nicht so, dass man der Unsicherheit von Computern hilflos gegenüberstünde — auch wenn man längst nicht für alle Probleme eine Lösung hat (und es für manche auch keine geben mag), so hinkt die Praxis bekannten Sicherheitsstandards weit hinterher. Das ist insofern keine Kleinigkeit, als weite Teile unserer Infrastruktur von der Sicherheit von Computersystemen abhängen.

Eine Ursache für diesen Rückstand ist, dass Sicherheit ihren Preis hat: Sichere Computer sind langsamer oder teurer und da Sicherheit ein Merkmal ist, das nicht unmittelbar in Erscheinung tritt, wenn es fehlt, zieht es gegenüber höherem Preis oder geringerer Geschwindigkeit (die sofort in Erscheinung treten) zu häufig den Kürzeren.

Eine andere Ursache ist, dass die Disziplinen, die den Problemen wirksam begegnen könnten, Mangeldisziplinen sind: Es gibt einfach nicht genügend viele Fachleute, die wissen, welche Maßnahmen zu ergreifen sind, um Computersysteme sicherer zu machen, oder wie solche Maßnahmen im Einzelfall umgesetzt werden können.

Eine dritte Ursache ist Nachlässigkeit: Selbst da, wo umfangreiche Standards existieren und wo allseits bekannt ist, wie diese umgesetzt werden können (MISRA-C in der Automobilindustrie etwa), hält man sich nicht an diese, weil man es nicht muss (zumindest solange nicht, wie alles gutgeht).

All diese Gründe sind nachvollziehbar, jedoch nicht zwingend. Da nicht zu erwarten ist, dass Industrie und Wirtschaft freiwillig die Probleme aus der Welt schaffen (man kann ja sogar an ihnen verdienen), wäre es Sache des Staates, wirksame Standards zu erarbeiteten, zu verordnen und durchzusetzen.

Kurseinheit 3: Programmierung

Die Mächtigkeit der Computer liegt darin, dass sie für nahezu beliebige Aufgaben programmiert werden können. Programme (Kapitel 18) spielen deswegen eine zentrale Rolle in der Digitalisierung. Dabei sind der Programmierung aber, genau wie Computern selbst, auch Grenzen gesetzt: Abgesehen davon, dass nicht alle Probleme überhaupt lösbar sind (und zwar weder praktisch noch theoretisch), kann die Komplexität von Programmen erhebliche Ausmaße annehmen. Um das Programmieren dennoch beherrschbar zu halten, abstrahiert die Programmierung durch die Einführung immer neuer Programmiersprachen (Kapitel 20) in zunehmendem Maße von Computern und versucht so, Programme von rein technisch bedingter, d. h. nicht unmittelbar mit der Aufgabenstellung zusammenhängender, Komplexität zu befreien. Allerdings greift hier das *No-free-lunch-Theorem*, das besagt, dass dies nicht immer gelingen kann, und so wird es nie eine ideale Programmiersprache für alle Zwecke geben. Stattdessen lohnt sich ein Blick auf die unterschiedlichen Programmierparadigmen (Kapitel 19), die den einzelnen Programmiersprachen zugrunde liegen und die ihre Stärken und Schwächen ganz wesentlich mit bestimmen.

Um mit Programmiersprachen programmieren zu können bedarf es sogenannter Programmierwerkzeuge (Kapitel 21), die selbst Programme sind. Viele aktuelle Programmierwerkzeuge sind für eine bestimmte Programmiersprache bestimmt und können Programmierer somit aktiv beim Programmieren in dieser Sprache unterstützen, wodurch die Grenze zwischen Programmiersprache und Programmierwerkzeug zunehmend verschwimmt. Die allermeisten Programme gehören jedoch in eine von zwei anderen Kategorien: die der Anwendungsprogramme (Kapitel 23) oder die der eingebetteten Software (Kapitel 25).

Vor der Programmierung kommen ein Problem und seine Lösung. Je genauer das Problem spezifiziert ist, desto eher kann man sagen, ob seine Lösung tatsächlich eine ist. Die Spezifikation (Kapitel 17) steht daher am Anfang und am Ende der Programmierung — am Anfang, weil ohne Spezifikation nicht klar ist, welches Problem zu lösen ist, und am Ende, weil ohne Spezifikation nicht klar ist, ob es gelöst wurde. Es ist sinnvoll, sich vor der Lösung eines spezifizierten Problems mithilfe eines Programms über die Art der Lösung Gedanken zu machen; das ist die Stunde des Algorithmus (Kapitel 16).

16 Algorithmus

Algorithmus ist ein Begriff, der im traditionellen Informatikunterricht ganz am Anfang steht, der es aber so lange nicht in den Kanon der Allgemeinbildung schaffte, bis die Medien mit

ihren Versuchen, nicht mehr zu ignorierende Digitalisierungsphänomene zu ergründen, begannen, ihn mit ihren mystisch anmutenden Interpretationen zu deuten. Zählten so früher zu den bekanntesten Algorithmen die Sortieralgorithmen, die eine Menge von Datensätzen in eine durch ein Sortierkriterium bestimmte Reihenfolge bringen, so sind es heute eher die Matching-Algorithmen der Suchmaschinenbetreiber, die einer Suchanfrage eine gereihte Folge von dazu passenden Datensätzen zuordnen, die aber keine klassischen Vertreter der Gattung Algorithmen sind, schon weil ihre *Korrektheit* mangels scharfer *Spezifikation* (Kapitel 17; was heißt denn schon passend?) nicht klar bestimmbar ist.

Ein einfacher Sortieralgorithmus

Um zu veranschaulichen, was ein Algorithmus ist und was er leisten kann, nehmen wir ein einfaches Beispiel: das Sortieren der Karten eines Kartenspiels mit 32 Karten. Dazu nehmen wir an, dass es eine vorgegebene Sortierreihenfolge gibt, also etwa 7-8-9-10-Bube-Dame-König-As sowie übergeordnet Karo-Herz-Pik-Kreuz. Wir können also von zwei beliebigen Karten sagen, welche „höher" (oder „größer") als die andere ist. Man kann einen Stapel von 32 Karten dann wie folgt aufsteigend sortieren (dabei sollen alle Karten stets mit dem Gesicht nach oben abgelegt werden):

1. Verteile die (unsortierten) Karten des (ersten) Stapels gleichmäßig auf zwei neue Stapel: eine links, eine rechts usw. {*Der (erste) Stapel ist danach leer*[49]*; die beiden neuen Stapel haben gleich viele Karten.*}
2. Von jedem der neuen Stapel nimm nun jeweils die erste Karte, vergleiche die beiden und lege beide, die höhere zuerst (zuunterst), auf den ersten Stapel.
3. Wiederhole Schritt 2, bis beide Stapel leer sind. {Sie sind immer zugleich leer, da sie gleich groß sind. Stapel 1 ist danach teilsortiert: Von jedem Kartenpaar ist die erste, obere die niedrigere.}
4. Verteile nun wieder die Karten vom ersten Stapel gleichmäßig auf die zwei anderen Stapel, aber diesmal in Paaren: eine links, noch eine links (nicht zwei auf einmal!), eine rechts, noch eine rechts usw. {*Beide Stapel sind nun ebenfalls teilsortiert, aber in umgekehrter Reihenfolge: Es kommt jeweils die höhere Karte zuerst.*}
5. Von jedem der beiden neuen Stapel nimm nun eine Karte und vergleiche die beiden. Leg die höhere auf den (zunächst leeren) ersten Stapel und nimm vom dem Stapel, von dem sie stammte, eine weitere (die zweite) Karte.
    a. Ist auch sie höher als die vom anderen Stapel, leg auch sie auf den ersten Stapel und danach die andere (niedrigere) Karte sowie danach zusätzlich die zweite Karte von ihrem Stapel {*von der wir aufgrund der vorherigen Sortierung wissen, dass sie niedriger als die erste vom selben Stapel ist*}.

---

[49] In der Informatik kann ein Stapel leer sein, auch wenn man im Fall eines echten (physischen) Kartenstapels von dem leeren Platz, an dem einmal der Stapel war, nicht mehr von einem Stapel sprechen würde. Technisch kann man einen Stapel als ein *Objekt* mit einer *Identität* und einem *Zustand* auffassen; die Identität (und damit der Stapel) bleibt erhalten, wenn sich der Zustand von „belegt" auf „leer" ändert.

b. Ist sie dagegen niedriger als die vom anderen Stapel, leg zunächst die andere Karte auf den ersten Stapel und nimm hiernach von deren Stapel die zweite. Leg von den beiden verbleibenden Karten nun erst die höhere und dann die niedrigere auf den dritten Stapel.

6. {Von den vier obersten Karten auf Stapel 1 ist nun die erste — oberste — die niedrigste, die zweite die zweitniedrigste, die dritte die zweithöchste und die vierte die höchste.} Wiederhole Schritt 5, bis beide neuen Stapel leer sind. {Sie sind auch diesmal zugleich leer, da sie gleich groß sind und in allen Durchführungen von Schritt 5 von jedem Stapel zwei Karten genommen werden. Der erste Stapel ist danach in Quartetten teilsortiert.}

7. Verteile nun wieder die Karten vom ersten Stapel gleichmäßig auf zwei Stapel, aber diesmal in Quartetten (jeweils immer vier nacheinander auf einen Stapel).

8. Lege nun wie in Schritt 2 und 5, aber diesmal je vier von jedem Stapel, die Karten der beiden neuen Stapel auf den ersten Stapel, so dass die zusammen acht Karten auf dem ersten Stapel sortiert sind (die niedrigste zuoberst).

9. Wiederhole Schritt 8, bis beide Stapel leer sind. {Der erste Stapel besteht nun aus sortierten Oktetten.}

10. Verteile die Oktette auf die zwei neuen Stapel und füge diese beiden Stapel wie oben in den ersten bestehend aus zwei sortierten Gruppen à 16 Karten zusammen.

11. Verteile diese Karten wie gehabt auf die zwei anderen Stapel, die nun jeder für sich vollständig sortiert sind; füge diese beiden wie oben auf den ersten zusammen und fertig. {Auf dem ersten Stapel liegen nun wieder 32 Karten, die jetzt aber vollständig sortiert sind.}

Dieses konkrete Beispiel von einem Algorithmus gibt Anlass zu einigen Anmerkungen:

- Der Algorithmus ist auch auf Kartenspiele, die aus zwei, vier, acht oder 16 Karten bestehen, anwendbar, wenn man ihn entsprechend verkürzt (früher aufhört).

- Man kann vermuten, dass er auch auf Kartenspiele mit 64 Karten und allen höheren Zweierpotenzen anwendbar ist, wenn man ihn entsprechend erweitert; ganz sicher kann man sich aber nicht sein, da der Schein auch trügen kann.

- Die Vermutung der Erweiterbarkeit des Algorithmus auf größere Kartenspiele wird unterstützt durch seine Formulierung in den obigen elf Schritten. Der Algorithmus weist nämlich eine gewisse Regelmäßigkeit auf, die man sprachlich daran erkennt, dass er gegen Ende immer knapper formuliert ist. Dabei habe ich darauf vertraut, dass der Leser das Schema zu erkennen und selbständig anzuwenden vermag. Tatsächlich verlangt aber schon die Modifikation von Schritt 5 (mit zweimal zwei Karten) für die Anwendung in Schritt 8 (mit zweimal vier Karten) eine intelligente Eigenleistung, da nicht beschrieben steht, wie man acht Karten durch Vergleich von stets nur zweien von zwei verschiedenen Stapeln (die die Reihenfolge, in der die Karten betrachtet werden, festlegen) so zusammenführt, dass sie anschließend in

der richtigen Sortierreihenfolge vorliegen. Dabei ist die Beschreibung von Schritt 5 (für nur vier Karten) schon die komplexeste von allen elf; wie mag da erst ein vollständig ausformulierter Schritt 8 ausfallen?

- Hat man diese Regelmäßigkeit allerdings erkannt, dann ist es nicht mehr so schwierig, den Algorithmus für größere Zweierpotenzen zu erweitern. Dabei fällt dann weiter auf, dass die Anzahl der Karten nicht im Algorithmus festgeschrieben sein muss, sondern vielmehr ein Parameter des Algorithmus sein sollte. Man würde also so etwas wie eine *Schleife* erwarten, die, mit zweimal einer Karte beginnend, solange die Kartenzahl verdoppelt, bis die Hälfte der Gesamtkartenzahl erreicht ist (bei 32 Karten also zweimal 16 Karten betrachtet wurden, bei 64 Karten zweimal 32 usw.).

- Eine solche Generalisierung von kleinen auf große Beispiele ist ein bewährtes Verfahren, einen Algorithmus zu entwickeln: Man löst erst eine kleine Version des Problems und dann schrittweise immer größere, bis man das allgemeine Schema zu erkennen glaubt, welches man dann überprüft, indem man es auf noch größere Probleme anwendet.

- Der Algorithmus sollte für *jeden* unsortierten Kartenstapel, auf den er anwendbar ist (weil die Anzahl der Karten eine Zweierpotenz ist), auch das korrekte Ergebnis, also einen aufsteigend sortierten Stapel, liefern. Für kleine Kartenspiele (z. B. mit nur zwei oder vier Karten) kann man das noch durch systematisches Ausprobieren (auch *Testen* genannt) herausfinden, für größere aber wird das schwierig, da man alle möglichen Eingangsfolgen (Mischungen) von Karten durchprobieren und jeweils überprüfen müsste, ob das Ergebnis stimmt. Bei zwei Karten gibt es nur zwei mögliche Folgen, für vier 24, für acht schon 40.320 und für 16 gar knapp 21 Billionen. Für beliebige Zweierpotenzen taugt dieses Verfahren überhaupt nicht, da man in begrenzter Zeit niemals alle ausprobieren kann.

- Hat man den Algorithmus allerdings so formuliert, dass er auf beliebige Zweierpotenzen anwendbar ist, dann bietet sich ein formales Beweisschema an: Man muss dann nur noch zeigen, dass erstens der Algorithmus für zwei Karten korrekt ist (z. B. durch Ausprobieren) und dass zweitens, wenn er für 2^n Karten korrekt ist, er auch für 2^{n+1} Karten korrekt ist. Dieses (aus der Mathematik bekannte) Beweisschema spielt in der Informatik eine herausgehobene Rolle: Es nennt sich *Induktion*.

- Allgemeiner kann man die *Korrektheit eines Algorithmus* beweisen, indem man zeigt, dass er die sog. *Vorbedingung* in die sog. *Nachbedingung* überführt (s. Abschnitt 17.2). Im gegebenen Beispiel wäre die Vorbedingung etwa, dass die Anzahl der Karten eine Zweierpotenz ist, und die Nachbedingung, dass die Karten sortiert sind. Dieser Beweis erstreckt sich über alle Zwischenschritte des Algorithmus und setzt sich aus dem Beweis vieler Zwischenergebnisse zusammen (in obigem Beispiel durch Text in geschweiften Klammern angedeutet). Auch wenn der *Korrektheitsbeweis* des obigen Algorithmus relativ einfach ist, sind Beweise für komplexere Algorithmen schwierig. In der Praxis der Programmierung (Gegenstand von Kapitel 18) bleiben sie regelmäßig aus.

- Neben der Korrektheit eines Algorithmus spielen auch seine *Laufzeit* und sein *Speicherverbrauch* eine wichtige Rolle. Beide hängen in der Regel von der Größe der Eingabe ab (je mehr Karten, desto länger braucht er und desto mehr Speicher verbraucht er). Für beide Größen erhält man mittels einer sog. *Komplexitätsanalyse* Angaben zum schlechtesten Fall (*Worst case*), zum besten Fall (*Best case*) sowie zum durchschnittlichen Fall (*Average case*). Gemessen an beiden Größen (Laufzeit und Speicherverbrauch) ist der obige Sortieralgorithmus kein besonders guter; er hat allerdings den Vorteil, dass er auch mit rein *sequentiellen Speichern* (wie etwa Magnetbändern) funktioniert. Die meisten anderen Sortieralgorithmen verlangen dagegen *wahlfreien Zugriff* auf den Speicher, in dem die zu sortierenden Elemente vorliegen, was bei sehr großen Datenmengen auch heute noch ein Problem sein kann.
- Die Tatsache, dass der Algorithmus für eine bestimmte Speicherform (hier: sequentiellen Speicher) geeignet ist, legt einen Zusammenhang zwischen Algorithmen und Datenstrukturen nahe. Dieser wird in Kurseinheit 4 wieder aufgenommen.
- Zuletzt wäre auch interessant zu wissen, wie der Algorithmus abgeändert werden muss, damit er auch für andere Kartenzahlen als Zweierpotenzen funktioniert, also etwa für ein Spiel mit zehn Karten. Mit dem Algorithmus müsste dann allerdings auch der Korrektheitsbeweis angepasst werden (sofern er überhaupt erbracht wurde).

Algorithmen gehören der Menschheit!

Die Algorithmen der Informatik sind, ähnliche wie die Sätze (Theoreme) der Mathematik, in aller Regel Allgemeingut und werden entsprechend in Aufsätzen oder Büchern veröffentlicht, nicht zuletzt, weil ein kluger Algorithmus seinem Autor einiges an persönlichem Ruhm bescheren kann. Zudem können Algorithmen zumindest in Europa nicht urheberechtlich geschützt und nur bedingt patentiert werden. Wenn man eine Verwertung durch andere verhindern will, sollte man selbst entdeckte Algorithmen also geheim halten.[50]

17 Spezifikation

Während ein *Algorithmus* vorschreibt, *wie* etwas zu tun ist (beispielsweise wie man sortiert), schreibt eine **Spezifikation** vor, *was* zu tun ist (im Beispiel ein Sortiergut in die richtige Reihenfolge bringen). Die Spezifikation einer *Software* (Algorithmus oder Programm) dient

[50] Historische Fußnote: *Newton* hielt seine Verfahren („Algorithmen") zur Infinitesimalrechnung zunächst geheim, um sich in damals geübten Mathematiker- und Physikerwettbewerben Vorteile zu verschaffen. Damit war es vorbei, nachdem *Leibniz* seine gleichwertigen Verfahren publiziert hatte. Es ist bis heute niemand gezwungen, seine algorithmischen Entdeckungen zu veröffentlichen, und so ist Geheimhaltung immer noch der beste Weg, sich die alleinige Verwertung von selbst entdeckten Algorithmen zu sichern. S. dazu auch Abschnitt 18.

der Überprüfung der *Korrektheit* der Software — eine Software ist korrekt, wenn mit dem Wie das Was erreicht wird. Ohne Spezifikation ist jede Software korrekt.

Eine Spezifikation listet die *Anforderungen an eine Software* auf. Spezifikationen sind regelmäßig Bestandteile von Verträgen zwischen einem Auftraggeber und einem Auftragnehmer mit dem Gegenstand der Lieferung einer Software; sie sind nicht selten Beweismittel in juristischen Auseinandersetzungen, nämlich wenn die gelieferte Software den an sie gerichteten Erwartungen nicht gerecht wird. Es stellt sich dann die Frage, ob die Software ihre Spezifikation erfüllt — ob die Spezifikation im Einklang mit den Erwartungen steht, ist eine unabhängige Fragestellung (die gleichwohl zu Streitigkeiten führen kann, dann aber unter anderen Parteien).

It is easier to change the specification to fit the program than vice versa.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

17.1 Spezifikation als Teil eines Softwareentwicklungsprozesses

Die sich daraus unmittelbar ergebende Frage ist, wer denn die Spezifikation erstellt und dafür verantwortlich ist. Für viele Problemstellungen ist die Spezifikation zu erstellen nämlich ein nichttriviales Problem, da diese häufig, anders als beispielsweise beim Sortieren, nicht auf der Hand liegt, sondern über die Köpfe von vielen Menschen verteilt und in etablierten, ggf. abzulösenden Arbeitsprozessen dokumentiert ist. *Anforderungserhebung* und darauf basierend die Spezifikation sind daher eigene Phasen eines in Phasen gegliederten *Softwareentwicklungsprozesses*. Da die Abnahme (durch den Auftraggeber) erst am Ende eines solchen Prozesses steht, können Fehler bei der Anforderungserhebung (die nur schwer zu vermeiden sind) fatale Auswirkungen für ein Projekt haben. Man versucht daher in dafür geeigneten Projekten (und das sind längst nicht alle!), von einem phasischen Entwicklungsprozess abzukehren und kurze Rückkopplungsschleifen einzubauen (sog. *agile Softwareentwicklung*). Das Problem solcher Prozessmodelle ist allerdings, dass die Anforderungen nicht von Anfang an feststehen und es somit unmöglich ist, den Aufwand für die Entwicklung (gemessen in Zeit und Geld) schon am Anfang des Projekts abzuschätzen.

Walking on water and developing software from a specification are easy if both are frozen.

Edward V Berard

Lasten- und Pflichtenheft

In der Praxis wird die Anforderungserhebung oft in die Erstellung eines **Lasten-** und eines **Pflichtenhefts** aufgeteilt. Das Lastenheft stellt die Anforderungen aus Sicht des Auftraggebers dar, der häufig über wenig Informatikexpertise

verfügt; das Pflichtenheft detailliert diese Anforderungen, indem es sie technisch aufbereitet und auf Vollständigkeit und Widerspruchsfreiheit überprüft. Wegen der hohen technischen Anforderungen an ein Pflichtenheft wird es häufig vom Auftraggeber auf Basis des Lastenhefts ausgeschrieben und beauftragt; die Umsetzung des Pflichtenhefts erfolgt dann nach gesonderter Ausschreibung und Beauftragung (und somit nicht immer durch dieselben, die das Pflichtenheft erstellt haben).

Arten von Anforderungen

Die Anforderungen an ein System werden (in einem Pflichtenheft) häufig in sog. **funktionale** und **nichtfunktionale** aufgeteilt. Funktionale Anforderungen spezifizieren in der Regel die Daten, die ein System verarbeiten können muss, und die Funktionen, die es dem Nutzer bietet; nichtfunktionale Anforderungen betreffen häufig Qualitätsansprüche an ein System wie Verfügbarkeit (Ausfallzeiten), Wartbarkeit und Skalierbarkeit. *Korrektheit* zählt hingegen in der Regel zu den funktionalen Eigenschaften und insgesamt ist die Abgrenzung so schwammig, dass ich eine Unterscheidung zwischen **fachlichen** (also aus der Anwendungsdomäne einer Software stammenden) und **nichtfachlichen** Anforderungen bevorzuge. Erstere treffen häufig auf genau ein (zu entwickelndes) Softwaresystem zu, während letztere sich oft von einem System auf ein anderes übertragen lassen.[51]

17.2 Formale Spezifikation

Da es in der oben beschriebenen Form der Spezifikation (mit Lasten- und Pflichtenheft) erheblich menschelt, bevorzugen Informatiker in der Regel die **formale Spezifikation**, deren Einhaltung sich im Idealfall mit strengen (formalen) Methoden beweisen lässt. So lässt sich beispielsweise das geforderte Ergebnis des aufsteigenden Sortierens der Elemente einer Liste durch die *Zusicherung*, dass von je zwei aufeinanderfolgenden Elementen der Ergebnisliste das erste kleiner sein muss als das zweite, spezifizieren. Ein Sortierverfahren, das diese Bedingung für jede mögliche Eingabeliste erfüllt, ist korrekt; eines, das das nicht tut, ist es nicht. Da gibt es keine Diskussionen.

Vor- und Nachbedingung

Allgemein (und wie in Kapitel 16 schon erwähnt) nennt man eine Bedingung, die am Ende eines Verfahrens (Algorithmus oder Programm; für Programm s. Kapitel 18) erfüllt sein muss, seine **Nachbedingung** (engl. *Postcondition*). Da es in der Regel nicht sinnvoll ist, von einem Verfahren für jede mögliche Eingabe die Einhaltung der Nachbedingung zu erwarten, paart man sie mit einer **Vorbedingung** (engl. *Precondition*). So ist beispielsweise die Vorbedingung für den in Kapitel 16 beschriebenen, auf Listen von Zahlen anstelle von Kartenstapeln angewendeten Sortieralgorithmus, dass

---

[51] Andererseits ist auch denkbar, dass ein vorhandenes System unter Beibehaltung der fachlichen Anforderungen komplett neu entwickelt wird, weil sich die nichtfachlichen Anforderungen geändert haben. Dies ist insbesondere bei sog. *Altsystemen* (engl. auch als *legacy systems* bezeichnet) der Fall; die Altsysteme selbst stellen dann eine fachliche Spezifikation dar, die vollständiger ist, als es ein Pflichtenheft realistischerweise je sein könnte.

die zu sortierende Liste 32 Zahlen enthält; für alle anderen Eingaben muss er nicht das korrekte Ergebnis, ja muss überhaupt keines liefern. Nachbedingungen sind, dass die Elemente der Liste in der Tat sortiert sind sowie dass keine verlorengegangen und keine hinzugekommen sind.

Invariante und Zusicherung

Vorbedingungen und Nachbedingungen kann man nicht nur für ganze Algorithmen (oder Programme; s. Abschnitt 18.1) angeben (wobei die Vorbedingungen in der Regel von den Eingaben erfüllt sein müssen und Nachbedingungen von den Ausgaben), sondern auch für Teile davon. Tatsächlich verlangt der formale *Beweis der Korrektheit eines Algorithmus* (s. Abschnitt 17.3) die Formulierung von Vor- und Nachbedingungen für jeden einzelnen Schritt. Neben Vor- und Nachbedingungen (von ganzen Algorithmen oder Teilen davon) verwendet man auch sog. **Invarianten**, die immer erfüllt sein müssen. Vorbedingungen, Nachbedingungen und Invarianten bezeichnet man auch als **Zusicherungen**; damit sind zusammenfassend Aussagen gemeint, die erfüllt sein müssen, damit ein Algorithmus korrekt arbeitet.

Zusicherungen sind Best practice

Damit Sie sich eine Vorstellung davon machen können, welchen Charakter Zusicherungen haben, sind solche bereits in den (verbal spezifizierten) Sortieralgorithmus von Kapitel 16 eingestreut, und zwar als Text in geschweiften Klammern. Jede dieser Zusicherungen sollte erfüllt sein, wenn der oder die vorherigen Schritte richtig umgesetzt wurden. Neben dieser Prüffunktion dienen Zusicherungen aber auch dazu, sich beim Entwickeln eines Algorithmus im Klaren darüber zu sein, was man mit jedem einzelnen Schritt erreicht hat und wie die Folge der Schritte zum gewünschten Ergebnis führt. Zusicherungen dienen also der Konstruktion, der *Verifikation* (s. u.) und der *Dokumentation* — ich möchte jedem, der einen Algorithmus entwickelt, wärmstens empfehlen, sich schon beim Aufschreiben der einzelnen Schritte (und nicht erst hinterher) dazu passende Zusicherungen zu überlegen. Gleiches gilt auch für die Programmierung (Kapitel 18).

17.3 Formaler Beweis und Testen der Korrektheit: Verifikation und Falsifikation

Der Wert einer formalen Spezifikation liegt darin, dass man von einem Algorithmus oder Programm mit Strenge sagen kann, ob es diese Spezifikation umsetzt. Einen solchen Nachweis der korrekten Umsetzung der Spezifikation nennt man auch **Verifikation**. Ein Beispiel für eine Verifikation in Form eines *Korrektheitsbeweises* finden Sie in Abschnitt 18.3.1.

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald E. Knuth (* 10. Januar 1938)

Falsifikation durch Testen

Gelingt eine *Verifikation* nicht (etwa weil der Beweis zu aufwendig oder die Spezifikation selbst unvollständig oder falsch ist), kann man aus einer

(formalen) Spezifikation immer noch *Tests* ableiten, die die *Korrektheit eines Algorithmus* widerlegen, was einer **Falsifikation** gleichkommt. Dazu müssen Eingaben gefunden werden, die die Vorbedingungen des Algorithmus erfüllen, deren dazugehörige, vom Algorithmus gefundene Ausgaben aber eine Nachbedingung verletzen. Im Beispiel eines Sortieralgorithmus wäre also eine Liste von Zahlen zu finden, die nicht richtig sortiert wird oder aus der Elemente verloren gehen oder der Elemente hinzugefügt werden. Zwar kann eine solche Verletzung einer Nachbedingung immer noch auf falsch formulierte Vor- oder Nachbedingungen (eine falsche Spezifikation) zurückgeführt werden, doch zwingt jede Diskrepanz zwischen erwartetem und geliefertem Ergebnis dazu, noch einmal alles genau zu überprüfen (ganz so, wie eine starke Diskrepanz zwischen einer Überschlags- und einer Hauptrechnung Anlass zu Misstrauen gibt). Lässt sich das unerwartete Ergebnis nicht auf einen Fehler in der Spezifikation zurückführen, dann muss er im Algorithmus liegen und dort behoben werden. Man beachte jedoch, dass der Versuch, durch Testen die Korrektheit eines Algorithmus zu beweisen, in aller Regel aussichtslos ist, denn dafür müsste man zeigen, dass keine mögliche Eingabe zu einem Fehler führt (also für jede mögliche Eingabe die Falsifikation misslingt), und dafür gibt es einfach zu viele verschiedene mögliche Eingaben (s. dazu auch die entsprechende Anmerkung zum Sortieralgorithmus von Kapitel 16).

Testing shows the presence, not the absence of bugs.

Edsger Dijkstra (* 11. Mai 1930, † 6. August 2002)

Es bleibt also zum strengen Nachweis der Korrektheit eines Algorithmus nur die (formale) Verifikation. Diese wird für in wissenschaftlichem Kontext veröffentlichte Algorithmen auch regelmäßig verlangt (und häufig geliefert); für Programme, die (implizit oder explizit) Algorithmen umsetzen, ist sie (nicht zuletzt wegen des *niedrigeren Abstraktionsniveaus* und des größeren Funktionsumfangs von Programmen; s. Kapitel 18) leider außerordentlich aufwendig und bis heute nur für sehr wenige größere Programme vollständig durchgeführt worden (vgl. dazu Abschnitt 3.2.2 in Kurseinheit 1).[52]

17.4 Algorithmen und Korrektheit heute

Seit einigen Jahren zeichnet sich ein gewisser Bedeutungswandel von Algorithmus und *Korrektheit* ab: Statt „korrekt“ heißt es immer häufiger von einem Algorithmus, er sei „gut genug“. Ein Matching-Algorithmus etwa ist gut genug, wenn er genügend Menschen überzeugt und somit Marktanteile sichert, ein selbstfahrendes Fahrzeug ist gut genug, wenn es im Mittel erkennbar weniger Schäden verursacht als ein menschengesteuertes usw. Das Ziel

---

[52] Tatsächlich gilt dies vor allem für *imperativ* formulierte Algorithmen und Programme; *funktionale* und *logische* sind näher an einer Spezifikation und insofern einfacher als korrekt zu beweisen. S. Kapitel 19 für die Unterscheidung von imperativer, funktionaler und logischer Programmierung.

der Korrektheit im Sinne einer Unfehlbarkeit wird damit aufgegeben — Verluste müssen in Kauf genommen werden.

Es scheint heute noch schwer vorstellbar, in Zusammenhang mit Sortierung etwa oder Banküberweisungen von „gut genug" zu sprechen, wenn nur selten Elemente falsch einsortiert oder die Kontenstände meistens stimmen würden. An den Absturz von Programmen haben wir uns jedoch längst gewöhnt und letztlich entscheidet die Ökonomie, ob sich Korrektheit überhaupt lohnt. Außerdem ist überall da, wo die Datenlage unsicher ist, Korrektheit kein Garant für Nützlichkeit: Was in unserem Alltag ließe sich schon so exakt erfassen, dass ein korrekter Algorithmus stets das brauchbarste Ergebnis erzielt? Würde es in einer Welt, in der Menschen nach dem Algorithmus der stabilen Heirat miteinander vermählt würden, keine Scheidungen mehr geben? Nicht zuletzt ist ja auch der Begriff der *künstlichen Intelligenz* — zumindest nach *Turing* — nur statistisch gefasst (s. Abschnitt 3.4 in Kurseinheit 1).

Link

18 Programm

Algorithmus ≠ Programm

Damit ein Algorithmus einen praktischen Nutzen hat, muss er in ein Programm umgesetzt werden, das von einem Computer ausgeführt werden kann. Dabei werden die Begriffe *Algorithmus* und *Programm* gelegentlich synonym verwendet, aber viele, insbesondere umfangreiche (große) Programme setzen einerseits mehrere Algorithmen um und bestehen andererseits auch aus Teilen, die so schlicht sind, dass man von ihnen nicht als Umsetzung eines Algorithmus sprechen würde (beispielsweise bei der Ein- und Ausgabe von Daten). In diesem Kurstext werden daher *Algorithmus* und *Programm* nicht synonym verwendet (und in den meisten Erwähnungen des Begriffs *Algorithmus* in den Medien sollet er nach der Lesart dieses Kurstextes durch *Programm* ersetzt werden).

Programme sind in Deutschland anders als Algorithmen (s. Kapitel 16) grundsätzlich urheberrechtlich geschützt. Da sich aus Programm(text)en aber die darin umgesetzten Algorithmen rekonstruieren und diese sich in anderer Form wieder in ein (dann urheberrechtlich unbedenkliches) Programm umsetzen lassen, werden schützenswerte Programme vor der Auslieferung gelegentlich für Menschen „unlesbar" gemacht (ein engl. *obfuscation* genannter Vorgang).[53]

18.1 Programme als Texte

Wie wir in Abschnitt 11.5 gesehen haben, kann die *Hardware* eines Computers nur *Maschinenbefehle* ausführen. Sieht man von Aufrufen von Funktionen, die das Betriebssystem eines Computers zur Verfügung stellt, ab, so müssen alle Programme zur direkten Ausführung

[53] Mit Obfuscation können auch illegale Programmteile (wie beispielsweise Abschalteinrichtungen) verborgen werden. Im Gegensatz dazu steht *quelloffene*, oder *Open-source-Software*, bei der der originale Programmtext allgemein zugänglich ist.

auf dem Computer als Folge solcher Befehle vorliegen. Doch selbst bei der Programmierung in *Maschinensprache* (die heute höchstens noch bei eingebetteten System vorkommt; s. Kapitel 25) werden die Programme nicht direkt in den Programmspeicher geschrieben — ein *Maschinenprogramm* (und jedes andere Programm sowieso) ist ein von der Maschine losgelöstes (gleichwohl für sie bestimmtes) Artefakt, das in der Regel die Gestalt eines Textes hat.

Editoren

Programme werden heute ausschließlich mithilfe von anderen Programmen, die auf demselben oder einem anderen Computer laufen, geschrieben. Das wichtigste Programm ist dabei der **Editor**, eine Art Textverarbeitung, mit der die Programme in Textform eingegeben werden können. Solche Editoren unterscheiden sich von anderen Textverarbeitungen dadurch, dass sie den Programmtext in Zeilen und nicht in Absätze unterteilen und dass sie das Konzept der Seite nicht kennen. Außerdem ist die Bedienung vieler klassischer Editoren kommando- und nicht wie die heutiger Textverarbeitungen menüorientiert.

Programmzeilen

Während ein Programm in Maschinensprache noch als eine Folge von Bits, Bytes oder Wörtern verstanden werden kann (so wie in Kapitel 9 in Kurseinheit 1 dargestellt), bestehen Programme spätestens mit der Einführung der sog. *Assemblersprachen* aus **Programmzeilen**. In vielen frühen, auch sog. *höheren*, Programmiersprachen dienten Zeilen als Ziele von Sprüngen in einem Programm (zum Zweck der *Verzweigung*, der *Wiederholung* und des *Unterprogrammaufrufs*; so z. B. in *BASIC*; s. Kapitel 24) und waren zu diesem Zweck nummeriert. Sprünge, zu Programmzeilen zumal, gelten jedoch spätestens seit 1968 als verpönt und wurden durch die *Blockform* der *strukturierten Programmierung* ersetzt (s. dazu auch Abschnitt 19.1). Obwohl damit das Konzept der Programmzeile eigentlich obsolet wurde, dominiert es immer noch die Sicht auf Programme, selbst in Programmiersprachen, in denen es gar nicht mehr passt. Tatsächlich wird der Zeilenumbruch in Programmen heute vor allem als stilistisches Mittel zur Steigerung der Lesbarkeit verwendet und in nur wenige Programmiersprachen (darunter *Python*!) hat der Zeilenumbruch einen Einfluss auf die Bedeutung (*Interpretation* oder *Semantik*) eines Programms.

Bestandteile eines Programmtextes

Genau wie Texte natürlicher Sprachen besteht auch ein Programm aus Wörtern und Satzzeichen. Wörter werden durch *Leerzeichen* voneinander getrennt, wobei zu den Leerzeichen auch Zeilenumbrüche und Tabulatoren zählen (sog. *White spaces*). Unter den Satzzeichen finden sich auch solche, die einen sog. **Kommentar** vom Rest des Programmtextes abgrenzen. Kommentare dienen der Erläuterung eines sonst vielleicht unverständlichen Programms sowie der Dokumentation von Entscheidungen des Programmierers („darum habe ich das so und so gemacht und nicht so und so"). Ein Satzzeichen vieler Programmiersprachen ist das Semikolon, das (je nach Sprache) entweder eine Anweisung abschließt oder zwei Anweisungen voneinander trennt. Komma und Punkt können ebenfalls Satzeichen sein, sind in manchen Programmiersprachen aber auch Operatoren.

Schlüsselwörter und Namen

Die Wörter eines Programms werden weiter unterschieden in sog. **Schlüsselwörter** (wie `if`, `then`, `else`, `begin` oder `end`[54]), **Operatoren** (wie `+`, `-`, `*`, `/` und `=`, wobei `*` für „mal" und `/` für „geteilt durch" stehen) und **Namen** oder **Bezeichner** (im Englischen auch „identifier" genannt, nicht zu verwechseln mit den *Identifiern* von Datensätzen) wie `x` oder `Wurzel`, die in der Regel von Schlüsselwörtern verschieden sein müssen, die aber ansonsten vom Programmierer frei gewählt werden dürfen und die Elemente eines Programms wie Variablen, Funktionen oder Prozeduren bezeichnen.

Programme im Wandel der Zeit

Bei älteren Programmiersprachen ist die Anzahl der Schlüsselwörter eher groß (bei Cobol etwa). Zugleich sind in älteren Programmen die von Programmierern vergebenen Namen eher kurz. Ein Beispiel hierfür ist etwa das *Pascal*-Programm

```
1 PROCEDURE bubblesort;
2   VAR i, j: index;  x: item;
3 BEGIN FOR i := 2 TO n DO
4   BEGIN FOR j := n DOWNTO i DO
5     IF a[j-1].key > a[j].key THEN
6     BEGIN x := a[j-1]; a[j-1] := a[j]; a[j] := x
7     END
8   END
9 END {bubblesort}
```

aus *Niklaus Wirths* Klassiker „Algorithmen und Datenstrukturen", das ein *Array* `a` mit `n` *Records* anhand deren Element `key` gemäß einem *Bubblesort* genannten Algorithmus sortiert (s. Abschnitt 27.2 in Kurseinheit 4 für die verwendeten Datenstrukturen Array und Record). Sie müssen das Programm nicht verstehen, sollten aber erkennen, dass die Schlüsselwörter den Programmtext dominieren, und das nicht nur, weil sie großgeschrieben sind. Fast dasselbe Programm in der Programmiersprache *Java* sieht heute eher so aus:

```
10 private void tausche(int index1, int index2)
11 {
12    int temp = zahl[index1];
13    zahl[index1] = zahl[index2];
14    zahl[index2] = temp;
15 }
16
17 public void bubblesort()
18 {
19   for (int durchgang = 1; durchgang < MAX; durchgang++)
20     for (int index = 0; index < MAX-durchgang; index++)
21       if (zahl[index] > zahl[index+1])
22         tausche(index,index+1);
23 }
```

---

[54] Man beachte, dass von hier an Texte, Wörter und Zeichen, die so in Programmen vorkommen sollen, durch einen anderen Font gekennzeichnet werden. In Kurseinheit 4 werde ich das für Daten fortsetzen. Dies folgt einer Konvention, deren Ursprung darin begründet liegt, dass Computer und Drucker früher nur einen fest eingebauten Font hatten. Heute kann man natürlich auch in Frutiger programmieren, wenn man es schön findet.

Abgesehen davon, dass das Java-Programm in zwei Teile zerlegt wurde, deren erster ein *Unterprogramm* des zweiten darstellt, sind seine Namen allesamt selbsterklärend (was man vom obigen Pascal-Programm nicht unbedingt sagen kann) und dominieren so den Programmtext. Dies liegt aber nicht an den Programmiersprachen (in beiden Sprachen lassen sich kurze und lange Namen verwenden), sondern am *Programmierstil*, der sich über die Jahre vom mathematisch-kompakten entfernt und mehr dem verbosen, selbsterklärenden zugewandt hat (es spricht auch heute kaum noch jemand von Codieren, wenn er Programmieren meint). Übrigens: In Film und Fernsehen werden Sie bevorzugt Programme der ersten Sorte finden — sie passen einfach besser zum Klischee des Unverstehbaren, das der Programmierung anhaftet.

Syntax highlighting

Bei der Darstellung von Programmen in einem Editor oder auf Papier hat es sich eingebürgert (und bewährt), Vorkommen von *Schlüsselwörtern* hervorzuheben (das sog. *Syntax highlighting*); die somit deutlich hervortretenden Schlüsselwörter bilden gewissermaßen das Skelet eines Programms. Die Namen zwischen den Schlüsselwörtern bilden demnach das Fleisch: Hier selbsterklärende Namen zu verwenden macht das Lesen und Verstehen eines Programms einfacher und so manchen *Kommentar* als Erläuterung überflüssig. Naturgemäß ist Syntax highlighting bei Sprachen mit wenigen oder gar keinen Schlüsselwörtern (wie etwa der logischen Programmiersprache *Prolog*) schlecht möglich.

18.2 Vom Algorithmus zum Programm

Wenn man nun also einen Editor hat, mit dem man Programme als Texte eingeben kann, bleibt immer noch die Frage, wie man vom Algorithmus zum Programm kommt. Dazu zunächst wieder ein Beispiel. Wir nehmen dazu einen Algorithmus zur näherungsweisen Bestimmung der *Quadratwurzel* einer positiven Zahl, den schon die alten Sumerer verwendeten. Nach diesem Algorithmus beginnt man mit einem Schätzwert x_0 (der beliebig falsch, aber nicht **0** sein darf) der Wurzel einer gegebenen Zahl a und berechnet davon ausgehend den nächsten Wert x_1 durch Einsetzen von x_0 in die allgemeine Formel

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

sowie weiter die Werte x_2, x_3, …, bis das Quadrat von x_n für ein n hinreichend nahe an a ist, also bis

$$\left|x_n^2 - a\right| < \epsilon$$

für ein frei zu wählendes, kleines ϵ gilt (wobei die Betragsstriche | | für den Absolut- oder positiven Wert einer Zahl stehen). Wie dieser Algorithmus funktioniert, lässt sich am besten veranschaulichen, indem man ihn per Hand auf einem Blatt Papier oder in einem *Tabellenkalkulationsprogramm* für ein paar *Iterationen* „ausrollt", d. h., die Werte in die Formel einsetzt und zusammen mit den Ergebnissen neben- oder untereinander hinschreibt. Für $a = 2$, $e = 0{,}001$, $x_0 = a$ und n von **0** bis **3** in einer Tabellenkalkulation ausgerollt ergibt dies

	A	B	C	D	E
1	a =	2			
2	e =	0,001			
3	n =	0	1	2	3
4	x_n =	=B1	=(B4+B1/B4)/2	=(C4+B1/C4)/2	=(D4+B1/D4)/2
5	abs(x_n^2-a)<e =	=ABS(B4^2-B1)<B2	=ABS(C4^2-B1)<B2	=ABS(D4^2-B1)<B2	=ABS(E4^2-B1)<B2

(wobei `abs(x)` $|x|$ entspricht) oder, wenn man sich anstelle der Formeln die Werte anzeigen lässt,

	A	B	C	D	E
1	a =	2			
2	e =	0,001			
3	n =	0	1	2	3
4	x_n =	2	1,5	1,41667	1,41422
5	abs(x_n^2-a)<e =	FALSCH	FALSCH	FALSCH	WAHR

Um diesen Algorithmus nun zu programmieren, also in ein Programm umzusetzen, das auf einem Computer ablaufen kann (und in dem wir die Iterationen nicht wie oben „von Hand" ausrollen müssen), nehmen wir eine einfache, *Pascal*-ähnliche Programmiersprache an und schreiben (in einen Editor)

```
24 function Wurzel(a : float, e : float) : float;
25 begin
26   Wurzel := a;
27   repeat
28     Wurzel := (Wurzel + a / Wurzel) / 2
29   until abs(Wurzel * Wurzel – a) < e
30 end
```

Hierbei bedeutet das *Schlüsselwort* `function` in Zeile 24, dass im folgenden eine Funktion[55] definiert wird, die den auf das Schlüsselwort folgenden Namen (hier: `Wurzel`) trägt. In den Klammern hinter dem Namen finden sich die *Eingabeparameter* (*Variablen*), die die Eingabewerte aufnehmen, hier `a` für die Zahl, deren Wurzel zu bestimmen ist, und `e` für die geforderte Genauigkeit des Ergebnisses. Das Schlüsselwort `float` hinter den Doppelpunkten nach `a` und `e` gibt den *Typ* der Variablen an; es legt fest, dass die Werte der Variablen *Fließkommazahlen* sein müssen. `float` nach dem Doppelpunkt am Ende der Zeile gibt an, dass der Funktionswert (also das Ergebnis, das die Funktion liefert; die *Ausgabe der Funktion*) ebenfalls eine Fließkommazahl ist.[56]

---

[55] Eine Funktion ist eine Vorschrift, die eine Reihe von Eingabewerten, die Funktionsargumente, auf einen Ausgabewert, den Funktionswert, abbildet. In der imperativen Programmierung kann der Funktionswert zusätzlich vom *Zustand* des Programms abhängen (s. Abschnitt 19.1). Sie ist ansonsten mit einer mathematischen Funktion vergleichbar.

[56] In Pascal heißt der Typ von Fließkommazahlen `real`, da er zur Darstellung reeller Zahlen verwendet wird. Da sich reelle Zahlen in einem Computer schlecht darstellen lassen (s. Abschnitt 2.6 in Kurseinheit 1), ist diese Bezeichnung trügerisch und so verwende ich hier `float`, wie es in C und damit verwandten Sprachen üblich ist.

Auf den Kopf der Funktion in Zeile 24 folgt ihre Definition, oder ihre *Implementierung*, in Form einer Folge von Anweisungen, die in Pascal von den Schlüsselwörtern `begin` und `end`, die zusammen einen **Block** bilden, eingeschlossen wird (Zeilen 25–30). Dort wird, in Zeile 26, zunächst der Funktionswert, repräsentiert durch den Namen der Funktion, `Wurzel`, mit dem ersten Argument der Funktion, `a` (der Zahl, deren Wurzel bestimmt werden soll), gleichgesetzt. Diese Gleichsetzung nennt man eine **Wertzuweisung**, oder kurz **Zuweisung** (engl. *assignment*); sie erfolgt mithilfe eines Zuweisungsoperators, in Pascal `:=`, der englisch als „becomes", deutsch aber leider häufig als „gleich" (und seltener als „wird zu") gelesen wird. Die Wertzuweisung unterscheidet sich von der mathematischen Gleichheit und Gleichsetzung, dargestellt durch das Gleichheitszeichen `=`, dadurch, dass es sich um eine *Anweisung* (einen *Befehl*) handelt, der bewirkt, dass die linke und die rechte Seite hinterher den gleichen Wert haben, wobei sie vorher verschiedene Werte haben dürfen. In der Mathematik gibt es aber kein *Vorher* und kein *Nachher* oder, allgemeiner, keinen *Zustand* (vgl. Abschnitt 2.7) und die Werte müssen gleich sein, d. h., das Gleichheitszeichen ist eine *Zusicherung* und keine Anweisung.[57]

Bei der ersten Zuweisung in Zeile 26 handelt es sich nur um die des durch den Algorithmus geforderten Schätzwerts, der aber in den meisten Fällen zu ungenau ist, um das *Abbruchkriterium* zu erfüllen. Es folgt also die Berechnung besserer Werte in einer *Schleife* (Zeilen 27–29), die in diesem Fall als *Wiederhole-bis-Schleife* ausgeführt ist. In der Schleife wird dann der nächste Näherungswert der Wurzel berechnet und wiederum `Wurzel` zugewiesen. An Zeile 28 erkennt man sehr schön, was der Unterschied zu einer mathematischen Vorschrift wie der in obiger Formel zur Wurzelberechnung ist: Da die Mathematik wie gesagt keinen *Zustand* kennt, müssen die Schätzwerte als eine Reihe von Variablen, hier $x_0, x_1, \ldots$, dargestellt werden, deren Werte durch ein System von Gleichungen bestimmt ist. In der Implementierung durch obiges Programm ist diese Reihe hingegen in eine *zeitliche Abfolge* aufgelöst, d. h., dieselbe Variable `Wurzel` „enthält" (und gleicht damit) zunächst den (dem) Wert von x_0, dann den (dem) von x_1 usw. Tatsächlich ist die Zuweisung von Zeile 28 als mathematische Gleichung interpretiert in der Regel falsch, da sich der Wert der linken und rechten Seite weder vor noch nach der Zuweisung gleichen. Das Programm durchläuft eine Reihe von Zuständen, und in (fast) jedem Zustand hat `Wurzel` einen anderen Wert.

> Selbsttest: Unter welchen Umständen bekommt `Wurzel` zweimal hintereinander denselben Wert zugewiesen?

---

[57] Ich betrachte es als eine Tragödie, dass in manchen Programmiersprachen, so in den sehr weit verbreiteten Sprachen *C*, *C++*, *Java* und *C#*, das einfache Gleichheitszeichen = als Zuweisungsoperator und das doppelte == für den Test auf Gleichheit verwendet wird. Tragödie deshalb, weil in diesen Sprachen die Zuweisung keine *Anweisung* ist, sondern ein *Ausdruck*, der einen booleschen Wert hat und der überall da erscheinen darf, wo ein boolescher Wert erwartet wird. `if (x=1) then … else …` führt damit zur Ausführung des then-Teils, selbst wenn die Variable `x` zum Zeitpunkt der Ausführung den Wert `0` hat, so dass das mit `x=1` vermutlich gemeinte `x==1` zu `false` auswerten würde. Keiner weiß, wie viel Schaden diese willkürliche Festlegung der Menschheit schon zugefügt hat.

Das Abbruchkriterium der Schleife, die (wie für *Wiederhole-bis-* oder *Repeat-until-Schleifen* charakteristisch) mindestens einmal durchlaufen wird, in Zeile 29 enthält einen Aufruf der Funktion `abs`, die den Betrag (Absolut- oder positiven Wert) einer vorzeichenbehafteten Zahl zurückgibt. Diese Funktion ist entweder Teil einer *Bibliothek von Funktionen*, die der Programmiersprache zugerechnet wird, oder muss an anderer Stelle programmiert werden, etwa als

```
31 function abs(x : float) : float;
32 begin
33   if x < 0 then abs = -x else abs = x
34 end
```

Ist das Abbruchkriterium nach einem Schleifendurchlauf erreicht, sind alle Anweisungen abgearbeitet und die Berechnung des Funktionswerts somit abgeschlossen. Der Aufrufer der Funktion bekommt dann den letzten Wert von `Wurzel` als Ergebnis. Ein solcher Aufruf hat etwa die Form

```
35 x := Wurzel(2, 0.001)
```

wobei hier `2` und `0.001` die Eingaben in die Funktion `Wurzel` sind und das Ergebnis des Funktionsaufrufs einer Variable `x` zugewiesen wird.[58]

18.3 Korrektheit von Programmen

Ist man mit dem Programmieren fertig, stellt sich die Frage, ob das Programm auch *korrekt* ist. Genau wie bei Algorithmen bieten sich hier Möglichkeiten der *Verifikation* und der *Falsifikation*. Für beides braucht man *Vor-* und *Nachbedingungen*.

> Selbsttest: Die obige Implementierung des Algorithmus macht einen Fehler, der nicht dem Algorithmus zugeschrieben werden kann und der in einem bestimmten Fall zu einem *Programmabbruch* ohne Ergebnis führt. Können Sie erkennen, worin der Fehler liegt, und ihn korrigieren?

18.3.1 Verifikation von Programmen

Das obige Programm zur näherungsweisen Berechnung der Quadratwurzel ist bezüglich seiner (mathematischen) Spezifikation *korrekt*, wenn es zu einer Eingabe a eine Ausgabe x liefert, deren Quadrat x^2 sich um weniger als ϵ von a unterscheidet (Nachbedingung). Das

[58] In Programmiersprachen ersetzt der (englische) Dezimalpunkt das (deutsche) Dezimalkomma. S. dazu auch Abschnitt 2.6.3 in Kurseinheit 1.

Programm liefert den Wert von `Wurzel`, wir können also x mit `Wurzel` gleichsetzen und die Bedingung muss für `Wurzel` gelten. Nun endet das Programm (und liefert somit einen Wert) erst dann, wenn `Wurzel` die Bedingung `abs(Wurzel * Wurzel - a) < e` erfüllt, was genau obiger Nachbedingung entspricht. Wenn das Programm endet oder, wie man in der Informatik sagt, **terminiert**, dann ist das Ergebnis gemäß Spezifikation korrekt, und zwar unabhängig von der Eingabe.[59]

Die Frage ist jedoch, ob das Programm terminiert. Dazu ist es zum einen notwendig, dass sich der Wert von `Wurzel` dem gesuchten Ergebnis schrittweise annähert, und zwar zumindest so sehr, dass das *Abbruchkriterium* irgendwann erfüllt ist. Mathematisch ausgedrückt ist für eine solche Annäherung die Bedingung

$$\left|x_{n+1}^2 - a\right| < \left|x_n^2 - a\right| \quad \text{für alle } n$$

Voraussetzung; ausgedrückt in den Termini des Programms von Zeile 24–30 ergibt sie die *Schleifeninvariante*

```
abs(Wurzel * Wurzel - a) < abs(Wurzel' * Wurzel' - a)
```

in der `Wurzel'` für den Wert von `Wurzel` aus dem vorherigen Schleifendurchlauf steht. Man beachte, dass aufgrund der beschränkten Darstellbarkeit von rationalen Zahlen durch den Typ `float` (Gleitkommazahlen; s. Abschnitt 2.6.3) die *Invariante* praktisch verlangt, dass der Wert von `abs(Wurzel * Wurzel - a)` irgendwann Null wird, was jedoch aufgrund von Genauigkeitsfehlern in der Gleitkommaarithmetik nicht notwendigerweise passiert (der Wert kann sogar wieder größer werden). Daher ist es wichtig, dass `e` so gewählt wird, dass die Schleife vorher beendet wird. Das macht allerdings die Wahl von `e` zu einer eigenen Wissenschaft.

Zum andern ist dem aufmerksamen Leser sicher aufgefallen, dass man ja auch einen negativen Wert für `e` eingeben könnte, was, da `abs` stets einen positiven Wert oder `0` zurückliefert, dazu führen würde, dass die Schleife und somit auch das Programm niemals beendet wird. Des Weiteren ist eine gewisse Skepsis angebracht, was die Eingabe von negativen Werten für `a` angeht, da die Wurzel einer negativen Zahl zumindest im Bereich der reellen Zahlen nicht bestimmbar ist (und die Sumerer ziemlich sicher noch keine imaginären Zahlen kannten). Und tatsächlich kann man, wie man sich durch Ausprobieren (Einsetzen in die obige Tabelle) überzeugen kann, nicht erwarten, dass der Wert von `Wurzel` für negative `a` konvergiert und die Schleife und somit das Programm terminiert. Zwar kann man negative

---

[59] Hierbei seien Fehler, die durch die Gleitkommaarithmetik entstehen, vernachlässigt; vgl. dazu Abschnitt 2.6 in Kurseinheit 1. Man beachte, dass man — paradoxerweise, wo es hier doch um eine näherungsweise Berechnung geht — beim strikten *Beweisen der Korrektheit* die Fehler der Gleitkommaarithmetik einbeziehen muss, was bedingt, dass man im Beweis abbilden muss, wie der Computer mit Gleitkommazahlen rechnet. Das Arbeiten mit Gleitkommazahlen bleibt denn auch eines der lästigsten Probleme der Informatik — wenn Sie können, sollten Sie die Verwendung rationaler Zahlen (Brüche; s. Abschnitt 2.6.1) in Betracht ziehen.

Werte für `a` und `e` als Unsinn abtun und entsprechende Eingaben als fehlerhaft dem Nutzer anlasten („das Problem sitzt vor dem Bildschirm"), aber die Funktion erlaubt, per Deklaration der *Typen* von `e` und `a` als `float`, auch negative Eingaben, für die sie eben nicht funktioniert. Sie ist also nicht korrekt (genau spricht man hier von mangelnder *totaler Korrektheit*, die die *mangelnde Terminierung* mit einschließt), was allerdings leicht durch Hinzufügen der beiden Vorbedingungen $a \geq 0$ und $e > 0$ behoben werden kann (wobei wie gesagt $e > 0$ die Fehler der Gleitkommaarithmetik nicht berücksichtigt — e darf nicht beliebig nah an 0 sein).

18.3.2 Falsifikation von Programmen: Testen

Wie bereits bei den Algorithmen in Abschnitt 17.3 erwähnt lässt sich die *Korrektheit eines Programms* zwar durch Testen nicht (oder nur bei trivialen Programmen) belegen, dafür aber widerlegen. Dafür muss man „nur" Eingaben finden, die die Vorbedingungen erfüllen und die zu Ausgaben führen, die die Nachbedingungen verletzen. Das effektive Testen (das in der Praxis dazu verwendet wird, sich der Korrektheit eines Programms anzunähern) ist eine eigenständige Disziplin der Softwareentwicklung und nebenbei eine Wissenschaft für sich; auch wenn es hier nicht weiter Thema sein soll, will ich nicht unerwähnt lassen, dass sich aus Vor- und Nachbedingungen relevante Tests ableiten lassen (und das Testen ohne Kenntnis der Vor- und Nachbedingungen nur Programmabstürze finden kann). Testen wird durch Werkzeuge unterstützt (s. Abschnitt 21.2).

18.3.3 Defensive Programmierung

Wie wir gesehen haben, erlaubt zwar die Angabe von *Typen* wie `float` für *Ein-* und *Ausgabeparameter* von Funktionen eine gewisse Einschränkung der Werte und damit die Formulierung bestimmter, einfacher Vor- und Nachbedingungen, aber erstens verwenden nicht alle Programmiersprachen Typen und zweitens sind diese allein häufig nicht ausreichend, um alle Vor- und Nachbedingungen abzudecken. In der sog. **defensiven Programmierung** würde man daher zu Beginn einer Funktion im Programm eine Überprüfung der Eingaben vornehmen und bei fehlerhaften Eingaben (das sind Eingaben, die die Vorbedingung nicht erfüllen) den Dienst der Funktion verweigern. Dies würde jedoch erst zur *Laufzeit* des Programms greifen und es stellt sich dann die große Frage, wie man auf diese Dienstverweigerung reagieren soll. Ziel der Verifikation ist hingegen, die Korrektheit eines Programms (Freiheit von Fehlern) nachzuweisen, ohne es auszuführen. Dennoch ist die defensive Programmierung sicher besser als nichts, zumindest solange man nicht ein falsches Ergebnis keinem Ergebnis vorzieht.

18.3.4 Design by Contract

Zwar dienen *Vor-* und *Nachbedingungen* sowie *Invarianten* eigentlich der der Programmierung vorgelagerten Spezifikation und der formalen Verifikation eines Algorithmus oder Programmes (Kapitel 17), aber so, wie man die Einhaltung der Vorbedingungen im Zuge der defensiven Programmierung durch das Programm selbst überprüfen lassen kann, kann man

auch Nachbedingungen und Invarianten zur *Laufzeit* eines Programms prüfen. Werden sie verletzt, kann das Programm mit einer entsprechenden Fehlermeldung abgebrochen werden und man weiß zumindest ungefähr, wo man den Fehler zu suchen hat.

Redundanz deckt Fehler auf

Tatsächlich möchte ich, wie schon bei der Entwicklung von Algorithmen (Kapitel 16), bei der Entwicklung von Programmen dringend empfehlen, sich schon beim Programmieren (und nicht erst hinterher!) ausgiebige Gedanken über Vor- und Nachbedingungen sowie Invarianten zu machen und eine Überprüfung derselben in die Programme einzubauen (vgl. Abschnitt 17.2). Zwar ist diese Arbeit für ein fehlerfrei funktionierendes Programm überflüssig, aber da kaum ein Programm fehlerfrei ist, ist dies eine „Best practice", gegen die sich nur schwer argumentieren lässt — *Redundanz* ist letztlich der einzige Weg, um Fehler innerhalb des Systems (also ohne ein Orakel von außen) zu finden.

18.4 Vom Problem zum Programm: funktionale Dekomposition

Längst nicht immer führt der Weg von einem Problem zu einem Programm über die vorherige Formulierung eines *Algorithmus*: Besteht die Schwierigkeit eher im Umfang der durch ein Programm umzusetzenden Funktionen denn in ihrer algorithmischen *Komplexität*, dann wird man versucht sein, die Funktionen und Daten, die in einer Problemstellung (Spezifikation oder Lastenheft) enthalten sind, direkt in ein Programm zu übersetzen. Der oder die Algorithmen, die das Programm dann umsetzt, bleiben bei dieser Vorgehensweise implizit (oder werden, wenn man überhaupt von Algorithmus sprechen will, mit dem Programm gleichgesetzt; vgl. dazu die Abgrenzung von Algorithmus und Programm zu Beginn dieses Kapitels).

In der Regel sind die Funktionen, die eine Problemstellung zu ihrer Lösung benötigt, aber trotzdem zu komplex, um sie „mal eben" (wie im Beispiel der Funktion `Wurzel`) als eine lineare Folge von Anweisungen hinzuschreiben. Das gilt bereits für das Beispiel vom Sortieren aus Kapitel 16. In solchen Fällen hat es sich bewährt, das Problem *rekursiv* in immer kleinere Teilprobleme zu zerlegen, bis sich die einzelnen Teile gut, d. h. einfach und leicht nachvollziehbar, lösen lassen. Die Lösung des Gesamtproblems ergibt sich dann als Zusammensetzung der Lösung der Teilprobleme. Wenn die Zerlegung anhand der Funktionen, die ein Programm bieten soll, erfolgt, spricht man auch von **funktionaler Dekomposition**. (Eine andere Dimension der Dekomposition bilden die Daten eines Programms; s. Abschnitt 19.4 sowie Kurseinheit 4.) Das Prinzip der (funktionalen) Dekomposition kommt auch beim Algorithmenentwurf zur Anwendung, verlangt dann aber eine etwas formalere Darstellung der Algorithmen als noch in Kapitel 16.

Um ein Beispiel für die funktionale Dekomposition zu geben, nehmen wir uns das Sortierproblem vor und lassen uns von dem Algorithmus aus Kapitel 16 inspirieren. Zunächst definieren wir dazu eine Funktion mit dem Kopf

```
36 procedure sortiere(var l : Liste);
```

und vereinbaren, dass die Länge der Liste stets eine Zweierpotenz sei. Sodann zerlegen wir im Rumpf der Prozedur das Problem der Sortierung wie folgt:

```
37 begin
38   var l1, l2 : Liste;
39   teile(l, l1, l2);
40   if länge(l1) > 1 then begin
41     sortiere(l1);
42     sortiere(l2)
43   end;
44   mische(l1, l2, l)
45 end
```

Die Funktion `sortiere` ruft demnach die Funktionen `teile` und `mische` auf, die jeweils ein (einfacheres) Teilproblem lösen. Die erste halbiert die Liste `l` in die beiden (gleich langen) Teillisten `l1` und `l2` und die zweite setzt die beiden Teillisten wieder zusammen, und zwar so, dass `l` sortiert ist, wenn dies auch `l1` und `l2` sind. Damit `l1` und `l2` sortiert sind, ruft sortiere sich zuvor zweimal selbst auf, und zwar mit den beiden neuen Teillisten `l1` und `l2` als Argument. Man beachte, dass sortiere hier nicht als Funktion, sondern als Prozedur definiert ist, deren einziger Parameter `l` gleichzeitig der Ein- und der Ausgabe dient. `sortiere` liefert also keine neue, sortierte Liste, sondern die alte in sortiertem Zustand. Dass `l` nicht nur *Ein-*, sondern auch *Ausgabeparameter* der Prozedur `sortiere` ist, wird durch Voranstellung des Schlüsselworts `var` (Zeile 36) gekennzeichnet; bei `teile` (deren Deklaration hier nicht gezeigt wird) sind `l` die Eingabe und `l1` und `l2` die Ausgaben.

Natürlich funktioniert die Prozedur `sortiere` solange nicht, bis auch die Prozeduren `teile` und `mische` definiert wurden. Diese können wiederum auf einfachere Teilprozeduren oder -funktionen zurückgreifen, die dann wieder definiert werden müssen, usw., bis nur noch bereits vorhandene Prozeduren oder Funktionen (wie beispielsweise `länge`) aufgerufen werden. Man beachte, dass das auf diese Weise entstandene Programm zwar vom Algorithmus aus Kapitel 16 inspiriert wurde, sich aber auch fundamental von diesem unterscheidet: Anstatt wie der Algorithmus stets abwechselnd zu teilen und wieder zusammenzuführen, teilt es erst *rekursiv* immer weiter auf und fügt dann (in umgekehrter Reihenfolge) alles wieder zusammen. Dafür braucht es mehr als nur drei Listen (für jeden rekursiven Aufruf zwei neue), im Gegensatz zu dem Algorithmus, der mit drei Stapeln auskam.

Achtung top-down!

Bei der funktionalen Dekomposition handelt es sich um eine Top-down-Vorgehensweise. Sie setzt voraus, dass die nachfolgende, weitere Zerlegung eines Problems (oder seiner Lösung) nicht unter vorausgegangenen Entscheidungen leidet. Dies ist immer dann ein Problem, wenn bei der Zerlegung das Problem noch nicht zur Gänze verstanden wurde, man also nur hoffen kann, dass die Zerlegung im weiteren Verlauf tragfähig ist, sich dort also keine starken Zusammenhänge zwischen Teilen, die man ursprünglich einmal für gut trennbar gehalten hatte, ergeben. Dafür gibt es aber keine Garantie und so kann das Scheitern eines Projekts (das Lösen eines gegebenen Problems) schon in seinem Anfang begründet sein, nämlich wenn aufgrund der Größe des Projekts und seines Fortschritts eine Revision anfänglicher Entscheidungen unmöglich scheint und man sich genötigt sieht, an

ihnen festzuhalten. So verlangt die funktionale Dekomposition Erfahrung und ein gutes Händchen.

Everything should be built top-down, except the first time.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

18.5 Ausführung von Programmen auf einem Computer

Damit ein Computer Programme, die in einer Programmiersprache geschrieben wurden, ausführen kann, müssen die Programme in *Maschinenbefehle* (s. Abschnitt 11.5 und Abschnitt 9 in Kurseinheit 1) umgesetzt werden. Dazu haben sich im Wesentlichen zwei alternative Vorgehensweisen etabliert:

- die **Interpretation** von Programmen und
- die **Übersetzung** von Programmen.

Sowohl die Interpretation als auch die Übersetzung eines Programms werden dabei von einem (anderen) Programm vorgenommen. Zur Programmierung in einer Programmiersprache ist also neben dem Editor mindestens ein weiteres Programm erforderlich, nämlich

- ein **Interpreter** für die Interpretation von Programmen oder
- ein **Compiler** für die Übersetzung von Programmen.

Da ihre Eingaben (und im Falle des Compilers auch ihre Ausgaben) Programme sind, nennt man diese Programme auch **Metaprogramme**.

Naturgemäß läuft der Interpreter *während* der Ausführung des Programmes und der Compiler *davor*. Wichtiger aber ist, dass ein übersetztes (kompiliertes) Programm auch auf einem anderen Computer ausgeführt werden kann, während für ein interpretiertes Programm der Interpreter auf demselben Computer laufen muss — es sind also zur Ausführung stets zwei Programme vonnöten, die auf dem Computer installiert werden müssen. Die Installation eines Interpreters kennen Sie vielleicht von der Programmiersprache *Java*, deren Programme auf einer sog. **virtuellen Maschine**, der **Java Virtual Machine** (**JVM**), laufen, die vor der Ausführung von Java-Programmen auf Ihrem Rechner auf diesem installiert werden muss. Bei dieser virtuellen Maschine handelt es sich im wesentlichen um einen Interpreter für sog. **Bytecode**, eine Art von *Maschinencode*, der aber keine echte *Hardware* (wie einen bestimmten Prozessortyp), sondern eben eine virtuelle anweist. Dabei ist der Java Bytecode das Ergebnis der Übersetzung von Java-Programmen. Der Vorteil dieser (etwas umständlich anmutenden) Vorgehensweise ist, dass in Java Bytecode übersetzte Programme auf allen Computertypen laufen, für die eine JVM verfügbar ist.

19 Programmierparadigmen

Während man Programmiersprachen früher noch in **Generationen** einteilte (erste bis fünfte, wobei man den Eindruck haben kann, dass mit der dritten Generation die Generationenfolge im Wesentlichen abgeschlossen war), unterscheidet man heute eher sog. **Programmierparadigmen**. Diese sind tatsächlich so unterschiedlich, dass die Verwendung des etwas vollmundig anmutenden Begriffs des Paradigmas durchaus angemessen scheint.

Die heute gelehrten Programmierparadigmen sind

1. die imperative Programmierung,
2. die funktionale Programmierung und die
3. logische Programmierung,

wobei die letzten beiden auch gern zur **deklarativen Programmierung** zusammengefasst (und damit von der imperativen abgegrenzt) werden. Die heute weit verbreitete *objektorientierte Programmierung* hat bei manchen Autoren ebenfalls den Status eines Paradigmas, aber die meisten objektorientierten Programmiersprachen sind imperativ (mit Anleihen aus der funktionalen Programmierung), so dass die Aufnahme der objektorientierten Programmierung in den Kanon der Paradigmen die ansonsten überlappungsfreie Einteilung zerstören würde. Trotzdem ist der objektorientierten Programmierung (genau wie der prototypenbasierten) nachfolgend ein eigener Abschnitt gewidmet.

19.1 Imperative Programmierung

Bei der **imperativen Programmierung** besteht ein Programm aus einer Folge von **Anweisungen** oder **Befehlen** (imperativ!). Sie ist damit am nächsten an der *Hardware*, die ja eine Folge von Befehlen in *Maschinensprache* ausführt. Allerdings befinden sich die Anweisungen imperativer Programmiersprachen (der dritten Generation) auf einem *höheren Abstraktionsniveau* als die der Maschinensprache. Das obige Beispiel zur Berechnung der Wurzel (Abschnitt 18.2, Zeilen 24–30) ist ein *imperatives Programm*.

A programming language is low level when its programs require attention to the irrelevant.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

Zu den Anweisungen der imperativen Programmierung zählen solche, die den Programmablauf steuern (sog. **Kontrollstrukturen** wie *Verzweigung*, *Wiederholung* oder *Schleife*

und *Prozeduraufruf*, zu dem hier auch der *Funktionsaufruf* zählt[60]), und solche, die den vom Programm verwalteten Speicher mit Werten belegen (*Wertzuweisung*; s. Abschnitt 18.2) oder diesen auslesen. Dabei kommt jedes Schreiben in den Speicher einem *Zustandswechsel des Programms* gleich. *Zustände* und Zustandswechsel sind typisch für die imperative Programmierung — sie kommen in der reinen funktionalen (Abschnitt 19.2) und logischen (Abschnitt 19.3) Programmierung nicht vor.

strukturierte Programmierung

Während frühe Vertreter der imperativen Programmiersprachen noch stark an Maschinensprache erinnerten (in *BASIC* beispielsweise gibt es noch Zeilennummern, die — wie *Programmspeicheradressen* — mittels des Go-to-Befehls einzeln angesprungen werden können), haben sich die heutigen Vertreter deutlich davon entfernt. So folgen heute praktisch alle imperativen Programmiersprachen den Regeln der **strukturierten Programmierung**, bei der jeder Programmabschnitt genau einen Eingang und ein Ausgang hat, so dass die lineare Folge der Anweisungen auf dem Bildschirm (im Editor) oder auf einem Blatt Papier (nach dem Ausdrucken des Programms) weitgehend dem Ablauf des Programms entspricht. Auf diese Weise soll sog. *Spaghetticode*, bei dem die Pfade durch ein Programm verschlungen sind (vor allem ausgelöst durch Sprunganweisungen wie das Go to[61]), verhindert werden. Spaghetticode ist für Menschen schlecht les- und wartbar.

WIKIPEDIA

WIKIPEDIA

Zustandsabhängigkeit als Problem

Nachdem durch die quasi-verpflichtende Einführung der strukturierten Programmierung in allen imperativen Programmiersprachen dem Spaghetticode sein wohlverdientes Ende bereitet wurde, bleibt als Hauptkritikpunkt an der imperativen Programmierung die *Zustandsabhängigkeit* (die sie von der unterliegenden *Hardware* erbt und nicht, wie das funktionale oder logische Programmierparadigma, wegzuabstrahieren versucht). So kann man kritisieren, dass bei wiederholtem Aufrufen der Funktion

```
46 function WasDennNun : string;
47 begin
48   if status then WasDennNun := "ja" else WasDennNun := "nein";
49   status := not status
50 end
```

das Ergebnis zwischen "ja" und "nein" wechselt, wenn `status` eine *boolesche Variable* ist, die die Werte `true` und `false` annehmen kann und die ihren Wert zwischen zwei Proze-

---

[60] Der Prozeduraufruf und die Existenz von Prozeduren in Programmiersprachen hat auch zum Begriff der *prozeduralen Programmierung*, der in etwa gleichbedeutend mit dem der *imperativen Programmierung* ist, geführt. *Pascal-Programme* sind typische Vertreter dieser Gattung.

[61] Der Artikel „Go To Statement Considered Harmful“ von Edsger Dijkstra aus dem Jahr 1968 ist eine Ikone der Programmierung, wenn auch heute nur noch historisch interessant. Die strukturierte Programmierung wurde zwischenzeitlich zumindest insofern aufgeweicht, als dass man eine Schleife auch mittendrin beenden kann. Es dient dies der Umsetzung von sog. *n*+1/2-*Schleifen*, die Sie, falls Sie programmieren, entweder schon kennen bzw. sofort erkennen werden, wenn Sie das erste Mal eine brauchen.

duraufrufen behält (eine sog. **globale Variable**). Diese Zustandsabhängigkeit mag im gegebenen (konstruierten) Fall für den Aufrufer der Funktion überraschend sein, aber Zustand ist letztlich gleichbedeutend mit Gedächtnis und Zustandsunabhängigkeit würde bedeuten, vom Gedächtnis keinen Gebrauch zu machen, was den Nutzen der Programmierung doch eher einschränken würde. Es bleibt das Problem, dass Zustand den formalen Beweis der *Korrektheit von Programmen* (s. Abschnitt 18.3.1) erschwert, da man einem Funktionsaufruf allein nicht ansehen kann, was er zurückliefert — dafür muss man schon den Programmzustand und damit auch den bisherigen Programmablauf kennen. Außerdem verlangt Zustandsabhängigkeit an manchen Stellen Festlegungen, die bei der zustandsfreien Programmierung nicht notwendig sind: So ist für das Ergebnis des Prozeduraufrufs `SagEs(WasDennNun, HinOderHer)` bei gegebenem Programmfragment

```
51 procedure SagEs(Was : string, Warum : boolean);
52 begin
53   print(Was);
54   print(Warum)
55 end;
56 function WasDennNun : string;
57 begin
58   if HinOderHer then WasDennNun := "ja" else WasDennNun := "nein"
59 end;
60 function HinOderHer : boolean;
61 begin
62   status := not status;
63   HinOderHer := status
64 end;
```

die Reihenfolge, in der die Funktionen `WasDennNun` und `HinOderHer` (deren Werte der Prozedur als Parameter übergeben werden) aufgerufen und ausgeführt werden, maßgeblich.

didaktische Eignung?

Dessen unbeschadet erfreut sich die imperative Programmierung nach wie vor großer Beliebtheit. Dies mag daran liegen, dass sie die unmittelbarste Form der Programmierung ist, zumindest wenn man heutige Computer vor Augen hat: Man erteilt der Maschine Befehle, die diese dann ausführt. So sind auch viele *Algorithmen* wie der Sortieralgorithmus aus Kapitel 16 imperativ formuliert. Persönlich bin ich bis vor kurzem davon ausgegangen, dass die imperative Programmierung auch für Anfänger und Kinder am leichtesten zu begreifen ist und sich daher am ehesten dazu eignet, einen Zugang zur Programmierung zu vermitteln. Mittlerweile bin ich mir aber nicht mehr ganz so sicher, denn gerade die im Bereich der Datenauswertung vorkommenden Programme sind typischerweise imperativ um einiges schwieriger zu formulieren als funktional (als eine Folge — oder *Pipeline* — von Abbildungen, Filtern und Aggregationen). So wird denn auch insbesondere in den USA seit einiger Zeit ein anderes Heranführen an die Programmierung propagiert (und teilweise auch exerziert): das über die *funktionale Programmierung*. Man sollte aber (wie immer) vermeiden, über diese Frage in Glaubenskriege zu verfallen (eine Neigung, die dem einen oder anderen Programmierer nicht fremd sein dürfte).

Link

19.2 Funktionale Programmierung

Während die imperative Programmierung das Bild einer Maschine pflegt, die Befehle entgegennimmt und in Reaktion darauf ihren Zustand wechselt, orientiert sich die **funktionale Programmierung** am mathematischen Begriff der **Funktion**, die Eingaben, die **Funktionsargumente**, auf Ausgaben, die **Funktionswerte**, abbildet. Eine typische Funktion ist etwa (in einer gedachten, *Pascal* syntaktisch ähnlichen funktionalen Programmiersprache) durch

```
65 function Quadrat(x : float) : float = x * x;
```

gegeben, die eine Eingabe `x` auf ihr Quadrat `x * x` als Ausgabe abbildet. Dabei sind Funktionen nicht auf arithmetische Operationen beschränkt — logische etwa (wie Abschnitt 3.1 in Kurseinheit 1) und auch Zeichenkettenoperationen (wie in Abschnitt 4.4) sind ebenso Standard und Funktionen auf anderen Datentypen können vom Programmierer selbst hinzugefügt (programmiert) werden.

Anders als ≔ (oder, je nach Sprache, auch =) bei der *imperativen Programmierung* steht das Gleichheitszeichen in der obigen Funktionsdefinition (Zeile 65) nicht für eine *Wertzuweisung* (und damit auch nicht für eine *Zustandsänderung*), sondern für mathematische Gleichheit: Der Wert der Funktion `Quadrat`, auf eine Zahl `x` angewendet, ist gleich dem Ergebnis von `x * x` — genau, wie das Quadrat einer Zahl mathematisch definiert ist. Verschiedene *Funktionsanwendungen*, also etwa `Quadrat(2)` und `Quadrat(3)`, stehen für verschiedene (Funktions-)Werte, aber der Wert ist, anders als bei der Implementierung von `Wurzel` in Abschnitt 18.2, zu keinen Zeitpunkt in einer Variable gespeichert.

Bleibt die Frage, was man dann mit den Funktionswerten machen soll, wenn man sie nicht speichern kann. Antwort gibt die folgende Funktion zur Berechnung der Fakultät:

```
66 function Fakultät(x : integer) : integer =
67   if x = 0
68     then 1
69     else x * Fakultät(x - 1);
```

In Zeile 69 erkennt man, wie der Funktionswert (hier der Wert der Anwendung der Funktion `Fakultät` auf den Wert `x - 1`) in der Berechnung einer (anderen oder derselben) Funktion verwendet werden kann. Wird die definierte Funktion für ihre eigene Definition verwendet (wie im gegebenen Beispiel), spricht man auch von einer **rekursiven Definition**. Diese ist aber, wie wir in Abschnitt 18.4 gesehen haben, nicht der funktionalen Programmierung vorbehalten.

Zwar ist die rekursive Definition von `Fakultät` einfach (und irgendwie auch elegant — nicht umsonst ist sie *das* Standardbeispiel der funktionalen Programmierung), aber das gilt längst nicht für alle rekursiven Definitionen. So stellt sich beispielsweise die funktionale Version der

Berechnung der Wurzel, die sich einer Hilfsfunktion `WurzelR` (mit „R“ für „rekursiv“) bedient, im Gegensatz zur imperativen Version (s. Zeilen 24–30 in Abschnitt 19.1) wie folgt dar:

```
70 function Wurzel(a : float, e : float) : float = WurzelR(1, a, e);
71 function WurzelR(x : float, a : float, e : float) : float =
72   if abs(x * x − a) < e
73     then x
74     else WurzelR((x + a / x) / 2, a, e);
```

Die Definition von `Wurzel` legt zunächst fest, dass die Wurzel einer Zahl `a` mit Genauigkeit `e` gleich dem Wert der Funktion `WurzelR` auf die Zahl `1` mit `a` und `e` angewendet ist. Wie bei einem imperativen Programm können also sowohl Ein- als auch Ausgabewerte von Funktion zu Funktion weitergereicht werden (vgl. den Aufruf von `SagEs` in Abschnitt 19.1). Die Funktion `WurzelR` wiederum definiert ihren Wert mit einer *Fallunterscheidung* (`if … then … else …`): Wenn der Eingabewert `x` die Wurzel von `a` schon mit Genauigkeit `e` annähert, dann ist das Ergebnis gleich `x`; andernfalls ist es gleich dem Ergebnis von derselben Funktion `WurzelR`, angewendet auf den Mittelwert von `x` und `a/x` (wobei dieses Ergebnis wiederum nur weitergereicht und nicht in einer Variable gespeichert wird[62]). Diese *rekursive Funktionsanwendung*[63] ersetzt die *Schleife* aus dem imperativen Programm aus Abschnitt 18.2. Man beachte, dass die Rekursion mit Erreichung des Genauigkeitskriteriums (der `then` Zweig aus der Fallunterscheidung) beendet (terminiert) wird.

Trotz der mathematischen Begriffe, auf denen die funktionale Programmierung beruht, besteht doch eine gewisse Distanz zwischen einer mathematischen Formulierung des Problems und obigem funktionalen Programm mit seiner Rekursion in Zeile 74 und seinem *Abbruchkriterium* in Zeile 72. So schreibt man etwa schlicht

$$x = \frac{x + \frac{a}{x}}{2}$$

für die Fixpunktgleichung, die den Wert von $\sqrt{a}$ bestimmt, und die Definition einer Folge, die gegen $\sqrt{a}$ konvergiert, schreibt man als

$$x_n = \begin{cases} 1 & \text{für } n = 0 \\ \dfrac{x_{n-1} + \frac{a}{x_{n-1}}}{2} & \text{für } n > 0 \end{cases}$$

Beide verwenden zwar den gleichen Term $(x + a/x)/2$ wie die obige Implementierung der Funktion `Wurzel` (wie im übrigen auch die imperative Implementierung aus Abschnitt 18.2),

---

[62] Dies bezieht sich allein auf das funktionale Programm — dessen Übersetzung in *Maschinensprache* kann den Funktionswert schon irgendwo zwischenspeichern.

[63] Der (mathematisch motivierte) Begriff der *Funktionsanwendung* der funktionalen Programmierung entspricht dem des *Funktionsaufrufs* in der *imperativen Programmierung*.

unterscheiden sich aber ansonsten nicht unerheblich, schon weil die funktionale Implementierung (anders als bei der Fakultät) neben dem Rekursionsanfang (Schätzwert 1) noch ein Rekursionsende ($|x^2 - a| < \epsilon$) vorsehen muss. Es reicht also nicht, mathematisch zu denken, um funktional zu programmieren — man muss funktional denken und es ist zumindest nicht offensichtlich, ob das mathematische Denken imperatives oder funktionales Programmieren begünstigt.

Zustandslosigkeit

Dennoch hat die funktionale Programmierung starke mathematische Anleihen und kommt zunächst, wie die Mathematik, ohne den Begriff des *Zustands* (der ja ein *Vorher* und ein *Nachher* und damit den Begriff einer — diskreten — *Zeit* impliziert) aus. Die Korrektheit einer Funktion hängt damit allein an ihrer Definition (und der der Funktionen, die sie aufruft) und ihr Ergebnis ist stets unabhängig von Zeitpunkt oder Reihenfolge ihres Aufrufs, was die *parallele Programmierung* enorm erleichtert. Gleichwohl bieten die meisten funktionalen Programmiersprachen dennoch eine Form von Zustand, weswegen sie strenggenommen hybride Sprachen sind. Zugleich halten funktionale Elemente zunehmend Einzug in imperative Programmiersprachen (*objektorientierte* insbesondere), so dass man von einer Verheiratung der beiden Paradigmen sprechen könnte.

Funktionen höherer Ordnung

Eine Besonderheit der funktionalen Programmierung ist, dass Funktionen selbst als Argumente oder Ergebnisse von anderen Funktionen, sog. **Funktionen höherer Ordnung**, auftreten können. Da Funktionen (und damit Programme) so zu Daten werden, spricht man hier auch von **Metaprogrammierung**. Metaprogrammierung ist eine sehr mächtige Technik, die jedoch die Analyse von Programmen (und damit auch deren *Korrektheitsbeweise*) erschwert und die nicht zuletzt deswegen mit Vorsicht zu genießen ist. Sie ist nicht auf die funktionale Programmierung beschränkt, sondern kommt in allen Programmierparadigmen zum Einsatz (insbesondere auch in der logischen Programmierung).

19.3 Logische Programmierung

Kurs

So wie die funktionale Programmierung auf dem mathematischen Begriff der Funktion basiert, basiert die **logische Programmierung** auf der (mathematischen oder formalen) *Logik*, genauer auf der *Prädikatenlogik* (s. Abschnitt 3.3.2 in Kurseinheit 1). Dabei wird ausgenutzt, dass sich bestimmte *logische Ausdrücke* als Regeln auffassen lassen, die wiederum als Programme interpretiert werden können. Die bekannteste (und am weitesten — wenn nicht einzige — verbreitete) *logische Programmiersprache* ist **Prolog** (abgeleitet aus „Programmation en Logique“).

WIKIPEDIA

Ein einfaches logisches Programm besteht aus den beiden Regeln (in Prolog-Syntax)

```
mensch('Sokrates').
sterblich(X) :- mensch(X).
```

Bei der ersten (in Zeile 75) handelt es sich um eine degenerierte Regel, bei der die *Prämisse* fehlt; die *Konklusion* `mensch('Sokrates')` wird als unbedingt wahr angenommen und stellt das *Fakt* „Sokrates ist Mensch“ dar. Die zweite (Zeile 76) ist eine echte Regel: Sie entspricht einer *Implikation* von rechts nach links und wird als „*x* ist sterblich, wenn *x* Mensch ist“ gelesen. Hierbei ist *x* (bzw. `X` in Zeile 76) eine Variable, die für beliebige Terme stehen kann; `'Sokrates'` ist so ein Term (in Anführungszeichen, weil in Prolog alles, was außerhalb von Anführungszeichen steht und wie `X` mit einem Großbuchstaben beginnt, eine *Variable* ist). Mithilfe dieses Programms wird nun die *Anfrage* (engl. *query*)

```
77 :- mensch('Sokrates')
```

(ebenfalls eine degenerierte Regel, bei der allerdings die Konklusion fehlt) positiv beantwortet, da zu der Anfrage ein identisches Fakt existiert. Die Anfrage

```
78 :- sterblich('Sokrates')
```

wird ebenfalls positiv beantwortet; hierzu ist allerdings der *Schlussfolgerungsschritt*

$$\frac{\texttt{mensch('Sokrates')} \qquad \texttt{sterblich(X) :- mensch(X)}}{\texttt{sterblich('Sokrates')}}$$

notwendig, den die logische Programmierung mittels der sog. *Resolution* erledigt (vgl. die Schlussfolgerung mittels *Modus ponens* in Abschnitt 3.2.2; die Resolution verallgemeinert den Modus ponens). Dabei kann man den Schlussfolgerungsschritt wie die Abarbeitung einer Prozedur lesen: Der Prozeduraufruf `sterblich('Sokrates')` führt zur „Ausführung“ der Prozedur `sterblich(X)`, die wiederum die Prozedur `mensch(X)` aufruft, wobei hier die Variable `X` vorübergehend mit dem Wert `'Sokrates'` gleichgesetzt ist (in der logischen Programmierung spricht man hier von *Unifikation* der Variable). Die analoge Anfrage

```
79 :- sterblich(ich)
```

wird dagegen negativ beschieden, weil dem obigen Programm über mein Menschsein (oder das Menschsein des Terms `ich`) nichts bekannt ist und Prolog die sog. **Closed world assumption** verfolgt, also festlegt, dass alles, was nicht als wahr erwiesen ist, falsch sein muss.

Um herauszufinden, wer alles als Mensch bekannt ist, kann man die Anfrage

```
80 :- mensch(M)
```

stellen (zu lesen als „Gibt es einen Wert von `M`, so dass `mensch(M)` wahr ist?“); hierbei wird dann die Variable `M` der Anfrage mit der Variable `X` des Prädikats (Prozedurkopfs) `sterblich` gleichgesetzt (unifiziert). Die Anfrage würde bei obiger Faktenlage exklusiv mit

```
81 M = 'Sokrates'
```

beantwortet, da keine anderen Fakten vorliegen. Würde man das Fakt `mensch(ich)` hinzufügen, würde dieselbe Anfrage zunächst wieder `'Sokrates'` und erst auf Nachfrage

auch `ich` liefern. Diese zweite, alternative Antwort wird über das sog. *Backtracking* gefunden, das nach alternativen Regelanwendungen sucht (und das ein Merkmal der logischen Programmierung ist; vgl. dazu auch Abschnitt 27.4.2 in Kurseinheit 4).

Während obiges Beispiel nahelegt, dass die logische Programmierung gut geeignet ist, Systeme mit *künstlicher Intelligenz* zu bauen (s. Abschnitt 3.4 in Kurseinheit 1), so wird sie allgemeiner überall da geschätzt, wo eine Lösung nicht berechnet werden kann, sondern durch systematisches Ausprobieren (Backtracking) gesucht werden muss. Dennoch konnte sie sich auch für solche Probleme nicht wirklich durchsetzen (und weiterreichende Ansprüche muss man wohl — Stand heute — als gescheitert ansehen[64]).

Logik und Datenbanken

In bestimmten Nischen ist die logische Programmierung dennoch durchaus eine sehr interessante Alternative zur imperativen und funktionalen. So wird sie — Aufgrund der Ähnlichkeit der Regeln, die ein Programm darstellen, mit den Regeln einer *Grammatik* — gern bei der *Verarbeitung natürlicher Sprachen* (genauer: in der *Computerlinguistik*) verwendet. Zudem wird eine Teilsprache von Prolog, **Datalog**, in bestimmten Nischen als *Datenabfragesprache* (s. Abschnitt 30.1; Anfragen werden im Kontext von Datenbanken *Abfragen* genannt; beides heißt im Englischen *query*) eingesetzt. Wenn Sie also in einem Bereich unterwegs sind, in dem *Datenbanken*, *Sprachverarbeitung* und *Logik* (*künstliche Intelligenz*) unter einen Hut gebracht werden müssen, dann sollten Sie die logische Programmierung zumindest in Erwägung ziehen. Insbesondere kann ich mir gut vorstellen, dass die logische Programmierung als Vehikel der *Digital humanities* noch einmal eine Renaissance erleben wird.

Funktionen in der logischen Programmierung

Übrigens: Zwar gibt es in Prolog auch *Funktionen*, doch bleiben diese *uninterpretiert* (d. h., sie werden nicht ausgewertet). So bedeutet beispielsweise das Fakt `positiv(quadrat(X))`, dass das Prädikat `positiv` für alle möglichen Terme `quadrat(X)` wahr ist. Dabei kann `X` durch beliebige andere Terme, so auch Zahlen, ersetzt werden, ohne dass auch nur der Versuch gestartet würde, das Quadrat des Arguments auszurechnen (tatsächlich ist in Prolog, da es nicht *typisiert* ist, auch `quadrat('Sokrates')` ein gültiger Term). Um in Prolog das Quadrat einer Zahl zu berechnen, kann man folgendes schreiben:

```
82 berechne(quadrat(X), Y) :- Y is X * X.
```

Dabei hat `is` den Charakter einer Wertzuweisung, die Prolog ansonsten fremd ist.

---

[64] Nach dem Aufkommen der logischen Programmierung in den 70er Jahren ging man schnell davon aus, dass sie die Programmierung revolutionieren würde, und Japan startet sogar ein groß angelegtes *5th Generation Project* (wobei hier mit fünfter Generation wohl eher die Hardware gemeint war, Prolog aber dennoch eine maßgebliche Rolle spielen sollte). Daraus wurde jedoch nichts. Dennoch hat sich die scherzhafte Deutung des Namens „Prolog“ als „probably language of God“ gehalten.

Sie sehen hoffentlich, dass die Wahl des Programmierparadigmas ganz wesentlich von der Art des zu lösenden Problems beeinflusst werden sollte. Es ist daher wichtig, dass Programmierer die verschiedenen Programmierparadigmen kennen (und idealerweise auch schon damit gearbeitet haben), bevor sie sich für eines entscheiden.

19.4 Objektorientierte Programmierung

In den ursprünglichen *prozeduralen Programmiersprachen* wie *Pascal* etc. sind Programme aus Prozeduren und Funktionen zusammengesetzt und die Daten eines Programms seinen Prozeduren und Funktionen beigeordnet. In der **objektorientierten Programmierung** ist das Verhältnis umgekehrt: Ein Problem wird zunächst anhand der Struktur seiner Daten zerlegt und die Prozeduren und Funktionen, in der objektorientierten Programmierung **Methoden** genannt, dem Ergebnis dieser Zerlegung, **Klassen** genannt, beigeordnet. Da Klassen in der Regel nicht weiter zerlegt werden können und Methoden nur selten geschachtelt werden (können), sind objektorientierte Programme in der Regel flacher strukturiert als herkömmliche imperative Programme (die ja häufig *rekursiv funktional dekomponiert* und damit deutlich tiefer strukturiert sind; s. Abschnitt 18.4). Die ersten objektorientierten Programmiersprachen waren Simula und Smalltalk; heute verbreitete objektorientierte Programmiersprachen sind *C++*, *Java* und *C#* sowie, als Skriptsprache, *Python*.

Kurs

Klasse und Instanz

Die Klassen der objektorientierten Programmierung sind *Datentypen* (s. Kapitel 27 in Kurseinheit 4). Als solche haben sie Werte, oder *Instanzen* (Abschnitt 27.1), die (für die objektorientierte Programmierung namensgebend) **Objekte** genannt werden. Anders als die Werte beispielsweise *primitiver Typen* wie `integer` oder `boolean` sind die Instanzen von Klassen in der Regel dynamisch, d. h., sie müssen zur *Laufzeit* durch eine Anweisung im Programm (meistens `new`) erzeugt werden. Man spricht dann auch von der **Instanziierung** einer Klasse. Da die Instanzen Speicher belegen, müssen sie auch wieder entfernt werden, wenn sie nicht mehr gebraucht werden — je nach Programmiersprache geschieht dies entweder durch Anweisungen im Programm oder automatisch durch einen Prozess, der **Garbage collection** genannt wird.[65]

Vererbung

Der mangelnden rekursiven Zerlegbarkeit von Datentypen wird in der objektorientierten Programmierung die sog. **Vererbung** entgegengesetzt. Dieses in der Programmierung ursprünglich auf *Wiederverwendung* abzielende Konzept kommt auch in der *Wissensrepräsentation* (so den *semantischen Datenmodellen*; s. Abschnitt 28.2.4 in Kurseinheit 4) vor und beschreibt — nach heutiger Lesart — eher eine *Subsumtionsbeziehung*, wie sie etwa zwischen den Klassen `Lebewesen` und `Mensch` besteht, denn eine (genetische) Vererbung: Wenn Menschen Lebewesen sind, dann gilt alles, was für Lebewesen gilt, auch für Menschen. Wenn der Klasse `Lebewesen` also eine Methode `sterben` zugeordnet ist,

WIKIPEDIA

[65] Natürlich belegen auch die Werte anderer Typen Speicher, der wieder freigegeben werden muss. Der Unterschied ist vielmehr der, dass die Lebensdauer dynamisch erzeugter Werte nicht an die Struktur eines Programms gebunden und somit nicht automatisch klar ist, wann sie wieder entfernt werden können.

dann gibt es die Methode auch für die Klasse `Mensch`. Man nennt dann `Mensch` auch eine **Subklasse** von `Lebewesen` und `Lebewesen` eine **Superklasse** von `Mensch`; dabei impliziert die Subsumtionsbeziehung eine *Subtypbeziehung* der durch die Klassen definierten Datentypen. Vererbung bezeichnet dabei strenggenommen lediglich den Mechanismus der Übertragung von Methoden von einer Klasse auf ihre Subklassen, die durch das Bestehen einer Subsumtionsbeziehung begründet wird; entsprechend findet man in manchen objektorientierten Programmiersprachen Vererbung auch von der Subsumtionsbeziehung abgekoppelt (also Subsumtion ohne Vererbung und Vererbung ohne Subsumtion).

dynamisches Binden

Mit der Vererbung verwandt und für die objektorientierte Programmierung mindestens genauso prägend ist das sog. **dynamische Binden**. Damit wird ein Methoden-, Prozedur- oder Unterprogrammaufruf bezeichnet, dessen Ziel erst zur *Laufzeit* (also während das Programm läuft und nicht schon zur Übersetzungszeit) bestimmt wird. Die Idee dahinter ist, dass in einer Subklasse von der Superklasse geerbte Methoden *überschrieben* werden können. Da die Subsumtionsbeziehung (oder Subtypenbeziehung) erlaubt, dass überall dort, wo ein Objekt einer bestimmten Klasse (genauer: eines durch die Klasse bestimmten Typs) erwartet wird, auch Objekte ihrer Subklassen auftreten dürfen (das *Prinzip der Substituierbarkeit* der objektorientierten Programmierung), kann erst zur Laufzeit entschieden werden, welche Version einer Methode aufgerufen werden muss. Das dynamische Binden bietet somit eine besondere Form der *Verzweigung* in Programmen.

Objektorientierung und Datenbanken

Die Zerlegung eines Problems anhand seiner Daten scheint die objektorientierte Programmierung zunächst in die Nähe von Datenbanken (Kapitel 28 in Kurseinheit 4) zu rücken. Paradoxerweise (wenn nicht sogar tragischerweise) gibt es aber zwischen den heute am weitesten verbreitenden sog. *relationalen Datenbanken* und der objektorientierten Programmierung gravierende Unterschiede, die eine Speicherung von Objekten in relationalen Datenbanken außerordentlich umständlich machen. Schuld ist hier neben der Vererbung der Umstand, dass objektorientierte Datenstrukturen auf **Verweisen** (*Zeigern*, *Referenzen* oder *Pointern*; s. Abschnitt 27.2 in Kurseinheit 4) beruhen, während relationale Datenbanken wertbasiert sind (in ihnen werden nur primitive Werte wie Zahlen oder Texte gespeichert).

19.5 Prototypenbasierte Programmierung

Wie bereits angedeutet ist ein Reiz der objektorientierten Programmierung ihre Nähe zur *Wissensrepräsentation*. Tatsächlich ist der von ihr verwendete Klassenbegriff an *Aristoteles' Genus et differentia* angelehnt. Deren Universalität kann aber angezweifelt werden und so stellte schon der späte *Wittgenstein* der eindeutigen Klassifikation von Objekten *Familienähnlichkeiten* gegenüber. Demnach gibt es beispielsweise keine fest umrissene Klasse aller Spiele, sondern lediglich mehr oder weniger typische Spiele. Interessanterweise findet auch diese Auffassung innerhalb der objektorientierten Programmierung ihre Anhänger: Sie hat sich in Form der sog. **prototypenbasierten Programmierung** niedergeschlagen, deren

derzeit prominenteste Materialisierung die Skriptsprache *JavaScript* ist.[66] Ob die protypenbasierte Programmierung auch als objektorientiert bezeichnet werden kann ist Ansichtssache; sie basiert jedenfalls auch auf *Objekten*, *Vererbung* und *dynamischem Binden*, eben nur ohne Klassen (und ohne Subsumtion).[67]

20 Programmiersprachen

Programmiersprachen gibt es so viele, dass eine Zählung kaum möglich ist. Tatsächlich kommen die meisten ambitionierten Programmierer irgendwann einmal in Versuchung, ihre eigene Programmiersprache zu entwerfen, und so mancher gibt dieser Versuchung nach. Allerdings ist eine Programmiersprache ohne *Implementierung* (durch einen *Compiler* oder *Interpreter*; s. Abschnitt 18.5) und ohne umfangreiche Werkzeugunterstützung (s. Kapitel 21) bestenfalls von didaktisch-theoretischem Wert. Diese Werkzeuge in brauchbarer Qualität zu entwickeln ist aber mit erheblichem Aufwand verbunden, so dass die allermeisten Programmiersprachen das Embrionalstadium nicht überwinden und keine nennenswerte Verbreitung erfahren.

A language that doesn't affect the way you think about programming, is not worth knowing.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

20.1 Syntax und Semantik von Programmiersprachen

Die **Syntax** einer Programmiersprache legt fest, welcher Form gültige Programme genügen müssen. Dabei wird die Syntax der Programmiersprache durch eine *Grammatik* festgelegt, die, anders als bei natürlichen Sprachen, immer streng ausgelegt wird: Ein Programm, das den Syntaxregeln einer Programmiersprache nicht genügt, ist kein Programm dieser Sprache und kann nicht ausgeführt werden. Ebenfalls zurückgewiesen werden Programme, die zwar

---

[66] JavaScript ist eigentlich eine Sprache für die Einbettung von Programmfragmenten (Skripten) in Webseiten, um diese an die jeweilige Umgebung, in der sie angezeigt werden, dynamisch anpassen und auf Ereignisse dieser Umgebung reagieren zu können. JavaScript wird aber zunehmend wie eine Programmiersprache (genauer: wie Java, mit dem es allerdings nur den ursprünglichen Verwendungskontext, das Web, gemein hat) eingesetzt. Das ist in etwa so, als würde man anfangen, Verkehrsflugzeuge aus Papier zu bauen — man sollte das nicht tun, denn irgendwann folgt ein Erklärungsnotstand.

[67] Interessanterweise gibt es auch in der Natur keine Klassen und Vererbung basiert in der Tat auf Prototypen: Gene werden von Individuum zu Individuum vererbt. Auf der anderen Seite bietet es sich für die Programmierung nicht unbedingt an, der Natur mit ihrer unermesslichen Vielfalt nachzueifern — Klassen sorgen in der Regel für ein deutlich vereinfachtes, und damit leichter handhabbares, Weltbild.

syntaktisch wohlgeformt sind (die also der Grammatik genügen), die aber *semantische Regeln* der Sprache verletzen. Diese sind in etwa vergleichbar mit den Kongruenzregeln natürlicher Sprachen wie z. B. der, nach der das Subjekt und das Prädikat eines Satzes stets denselben Numerus (Singular oder Plural) haben müssen (Textverarbeitungsprogramme wie Word kennen solche Regeln und weisen Autoren auf einfache — manchmal nur vermeintliche — Grammatikfehler hin). Sofern diese Regeln vor der Ausführung eines Programms geprüft werden, fallen sie unter die sog. **statische Semantik** einer Programmiersprache; andernfalls sind sie Teil der **dynamischen Semantik**, zu der auch die Regeln gehören, die die Programmausführung bestimmen. Bei der dynamischen Semantik spricht man auch von der Bedeutung eines Programms; sie ist immer eindeutig und leitet sich aus seinem Aufbau ab.

20.2 Faktoren beim Programmiersprachenentwurf

Der Entwurf einer neuen Programmiersprache kann von verschiedenen, teilweise widersprechenden Zielen geprägt sein. Ein (inzwischen überholtes) Ziel ist, eine Programmiersprache so zu gestalten, dass sich Programme wie in einer natürlichen Sprache formulierte *Algorithmen* schreiben und lesen lassen (am Beispiel Cobol gut zu sehen); es ist allerdings fraglich, welche Vorteile dies tatsächlich hat und ob nicht die Nachteile (lange, wortreiche Programme und eine Banalisierung des Programmierens nach dem Motto „es ist ganz einfach, jeder kann das!“) überwiegen. Ein alternatives Ziel ist, eine Programmiersprache an eine in einer bestimmten Anwendungsdomäne gebräuchliche Notation anzulehnen (sog. **domänenspezifische Sprachen**). Dies ist durchaus erfolgversprechend, insbesondere wenn in dieser Domäne viel programmiert werden muss, so dass genügend Programmierer und Programme (und damit eine „kritische Masse“) zusammenkommen.

Redundanz

Während die Annäherung an eine natürliche oder eine Fachsprache das Ziel hat, das Programmieren zu vereinfachen und so Fehlerquellen bei der Programmierung zu vermeiden, setzen natürliche Sprachen für die Fehlervermeidung bei der Kommunikation auf **Redundanz**, also auf Information, die man weglassen könnte, wenn es keine Übertragungsfehler gäbe, mit deren Hilfe man aber Übertragungsfehler erkennen kann. Wenn beispielsweise an einem übermittelten (gesprochen oder geschriebenen) Satz grammatikalisch etwas nicht stimmt, obwohl sein Autor sich stets an die Grammatik der Sprache hält, dann ist bei der Übermittlung wohl etwas schiefgegangen und man fragt besser nach, was denn gemeint war. In Programmiersprachen gibt es das Konzept der Redundanz auch, allerdings weniger, um Übermittlungsfehler, sondern, um inhaltliche (semantische) Programmierfehler zu entdecken. So enthält beispielsweise der Programmausschnitt

```
83 var i : integer;
84 i := true;
```

einen semantischen Fehler, der sich daran festmacht, dass die Variable `i` per Deklaration in der ersten Zeile den *Typ* `integer` hat, also nur ganze Zahlen aufnehmen kann, ihr aber in

der zweiten Zeile der Wahrheitswert `true` zugewiesen wird (ein sog. *Typfehler*). Irgendetwas stimmt hier nicht und der Programmierer muss nochmal ran. Dabei ist die Deklaration der Variable in dem Programm

```
85 var i : integer;
86 i := 1;
```

redundant: Dass sie den Typ `integer` hat, kann man auch aus der Zuweisung des Wertes 1 ablesen. Redundanzfreie Programme sind kompakter, aber weniger sicher; Programmiersprachen, die Programmierer zur Redundanz zwingen, sind zwar unbeliebt (weil ihre Programme unnötig lang scheinen), aber sicherer.

einfache Umsetzung

Während die vorgenannten Eigenschaften einer Programmiersprache darauf abzielen, das Programmieren zu vereinfachen und Programmierfehler zu vermeiden, ist die Minimierung des Aufwands, der für die Werkzeugunterstützung einer Programmiersprache getrieben werden muss, ein konkurrierendes Ziel. Das fängt mit der *Syntax der Programmiersprache* an (die Programme, die ja, wie in Abschnitt 18.1 dargestellt, zunächst nur Zeichenketten oder Texte sind, sollen leicht und schnell in ihre Bestandteile zerleget werden können) und endet damit, dass die Programme ohne erheblichen zusätzlichen Aufwand (wie automatische Optimierungen) effizient, d. h. mit möglichst geringem Speicher- und Zeitbedarf ausgeführt werden können (Stichwort *Zero cost abstractions*). Solche Sprachen muten häufig — gerade für Anfänger — etwas kryptisch an; allerdings erleichtert eine geeignete Werkzeugunterstützung die Programmierung zum Teil erheblich und einfach zu implementierende Programmiersprachen sind nicht notwendigerweise auch ausdrucksarm (und deswegen umständlich).

20.3 Verbreitung von Programmiersprachen

Angesichts der großen Vielzahl von vorgeschlagenen (und auch zumindest prototypisch implementierten) Programmiersprachen wäre es interessant zu wissen, was einer Programmiersprache zu ihrer Verbreitung verhilft. Wie man an prominenten Beispielen (*Fortran*, *Cobol*, *BASIC*, *Algol*, *Pascal*, *C*, *Smalltalk*, *C++*, *Java*, *C#*, *JavaScript*) ablesen kann, sind dies häufig Gründe, die eher im sozialen oder kommerziellen als im technischen Umfeld zu finden sind, nämlich z. B. Verbreitung mit *Hardware* und/oder *Betriebssystem*, akademische Lehre, Marktmacht oder Verbreitung mit dem *Internet*. Technische Durchbrüche haben einer Sprache selten zum Erfolg verholfen; gute Ideen werden dann eher von anderen, bereits etablierten Sprachen im Rahmen deren Weiterentwicklung absorbiert und verbreiten sich dann auf diese Weise.

Ein anderes Kriterium für die Verbreitung von Programmiersprachen ist die Unterstützung einer Sprache durch

1. *Programmierwerkzeuge*,
2. *Standardbibliotheken* und *Frameworks* sowie

3. eine große *Community*.

Der Aufwand der Entwicklung brauchbarer Programmierwerkzeuge (jenseits von Compiler oder Interpreter) für eine Programmiersprache übersteigt häufig den Aufwand für die Entwicklung der Sprache selbst, ist aber für die Akzeptanz jenseits akademischer Zirkel immens wichtig. Standardbibliotheken enthalten Prozeduren und Funktionen, die in vielen Programmen benötigt werden, die jedoch nicht den Status eines Sprachkonstrukts (wie etwa Kontrollflussanweisungen, Definitionen von Datenstrukturen oder Zuweisungen) haben. Typische Beispiele hierzu sind spezielle Ein- und Ausgabefunktionen und Datenstrukturen wie beispielsweise `Datum` oder `Matrix`. Frameworks bilden fertige Programmgerüste, die von einer Programmierung nur noch mit anwendungsspezifischen Funktionen ergänzt werden müssen. Nur wenn Bibliotheken und Frameworks verfügbar sind, erreichen routinierte Programmierer schnell die Produktivität, die sie von sich und anderen Sprachen kennen. Ein häufig unterschätzter Erfolgsfaktor ist ein hinreichend großer Nutzerkreis: Nur wenn in einer Programmiersprache aktiv programmiert wird, werden Fehler in ihrem Entwurf und ihrer Implementierung evident (und behoben) und nur dann entwickelt sich das Wissen, das man braucht, um konkurrenzfähige Programme in der Sprache entwickeln zu können. So ersetzen heute Online-Foren wie „Stack Overflow" Lehrbücher und Lehrer gleichermaßen. Google kann aber nicht dein Freund sein, wenn es niemanden gibt, der sein Wissen über und seine Erfahrung mit einer Sprache in irgendeinem Forum zur Verfügung stellt — ohne eine *aktive Community* ist eine Programmiersprache dem Vergessen geweiht.

20.4 Klassifikation von Programmiersprachen

imperativ, funktional oder logisch

Wie bereits in Kapitel 19 angemerkt, lassen sich Programmiersprachen grob anhand der von ihnen unterstützen Programmierparadigmen einteilen. Am erfolgreichsten in praktischer Hinsicht sind dabei sicher die Sprachen, die auf die *imperative Programmierung* setzen, wozu auch die meisten *objektorientierten Sprachen* gehören, wobei letztere kommerziell eine immer größere Rolle spielen (im wesentlichen *C++*, *Java* und *C#*; in letzter Zeit auch sog. *Skriptsprachen* wie *Ruby* und *Python* sowie, obwohl nicht klassisch objektorientiert, *JavaScript*, die allesamt auch an Universitäten gelehrt werden). In akademischen Kreisen, besonders in Nordamerika, werden allerdings *funktionale* Sprachen (wie *Lisp*, *ML* oder *Scheme*) zumindest in der universitären Bildung bevorzugt; über die Praxistauglichkeit hatte ich mich ja schon ausgelassen. Von den *logischen Programmiersprachen* gibt es praktisch nur eine (*Prolog*); von jüngeren Wiederbelebungsversuchen (durch Verheiratung mit anderen Paradigmen bzw. deren Sprachen) habe ich zwar schon gehört, aber noch nichts gesehen.

kompiliert oder interpretiert

Eine andere Dimension der Unterteilung ist die danach, ob eine Sprache interpretiert oder kompiliert (übersetzt) wird (s. Abschnitt 18.5). Diese Unterscheidung taugt jedoch nur bedingt, da zum einen für viele Sprachen sowohl Compiler als auch Interpreter existieren und zum anderen die Übergänge fließend sind: Wie bereits in Abschnitt 18.5 erwähnt, wird beispielsweise Java zunächst kompiliert, der entstandene

Code (*Java Bytecode*) dann aber (von der *JVM*) interpretiert. Allgemein können auch Interpreter vor der Interpretation eines Programms bestimmte Analysen durchführen, um (wie ein Compiler) Fehler zu finden (s. u.) oder die Ausführung des Programms zu beschleunigen.

einfach oder komplex

Eine andere Unterscheidung der Programmiersprachen ist die nach dem Umfang ihrer Grammatik (*Syntax*), gelegentlich (und irreführenderweise) mit *Ausdrucksstärke* gleichgesetzt (irreführend, weil man mit allen ernstzunehmenden Programmiersprachen dasselbe ausdrücken kann; gemeint ist wohl eher, wie leicht man sich darin ausdrücken kann). ABAP (die Programmiersprache, in der die meisten SAP-Systeme geschrieben sind) hat angeblich 400 Schlüsselwörter, was sie schon in die Sphären reduzierter natürlicher Sprachen katapultiert, aber auch schon Cobol ist von dem Bestreben gekennzeichnet, für jede Operation eine syntaktisch eigene Ausdrucksform zu bieten. Modernere Sprachen gehen einen anderen Weg und setzen auf eine kleine, möglichst einfache Syntax; die Vielfalt der Ausdrucksmöglichkeit, die sich Programmierer wünschen, wird dann in *Bibliotheken* verlagert (Sammlungen von benannten Prozeduren und Funktionen, die in der Programmiersprache geschrieben sind und mit ihr zusammen ausgeliefert werden). So ist in älteren Sprachen wie beispielsweise *BASIC* `print` ein *Schlüsselwort* (also als Teil der Grammatik der Sprache fest vorgegeben), in moderneren Sprachen nur eine Bibliotheksfunktion (die beliebig umbenannt werden kann, ohne die Grammatik zu ändern). Tatsächlich ist die Verfügbarkeit guter (das heißt hinreichend erprobter und bewährter) Bibliotheken heute ein ganz wesentliches Kriterium für die Entscheidung für eine bestimmte Programmiersprache in einem gegebenen Projekt (s. Abschnitt 20.3).

Typisierung und Typprüfung

In frühen Programmiersprachen wie beispielsweise *Fortran* oder *BASIC* konnte man ein und derselben Variable Werte verschiedenen *Typs* zuweisen. Dies ließ zum einen die Natur der Maschine, die die Programme ausführt und die ja auch keine verschiedenen Arten, oder Typen, von Werten unterscheidet, durchscheinen, zum anderen hatte es pragmatische Gründe, nämlich dass es früher (wie in der Mathematik und den Naturwissenschaften) üblich war, in Programmen möglichst einsilbig zu bleiben, also insbesondere für Variablen stets (nur) einen Buchstaben zu verwenden. Es konnte also durchaus vorkommen, dass die Buchstaben ausgingen und derselbe Name nacheinander für verschiedene Dinge verwendet wurde, diese Dinge aber von so unterschiedliche Natur waren, dass die Werte, für die die Namen standen, verschiedene Typen hatten.

Nun steht eine Variable aber meistens für eine bestimmte Größe oder für ein bestimmtes Attribut und der Typ der Variable ist mit dieser Größe oder diesem Attribut festgelegt. So ist es nicht sinnvoll, einer Variable mit Namen „Gewicht“ einen Wahrheitswert, ein Zeichen oder gar eine Zeichenkette zuzuweisen; wird dies dennoch versucht, kann man mit hoher Wahrscheinlichkeit von einem Programmierfehler ausgehen. Die **Typprüfung** ist ein Mechanismus, der versucht, solche **Typfehler** zu finden und anzuzeigen. Wie potent die Typprüfung ist, hängt dabei maßgeblich von der Programmiersprache ab, also z. B. davon, ob sie vorschreibt, dass für alle Variablen ein Typ deklariert oder aus dem Programm eindeutig ableitbar sein muss.

Umgehung der Typprüfung

An einem Typ hängen nicht nur die Werte, die eine Variable annehmen kann, sondern auch die *Operationen*, die auf den Werten ausgeführt werden können. So wird beispielsweise verhindert, dass ein Wahrheitswert zu einer Zahl addiert wird. Gleichwohl kann so etwas manchmal sinnvoll sein, weswegen eine strikte Typprüfung gelegentlich als zu starr empfunden wird. Viele Programmiersprachen sehen daher Mechanismen vor, die Typprüfung bewusst zu umgehen. In manchen Sprachen müssen dann entsprechende Operationen explizit als *unsicher* („unsafe") gekennzeichnet werden.

statische vs. dynamische Typprüfung

Bei der Typprüfung unterscheidet man prinzipiell zwischen **statischer** und **dynamischer**. Bei der statischen Typprüfung wird vor der Ausführung eines Programms, in der Regel vom Compiler, nach Typfehlern gesucht, bei der dynamischen während der Ausführung. Extremisten würden behaupten, jede Programmiersprache wäre mindestens dynamisch typgeprüft, da ein Typfehler, so er denn überhaupt auftritt, sich immer irgendwie bemerkbar macht, jedoch ist eine Prüfung nur dann sinnvoll, wenn sie im Fehlerfall auch eine Fehlermeldung produziert, die geeignet ist, den Fehler im Programm leicht zu lokalisieren. Eine solche dynamische Prüfung, die vom Programm selbst (bei kompilierten Programmen) oder vom Interpreter vorgenommen werden muss, verlangsamt jedoch die Ausführung des Programms, was allein schon für eine statische Typprüfung spricht. Zudem findet die statische Typprüfung Fehler in einem Programm, bevor es ausgeführt wird, was leicht erkennbar von hohem Wert ist. Andererseits beschränkt eine vollständige statische Typprüfung die Freiheiten in der Programmierung, so dass aktuelle Programmiersprachen wie *Java* oder *C#* eine Kombination statischer und dynamischer Typprüfung vorsehen.

statische Typprüfung als formale Verifikation

Übrigens: Die statische Typprüfung ist zum derzeitigen Stand die einzige Form der *formalen Verifikation* von Programmen, die breiten Einzug in die Programmierpraxis gehalten hat (s. dazu auch Abschnitt 18.3.1). Ein Programm, das die statische Typprüfung besteht, hat garantiert keine Fehler der Art, die die Prüfung aufzudecken erlaubt. Doch obwohl der Nutzen von Typen und strenger Typprüfung für die Entwicklung fehlerarmer Programme unbestritten ist, finden *schwach typisierte* Sprachen (wie *JavaScript*, *PHP*, *Python* oder *Ruby*) massive Verbreitung. Erklärt wird dies häufig mit einer größeren Produktivität, konkret weil ein strenges Typsystem Programmierern „nur im Weg" sei; dieselben fahren dann wohl auch Auto, ohne sich anzuschnallen (solange man keinen Unfall hat, ist der Gurt auch nur im Weg).[68]

seriell oder parallel

Wie wir bereits in Abschnitt 12.6 von Kurseinheit 2 gesehen haben, erlaubt es das Betriebssystem eines Computers, mehrere Programme tatsächlich oder schein-

---

[68] Fairerweise muss man hinzufügen, dass es auch möglich ist, in Programmiersprachen mit schwacher Typprüfung oder ganz ohne Typprüfung fehlerarme Programme zu schreiben. So setzte Apple beispielsweise bis vor kurzem auf die Programmiersprache Objective-C, die in puncto Typprüfung eher als schwach anzusehen ist. Jedoch erfordert die Verwendung solcher Sprachen ungleich mehr Hingabe und Disziplin, zwei Eigenschaften, die unter Programmierern auch nicht häufiger anzutreffen sind als unter anderen Menschen.

bar gleichzeitig auszuführen, und zwar unabhängig davon, wie viele Prozessoren der Computer besitzt. Diese Programme müssen nichts miteinander zu tun haben, können aber, über Mechanismen des Betriebssystems oder gemeinsam genutzte Ressourcen, miteinander kommunizieren.

Dieselben Gründe, die die parallele Ausführung mehrerer Programme wünschenswert machen, können auch in einem einzelnen Programm vorliegen. So können in einem *parallelen Programm* beispielsweise bereits erfolgte Eingaben verarbeitet werden, während das Programm auf neue wartet. Hinzu kommt, dass moderne Computer meistens mehrere Prozessoren oder Prozessorkerne haben, die man nutzen kann, um die Ausführung eines Programms durch echte Parallelisierung zu beschleunigen. Es ist also sinnvoll, Parallelität auch innerhalb eines Programms zuzulassen.

Leider entpuppt sich die Parallelität von Programmen als eine erhebliche Herausforderung für Programmierer und Programmiersprachenentwickler gleichermaßen, so dass die meisten aktuellen Programmiersprachen zwar „auch" parallel sind, aber Parallelität selten eine zentrale Eigenschaft ist. Auch werden die meisten Programme immer noch „erst einmal" als serielle entwickelt; Parallelität wird später hinzugefügt, wenn sie sich als notwendig erweist. Eine erwähnenswerte Ausnahme ergibt sich aus der Verwendung von sogenannten *Anwendungsframeworks*, die Parallelität kapseln und in die die Programmiererin ihre (weitgehend sequentiell ausgelegten) Funktionen einhängt. Die automatisierte Parallelisierung serieller Programme (durch Compiler oder virtuelle Maschine) fällt dagegen eher unter den Begriff der (automatischen) Optimierung von Programmen.

20.5 Abstraktion und essenzielle vs. akzidentelle Komplexität

Wie wir bereits in Kapitel 19 gesehen haben, hat die Wahl des Programmierparadigmas einen erheblichen Einfluss auf die Lösung eines Problems in Form eines Programms. Dies gilt ebenso für die Wahl der Programmiersprache. Während das Paradigma die *Kernabstraktionen* (Befehle, Funktionen oder Regeln) festlegt, wirbt jede einzelne Programmiersprache mit ihrem eigenen Satz von Abstraktionen um die Gunst der Programmierer.

Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

essenzielle Komplexität

Sieht man einmal von (absolut nicht zu vernachlässigenden!) Randbedingungen wie Beherrschung einer Sprache und Einbettung in einen Kontext ab, wählt man die Programmiersprache am besten so, dass ihre Abstraktionen beim zu lösenden Problem zur Geltung kommen. Passen Problem und Abstraktionen der Programmiersprache optimal zusammen, so erhält man kompakte Programme, deren Komplexität

ausschließlich **essenziell** ist, d. h., die nur komplex sind, weil das Problem selbst ein komplexes ist. Diese Komplexität lässt sich nicht beseitigen, wenn man das Problem zur Gänze lösen will.

akzidentelle Komplexität

Passen die Abstraktionen der Programmiersprache nicht zum Problem, dann erhält man Programme, deren essenzielle Komplexität durch eine **akzidentelle** überlagert wird. Akzidentell komplexe Programme wirken aufgebläht und sind mühsam zu schreiben und zu lesen; ihr Text ist durchsetzt mit umständlich anmutenden Ausdrücken, die bei Autoren wie Lesern Gefühle wie „hätte man da nicht besser die Sprache X genommen" auslösen. Das Problem ist allerdings meistens, dass wenn man die Sprache X genommen hätte, sich die akzidentelle Komplexität an anderer Stelle (im selben Programm) materialisieren würde. Die wenigsten realen Programmieraufgaben lassen sich nämlich eindeutig und ausschließlich einem Paradigma zuordnen und der Grund für die Existenz so vieler Programmiersprachen ist, dass jede ihre eigenen Stärken und Schwächen hat.

Nun könnte man meinen, akzidentelle Komplexität ließe sich vermeiden, indem man alle (wichtigen) Programmierabstraktionen in einer Sprache vereint. Solche Versuche gibt es in der Tat (die Programmiersprache *Scala* ist ein gutes aktuelles Beispiel hierfür), aber die Sprachen selbst werden dadurch aufgebläht und schwieriger weiterzuentwickeln und zu lernen. Letztlich ist die Wahl der Programmiersprache immer ein Kompromiss, der, wie eingangs erwähnt, häufig von anderen Einflussfaktoren als der besonderen Eignung einer Sprache für ein gegebenes Problem dominiert ist.

There will always be things we wish to say in our programs that in all known languages can only be said poorly.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

21 Programmierwerkzeuge

Wie bereits in Abschnitt 18 erläutert braucht man zum Programmieren in einer Programmiersprache neben einem Editor mindestens noch einen Compiler oder einen Interpreter. Damit ist es dann möglich, ein Programm einzugeben und auf einem passenden Computer auszuführen. Programmierer wären aber keine Programmierer, wenn sie nicht ständig auch über den Prozess des Programmierens nachdenken und ihn mit weiteren Programmen zu unterstützen suchen würden. Diese weiteren Programme — *Metaprogramme* — nennt man **Programmierwerkzeuge**.

21.1 Debugger

Beobachtungen legen nahe, dass Programmierer mehr Zeit damit verbringen, die Fehler in ihren Programmen zu suchen als die Programme zu schreiben (zu programmieren). So oder so ist Fehlersuche und -behebung ein fester Bestandteil im Leben von Programmierern. Man nennt diesen Bestandteil **Debugging**. Er kann *erhebliche Frustration* auslösen.

Steinzeit-Debugging

Eine immer noch sehr weit verbreitete Methode, Fehler zu finden, ist es, Ausgabebefehle (sog. Print statements) in ein Programm einzustreuen, bei deren Ausführung ein Hinweis auf den Ausführungsstand (z. B. Zeilennummer oder interessierende Variableninhalte) des Programms ausgegeben werden. Der Vorteil dieser Methode ist, dass man kein zusätzliches Werkzeug braucht, dessen Bedienung man ggf. erst erlernen muss. Nachteile sind, dass diese Ausgabebefehle später wieder entfernt werden müssen, dass man sich vor der Ausführung des Programms festlegen muss, was man sehen will, und dass man den Programmablauf damit weder anhalten noch beeinflussen kann. Die Fehlersuche wird so zu einem äußerst mühsamen Geschäft.

Werkzeugunterstützung

Eines der wichtigsten Programmierwerkzeuge ist daher der sog. **Debugger**. Er erlaubt es, ein Programm in kleinen Schritten auszuführen und nach jedem Schritt den Zustand des Programms zu inspizieren, also beispielsweise die Werte von Variablen einzusehen. Auf diese Weise können der Ablauf des Programms und seine Wirkung im Detail studiert werden, ohne dass man sich vorher einen Plan machen muss, was einen interessiert. Wichtig ist dabei allerdings, dass die Schritte des Programms auf Ebene des *Quellcodes* angezeigt werden und nicht auf Ebene des *Maschinencodes*, der ja bei kompilierten Programmen anstelle des Quellcodes ausgeführt wird. Solche sog. **Source-level Debugger** sind aufwendiger herzustellen und verlangen in aller Regel, dass ein Programm speziell fürs Debuggen kompiliert wird (um eine Zuordnung des Maschinencodes zum Quellcode zu ermöglichen). Viele Debugger erlauben dem Programmierer außerdem, den Zustand des Programms während seiner Ausführung zu verändern, also beispielsweise aktuelle Variablenwerte mit neuen zu überschreiben.

21.2 Testwerkzeuge

Um einen Fehler zu finden, muss man erst einmal wissen, dass er da ist. Sieht man einmal von statischen Prüfmethoden (wie der *Typprüfung*) ab, erlangt man Kenntnis von Fehlern in der Regel durch beobachtetes Fehlverhalten eines Programms: Es verhält sich nicht so wie verlangt. Um das zu beobachten, muss man das Programm ausführen und mit Eingaben füttern. Liefert es dann falsche oder gar keine Ausgaben (letzteres weil es hängenbleibt oder abstürzt), dann ist das Programm wohl fehlerhaft.

Die gezielte Provokation von Fehlverhalten eines Programms nennt man **Testen**. Zum systematischen Testen müssen neben Standardeingaben, die Standardabläufe des Programms bedingen, auch ungewöhnliche Eingaben verwendet werden. Da es in der Regel unmöglich ist, alle möglichen Eingaben durchzuprobieren (s. Abschnitt 17.3), haben sich Heuristiken

etabliert, welche Eingabewerte und -kombinationen man, bei Kenntnis oder Unkenntnis des Programmtextes, ausprobieren sollte.

Eine besonders erwähnenswerte Form des Testens ist das sog. **Regressionstesten**, das sich in der Regel automatisch durchführen lässt und das nach jeder (größeren) Änderung eines Programms laufen sollte, um zu überprüfen, ob das, was vorher funktionierte, hinterher auch noch funktioniert. Denn die Wahrscheinlichkeit, mit einer Änderung Fehler einzubringen, ist groß.

Tests als Spezifikation

Das Problem der mangelnden *Verifizierbarkeit* von Programmen gemäß ihrer *Spezifikation* mittels Testens (s. Abschnitt 18.3) lässt sich beseitigen, indem man die Tests zur Spezifikation erhebt. Nach dem sog. *Tests-first-Ansatz* (auch *test-getriebene Entwicklung* genannt) werden zuerst die Tests für ein Programm geschrieben und dann das Programm. Man könnte dann meinen, ein Programm sei fertig, wenn es alle seine Tests erfüllt — wenn dies tatsächlich so wäre, dann könnte man das Programm allerdings durch seine Tests ersetzen (denn dann bräuchte man ja das Programm nicht mehr, um Eingaben Ausgaben zuzuordnen). Stattdessen braucht man weitere Gründe, warum man glauben soll, dass das Programm sich auch bei anderen als den getesteten Ein- und Ausgabepaaren wohlverhält. In der Regel wird man darauf vertrauen müssen, dass der Programmierer das Problem, dass durch die Tests nur fallweise beschrieben wird, zur Gänze verstanden und mit seinem Programm eine gültige Lösung auch für die nicht in den Tests genannten Fälle gefunden hat. Dazu braucht es jedoch nicht nur die Bereitschaft zum Mitdenken, sondern auch profundes Wissen aus dem Kontext der Aufgabenstellung (das sog. *Anwendungswissen*). Letzteres erklärt, warum sich Programmierung schlecht outsourcen oder gar auf Programme (die sog. *Programmsynthese*) übertragen lässt.

Überbewertung des Testens

Manche Programmierer vertrauen so sehr auf das (automatisierte) Testen, dass sie dafür auf andere Mechanismen der Fehlerentdeckung (wie beispielsweise die *statische Typprüfung*; s. Abschnitt 20.4) gern verzichten. Ich teile dieses Vertrauen nicht — wenn das Ziel fehlerfreie Software ist, muss man jede Hilfe annehmen, die sich anbietet. Testen ist da nur ein — wenn auch ein wichtiger — Baustein.

21.3 Versionskontrolle

Programme wachsen und werden laufend geändert. Zudem arbeiten häufig mehrere Programmierer (gleichzeitig oder abwechselnd) am selben Programm. Damit die dadurch entstehenden Programmversionen nicht verlorengehen und später abgeglichen oder zusammengeführt werden können, gibt es **Versionskontrollsysteme**, die man sich als eine Art chronologische Datenbank für Programmtexte vorstellen kann. Die meisten dieser Versionskontrollsysteme sind dateigebunden; sie funktionieren am besten, wenn der Text eines Programms auf mehrere Dateien aufgeteilt ist, die dann unabhängig voneinander bearbeitet werden können.

21.4 Automatische Programmierung

Obwohl Programmiersprachen im Vergleich zu natürlichen Sprachen stark reduziert sind, enthalten Programme doch Regelmäßigkeiten und *Redundanzen*, die sich ausnutzen lassen, um das Schreiben von Programmen per Automation zu unterstützen. Dies tun u. a. Werkzeuge zur automatischen Vervollständigung von angefangenen Programmfragmenten (wie z. B. Befehlen oder Namen), Werkzeuge zur automatischen Korrektur von syntaktisch oder semantisch fehlgeformten (fehlerhaften) Programmen und Werkezuge zur systematischen Umstrukturierung von Programmen. In letzter Zeit werden auch vermehrt auf großen Programmcorpora basierende statistische Verfahren eingesetzt, um zu erraten, was ein Programmierer schreiben will; vergleichbare Verfahren kennen Sie vielleicht schon aus der Spracherkennung und aus automatischen Übersetzungen, wo man manchmal staunt, wie gut das funktioniert, aber sich auch öfters denkt: „wie hirnlos!" Ich bin mir nicht sicher, ob die durch solche statistischen Verfahren erzielbaren Produktivitätsgewinne nicht nur scheinbare sind — was, wenn eine automatische Vervollständigung einen Fehler einbringt, sich der Programmierer aber an diesen Teil seines Werks, mangels genügender eigener Befassung damit, gar nicht erinnern kann? Dann verbringt man ein vielfaches der Zeit, die einem die automatische Vervollständigung gespart hat, mit der Fehlersuche.

21.5 Integrierte Entwicklungsumgebungen

Editor, Compiler oder Interpreter und andere Programmierwerkzeuge wie die oben erwähnten werden heute häufig in sog. **integrierten Entwicklungsumgebungen** zusammengefasst, die sich somit als allumfassende Programmierwerkzeuge anbieten. Solche Entwicklungsumgebungen sind extrem komplexe Programme mit einem enormen Funktionsumfang, deren Bedienung man nach und nach erlernen muss; ihre Entwicklung ist extrem aufwendig, zumal sich Entwicklungsumgebungen für verschiedene Programmiersprachen technisch weit mehr unterscheiden, als man vielleicht annehmen könnte. Gleichwohl lässt sich mit integrierten Entwicklungsumgebungen nur schlecht Geld verdienen, so dass die meisten frei verfügbar sind (die Kausalität ist hier nur scheinbar vertauscht — wenn eine solche Umgebung nicht frei verfügbar ist, entwickelt sich kein großer Nutzerkreis mit der Folge, dass die wenigen Nutzer vom Hersteller selbst betreut werden müssen; s. a. Abschnitt 20.3 für eine parallele Beobachtung bei Programmiersprachen).

Ob eine integrierte Entwicklungsumgebung zum Erlernen einer Programmiersprache eher schädlich oder nützlich ist, hängt nicht nur von der Entwicklungsumgebung und dem Vorwissen des Lernenden ab, sondern maßgeblich auch von der Programmiersprache: Für eher einfach zu erlernende Sprachen kann eine Entwicklungsumgebung den Blick auf die Einfachheit verstellen, für eine eher schwierige kann sie helfen, Hürden zu überwinden (z. B. mit sog. Wizzards, das sind Formulare, die man als Programmierer ausfüllt und aus denen dann der entsprechende Code generiert wird).

22 Zum Beispiel Scratch

Link

Scratch ist ein Programmiersystem, das aus einer einfachen Programmiersprache und einer einfachen Entwicklungsumgebung besteht. Es wird seit 2007 von der Lifelong Kindergarten Group am amerikanischen MIT entwickelt und verbreitet.

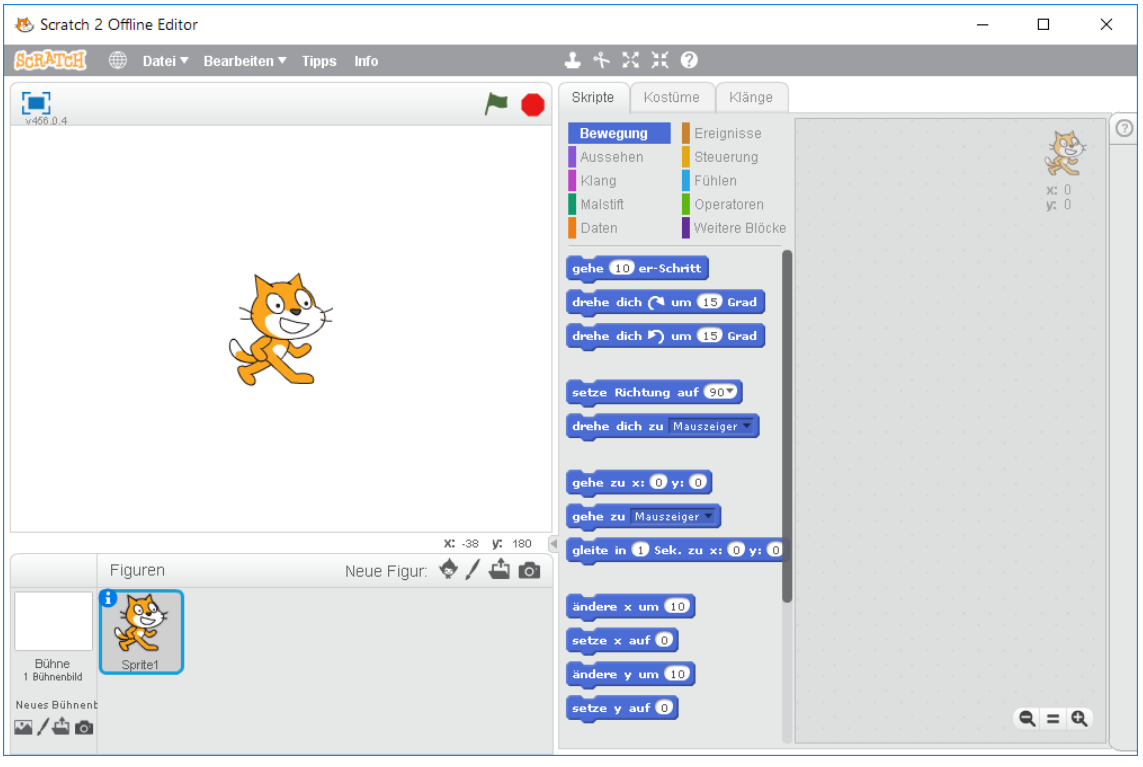


Abbildung der Scratch-Umgebung nach dem Starten und Auswählen von Deutsch als Sprache (über den Globus links oben, rechts neben dem Schriftzug „Scratch“, einzustellen)

Der Vorteil von Scratch gegenüber anderen Versuchen, eine Programmiersprache mit möglichst niedrigen Hürden zu entwickeln, liegt zum einen darin, dass die Sprache grafisch und nicht textbasiert ist und der (graphische) Editor weitgehend verhindert, fehlgeformte (d. h., syntaktisch oder semantisch nicht korrekte und damit nicht ausführbare) Programme einzugeben (eine häufige Quelle von *erheblicher Frustration* gerade bei Anfängern), und zum anderen darin, dass die Sprachkonstrukte von Scratch geeignet sind, kleine Spiele oder einfache Videoclips zu programmieren (was in anderen Sprachen erhebliche Beschlagenheit erfordert). Ein zusätzlicher Anreiz wird dadurch geschaffen, dass man sein Werk leicht online mit anderen teilen kann (eigene Spiele können sofort auch von anderen nicht nur gespielt, sondern auch untersucht werden) und sich umgekehrt vom Werk anderer vieles abgucken kann.

22.1 Das Programmiermodell von Scratch

Technisch gesehen ist Scratch eine *imperative*, *strukturierte*, *objektbasierte Programmiersprache*, die *ereignisgesteuerte*, *parallele Programmierung* mit Nachrichtenaustausch zwischen Objekten erlaubt. Diese Eigenschaften kommen so oder so ähnlich auch in anderen Programmiersprachen vor; das besondere von Scratch sind ihre spezifischen Ausprägungen.

Seiner Bestimmung folgend pflegt Scratch die Metapher von einem zu programmierenden Theater- oder Bühnenstück. Die an Theater erinnernde Namen von Programmierkonzepten werden Sie in anderen Programmiersprachen nicht wiederfinden.

22.1.1 Objekte

Ein Programm besteht in Scratch aus einem Hintergrund, Bühne genannt, sowie einem oder mehreren sog. Sprites. Sowohl Bühne als auch Sprites sind grafische Objekte, die in einem Fenster (rechteckigen Areal des Bildschirms) angezeigt werden. Während die Bühne starr ist und das Fenster stets voll ausfüllt, sind Sprites beweglich: sie können ihre Position verändern, rotieren sowie größer oder kleiner werden. Dabei bewegen sich die Sprites in einem kartesischen Koordinatensystem mit $-240 \leq x \leq 240$ (Horizontale) und $-180 \leq y \leq 180$ (Vertikale). Sowohl Bühne als auch Sprites können ihr Erscheinungsbild wechseln; bei der Bühne nennt man die Bilder Bühnenbilder, bei Sprites Kostüme. Für Sprites kann man durch wechselnde Kostüme wie in einem Trickfilm den Eindruck einer echten Bewegung erzeugen (um dies zu sehen, können Sie in der Scratch-Umgebung „Aussehen“ auswählen und dann einmal auf „nächstes Kostüm“ klicken). Bilder können mit Scratch selbst gezeichnet (mit einem einfachen grafischen Editor), mit einer Kamera aufgenommen oder von einer anderen Quelle importiert werden.

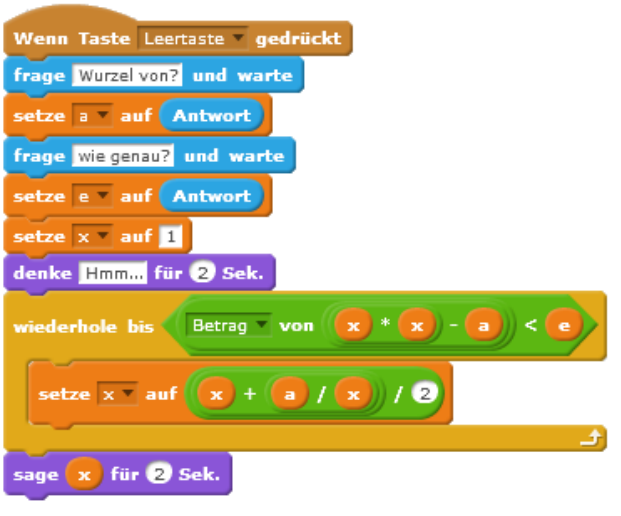
Neben Bewegungen simulieren können Scratch-Programme auch Geräusche erzeugen. Dazu stehen ein einfacher Synthesizer (mit dem man Noten auf simulierten Instrumenten oder Schlagzeug spielen kann), eine Aufnahmemöglichkeit mit einem vorhandenen Mikrofon oder wiederum das Importieren aus anderen Quellen zur Verfügung. Zusätzlich gibt es Unterstützung für einfache Umgebungssensoren, für die man allerdings, abgesehen von der Wahrnehmung der Umgebungslautstärke bei vorhandenem Mikrofon, eine extra Hardware benötigt.

22.1.2 Ereignisgesteuerte, strukturierte Skripte

Die Programmierung erfolgt in Scratch mithilfe sog. Skripte, die man entweder der Bühne oder einzelnen Sprites zuordnet. Ein Skript kann ausgeführt werden, indem man es anklickt oder indem man es mit einem Ereignis (wie beispielsweise das Klicken der grünen Flagge, dem Startsymbol) verbindet. Alle in einem Skript vorkommenden Kommandos bzgl. des Aussehens oder der Bewegung beziehen sich auf das Objekt, dem sie zugeordnet sind. Insofern ist Scratch *objektorientiert*; da es aber weder Klassen noch Vererbung bietet, würde man eher von *objektbasiert* sprechen.

Ein Skript definiert einen Teil eines Scratch-Programms. Wie bereits erwähnt ist Scratch als Programmiersprache *strukturiert*, d. h., es kennt kein Goto, sondern lediglich *Sequenz*, *Verzweigung* und *Wiederholung* (*Schleife*). Anders als die meisten anderen Programmiersprachen kennt Scratch keinen Unterprogrammaufruf; dieser muss mittels Nachrichtenaustauschs simuliert werden. Verschiedene Scratch-Skripte laufen dabei grundsätzlich parallel (d. h., gleichzeitig oder quasi-gleichzeitig); eine Synchronisation findet nicht statt. Dabei ist der Nachrichtenaustausch grundsätzlich ein Broadcast, d. h., Nachrichten werden nicht gezielt an bestimmte Sprites verschickt, sondern gehen grundsätzlich an alle und lösen dort ein Ereignis, den Empfang einer Nachricht, aus. Damit wird vermieden, dass Sprites einander kennen müssen (voneinander abhängen). Auf der anderen Seite entsteht so das Problem, dass ein Sprite selbst erkennen muss, ob es mit einer Nachricht gemeint ist.

Nachfolgendes Skript implementiert die Bestimmung der Quadratwurzel nach der Methode der Sumerer, wie wir sie schon in Abschnitt 18.2 kennengelernt haben. Es ist einem Sprite zugeordnet und mit Anweisungen zur Eingabe (`frage … und warte`) und Ausgabe (`sage … für … Sek.`) versehen. Man erkennt hier nicht nur, dass Scratch *ereignisgesteuert* ist (`Wenn … gedrückt`, `… und warte`, `… für … Sek.`), sondern auch die *Blockstruktur* der Sprache: Neben der Sequenz von Anweisungen, dargestellt durch die Folge der Bausteine, findet sich auch eine *Wiederholung*, die eine Sequenz (hier bestehend aus nur einer Anweisung) klammert. Die hier grafisch dargestellte Klammer (die an die beiden Schenkel einer Schraubzwinge erinnert) entspricht dem `begin … end` der *strukturierten Programmierung* und dient der Schachtelung von *Blöcken*. Man erkennt hier sehr schön, dass die Wiederholung genau einen Eingang (oben) und einen Ausgang (unten) hat, wie es in der (reinen) strukturierten Programmierung sein soll. Man erkennt außerdem die *Schachtelung von Ausdrücken*, die durch das Einsetzen von Ausrücken in Ausdrücken entsteht und die dem Klammern mit „(“ und „)“ in der Mathematik entspricht: $abs((x \cdot x) - a)$ und $((x + a)/x)/2$.



Durch die grafische Form der Programmierung ist es nicht möglich, in Scratch *syntaktisch fehlerhafte* Programme einzugeben: Wenn die Bausteine zusammenpassen, ist das Programm syntaktisch wohlgeformt, und die Eingabe eines `begin` ohne `end` oder eines `end` ohne `begin` ist nicht möglich (ganz zu schweigen von falsch gesetzten Satzzeichen). Bis zu

einem gewissen Grad gilt das auch für Teile der (*statischen*) *Semantik* von Scratch: So erlaubt `wiederhole bis …` beispielsweise nur sechseckige Ausdrücke einzufügen, die allesamt *boolesch* sind (also einen Wahrheitswert zum Ergebnis haben). Scratch führt also keine nachgelagerte *Typprüfung* durch, die dem Progammierer ggf. einen Fehler signalisieren würde, die er dann selbst korrigieren müsste, sondern erlaubt von Anfang an nur die Eingabe von Programmen ohne bestimmte *Typfehler*.

Wie das obige Beispiel zeigt, kann der Benutzer eines Scratch-Programms durch Drücken von Tasten auf der Tastatur Ereignisse auslösen, die dann zur Ausführung an die Ereignisse gebundener Skripte führen. Durch das Bedienen der Maus können ebenfalls Ereignisse ausgelöst werden. Beide Sorten von Ereignissen und deren Behandlung durch damit verbundene Programmfragmente (den Scratch-Skripten entsprechend) findet man heute auch in den Bedienoberflächen der allermeisten Anwendungsprogramme; insofern dient die (spielerische) Scratch-Programmierung durchaus dem Erlernen praxisrelevanter Programmiertechniken.

22.1.3 Zustand und Variablen

Scratch-Programme sind *imperativ*: Sie bestehen nicht nur aus *Anweisungen*, sondern verfügen auch über einen *Zustand*. Dieser ist zum einen in Position, Richtung, Größe und Kostüm der einzelnen Sprites codiert, kann aber zum anderen auch durch vom Programmierer eingeführte *Variablen* erweitert werden, die sich anhand ihres Namens unterscheiden. Diese Variablen sind entweder global (beispielsweise zur Speicherung eines aktuellen Punktestandes in einem Spiel) oder einem Sprite zugeordnet und dann zu diesem lokal. Lokalität bedeutet hier, dass a) nur das Sprite selbst auf „seine" Variablen zugreifen und b) derselbe Variablenname für lokale Variablen verschiedener Sprites verwendet werden kann. Diese lokalen Variablen entsprechen den sog. Feldern der *objektorientierten* (und auch *objektbasierten*) *Programmierung*. Leider sind in Scratch Variablen nicht *typisiert*, d. h. konkret, dieselbe Variable kann sowohl Zahlen als auch Text speichern. Damit werden in Scratch dann doch Typfehler möglich.[69]

Link

Timer-Steuerung

Neben den systemeigenen lokalen Variablen `Position`, `Richtung` etc. eines Sprites gibt es mit `Stoppuhr` auch eine globale Systemvariable, die automatisch die verstrichene Zeit in Sekunden zählt (laut Handbuch mit einer Schrittweite von einer zehntel Sekunde, wovon man sich durch Sichtbarmachen des Inhalts der Variable `Stoppuhr` überzeugen kann; man setzt dazu lediglich ein Häkchen an der Stelle, an der die Stoppuhr zur Verfügung gestellt wird). `Stoppuhr` kann durch Skripte auf 0 zurückgesetzt werden und per Vergleich mit einem anderen Wert als Bedingung in einer Verzweigung oder Wiederholung genutzt werden. Solche Systemvariablen (für Datum und Zeit) gibt es auch in vielen anderen Programmiersprachen. Leider wird in Scratch durch das Erreichen einer bestimmten

---

[69] Hierbei handelt es sich um einen jener Kompromisse, zu denen sich jeder Sprachdesigner früher oder später genötigt sieht, z. B. wenn es (wie im gegeben Fall) darum geht, Einfachheit der Sprache gegen ihre Sicherheit abzuwägen. Vgl. dazu auch Abschnitt 20.2.

(einstellbaren) Zeit kein Ereignis ausgelöst, dem man dann Skripte zuordnen könnte; ein solches „timergesteuertes“ Ereignis lässt sich aber, dank der Parallelität von Scratch, leicht selbst programmieren:

Hierdurch soll nach 5 Sekunden die Nachricht „klingeling“ an alle (Sprites) gesendet werden. Man beachte allerdings, dass wenn Scratch tatsächlich die Zeit jede zehntel Sekunde um 0,1 erhöht und die Schleife innerhalb einer zehntel Sekunde mehrfach durchlaufen wird, obiges Programmfragment (Skript) die Nachricht mehrfach aussenden müsste (weil `Stoppuhr` eine zehntel Sekunde lang den Wert `5` hat). Um diese Mehrfachaussendung zu verhindern, müsste man also dafür sorgen, dass die Bedingung unmittelbar nach ihrer ersten Erfüllung nicht mehr erfüllt ist, z. B. indem man die Stoppuhr auf 0 zurücksetzt, oder indem man die umgebende Schleife verlässt. Tatsächlich wäre für die einmalige Auslösung des Timer-Ereignisses die Implementierung

geeigneter (wobei der leere Schleifenrumpf als „tue nichts“ gelesen werden kann). Programmierfehler der obigen Art können recht tückisch sein (weil sie sich nicht bei jedem Programmlauf bemerkbar machen, da das davon abhängt, wie lange ein Schleifendurchlauf benötigt, was wiederum von vielen, teilweise nur schwer zu kontrollierenden, auch außerhalb des Programms liegenden Faktoren abhängen kann) und unterlaufen selbst routinierten Programmierern.[70]

Abstraktionsversagen

Tatsächlich offenbaren beide obigen Programme aber noch ein ganz anderes Problem, das offenbar wird, wenn man sie ausführt: Das Ereignis `klingeling` wird überhaupt nicht ausgelöst. Den Grund hierfür zu finden ist etwas schwerer, weil man ihn nicht erkennen kann, ohne zu wissen, wie Scratch die Zeit intern repräsentiert: durch eine *Gleitkommazahl* im *Dualsystem*! Deren Genauigkeit reicht zwar, wie wir in Abschnitt 2.6.3 gesehen haben, immer aus, um z. B. 0 oder 5 richtig (d. h., ohne Rundungsfehler) darzustellen, aber schon 0,1 hat ja immer einen Rundungsfehler, so dass durch die 50-malige Addition von 0,1 zu 0 nicht genau 5,0 erreicht wird. Auch wenn hier wohl eher eine Inkonsistenz im Scratch-Interpreter zu vermuten ist (die Anzeige des Wertes von

[70] So stehen bei mir zuhause zwei neue Radiowecker eines namhaften Herstellers, die beide während der Minute der eingestellten Weckzeit nach dem Ausstellen sofort wieder angehen, da ja die eingestellte Weckzeit immer noch anliegt und der Alarm so immer wieder neu ausgelöst wird.

Stoppuhr erfolgt auf eine zehntel Sekunde gerundet, beim Vergleich mit einer Zahl unterbleibt diese Rundung aber), so sieht man doch an diesem Beispiel sehr schön, wie eine *Abstraktion* einer Programmiersprache *versagen* kann, und dass man, wenn man nicht weiß, wie Programme „unter der Haube" ausgeführt werden, Probleme hat, ihr Verhalten zu verstehen. Solche Probleme können die Quelle *erheblicher Frustration* sein.

To understand a program you must become both the machine and the program.

Alan Jay Perlis (* 1. April 1922, † 7. Februar 1990)

22.2 Arbeiten mit Listen

Nicht selten muss ein Programm nicht nur einzelne Werte verarbeiten, von denen jeder in einer eigenen Variable gehalten wird, sondern eine ganze Reihe von Werten, wobei deren Anzahl selbst variabel sein kann. Man speichert dann nicht jeden Wert in einer einzelnen Variable, sondern alle in einer, die die Form einer Liste hat. Listen sind eine äußerst populäre Datenstruktur, die man in der einen oder anderen Form in so gut wie allen Programmiersprachen findet (s. dazu auch Abschnitt 27.4.1 in Kurseinheit 4).

Ein typisches Anwendungsbeispiel für Listen ist der Sortieralgorithmus aus Kapitel 16. Zwar könnte man, wenn man den Algorithmus für eine maximale Anzahl von Karten implementiert, für jede Position auf jedem der drei benötigten Stapel eine Variable vorsehen, aber das Programm müsste, damit es halbwegs kompakt ausfällt, die Namen der Variablen vor jedem Zugriff darauf irgendwie berechnen. Wenn man beispielsweise die Stapel A, B und C nennt und die einzelnen Positionen auf jedem Stapel $A1$, $A2$, …, $B1$, $B2$, …, $C1$, $C2$, …, dann würde $A1$ die erste Karte auf Stapel A halten und so weiter. Mit der festen Benennung der einzelnen Stapelposition ist es aber schwierig, einen Sortieralgorithmus allgemein zu formulieren. Vielmehr möchte man die i-te Position eines Stapels benennen können, ohne den Wert von i zu kennen (i soll eine Variablen sein können).

Listen sind nun Variablen, die mehr als nur eine Speicherstelle haben. Jede dieser Speicherstellen trägt eine Nummer und kann über diese angesprochen werden. Je nach Programmiersprache (und Konvention) trägt die erste Speicherstelle die Nummer 0 oder 1 und entsprechend die letzte die Nummer $n-1$ oder n, wenn die Liste n Speicherstellen hat. Dabei kann die Programmiersprache vorsehen, dass die Anzahl der Speicherstellen einer Liste bei ihrer Deklaration (gemeinsam mit ihrem Namen) angegeben werden muss und dann unveränderlich ist (in diesem Fall spricht man auch häufig von einem *Array* oder genauer von einem *statischen Array*; s. Abschnitt 27.2.3 in Kurseinheit 4) oder aber wachsen und schrumpfen kann. Bei Listen, deren Länge nicht fest vorgegeben ist, kann man die Länge im Programm abfragen (man spricht dann auch von einem *dynamischen Array*).

Die nachfolgende Abbildung zeigt die Implementierung des Sortieralgorithmus aus Kapitel 16 in Scratch. Der Algorithmus ist über die Länge der Liste (Variable n) parametrisiert, die hier nur beispielsweise auf 8 festgesetzt wurde. Man erkennt zwar sehr schön die (syntaktische) *Blockstruktur* des Skripts — seine inhaltliche Struktur bleibt aber ziemlich verborgen (die *Kommentare* sollen hier helfen). Auch wenn man solche Programme in praktisch jeder Programmiersprache schreiben kann, so würde man doch in einer prozeduralen Sprache (was Scratch nicht ist!) eher nach dem Prinzip der *funktionalen Dekomposition* (s. Abschnitt 18.4) vorgehen und das Programm als eine Hierarchie von einander aufrufenden Funktionen aufschreiben. Tatsächlich enthält das Skript mehrere annähernd gleichlautende Blöcke, was bei Programmiererinnen geradezu reflexhaft einen *Refaktorisierungswunsch* auslöst, der in Scratch aber ins Leere läuft. In einer prozeduralen Sprache wäre es ein leichtes, das Programm kürzer und verständlicher zu formulieren; Skriptsprachen wie Scratch sind aber für die *funktionale Dekomposition* nicht gedacht.

```
setze n auf 8                                   ▸ Anzahl zu sortierender Elemente
wiederhole Länge von A mal                      Lösche alle Einträge aus Liste A.
    lösche 1 aus A
wiederhole bis Länge von A = n                  Füge solange eine Zufallszahl i zwischen 1 und n, die noch nicht enthalten ist, in Liste A ein, bis sie n Elemente hat. Kann dauern ...
    setze i auf Zufallszahl von 1 bis n
    falls nicht A enthält i ? dann
        füge i zu A hinzu
warte 1 Sek.
setze i auf n                                   ▸ Anzahl der sortierten Blöcke
wiederhole bis i = 1
    setze i auf i / 2
    wiederhole i mal                            Verteile die sortieren Blöcke von Liste A blockweise auf die Listen B und C..
        setze j auf n / i / 2                   ▸ Größe der sortierten Blöcke
        wiederhole j mal
            füge Element 1 von A zu B hinzu
            lösche 1 aus A
        warte 0.5 Sek.
        wiederhole j mal
            füge Element 1 von A zu C hinzu
            lösche 1 aus A
        warte 0.5 Sek.
    warte 1 Sek.
    wiederhole i mal                            Füge je einen Block von Liste B und Liste C so in Liste A zusammen, dass der resultierende, doppelt so große Block in Liste A sortiert ist.
        setze b auf 1
        setze c auf 1
        wiederhole bis b > j oder c > j         ▸ solange beide Blöcke noch Elemente haben ...
            falls Element 1 von B < Element 1 von C dann
                füge Element 1 von B zu A hinzu
                lösche 1 aus B
                ändere b um 1
            sonst
                füge Element 1 von C zu A hinzu
                lösche 1 aus C
                ändere c um 1
        wiederhole j - b + 1 mal                ▸ Rest von Block B hinzufügen
            füge Element 1 von B zu A hinzu
            lösche 1 aus B
            ändere b um 1
        wiederhole j - c + 1 mal                ▸ Rest von Block C hinzufügen
            füge Element 1 von C zu A hinzu
            lösche 1 aus C
            ändere c um 1
    warte 1 Sek.
```

Abbildung eines Scratch-Programms, das den Sortieralgorithmus von Kapitel 16 auf Listen umsetzt

22.3 Programmieren in Scratch

Die Programmierung in Scratch erfolgt primär über Drag&Drop: Anweisungen und deren Parameter werden aus der Auswahlpalette in den Skriptbereich, der dem Textfenster eines klassischen Editors entspricht, gezogen und dort positioniert. Das Ändern von Skripten erfordert etwas Übung; im Zeitalter der Apps möchte ich hier aber keine Hilfestellungen geben, sondern vertraue ganz auf die Selbsthilfefähigkeiten meiner Leser (wo gibt es heute noch Bedienungsanleitungen?).

Debugging in Scratch

Während ältere Versionen von Scratch noch eine verlangsamte Ausführung von Skripten erlaubten, bei der die Ausführung jedes einzelnen *Befehls* und damit der Programmablauf beobachtet werden konnte, wird dies in der aktuellen Version 2 nicht mehr angeboten. Allerdings führt eine verlangsamte Ausführung von parallelen Programmen (was die meisten Scratch-Programme sind) in der Regel zu einem anderen Verhalten, so dass der Nutzen dieser Funktion beschränkt wäre. Stattdessen muss man Scratch-Programme aber leider wie in der Steinzeit debuggen (vgl. Abschnitt 21.1; hier mithilfe des „sage"-Befehls).

Link

22.4 Entwicklung eines Programmierstils

There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.

Lawrence Flon

Auch wenn Scratch, was die Möglichkeiten der Programmierung angeht, recht eingeschränkt ist, so hat man als Programmierer doch viele Freiheiten und muss entsprechend viele Entscheidungen treffen. So ist es z. B. nicht immer unmittelbar klar, ob man ein Skript einem Sprite oder der Bühne zuordnet. Als Anfänger fühlt man sich bei diesen Entscheidungen schnell alleingelassen und tatsächlich gibt es neben richtig und falsch bei der Programmierung auch so etwas wie **Programmierstil**, der nicht durch die Programmiersprache oder -umgebung vorgegeben ist, der aber doch, aufgrund von Bewährung, häufig in gut und schlecht eingeteilt wird. Am Anfang sollte man sich damit jedoch nicht belasten, da man aus eigenen Fehlern und eigener Anschauung im Zweifel mehr lernt als aus dem Studieren von Stilhandbüchern. Fortgeschrittenen empfehle ich aber nachdrücklich, sich auch den Programmierstil anderer anzusehen, denn nur im eigenen Saft zu schmoren führt bekanntlich zu Wunderlichkeit und ein bewährter Programmierstil führt in der Regel nicht nur zu besser les- und wartbaren Programmen, sondern auch zu Anerkennung unter den Programmierkollegen. Guten Programmierstil zu schätzen und schlechten zu erkennen lernt man am besten durch Studieren von Programmen anderer.

23 Anwendungsprogramme

Anwendungsprogramme (deutsch auch kurz *Anwendungen*, engl. *application programs* oder nur *applications*, neuerdings besonders im Kontext von Mobilgeräten zu *Apps* verkürzt; aber s. Abschnitt 23.2) sind Programme, die auf dem Betriebssystem aufsetzen, die eine bestimmte, meist spezielle (problembezogene) Funktionalität zur Verfügung stellen und die letztlich der Grund sind, warum man überhaupt einen Computer benutzt. Typische weit verbreitete Anwendungsprogramme sind Textverarbeitungen und Tabellenkalkulationen sowie betriebliche Software wie Lagerverwaltung oder Finanzbuchhaltung. Die Verfügbarkeit bestimmter Anwendungsprogramme und deren technische Anforderungen sind häufig die Grundlage für die Anschaffung eines bestimmten Computertyps (*Hardware* plus *Betriebssystem*).

Herausforderung Vielfalt

Entsprechend der universellen Verwendbarkeit von Computern (s. Abschnitt 11.8 in Kurseinheit 2) gibt es sehr viele verschiedene Anwendungsprogramme. Dabei wird der Markt nicht wie der für Betriebssysteme von einer kleinen Zahl von großen Anbietern dominiert (sieht man einmal von Standardanwendungen wie Textverarbeitung und Tabellenkalkulation ab; s. oben), so dass es in der Regel auch verschiedene Anwendungsprogramme für den gleichen Zweck gibt. Dabei ist es jedoch gerade für die kleinen Hersteller von Anwendungsprogrammen ein großes Problem, dass es nicht nur ein Betriebssystem gibt: Schon die parallele Erstellung einer Anwendung für Windows, Linux und MacOS ist aufwendig — die parallele Wartung (Fehlerbehebung und Anpassung an die Weiterentwicklung der Betriebssysteme) ist ein Alptraum und hat schon so manchem Hersteller das Aus beschert. Dies ist umso ärgerlicher, als die von Kunden verwendete Hardware (PC) seit Jahrzehnten weitgehend normiert ist (spätestens seit auch Apple auf PC-Hardware gewechselt hat). Dennoch gibt es nur sehr wenige Anwendungsprogramme, die ein Betriebssystem vollständig umgehen und direkt auf der Hardware aufsetzen (bei *eingebetteter Software* spricht man in der Regel nicht von Anwendung; s. Kapitel 25). Stattdessen werden manche Anwendungsprogramme für eine *virtuelle Maschine* (wie die *JVM*; s. Abschnitt 18.5) geschrieben, können dann aber nicht alle Besonderheiten der jeweiligen Zielsysteme ausnutzen.

maßgeschneiderte Software

Neben Anwendungsprogrammen, die von Herstellern für einen Markt entwickelt werden, gibt es auch die Programme, die von einem Auftragnehmer gemäß Spezifikation für einen Auftraggeber gefertigt werden. Solch maßgeschneiderte Software verspricht, ein spezielles Problem genau zu lösen und so zu vermeiden, dass die Geschäftsprozesse des Auftraggebers an die oft starren Vorgaben einer „Standardlösung“ angepasst werden müssen. Allerdings wird der Aufwand, der für eine hinreichend genaue Spezifikation einer solchen „Individuallösung“ investiert werden muss, regelmäßig unterschätzt und die Erwartungen an das Endprodukt stehen oft in einem Missverhältnis zum dem, was mit vertretbarem Aufwand machbar ist, so dass solche Projekte nicht selten in Streit und *Frustration* enden. Insbesondere die Erwartungen von Auftraggebern, die sich an der Funktionsvielfalt von Standardsoftware orientieren, müssen regelmäßig enttäuscht werden: Wer zu viel will, bekommt am Ende oft nichts. Insofern ist es geboten, bei der

Auftragssoftwareentwicklung nicht ins Träumen zu geraten, sondern sich auf dem Boden des Realistischen langsam voran zu bewegen. Diesen Boden kennen in der Regel die am besten kennen, die auf ihm stehen, sprich, die die *Anforderungen* in ein Programm umsetzen können. Gerade die werden jedoch häufig nicht gefragt.

Verteilung von Anwendungsprogrammen

Anwendungsprogramme können auf Endgeräten (Arbeitsplatzrechnern oder Mobilgeräten) verteilt oder zentral auf Servern, sog. **Application servern**, laufen. Typische Beispiele für die erste Kategorie sind die sog. Office-Anwendungen, mit denen jeder an etwas anderem arbeitet, und für die zweite unternehmensweite Programme wie Finanzbuchhaltung oder Warenwirtschaftssysteme, mit denen typischerweise mehrere Personen gleichzeitig an nur einem Datenbestand arbeiten. Auf Endgeräten laufende Programme haben den Vorteil, dass die Rechenlast über viele Geräte verteilt ist und dass in der Regel keine feste (stehende) Verbindung zwischen Rechnern benötigt wird (außer bei zentraler Datenhaltung), so dass auch der Datenverkehr in Grenzen bleibt; ein Nachteil ist, dass die Anwendungsprogramme so auf vielen verschiedenen Endgeräten installiert und gewartet werden müssen, was zumindest dann, wenn die Benutzer sich nicht selbst darum kümmern, einen beträchtlichen administrativen Aufwand bedeuten kann. Entsprechend haben die zentral laufenden Anwendungsprogramme den Vorteil, dass Installation und Wartung nur auf einem Rechner erfolgen müssen, und den Nachteil, dass sie zuverlässige und leistungsfähige Datenverbindungen zu den Endgeräten brauchen.

standardisierte Benutzungsschnittstelle

Für die Benutzung von Anwendungsprogrammen über das sog. *User interface* (UI, zu Deutsch auch *Benutzungsschnittstelle*) wurde gegen Ende der 80er Jahre im Rahmen der Systems Application Architecture von IBM eine Richtlinie, *Common User Access* (CUA) genannt, festgelegt, an die sich die Entwickler solcher Programme betriebssystemübergreifend für viele Jahre gehalten haben. Diese Richtlinie sieht u. a. vor, dass die Funktionen von Anwendungsprogrammen über ein am oberen Fensterrand befindliches Menü gesteuert werden können und dass dieses Menü sowohl über die Tastatur als auch die Maus bedient werden kann. Viele gebräuchliche Tastaturkürzel wie Alt + F4 für das Beenden eines Programms wurden darin ebenfalls festgelegt. Im Zeitalter von Mobilgeräten mit ihren Touchscreens und Gestensteuerung wurde der Standardisierungsgedanke jedoch dem Wettbewerb um das „intuitivste" UI geopfert und so fühlt man sich als Neuling nicht selten wie in einem Abenteuerspiel („wo ist hier der Ausgang?") oder ist auf eine *Community* angewiesen, um zu erfahren, wie man ein Programm bedient (die Idee der Bedienanleitungen wurden mit den Standards entsorgt).

23.1 Anwendungsprogramme mit Webschnittstelle

Insbesondere die (aus betriebswirtschaftlicher Sicht) häufigen Versionswechsel von Betriebssystemen und Anwendungsprogrammen, die unter Umständen erheblichen Konfigurations- und Wartungsaufwand bedeuten, haben dazu geführt, dass seit Beginn dieses Jahrtausends sog. **Thin clients**, also abgespeckte Arbeitsplatzrechner weitgehend ohne installierte Anwendungsprogramme, eingesetzt werden. Anstatt nun aber auf die gute alte Idee des Terminals (also reine Bildschirme mit Tastatur und Maus, wie man sie früher regelmäßig antraf)

zurückzufallen, werden als solche Clients häufig PCs mit Webbrowser eingesetzt. Die zentral gehaltenen Anwendungsprogramme werden dann, unter Verwendung einer Webadresse (URL), über einen Browser, wie er ansonsten auch zum Surfen verwendet wird, bedient. Was man dabei allerdings weniger bedacht hatte, ist, dass sich Browser (und Browser-Standards) ebenfalls ständig weiterentwickeln und unterschiedliche Anwendungsprogramme nicht alle mit denselben Browsern und Browser-Versionen gleichermaßen funktionieren, so dass man die alten Probleme nur gegen neue eingetauscht hat. Dazu kommen *eklatante Sicherheitsprobleme*, die Webbrowser und ihre Funktionserweiterungen (sog. Plugins) sowie das von ihnen verwendete und nicht für den Betrieb von Anwendungsprogrammen geschaffene Internetprotokoll *HTTP* (s. Abschnitt 13.3 in Kurseinheit 2) mit sich bringen. Auf welches Wagnis sich Unternehmen bei der Entscheidung für eine solche Lösung einlassen, kann man erahnen, wenn man bedenkt, dass kein Anbieter von Webbrowsern verpflichtet ist, diese — oder auch nur alle davon jemals angebotenen Funktionen — weiter anzubieten. Entsprechend groß war die Aufruhr, als Microsoft ankündigte, seinen Internet Explorer durch einen neuen Browser (Edge) abzulösen, den man um einen Gutteil der Funktionen des Internet Explorers erleichtert hatte.

Widgets und die Proliferation ihrer Programmiersprache

Die zunehmende Bedienung von Anwendungsprogrammen über das Web offenbarte schnell, dass die ursprünglichen Webformate, die ja vor allem der Anzeige von Inhalten dienen (Abschnitt 13.3), nicht ausreichen: Vielmehr werden auch komplexe Bedienelemente, wie man sie von grafischen Bedienoberflächen her kennt (sog. *Widgets*, kurz für Window gadgets; s. Abschnitt 12.7 in Kurseinheit 2), benötigt. Diese Web-Bedienelemente verwenden eine eigene Programmiersprache (meistens *JavaScript*), die das ursprünglich für *ausführbaren Inhalt* von Webseiten vorgesehene Java ersetzt. Tatsächlich haben sich JavaScript und die JavaScript-Programmierung inzwischen so weit verbreitet, dass ganze Anwendungsprogramme darin geschrieben werden. Dafür war JavaScript jedoch nie gedacht (und ist auch nicht wirklich dafür geeignet; s. Fußnote 66 auf Seite 128) — eines von zahllosen Beispielen, wie „organisch" (soll heißen: eher soziologisch denn technisch erklärbar) die Entwicklung der Systeme ist, von denen wir immer mehr abhängen.

23.2 Apps

Der oben beschriebene (und bemängelte) Trend hin zu *Thin clients* und der Verwendung von Webbrowsern als Benutzungsschnittstelle wurde merkwürdigerweise gerade auf den Geräten, die mehr als alle anderen am Internet hängen, durch die Einführung von **Apps** wieder rückgängig gemacht. Eine App kann man als „Fat client" verstehen, also als ein Anwendungsprogramm, das auf dem Endgerät installiert wird und das dort läuft. Zur Verbindung mit einem zentralen Server (häufig zum Austausch von Daten) oder mit anderen Clients kommunizieren Apps häufig über das (eigentlich für Webseiten gedachte) HTTP, schon weil dies in der Regel am wenigsten durch Sicherheitseinstellungen blockiert wird; damit werden freilich auch Sicherheitsvorkehrungen außer Kraft gesetzt, die beispielsweise Anwendungsdaten nur über spezielle *Portnummern* (und nicht den Port 80, über den das

Web läuft) herauszugeben erlauben würden (und die damit das einfache Blockieren der Herausgabe von Daten über *Firewalls* erlauben).

Da Apps vollwertige Programme (und keine in Webseiten versteckte Programmfragmente) sind, unterliegen sie auch keinen Einschränkungen eines Webbrowsers. Sie können sich also ihrem Benutzer unter Ausnutzung aller Funktionen, die von der jeweiligen Hardware nebst Betriebssystem bereitgestellt werden, präsentieren (und das immer noch weit verbreitete Zittern, Ruckeln und Wackeln nach jedem Klick auf einer Webseite kann entfallen). Die Kehrseite der Medaille ist allerdings, dass sich dieselbe Anwendung auf verschiedenen Geräten unterschiedlich präsentieren (da sich die Geräte nicht zuletzt in ihrer Bedienphilosophie teilweise erheblich voneinander unterscheiden) und deswegen jede Anwendung in verschiedenen Varianten implementiert werden muss. Erschwerend hinzu kommt, dass die *App-Entwicklungsumgebungen* der verschiedenen Plattformen auch noch auf verschiedene Programmiersprachen setzen, so dass selbst funktional identische, gemeinsame Teile in verschiedenen Sprachen implementiert werden müssen. *Plattformübergreifende App-Entwicklungsumgebungen* versprechen hier Abhilfe, sind bis heute aber nur eingeschränkt erfolgreich.

24 Zum Beispiel Visual Basic for Applications (VBA)

Viele Anwendungsprogramme bewegen sich im Bereich der Büroautomatisierung, in dem auch Standardanwendungsprogramme wie Microsoft Office oder LibreOffice weit verbreitet sind. Dabei können manche Anwendungen durchaus als Erweiterungen einer solchen Standardanwendung aufgefasst werden. So ist es beispielsweise denkbar, eine Standardtextverarbeitung so zu erweitern, dass sie Auftragsbestätigungen oder Rechnungen erstellt. Da trifft es sich gut, wenn man den Funktionsumfang der Textverarbeitung durch Programmierung erweitern kann. Für obengenannte Office-Produkte kann man dies beispielsweise mithilfe der Programmiersprache BASIC.

BASIC (für Beginner's All-purpose Symbolic Instruction Code) ist eine ziemlich alte, einfache *imperative Programmiersprache*, die für viele Jahre die Lieblingssprache eines gewissen *William Gates* war, der sie in seiner Firma weiter hegte und pflegte. BASIC basierte ursprünglich auf dem Konzept nummerierter Programmzeilen, wobei die Zeilennummern als Ziel von Sprunganweisungen, dem allseits verpönten „GOTO", aber auch dem Aufruf von *Unterprogrammen* (dem „GOSUB") dienten. Zumindest in dieser Beziehung ähnelt BASIC erheblich einer *Maschinensprache*.

Mit **Visual Basic**, einer Weiterentwicklung der Sprache BASIC durch Microsoft (wobei sich „Visual" hier nicht auf die Sprache, die keineswegs visuell wie etwa Scratch ist, sondern auf die Entwicklungsumgebung bezieht, mit der graphische Bedienoberflächen „visuell" entwickelt werden können) wurden viele der Schwächen der Programmiersprache BASIC ausgemerzt, so dass es bei der aktuellen Version nur noch vernachlässigbare Unterschiede zu

Sprachen wie *C#* (dem Microsoft-Derivat von *Java*) gibt. Man kann also mit Visual Basic zeitgemäß programmieren.

Bei **Visual Basic for Applications** (**VBA**) handelt es sich schließlich um eine Einbettung der Programmiersprache Visual Basic in die *Büroanwendungsprogramme* von Microsoft (besser bekannt als die Office-Programme, also Word, Excel etc.). In VBA geschriebene Programme werden ihrem Charakter entsprechend als **Makros** bezeichnet (es sind tatsächlich eher *Skripte* als Programme) und in Dokumenten oder Dokumentvorlagen gespeichert. Im einfachsten Fall handelt es sich bei diesen Makros um Aufzeichnungen von Aktionen des Benutzers eines Office-Programms (Tastatureingaben, Ausführen von Befehlen wie Speichern oder Formatieren), die durch einen sog. **Makrorecorder** vorgenommen werden. Diese aufgezeichneten Makros dienen häufig als Ausgangspunkt für die Entwicklung komplexerer Programme (*Unterprogramme*, oder *Subroutinen* im Jargon von VBA) per Modifikation der Aufzeichnung. Dabei werden u. a. Kontrollflussanweisungen hinzugefügt (`If … Then`, `Do … Loop` usw.), die nicht als Aktionen im Anwendungsprogramm zur Verfügung stehen und die deswegen nicht aufgezeichnet werden können.

Makros können den Funktionsumfang von Anwendungen erheblich erweitern und verändern. So lassen sich beispielsweise in Excel komplexe Datenanalysen programmieren und in Word können Makros mit Dokumentvorlagen verbunden werden, die dann für jedes Dokument auf Basis dieser Vorlagen zur Verfügung stehen. Da Makros zudem in die Menü-Struktur der Office-Programme eingebunden werden können, lassen sich diese mit Makros nahezu beliebig erweitern. Nachteil dieser Art von Anwendungsentwicklung ist, dass alle Benutzer so entstandener Programme auch die jeweiligen Office-Programme haben müssen (denn ohne diese können die Makros nicht ausgeführt werden).

Beispiel

Ein Beispiel für ein Word-Makro, das in VBA geschrieben ist, ist das folgende:

```
87 Sub genderisiere()
88   Dim Text As String
89   Text = Selection.Text
90   If Not Text = "" Then
91     Selection.Cut
92     Selection.TypeText Text:=" "
93     Selection.MoveLeft Unit:=wdCharacter, Count:=1
94   End If
95   WText = InputBox("weiblich", "Gendertext", Text)
96   MText = InputBox("männlich", "Gendertext", Text)
97   Selection.Fields.Add
             Range:=Selection.Range,
             Type:=wdFieldEmpty,
             PreserveFormatting:=False
98   Selection.TypeText Text:="IF "
99   Selection.Fields.Add
             Range:=Selection.Range,
             Type:=wdFieldDocProperty,
             Text:="gegendert",
             PreserveFormatting:=False
```

```
100   Selection.TypeText Text:="""Y"" """ + WText + """ """ + MText +
        """"
101   Selection.MoveRight Unit:=wdCharacter, Count:=2
102 End Sub
```

Sie erkennen (in Zeile 87) den Beginn eines *Unterprogramms*, durch das *Schlüsselwort* `Sub` eingeleitet.[71] Sein Name legt nahe, dass es der Genderisierung eines Textes dienen soll. *Zeichenkettenliterale* werden im Programmtext mit doppelten Anführungsstrichen gekennzeichnet; so steht `""` in Zeile 90 für die leere Zeichenkette und `" "` für die Zeichenkette, die nur aus dem Leerzeichen besteht (Zeile 92). Zwei aufeinanderfolgende doppelte Anführungsstriche als Bestandteil eines Zeichenkettenliterals (also innerhalb von doppelten Anführungsstrichen wie in Zeile 100) bezeichnen einen doppelten Anführungsstrich (vgl. Abschnitt 4.1 in Kurseinheit 1). Der Operator **+** *konkateniert* (*verkettet*) zwei Zeichenketten zu einer (ebenfalls Zeile 100).

Das Unterprogramm (oder Makro) greift auf den Text des Dokuments, auf dem es ausgeführt wird, mittels der Variable `Selection` zu. Sie bezeichnet den aktuell markierten Text im Dokument, die Auswahl. Wenn kein Text ausgewählt ist, bezeichnet `Selection` die Position im Text, an der die Schreibmarke (der Cursor) gerade steht. An dieser Stelle kann dann das Makro alles machen, was man auch per Tastatur machen könnte, also beispielsweise löschen (mit `.Cut`), etwas schreiben (mit `.TypeText`) oder die Schreibmarke bewegen (mit `.MoveLeft`). `InputBox` (Zeilen 95 und 96) ruft eine Eingabemaske auf und gibt einen vom Benutzer eingegebenen Text zurück; im obigen Programm stellen die Eingaben die weibliche und die männliche Textvariante dar. In den folgenden Zeilen werden dann Felder in das Dokument eingefügt, die in Abhängigkeit vom Wert einer Dokumenteigenschaft mit Namen `"gegendert"` entweder die weibliche oder die männliche Variante anzeigen.

Vergleich mit Scratch

Auf den ersten Blick scheint obiges VBA-Programm mit dem Scratch-Programm von Abschnitt 22.2 in Kurseinheit 3 nicht viel gemeinsam zu haben. Doch lassen Sie sich von den Erscheinungsbildern nicht täuschen. Tatsächlich ist auch Visual Basic strukturiert und die „Zwingen" von Scratch entsprechen den durch Einrückungen abgesetzten Blöcken des VBA-Programms (`Sub … End Sub`, `If … End If`), die ebenfalls beliebig tief geschachtelt werden können. VBA ist wie Scratch ereignisgesteuert — das Makro kann mit einem Menüeintrag oder einer Tastenkombination verknüpft werden und wird dann, ganz wie in Scratch, durch Eintreten des Ereignisses gestartet. Die vielen *Bezeichner* (oder Namen, wie `Selection`, `wdFieldEmpty`, `PreserveFormatting` etc.), die das Makro verwendet und die das Gros des Programmtextes ausmachen, sind durch Word vordefiniert und werden vom Makro lediglich verwendet — tatsächlich führt es außer seinem eigenen lediglich den Namen `Text` (als Name einer Variable; deklariert in Zeile 88) selbst ein. Sowohl Scratch-Programme (Skripte) als auch VBA-Programme (Makros) sind Prozeduren, die in ein *Framework* eingehängt und von diesem in seinem Kontext ausgeführt werden. Und nicht zuletzt sind beide objektbasiert: Auch wenn das in der Implementierung des Sortieralgorithmus in

---

[71] Schlüsselwörter werden heute meistens klein geschrieben; *BASIC*, und somit auch *Visual Basic*, beachtet aber die Groß-/Kleinschreibung gar nicht und der zu VBA gehörende Editor normiert die Schreibweise aller Schlüsselwörter automatisch.

Scratch nicht unmittelbar deutlich wird, so ist der Algorithmus doch einem Objekt (dem Hintergrundbild) zugeordnet und die Namen in VBA, die hinter einem Punkt stehen, sind dem Objekt zugeordnet, für das der Name vor dem Punkt steht.

25 Eingebettete Software

Eingebettete Software nennt man die Programme, die in Geräten laufen, die man selbst nicht als Computer bezeichnen würde, in denen aber ein Computer verbaut ist. Solche Software bezieht ihre Eingaben in der Regel von Sensoren (Temperaturfühler, Schalter etc.) und liefert ihre Ausgaben an Aktoren (Motoren, Relais etc.) — Bedienelemente wie Knöpfe, Schalter und einfache Anzeigen sind unter den Sensoren eingereiht. Für komplexere Geräte wird die Bedienung zunehmend durch sog. Touch screens unterstützt (wie beispielsweise bei Fotokopierern); damit wird dann auch die Versuchung größer, selbst bei eingebetteter Software auf einem *Betriebssystem*, wie es für Anwendungsprogramme verwendet wird, aufzusetzen. Das führt dann unter anderem dazu, dass ein Gerät, das man nicht als Computer gekauft hat, wie ein Computer „abstürzt" und dann „gebootet" werden muss. Auch können solche Geräte das Ziel von Schadsoftware werden, wenn sie mit dem Internet verbunden sind oder zwecks Update von einem USB-Stick o. ä. Programme oder Daten (manchmal Patches oder Images genannt) auslesen müssen. Nachdem in der Vergangenheit vor allem Windows als Betriebssystem von eingebetteten Systemen für negative Schlagzeilen gesorgt hat (Stichwort Ransomware in Registrierkassen oder Ticketsystemen), dürften Angriffe auf den Linux-Kern wesentlich mehr Menschen betreffen (fast jeder private Internet-Router und fast jedes größere Unterhaltungselektronikgerät zuhause basieren heute auf einem Linux-Kern). Besonders bedenklich ist in diesem Zusammenhang, dass selbst infrastrukturkritische Systeme (wie etwa die Steuergeräte von Windrädern) „am Internet hängen" und somit potentiell von jedem angegriffen werden können, der einen Computer mit Internetzugang besitzt. Wenn so etwas passiert, muss man das wohl grob fahrlässig nennen.

Bedeutung eingebetteter Software

Der Umstand, dass eingebettete Software nicht so sichtbar ist wie Anwendungsprogramme, sollte nicht darüber hinwegtäuschen, dass die meiste verkaufte (und in Deutschland auch die meiste entwickelte) Software eingebettet ist. Dabei ist eingebettete Software nicht notwendig klein: Eine moderne Getriebesteuerung beispielsweise kann leicht über eine Millionen Programmzeilen umfassen. Insofern ist es wenig nachvollziehbar, wenn Softwareentwicklung vor allem mit den USA in Verbindung gebracht wird — praktisch jeder Hersteller von technischen Geräten (jeder Maschinenbauer!) ist heute auch Softwarehersteller. Allerdings wird diese Software typischerweise von Ingenieuren entwickelt, die Informatik oder auch nur Programmierung höchstens im Nebenfach gelernt haben — *Anwendungswissen* scheint Entwicklerteams oftmals wichtiger als Grundlagenwissen von der Softwareentwicklung und so hört man Sätze wie „das bisschen Programmieren bringen wir unseren Leuten dann schon bei" immer noch viel zu oft. Dabei kann sich mangelnde Expertise auf dem Gebiet der Softwareentwicklung

durchaus zu schwerwiegenden Problemen in der Produktentwicklung auswachsen, eine Erkenntnis, die jedoch erst nach und nach die Chefetagen erreicht.

26 Softwarehaftung und Stand der Technik

Jeder, der einmal so etwas wie eine Endbenutzerlizenzvereinbarung von Software gelesen hat, wird das Bemühen der Hersteller, Haftung auszuschließen, kennen. Häufig bezieht man sich dabei auf den Stand der Technik, nach dem fehlerfreie Software herzustellen unmöglich sei.

Dies ist natürlich in seiner Allgemeinheit nicht richtig und selbst wenn, dann nur in dem Umfang, in dem Fehlerfreiheit auch in anderen Produkten nicht ausgeschlossen werden kann. Richtig ist hingegen, dass fehlerarme Software sehr viel teurer herzustellen ist als fehlerreiche. Die Frage ist also eher, wie wenige Fehler man für das Geld, das mit einer Software verdient werden kann, erwarten darf. Diese Frage ist aber nicht nur schwer zu beantworten, sondern für einen konkreten Schadensfall wohl auch eher irrelevant, so dass man im Einzelfall entscheiden muss, ob mit der gebotenen Sorgfalt vorgegangen wurde. Dabei könnte man dann auch mal der Meinung sein, dass diese Sorgfalt durch den *Stand von Wissenschaft und Technik* vorgegeben ist, also den höchsten Ansprüchen genügen muss, insbesondere, wenn es um *Sicherheit* geht (vgl. Kapitel 15) und ein Schaden auf Sicherheitsmängel zurückgeführt werden kann. Umso problematischer ist es dann, wenn sich die Softwareentwicklung Wege geht, die von der Wissenschaft nicht gutgeheißen werden.

Kurseinheit 4: Daten

Get your data structures correct first,
and the rest of the program will write itself.

David Jones

Da die Macht der Computer in ihrer Programmierbarkeit liegt, liegt es nahe, dass man sich zunächst, wie in Kurseinheit 3 geschehen, mit der Programmierung befasst. Daten sind bei dieser Betrachtung vor allem Ein- und Ausgaben und den Programmen bei- oder gar untergeordnet. Mit komplexer werdenden Problemstellungen wird jedoch deutlich, dass eine günstige Organisation — oder Strukturierung — der Daten Programme entscheidend vereinfachen kann. *Algorithmen* und Datenstrukturen bilden daher in der Didaktik der Informatik, spätestens seit *Niklaus Wirths* gleichnamigem Buch, eine Einheit und je nach Zielgruppe kann es sogar sinnvoll sein, mit den Daten anzufangen. In diesem Kurstext erfolgt eine eigenständige Betrachtung von Daten jedoch erst in dieser Kurseinheit, zunächst noch mit Bezug zu Programmiersprachen (Kapitel 27) und dann als eigenständige Disziplin (Kapitel 28).

Mit dem Einzug von Computern in die Wirtschaft und ihre Betriebe wurde schnell klar, dass anstelle der Programme die Daten das eigentliche Investitionsgut sind: Ihre Erfassung und Speicherung mit einem Computer stellen einen Wert an sich dar, der den der sie bearbeitenden Programme in der Regel bei weitem übersteigt. Außerdem greifen verschiedene Anwendungsprogramme, die in einem Betrieb im Einsatz sind, in der Regel zumindest teilweise auf dieselben Daten zu (regelmäßig auf die sogenannten *Stammdaten*), die deswegen keinem einzelnen Programm zugeordnet werden können. Diese gemeinsame Datennutzung hat dazu geführt, dass die Daten aus den Programmen herausgelöst und in sog. *Datenbanken* ausgelagert werden (Kapitel 30), die nicht nur Programmen Zugriff auf die Daten gewähren, sondern auch unmittelbare Möglichkeiten der Datenerfassung und -auswertung bieten und damit beispielsweise einer Datenvisualisierung (Kapitel 32) dienen. Damit wird aber die Strukturierung der Daten endgültig von konkreten Algorithmen, wie sie von den die Daten verwendenden Programmen umgesetzt werden, unabhängig und zu einer eigenständigen Aufgabe, die man auch als Datenmodellierung bezeichnet (Kapitel 28).

Die Datenmodellierung geht von großer Regelmäßigkeit in einem Datenbestand aus, die man aber nicht immer vorfindet (so z. B. nicht bei Webseiten). Anstatt nun unregelmäßige

Daten über eine regelmäßige Struktur zu brechen und die Behandlung der Unregelmäßigkeiten den die Daten verarbeitenden Programmen zu überlassen, kann man die schwache Strukturierung auch schon auf Datenebene anerkennen und entsprechend auszeichnen (Kapitel 29). Dies führt allerdings zu anderen Arten von Datenbanken als den traditionellen. Verzichtet man gänzlich auf eine regelmäßige Struktur, kann man alles als Daten auffassen und auszuwerten versuchen; dies führt beinahe automatisch zu den Problemen, die heute (Stand 2017) publikumswirksam unter dem Begriff Big Data behandelt werden (Kapitel 31).

27 Datentypen und -strukturen

Wie wir schon gesehen haben (u. a. in Kapitel 10 sowie den Abschnitten 18.2, 18.3.1, 20.2 und 20.4), kann man Werte wie Zahlen, Wahrheitswerte oder Zeichenketten in *Typen* einteilen, wie etwa `integer`, `float`, `boolean` oder `string`. Durch Typisierung von Variablen, Funktionen und Prozeduren kann sichergestellt werden, dass diese stets nur (Daten-)Werte des gewollten Typs annehmen oder liefern (*Typfehler* können vermieden werden). Außerdem erlauben sie den Programmen, die sie verwenden, Speicher für die jeweiligen Werte zu reservieren. Aus Sicht eines Programms sind die Werte Daten, die es verarbeitet; die Typen der Werte sind somit **Datentypen**. Dabei definiert ein Datentyp für ein Programm nicht nur einen Wertebereich, sondern auch die Operationen, die auf den Werten durchgeführt werden können (also etwa die Addition für `integer` und `float` oder die *Konjunktion* für `boolean`).

Nun sind die Daten einer Anwendungsdomäne häufig stärker strukturiert als es sich durch elementare Datentypen wie `integer` oder `string` allein ausdrücken lässt. Eine Adresse beispielsweise ist ein Verbund aus solchen Datentypen: Sie setzt sich aus einem Straßennamen (`string`), einer Hausnummer (`integer`), einer Postleitzahl (`integer`) und einem Städtenamen (`string`) zusammen. Die Kinder einer Person werden vielleicht als Liste von Paaren, bestehend aus Name und Geburtsdatum, repräsentiert. Eine Personalakte schließlich enthält noch weitere Daten, die einer Person zugeordnet und die ggf. selbst weiter untergliedert sind. Auch solchermaßen strukturierte Daten, oder **Datenstrukturen**[72], können *typisiert* werden. Neben der Funktion, Typfehler zu entdecken und zu vermeiden (und um den von einem Programm für seine Daten benötigten Speicher zu organisieren), hat die Typisierung von Datenstrukturen eine ordnende Funktion: Sie gibt eine Struktur vor, an die sich die Daten halten müssen.

[72] Der Begriff der Datenstruktur bleibt hier bewusst etwas vage — es muss aus dem Kontext erschlossen werden, ob mit einer Datenstruktur ein Schema (Typ), eine Ausprägung eines Schemas (Wert) oder überhaupt nur ein strukturierter Wert gemeint ist (vgl. Kapitel 29).

27.1 Typ und Instanz

Wenn in den vorangegangenen Kurseinheiten von *Typen* die Rede war, dann waren damit die Typen von Werten gemeint: `integer`, `float`, `boolean` etc. Verallgemeinert man Werte zu Daten, dann erfährt auch der Begriff des Typs eine Verallgemeinerung und man spricht von Typen (oder *Datentypen*) und ihren **Instanzen**. Typen können dann ganz allgemeine Dinge bezeichnen wie `Mensch` oder `Fahrzeug`; ein bestimmter Mensch oder ein bestimmtes Fahrzeug sind dann Instanzen dieser Typen. Werden solche Instanzen von einem Typ in einem Computer erzeugt (als Datenabbilder realer Instanzen, letztlich dargestellt als Folge von Einsen und Nullen), dann spricht man von einer Ausprägung, oder **Instanziierung**, des Typs.

Typen klassifizieren („typisieren") nicht nur ihre Instanzen, sie dienen auch dazu, bestimmte Festlegungen zu treffen, die für alle Instanzen eines Typs gelten. Für den Typ `Mensch` könnte das etwa sein, dass seine Instanzen zwei Beine haben. Wenn man es dann mit einer Instanz des Typs `Mensch` zu tun hat, kann man davon ausgehen, dass sie zwei Beine hat; für Instanzen anderer Typen gilt das ggf. nicht. Typen geben somit ein *Schema* für Daten vor und was man mit ihnen machen kann.

Metatypen

Typen können selbst Instanzen von Typen sein. Man spricht dann von deren Typen als **Metatypen**. Man beachte, dass die Instanz-Beziehung nicht transitiv ist: Wenn *a* eine Instanz des Typs *b* ist und *b* eine Instanz des Typs *c*, dann folgt daraus nicht, dass *a* auch eine Instanz von Typ *c* ist (vgl. dazu Abschnitt 28.3: Sokrates ist Mensch, aber keine Spezies). Außerdem sollte ausgeschlossen sein, dass ein Typ Instanz von sich selbst ist — bei einer Interpretation von Typen als Mengen von Instanzen würde das nämlich implizieren, dass es eine Menge gibt, die sich selbst enthält, was letztlich zur *Russellschen Antinomie* (von der Menge aller Mengen, die sich nicht selbst enthalten) führt.

Typ- und Instanzebene

Basierend auf der Unterscheidung von Typ und Instanz kann man bei der Betrachtung von Daten zwei Ebenen unterscheiden: die **Typebene** und die **Instanzebene**. Auf der Instanzebene finden sich demnach nur Instanzen (zusammen mit den Beziehungen zwischen Instanzen) und auf der Typebene nur Typen (zusammen mit Beziehungen zwischen Typen). Da ein Typ auch eine Instanz sein kann (s. o.), verlangt eine strikte Ebenentrennung, dass auf der Instanzebene nur Instanzen von Typen der Typebene sind, die ggf. selbst Instanzen einer Metatypebene sind usw. Es ergibt sich daraus eine Schichtung von Ebenen, die weder nach oben noch nach unten abgeschlossen sein muss, und deren einzige schichtübergreifende Beziehung die Instanz-Beziehung (zwischen einer Instanz und ihrem Typ) ist. Eine abgeschlossene Einteilung in vier Schichten wurde von der ISO mit dem *Information Resource Dictionary System* (IRDS) Standard festgelegt (ISO/IEC 10027).

27.2 Typkonstruktoren

Datentypen, die man für die Typisierung von Datenstrukturen (wie der oben erwähnten Adresse) benötigt, lassen sich aus elementaren Typen konstruieren, und zwar mithilfe sog. **Typkonstruktoren**. Der Vorrat an Typkonstruktoren ist nicht normiert; dennoch hat sich über die Jahre ein Kanon gebräuchlicher Typkonstruktoren etabliert, von denen die meisten schon in der Programmiersprache *Pascal* zu finden sind (die hier daher noch einmal als Beispiel dienen soll). Ein solch etablierter Typkonstruktor ist der **Verbund** (engl. **record**). Er kommt in ähnlichen Formen sowohl in Programmiersprachen als auch in *Datenbeschreibungssprachen* (von *Datenbanken*; s. Kapitel 28) vor.

27.2.1 Datenverbünde (Records)

Mit einem Verbund lassen sich Daten konstruieren und typisieren, die aus mehreren elementaren oder konstruierten Daten zusammengesetzt sind. So wird (in Pascal-Notation) ein **Verbundtyp** (engl. **Record**) `Adresse` mittels

```
103 type Adresse = record
104   Strasse : string;
105   Hausnummer : integer;
106   Postleitzahl : integer;
107   Ort : string
108 end
```

vereinbart (deklariert); Werte dieses Verbundtyps sind dann Quartette, deren erstes Element, eine Zeichenkette, über den Namen `Strasse` ausgewählt wird, deren zweites über `Hausnummer` und so weiter. Werte solcher Verbünde können komplett zugewiesen werden wie in

```
109 meineAdresse := deineAdresse
```

oder elementweise wie in

```
110 meineAdresse.Strasse := deineAdresse.Strasse
111 meineAdresse.Hausnummer := deineAdresse.Hausnummer
```

usw. (wobei hier `meineAdresse` und `deineAdresse` Variablen des Typs `Adresse` sein sollen). Verbundtypen können auch geschachtelt sein:

```
112 type ErsteZeile = record
113   Strasse : string;
114   Hausnummer : integer
115 end;
116 type ZweiteZeile = record
117   Postleitzahl : integer;
118   Ort : string
119 end;
120 type Adresse = record
121   Zeile1 : ErsteZeile;
122   Zeile2 : ZweiteZeile
```

```
123 end
```

Die Elemente eines Verbundtyps, also hier `Strasse`, `Hausnummer` etc., werden auch *Felder* genannt; dies ist insofern problematisch, als im Deutschen mit „Feld“ auch Arrays bezeichnet werden (s. Abschnitt 27.2.3).

27.2.2 Rekursive Datentypen und Zeiger

Nicht zuletzt kann bei einem Verbundtyp der definierte Typ auch Teil der Definition sein, wie in

```
124 type Liste = record
125   Element : Objekt;
126   Rest : ^Liste
127 end
```

Hierbei sei `Objekt` der Typ der Elemente, die eine Liste aufnehmen kann. Einen solchen Datentyp nennt man **rekursiv**, weil er selbstbezüglich ist: `Liste` kommt in der Definition von `Liste` (an der mit `Rest` bezeichneten Stelle) selbst vor. Da Werte solcher rekursiven Datentypen beliebig groß (jede als Rest angegebene Liste kann selbst wieder einen Rest haben) und auch zirkulär sein können (z. B. wenn man eine Liste als Rest ihrer selbst angibt), brauchen sowohl die Deklaration als auch die Implementierung rekursiver Datentypen einen weiteren Typkonstruktor: den des *Zeigers*, hier angedeutet durch das Dach (`^`) in `^Liste`. Werte des Typs `^Liste` sind Zeiger auf Werte des Typs `Liste`.

Bei einem **Zeiger**, auch **Referenz** genannt, (engl. *pointer* oder *reference*) handelt es sich um eine *Speicheradresse* (virtuell oder real; s. Abschnitt 12.4 in Kurseinheit 2), an der ein Datum (Wert oder weiterer Zeiger) steht. Ein Zeiger verweist also auf ein Datum (weswegen er auch *Verweis* genannt wird). Damit die Verwendung des Datums durch seinen Typ abgesichert ist, hat ein Zeiger nicht einfach den Typ `Zeiger`, sondern den Typ `Zeiger auf Datentyp`, wobei `Datentyp` hier ein Platzhalter für einen beliebigen Datentyp (wie `Adresse` oder `Liste`; möglich ist aber auch ein Zeigertyp) ist. In manchen Programmiersprachen (wie *C* und *C++*) sind Zeiger Werte, mit denen man rechnen kann (die sog. **Zeigerarithmetik**); da man mit Zeigerarithmetik Zeiger auf nahezu beliebige Speicherareale eines Computers erzeugen kann (auf die man damit lesenden und schreibenden Zugriff erhält), ist sie inhärent *unsicher* und sollte damit gar nicht mehr angeboten werden.

Zeiger-dereferenzierung

Um über einen Zeiger auf ein Datum zugreifen zu können, muss der Zeiger **dereferenziert** werden. Dazu wird der Wert des Zeigers, der selbst veränderlich ist (der Wert kann sich zur *Laufzeit* des Programms ändern), als Adresse einer Speicherstelle interpretiert, an der dann der eigentlich interessierende Wert steht. Man beachte, dass dies gegenüber der sowieso schon erfolgenden *Indirektion* (beim Zugriff auf den Wert des Zeigers; vgl. Kapitel 9 in Kurseinheit 1) eine weitere Ebene der Indirektion darstellt (der Zugriff erfolgt also gewissermaßen *doppelt indirekt*). Im obigen Beispiel des Typs `Liste`, bei dem der Rest einer Liste wieder eine Liste ist, erfolgt die Dereferenzierung mittels

des Ausdrucks `Rest^`; dabei hat `Rest^` den Typ `Liste` (und nicht `^Liste`, der Typ von `Rest`).

Nullzeiger

Eine rekursive Datenstruktur, wie sie beispielsweise durch den obigen Datentyp `Liste` vorgegeben ist, muss irgendwo ein Ende haben — `Rest` kann nicht immer wieder auf eine neue Liste mit Rest verweisen. Ein gängiges Verfahren, rekursive Datenstrukturen zu terminieren, ist daher die Verwendung eines sog. **Nullzeigers** (engl. null pointer oder null reference), der nicht die Speicheradresse 0 adressiert, sondern für „kein Zeiger" (oder „Zeiger unbekannt"; vgl. Abschnitt 3.3.1) steht. Es ist ein *schwerwiegender Programmierfehler*, den Nullzeiger zu dereferenzieren — es bewirkt praktisch immer einen sofortigen *Programmabbruch*.[73]

27.2.3 Felder (Arrays)

Unter einem **Feld** (oder besser **Array**, denn der Name „Feld" wird für viele andere Dinge auch gebraucht, so ein Element eines Records; s. o.) versteht man eine Datenstruktur, deren Elemente nicht wie beim Verbund benannt, sondern nummeriert sind. Ein Quartett wie die obige Adresse als Array hat demnach vier Elemente, die über einen Index (eine ganze Zahl) ausgewählt werden. Dabei ist es nur eine Sache der Konvention (und der Implementierung von Arrays in einer Programmier- oder Datenbeschreibungssprache), ob das erste Element den Index 0 oder den Index 1 hat (und entsprechend, im Falle des Quartetts, das letzte den Index 3 oder 4).[74]

Damit genügend Speicher für die Aufnahme der Werte zur Verfügung gestellt werden kann, werden Arraytypen in der Regel unter Angabe des höchsten vorgesehenen Index und des Elementtyps deklariert. In *Pascal* gibt man zusätzlich noch den Startindex an:

```
128 type Kinder = array [1..10] of Person
```

Eine Datenstruktur von Typ `Kinder` kann demnach 10 Datenstrukturen vom Typ `Person` aufnehmen, die erste unter dem Index 1 und die letzte unter dem Index 10. Das erste und das zweite Kind tauschen durch die Zuweisungen

```
129 kind := meineKinder[1];
130 meineKinder[1] := meineKinder[2];
131 meineKinder[2] := kind
```

die Position, wenn `kind` eine Variable vom Typ `Person` ist und `meineKinder` eine vom Typ `Kinder`.

---

[73] Der Turing-Preisträger *Sir Charles Antony Richard Hoare*, der viele wichtige Beiträge zur Informatik geleistet hat, nennt wohl deswegen seine Erfindung der Null reference einen „billion dollar mistake".

[74] Es gibt gute Gründe für und gegen beide Alternativen und man muss sich immer erst schlau machen, mit welcher man es zu tun hat. Ich wüsste zu gerne, welchen Schaden (aufgrund einer irrigen Annahme von 0- oder 1-Basierung) diese Wahlfreiheit schon verursacht hat.

Wenn man (wie hier) die Anzahl der Elemente, die ein Array aufnehmen kann, bei der Deklaration fest vorgibt, dann spricht man von einem *statischen Array*, andernfalls von einem *dynamischen*. In beiden Fällen müssen nicht alle Elemente eines Arrays auch mit einem Wert belegt sein. Das ist insbesondere dann nicht der Fall, wenn ein Array erst nach und nach befüllt wird. Dabei passt sich die Größe eines dynamischen Arrays automatisch an die Bedürfnisse an (es ist immer groß genug für den größten bisher verwendeten Index), es sei denn, es steht dafür kein Hauptspeicher mehr zur Verfügung (in welchem Fall man einen *Out-of-memory-Fehler* erhält).

Arrays als Sicherheitsproblem

Arrays sind eine sehr weit verbreitete Datenstruktur. Da sie direkt auf den *Hauptspeicher* eines Computers abgebildet werden (man kann den Hauptspeicher als ein Array von Bytes betrachten), können Fehler im Umgang mit Arrays zu *Programmabstürzen* führen, nämlich, wenn man über die obere oder untere Indexgrenze eines Arrays hinaus in den Speicher schreibt und dabei nicht nur in andere Datenstrukturen hinein, sondern in Bereiche vordringt, in denen auszuführende Programme liegen. Deswegen wird in *sicheren Programmiersprachen* der Zugriff auf Arrays dadurch geschützt, dass vor einem Zugriff geprüft wird, ob der Index innerhalb des vorgesehenen Bereichs liegt. Leider zählen die u. a. für die Programmierung von Betriebssystemen verwendeten Sprachen *C* und *C++* nicht zu den sicheren, so dass diese Sprachen Angriffe auf einen Computer geradezu einladen: Man muss dazu nur mit einem Schadprogramm ein Array über seine Grenzen hinaus beschreiben und kann so Programme (als Daten) in Bereichen platzieren, an denen andere, noch auszuführende Programme (oder zumindest Sprungadressen solcher Programme) stehen. Wenn das Anlegen von Sicherheitsgurten im Straßenverkehr oder das Geheimhalten von Passwörtern im Büro gesetzlich vorgeschrieben werden kann, ist es schwer zu begründen, warum die Verwendung *unsicherer Programmiersprachen* für sicherheitsrelevante Programme heute noch erlaubt ist.[75]

27.2.4 Weitere Typkonstruktoren

Eine anderer weit verbreiteter Typkonstruktor ist der der **Aufzählung** der Werte des Typs, in Pascal etwa durch

```
132 type Wochentag = (Sonntag, Montag, Dienstag, Mittwoch, Donnerstag,
        Freitag, Samstag)
```

deklariert. Eine Variable vom Typ `Wochentag` kann demnach einen der sieben aufgezählten Werte annehmen, wobei diese Werte keine Zeichenketten (*Strings*) sind, sondern Symbole, die dem Programm oder Datenbanksystem, das diese Typdeklaration verwendet, durch die Typdeklaration bekannt gemacht werden. Man kann auch den Typ `boolean` als einen (fest

[75] Eine etablierte Abwehrmaßnahme gegen solche Angriffe stellt die sog. *Address Space Layout Randomization* (*ASLR*) dar, bei der nicht mit Sicherheit vorhersagbar ist, an welchen Stellen im Speicher auszuführende Programme stehen. Sie kann man aber mit dem sog. *Spraying*, also der Platzierung von Schadcode an allen möglichen Stellen, umgehen.

vorgegebenen) Aufzählungstyp begreifen (mit den Werten `true` und `false`); allerdings fehlen an einer Deklaration dieses Typs nach obigem Schema noch die Operationen, die für den Typ vorgesehen sind (die *logischen Verknüpfungen*).

Typkonstruktoren, die eine Datenstruktur mit den auf sie definierten Operationen verbinden, sind vor allem aus der *objektorientierten Programmierung* bekannt. Wichtige Typkonstruktoren sind hier *Klasse* und *Interface*, wobei diese eng mit den sog. abstrakten Datentypen verwandt sind.

27.3 Abstrakte Datentypen

Ein **abstrakter Datentyp** abstrahiert von der (internen) Repräsentation von Daten und spezifiziert stattdessen nur die Operationen, die auf den Daten durchgeführt werden können. Mit einem abstrakter Datentyp `Liste` etwa ist nicht festgelegt, ob die Elemente der Liste in einem *Array* oder mit *Zeigern* verkettet gespeichert werden, sondern, was man mit der Liste machen kann, also etwa ein Element hinzufügen oder löschen. Dabei wird nicht nur die *Syntax* der Operatoren spezifiziert, sondern auch ihre *Semantik* (letzteres, ohne dabei auf eine Programmiersprache zurückzugreifen).

Trennung von Spezifikation und Implementierung

Abstrakte Datentypen fördern die Trennung von *Spezifikation* und *Implementierung* und machen die Implementierung austauschbar. Die Wichtigkeit einer Spezifikation für die Korrektheit eines Programms hatten wir ja schon in Kurseinheit 3 (insbesondere Kapitel 17 und 18) kennengelernt; hier geht es hingegen um die Spezifikation von Komponenten eines Programms. Wenn man ein Programm in die Definition von mehreren abstrakten Datentypen zerlegen kann, die sich gegenseitig verwenden, dann erleichtert dies u. a. die *Verifikation von Programmen*: Man muss nämlich nur noch nachweisen, dass die Implementierung jedes abstrakten Datentyps seine jeweilige Spezifikation erfüllt.

abstrakte Datentypen in der objektorientierten Programmierung

Genau diese Zerlegung erfolgt bei der *objektorientierten Programmierung* (Abschnitt 19.4 in Kurseinheit 3): Ein objektorientiertes Programm besteht aus einer Menge von *Klassen*, von denen jede einen *Typ* definiert und, sofern die Klasse nicht als abstrakt deklariert ist, auch mit einer Implementierung versieht. Die *Interfaces* von Programmiersprachen wie *Java* und *C#* sind dagegen reine Spezifikationen, die durch Klassen implementiert werden; sie bilden abstrakte Typen (also Typen ohne Implementierung). Weder Klassen noch Interfaces sind jedoch abstrakte Datentypen im obigen Sinn: Dazu fehlt ihnen eine abstrakte Spezifikation der Operationen.

27.4 Wichtige Datentypen

Insbesondere die *Verknüpfung von Daten* mit den Operationen, die auf ihnen ausgeführt werden können, machen Datentypen zu mächtigen Konstrukten, die in der Programmierung eine wichtige Rolle spielen. Besonders populär sind dabei Datentypen, die in vielen verschiedenen Problemstellungen nützlich sind und deshalb immer wieder verwendet werden können. Sie werden typischerweise in *Bibliotheken* zusammengefasst, aus denen sich Programmierer bedienen können. Drei wichtige Datentypen werden im folgenden kurz vorgestellt.

27.4.1 Listen

Listen repräsentieren Folgen von Elementen, die man fallweise als Ganzes oder elementweise betrachten möchte. Dabei *reifiziert* eine Liste eine Folge von Objekten, d. h., die Folge ist selbst ein Objekt. Typische Operationen auf oder für Listen sind das Erzeugen einer (leeren) Liste, das Anfügen eines Elements am Anfang oder Ende einer Liste, das Einfügen an einer bestimmten Position der Liste sowie die komplementären Löschoperationen. Auch das Suchen eines Elements innerhalb einer Liste, das Umkehren der Reihenfolge der Elemente einer Liste oder das Anfügen einer Liste an eine andere können als Listenoperationen definiert sein. Zwar könnte ein Nutzer des Datentyps `Liste` solche Operationen leicht selbst (unter Verwendung der anderen Operationen) implementieren, aber dass er das nicht muss ist gerade der Vorteil eines vorgegebenen Listendatentyps.

externe und interne Iteration

Wie bereits erwähnt können Listen als Ganzes oder elementweise bearbeitet werden. Als Ganzes kann beispielsweise eine Liste eine andere ersetzten (bei einer *Zuweisung*), an eine Prozedur übergeben oder auf einem Datenträger gespeichert werden. Einzeln können Elemente beispielsweise verglichen (z. B. beim Sortieren) oder aggregiert (z. B. beim Aufaddieren) werden. Dabei erfolgt die elementweise Verarbeitung häufig mittels einer **Schleife**, in der die Elemente der Liste der Reihe nach besucht werden. Diese Schleife hat klassischerweise die Form einer sog. *For-Schleife*:

```
133 for i := 1 to liste.length do … end
```

wobei hier `liste` eine Variable vom Typ `Liste` sei (also eine Liste zum Inhalt hat) und `i` eine Laufvariable ist, die von 1 bis zur Länge der Liste (hier durch `liste.length` repräsentiert) hochgezählt wird. Innerhalb der Schleife kann man dann `i` verwenden, um auf das *i*-te Element der Liste zuzugreifen (mit einer Syntax, die von Sprache zu Sprache variiert). Dabei kann man das *i*-te Element lesen und schreiben (ersetzen); es zu löschen oder eines davor oder danach einzufügen würde die Länge der Liste und die Indizes der nachfolgenden Elemente betreffen und ist daher — wenn überhaupt erlaubt — mit Vorsicht zu genießen.

Für den häufigen Fall, dass die Elemente einer Liste einfach nur der Reihe nach besucht werden sollen, gibt es eine Variante der For-Schleife, die *For-each-Schleife*:

```
134 for each element in liste do … end
```

Hierbei werden im Schleifenrumpf, der zwischen `do` und `end` steht und der für jedes Element der Liste einmal durchlaufen wird, der Variable `element` der Reihe nach alle Elemente der Liste zugewiesen. Man spart sich also den Zugriff auf die Elemente über den Index, den man im Gegenzug aber auch nicht zur Verfügung hat (man weiß also im Schleifenrumpf nicht, an welcher Stelle der Liste das Element steht, das man gerade bearbeitet).

Da man mit den beiden obigen Konstruktionen in einer expliziten Schleife über die Elemente einer Liste iteriert, nennt man sie auch **externe Iterationen**. Die externe Iteration ist von der sog. **internen Iteration** abzugrenzen, bei der die Iteration keine Anweisung der Programmiersprache, sondern eine Operation des Datentyps `Liste` ist. Dieser Operation auf Listen gibt man als weiteren Operanden mit, was mit den Elementen der Liste zu tun ist. Sie hat die (je nach Programmiersprache variierende) Form

```
135 liste.do(…)
```

wobei hier die Punkte durch eine Prozedur, eine Funktion oder einen Anweisungsblock mit einem *Eingabeparameter*, der der Reihe nach die Elemente von `liste` aufnimmt, zu ersetzen sind. Interne Iterationen stammen aus der *funktionalen Programmierung* (s. Abschnitt 19.2); sie werden dort durch Abbildungen von Listen auf Listen (sog. Maps und Filter) und Aggregationen, wie man sie auch in *Datenabfragesprachen* (s. Abschnitt 30.1) vorfindet, ergänzt. Die extreme Nützlichkeit der internen Iteration für die Programmierung ist einer der Hauptgründe, warum heute praktisch alle *objektorientierten Programmiersprachen funktionale Anteile* haben. Auch prägt die Verwendung der internen Iteration einen eigenen *Programmierstil*.

27.4.2 Bäume

Während in einer Liste jedes Element außer dem ersten und dem letzten genau einen Vorgänger und einen Nachfolger hat, kann in einem **Baum** jedes Element — Knoten genannt — mehrere Nachfolger haben (aber immer noch höchstens einen Vorgänger). Der erste Knoten eines Baums (der genau wie das erste Element einer Liste keinen Vorgänger hat) nennt man seine **Wurzel**; die Knoten, die keine Nachfolger haben, nennt man seine **Blätter**. Aufgrund der Richtung (die Wurzel ist der erste Knoten) werden in der Informatik Bäume grafisch fast immer mit der Wurzel oben und den Blättern unten (also auf dem Kopf stehend, wie Stammbäume häufig auch) dargestellt. Übrigens: Wenn jeder Knoten eines Baums höchstens einen Nachfolger hat, dann hat man es mit einer Liste zu tun (weswegen Listen gelegentlich auch entartete Bäume genannt werden).

Bäume kommen in vielen Problemstellungen vor. So haben beispielsweise ein Inhaltsverzeichnis, der Inhalt einer Webseite oder auch die Menüstruktur eines Anwendungsprogramms Baumform; gleiches gilt (naheliegenderweise) für *Entscheidungsbäume*. Aber auch Strukturen, die man nicht auf den ersten Blick als solche betrachten würde, entpuppen sich bei genauerer Betrachtung als Bäume.

Ein besonders schönes Beispiel für eine verborgene Baumstruktur ist ein *Labyrinth*, das genau einen Eingang besitzt, der zugleich Ausgang ist und von dem aus man zu Verzweigungen gelangt, die allesamt, ggf. über weitere Verzweigungen, in einer Sackgasse enden.[76] Hierbei ist der Eingang des Labyrinths die Wurzel eines Baums, die Verzweigungen sind die inneren Knoten und die Enden der Sackgassen die Blätter. Damit es wirklich Baumform hat, müssen wir verlangen, dass das Labyrinth so beschaffen ist, dass es keine zwei direkten (also ohne umzukehren) Wege bis zum selben Knoten (Verzweigung oder Sackgasse) gibt, dass man also insbesondere nicht abkürzen oder im Kreis laufen kann. Wenn es aber Baumform hat, dann kann man zu seiner Eroberung auf einen ganzen Fundus von *Algorithmen* für Bäume zugreifen.

Beispiel Labyrinth

Der Sage nach gab es ein solches Labyrinth einst auf Kreta: das des *Minotaurus*. Diesen zu besiegen und den damit verbundenen Ruhm zu Lebzeiten zu genießen setzte zweierlei voraus: erstens den Minotaurus zu finden und zweitens aus dem Labyrinth wieder herauszukommen. Dies gelang schließlich *Theseus*, u. a. deswegen, weil *Ariadne* ihm einen Faden mitgegeben hatte, den er am Eingang befestigte und der ihm wieder hinauszufinden half. Mir ist nicht bekannt, ob überliefert ist, ob Theseus den Faden auch nutzte, um den Minotaurus zu finden, also systematisch das ganze Labyrinth ab- und dabei nicht im Kreis zu laufen, aber wenn das Labyrinth Baumform hatte, hätte er den Faden weder für das eine noch für das andere gebraucht: Er hätte einfach nur in das Labyrinth hineingehen und dann an jeder Abzweigung ganz links abbiegen müssen. Am Ende einer Sackgasse angelangt wäre er umgekehrt und wäre bei der nächsten Verzweigung (auf dem Rückweg) wieder links abgebogen und so weiter. Auf diese Weise hätte er den Minotaurus auf jeden Fall gefunden (zumindest wenn dieser sich nicht selbst im Labyrinth bewegte) und wäre auch wieder herausgekommen (wenn auch nur, nachdem er das restliche Labyrinth unnötigerweise auch noch abgelaufen wäre — für den direkten Rückweg ist tatsächlich ein Faden hilfreich). Hätte das Labyrinth Zyklen (d. h., wäre es kein Baum), hätte er zusätzlich den Weg markieren müssen, den er schon einmal gegangen ist, um ein Laufen im Kreis zu vermeiden — auch hier kann ein Faden (oder alternativ der Fundus an Algorithmen für beliebige Graphen) tatsächlich helfen.

27.4.3 Assoziativspeicher

Ein *Array* (Abschnitt 27.2.3) speichert mehrere Werte unter Verwendung ganzer Zahlen als Indizes; es assoziiert also einen numerischen Index mit einem Wert. So greift etwa `kinder[1]` beispielsweise auf das erste Element eines Arrays von Werten zu, das den Namen „kinder" trägt, `kinder[i]` dagegen auf das *i*-te Element (was auch immer der Wert von `i` ist). Ein *Verbund* (Abschnitt 27.2.1) dagegen speichert Werte unter Namen, die jedoch, anders als die Indizes eines Arrays, selbst keine Werte sind (also nicht vom Programm berechnet werden können), sondern fester Bestandteil des Programmtextes. So greift etwa

[76] Strenggenommen ist somit das ganze Labyrinth eine Sackgasse, da man nicht ohne zu wenden wieder aus ihm hinauskommt. Mit Sackgasse ist hier ein Weg gemeint, der ohne weitere Verzweigung endet.

`adresse.Ort` auf die Ortskomponente der Adresse zu, die in `adresse` gespeichert ist. Dabei ist der Name „Ort“ nicht wie oben „i“ der einer Variable und anstelle von `Ort` kann auch keine Variable (kein Platzhalter für einen beliebigen Namen) stehen. In der Praxis möchte man aber häufig beliebige Werte mit beliebigen Indizes assoziieren, also gewissermaßen auf die Werte eines Arrays über andere Werte als ganze Zahlen zugreifen, die dazu auch in Variablen gespeichert können sein sollen. So könnte man beispielsweise die Zählwerte von Spielkarten in einer Variable `Zählwert` so abspeichern wollen, dass den Zeichenketten „Bube“, „Dame“, „König“ und „Ass“ die Werte 2, 3, 4 und 11 zugeordnet werden. Hat man dann eine Variable `Karte` mit Wert `"Ass"`, dann würde der Zugriff `Zählwert[Karte]` den Inhalt des Arrays `Zählwert` an der Stelle `"Ass"` liefern. Dies gelingt mithilfe sogenannter **Assoziativspeicher**.

Assoziativspeicher verallgemeinern Verbünde und Arrays und können diese ersetzen: Ein Array begrenzt die Werte, mit denen assoziiert wird, auf ein Intervall von ganzen Zahlen und ein Verbund begrenzt sie auf eine Menge von Namen, die zudem auch noch als *Literale* im Programmtext stehen müssen. Damit sind Assoziativspeicher flexibler als Verbünde und Arrays. Insbesondere kann man wegen der fehlenden Festlegung auf die Menge der zulässigen Namen per Deklaration (wie bei einem Verbund) zur *Laufzeit* eines Programms weitere Elemente hinzufügen. Manche Programmier- oder Skriptsprachen (wie etwa *JavaScript*) erheben deswegen Assoziativspeicher zur universalen Datenstruktur.

Implementierung

Assoziativspeicher können durch eine eigene *Hardware* realisiert werden; diese ist jedoch recht aufwendig und entsprechend teuer. Viel häufiger findet man stattdessen sog. *assoziative Arrays*, das sind (*abstrakte*) *Datentypen*, die über ihre Operationen den assoziativen Zugriff auf Werte anbieten. Diese Datentypen findet man in zahlreichen *Bibliotheken*, häufig unter den Namen `Dictionary`, `Map` oder `HashMap`. Ihnen allen ist gemein, dass sie, im Vergleich zu einem Array, relativ viel Speicher brauchen und relativ langsam sind. *Assoziation* ist keine Stärke heutiger Computer.

28 Datenmodellierung

Die Befassung mit Datenstrukturen ist relativ kleinteilig und häufig an die Wahl eines konkreten *Algorithmus* oder gar an eine konkrete *Implementierung* gebunden. In vielen Problemstellungen sind aber nicht Algorithmen, sondern die Daten selbst zentral. In diesen Fällen ist es sinnvoll, die Repräsentation der Daten ihrer „Natur“ folgen zu lassen. Mit anderen Worten: Man sucht das *Schema* hinter den Daten, unabhängig von ihrer konkreten Verwendung.

Logik als Vorbild

Um ein Schema finden zu können, braucht man erst einmal ein „Mobiliar“, aus dem sich ein Schema bauen lässt. Philosophisch gesehen ist das eine Frage der *Ontologie*; in der Datenmodellierung hat man sich, wohl aufgrund ihrer Anleihen aus der *Logik* (s. Abschnitt 3.3.2), im Wesentlichen für eine kleine Anzahl zentraler Konzepte entschieden (und dabei leider die *zeitliche Dimension* vergessen; s. dazu Abschnitt 28.6):

- *Entitäten*, die für wohl abgegrenzte Objekte oder Individuen stehen,
- *Attribute oder Eigenschaften* dieser Entitäten und
- *Relationen oder Beziehungen* zwischen diesen Entitäten.

Ein typisches Beispiel für eine Entität ist eine bestimmte Person, für ein Attribut ihr Alter und für eine Relation die Freundschaft mit einer anderen Person. Dabei ist Entität zunächst ein eher exklusiver Begriff: Mit „2 kg Mehl" beispielsweise ist wohl kein Attribut einer Entität Mehl gemeint, schon weil Mehl keine Entität ist. Da Mehl aber auch kein Attribut und keine Relation ist, bleiben 2 kg Mehl bei der obigen „Möblierung" auf der Strecke. Ähnlich verhält es sich bei dem Sachverhalt „Sokrates ist ein Freund der Weisheit", wenn Weisheit keine Entität ist. Da es in der Datenmodellierung aber meistens um sehr viel konkretere Dinge geht, werden solche Fragen in der Regel pragmatisch beantwortet: Wenn die Entität Sokrates ein Freund der Weisheit ist und Freundschaft eine Relation zwischen zwei Entitäten, dann ist zweckmäßigerweise auch Weisheit eine Entität.

Vergleich zu Sprache

Auffällig ist bei obiger ontologischer Dreiteilung die Verwandtschaft zu einfachen Sätzen einer *natürlichen Sprache*, die, zumindest in hiesigen Sphären, ein ähnliches Mobiliar aufweist: Ein Satz drückt durch sein Prädikat eine Beziehung zwischen Objekten (in einfachen Sätzen durch Substantive vertreten) aus, denen über Adjektive Eigenschaften (Attribute) beigemessen werden. Dabei ist die Grammatik einer Sprache, was die Einordnung von Sokrates, Mehl und Weisheit angeht, vergleichsweise ignorant: Es sind alles Substantive — für deren Bedeutung interessiert sich die Grammatik nicht.

Wohl aber die Linguistik! Die unterscheidet bei den Objekte bezeichnenden Substantiven sehr wohl zwischen Namen, die eben Entitäten benennen (wie Sokrates) und Allgemeinbegriffen wie Mensch oder Pferd, die einen *Typ* von Entitäten bezeichnen. Die Informatik schlägt sich im damit verbundenen, philosophischen *Universalienstreit* aus pragmatischen Gründen auf die Seite der Nominalisten und geht so weit, dass sie Allgemeinbegriffe auf die Ebene der Typen hebt (vgl. Abschnitt 27.1).

WIKIPEDIA

28.1 Schema

Konkreter unterscheidet man in der Datenmodellierung zwischen einem **Schema**, das die Struktur der Daten festlegt, und einer *Ausprägung* (*Instanziierung*) des Schemas, die die Daten selbst repräsentiert. Ein Schema wird umgangssprachlich etwa durch den Satz

> Freundschaft ist eine zweistellige Relation zwischen Menschen.

repräsentiert, eine (ebenfalls umgangssprachliche) Ausprägung dieses Schemas ist der Satz

> Sokrates und Chairephon sind Freunde.

Bei ersterem handelt es sich um eine Aussage auf der *Schema-* oder *Typebene*, bei letzterem um eine Aussage auf der *Instanzebene* (vgl. Abschnitt 27.1). *Mensch* bezeichnet eine Entität auf der Schemaebene, einen *Typ* (oder *Entitätstyp*), *Sokrates* und *Chairephon* bezeichnen Entitäten auf der Instanzebene (Daten oder Instanzen). *Freundschaft* wird auf der Schemaebene als eine zweistellige Relation eingeführt, die auf der Instanzebene, hier mir *Sokrates* und *Chairephon* als Paar, ausgeprägt wird. Das Schema schreibt vor, dass Freundschaft nur zwischen Menschen bestehen kann; ein (Daten)Satz ist *wohlgeformt* (d. h., er entspricht dem definierten Schema), wenn er die Freundschaft zwischen zwei menschlichen Entitäten feststellt. Da Sokrates und Chairephon Menschen sind, ist der zweite Satz bezüglich des ersten wohlgeformt. Der ähnliche Satz

Sokrates und die Weisheit sind Freunde.

ist hingegen nicht wohlgeformt (er ist fehlgeformt), es sei denn, man zählt „Weisheit" zu den Menschen. Übrigens: Auch der Satz

Sokrates, Chairephon und Hermogenes sind Freunde.

ist strenggenommen nicht wohlgeformt, weil hier ein dreistelliges Prädikat (und nicht ein zweistelliges wie in der Definition von Freundschaft) ausgeprägt wird.

Sorten

Aufgrund möglicher Fehlinterpretationen verwendet man weder für die Schemadefinition noch für die Daten selbst natürliche Sprache. Gebräuchlich sind vielmehr einfache, an die Mengentheorie oder die (*sortierte) Prädikatenlogik* (und damit auch an die *logische Programmierung*; s. Abschnitt 19.3) angelehnte Notationen. So entspricht etwa ein Entitätstyp `Mensch` einer (mathematischen) *Sorte* ***Mensch*** oder auch einem einstelligen *Prädikat* ***Mensch***(x), das für all jene Entitäten x, die Mensch sind, zutrifft und für alle anderen nicht. Die Menge der Entitäten x, für die ***Mensch***(x) zu*trifft*, nennt man auch die **Extension** des Prädikats ***Mensch***.

Beziehungen oder Relationen

Wenn Entitätstypen einstellige Prädikate sind, dann können *Beziehungen* oder *Relationen* als zwei- und mehrstellige (je nach Relation) Prädikate aufgefasst werden. Und tatsächlich kann man eine Relation ***Freundschaft*** als Prädikat ***Freundschaft***(x, y) darstellen, das für alle Paare (x, y) von Entitäten, die befreundet sind, zutrifft (und für alle anderen nicht). Man kann (und wird in der Datenmodellierung) sogar noch weiter gehen und verlangen, dass sowohl x als auch y Entitäten vom Typ *Mensch* sein müssen, damit sie befreundet sein können — die beiden Stellen der Relation *Freundschaft* sind also *typisiert* und eine *Typprüfung* wird ***Freundschaft***(***Sokrates***, ***Weisheit***) zurückweisen. Das heißt natürlich nicht, dass alle Menschen automatisch befreundet sind, sondern nur, dass Nicht-Menschen nicht befreundet sein können (es sei denn, man überlädt die Relation ***Freundschaft***; s. Fußnote 77). Eine Relation in der Datenmodellierung ist eben auch mathematisch eine Relation: sie ist eine Teilmenge eines kartesischen Produkts (hier: ***Freundschaft*** $\subseteq$ ***Mensch*** $\times$ ***Mensch***).

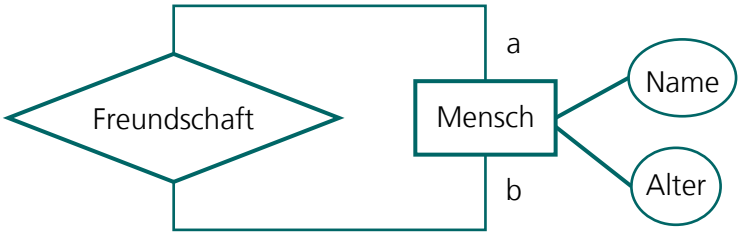
28.2 Datenmodelle

Es liegt auf der Hand, das Ergebnis einer Datenmodellierung, ein *Schema*, **Datenmodell** zu nennen. Allerdings hat die Informatik — unglückseligerweise — diesen Begriff mit einer zweiten Bedeutung überladen[77]: Man meint damit auch eine Festlegung auf das (ontologische) Mobiliar von Schemata. Diese Bedeutung ist auf der *Metaebene* angesiedelt (wo sie dann der ersten Bedeutung entspricht — das Mobiliar ist selbst Ergebnis eines Modellierungsprozesses).

Je nach Lehrmeinung gehören zu einem Datenmodell neben einer Sprache zur Schemadefinition auch eine Sprache zur Datenmanipulation (anlegen, ändern und löschen von Daten) und eine zur Abfrage oder Auswertung der Daten. In diesem Kurs werden diese Sprachen den *Datenbanken* zugerechnet und in Abschnitt 30.1 behandelt.

28.2.1 Das Entity-Relationship-Modell

Ein Klassiker der Datenmodelle im zweiten obigen Sinn ist das **Entity-Relationship-Modell**, das vor allem durch seine grafische Notation, die sog. **Entity-Relationship-Diagramme**, weite Verbreitung gefunden hat. In Entity-Relationship-Diagrammen stehen Rechtecke für *Entitätstypen*, Rauten für *Relationen* und Ovale für *Attribute*. Die obige Datendefinition (das Schema) von Menschen und Freundschaft liest sich in der Sprache der Entity-Relationship-Diagramme beispielsweise so:



Hierbei drücken die beiden Linien zwischen *Freundschaft* und *Mensch* aus, dass *Freundschaft* eine zweistellige Relation ist, deren beide Stellen mit Entitäten vom Typ *Mensch* besetzt sein müssen. Da beide Stellen der Relation den gleichen Entitätstyp haben, werden sie durch sog. *Rollennamen* an den Linien (hier *a* und *b*) unterschieden (andernfalls würde der Entitätstyp zur Unterscheidung der Stellen ausreichen).

28.2.2 Das relationale Datenmodell

Das Entity-Relationship-Modell ist verwandt mit dem sog. **relationalen Datenmodell**, das Basis der heute immer noch weit verbreiteten *relationalen Datenbanken* ist (s. Ab-

[77] In der Informatik spricht man von „überladen“, wenn derselbe Name unterschiedliche Bedeutungen hat. Überladung suggeriert keine Überlastung.

schnitt 30.1). Es unterscheidet allerdings nicht zwischen Entitätstypen und Relationen, sondern kennt nur Relationen, die hier auch als *Tabellen* bezeichnet werden. Die Spalten dieser Tabellen enthalten Werte primitiver Datentypen (wie `integer`, `float`, `boolean` und `string`); die Zeilen repräsentieren Entitäten, Beziehungen zwischen Entitäten, oder beides gemischt.

Das relationale Datenmodell ist stark mathematisch geprägt: Eine Relation ist demnach eine Teilmenge des kartesischen Produkts, das durch die Typen der Spalten vorgegeben wird; die Elemente einer solchen Relation werden **Tupel** genannt und setzen sich aus Werten dieser Typen zusammen. Eine Relation `Mensch` beispielsweise hätte hier (nach dem Schema des obigen Entity-Relationship-Diagramms) zwei Spalten, `Name` und `Alter`, vom Typ `string` bzw. `integer`; die Relation `Freundschaft` hätte ebenfalls zwei Spalten, `a` und `b`, deren Typ aber nicht `Mensch` ist, sondern `string`, weil `Mensch` kein gültiger Spaltentyp ist (`Mensch` ist nicht primitiv) und in diesem Schema Menschen durch ihren Namen eindeutig identifiziert werden (können). Man nennt ein solches identifizierendes Attribut auch **Primärschlüssel** und sein Vorkommen in einer anderen Tabelle **Fremdschlüssel**. Primär- und Fremdschlüssel dienen der Herstellung von Beziehungen[78] zwischen den Zeilen von Tabellen, die somit Entitäten, Beziehungen oder Mischformen davon darstellen können.

Mensch

Name	Alter
Sokrates	…
Chairephon	…
Hermogenes	…
…	…

Freundschaft

a	b
Sokrates	Chairephon
Chairephon	Hermogenes
…	…

Das relationale Datenmodell kann man als eigentlich obsolet betrachten — es ist viel zu primitiv, um Daten problemangemessen zu modellieren. Auch verlangt seine effiziente Verwendung in der Praxis einen riesigen Apparat und einiges an Wissen und Erfahrung. Gleichwohl stellt es einen etablierten Standard dar und die verfügbaren relationalen Datenbanksysteme sind so weit verbreitet (und arbeiten so zuverlässig), dass sie wohl noch auf Jahrzehnte Verwendung finden werden (nicht zuletzt bieten auch die Office-Suiten MS Office und OpenOffice mit Access bzw. Base einfache relationale Datenbanksysteme an). Bei neuen Projekten sollte man jedoch genau abwägen, was man gewinnt und was man sich einhandelt, wenn man auf relationale Datenmodellierung setzt.

28.2.3 Ältere Datenmodelle

Vorläufer des Entity-Relationship- und des relationalen Datenmodells waren das **Netzwerk-** und das **hierarchische Datenmodell**. Beide hatten durchaus ihre Berechtigung und ihre

[78] Ich verwende hier Beziehung anstelle von Relation, weil Relationen im relationalen Datenmodell sowohl Entitätstypen als auch Relationen (Beziehungen) zwischen diesen repräsentieren. Im Englischen wird dieser Unterscheidung durch Verwendung des Wortes Relationship Rechnung getragen.

(weitgehende) Ablösung ging vermutlich mit der Ablösung der Hard- und Software, auf bzw. in der sie zur Anwendung kamen, einher. Dabei besitzt das Netzwerkdatenmodell Gemeinsamkeiten mit dem Datenmodell, das der *objektorientierten Programmierung* zugrunde liegt (beide basieren auf verzeigerten Strukturen) und das in Form sog. Klassendiagramme der *Unified Modeling Language* (*UML*) eine grafische Normierung gefunden hat. Insbesondere überall dort, wo objektorientiert programmiert wird, ist also die Verwendung eines solchen Datenmodells überlegenswert.

28.2.4 Semantische Datenmodelle

In einfachen Datenmodellierungssprachen wie dem Entity-Relationship-Modell gibt es nur sehr wenige vorgegebene Konstrukte, mit denen man Daten beschreiben kann: Entitäten, Relationen und Attribute. Bei der Verwendung dieser Sprachen sieht man sich dann genötigt, bestimmte, immer wieder vorkommende Muster immer wieder mit denselben einfachen Mitteln darzustellen. Dabei ist die Bedeutung dieser Muster, anders als die von Entitäten, Relationen und Attributen, nicht normiert. Die Sprachen sind unter diesem Gesichtspunkt zu ausdrucksschwach.

Teil-Ganzes-Beziehung

Ein typisches Beispiel für eine immer wieder vorkommende Relation, die im Entity-Relationship-Modell nicht bekannt ist, ist die **Teil-Ganzes-Beziehung**, auch **Aggregation** oder **Komposition** genannt. Viele Entitäten bestehen nun einmal aus Teilen, die wiederum selbst aus Teilen bestehen können usw. Zwar kann man diese Beziehung zunächst wie jede andere Relation darstellen, aber mit ihr ist häufig eine besondere Semantik verbunden: So hat sie häufig *Baumform* (keine einzelne Entität kann Teil verschiedener Ganzer sein) und manchmal besteht eine Existenzabhängigkeit — das Ganze kann nicht ohne seine Teile existieren. Um solche immer wiederkehrenden Sonderfälle angemessen abbilden zu können[79] bieten sog. **semantische Datenmodelle** einen größeren Vorrat an Konstrukten, als ihn einfache Datenmodelle vorsehen.

Is-a-Beziehung

Eine andere Beziehung, die man in semantischen Datenmodellen immer wieder antrifft, ist die **Subtyp**- oder **Vererbungsbeziehung**, auch **Is-a-Beziehung** genannt (wobei letztere Bezeichnung irreführend ist; s. dazu Abschnitt 28.2.4). Sie unterscheidet sich von der Teil-Ganzes-Beziehung und auch von allen anderen, „normalen" Relationen dadurch, dass ihre *Extension* nicht aus Paaren von Entitäten, sondern aus Paaren von *Entitätstypen* besteht: *Mensch is-a Säugetier* etwa sagt nicht aus, dass irgendein Mensch irgendein Säugetier ist, sondern dass alle Menschen Säugetiere sind oder, anders ausgedrückt, dass die Menge der Säugetiere die Menge der Menschen enthält. Es handelt sich also um eine Beziehung zwischen Typen und damit um eine auf der *Metaebene*.

---

[79] „Angemessen" ist hier sicherlich relativ: Die *Mereologie* füllt ganze Bücherregale, ohne zu einem Konsens zu gelangen, den man in einem semantischen Datenmodell umgesetzt sehen würde.

28.3 Ontologien

Ontologie als Metamodell

Eine naheliegende Interpretation des aus der Philosophie entlehnten Begriffs **Ontologie** im Kontext der Datenmodellierung ist die Befassung mit dem „Mobiliar" eines Datenmodells, also ob es darin so etwas wie Entitäten, Relationen und Attribute geben soll und ob man darüber hinaus noch weitere Konstrukte (wie Aggregation oder Subsumtion) zur Modellierung anbieten möchte. Die Ontologie würde demnach ergründen, was ein Datenmodell (im Sinne von Abschnitt 28.2) ausmacht; ihr Gegenstandsbereich wäre damit die **Metamodellierung**, also die Bildung von (Daten-)Modellen von (Daten-)Modellen. Eine solche Ontologie definiert damit *Metatypen*, also Typen, deren *Instanzen* selbst wieder Typen sind; ein bestimmtes Schema wird aus so einer Ontologie per *Instanziierung* abgeleitet (Abschnitt 27.1).

Ontologie als Referenzmodell

Stattdessen spricht die Informatik aber häufig von Ontologie im Plural. Mit diesen **Ontologien** gemeint sind **Referenzmodelle**, d. h., standardisierte Modelle eines Gegenstandsbereichs wie beispielsweise der Warenwirtschaft oder der Medizin. Elemente dieser Ontologien sind dann Entitätstypen wie `Konto` oder `Patient` sowie Relationen zwischen und Attribute von solchen Entitätstypen, und zwar solche, die man in allen Problemstellungen des mit der Ontologie verbundenen Gegenstandsbereichs antreffen wird. Ein Schema für eine bestimmte Problemstellung wird dann per *Spezialisierung* und nicht per *Instanziierung* aus einer solchen Ontologie (oder einem solchen Referenzmodell) abgeleitet. Ein Beispiel hierfür aus dem Bereich des *kulturellen Erbes* ist das *Conceptual Reference Model* des *Comité international pour la documentation*, *CIDOC CRM*.

Link

Ontologien dieser zweiten Sorte werden manchmal durch sog. **Upper ontologies** auf eine gemeinsame Basis gestellt. Diese umfassen sehr allgemeine Entitätstypen wie `Entität`, `Sache`, `Qualität` oder `Rolle`, von denen dann die Typen einer Ontologie per *Spezialisierung*, *Subtyp-* oder *Subklassenbeziehung* (und wieder nicht per *Instanziierung*) ableitet werden. Die inhaltliche Abgrenzung einer Upper ontology von einer Ontologie im erstem obigen Sinn (als Metamodell) ist schwierig; technisch gelingt sie, wenn man den Ableitungsmechanismus (Instanziierung vs. Spezialisierung) zur Unterscheidung heranzieht.

trügerische Is-a Beziehung

Interessanterweise verschwindet die hier dargestellte Doppeldeutigkeit des Begriffs der Ontologie in der Informatik, wenn man Instanziierung und Spezialisierung durch eine einheitliche Beziehung ersetzt. Eine solche Beziehung, häufig *Is-a* genannt, ist aber im philosophischen (einschließlich dem mathematischen bis zur vorletzten Jahrhundertwende) Diskurs durchaus geläufig: „Sokrates ist ein Mensch" etwa und „Mensch ist ein Lebewesen" scheinen zunächst dieselbe Beziehung auszudrücken, zumal man aus den beiden Aussagen ja auch noch „Sokrates ist ein Lebewesen" ableiten kann. Augenfällig wird ein Unterschied erst, wenn man die dritte Aussage „Mensch ist eine Spezies" hinzunimmt — nach obiger Schlussfigur müsste auch „Sokrates ist eine Spezies" folgen, aber das stimmt ja nun nicht. Die Schöpfung, die diesen Trugschluss entlarvte, war die der (mathematischen) Menge und der damit zusammengehenden Unterscheidung der Ele-

ment- (∈) und der Teilmengenbeziehung (⊂): Sokrates ist ein Element der Menge der Menschen, die Menge der Menschen ist eine Teilmenge der Menge der Lebewesen und Mensch ist ein Element der Menge der Spezies. Die Instanziierung entspricht der Element-, die Spezialisierung der Teilmengenbeziehung. In der (häufig nicht formal betriebenen) Diskussion von Ontologie, in der man die Is-a-Beziehung immer wieder antrifft, bleibt diese Unterscheidung eben manches Mal auf der Strecke.

28.4 Begriffssysteme

Während die Ontologie sich mit den Grundlagen der Wissensrepräsentation befasst, ist es für die Praxis oftmals ausreichend, sich auf ein gemeinsames Vokabular zu einigen. Dieses besteht zweckmäßigerweise nicht nur aus einer bloßen Auflistung von Begriffen, sondern setzt diese ins Verhältnis zueinander. Man spricht dann von einem **Begriffssystem**.

Begriffssysteme folgen stets einer vorgegebenen Ordnung. So hatte beispielsweise *Leibniz* die Idee, primitive oder atomare Begriffe (also solche, die nicht selbst zusammengesetzt sind) durch Primzahlen zu repräsentieren und zusammengesetzte Begriffe als Produkte, wodurch sich jeder Begriff eindeutig (per Primteilerzerlegung) auf seine primitiven Anteile zurückführen lassen ließe. Das Motiv hinter seiner *Characteristica universalis* war wohl, dass man so mit Begriffen würde rechnen können, was ihm dem Ziel seines *Calculus ratiocinator* näherbrachte (für den er wohl auch seine Rechenmaschine und damit schließlich das *Dualsystem* ersann; s. dazu auch Abschnitt 2.3 in Kurseinheit 1). Das „Rechnen" mit Begriffen wird heute jedoch eher mit der (mathematischen) Logik (s. Kapitel 3 in Kurseinheit 1) verbunden.

WIKIPEDIA

Systematik

Gleichwohl spielt in Begriffssystemen eine **Systematik** der Begriffe eine wichtige Rolle. So werden beispielsweise bei der Verschlagwortung von Dokumenten in Bibliotheken und Archiven oder bei der Abrechnung medizinischer Leistungen Begriffssysteme verwendet (*Systematik für Bibliotheken*, SfB, und *International Classification of Diseases*, ICD) die in der Regel als Hierarchie von Ober- und Unterbegriffen aufgebaut sind. Auf diese Weise lassen sich auch Verwandtschaftsbeziehungen zwischen verschiedenen Begriffen (über gemeinsame Oberbegriffe) ableiten, ohne dass diese mit den Begriffen direkt verbunden sein müssten — sie ergeben sich aus der Systematik.

Link

Link

kontrolliertes Vokabular

Eine Vorstufe von Begriffssystemen stellen sog. **kontrollierte Vokabulare** dar; mit ihnen werden die zu verwendenden Termini einer Domäne normiert, um durch unterschiedliche Wortwahl entstehenden Missverständnissen vorzubeugen. Kontrollierte Vokabularien sollten daher möglichst auch in der *Anforderungserhebung* zu Softwareprojekten (s. Kapitel 17) verwendet werden. Dort findet man sie dann auch häufig, und zwar in Form eines *Glossars*.

28.5 Datenmodellierung mit Grammatiken

Obwohl mit Begriffssystemen in gewisser Weise verwandt werden *Grammatiken* üblicherweise nicht mit der Datenmodellierung in Verbindung gebracht. Dabei werden Grammatiken, in der Linguistik auch als *Phrasenstrukturgrammatiken* bekannt, dazu verwendet, um die Struktur in Sätzen einer (natürlichen oder künstlichen) Sprache zu erkennen oder wohlstrukturierte Sätze der Sprache zu erzeugen. Die Grammatik legt also die Struktur einer Sprache fest, die, wenn man Sätze als Daten auffasst, eine Datenstruktur ist. Letzteres gilt insbesondere für Programmiersprachen — wie bereits in Kapitel 10 (Kurseinheit 1) bemerkt, sind Programme auch Daten. Grammatiken können somit auch der Datenmodellierung dienen.

Tatsächlich besteht eine enge Verwandtschaft zwischen einem bestimmten Grammatiktyp, den sog. *attributierten* oder *Attributgrammatiken*, und *objektorientierten* oder *semantischen Datenmodellen* (s. Abschnitt 28.2.4). Diese Verwandtschaft wird sichtbar, wenn man die *Nichtterminale* der Grammatik (die im wesentlichen den Phrasen entsprechen, also so etwas wie Nominalphrase, Verbalphrase oder auch ganzer Satz) als *Typen* auffasst, die in einem konkreten Satz ausgeprägt werden (s. Abschnitt 27.1) und deren *Attribute* den Attributen der Grammatik entsprechen. Die Grammatik definiert auf den Typen eine *Teil-Ganzes-Beziehung* (ein konkreter Satz besteht beispielsweise aus einer Nominalphrase und einer Verbalphrase) und die Werte der Attribute werden aus den *Terminalen* des Satzes (den den Satz ausmachenden Wörtern) abgeleitet. Dabei geht eine Attributgrammatik über ein semantisches Datenmodell insofern hinaus, als sie eine *Instanz* eines solchen Modells in eine lineare Form, die Sätze, überführt bzw. aus einer linearen Form eine Instanz des Modells bildet. So werden u. a. die Sätze, die Programme darstellen, von einem *Compiler* (s. Abschnitt 18.5 in Kurseinheit 3) in Instanzen eines semantischen Datenmodells überführt, das dann Grundlage der Erzeugung von *Maschinencode* ist.

WIKIPEDIA

28.6 Zeitliche Dimension

Während ein Datenmodell von Natur aus statisch ist (Änderungen am Schema, also auf der Typebene, über die Zeit werden als *Schemaevolution* bezeichnet und fallen unter die *Wartung* eines Systems), sind seine *Ausprägungen* (*Instanziierungen*), also die nach ihm strukturierten Daten, von Natur aus dynamisch: Instanzen werden angelegt und auch wieder gelöscht, genau wie Beziehungen zwischen diesen. Diese Änderungen im Datenbestand entsprechen Änderungen der Werte von Variablen in einem *imperativen Programm* (wobei das Programm gewissermaßen das Schema der Daten definiert); man kann sie als *Zustandswechsel* einer Datenbasis auffassen. Dabei ist, genau wie in einem Programm, immer nur der aktuelle Zustand sichtbar: Jede Änderung überschreibt das, was vorher war, ohne eine Spur zu hinterlassen — ohne weitere Maßnahmen sind Daten „geschichtslos".

temporale Datenmodelle

Nun verlangen aber viele Problemstellungen einen Zugriff auf die Historie von Daten, also beispielsweise die Beantwortung von *Anfragen* wie „Wie

viele Menschen lebten in einer Region x im Jahre y?". Zu diesem Zweck hat man sog. **temporale Datenmodelle** eingeführt, die von *temporalen Datenbanken* umgesetzt werden.

Bei temporalen Datenmodellen kann man verschiedene Dimensionen der Zeit vorsehen, etwa wann (zu welchem Zeitpunkt) ein Datum eingegeben wurde oder wann (oder in welchem Zeitraum) es Gültigkeit haben sollte. Dabei können temporale Datenmodelle eine automatische Versionierung vorsehen, wodurch Änderungen bestehende Daten nicht überschreiben (oder löschen), sondern fortschreiben (also, mit entsprechenden *Zeitstempeln* versehen, ergänzen).

Obwohl die (angemessene Behandlung von) Zeit für viele praktische Problemstellungen zentral ist, ist die Standardisierung hier weit weniger fortgeschritten als in anderen Bereichen der Datenmodellierung. Die bedauerliche Konsequenz ist, dass sich viele selbst an einer Abbildung der Zeit auf die Konstrukte eines zeitlosen Datenmodells versuchen, was ihnen aber nicht nur bei der fehlenden besonderen Unterstützung der zeitlichen Dimension bei *Datenabfragen* auf die Füße fällt, sondern auch die Zusammenführung von temporalen Datenbeständen aus verschiedenen Quellen erschwert. Entsprechendes gilt auch für die räumliche Dimension, also etwa die Abbildung geographischer Regionen auf Datenbanken.

Kurs

29 Schwach strukturierte Daten

Während Datentypen und -schemata feste Leisten vorgeben, über die die Daten einer Problemstellung geschlagen werden müssen, sind Daten manchmal nur schwach (oder unregelmäßig) strukturiert, so dass sie sich nicht (oder nicht richtig) in ein einheitliches Format bringen lassen. Dies ist regelmäßig bei (multi-)medialen Daten wie Texten, Bildern oder Tonkonserven der Fall. Gleichwohl sollen solche Daten nicht nur archiviert, sondern auch bearbeitet und ausgewertet werden können. Je nach Bearbeitung oder Auswertung ist dafür eine gewisse Strukturierung notwendig, die allerdings auch fallweise und erst im Rahmen der Bearbeitung oder Auswertung selbst erfolgen kann.

Die Lösung liegt hier in der Abkehr von starren Strukturen, wie sie durch Datenmodelle oder Schemata vorgegeben werden. Stattdessen wird jeder Datensatz, den man in diesem Kontext auch *Objekt* nennt, einzeln strukturiert. Die unmittelbare Konsequenz dessen ist, dass diese Datensätze ihre Struktur in sich tragen, sich also gewissermaßen selbst beschreiben müssen.

29.1 Attribut/Wert-Paare

Die einfachste Form selbstbeschreibender Daten sind die sog. *Label/value-Paare* (auch *Name/value-* oder *Key/value-Paare* genannt): Sie ordnen einem Namen einen Wert zu. Interpretiert man die Namen als Namen von *Attributen* eines Objekts (oder einer Entität), dann

erhält man **Attribut/Wert-Paare**, die, da sie ein Objekt beschreiben, eigentlich Tripel der Form (*Objekt*, *Attribut*, *Wert*) sind. So beschreibt die Menge von Attribut/Wert-Paaren

```
{ Straße = "Universitätsstraße", Hausnummer = 1,
    Postleitzahl = 58097, Ort = "Hagen" }
```

eine Adresse (als Objekt). Man könnte auch sagen, dass die Menge ein Objekt vom Typ `Adresse` (beispielsweise durch einen Verbund repräsentiert; s. Abschnitt 27.2) beschreibt, aber in diesem Kapitel gehen wir ja von schwach strukturierten Daten aus und damit davon, dass wir keine Typen haben, die die Struktur (Attribute) von Objekten vorgeben (tatsächlich würde ein Typ `Adresse` ja die Namen der Attribute festlegen, so dass sie in obiger Darstellung redundant wären). Und so kann

```
{ Postfach = 58084 }
```

ebenfalls eine Adresse beschreiben. Das Problem, beides als Adresse zu erkennen und zu behandeln, bleibt dem verarbeitenden Programm überlassen, das mit einem Datensatz wie

```
{ Phantasie = "Unsinn" }
```

vermutlich nichts anfangen kann: Programme, die solchermaßen schwach strukturierte Daten verarbeiten, orientieren sich an (den Namen von) Attributen, die ihnen bekannt sein müssen, damit sie ihnen eine Bedeutung beimessen können. Insofern legen die Programme, die Attribut/Wert-Paare verarbeiten, implizit Schemata fest; was passiert, wenn sich die Daten nicht an ein implizites Schema halten, ist ebenfalls in den Programmen festgelegt.

Vor- und Nachteile

Die Vorteile einer solchen Selbstbeschreibung liegen auf der Hand: Jeder Datensatz definiert seine eigene Struktur, wodurch eine maximale Flexibilität ermöglicht wird.[80] Außerdem müssen Daten und die Programme, die sie verarbeiten, nicht zwingend ein gemeinsames Schema verwenden und es können Daten aus Quellen verarbeitet werden, die mit den verarbeitenden Programmen in keinem unmittelbaren Zusammenhang stehen (wobei ein mittelbarer Zusammenhang über die Verwendung und Bedeutung von Attributnamen kaum zu vermeiden ist). Ein offensichtlicher Nachteil ist, dass bei gleichförmigen Massendaten sehr viel Speicherplatz für die Selbstbeschreibung verschwendet wird — sofern die Namen und Reihenfolge der Attribute für alle Datensätze gleich sind, könnte man sie auch weglassen oder nur einmal bekanntgeben (wie das beispielsweise beim *Comma-separated-value-Format CSV* der Fall ist).

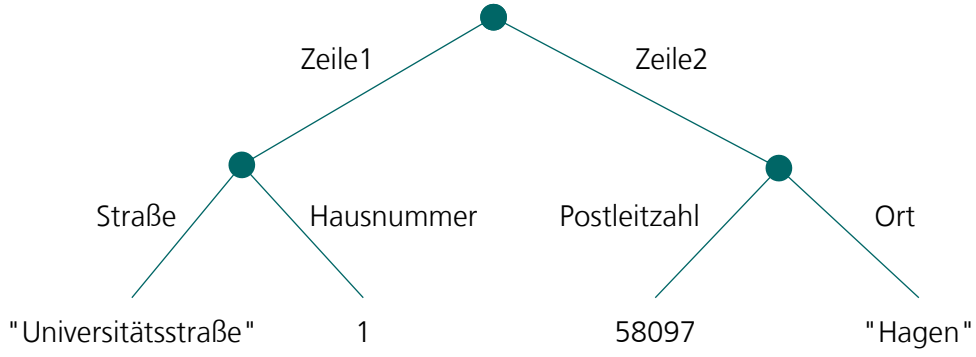
Schachtelung

Mengen von Attribut/Wert-Paare können auch geschachtelt werden. Dabei ist dann der Wert eines Attributs selbst wieder eine Menge von Attribut/Wert-Paaren, wie in

[80] Interessanterweise ist eine solche individuelle, oder objektweise, Strukturierung von Daten auch Basis der *prototypenbasierten Programmierung*, wie sie beispielsweise *JavaScript* zugrunde liegt.

```
{ Zeile1 = { Straße = "Universitätsstraße", Hausnummer = 1 },
    Zeile2 = { Postleitzahl = 58097, Ort = "Hagen" } }
```

Man erhält so einen *Baum*, dessen *Kanten* die Namen der Attribute tragen, dessen innere *Knoten* die durch die Attribut/Wert-Paare beschriebenen Objekte und deren *Blätter* die primitiven Attributwerte sind:



Eine solche Schachtelung von Attribut/Wert-Paaren liegt auch der *JavaScript Object Notation* (*JSON*) zugrunde, die ursprünglich der Serialisierung von Objekten der Scriptsprache *JavaScript* diente, die aber immer häufiger zum Datenaustausch verwendet wird. Dabei sind Objekte in JavaScript keine *Instanzen* von Typen, sondern *Prototypen*. Die Verwendung von JSON für die Auszeichnung schwach strukturierter Daten ist also durchaus konsequent.

Wenn man als Werte von Attributen auch (*Verweise* auf) Objekte zulässt, kann man mit geschachtelten Mengen von Attribut/Wert-Paaren nahezu beliebige Datenstrukturen nachbilden: Sie bilden ein *Graph*, dessen Knoten Objekte oder Werte und dessen Kanten Attribute sind. Die Frage ist allerdings, wie man Verweise auf Objekte oder Objekte selbst textuell repräsentiert — in der Regel werden den Objekten dazu sogenannte **Identifier**, das sind innerhalb eines Kontextes eindeutige Zeichenketten (die im Wesentlichen einem *Primärschlüssel* aus dem relationalen Datenmodell entsprechen; s. Abschnitt 28.2.2), zugewiesen. Diese nehmen dann auch die Stelle des Objekts in **(*Objekt*, *Attribut*, *Wert*)**-Tripeln ein. Solche Tripel bilden ein universelles Format der *Wissensrepräsentation*; sie werden beispielsweise für das *Resource Description Framework* (RDF) verwendet.

29.2 Auszeichnung

Eine mit den Attribut/Wert-Paaren verwandte Form der Selbstbeschreibung ist die sog. **Auszeichnung** (engl. **Markup**), die zunächst auf Texte, aber auch auf Daten allgemein angewendet wird. So lässt sich beispielsweise in einem Text, der als eine *Zeichenkette* vorliegt, ausdrücken, welcher Teil den Titel, welche Teile die Überschriften und welche die Absätze darstellen. Dazu werden in den Text spezielle Markierungen eingefügt, die den markierten Textfragmenten ihre Bedeutung zuordnen. Die Markierung kann dann genutzt werden, um den Text zu formatieren, d. h., satztechnisch in eine den Lesegewohnheiten entsprechende Form zu bringen, die den jeweiligen Bedeutungen der Textfragmente (Titel, Überschrift, Absatz) Rechnung trägt.

Dasselbe Markup kann aber auch anderen Zwecken dienen, beispielsweise um den Titel eines Textes zu ermitteln oder ihn in seine Kapitel aufzuteilen und so den Text seiner Struktur entsprechend zu verarbeiten (beispielsweise in eine Datenbank einzutragen). Wenn man nun noch Text zu Daten verallgemeinert, wird klar, dass man ein Markup verwenden kann, um Attribut/Wert-Paare zu codieren, also beliebige selbstbeschreibende Daten zu repräsentieren. Ein Vorteil gegenüber anderen Datenrepräsentationen ist dabei, dass die Daten — auch in Rohform — von Menschen gelesen werden können: Es handelt sich dabei stets nur um ausgezeichneten Text.

Auszeichnungssprachen

Die Auszeichnung erfolgt häufig nach bestimmten syntaktischen Regeln. So besagt eine Regel etwa, dass auf einen öffnenden Tag (zur Markierung des Beginns einer Fußnote etwa) ein schließender Tag (zur Markierung des Endes) folgen muss. Gemeinsam mit der Beschreibung der möglichen Auszeichnungen (Tags) bilden diese Regeln dann eine **Auszeichnungssprache** (**Markup language**). Der Vorteil der Definition und Verwendung von Auszeichnungssprachen ist, dass man mithilfe ihrer Regeln Fehler in einer Auszeichnung finden kann.

Standardisierung

Für die Auszeichnung von Text sowie allgemein von Daten haben sich verschiedene Standards etabliert. Einer der heute bekanntesten ist die *Hypertext Markup Language* (*HTML*), mit der der (textuelle) Inhalt von Webseiten logisch strukturiert wird (indem Teilen des Textes durch die sog. HTML-Tags eine Bedeutung zugewiesen wird, die von Webbrowsern verstanden wird). In einem Anflug von Wortwitz wird eine sehr einfache, alternative Markup-Sprache für Texte *Markdown* genannt; Varianten davon werden von Wikipedia und anderen webbasierten Gemeinschaftsprojekten verwendet. Für die Strukturierung beliebiger, als Zeichenketten vorliegender Daten wird die **Extensible Markup Language** (**XML**) eingesetzt, die nicht nur die Auszeichnung dieser Daten, sondern auch die Spezifikation einer zugrundeliegenden Struktur (einer Datenstruktur) erlaubt (ein sog. *XML Schema*). Jenseits des Webs spielen aber auch andere Auszeichnungsstandards eine Rolle. Da es sich bei all diesen Standards um leichte Variationen derselben Idee handelt, soll an dieser Stelle auf keinen weiter eingegangen werden. Insgesamt lässt sich konstatieren, dass der Wirbel, der um ihre Einführung jeweils gemacht wurde, größtenteils aus heißer Luft bestand: Es handelt sich im Wesentlichen um neue Kleider für die guten alten Attribut/Wert-Paare aus Abschnitt 29.1. Dass so viel Wirbel darum gemacht wurde ist insofern erklärlich, als der Wert solcher Standards nicht in ihrer Existenz, sondern in ihrer Verbreitung liegt.

WIKIPEDIA

WYSIWYG

Was für Menschen als gut lesbar gilt ist relativ, und so gibt es für viele Auszeichnungssprachen spezielle *Editoren*, die die Auszeichnung hinter der Darstellung verbergen. In solchen Editoren kann man dann beispielsweise eine Textpassage auswählen und per Mausklick als fett oder kursiv formatieren, was sich sofort in der Darstellung der Passage im Editor niederschlägt. Ein solcher sogenannter **WYSIWYG-Editor** (WYSIWYG für „what you see is what you get") speichert den Text dann mit der entsprechenden Auszeichnung, die der Benutzer freilich nie zu Gesicht bekommt. Für komplexere Auszeichnungssprachen ist WYSIWYG jedoch schwierig umzusetzen und um Fehler in der Auszeichnung zu finden, muss man sich dann häufig doch die Tags ansehen. Gute Beispiele für WYSIWYG-Editoren sind übrigens Textverarbeitungsprogramme wie Word oder Write; für stark strukturierte

WIKIPEDIA

Dokumente müssen sie aber auch Ansichten mit expliziten Auszeichnungen zur Verfügung stellen, die freilich von den meisten Nutzern eher gemieden werden. Für solche Dokumente sind dann spezielle Auszeichnungssprachen und ihre Werkzeuge (wie beispielsweise LaTeX) in Erwägung zu ziehen.

30 Datenbanken

In einer **Datenbank** werden (zumeist größere) Datenmengen unabhängig von einem konkreten Anwendungsprogramm gespeichert. Die somit ermöglichte Entkopplung der Datenverarbeitung (durch das Anwendungsprogramm) von der Datenhaltung (durch die Datenbank) stellt einen der größten Schritte in der Entwicklungsgeschichte von Informationssystemen dar. Auch wenn Datenbanken heute nicht mehr für große Aufregung sorgen, so kann man mit einem modernen Datenbanksystem viele Problemstellungen umfassend abdecken.

Zwar wird der Begriff Datenbank recht universell verwendet, doch meint man mit Datenbank häufig die gespeicherten Daten selbst. Das Programm, das die Daten speichert und den Zugriff auf sie ermöglicht, wird **Datenbankmanagementsystem** genannt. Es stellt alle Funktion zur Verfügung, die benötigt werden, um die Struktur der zu speichernden Daten festzulegen, um die Daten zu speichern, um sie wiederzufinden und um sie auszuwerten. Als eigenständiges System hat ein Datenbankmanagementsystem *Schnittstellen* zum *Betriebssystem* eines Computers, zu *Anwendungsprogrammen* (es bietet sein eigenes *Application Programming Interface*) und zum Benutzer des Computers (die *Benutzungsschnittstelle*); ein Benutzer kann also auch ohne ein Anwendungsprogramm eine Datenbank einrichten und verwenden. Tatsächlich interagieren sog. *Datenbankadministratoren* mit Datenbankmanagementsystemen zumeist direkt (über die Benutzungsschnittstelle).

30.1 Datenbanksprachen

Da Datenbanken unabhängig von Anwendungsprogrammen existieren, ist es sinnvoll, sie auch unabhängig von solchen benutzen zu können. Dies geschieht häufig mittels eigener **Datenbanksprachen**, von denen man anhand ihres Zwecks drei unterscheiden kann: eine *Datendefinitionssprache* (data definition language, DDL), eine *Datenmanipulationssprache* (data manipulation language, DML) und eine *Datenabfragesprache* (data query language, DQL).

Datendefinitionssprache

Mit der **Datendefinitionssprache** wird das Schema einer Datenbank festgelegt, das in der Regel aus einer vorangegangenen *Datenmodellierung* (Kapitel 28) hervorgegangen ist. Da ein Schema auf der *Typebene* angesiedelt ist, entspricht seine Definition in Teilen der Vereinbarung von Datentypen (s. Kapitel 27) in Programmiersprachen und tatsächlich findet man häufig eine enge Korrespondenz zwischen

WIKIPEDIA

dem Schema einer Datenbank und den Typen der Anwendungsprogramme, die sie benutzen. Neben der reinen Schemadefinition können mit einer Datendefinitionssprache aber auch administrative Dinge wie die Art der Speicherung oder Strukturen zum beschleunigten Zugriff (Indizierung der Daten) festgelegt werden.

Ebenso wichtig wie die (ursprüngliche) Schemadefinition ist die **Schemaänderung** oder **Schemaevolution**, deren technische Herausforderung darin besteht, nach dem alten Schema bereits erfasste Daten an das neue Schema anzupassen. Dies ist dann nur ein kleines Problem, wenn das Schema lediglich erweitert werden muss (die neuen Datenfelder bleiben dann einfach leer oder werden mit Vorgabewerten automatisch belegt), dann aber ein großes, wenn vorhandene Daten an das neue Schema angepasst werden müssen. Man spricht dann auch von **Schemamigration**. Eine besondere Herausforderung stellt die Schemamigration im laufenden Betrieb dar (also während die Datenbank benutzt wird). Man sollte sich vor der Entscheidung für ein bestimmtes Datenbankmanagementsystem im Klaren darüber sein, ob Schemamigration im laufenden Betrieb benötigt wird und sich ggf. vergewissern, dass sie auch unterstützt wird.

Datenmanipulationssprache

Ist das Schema angelegt, können Daten erfasst und die Datenbank somit gefüllt werden. Die meisten Datenbankmanagementsysteme bieten dazu eigenständige, d. h. von Anwendungsprogrammen unabhängige Möglichkeiten, beispielsweise über (automatisch oder manuell konfigurierte) Eingabemasken. Eine andere, von Programmierern häufig bevorzugte Möglichkeit ist die Verwendung einer eigenen **Datenmanipulationssprache**, die auch in *Skripte* eingebunden werden kann und die somit den (programmierten) Import aus anderen Datenquellen (einschließlich anderer Datenbanken) erlaubt.

WIKIPEDIA

Datenabfragesprache

Sind die Daten erfasst, sollen sie auch genutzt werden. Während dies regelmäßig durch Anwendungsprogramme erfolgt, gibt es auch einen Bedarf an sog. *Ad-hoc-Abfragen*, für die man keine extra Programme schreiben möchte. Für diesen Zweck sind sog. **Datenabfragesprachen** vorgesehen, die neben der Auswahl (Suche) und *Verknüpfung von Daten* auch sog. *Aggregationsfunktionen* bieten, also beispielsweise die Bildung von Summen oder Mittelwerten.

WIKIPEDIA

Structured Query Language (SQL)

Eine der bekanntesten (und verbreitetsten) Datenbanksprachen ist die **Structured Query Language** (**SQL**), die entgegen ihrem Namen eben nicht nur eine Abfragesprache, sondern auch eine Definitions- und Manipulationssprache umfasst. Sie standardisiert den Umgang mit relationalen Datenbanken, die zwar in die Jahre gekommen, aber immer noch weit verbreitet und für viele Zwecke gut einsetzbar sind, schon weil SQL heute immer noch Teil einer jeden Informatikausbildung ist.

WIKIPEDIA

30.2 Datenbankprogramme

Wenn aus Programmen auf die Daten einer Datenbank zugegriffen werden soll, dann geschieht dies zunächst mit den Mitteln einer Programmiersprache. Das Datenbankmanagementsystem bietet für diesen Zweck eine *Programmierschnittstelle* (*Application Programming Interface*), die in der Regel aus einer Menge von Funktionen (einer *Funktionsbibliothek*) besteht, die vom Programm aus aufgerufen werden können. Die dabei übergebenen Daten haben die (Daten-)Typen der Programmiersprache; passen diese nicht zu den Datentypen der Datenbank, muss eine Konversion erfolgen, die zumeist impliziter Teil der Schnittstelle ist.

Einbettung von Datenbanksprachen in Programmiersprachen

Erfolgt der Zugriff auf die Daten einer Datenbank ausschließlich mit den Mitteln einer Programmiersprache, so werden in dieser Programmiersprache gelegentlich auch Funktionen nachgebildet, die sich mit den Mitteln der von der Datenbank vorgesehenen Datenmanipulations- oder -abfragesprache direkt angeben lassen würden. Deswegen sehen die meisten Programmierschnittstellen von Datenbankmanagementsystemen eine „Einbettung" ihrer Sprachen in Programmiersprachen vor. Dabei werden dann auf der Seite der Programmiersprache Zeichenketten zusammengebaut, die *Ausdrücke* der Datenbanksprachen darstellen und vom Datenbankmanagementsystemen interpretiert werden können. Fortschrittlichere, echte Einbettungen gehen weiter, indem sie die Programmiersprache selbst um Konstrukte zur Datenmanipulation und -abfrage erweitern. Dies ist beispielsweise durch die Erweiterung von .NET Programmiersprachen um sog. *Language Integrated Queries* (LINQ) geschehen.

Persistenzframeworks

Datenbankmanagementsysteme (und damit auch Datenbanken) implementieren in der Regel ein bestimmtes *Datenmodell* (s. Abschnitt 28.2). Die vergangenen Dekaden dominiert haben dabei die sog. **relationalen Datenbanken** (mit deren prominenten Vertretern Oracle und MySQL), die auf dem *relationalen Datenmodell* (Abschnitt 28.2.2) basieren. Auch wenn die relationalen Datenbanken unbestritten Vorteile haben, so ist das zugrundeliegende Datenmodell (das relationale) doch unflexibel und semantisch schwach (s. dazu Abschnitt 28.2.4). Insbesondere das Fehlen von Objekten, Verweisen auf diese und die fehlenden Subtypen (Vererbung) machen die Übertragung von Daten aus und in *objektorientierte Anwendungsprogramme* (wie sie beispielsweise in *Java* geschrieben werden) kompliziert. Dies hat zum Aufkommen sog. **objektorientierter Persistenzframeworks** geführt (mit *Hibernate* als prominentem Vertreter), die den Repräsentationswechsel zwischen relationaler Speicherung und objektorientierter Verarbeitung von Daten erleichtern sollen. Tatsächlich bringen diese Persistenzframeworks erhebliche eigene *akzidentelle Komplexität* mit sich und wenn man sie verwenden muss, kann man sich zurecht gründlich ärgern.

Tatsächlich haben nämlich objektorientierte Daten eher Netzwerk- als relationalen Charakter und das *Netzwerkdatenmodell*, das kurz nach seinem Aufkommen vom relationalen verdrängt wurde (Abschnitt 28.2.3), bietet eine gute Grundlage für **objektorientierte Daten-**

banken, wie sie eine Zeit lang en vogue waren, sich aber leider nicht durchsetzen konnten.[81] Heute haben die großen Internetkonzerne alle ihre eigenen Datenbanksysteme entwickelt, die ihren jeweiligen Anforderungen genügen und die weder relational noch objektorientiert im strengen Sinne sind.

30.3 Transaktionssicherheit

Eines der wichtigsten Versprechen, das Datenbanksysteme liefern, ist das der Transaktionssicherheit. Gemeint ist damit,

1. dass Daten nicht verlorengehen (die **Dauerhaftigkeit** der Speicherung),
2. dass in einer Transaktion zusammengefasste Änderungen an Daten entweder ganz oder gar nicht durchgeführt werden (die **Atomarität** von Änderungen),
3. dass der Datenbestand nach einer Transaktion immer in einem konsistenten Zustand ist, wenn er es auch vorher war, (die **Konsistenzerhaltung**) und
4. dass gleichzeitige Transaktionen unabhängig voneinander ausgeführt werden (die **Isolation**).

Dauerhaftigkeit

Die erste Eigenschaft verlangt ein automatisches Datensicherungssystem, bei dem in regelmäßigen Abständen der gesamte Datenbestand kopiert wird (Vollsicherung) und zwischen zwei Vollsicherungen die Änderungen am jeweils letzten Datenbestand so geloggt (mitgeschrieben) werden, dass sie sich bei einem Datenverlust rekonstruieren lassen. Je nach Verwendung der Datenbank kann es erforderlich sein, dass sich sowohl das Sichern (engl. backup) als auch das Zurückspielen (engl. recover) während des laufenden Betriebs (also auch während Änderungen am Datenbestand vorgenommen werden) durchführen lassen.

Atomarität

Die zweite Eigenschaft ist wichtig, wenn elementare Änderungen nur im Konzert durchgeführt werden dürfen, weil sonst die Daten die Realität nicht richtig abbilden. Ein klassisches Beispiel hierfür ist eine Überweisung, die sich aus der Belastung des abgebenden Kontos und einer Gutschrift auf dem empfangenden Konto zusammensetzt. Findet nur eine der beiden Änderungen statt, ist die Überweisung unvollständig. In der Informatik wird der Begriff der Transaktion übrigens allgemeiner gebraucht als im Finanzwesen und so kann eine Transaktion durchaus viele, u. U. auch komplexe und längerdauernde Einzelaktionen zusammenfassen.

---

[81] Insbesondere die Weiterentwicklung des Netzwerkmodells zum *Role Data Model* durch *Charles Bachman* nimmt bereits viele Entwicklungen der Objektorientierung vorweg. Zwar bekam Charles Bachman für sein Werk den Turing Award („Nobelpreis der Informatik"), doch seine Beobachtung vom „programmer as navigator" kam wohl für die breite Masse zu früh. Objektorientierte Programmierer machen jedoch nichts Anderes: Sie *navigieren* durch Daten.

Konsistenzerhaltung

Die dritte Eigenschaft unterstellt, dass Daten bestimmten Bedingungen unterliegen — sind diese nicht eingehalten, sind die Daten inkonsistent und damit fehlerhaft. Ein einfaches Beispiel für eine solche *Konsistenzbedingung* ist, dass der Kontostand eines Sparbuchs nicht negativ sein kann — ist er es doch, hat es eine unzulässige Abbuchung gegeben. Aufgabe eines Datenbankmanagementsystems ist es nun, am Ende einer Transaktion zu prüfen, ob alle Bedingungen (weiterhin) eingehalten werden; ist eine Bedingung verletzt, muss die Transaktion abgebrochen werden, was, aufgrund der zweiten Bedingung „ganz oder gar nicht", bedeutet, dass alle im Rahmen der Transaktion erfolgten Änderungen rückgängig gemacht werden müssen. Wenn man zudem sicherstellt, dass Änderungen ausschließlich im Rahmen von Transaktionen durchgeführt werden, dann ist „nach der Transaktion" „vor der Transaktion" und alle Bedingungen sind auch vor (zu Beginn) einer Transaktion eingehalten.

Isolation

Die vierte Eigenschaft stellt sicher, dass in einer Mehrbenutzerumgebung mehrere Personen gleichzeitig oder sich zeitlich überlappend Transaktionen durchführen können, ohne dass es dadurch zu Effekten kommt, die man bei einer (beliebigen) Nacheinanderausführung derselben Transaktionen nicht beobachten könnte. Wenn also beispielsweise zwei Kunden bei einem Online-Warenhaus die gleiche Ware kaufen wollen und deren Bestand bei nur noch einem Stück liegt, dann können zunächst beide dieselbe Ware in ihren Warenkorb legen (da ja nicht sicher ist, ob beide den Kauf auch abschließen wollen). Beim Kaufen kann aber nur einer Erfolg haben — für die andere ändert sich der Bestand während seiner Transaktion auf Null, so dass der Kauf nicht durchgeführt werden kann, da der Bestand sonst negativ würde (die entsprechende Transaktion also abgebrochen werden muss, eine entsprechende Konsistenzbedingung vorausgesetzt). Bei einer Nacheinanderausführung der beiden Transaktionen könnte der zweite Kunde die Ware erst gar nicht in seinen Warenkorb legen, da nach der ersten Transaktion der Bestand schon auf Null gesunken ist.

ACID

Alle vier Eigenschaften werden auch als die **ACID-Eigenschaften** von Datenbanktransaktionen bezeichnet, wobei A für Atomicity (die zweite Eigenschaft aus obiger Liste), C für Consistency (die dritte), I für Isolation (die vierte) und D für Durability (die erste) steht. Die Garantie dieser Eigenschaften verlangt einen erheblichen technischen Aufwand, in den Jahrzehnte intensiver Forschungsarbeit geflossen sind. Wie groß der Aufwand tatsächlich ist kann man auch daran ermessen, dass große Internetgeschäfte wie Amazon mit ihren eigenen Datenbanken auf die ACID-Eigenschaften zumindest teilweise verzichten — es erscheint günstiger, etwaige Schäden zu kompensieren (beispielsweise die Nicht- oder zu späte Lieferung einer bestellten Ware) als Transaktionssicherheit zu garantieren (wobei der tatsächliche Warenbestand auch aus anderen Gründen mit dem rechnerischen nicht übereinstimmen muss — „ein bisschen Schwund ist immer").

Forschungsdatenbanken

In einem Forschungskontext, in dem zwar Daten erfasst, aber nicht manipuliert, sondern nur ausgewertet werden, erscheinen die ACID-Eigenschaften grundsätzlich verzichtbar. Das Augenmerk liegt hier vielmehr darauf, dass die Daten ohne viel (Programmier-)Aufwand anderen Auswertungswerkzeugen zugänglich gemacht werden können. Ein Beispiel für dafür geeignete Datenbanksysteme sind die mit den sog. Office-Paketen ausgelieferten (z. B. Access oder Base), die neben einer erleichterten

Datenabfrage auch die Verknüpfung mit den paketeigenen Tabellenkalkulationen und so beispielsweise die einfache diagrammatische Darstellung von Auswertungen (s. Kapitel 32) erlauben. Die gleichzeitige Nutzung des Datenbestandes durch mehrere Personen ist in diesen Konstellationen jedoch stark eingeschränkt und man wird in der Regel mit Kopien der Datenbanken arbeiten. Die heute ebenfalls anzutreffende Verwendung von Tabellenkalkulationsprogrammen für die Datenhaltung ist nicht zu empfehlen, schon weil die Anzahl der Datensätze hier zumeist beschränkt ist.

31 Big Data

Durch die zunehmende Digitalisierung von allen Teilen des Lebens fallen immer mehr Daten in immer höherer Geschwindigkeit an. Neben dem primären Zweck der Datenerfassung ergibt sich durch die unbegrenzt anmutenden Kapazitäten der Datenspeicherung die Möglichkeit, diese Daten auch für andere Zwecke zu verwenden, wobei der Wert hier häufig gerade in der Masse der verfügbaren Daten liegt. Dabei ist der Zweck zum Zeitpunkt der Speicherung häufig noch gar nicht genau bestimmt — entsprechend sind die Daten auch für diesen Zweck nicht strukturiert.

Unter dem Begriff **Big Data** sammeln sich alle Bemühungen, den Schatz in Daten zu heben, der mit allein den Mitteln der „hergebrachten" Datenverarbeitung (insbesondere denen traditioneller Datenbanksysteme) im Verborgenen bleibt. Kennzeichnend ist, dass man häufig gar nicht so genau weiß, wonach man sucht, es also mit gewöhnlichen *Datenabfragen* (s. Abschnitt 30.1) nicht getan ist. Entsprechend vielfältig sind die Methoden, die für den Umgang mit Big Data herangezogen werden. Dabei kommt auch das sog. *Data Mining* zum Einsatz.

WIKIPEDIA

WIKIPEDIA

Ein besonderes Problem, das auch unter dem Begriff Big Data abgehandelt wird, ist die Auswertung von Daten, die schneller anfallen, als sie gespeichert werden können (ja, das gibt es!), so dass sie sofort verarbeitet werden müssen. Ein schon etwas älteres Beispiel hierfür ist die Suche nach außerirdischer Intelligenz durch das *SETI@home-Projekt*. Dabei werden aus dem Weltall aufgenommene Radiosignale weltweit auf Computer verteilt, die sich dem Projekt angeschlossen haben und die in ihrer Leerlaufzeit diese Signale auf mögliche Zeichen einer Kommunikation hin untersuchen. Die dabei erreichten Rechenkapazitäten sind so groß, dass sie kaum eine Organisation mit eigenen Mitteln für diesen Zweck würde aufbringen wollen.[82] Big-Data-Projekte von Unternehmen und Regierungen werden da freilich ganz anders ausgestattet; dennoch haben auch diese ein Problem, wenn die Datenverarbeitungsrate dauerhaft unter die Datenproduktionsrate fällt.

WIKIPEDIA

[82] Auf der anderen Seite ist auch dieses Projekt nicht kostenlos, da moderne CPUs kaum noch Leerlauf kennen — wenn die Rechenlast fällt, verringern sie ihre *Taktfrequenz*, um Strom zu sparen (vgl. Abschnitt 11.2.3 in Kurseinheit 1).

das Web als Quelle von Big Data

Eine aktuelle und jedermann zugängliche Quelle von Big Data ist das *Web*. Sog. *Web crawler* erlauben es, das Web automatisch abzusuchen und dabei Information zu sammeln und auszuwerten. Besonders ergiebig erscheint dabei das „Ernten" der Daten von ausgewählten Teilen des Webs wie den sog. sozialen Netzen oder Wikipedia. Eine besondere Herausforderung stellen dabei die natürlichsprachlichen Inhalte dar, deren Bedeutung nicht durch das Vorkommen einzelner Wörter, sondern durch deren grammatikalisches Zusammenwirken und die Einbettung in einen — häufig zu erheblichen Teilen impliziten — Kontext bestimmt ist. Will man diese Bedeutung berücksichtigen, reicht eine grammatikalische Analyse (für die es umfangreiche Programmpakete wie beispielsweise den *Stanford Natural Language Parser* gibt) gar nicht aus — ohne das Hintergrundwissen, das beim menschlichen Leser vorausgesetzt wird, bliebt der Sinn vieler Texte im Verborgenen. Zudem ist die syntaktische Zerlegung von Sätzen selten eindeutig und Mehrdeutigkeiten (wie sie obendrein durch die Mehrdeutigkeit einzelner Wörter entstehen) lassen sich nur durch aufwendige semantische Analysen reduzieren, die den (expliziten und impliziten) Kontext zu berücksichtigen haben. Es ist fraglich, ob dies jemals gelingen wird (und noch fraglicher, was dies bedeuten würde — müsste man dann die Deutungshoheit, über die sich Menschen in der Regel nicht einig werden, Maschinen überlassen?).

Link

Ein anderes prominentes Beispiel für Big Data ist die Auswertung von Suchanfragen zu anderen Zwecken als der Beantwortung der Anfragen. Hier fällt einem zunächst die personalisierte Werbung ein, die sich mittlerweile auf die Erstellung umfassender Persönlichkeitsprofile stützt (die im Übrigen nicht nur mithilfe von Suchanfragen, sondern auch durch die Auswertung von Facebook-Likes oder die Zuordnung von Einkäufen zu Kunden über Rabattkarten betrieben wird). Es sind aber auch schon gemeinnützige Zwecke damit verfolgt worden wie beispielsweise die Dokumentation der Ausbreitung von Grippe (das *Flu-trends-Projekt* von Google, dessen Daten trotz Einstellung zu Forschungszwecken nach wie vor zur Verfügung stehen) — die Abwägung von Datenschutz gegen andere Allgemeininteressen stellt hier ein interessantes *ethisches Problem* dar. In eine ähnliche Kategorie fällt die Auswertung von Handybewegungsdaten bei Seuchen wie der Ebola-Viruserkrankung oder Flächenkatastrophen wie beispielsweise Erdbeben oder Überschwemmungen, wo den bereitstehenden Hilfskräften oftmals nicht klar ist, wohin die Hilfe (zuerst) gehen soll. Nicht zuletzt stellen Handydaten eine Basis für das kritische Hinterfragen von Maßnahmen wie beispielsweise Ausgangssperren zur Vermeidung der Seuchenausbreitung dar.

Link

Human in the loop

Eine Möglichkeit der Suche nach etwas, das man nicht genau kennt, ist es, Menschen einzubinden. Ein Ansatz hierfür ist die sog. **Gamification**, bei der Probleme aus ihrem Gegenstandsbereich in ein Spiel übertragen werden, wodurch es Menschen als (Teil-)Ziel des Spiels zu lösen versuchen, ohne dies als mühselige Arbeit wahrzunehmen. Wenn man bedenkt, dass im Jahr 2013 allein mit dem Ego shooter „Call of Duty" mehr als vier Milliarden Stunden verdaddelt wurden und im Jahr 2016 die Menschheit im Schnitt mehr als eine Milliarde Stunden pro Monat mit Spielen auf Mobilgeräten verbrachte, bietet Gamification ein kaum zu ignorierendes Potenzial zur Hebung intelligenter menschlicher Leistung (vgl. dazu auch Abschnitt 3.4 in Kurseinheit 1).

Link

Link

Eine andere Möglichkeit, mithilfe menschlicher Fähigkeiten etwas in Daten zu finden, ist ihre Visualisierung.

32 Datenvisualisierung

Daten, numerische zumal, sind für das menschliche Gehirn nicht immer leicht zu erfassen, insbesondere wenn sie umfangreich (und einzelnen Zahlen extrem groß oder klein) sind. Sie werden daher gern grafisch dargestellt. Man spricht dann auch von einer **Visualisierung der Daten**. Dabei können Computer hilfreich sein und es ist wohl nicht übertrieben zu behaupten, die Verfügbarkeit von Computern und entsprechenden Programmen hätten zu einer Visualisierungsflut geführt.

Die Visualisierung von Daten dient immer einem Zweck. Ein Zweck ist der des Informierens: Die Information, die in den Daten steckt, soll auf anschauliche Art transportiert werden. Ein anderer ist der des Verstehens: Durch die Anschauung der Visualisierung kann man eine Theorie formen, die die visualisierten Daten erklärt. Wir beginnen mit dem zweiten.

32.1 Visualisierung zum Zweck der Theoriebildung

Wenn Daten mit dem Zweck gesammelt werden, einen Sachverhalt zu verstehen, dann sind die Daten Indizien. Gesucht wird dann eine Theorie, die die Daten (Indizien) erklärt, und diese Suche bedarf in aller Regel Menschen mit Sachverstand und der richtigen Intuition. Im Idealfall kann eine Datenvisualisierung die richtigen Rückschlüsse[83] nahelegen; genauso kann sie aber auch in die Irre führen und damit Jahre intensiver Forschungsarbeit vergeuden. Visualisierungen orientieren sich daher häufig an Modellen, die sich in einem bestimmten Fachgebiet bewährt haben, wie beispielsweise dreidimensionale Kugelmodelle für Moleküle. Für andere Theorien wie die Krümmung von Raum und Zeit versagen Visualisierungen in den Menschen vertrauten Koordinatensystemen aber. Außerdem können Visualisierungen viel komplexer erscheinen als der ihnen zugrundeliegende Zusammenhang, wie man sich am Beispiel von Fraktalen vor Augen führen kann.

Gefahr der Fehlleitung

Wichtig ist, dass die Visualisierung zum Zweck der Erkenntnis immer nur eine Hilfestellung sein kann — die Erkenntnis liegt nie in der Visualisierung selbst. Es ist daher gefährlich, mit Visualisierungen zu arbeiten, die eine bestimmte Erkenntnis nahelegen, ohne die nahegelegte Erkenntnis zu überprüfen — tut man dies trotzdem, dann manipuliert man den Betrachter, insbesondere, wenn andere Visualisierungen derselben Daten alternative Erkenntnisse nahelegen.

---

[83] Ein Rückschluss sei hier ein Schluss von Symptomen auf Ursachen. Da in aller Regel die Symptome Folge der Ursache sind und nicht umgekehrt, hat ein solcher Rückschluss keine Beweiskraft (s. dazu auch Abschnitt 3.2.2 zum logischen Schließen); es sind also auch falsche Rückschlüsse möglich.

32.2 Visualisierung zum Zweck der Information

Umfangreiche Zahlenkolonnen entziehen sich in der Regel der schnellen menschlichen Erfassung. So stechen weder Extremwerte (Maximum und Minimum) hervor noch lassen sich Durchschnittswerte oder Varianz leicht ablesen. Die Visualisierung in Form von Diagrammen (Balken- oder Linien-) oder sog. Charts erlaubt dagegen einen schnellen Überblick und vermittelt häufig ein intuitives Verständnis vom „Inhalt" der Zahlen.

gezielte Fehlinformation

Obwohl zunächst eher unverdächtig, so bietet doch auch die Visualisierung von Daten zum Zweck der Information reichlich Möglichkeiten zur Manipulation. Eine beliebte Spielart hiervon ist das Weglassen der Nulllinie in Balkendiagrammen: Wenn beispielsweise die Entwicklung eines Aktienkurses dargestellt wird, dann mag es zwar technische Gründe geben, warum die Länge der Balken nicht den absoluten Wert, sondern die Differenz zu einem Sockelbetrag wiedergibt (z. B. wenn die Schwankungen gemessen am absoluten Wert so klein sind, dass man sie in der Grafik kaum noch erkennen kann), doch hängt dann die visualisierte Amplitude der Schwankungen an diesem Sockelbetrag, mit dem man die Amplitude groß oder klein erscheinen lassen kann. Wenn die Amplitude kommuniziert werden soll, dann verbietet sich eine solche Visualisierung — trotzdem ist sie immer wieder anzutreffen. Ähnliches findet man auch auf der Horizontalen: Die aktuelle Entwicklung eines Aktienkurses kann im historischen Kontext ganz anders aussehen, was sich verschweigen lässt, indem man das Zeitfenster für den eigenen Zweck passend wählt.

32.3 Visualisierung multidimensionaler Daten

Während die grafische Darstellung rein numerischer Daten in Form von Diagrammen mehr oder weniger standardisiert ist, so hat man es doch, insbesondere in den nicht quantitativen Wissenschaften, häufig mit vielfältigen Informationen zu tun, die gleichwohl von einer gemeinsamen Visualisierung (zu welchem Zweck auch immer) profitieren können. Ein vielzitiertes Beispiel hierfür ist die nachstehende „Carte figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812–1813" von Charles Joseph Minard, die bisweilen als *das* Pionierwerk der modernen *Infografik* angesehen wird. Sie illustriert den Russlandfeldzug Napoleons und zeigt dabei nicht nur die Truppenbewegung zusammen mit der (abnehmenden) Truppenstärke (Breite des Stroms), sondern auch den Temperaturverlauf auf dem Rückzug, der so mit den Verlusten korreliert werden kann (wobei insbesondere die Überquerung von Flüssen die Truppenstärke regelmäßig reduziert zu haben scheint).

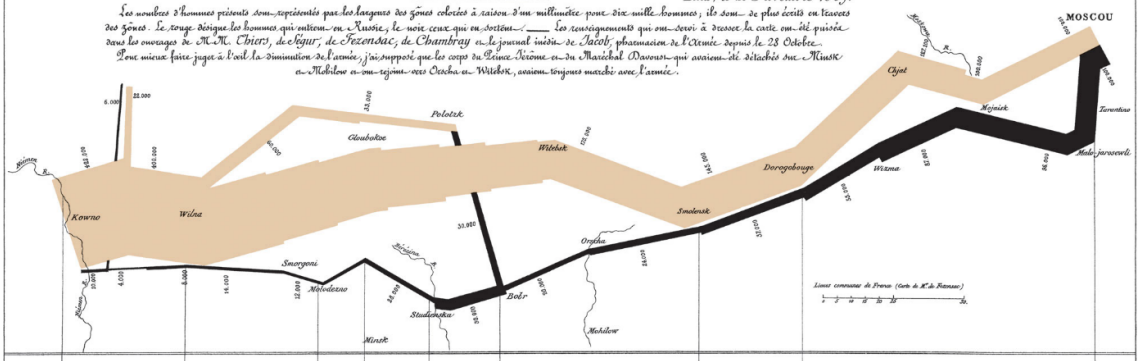
Carte Figurative des pertes successives en hommes de l'Armée Française dans la campagne de Russie 1812–1813.

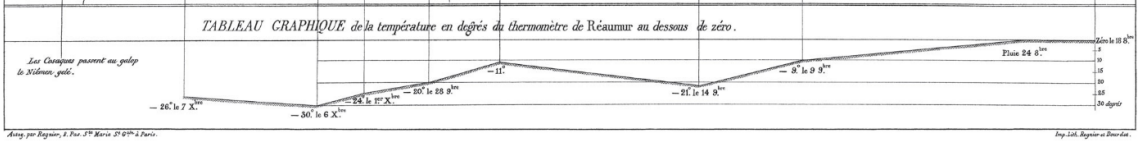
Dressée par M. Minard, Inspecteur Général des Ponts et Chaussées en retraite.

Paris, le 20 Novembre 1869.

Les nombres d'hommes présents sont représentés par des largeurs des zônes colorées à raison d'un millimètre pour dix mille hommes; ils sont de plus écrits en travers des zônes. Le rouge désigne les hommes qui entrent en Russie, le noir ceux qui en sortent. —— Les renseignements qui ont servi à dresser la carte ont été puisés dans les ouvrages de MM. Thiers, de Ségur, de Fezensac, de Chambray et le journal inédit de Jacob, pharmacien de l'Armée depuis le 28 Octobre.

Pour mieux faire juger à l'œil la diminution de l'armée, j'ai supposé que les corps du Prince Jérôme et du Maréchal Davoust qui avaient été détachés sur Minsk et Mohilow en sont rejoints avec Orcha et Witebsk, avaient toujours marché avec l'armée.





Aquar. par Regnier, 8 Pas. St Marie St Germ. à Paris.

Imp. Lith. Regnier et Deverdun.

Nachwort

Kürzlich hörte ich im (immer noch analogen!) Radio von der Forderung nach „flächendeckender Digitalisierung". Was könnte das sein? Vermutlich ist die flächendeckende Versorgung mit schnellen Internet-Verbindungen gemeint, aber diese Verkürzung des Begriffs der Digitalisierung in der Öffentlichkeit scheint mir doch bemerkenswert.

Etwa zur gleichen Zeit hörte ich, dass Schüler eines Kölner Leistungskurses in Informatik vom Land Nordrhein-Westfalen dafür ausgezeichnet worden sind, dass sie Manager namhafter deutscher Unternehmen „fit für die digitale Zukunft" machten, indem sie — als „Digital natives" — ihnen zeigten, wie man twittert oder per Snapchat Bilder verschickt. Was soll hier bloß wem vermittelt werden? Erstens sind alle diese Unternehmen längst in einem Maße „digital unterwegs", das sich kaum einer vorstellen kann, und zweitens suggeriert dies in der Öffentlichkeit ein Bild von digitalen Kompetenzen, das schlichter kaum sein könnte. Ich weiß nicht, was die Schüler den Managern tatsächlich vermittelt haben, aber ich hoffe, dass es nicht nur der Umgang mit den „Social Media" war und, insbesondere, dass sie nicht dafür den Preis bekommen haben — die Botschaft wäre verheerend.

Schließlich lese ich heute in der Zeitung, dass die Schüler in Nordrhein-Westfalen ihre eigenen Computer in die Schule mitbringen sollen, um diese im Unterricht zu nutzen. Damit soll der Rückstand in der Digitalisierung der Schulen aufgeholt werden. Welchen Schwächen im nordrhein-westfälischen Schulsystem mit Digitalisierung begegnet werden soll, steht da nicht — Computer sind im Unterricht mittlerweile gesetzt und ihr Nutzen wird nicht mehr hinterfragt. Wie allerdings ein „bring your own device" mit dem vielerorts aus guten Gründen geforderten (und teilweise bereits praktizierten) Handyverbot an Schulen in Einklang gebracht werden kann, bleibt abzuwarten — Universalmaschinen wie Computer in ihrer Funktion wirksam auf das vermeintlich Nützliche zu beschränken ist viel aufwendiger, als so manch ein Advokat der Digitalisierung vielleicht glauben möchte.

Nachrichten wie diese zeugen von einer tiefen Verunsicherung: Sind wir zu spät? Haben wir etwas verpasst? Findet die Zukunft ohne uns statt, wenn wir uns nicht bald auf den Hosenboden setzten und unsere Hausaufgaben machen, sprich, alles, was noch analog ist, digitalisieren? Nein. Die Digitalisierung ändert keine Naturgesetze, keine ökonomischen und auch keine sozialen. Wie andere technische Neuerungen vor ihr beschleunigt sie den Fortschritt, aber sie ändert keine menschlichen Konstanten. Die Macht, die die Digitalisierung einigen Menschen derzeit verleiht, ist von begrenzter Dauer, und Disruption als Geschäftsmodell wird sich, so wie die vielbeschworene New Economy zu Anfang dieses Jahrtausends, als nicht nachhaltig erweisen. Es wird eine Weile dauern, bis wir unsere Normen an die Änderungen, die die Digitalisierung bringt, angepasst haben (die erst kürzlich inkraftgetretene Datenschutzgrundverordnung der EU ist ein Beispiel dafür), aber es werden immer unsere, menschliche Normen bleiben. Voraussetzung dafür ist allerdings, dass wir uns nicht

weiter verunsichern lassen, und dafür wiederum, dass wir ein bestimmtes Grundwissen davon, was die Mittel der Digitalisierung sind (nicht: was sie bedeuten), zu unserer Bildung rechnen. Und zwar möglichst „flächendeckend“.

Verzeichnis der Weblinks im Rand

iii Wikipedia-Artikel ..... *de.wikipedia.org/wiki/Wikipedia:Artikel*
Android-App: Papier/Digital-Brücke *www.feu.de/ps/bins/pdb.apk*
1 Glyphen..... *de.wikipedia.org/wiki/Glyphe*
5 Ziffer..... *de.wikipedia.org/wiki/Zahlzeichen*
8 Fano-Bedingung ..... *de.wikipedia.org/wiki/Fano-Bedingung*
10 Bit..... *de.wikipedia.org/wiki/Bit*
Informationsgehalt ..... *de.wikipedia.org/wiki/Informationsgehalt*
Kurs 01608, Kapitel 2 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01608*
12 Cantors zweites Diagonalargument *de.wikipedia.org/wiki/Cantors_zweites_Diagonalargument*
14 Time-Memory Tradeoff ..... *de.wikipedia.org/wiki/Time-Memory_Tradeoff*
15 Zweierkomplement..... *de.wikipedia.org/wiki/Zweierkomplement*
17 euklidscher Algorithmus ..... *de.wikipedia.org/wiki/Euklidischer_Algorithmus*
19 Not a Number..... *de.wikipedia.org/wiki/NaN*
Zahlwort..... *de.wikipedia.org/wiki/Zahlwort*
21 Logik ..... *de.wikipedia.org/wiki/Logik*
22 Antinomie..... *de.wikipedia.org/wiki/Antinomie*
23 Kurs 01141, Kurseinheit 7 (mathematische Logik) *www.fernuni-hagen.de/fu-search/index.jsp?query=01141*
Wahrheitstabelle..... *de.wikipedia.org/wiki/Wahrheitstabelle*
26 Aussagenlogik ..... *de.wikipedia.org/wiki/Aussagenlogik*
28 Erfüllbarkeitsproblem der Aussagenlogik *de.wikipedia.org/wiki/Erfüllbarkeitsproblem_der_Aussagenlogik*
33 boolesche Algebra ..... *de.wikipedia.org/wiki/Boolesche_Algebra*
35 Fuzzylogik..... *de.wikipedia.org/wiki/Fuzzylogik*
37 Gödelscher Unvollständigkeitssatz *de.wikipedia.org/wiki/Gödelscher_Unvollständigkeitssatz*
Paradoxon des Epimenides..... *de.wikipedia.org/wiki/Paradoxon_des_Epimenides*
Beschreibungslogik ..... *de.wikipedia.org/wiki/Beschreibungslogik*
IBM Watson..... *de.wikipedia.org/wiki/Watson_(Künstliche_Intelligenz)*
38 Mastering the Game of Go without Human Knowledge *www.nature.com/articles/nature24270*
39 Turing-Test ..... *de.wikipedia.org/wiki/Turing-Test*
40 ASCII ..... *de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange*
Paritätsbit ..... *de.wikipedia.org/wiki/Paritätsbit*
43 Morsecode ..... *de.wikipedia.org/wiki/Morsezeichen*
Huffman-Codierung ..... *de.wikipedia.org/wiki/Huffman-Kodierung*
47 Abtasttheorem ..... *de.wikipedia.org/wiki/Nyquist-Shannon-Abtasttheorem*
48 MPEG ..... *de.wikipedia.org/wiki/Moving_Picture_Experts_Group*
MP3..... *de.wikipedia.org/wiki/MP3*
JPEG ..... *de.wikipedia.org/wiki/JPEG*

FLAC ..... *de.wikipedia.org/wiki/Free_Lossless_Audio_Codec*

52 Turingmaschine ..... *de.wikipedia.org/wiki/Turingmaschine*

Kurs 01609, Kapitel 1 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01609*

57 Blockschaltbild ..... *de.wikipedia.org/wiki/Blockschaltbild*

59 NAND-Gatter ..... *de.wikipedia.org/wiki/NAND-Gatter*

NOR-Gatter ..... *de.wikipedia.org/wiki/NOR-Gatter*

63 Endlicher Automat ..... *de.wikipedia.org/wiki/Endlicher_Automat*

65 Kurs 01608, Kapitel 3 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01608*

67 Bus ..... *de.wikipedia.org/wiki/Bus_(Datenverarbeitung)*

68 USB ..... *de.wikipedia.org/wiki/Universal_Serial_Bus*

69 Kurs 01608, Kapitel 5 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01608*

Arithmetisch-logische Einheit ..... *de.wikipedia.org/wiki/Arithmetisch-logische_Einheit*

70 Kurs 01609, Kapitel 2 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01609*

71 Memory-mapped I/O ..... *de.wikipedia.org/wiki/Memory_Mapped_I/O*

Controller ..... *de.wikipedia.org/wiki/Controller_(Hardware)*

75 Berechenbarkeitstheorie ..... *de.wikipedia.org/wiki/Berechenbarkeitstheorie*

76 Kurs 01801 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01801*

77 Systemaufruf ..... *de.wikipedia.org/wiki/Systemaufruf*

BIOS ..... *de.wikipedia.org/wiki/BIOS*

81 Dateizuordnungstabelle ..... *de.wikipedia.org/wiki/File_Allocation_Table*

83 Prozess ..... *de.wikipedia.org/wiki/Prozess_(Informatik)*

Multitasking ..... *de.wikipedia.org/wiki/Multitasking*

86 Internet ..... *de.wikipedia.org/wiki/Internet*

Kurs 01801 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01801*

88 Gemeingebrauch ..... *de.wikipedia.org/wiki/Gemeingebrauch*

90 Sandboxing ..... *docs.oracle.com/javase/tutorial/deployment/applet/security.html*

92 Cloud computing ..... *de.wikipedia.org/wiki/Cloud_Computing*

93 Kurs 01866 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01866*

94 Code injection ..... *en.wikipedia.org/wiki/Code_injection*

95 MISRA-C ..... *de.wikipedia.org/wiki/MISRA-C*

97 No-free-lunch-Theorem ..... *de.wikipedia.org/wiki/No-free-Lunch-Theoreme*

106 Algorithmus stabile Heirat ..... *books.google.de/books?id=kKMDBgAAQBAJ&pg=PA174*

107 Assembler ..... *de.wikipedia.org/wiki/Assembler_(Informatik)*

EW Dijkstra "Letters to the editor: go to statement considered harmful" CACM 11:3 (1968) 147-148 ..... *doi.org/10.1145/362929.362947*

108 N Wirth: Algorithmen und Datenstrukturen: Pascal-Version (B. G. Teubner, 1999) *dblp.org/rec/books/daglib/0095640*

Quelle: Ulrich Helmich ..... *www.u-helmich.de/*

118 Generationen von Programmiersprachen *en.wikipedia.org/wiki/Programming_language_generations*

Programmierparadigma ..... *de.wikipedia.org/wiki/Programmierparadigma*

Kurs 01613 ..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01613*

119 strukturierte Programmierung ..... *de.wikipedia.org/wiki/Strukturierte_Programmierung*

Spaghetticode ..... *de.wikipedia.org/wiki/Spaghetticode*
120 Bootstrap-Programm ..... *www.bootstrapworld.org/*
121 Kurs 01816..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01816*
123 Prolog..... *de.wikipedia.org/wiki/Prolog_(Programmiersprache)*
Kurs 01816..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01816*
125 Datalog..... *en.wikipedia.org/wiki/Datalog*
126 Kurs 01618..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01618*
Vererbung ..... *de.wikipedia.org/wiki/Vererbung_(Programmierung)*
127 dynamisches Binden ..... *de.wikipedia.org/wiki/Dynamische_Bindung*
Genus et differentia..... *de.wikipedia.org/wiki/Genus_proximum_et_differentia_specifica*
131 Stack Overflow ..... *stackoverflow.com/*
136 Programmfehler..... *de.wikipedia.org/wiki/Programmfehler*
137 Regressionstesten ..... *de.wikipedia.org/wiki/Regressionstest*
testgetriebene Entwicklung..... *de.wikipedia.org/wiki/Testgetriebene_Entwicklung*
139 Scratch-Download ..... *scratch.mit.edu/download/*
142 Variablen in Scratch..... *scratch-dach.info/wiki/Variablen*
145 Refaktorisierung ..... *de.wikipedia.org/wiki/Refactoring*
147 Debuggen von Scratch-Skripten.... *scratch-dach.info/wiki/Debuggen_von_Skripten*
149 Common User Access..... *de.wikipedia.org/wiki/Common_User_Access*
151 BASIC ..... *de.wikipedia.org/wiki/BASIC*
152 Makro..... *de.wikipedia.org/wiki/Makro*
155 Stand der Wissenschaft..... *de.wikipedia.org/wiki/Stand_der_Wissenschaft*
159 Russellsche Antinomie ..... *de.wikipedia.org/wiki/Russellsche_Antinomie*
160 Verbund (Record)..... *de.wikipedia.org/wiki/Verbund_(Datentyp)*
161 Zeiger ..... *de.wikipedia.org/wiki/Zeiger_(Informatik)*
162 Feld (Array) ..... *de.wikipedia.org/wiki/Feld_(Datentyp)*
164 abstrakter Datentyp..... *de.wikipedia.org/wiki/Abstrakter_Datentyp*
166 Entscheidungsbaum..... *de.wikipedia.org/wiki/Entscheidungsbaum*
167 Lösungsalgorithmen für Irrgärten.. *de.wikipedia.org/wiki/Lösungsalgorithmen_für_Irrgärten*
168 Literal ..... *de.wikipedia.org/wiki/Literal*
169 Universalienproblem ..... *de.wikipedia.org/wiki/Universalienproblem*
174 Ontologie ..... *de.wikipedia.org/wiki/Ontologie_(Informatik)*
CIDOC CRM ..... *www.cidoc-crm.org/*
Upper ontology ..... *en.wikipedia.org/wiki/Upper_ontology*
175 Universalsprache..... *de.wikipedia.org/wiki/Universalsprache*
Systematik für Bibliotheken..... *www.sfb-online.de/*
International Classification of Diseases *www.who.int/health-topics/international-classification-of-diseases/*
176 Attributgrammatik..... *de.wikipedia.org/wiki/Attributgrammatik*
177 Kurs 01675..... *www.fernuni-hagen.de/fu-search/index.jsp?query=01675*
178 CSV ..... *de.wikipedia.org/wiki/CSV_(Dateiformat)*
179 JSON..... *de.wikipedia.org/wiki/JavaScript_Object_Notation*
Graph..... *de.wikipedia.org/wiki/Graph_(Graphentheorie)*

Resource Description Framework .. *de.wikipedia.org/wiki/Resource_Description_Framework*
Auszeichnungssprache..... *de.wikipedia.org/wiki/Auszeichnungssprache*
180 XML..... *de.wikipedia.org/wiki/Extensible_Markup_Language*
WYSIWYG ..... *de.wikipedia.org/wiki/WYSIWYG*
181 Datendefintionssprache ..... *de.wikipedia.org/wiki/Data_Definition_Language*
182 Datenmanipulationssprache..... *de.wikipedia.org/wiki/Data_Manipulation_Language*
Datenabfragesprache..... *de.wikipedia.org/wiki/Abfragesprache*
SQL..... *de.wikipedia.org/wiki/SQL*
186 Big Data..... *de.wikipedia.org/wiki/Big_Data*
Data Mining ..... *de.wikipedia.org/wiki/Data-Mining*
SETI@home..... *de.wikipedia.org/wiki/SETI@home*
187 Stanford Natural Language Parser. *nlp.stanford.edu/software/lex-parser.html*
Persönlichkeitsprofile *doi.org/10.1073/pnas.1418680112*
Flu-trends-Projekt ..... *www.google.org/flutrends/about/*
The Washington Post: Humankind has now spent more time playing Call of Duty than it has existed on Earth..... *wapo.st/19guYlc*
1 Milliarde Stunden pro Monat verdaddelt *www.vertoanalytics.com/consumers-spend-1-billion-hours-month-playing-mobile-games/*
189 Artikel: Graphic Output *doi.org/10.1145/358080.364565*

Index

5

5th Generation Project *125*

6

6502-Prozessor *50*

A

A/D-Wandler **74**
Abbruchkriterium
einer Rekursion *122*
einer Schleife *111*, *113*
Abfrage ..... *125*, s. a. Anfrage u. Datenabfrage
Ableitungsregel **30**
abstrakter Datentyp **164**, *168*
Abstraktion
Kern- *134*
Versagen von *144*
Abstraktionsniveau
höheres *52*, *76*, *118*
niedrigeres *105*
Absturz ..... s. Programm:-absturz
Abtasttheorem **47**
Abtastung **47**
ACID-Eigenschaften **185**
Address Space Layout Randomization *163*
Ad-hoc-Abfrage *182*
Administrator/-in **82**
Adressbus **67**
Breite des *69*
Adresse ... **51**, *68*, *69*, *70*, *79*, s. a. Speicheradresse
auf einem Festspeicher *81*
Adressregister **70**
Aggregation **173**
Aggregationsfunktion *182*
agile Softwareentwicklung *102*
Akkumulator *50*
Algol *130*
Algorithmus *21*, *101*, *106*, *115*, *120*, *129*, *157*, *167*, *168*
euklidscher *17*
Unterschied zu Programm *106*
allgemeingültig *29*, *33*
allgemeingültige Aussage **28**, *37*
Allgemeingültigkeit **29**
Altlast
weltweite *90*
Altsysteme *103*
American Standard Code for Information Interchange **40**
analog *64*, *72*, *73*
Analog/Digital-Wandler **74**
Anforderung an eine Software *102*, *149*
fachliche **103**
funktionale **103**
nichtfachliche **103**
nichtfunktionale **103**
Anforderungserhebung *102*, *175*
Anfrage ..... *124*, *176*, s. a. Datenabfrage
Anmeldung **82**
Antinomie **22**
Anweisung ..... *111*, **118**, *142*, s. a. Befehl
Anwendung *148*
Anwendungsframework *134*
Anwendungsprogramm *76*, *82*, *85*, *89*, *90*, **148**, *181*
Büro- *152*
objektorientiertes *183*
Anwendungsschicht *86*
Anwendungswissen *137*, *154*
API ..... s. Application Programming Interface
App ..... *iii*, *148*, **150**
App-Entwicklungsumgebung *151*
plattformübergreifende *151*
application *148*
application program *148*
Application Programming Interface **77**, *85*, *181*, *183*
Application server **149**
Äquivalenz **24**
logische **25**
materiale *25*
semantische **25**, *27*, *29*, *32*
Ariadne *167*
Aristoteles *127*
arithmetisch-logische Einheit **69**
Array *108*, *144*, **162**, *164*, *167*
assoziatives *168*
dynamisches *144*, *163*
statisches *144*, *163*
ASCII **40**
ASLR *163*
Assemblersprache *107*
assignment *111*
Assoziation *168*
assoziatives Array *168*
Assoziativspeicher **168**
Atomarität **184**

Attribut *169*, *171*, *176*, *177*
Attribut/Wert-Paar **178**
attributierte Grammatik .....s. Grammatik
Aufzählung **163**
Ausdruck *23*, *111*, *183*
arithmetischer *24*
boolescher *142*
logischer *123*
Schachtelung *141*
Ausdrucksstärke *37*, *132*
ausführbarer Inhalt *89*, *150*
Ausgabe
einer Funktion..... *110*
Ausgabeparameter *114*, *116*
Ausnahmebehandlung..... **19**
Ausprägung *169*, *176*
Aussage *22*, **26**
allgemeingültige ....s. allgemeingültige Aussage
elementare **26**
zusammengesetzte **26**
Aussagenlogik **26**, *33*, *36*, *58*
Auszeichnung..... **179**
Auszeichnungssprache..... **180**
Authentifizierung..... *87*, *94*
Authentizität *49*, *93*
Automat.... s. deterministischer endlicher Automat
Average case *101*
Axiom **29**

B

Bachman, Charles..... *184*
Backtracking..... *125*
BASIC *107*, *119*, *130*, *132*, **151**, *153*
-Interpreter *73*
Basic Input/Output System **77**
Baum **166**, *179*
Baumform *173*
BCD-Format **18**, *44*
Bedeutung *23*
Befehl..... *94*, *111*, **118**, *147*, s. a. Anweisung
Maschinen-.....s. Maschinenbefehl
Befehlsregister **70**
Befehlszähler *78*
Begriffssystem **175**
Behälter..... **20**, *44*, *50*
Benutzungsschnittstelle *149*, *181*
Berechenbarkeit *75*
Beschreibungslogik *37*
Best case..... *101*
Betriebssystem *18*, *40*, *130*, *148*, *154*, *181*
Betriebssystemkern *76*
Bezeichner **108**, *153*
Beziehung..... *170*
Bibliothek *132*, *165*, *168*
von Funktionen *112*
Big Data..... **186**
binär codierte Dezimalzahl .... **18**, s. a. BCD-Format
Binärformat..... *82*
BIOS *73*, **77**
bistabile Kippstufe..... *65*
Bit **10**
Carry-..... *52*
Einheit bit..... *10*
Paritäts-..... *40*
Bit Block Image Transfer *45*
Bitfolge **10**, *39*
Bitmap **45**
Blatt..... *179*
Blatt eines Baums..... **166**
Block **111**, *141*
Blockform *107*
Blockschaltbild *57*
Blockstruktur..... *141*, *145*
Boole, George..... *33*
boolesche Algebra **33**, *58*, *62*
boolesche Variable *119*
boolescher Ausdruck..... *33*
Boot-Sequenz **78**
Browser .....s. Webbrowser
Bubblesort *108*
Bus **67**
Busbreite **69**
Byte **10**, *68*, *81*
Bytecode..... **117**

C

C *111*, *130*, *161*, *163*
C# *111*, *126*, *130*, *131*, *133*, *152*, *164*
C++..... *111*, *126*, *130*, *131*, *161*, *163*
Calculus ratiocinator *175*
Carry ..... *51*, s. a. Übertrag
-Bit..... *52*
Central Processing Unit **69**
character *39*
Character terminal *45*
Characteristica universalis..... *175*

Chiffrierung ... *40*, **48**
Church-Turing-These ... *50*
CIDOC CRM ... *174*
Client ... *87*
Client/Server-System ... *87*
Closed world assumption ... **124**
Cloud ... *87*
Cloud computing ... **92**
CMOS ... *64*
Cobol ... *130*
Code ... **40**
Code injection ... *94*
Codierung ... **40**
Comité international pour la documentation ... *174*
Comma-separated-value-Format ... *178*
Common User Access ... *149*
Community ... *131*, *149*
  aktive ... *131*
Compiler ... **117**, *128*, *176*
Computer ... *i*, **50**
Computerlinguistik ... *125*
Computersicherheit ... **93**
Conceptual Reference Model ... *174*
Controller ... **71**, *72*
CPU ... **69**
CSV ... *178*

D

Data Mining ... *186*
Datalog ... **125**
Datei ... **81**
Dateiverzeichnis ... **81**
Dateizuordnungstabelle ... **81**
Datenabfrage ... *177*, *186*
Datenabfragesprache ... *125*, *166*, *181*, **182**
Datenbank ... *87*, *125*, *157*, *160*, *171*, **181**
  relationale ... *127*, **183**
  temporale ... *177*
Datenbankadministrator/-in ... *181*
Datenbankmanagementsystem ... **181**
Datenbanksprache ... **181**
Datenbanksystem ... *35*
Datenbeschreibungssprache ... *160*
Datenbus ... **67**, *69*
  Breite des ... *69*
Datendefinitionssprache ... *181*
Datenmanipulationssprache ... *181*, **182**
Datenmodell ... **171**, *183*
  hierarchisches ... **172**
  Netzwerk- ... **172**, *183*
  objektorientiertes ... *176*
  relationales ... *183*
  semantisches ... *176*
  temporales ... **177**
Datenmodellierung ... *181*
Datenstruktur ... **158**
Datenträger ... **72**
Datentransportschicht ... *86*
Datentyp ... *126*, **158**, *159*
  rekursiver ... **161**
Datenvisualisierung ... **188**
Dauerhaftigkeit ... **184**
Debugger ... **136**
  Source-level ... **136**
Debugging ... **136**
defensive Programmierung ... **114**
deklarative Programmierung **118**, s. a. Programmierung
Dekomposition
  funktionale ... **115**
Dekrement ... **19**, *51*
Dereferenzierung ... **161**
deterministischer endlicher Automat ... **63**, *65*
Dezimalbruch ... **17**
Dezimalsystem ... **5**
Digital humanities ... *125*
Digital/Analog-Wandlung ... **74**
digitale Signalverarbeitung ... **47**
digitale Transformation ... *49*
Direct Memory Access ... s. DMA
Directory ... **81**
Disjunktion ... **23**, *36*, *58*
Diskretisierung ... **46**, *74*
DMA ... **71**, *72*, *74*
Dokumentation ... *104*
domänenspezifische Sprache ... **129**
DRAM ... **67**
dreiwertige Logik ... **35**
Driver ... *78*
Dualsystem ... **9**, *33*, *43*, *143*, *175*
Dualzahl ... *56*
dynamische Semantik ... s.
dynamisches Binden ... **127**, *128*

E

Editor ... **107**, *180*
  WYSIWYG- ... **180**

EEPROM ... *73*
Eingabeparameter ... *110*, *114*, *116*, *166*
eingebettete Software ... *77*, *148*
eingebettetes System ... *83*
Einloggen ... **82**
Einselement ... **33**
Einsen und Nullen
  Folge von ... *76*
Email ... *86*, *88*
endlicher Automat*75*, s. a. deterministischer endlicher Automat
Enigma ... *48*
Entität ... *4*, *169*
Entitätstyp ... *170*, *171*, *173*
Entity-Relationship-Diagramm ... **171**
Entity-Relationship-Modell ... **171**
Entscheidbarkeit ... *37*
Entscheidungsbaum ... *166*
EPROM ... *73*
ereignisgesteuert ... *141*
Erfüllbarkeit ... **28**
Erfüllbarkeitsproblem der Aussagenlogik ... **28**
Escape-Sequenz ... **40**
ethisches Problem ... *187*
euklidscher Algorithmus ... *17*
Expansion ... **48**
Extensible Markup Language ... **180**
Extension ... **170**, *173*

F

Fakt ... *124*
Fallunterscheidung ... *122*
Falsch ... *33*
Falsifikation ... **105**
  von Programmen ... *112*
Familienähnlichkeit ... *127*
Fano-Bedingung ... **8**, *40*, *43*
FAT ... **81**
Feld ... *161*, **162**
Fenster ... **84**
Festkommazahl ... **17**
Festspeicher ... *66*, *67*, **72**
File ... **81**
File Allocation Table ... **81**
File handle ... **81**, *83*
File Transfer Protocol ... *86*
Firewall ... *87*, *151*
FLAC ... *48*
Flash-Speicher ... **72**
Fließkommazahl ... **18**, *110*
Flipflop ... **65**
Floppy disk ... **72**
flüchtiger Speicher ... s. Speicher
Flu-trends-Projekt ... *187*
Fokus ... *84*
Folge von Zuständen ... *61*
Font ... *2*
  TrueType- ... *46*
  Typ-1- ... *46*
Fortran ... *130*, *132*
Fragmentierung
  des Festspeichers ... *81*
  des Hauptspeichers ... **79**
Framework ... *130*, *153*
Fremdschlüssel ... **172**
Frustration ... *148*
  erhebliche ... *136*, *139*, *144*
FTP ... *86*, *87*
Funktion ... **121**
  höherer Ordnung ... **123**
  uninterpretiert ... *125*
  uninterpretierte ... *36*
funktionale Dekomposition ... *126*, *145*
funktionale Programmierung ... *120*, **121**, *166*, s. a. Programmierung
Funktionsanwendung ... *36*, *121*
  rekursive ... *122*
  Unterschied Funktionsaufruf ... *122*
Funktionsargument ... **121**
Funktionsaufruf ... *119*
  Unterschied Funktionsanwendung ... *122*
Funktionsbibliothek ... *183*
Funktionswert ... **121**
Fuzzy controller ... *35*
Fuzzylogik ... **35**

G

Gamification ... *38*, **187**
Garbage collection ... **126**
Gates, William ... *151*
Gatter ... **57**, *69*, *74*
Gedächtnis ... *63*
Gemeingebrauch ... *88*
Generation einer Programmiersprache ... **118**
Genus et differentia ... *127*

Gerätetreiber ... *76*, **78**
Gleitkommazahl ... **18**, *143*
Glossar ... *175*
Glyphe ... **1**, *39*, *46*, *84*
Glyphen ... *45*
GPU ... **71**
grafische Benutzungsschnittstelle ... **84**
grafischen Bedienoberfläche ... *89*
Grammatik ... *125*, *128*, *176*
  Attribut- ... *176*
Graph ... *179*
Graphical Processing Units ... **71**
Graphical User Interface ... s. GUI
GUI ... **84**, *89*

H

Halbaddierer ... **33**, *57*
Halbleiter ... *59*
Hard disk ... **72**
Hardware *11*, *19*, *52*, **56**, *106*, *117*, *118*, *119*, *130*, *148*, *168*
Hauptspeicher ... **66**, *67*, *70*, *72*, *74*, *79*, *163*
herunterfahren ... *78*, **85**
Hexadezimalsystem ... **16**
Hexadezimalzahl ... **16**, *50*
Hibernate ... *183*
Hoare, Tony ... *162*
HTML ... **89**, *180*
HTTP ... *86*, *87*, **89**, *150*
Huffman-Codierung ... *43*
Hyperlink ... **89**
Hypertext ... **89**
Hypertext Markup Language ... s. HTML
Hypertext Transfer Protocol ... HTTP

I

Identifier ... *108*, **179**, s. a. Bezeichner
Identität eines Objektes ... *98*
IMAP ... *86*
imperative Programmierung **118**, s. a. Programmierung
Implementierung ... *111*, *128*, *164*, *168*
Implikation ... **24**, *124*
  logische ... s. Schlussfolgerung, logische
  materiale ... *25*, *29*
indirekte Adressierung ... *70*, *79*
Indirektion ... **70**, *77*, *161*
  doppelte ... *161*
Induktion ... *100*
Infografik ... *189*
Information Resource Dictionary System ... *159*
Informationsgehalt ... *10*
Inkrement ... **19**
Instanz ... *126*, **159**, *174*, *176*, *179*
Instanzebene ... **159**, *170*
Instanziierung ... *174*, *176*
  einer Klasse ... **126**
  eines Schemas ... *169*
  eines Typs ... **159**
integrierte Entwicklungsumgebung ... **138**
Integrität ... *93*
Interface ... *164*, s. a. Schnittstelle
International Classification of Diseases ... *175*
Internet ... **86**, *130*
Internet Message Access Protocol ... *86*
Internet Protocol ... *86*
Internet Relay Chat ... *88*
Interpretation ... *107*
  im Gegensatz zur Übersetzung ... **117**
Interpreter ... **117**, *128*
Interrupt ... **83**
Invariante ... **104**, *113*, *114*
Inverter ... *57*
IP *86*
  -Adresse ... *86*
  Telefonie ... *88*
IP-Adresse ... **86**
IRC ... *88*
Is-a-Beziehung ... **173**, *174*
Isolation ... **184**
Iteration ... *109*, s. a. Schleife
  externe ... **166**
  interne ... **166**

J

Java *108*, *111*, *117*, *126*, *130*, *131*, *133*, *152*, *164*, *183*
  Applets ... *89*
  Bytecode ... *132*
Java Virtual Machine ... s. JVM
JavaScript *128*, *130*, *131*, *133*, *150*, *168*, *178*, *179*
JavaScript Object Notation ... *179*
JPEG ... *48*
JSON ... *179*
JVM ... **117**, *132*, *148*

K

Kante ... *179*

Kardinalia ... *19*
Kettenschlussregel ... **32**
Key/value-Paar ... *177*
Klasse ... **126**, *164*
Knoten ... *179*
Kombinatorik ... *63*
kombinatorische Logik ... **63**
Kommentar ... **107**, *109*, *145*
Kommunikationsprotokoll ... *86*
Komplementdarstellung ... **15**
Komplexität ... *115*
  akzidentelle ... **135**, *183*
  essenzielle ... **135**
Komplexitätsanalyse ... *101*
Komposition ... **173**
Kompression ... *43*, **48**
  verlustbehaftete ... *48*
  verlustfreie ... *48*
Konjunktion ... **23**, *36*, *58*, *158*
Konkatenation ... **43**, *153*
Konklusion ... **30**, *124*
Konsistenzbedingung ... *185*
Konsistenzerhaltung ... **184**
Konsole ... **84**, *87*
Kontradiktion ... **28**
kontrolliertes Vokabular ... **175**
Kontrollstruktur ... **118**
Korrektheit ... *98*, *102*, *103*
  totale ... *114*
  von Algorithmen ... *100*, *105*
  von Programmen ... *112*, *114*, *120*
Korrektheitsbeweis ... *100*, *104*, *112*, *113*, *123*
  für einen Algorithmus ... *104*
kulturelles Erbe ... *174*
künstliche Intelligenz ... *35*, **37**, *106*, *125*

L

Label/value-Paar ... *177*
Labyrinth ... *167*
LAN ... *86*
Language Integrated Query ... *183*
Lastenheft ... **102**
Latch ... *65*, *67*
Laufzeit ... *79*, *101*, *114*, *115*, *126*, *127*, *161*, *168*
  von Signalen ... *60*
Leerzeichen ... *1*, *3*, *8*, *41*, *107*
legacy system ... *103*
Leibniz ... *22*, *101*
Leibniz, Gottfried Wilhelm ... *i*, *9*, *75*, *175*
Link ... s. Hyperlink
Lisp ... *131*
Literal ... *42*, *168*
  String- ... **42**
  Zeichenketten- ... **42**, *153*
Local Area Network ... *86*
Logik ... **21**, *62*, *123*, *125*, *168*
logische Programmiersprache s. Programmiersprache
logische Programmierung ... s. Programmierung

M

Makro ... **152**
Makrorecorder ... **152**
Markdown ... *180*
Markup ... **179**
Markup language ... **180**
Maschinenbefehl ... **50**, *70*, *106*, *117*
Maschinencode ... **50**, *117*, *136*, *176*
Maschinenprogramm ... *107*
Maschinensprache ... **50**, *94*, *107*, *118*, *122*, *151*
Mechanical turk ... *38*
mehr vom Selben ... *10*, *11*, *76*
Mehrbenutzer/-in ... *82*
Mehrbenutzung
  gleichzeitige ... *83*
Mehrkernprozessor ... *83*
Memory-mapped I/O ... *71*
Mengenlehre ... *33*
Mereologie ... *173*
Metaebene ... **2**, *25*, *171*, *173*
Metamodellierung ... **174**
Metaprogramm ... **117**, *135*
Metaprogrammierung ... **123**
Metasprache ... **22**, *30*
Metatyp ... **159**, *174*
Metavariable ... *30*
Methode ... **126**
Mikroprozessor ... **69**, *83*
Minotaurus ... *167*
ML *131*
Mnemonic ... *50*
Modul ... *76*
Modularität ... *59*
Modus ponens ... **31**, *124*
Modus tollens ... **31**
Morsealphabet ... *48*
Morsecode ... *43*

MP3 ... *48*
MPEG ... *48*
Multitasking ... **83**
Münchhausen ... *78*
-Trilemma ... *2*

N

Nachbedingung ... *100*, **103**, *112*, *114*
Nachher ... **21**, *61*, *111*, *123*
Name ... s. Bezeichner
Name/value-Paar ... *177*
NaN ... *19*
natürliche Sprache ... *26*, *36*, *169*
Abgeschlossenheit ... *22*
Verarbeitung ... *125*
Navigation ... **89**, *184*
Negation ... **23**, *57*
Network News Transfer Protocol ... *88*
neuronales Netz ... *35*
Newsgroup ... *88*
Newton, Isaac ... *101*
Nicht ... *33*
-Gatter ... **57**
nichtdeterministischer Automat ... **63**
nichtflüchtiger Speicher ... *66*, s. Festspeicher
Nichtterminal ... *176*
NNTP ... *88*
No-free-lunch-Theorem ... *97*
Not a Number ... *19*
Nullelement ... **33**
Nullzeiger ... **162**
Nur-Lese-Speicher ... **68**

O

obfuscation ... *106*
Objekt ... *41*, *44*, *98*, **126**, *128*, *177*
objektbasiert ... *140*, s. a. Programmierung
objektbasierte Programmierung ... *142*
Objektebene ... **2**
objektorientiert ... *140*, s. a. Programmierung
objektorientierte Datenbank ... **184**
objektorientierte Programmiersprache ... *123*, *131*
objektorientierte Programmierung ... **126**
objektorientiertes Persistenzframework ... **183**
Objektsprache ... **22**, *30*
OCR ... *73*
Oder ... *33*
exklusives ... **24**
-Gatter ... **58**
inklusives ... **24**, *27*
Ontologie ... *168*, **174**
Ontologien ... **174**
Open-source-Software ... *106*
Operation ... *50*, *133*
abstrakte ... *33*
Operationen
auf Zeichenketten ... *43*
Operationscode ... *50*
Operator ... **108**
Operatoren ... *23*
optische Zeichenerkennung ... *73*
optischer Datenträger ... **72**
Ordinalia ... *19*
Out-of-memory-Fehler ... *79*, *163*

P

Papier/Digital-Brücke ... *iii*
parallele Programmierung *123*, s. Programmierung
paralleler Bus ... **68**
paralleles Programm ... *134*
Paritätsbit ... *40*
Pascal ... *108*, *110*, *121*, *126*, *130*, *160*, *162*
-Programm ... *119*
PC *82*
permanenter Speicher ... *66*
persistenter Speicher ... s. Festspeicher
Persönlicher Computer ... *82*
Pflichtenheft ... **102**
PHP ... *133*
Phrasenstrukturgrammatik ... *176*
Picture element ... *45*
Pipeline ... *120*
Pixel ... **45**
Plug-and-play ... *78*
Pointer ... *127*, *161*
POP ... *86*
Portierung ... *76*
Portnummer ... **86**, *150*
Post Office Protocol ... *86*
Postcondition ... s. Nachbedingung
Prädikat ... *36*, *170*
Prädikatenlogik ... **36**, *123*
erster Stufe ... *37*
monadische ... *37*
sortierte ... *170*
Prämisse ... **30**, *124*

Precondition ..... s. Vorbedingung
Primärschlüssel **172**, *179*
Prinzip der Substituierbarkeit *127*
privilegierter Modus **77**, *95*
Programm **50**, *106*
-abbruch *79*, *95*, *112*, *162*
-absturz *94*, *163*
-aufruf *82*
imperativ *142*
imperatives *118*, *176*
Unterschied zu Algorithmus *106*
Programmierfehler
schwerwiegender *162*
Programmierparadigma **118**
Programmiersprache *18*
funktionale *131*
höhere *107*
imperative *140*, *151*
logische *123*, *131*
objektbasierte *140*
objektorientierte
funktionale Anteile *166*
prozedurale *126*
sichere *163*
strukturierte *140*
unsichere *163*
Programmierstil *109*, **147**, *166*
Programmierung
objektorientierte *142*
Programmierung
deklarative **118**
ereignisgesteuerte *140*
funktionale *105*, **121**
imperative *105*, **118**, *119*, *121*, *122*, *131*
logische *105*, **123**
objektorientierte *118*
parallele *140*
prozedurale *119*
Programmierung
objektorientierte *164*
Programmierung
objektorientierte *164*
Programmierung
logische *170*
Programmierung
objektorientierte *173*
Programmierwerkzeug *130*, **135**
Programmsynthese *137*
Programmzähler **70**
Programmzeile **107**
Prolog *109*, **123**, *131*
PROM *73*
Protokoll **86**, *89*
proprietäres *88*
Prototyp *179*
prototypenbasierte Programmierung **127**, *178*
Prozeduraufruf *119*
Prozess **83**
Prozessor *50*, **69**, *71*
Prozessumschaltung **83**
Pufferspeicher *82*
Python *107*, *126*, *131*, *133*

Q

Quantor **36**
Quellcode *136*
quelloffen *106*
query *124*, *125*

R

RAM **68**
Random Access Memory *67*, **68**
Read-Only Memory **68**, *73*
Rechner ..... *i*
Record *108*, **160**
Redundanz *54*, *115*, **129**, *138*
Refaktorisierung *145*
Reference ..... s. Referenz
Referenz *127*, **161**
Referenzmodell **174**
Register **69**
Registrierkasse *18*, *62*
Regressionstesten **137**
reifizieren *165*
rekursiv *115*, *116*, *126*
Datentyp ..... s. Datentyp
rekursive Definition **121**
Relais *56*, *57*, *59*
Relation *169*, *170*, *171*
relationale Datenbank *171*
relationales Datenmodell **171**
Remote Desktop *87*
Remote Procedure Call *87*
reserviertes Wort *3*
Resolution *124*
Resource Description Framework *179*
Ressourcenleck *83*

Role Data Model ... *184*
Rollenname ... *171*
ROM ... **68**, *73*, *77*
Root-Recht ... *82*
RPC ... *87*
Ruby ... *131*, *133*
Russellsche Antinomie ... *159*

S

Safety ... **93**
Sandboxing ... *90*
SAT-Problem ... **28**
Scala ... *135*
Schaltalgebra ... *33*, **58**
Schaltnetz ... *61*
Schaltsymbol ... *57*
Schaltwerk ... *61*
Schema ... *159*, *168*, **169**, *171*
Schemaänderung ... s. Schemaevolution
Schemaebene ... *170*
Schemaevolution ... *176*, **182**
Schemamigration ... **182**
Scheme ... *131*
Schleife ... *51*, *52*, *100*, *111*, *118*, *122*, *141*, **165**
  For- ... *165*
  For-each- ... *165*
  n+1/2- ... *119*
  Repeat-until- ... s. Wiederhole-bis
  Wiederhole-bis- ... *111*, *112*
Schleifeninvariante ... *113*
Schlüsselwort ... **108**, *109*, *110*, *132*, *153*
Schlussfolgerung ... *124*
  logische ... **29**
  semantische ... **29**
Schlussfolgerungsbegriff
  semantischer ... *23*
Schlussfolgerungskette ... **32**, *37*
Schlussfolgerungsmechanismus
  syntaktischer ... *23*
Schlussfolgerungsregel ... **30**, *37*
Schlussregel ... s. Schlussfolgerungsregel
Schnittstelle ... *76*, *77*
  Programmier-Application Programming Interface
Schnittstellen ... *181*
Schriftzeichen ... *39*
Sechzehnersystem ... s. Hexadezimalsystem
Security ... **93**
Seite ... *80*
Semantik ... **23**, *107*, *164*
  dynamische ... **129**
  statische ... **129**
semantische Regel einer Sprache ... *129*
semantisches Datenmodell ... *126*, **173**
sequentielle Datei ... **81**
sequentielle Logik ... **63**
sequentieller Zugriff ... **72**
Sequenz ... *141*
serieller Bus ... **68**
Server ... *87*
Session Initiation Protocol ... *88*
SETI@home-Projekt ... *186*
Sicherheit ... *54*, **93**, *155*
Sicherheitsproblem ... *82*
  eklatantes ... *150*
Signalstrom ... *46*
Simple Mail Transfer Protocol ... *86*
SIP *88*
Skript ... *152*, *182*
Skriptsprache ... *131*
Smalltalk ... *130*
SMTP ... *86*
Software ... *76*, *78*, *87*, *101*
Softwareentwicklungsprozess ... *102*
Solid State Drive ... **72**
Sonderzeichen ... *3*, *7*, *40*
Sorte ... *170*
Space-time trade-off ... *14*
Spaghetticode ... *119*
Speicher
  flüchtiger ... **66**
  Haupt- ... s. Hauptspeicher
  nichtflüchtiger ... s. Festspeicher
  persistenter ... s. Festspeicher
  sequentieller ... *101*
  statischer ... **67**
Speicheradresse ... **67**, *77*, *161*
  logische vs. physikalische ... *80*
  Programm- ... *119*
Speicheradressierung ... *79*
Speicherleck ... **79**
Speicherschutzverletzung ... **79**
Speicherseite ... *80*
Speicherverbrauch ... *101*
Spezialisierung ... *174*
Spezifikation ... *98*, **101**, *137*, *164*
  formale ... **103**

Spracherkennung *38*
Sprachverarbeitung *125*
Spraying *163*
Sprungtabelle **77**
SQL **182**
SRAM **67**
SSD **72**
Stammdaten *157*
Stand von Wissenschaft und Technik *89*, *155*
Standardbibliothek ..... *130*, s. a. Bibliothek
Stanford Natural Language Parser *187*
statische Semantik ..... *142*, s.
statischer Speicher ..... s. Speicher
Stelle **5**
Stellenwert **6**
Stellenwertsystem **5**
Steuerbus **67**
Steuerwerk **70**
String ..... **41**, *163*, s. a. Zeichenkette
Structured Query Language **182**
strukturierte Programmierung *107*, **119**, *141*
Subdirectory **81**
Subklasse **127**, *174*
Subroutine *152*
Subsumtionsbeziehung *126*
Subtyp *174*
Subtypbeziehung *127*, **173**
Superklasse **127**
Superuser/-in **82**
Syntax **23**, *132*, *164*
einer Programmiersprache **128**, *130*
Syntax highlighting *109*
Syntaxfehler *141*
Systematik **175**
für Bibliotheken *175*
Systemaufruf **77**

T

Tabelle
Additions- *10*
Divisions- *15*
im Sinne einer Relation *172*
Multiplikations- *14*
Wahrheits- ..... s. Wahrheitstafel
Tabellenkalkulationsprogramm *109*
Tabulator *40*, *41*
Takt *51*, **61**, *74*
Taktfrequenz **61**, *186*
Tautologie **28**, *29*, *33*
TCP *86*
Teil-Ganzes-Beziehung **173**, *176*
Telekommunikationsgeheimnis *88*
Telekommunikationsgesetz *88*
Teletype Network *87*
Telnet *87*
Terminal **84**, *87*, *93*
Terminal (Grammatik) *176*
Terminalemulation *84*, *87*
Terminierung **113**
mangelnde *114*
von Zeichenketten **42**
Testen *100*, **136**
eines Programms auf Erfüllung der Spezifikation *105*
testgetriebene Entwicklung *137*
Tests-first-Ansatz *137*
Theorembeweiser *37*
Theseus *167*
Thin client **149**, *150*
Time sharing **83**
Time-memory trade-off *14*
Transistor *59*
Transmission Control Protocol *86*
Treiber ..... **78**, s. a. Gerätetreiber
Trennung von Programm und Datenspeicher *70*
Tupel **172**
Turing, Alan *39*, *106*
Turingmaschine *52*
Turing-Test **39**
Typ **54**, *110*, *114*, *129*, *132*, *158*, *159*, *164*, *169*, *170*, *176*
primitiver *126*
Typebene **159**, *170*, *181*
Typfehler *130*, **132**, *142*, *158*
typisiert *94*, *125*, *158*, *170*
schwach *133*
Variable *142*
Typkonstruktor **160**
Typprüfung **132**, *136*, *142*, *170*
dynamische **133**
statische **133**, *137*

U

Überlauf **12**, *20*, *52*, *59*
überschreiben *127*
Übersetzung

Gegensatz zur Interpretation **117**
natürlichsprachlicher Texte *38*
Übertrag **6**, *10*, *11*, *51*, *52*, *59*
UDP *86*
Umcodierung *46*
UML *173*
Unbekannte *20*
Und *33*
-Gatter **58**
Unicode-Standard *42*
Unified Modeling Language *173*
Unifikation *124*
Universal Serial Bus *68*
Universalienstreit *169*
universeller Approximator *35*
unsicher *133*, *161*
Information *34*
unsinnig *22*
Unterprogramm **52**, *109*, *151*, *152*, *153*
Unterprogrammaufruf *77*, *107*
Unterverzeichnis **81**
Upper ontology **174**
USB *68*
User Datagram Protocol *86*
User interface ..... s. Benutzungsschnittstelle
UTF-8 ..... *42*

V

Variable *20*, *26*, *36*, *110*, *124*, *142*
gebundene *36*
globale **120**
quantifizierte *36*
typisierte *142*
VBA **152**
Vektorgrafik **46**
Verbindungsschicht *86*
Verbund **160**, *167*
Verbundtyp **160**
Vererbung **126**, *128*
Vererbungsbeziehung **173**
Verfügbarkeit *94*
Verifikation *104*, *137*
formale *133*
von Algorithmen *104*
von Programmen *112*, *164*
Verkettung **43**, *153*
Verknüpfung
logische *23*, *164*
von Daten *165*, *182*
Vermittlungsschicht *86*
Verschlüsselung *40*, **48**
asymmetrische **49**
symmetrische **49**
Versionskontrollsystem **137**
Vertraulichkeit *93*
Verweis ..... **127**, *161*, *179*, s. a. Zeiger
Verzweigung *52*, *107*, *118*, *127*, *141*
virtuelle Maschine *78*, *93*, **117**, *148*
virtueller Speicher **79**
Visual Basic **151**, *153*
for Applications **152**
Volladdierer **34**, *58*
Von-Neumann-Architektur **74**
Vorbedingung *100*, **103**, *112*, *114*
Vorher **21**, *61*, *111*, *123*
Vorzeichenbit **15**

W

Wagenrücklauf *40*, *41*
wahlfreier Zugriff **68**, *72*, *101*
Wahr *33*
Wahrheit
relative *35*
von Aussagen *26*
Wahrheitstabelle **23**
Wahrheitstafel **23**
Wahrheitswert **21**
Wahrscheinlichkeitstheorie *35*
WAN *86*
Wartung *176*
Web **88**, *187*
Web crawler *187*
Webbrowser **89**
Webserver **89**
Wert *41*
Wertzuweisung **111**, *119*, *121*
White space *41*, *107*
Wide Area Network *86*
widerspruchsfrei *30*
Widgets *150*
Wiederholung ..... *107*, *118*, *141*, s. a. Schleife
Wiederverwendung *126*
Wireless LAN *86*
Wirth, Niklaus *108*, *157*
Wissensrepräsentation *126*, *127*, *179*
Wittgenstein, Ludwig *127*

WLAN *86*
wohlgeformt *170*
World Wide Web **88**
Worst case *101*
Wort **10**, *81*
Wortbreite *68*
Wurzel
  der Sumerer/-innen *109*
  eines Baums **166**
WWW **88**
WYSIWYG-Editor **180**

X

XML **180**
XML Schema *180*

Z

Zähler **20**
  Befehls- *78*
Zehnersystem ..... s. Dezimalsystem
Zeichencode **40**
Zeichenfolge *3*, *41*
Zeichenkette **41**, *179*
  null-terminierte **42**
Zeichenkettenliteral ..... *153*, s. Literal
Zeichensatz *84*
Zeichenspiel ..... **i**, *6*, *12*, *59*, *76*
Zeiger *81*, *127*, *161*, *164*
Zeigerarithmetik **161**
Zeilenvorschub *40*, *41*
Zeit *60*
zeitliche Dimension *168*
  von Daten ..... s. Inhaltsverzeichnis
  von Hardware *60*
  von Programmen *111*, *123*
  von Signalen *47*
  von Zeichen *3*
  von Zeichenspielen *21*
Zeitreihe *47*
Zeitscheibe **83**
Zeitstempel *177*
Zero cost abstractions *130*
Ziffer **5**
Zusicherung *103*, **104**
  Unterschied Anweisung *111*
Zustand *98*, *110*, *111*, *119*, *123*, *142*
  in der Mathematik *111*
Zustandsabhängigkeit *119*
Zustandsänderung *121*
zustandslos *87*
Zustandsübergangsdiagramm **65**
Zustandsübergangstabelle **65**
Zustandswechsel *64*, *65*
  einer Datenbasis *176*
  eines Programms *119*
Zuweisung **111**, *165*
Zweiersystem ..... s. Dualsystem
Zwölfersystem *6*, *16*

000 000 000 (00/18)

00000-0-00-S1

Alle Rechte vorbehalten
© 2018 FernUniversität in Hagen
Fakultät für Mathematik und Informatik