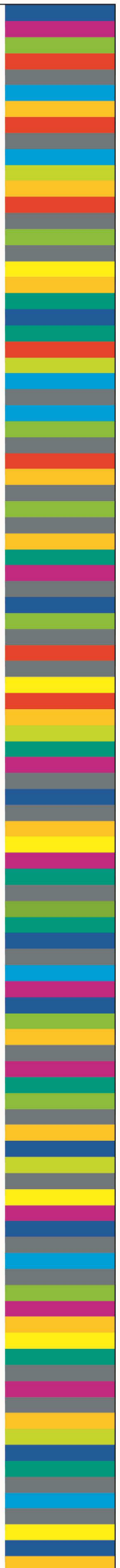


Grundlagen der Informatik

Skriptum

FH-Prof. DI Alexander Nimmervoll

Version 1.0



Allgemeines

Die Skripten inkl. des Begleitmaterials (z.B. Software) einschließlich Anweisungen für ihre Anwendung und sämtliche gedruckte oder elektronische Dokumentation wird im gegenwärtigen Istzustand und ohne jegliche Gewährleistung bereitgestellt. Des Weiteren gibt es keine Gewähr auf Richtigkeit, Vollständigkeit und/oder Funktionalität. Alle Risiken, die sich aus der Verwendung oder Funktion der Software und/oder der Dokumentation ergeben, trägt der Benutzer.

Die Softwareprogramme und Tools sind gemäß Urheberrechtsgesetz geschützt.

Wir weisen ausdrücklich darauf hin, dass wir keinen Einfluss auf die Inhalte und Aussagen von Links haben. Alle Angaben sind daher informativ und ohne Gewähr.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in den Studienbriefen berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Waren- und Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Vorgestellte Methoden und Vorgangsweisen beruhen auf den Büchern und Artikeln, die im Literaturverzeichnis angeführt sind.

Für Anmerkungen und Hinweise auf Fehler sind wir sehr dankbar!

Inhaltsverzeichnis

1	EINLEITUNG	VII
1.1	LEHRSTOFF UND MOTIVATION	VII
	<i>Kapitelübersicht</i>	VII
1.2	LERNERGEBNISSE	VIII
1.3	ZIELGRUPPE	VIII
1.4	VORAUSSETZUNGEN	VIII
1.5	LERNWEGEMPFEHLUNG	VIII
	<i>Wie gehen sie vor</i>	VIII
	<i>Termine</i>	IX
	<i>Zeitaufwand für Lernstoff und Übungen</i>	IX
2	GRUNDLAGEN DER INFORMATIONSVERRARBEITUNG	2—1
2.1	EINFÜHRUNG UND DEFINITION	2—1
	<i>Teilgebiete der Informatik</i>	2—1
	<i>Lernergebnisse</i>	2—3
2.2	INFORMATIONSVERRARBEITUNG UND CODIERUNG	2—3
	<i>Grundbegriffe und Abkürzungen</i>	2—3
	<i>Textcodierung</i>	2—5
2.3	BOOLESCHE ALGEBRA	2—7
	<i>Gesetze der booleschen Algebra</i>	2—9
2.4	ZAHLENSYSTEME	2—10
	<i>Umwandlung von Zahlen in verschiedene Zahlensysteme</i>	2—11
	<i>Rechnen im Binärsystem</i>	2—12
	<i>Darstellung von negativen Zahlen</i>	2—14
	<i>Gleitkommadarstellung reeller Zahlen</i>	2—17
	<i>Gleitkommaarithmetik und Genauigkeit</i>	2—20
2.5	INFORMATIONSTHEORIE	2—21
	<i>Shannon Entropie</i>	2—21
	<i>Huffman Codierung</i>	2—23
2.6	SELBSTLERNKONTROLLE	2—26
	<i>Fragen</i>	2—26
	<i>Lösungen</i>	2—27
2.7	ÜBUNGSTEIL	2—28
	<i>Übung 1: Boolesche Algebra</i>	2—28
	<i>Übung 2: Zahlensysteme</i>	2—28
	<i>Übung 3: 2 Komplement</i>	2—28
	<i>Übung 4: Gleitpunktdarstellung</i>	2—28
	<i>Übung 5: Huffman Kodierung</i>	2—28
3	EINFÜHRUNG IN DIE AUTOMATENTHEORIE	3—29
3.1	DEFINITION UND INFORMELLE EINFÜHRUNG	3—29
	<i>Automat</i>	3—29
	<i>Automatentheorie</i>	3—30
	<i>Lernergebnisse</i>	3—30
3.2	MOTIVATION	3—31
3.3	BEGRIFFE DER AUTOMATENTHEORIE	3—31
	<i>Alphabet</i>	3—31
	<i>Zeichenreihe</i>	3—31
	<i>Sprache</i>	3—32
	<i>Operationen auf Sprachen</i>	3—33
3.4	DETERMINISTISCHE ENDLICHE AUTOMATEN	3—34
	<i>Definition</i>	3—35
	<i>Darstellungsmöglichkeiten</i>	3—35
	<i>Beispiel für einen DEA</i>	3—36
	<i>Verhalten und Sprache eines DEA</i>	3—38
	<i>Aufgabenstellungen in der Praxis</i>	3—39
3.5	SELBSTLERNKONTROLLE	3—41
	<i>Fragen</i>	3—41
	<i>Lösungen</i>	3—43

3.6	ÜBUNGSTEIL	3—45
	Übungstool JFLAP.....	3—45
	Übung 1: Mausefalle.....	3—45
	Übung 2.....	3—45
	Übung 3: Getränkeautomat	3—45
	Übung 4.....	3—45
	Übung 5.....	3—46
	Übung 6: Marmelade.....	3—46
	Übung 7: Marmelade.....	3—46
4	FORMALE SPRACHEN UND GRAMMATIKEN.....	4—1
4.1	REGULÄRE SPRACHEN	4—1
	Lernergebnisse.....	4—1
4.2	REGULÄRE AUSDRÜCKE	4—1
	Motivation.....	4—1
	Zusammenhang mit endlichen Automaten.....	4—2
	Definition	4—2
	Beispiel.....	4—3
	Erweiterte Notationen.....	4—4
4.3	GRAMMATIKEN.....	4—5
	Definition	4—5
	Ableitung und Sprache.....	4—6
	Reguläre Grammatiken.....	4—7
4.4	KONTEXTFREIE SPRACHEN	4—7
	Grenzen regulärer Sprachen.....	4—7
	Kontextfreie Grammatiken	4—8
	Kellerautomaten.....	4—11
	Anwendungen.....	4—11
	Erweiterte Backus-Naur-Form und Syntaxdiagramme.....	4—12
4.5	SPRACHKLASSEN	4—13
	Chomsky-Hierarchie	4—14
4.6	SELBSTLERNKONTROLLE	4—15
	Fragen.....	4—15
	Lösungen.....	4—16
4.7	ÜBUNGSTEIL	4—17
	Übung 1: Reguläre Ausdrücke.....	4—17
	Übung 2.....	4—17
	Übung 3: Grammatiken	4—17
	Übung 4.....	4—18
	Übung 5.....	4—18
5	TURING MASCHINEN UND BERECHENBARKEIT (OPTIONAL).....	5—1
5.1	EINLEITUNG	5—1
5.2	DEFINITION UND FUNKTIONSWEISE	5—2
5.3	BEISPIEL	5—3
5.4	DER BEGRIFF ALGORITHMUS	5—4
5.5	BERECHENBARKEIT.....	5—5
5.6	SELBSTLERNKONTROLLE	5—7
	Fragen.....	5—7
	Lösungen.....	5—8
5.7	ÜBUNGSTEIL	5—9
	Übung 1: Einführung Visual Turing	5—9
	Übung 2: Addition von 1.....	5—9
	Übung 3: Sortierer.....	5—9
6	ALGORITHMEN	6—1
6.1	GRUNDLEGENDE BEGRIFFE.....	6—1
	Lernergebnisse.....	6—1
	Bestandteile eines Algorithmus.....	6—2
6.2	FLUSSDIAGRAMME UND STRUKTOGRAMME.....	6—2
6.3	EIGENSCHAFTEN VON ALGORITHMEN	6—6
	Beispiel Primzahlalgorithmus.....	6—6

	<i>Korrektheit von Algorithmen</i>	6—8
	<i>Terminierung</i>	6—9
	<i>Determinismus und Determiniertheit</i>	6—9
6.4	KOMPLEXITÄT VON ALGORITHMEN	6—10
	<i>Aufwandsabschätzung</i>	6—11
	<i>Die O-Notation</i>	6—12
	<i>Aufwandsabschätzung Beispiel Primzahlalgorithmus</i>	6—13
	<i>Ausblick Traveling Salesman Problem</i>	6—15
6.5	SELBSTLERNKONTROLLE	6—19
	<i>Fragen</i>	6—19
	<i>Lösungen</i>	6—20
6.6	ÜBUNGSTEIL	6—21
	<i>Übung 1: Quersumme</i>	6—21
	<i>Übung 2: Druckeinheiten</i>	6—21
	<i>Übung 3: Reihenentwicklung von π</i>	6—22
	<i>Übung 4: Determinismus und Determiniertheit</i>	6—22
	<i>Übung 5: Bonbonglas</i>	6—22
7	ANLAGEN	7—24
7.1	LITERATURVERZEICHNIS.....	7—24
7.2	INTERNETQUELLEN	7—24
7.3	ABBILDUNGSVERZEICHNIS.....	7—25

1 Einleitung

1.1 Lehrstoff und Motivation

Das Ziel der Lehrveranstaltung ist, den TeilnehmerInnen den Einstieg in die Grundlagen der theoretischen Aspekte der Informatik zu ermöglichen und die Voraussetzungen für weiterführende Programmierlehrveranstaltungen zu vermitteln.

Ein Ziel der Informatik ist sicherlich, Methoden und Werkzeuge zu liefern, mit denen Probleme mittels Informationstechnologie gelöst werden können. Bevor wir allerdings Algorithmen und Datenstrukturen für so eine Problemlösung angeben können, benötigen wir folgende zwei Hilfsmittel:

- ♦ **Formale Sprachen**, mit denen die Problemstellung und das Lösungsverfahren möglichst exakt beschrieben werden können.
- ♦ **Abstrakte Rechnermodelle** (Automaten, Maschinen) zur Beschreibung des konzeptionellen Aufbaus und der Wirkungsweise von Informationssystemen, um den Begriff Algorithmus präzisieren zu können, und die Komplexität von Algorithmen sowie die Grenzen der Berechenbarkeit von Problemen analysieren zu können.

Abstraktion und Verständnis von **formalen analytischen und strukturierten Denkprozessen** ist eine Grundfertigkeit, die von jeder Technikerin und jedem Techniker gefordert wird. Wenn wir uns also grundlegend auch mit den theoretischen Aspekten der Informatik beschäftigen, wird genau dieses Verständnis wesentlich gefördert und hilft auch bei der praktischen Umsetzung von Aufgabenstellungen der Programmierung und des Software Engineerings. So liefern endliche Automaten die Grundlagen für dynamische Modelle im objektorientierten Softwareentwurf sowie von Benutzerschnittstellen. Formale Sprachen und Grammatiken sind die Grundlage jedes Compilers aber auch für Datenaustauschformate wie z.B. XML. Diese Lehrveranstaltung unterstützt somit auch die parallele Einführung in die C-Programmierung.

Jeder, der diese theoretischen Grundlagen kennt und über die Grenzen der Berechenbarkeit und Grundzüge der Komplexitätstheorie Bescheid weiß, wird z.B. nicht nach einem exakten Algorithmus zur Lösung eines Traveling-Salesman-Problems¹ suchen. Die Chancen, einen solchen zu finden, sind aufgrund der Erkenntnisse der Komplexitätstheorie äußerst gering. Sehr wohl relevant sind in der Praxis allerdings Näherungslösungen, die zwar einen gewissen Fehler aufweisen, allerdings sehr rasch berechnet werden können.

Im Folgenden soll eine kurze Übersicht über den Aufbau des Skriptums gegeben werden.

Kapitelübersicht

Der Rest dieses Kapitels beschäftigt sich noch einmal mit einer Zusammenfassung der Lernergebnisse und mit organisatorischen Rahmenbedingungen sowie einer Lernwegempfehlung, um die definierten Lernergebnisse auch zu erreichen.

Kapitel 2 gibt einen Überblick über die Teilgebiete der Informatik und vermittelt die Grundlagen der Informationsverarbeitung von Codierung über Zahlensysteme, Boolesche Algebra bis zu Informationstheorie.

¹ Gesucht beim Traveling-Salesman-Problem ist eine Rundreise durch n Städte mit minimalen Wegkosten, was in der Praxis bei Logistik Unternehmen oft benötigt wird. Mehr dazu in Kapitel 6.

Kapitel 3 beschäftigt sich mit den Grundlagen der Automatentheorie. Es werden wichtige Begriffe definiert und endliche Automaten vorgestellt.

Kapitel 4 erläutert die Grundlagen formaler Sprachen und Grammatiken. Reguläre Ausdrücke als weiteres Beschreibungskonzept und die Zusammenhänge mit endlichen Automaten werden diskutiert. Kontextfreie Sprachen und eine vollständige Hierarchie von formalen Sprachklassen bilden den Abschluss dieses Kapitels.

Kapitel 5 stellt mit den Turing Maschinen ein wichtiges abstraktes Beschreibungsmittel für Algorithmen vor und bildet somit die Überleitung zu den Programmierlehrveranstaltungen.

Kapitel 6 schließlich, beschreibt Kontrollstrukturen von Algorithmen und Flussdiagramme sowie die wesentlichen Eigenschaften von Algorithmen und geht speziell auf die Komplexität von Algorithmen ein.

1.2 Lernergebnisse

Nach erfolgreichem Abschluss sind die Studierenden in der Lage,

- Informationen im Binärsystem zu kodieren und Umwandlungen in für die Informatik relevanten Stellenwertsystemen durchzuführen.
- einfache abstrakte Aufgabenstellungen der formalen Sprachen als endliche Automaten zu modellieren und/oder als regulären Ausdruck bzw. kontextfreie Grammatik darzustellen.
- eine Problemstellung mittels Flussdiagramm zu modellieren und damit den Begriff Algorithmus zu erklären.

1.3 Zielgruppe

StudentInnen des 1.Semesters des Bachelor Studiengangs Verkehr und Umwelt am Technikum Wien.

1.4 Voraussetzungen

Es sind keine speziellen Vorkenntnisse notwendig.

1.5 Lernwegempfehlung

Wie gehen sie vor

Lesen Sie das Skriptum in der vorgegebenen Reihenfolge durch und versuchen Sie alle Beispiele selbstständig durchzuarbeiten und zu verstehen. Die Kapitel sind aufbauend aufeinander gestaltet. Simulationsprogramme für Automaten und Turing Maschinen werden im Laufe des Skriptums präsentiert und sollen zusätzlich das Verständnis fördern. In jedem Kapitel gibt es am Ende eine Selbstlernkontrolle mit Multiple-Choice Fragen zur Überprüfung der Lehrinhalte des Kapitels samt Lösungen. Weiters gibt es zu jedem Kapitel eine Reihe von Übungsaufgaben, die bis zur nächsten Lehreinheit vorzubereiten sind.

Termine

Die Termine sind dem Semesterplan zu entnehmen.

Zeitaufwand für Lernstoff und Übungen

Der Gesamtaufwand der Lehrveranstaltung beträgt 1 SWS bzw. 1.5 ECTS Punkte. Ein ECTS Punkt bedeutet ca. 25 volle Stunden Aufwand, d.h. der Aufwand für Grundlagen der Informatik beträgt insgesamt ca. 37,5 volle Stunden. Darin inkludiert ist die Anwesenheitszeit, die Vorbereitungszeit auf die Prüfung und die Phasen des Selbststudiums sowie die Lösung von Übungsaufgaben.

2 Grundlagen der Informationsverarbeitung

Dieses Kapitel soll einen ersten Einstieg in die Welt der Informatik geben und einen Bogen von grundlegenden Definitionen und Begriffen wie Information, Daten und Kodierung über Logik und Zahlensysteme bis hin zur Informationstheorie spannen. Jedes dieser Themen würde natürlich eine eigene Lehrveranstaltung beanspruchen, um es im Detail zu studieren, hier geht es wirklich nur um einen ersten Überblick, um die Sprache der Informatik verstehen zu können.

2.1 Einführung und Definition

Im deutschen Sprachgebrauch kann **Informatik** wie folgt definiert werden:

„Informatik (Computer Science): Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (Computer).“ [Claus, 2006]

Im angloamerikanischen Raum spricht man meistens von Computer Science, also den Computerwissenschaften, der Begriff Informatics ist eher im Zusammenhang mit Anwendungen wie Bioinformatics oder Business Informatics geläufig und hat eine etwas andere Bedeutung.

In der obigen Definition ist Information der zentrale Begriff, jedoch als Begriff sehr vage, vieldeutig mit teilweise widersprechenden Bedeutungen und Definitionen. Informationen werden im Computer als Daten verarbeitet und als Bitmuster dargestellt, mehr dazu in Kapitel 2.2.

Der englische Begriff „Computer Science“ betont hingegen mehr, dass Informatik eine Wissenschaft ist. Hierbei wird die Informatik auch gerne als die Wissenschaft von der Informationsverarbeitung in Natur, Technik und Gesellschaft definiert.

Um einen Überblick über tatsächliche Anwendungsfelder der Informatik zu geben, sollen im nächsten Kapitel die Teilgebiete der Informatik dargestellt werden.

Teilgebiete der Informatik

Grob können wir folgende Teilgebiete der Informatik identifizieren:

- Theoretische Informatik
 - Theoretische Grundlagen
- Technische Informatik
 - „Hardware“
- Praktische Informatik
 - „Software“
- Angewandte Informatik
 - Einsatz der technischen und praktischen Informatik in anderen Forschungsfeldern
- Wirtschaftsinformatik
 - Informationssysteme und Infrastrukturen von Organisationen in Wirtschaft und Verwaltung

Die **theoretische Informatik** beschäftigt sich mit den theoretischen Grundlagen abstrahiert von den technischen Gegebenheiten von Computern. Da Theorie und Praxis in der Informatik sehr eng miteinander verwoben sind, können theoretische Erkenntnisse rasch praktisch eingesetzt werden. Es werden Modelle von Computern aufgestellt und die Frage gestellt, was sich überhaupt berechnen lässt, basierend auf mathematischen Begriffen und Methoden.

Grundlage dabei ist die **mathematische Logik** mit:

- Aussagenlogik
- Prädikatenlogik
- Gleichungslogik

Als Berechnungsmodelle kommen Modelle von Computern (**Algorithmen**) zum Einsatz. Die Berechnung im Sinne der Informatik ist die Ausführung eines Programms durch eine Maschine, die Beschreibung von Komponenten wird ohne Bezug auf konkrete Rechner oder Programmiersprachen sondern abstrakt durchgeführt.

Als Inhalte der theoretischen Informatik werden oft angeführt:

- Formale Sprachen und Automaten
 - Untersuchung der Struktur von Zeichenketten
 - z.B. Syntax von Programmiersprachen
 - Automaten sind Modelle von Algorithmen, die Sätze von Sprachen erkennen und syntaktisch analysieren
- Grenzen der Berechenbarkeit
 - Welche Funktionen können überhaupt berechnet werden?
 - Gibt es nicht berechenbare Probleme?
- Komplexitätstheorie
 - Welcher Aufwand (Zeit, Speicher) ist für eine Berechnung erforderlich?

Die **technische Informatik** behandelt die Konstruktion und den Aufbau von Computern (Hardware) insbesondere:

- Schaltkreise und digitale Logikschaltung
- Prozessoren
- Speicher- und Bussysteme
- Externe Speicher und Peripheriegeräte
- Rechnernetze

aber auch die Schnittstelle zu Systemsoftware und Embedded Systeme.

Die **praktische Informatik** behandelt alle Bereiche, die die Programmierung und Entwicklung von Software betreffen und stellt das umfangreichste Gebiet der Informatik dar. Dazu gehören insbesondere:

- Algorithmen und Datenstrukturen, Programmiersprachen, Compilerbau
- Betriebssysteme, verteilte Systeme
- Software Architektur, Design Patterns, Software Engineering

Die **angewandte Informatik** dient der Erforschung und Entwicklung von Anwendungen, wobei praktische und angewandte Informatik schwer zu trennen sind, weil bei beiden die Programmierung im Mittelpunkt steht. Die angewandte Informatik sieht jedenfalls den Computer als Werkzeug zur Lösung von Aufgaben. Beispiele für Anwendungsgebiete sind:

- Computergrafik
- Multimedia
- Datenbanksysteme
- Text- und Dokumentenverwaltung
- Simulationen
- Künstliche Intelligenz
- Internetdienste

Wirtschaftsinformatik wird manchmal als Teil der angewandten Informatik bezeichnet, aber ist mittlerweile ein großes eigenständiges Teilgebiet der Informatik und befasst sich mit Information und Kommunikation im betrieblichem Umfeld sowie der Architektur und Konstruktion von Informationssystemen und Informationsmanagement.

Lernergebnisse

Nach diesem Kapitel sind die Studierenden in der Lage,

- Informationen im Binärsystem zu kodieren und Umwandlungen in für die Informatik relevanten Stellenwertsystemen durchzuführen.
- Ein Gesetz der booleschen Algebra mit Wahrheitstabellen zu überprüfen.
- Für ein gegebenes Alphabet mit Auftrittswahrscheinlichkeiten der einzelnen Symbole den optimalen Huffman Code zu bestimmen.

2.2 Informationsverarbeitung und Codierung

Das folgende Kapitel soll einen Überblick über die Begriffe Information, Daten und Codierung liefern sowie gängige Abkürzungen definieren.

Grundbegriffe und Abkürzungen

Folgende Abbildung beschreibt den Zusammenhang zwischen Information und Daten.

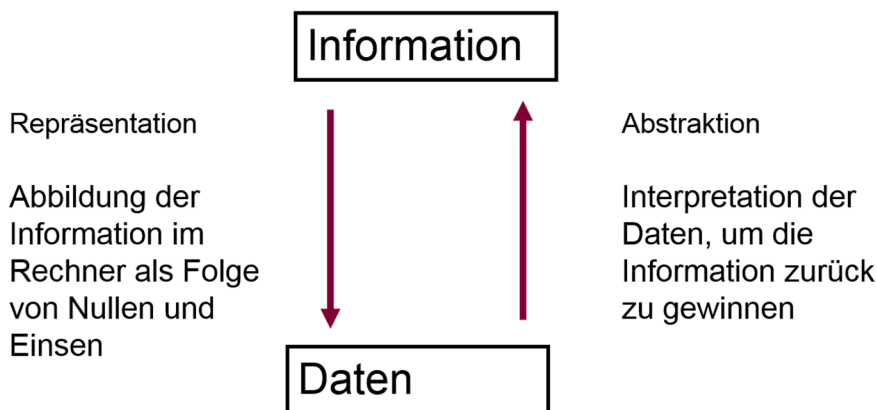


Abbildung 2-1 Zusammenhang Information und Daten

Um **Information** computergestützt verarbeiten zu können, muss sie geeignet repräsentiert werden, hierzu ist eine Abbildung auf **Daten** notwendig. Umgekehrt, haben Daten per se noch keine Bedeutung, sondern werden erst durch Interpretation zu Information.

Eine passende Informationsdarstellung muss hierbei für verschiedenste Quellen gefunden werden, von logischen Werten über einzelne Zeichen oder Zahlen bis hin zu ganzen Texten, Bildern, Tönen oder Videos. Eine geeignete **Repräsentation** dieser Informationen wird auch **Kodierung** genannt. Hierzu werden meist **Bitfolgen**, also Datenströme aus Nullen und Einsen verwendet.

Ein **Bit** stellt dabei die kleinste mögliche Informationseinheit dar, und leitet sich aus einer Wortschöpfung aus binary und digit her. Ein Bit kann zwei Zustände annehmen:

- ja / nein
- wahr / falsch
- 1 / 0

Oder elektrotechnisch interpretiert:

- geladen / ungeladen
- 5V Spannung / 0V Spannung
- magnetisiert / nicht magnetisiert

Die kleinste im Rechner adressierbare Einheit stellt nun das **Byte** dar, eine **Bitfolge** bestehend aus **8 Bits**.

Für komplexere Informationen reichen einzelne Bytes natürlich nicht aus und es werden folgende Präfixe verwendet: Kilobyte (kB), Megabyte (MB), Terabyte (TB) usw. Nach wie vor gibt es jedoch Mehrdeutigkeiten beim Begriff Kilo, ob hier standardkonform 10^3 also 1000 oder in der Informatik auch gebräuchlich die 2er Potenz $2^{10} = 1024$ verwendet wird. Dies kann gerade bei Herstellerangaben von Festplattengrößen oder Bandbreiten zu erheblichen Unterschieden führen.

Wenn wir 2 als Basis nehmen, gelten folgende Größenangaben:

- | | |
|---|------|
| – 1 k = $1024 = 2^{10}$ | Kilo |
| – 1 M = $1024 \times 1024 = 2^{20}$ | Mega |
| – 1 G = $1024 \times 1024 \times 1024 = 2^{30}$ | Giga |
| – 1 T = $1024 \times 1024 \times 1024 \times 1024 = 2^{40}$ | Tera |
| – 1 P = 2^{50} | Peta |

Betrachten wir z.B. 128 GB, so ist es ein Unterschied ob hier 128.000.000.000 Byte oder $128 \times 1024 \times 1024 \times 1024$ also 128.849.018.880 Byte gemeint sind.

Um Mehrdeutigkeiten zu vermeiden, wurden neue Einheiten Vorsätze definiert, die nur in der binären Bedeutung verwendet werden sollten. Dabei wird eine den SI-Präfixen ähnlich lautende Vorsilbe ergänzt um die Silbe „bi“, die klarstellt, dass es sich um binäre Vielfache handelt. [Wikipedia]

So gilt hier also:

1 Kibibyte (KiB) = 1024 Byte, 1 Mebibyte (MiB) = 1024×1024 Byte = 1.048.576 Byte.

Nicht alle Hersteller halten sich jedoch an diese Empfehlung, wie z.B. Microsoft.

Eine **Kodierung** ist nun definiert als Festlegung der Abbildungsvorschrift zwischen Informationen und Bitfolgen, wobei mathematisch gesehen, eine umkehrbar eindeutigen Abbildung benötigt wird.

Generell gilt, dass **mit n Bits 2^n Zustände kodiert** werden können.

Zum Beispiel können wir mit 3 Bits 8 Zustände modellieren und als folgende 8 unterschiedliche Bitfolgen darstellen: 000,001,010,011,100,101,110,111. Diesen 8 Zuständen kann nun eindeutig eine Information zugeordnet werden, z.B. jeweils eine der Himmelsrichtungen N,NE,E,SE,S,SW,W,NW. Eine konkrete Kodierung könnte hier also als Tabelle wie folgt aussehen:

Himmelsrichtung	Code
N	000
NE	001
E	010
SE	011
S	100

SW	101
W	110
NW	111

Für welche Art von Informationen müssen wir nun eine passende Kodierung von Nullen und Einsen finden? Neben logischen Werten, die wir uns noch etwas genauer im Kapitel 2.3 ansehen werden und Zahlen (natürliche Zahlen, ganze und reelle Zahlen), die wir im Kapitel Zahlensysteme 2.4 ausführlich behandeln werden, sind es natürlich sehr oft Texte, Bilder, Musik und Videos die in der Informatik verarbeitet werden müssen. Verfahren zur Speicherung von Multimediadaten würden hier den Rahmen sprengen, das folgende Kapitel beschreibt daher nur Grundlagen der Textcodierung.

Textcodierung

Auch einzelne Buchstaben und natürlich ganze Texte müssen in Bitfolgen kodiert werden. Benötigt werden neben den 26 Großbuchstaben und 26 Kleinbuchstaben, die 10 Ziffern sowie diverse Satz- und Sonderzeichen, also grob geschätzt einmal ca. 100 verschiedene Zeichen. Somit wäre ein Code der Länge 7 Bit fürs erste einmal ausreichend, da wir bereits wissen, dass mit 7 Bits 2^7 also 128 Zeichen kodiert werden können.

Der bekannte **ASCII Code** (American Standard Code for Information Interchange) ist nach wie vor die Grundlage der Textcodierung und wurde 1963 durch die American Standards Association festgelegt. Es handelt sich um einen 7 Bit Code, der neben den Buchstaben und Ziffern diverse Steuerzeichen enthält.

Die vollständige Tabelle der 128 Zeichen sieht wie folgt aus. So finden sich z.B. die Ziffer 0 an Position 48, "1" an 49, danach das große "A" an 65 und das kleine "z" an Position 122. Da 1 Byte also 8 Bit sowieso die kleinste Speichereinheit darstellt, wurde sehr bald der ASCII Code auf ein 8tes Bit erweitert, man spricht hier auch von **Extended ASCII Code**, um weitere regionale Sonderzeichen abbilden zu können. Hierbei gibt es aber mehrere Ausprägungen für die Kodierung der zweiten 128 Zeichen, die ersten sind immer ident mit dem Original ASCII Code. Bekannt sind z.B. der **ISO 8859-1 Code**, auch ISO Latin 1 genannt, der die Umlaute der meisten westeuropäischen Länder enthält, sowie ISO 8859-2 für osteuropäische und ISO 8859-5 für kyrillische Schriftzeichen.

Dec	Hx	Oct	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

Abbildung 2-2 ASCII Table [<http://www.asciitable.com/>]

Wenn wir allerdings weltweit alle möglichen Schriftzeichen abbilden wollen, man denke nur an alle chinesischen und japanischen Zeichen beispielsweise, erkennt man schnell, dass auch der erweiterte 8 Bit Code mit seinen 256 möglichen Kodierungen, bei weitem nicht ausreichen wird.

Das **Unicode** Consortium (internationaler Unicode Standard) hat seit 1984 das Ziel für jedes Zeichen aller bekannten Schriftkulturen und Zeichensysteme eine eindeutige Kodierung festzulegen. Hierbei sind bis zu 32 Bit pro Zeichen möglich, damit könnten über vier Milliarden (exakt 2^{32}) verschiedene Zeichen unterscheiden werden, in der Praxis gibt es aber eine Einschränkung auf etwa 1 Million erlaubte Code-Werte. Folgender Screenshot zeigt diverse Unicode Zeichen zur Auswahl in Microsoft Office 2013. Hier kann auch z.B. der Code in der Form U+2261 mit der Tastenkombination Alt+c verwendet werden, um das entsprechende Unicode Zeichen anzuzeigen, in diesem Fall „ \equiv “.

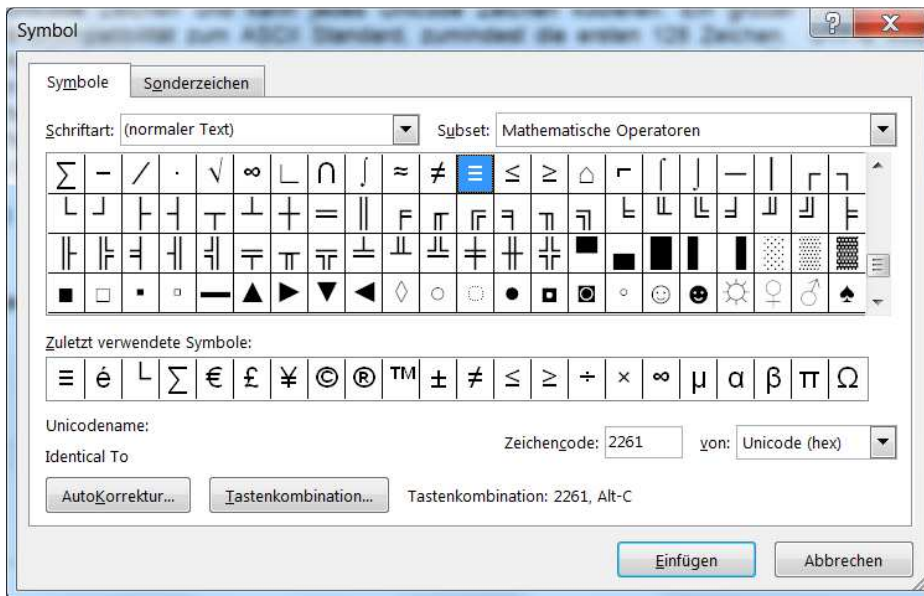


Abbildung 2-3 Unicode Symbolauswahl in Microsoft Office 2013.

UTF-8 (8-Bit UCS/Unicode Transformation Format) ist ein häufig verwendeter Code variabler Länge für Unicode Zeichen und kann jedes Unicode Zeichen kodieren. Ein großer Vorteil ist die Abwärtskompatibilität zum ASCII Standard, zumindest die ersten 128 Zeichen. UTF-8 wird in Internet Protokollen, Linux und vielen anderen Anwendungen verwendet. Jedes Zeichen wird dabei in ein bis vier sogenannte Octets kodiert (8-Bit Einheiten). Ein Byte wird für die 128 US-ASCII Zeichen benötigt, zwei Bytes für spezielle Zeichen des lateinischen Alphabets, drei Bytes für alle Zeichen der Basic Multilingual Plane (alle gebräuchlichen Unicodezeichen) und vier Bytes schlussendlich für selten genutzte Zeichen.

2.3 Boolesche Algebra

Nach dem kurzen Ausflug in die Kodierung soll nun auch ein Überblick über Grundbegriffe der **Aussagenlogik** gegeben werden, da der Umgang mit logischen Ausdrücken und den Wahrheitswerten „wahr“ und „falsch“ zu den zentralen Elementen der gesamten Informatik zählt. Von den Hardwaregrundlagen, über die Spezifikation bzw. Verifikation von Programmen bis hin zu logischen Operatoren in höheren Programmiersprachen, bedienen sich alle Bereiche diesem Teilgebiet der **mathematischen Logik**.

Formal sind Aussagen wie z.B. „Heute regnet es“ oder „7 ist eine Primzahl“ Elemente einer Menge und besitzen einen **Wahrheitswert**, der entweder den Zustand „wahre Aussage“ oder „falsche Aussage“ annehmen kann, was ja genau wieder der Informationseinheit 1 Bit (wahr/falsch, ja/nein, 1/0) entspricht. Keine Aussage darf sowohl wahr als auch falsch sein und jede Aussage muss einen dieser beiden Werte haben.

Die mathematische Grundlage definiert die **Boolesche Algebra**, entwickelt vom englischen Mathematiker George Boole. Diese basiert auf der Menge $\{0,1\}$ und den Operationen bzw. Verknüpfungen oder Wahrheitsfunktionen „und“, „oder“ sowie „nicht“. Damit können in der Aussagenlogik Aussagen verknüpft werden und die Ergebnisse von Verknüpfungen sind wiederum Aussagen. Folgende Abbildung zeigt die grundlegenden Verknüpfungen und deren unterschiedliche Schreibweisen je nach Anwendungsgebiet:

Verknüpfung	Name	gebräuchliche Schreibweisen
nicht a	Negation	$\neg a, \bar{a}$
a und b	Konjunktion	$a \wedge b, a \& b, a \cdot b, ab$
a oder b	Disjunktion	$a \vee b, a + b$
wenn a dann b	Implikation	$a \rightarrow b, a \Rightarrow b$
a genau dann wenn b	Äquivalenz	$a \leftrightarrow b, a \Leftrightarrow b$

Abbildung 2-4 Logische Operatoren [Ernst, 2016]

Dabei sind die Grundoperatoren „und“ bzw. „oder“ sogenannte binäre Operationen, die also 2 Operanden verlangen, der „nicht“ Operator jedoch ein unärer Operator, der sich nur auf einen Operanden bezieht.

Da es ja nur 2 Wahrheitswerte gibt, kann man die Operationen sehr einfach als Wahrheitstabeln oder **Wahrheitstabellen** angeben. Hier die Tabelle für die 3 Grundoperationen sowie die Äquivalenz:

a	b	$\neg a$	$a \cdot b$	$a + b$	$a \Leftrightarrow b$
0	0	1	0	0	1
0	1	1	0	1	0
1	0	0	0	1	0
1	1	0	1	1	1

Eine andere Darstellungsform mit T (True) und F (False) ist die folgende:

NOT	
F	T
T	F

AND	F	T
F	F	F
T	F	T

OR	F	T
F	F	T
T	T	T

XOR	F	T
F	F	T
T	T	F

Der „Nicht“ bzw. NOT Operator wird auch als **Negation** bzw. Komplement bezeichnet und bewirkt also eine Verneinung einer Aussage. Der logische „Und“ Operator wird auch **Konjunktion** genannt und ergibt nur dann 1 bzw. „wahr“ wenn beide Aussagen wahr sind. Das Ergebnis des „Oder“ Operators bzw. der **Disjunktion** ist „wahr“ bzw. 1 wenn die erste oder die zweite oder auch beide Aussagen wahr sind. Hier gibt es als Alternative auch den XOR Operator, bzw. das exklusive Oder, dass nur dann „wahr“ ergibt, wenn entweder die erste oder die zweite Aussage wahr ist, nicht jedoch wenn beide Aussagen wahr sind. XOR kann aber auch als Negation der Äquivalenz ausgedrückt werden, wenn man beide obigen Tabellen vergleicht, also $\neg(a \Leftrightarrow b)$.

Generell können alle theoretisch möglichen unären Operationen (2^1) und binären Operationen (2^4) durch Kombination der Grundoperatoren Negation, Konjunktion und Disjunktion dargestellt werden. Neben der Aussagenlogik ist eine mögliche Interpretation natürlich auch die Schaltalgebra, wo die Operationen Verknüpfungen von Schaltungen mit Eingängen und deren Ausgängen entsprechen.

Gerade die XOR Schaltung hat viele Anwendungen, von der Übertragsberechnung bei Binäraddierer bis hin zu kryptografischen Berechnungen und Prüfsummenbildung bei der Übertragung von Daten. Sollen zum Beispiel zwei Bitfolgen 1010 und 1110 übertragen werden, so kann mittels XOR

Verknüpfung die sogenannte **Parität** der beiden Bitfolgen berechnet werden. $1010 \text{ XOR } 1110 = 0100$. Diese muss mit den Daten mitübertragen werden und kann dann dazu verwendet werden, eine Bitfolge wiederherzustellen, falls diese verloren geht. So ist die zweite Bitfolge XOR-verknüpft mit der Parität, also $1110 \text{ XOR } 0100 = 1010$ wieder die rekonstruierte erste Bitfolge.

Hier geht es uns aber in erster Linie um den logischen Formalismus und die Abstraktion losgelöst von einer konkreten Implementierung oder Anwendung, da diese Formalismen in der Informatik von essentieller Bedeutung sind.

Gesetze der booleschen Algebra

Zum Abschluss dieses Kapitels können wir nun noch folgende Gesetze der booleschen Algebra bzw. Sätze der Aussagenlogik definieren, die man jeweils leicht mit Wahrheitstabellen auf Äquivalenz und damit Gültigkeit überprüfen kann:

1. Idempotenzgesetz: $a + a = a$
 $a \cdot a = a$
2. Kommutativgesetz: $a + b = b + a$
 $a \cdot b = b \cdot a$
3. Assoziativgesetz: $a + (b + c) = (a + b) + c$,
 $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
4. Absorptionsgesetz: $a + (a \cdot b) = a$
 $a \cdot (a + b) = a$
5. Distributivgesetz: $a + (b \cdot c) = (a + b) \cdot (a + c)$,
 $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
6. Involution: $\overline{\overline{a}} = a$
7. Komplementärgesetz: $a + \overline{a} = 1$
 $a \cdot \overline{a} = 0$
8. Identität: $a + 0 = a$
 $a \cdot 1 = a$
9. Extremalgesetz: $a + 1 = 1$
 $a \cdot 0 = 0$
10. De Morgansche Gesetze: $\overline{a + b} = \overline{a} \cdot \overline{b}$
 $\overline{a \cdot b} = \overline{a} + \overline{b}$
11. Dualitätsgesetz: $\overline{0} = 1$
 $\overline{1} = 0$

Die drei Gesetze Kommutativ-, Assoziativ- und Distributivgesetz sollten dabei bereits von den Grundrechnungsarten bekannt sein, gelten sie ja genauso (sogar mit der identen Schreibweise) bei der Addition und Multiplikation reeller Zahlen.

Sehr bekannt sind die **De Morganschen Gesetze**. Diese werden häufig in der Entwicklung digitaler Schaltkreise genutzt, um die Typen verwendeter logischer Schaltelemente gegeneinander auszutauschen oder Bauteile einzusparen. Eine negierte Konjunktion kann durch eine Disjunktion der jeweils negierten Operanden ausgedrückt werden und umgekehrt.

2.4 Zahlensysteme

Als nächstes wollen wir uns die in der Informatik wichtigen Zahlensysteme, deren Umrechnung untereinander und die interne Darstellung von negativen und reellen Zahlen im Rechner genauer ansehen.

Im Gegensatz zum römischen Zahlensystem etwa, ist unser heute gebräuchliches **dezimales Zahlensystem** (Zehnersystem) ein sogenannte **Stellenwertsystem**. Es wurde in Indien entwickelt und die Zahlendarstellung ist eng verbunden mit der Art, wie wir mit diesen Zahlen rechnen können. [Blieberger, 2005]

Eine Zahl im dezimalen Zahlensystem mit der Basis 10 und den Ziffern 0..9 kann wie folgt dargestellt werden:

$$z = a_n 10^n + a_{n-1} 10^{n-1} + \dots + a_2 10^2 + a_1 10^1 + a_0 10^0$$

So können wir z.B. Beispiel 305 als $3 \cdot 10^2 + 0 \cdot 10^1 + 5 \cdot 10^0$ darstellen.

Allgemein kann eine Zahl im Zahlensystem mit der Basis b dargestellt werden als:

$$z = (a_n a_{n-1} \dots a_2 a_1 a_0)_{(b)} = a_n b^n + a_{n-1} b^{n-1} + \dots + a_2 b^2 + a_1 b^1 + a_0 b^0$$

Die Basis b ist dabei >1 und die Ziffern sind im Intervall von 0 bis b-1. Die Ziffer hat also eine unterschiedliche Bedeutung abhängig von der Stelle in der Zahl, deswegen auch Stellenwertsystem.

In der Informatik sind neben dem Dezimalsystem wegen der internen Darstellung und Kodierung vor allem das **Binärsystem** (auch Dualsystem, Zweiersystem) mit der Basis 2 und wegen der kompakteren Darstellung das **Hexadezimalsystem** mit der Basis 16 in Verwendung. Zusätzlich verwendet wird auch noch das **Oktalsystem** mit der Basis 8.

Nachdem für ein Zahlensystem mit Basis b wie bereits erwähnt die Ziffern 0 bis b-1 benötigt werden, verwendet das Binärsystem daher die Ziffern 0 und 1 und das Oktalsystem die Ziffern 0 bis 7. Wie sieht das aber im Hexadezimalsystem aus? Hier werden 16 Ziffern benötigt, neben 0 bis 9 fehlen also noch 6 Zeichen. Hier „borgen“ wir uns einfach 6 Buchstaben aus und verwenden zusätzlich die „Ziffern“ A,B,C,D,E,F.


Folgende Tabelle zeigt die Zahlen von 0 bis 16 in den gängigen Zahlensystemen.

<i>Dezimal</i>	<i>Binär</i>	<i>Oktal</i>	<i>Hexadezimal</i>
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6

7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F


Umwandlung von Zahlen in verschiedene Zahlensysteme

Eine gängige Methode zur Umwandlung einer Dezimalzahl in eine Binärzahl ist die fortlaufende ganzzahlige Division durch die Basis 2 (auch Modulo Operation genannt). Die entstehenden Reste (nur 0 oder 1) bilden daraus die gesuchte Binärzahl. Diese Division ist solange durchzuführen, bis die Situation „1 DIV 2 = 0 Rest 1“ entsteht. Diese Rest-Eins stellt das höchstwertige Bit der gesuchten Binärzahl dar. Folgendes Beispiel verdeutlicht die Vorgehensweise:

43	DIV 2 =	21	Rest 1		Leserichtung von unten nach oben $43_{(10)} = 101011_{(2)}$
21	DIV 2 =	10	Rest 1		
10	DIV 2 =	5	Rest 0		
5	DIV 2 =	2	Rest 1		
2	DIV 2 =	1	Rest 0		
1	DIV 2 =	0	Rest 1		

Wichtig dabei ist die Leserichtung von unten nach oben, um die richtige Zahl im Binärsystem zu erhalten.

Diese Strategie kann genauso für die Umwandlung ins Hexadezimalsystem mit einer ganzzahligen Division durch die Basis 16 und ins Oktalsystem (ganzzahlige Division durch 8) verwendet werden, z.B.;

421	DIV 16 =	26	Rest 5		Leserichtung von unten nach oben $421_{(10)} = 1A5_{(16)}$
26	DIV 16 =	1	Rest 10 = A		
1	DIV 16 =	0	Rest 1		

In die andere Richtung, also vom Binärsystem ins Dezimalsystem reicht es, die Zweierpotenzen, die mit einer 1 in der Zahl gesetzt sind, zu addieren, wie wir in folgendem Beispiel sehr leicht erkennen können, da 0 mal irgendeine Zahl ja bekanntlich 0 ist:

Binärzahl	1 0 1 0 1 1
Zweierpotenzen	$2^5 2^4 2^3 2^2 2^1 2^0$
Dezimal	$2^5 + 2^3 + 2^1 + 2^0 = 32 + 8 + 2 + 1 = 43_{(10)}$

Somit haben wir für die Umwandlung von 43 erfolgreich die Probe durchgeführt.

Allgemein kann die Zahl mit der Basis b und $n+1$ Stellen wie folgt ins Dezimalsystem umgewandelt werden:

$$z = (a_n a_{n-1} \dots a_2 a_1 a_0)_{(b)} = a_n b^n + a_{n-1} b^{n-1} + \dots + a_2 b^2 + a_1 b^1 + a_0 b^0$$

Beispiel: $1A5_{(16)} = 1 \cdot 16^2 + 10 \cdot 16^1 + 5 \cdot 16^0 = 256 + 160 + 5 = 421_{(10)}$

Eine weitere elegante Möglichkeit ist das sogenannte **Hornerschema**.

Die Zahl $z = a_n b^n + a_{n-1} b^{n-1} + \dots + a_2 b^2 + a_1 b^1 + a_0 b^0$ kann durch Ausklammern der Basis b auch in folgende Schreibweise gebracht werden:

$$z = ((\dots (a_n b + a_{n-1})b + \dots + a_2)b + a_1)b + a_0$$

Das Hornerschema angewendet auf unsere beiden Beispiele wäre also:

$$101011_{(2)} = (((((1 \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 1 = 43_{(10)}$$

$$1A5_{(16)} = (1 \cdot 16 + 10) \cdot 16 + 5 = 421_{(10)}$$

Für die Umwandlung von Binärzahlen in Hexadezimal- oder Oktalzahlen müssen wir nicht den Umweg über das Dezimalsystem machen, sondern können direkt die Binärziffern gruppieren und die jeweiligen Gruppen z.B. mit der Tabelle 1 direkt umrechnen.

Aufgrund der Tatsache, dass $2^4 = 16$ ist, entspricht jede 4er Folge von Binärziffern einer Hexadezimalziffer, die getrennt betrachtet werden kann. Es reicht also, die Binärzahl in 4er Blöcke (mit führenden Nullen) aufzuteilen und diese einzeln umzuwandeln.

Beispiel:

$$110100101_{(2)} = \underbrace{0001}_{1} \underbrace{1010}_{10=A} \underbrace{0101}_{5} = 1A5_{(16)}$$

Bei der Umkehrung kann einfach jede Hexadezimalziffer einzeln in 4 Binärziffern nach Tabelle 1 umgewandelt werden. Aufgrund dieser einfachen Umrechnung zwischen Binär und Hexadezimalzahlen, und der viel platzsparenderen Darstellung (eine Ziffer für 4 Binärziffern) ist das Hexadezimalsystem in der Informatik sehr häufig in Tabellen (z.B. ASCII Code) als kompakte Notation zu finden.

Analog entspricht jede 3er Folge von Binärziffern einer Oktalziffer (da ja $2^3 = 8$ ist), so können also einfach Binärzahlen in 3er Blöcke mit führenden Nullen aufgeteilt und einzeln umgewandelt werden.

Beispiel:

$$110100101_{(2)} = \underbrace{110}_6 \underbrace{100}_4 \underbrace{101}_5 = 645_{(8)}$$

Rechnen im Binärsystem

Das Rechnen im Binärsystem unterscheidet sich grundsätzlich nicht vom Rechnen im Dezimalsystem, ist aber bedingt durch die geringe Anzahl der Ziffern etwas gewöhnungsbedürftig.

Bei der Addition z.B. können auch wieder die Zahlen untereinander geschrieben und Stelle für Stelle addiert werden, jedoch muss der Übertrag speziell betrachtet werden. Als Rechenregeln können wir hier folgendes zusammenfassen:

z1	z2	Summe	Übertrag (Carry)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Im Dezimalsystem ist ja bekanntlich $1 + 1 = 2_{(10)}$, die Ziffer 2 ist aber im Binärsystem nicht vorhanden, somit ist $1 + 1 = 10_{(2)}$ bzw. 0 und 1 Übertrag. Das Konzept ist aber auch hier wieder ident zum Dezimalsystem und tritt dort ja z.B. auch bei der Addition von $5 + 5 = 10_{(10)}$ bzw. 0 und 1 Übertrag auf.

Ein Beispiel für eine einfache Binäraddition sieht nun z.B. so aus:

$$\begin{array}{r}
 101010 \\
 + 1101111 \\
 \hline
 10011001
 \end{array}$$

Allgemein tritt ein Übertrag in einem Zahlensystem zur Basis b genau dann auf, wenn die Summe zweier Ziffern größer oder gleich der Basis b ist. [Blieberger, 2005]

Auch bei der Subtraktion muss man sich analog zum Dezimalsystem gegebenenfalls eine Ziffer von der höherwertigen (also weiter links stehenden) Stelle ausborgen, so ist z.B.

$$\begin{array}{r}
 10001 \\
 - 111 \\
 \hline
 1010
 \end{array}$$

da man sich bei $0 - 1$ stattdessen $10_{(2)} - 1_{(2)} = 1_{(2)}$ rechnet.

Was nun aber bei negativen Zahlen passiert, müssen wir uns im nächsten Kapitel genauer ansehen. Bei der Multiplikation hingegen gelten sehr einfache Rechenregeln:

z₁	z₂	Multiplikation
0	0	0
0	1	0
1	0	0
1	1	1

Bei genauer Beobachtung fällt hier vielleicht auf, dass dies genau der Wahrheitstafel der logischen Und Operation entspricht, die ja sehr oft auch mit dem Punkt Operator dargestellt wird.

Darstellung von negativen Zahlen

Nachdem wir nun die grundlegenden Rechenregeln im Binärsystem erläutert haben, stellt sich die Frage, wie wir negative Zahlen am besten darstellen können. Prinzipiell reicht von der Codierung her ein weiteres Bit (das ja bekanntlich 2 Zustände verwalten kann) aus, um das Vorzeichen abzuspeichern. Meist wird hierbei 0 für ein positives und 1 für ein negatives Vorzeichen kodiert. Negative Zahlen werden somit mit einem zusätzlichen Bit für das Vorzeichen und dem Betrag der Zahl codiert. Haben wir beispielsweise 8 Bit zur Verfügung, können wir damit ja 2^8 also 256 Zahlen darstellen. Bei positiven Zahlen wären das die Zahlen 0 bis 255. Wenn wir aber das erste Bit für das Vorzeichen verwenden, bleiben hier natürlich nur mehr 2^7 über. Wir könnten mit 8 Bit inkl. 1 Bit Vorzeichen nur mehr den Wertebereich von -127 bis +127 abdecken, oder bei 4 Bit von -7 bis +7, siehe folgende Tabelle:

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

Die Darstellungsform negativer Zahlen hat aber 2 gravierende Nachteile. Bei genauer Betrachtung fällt auf, dass es für die 0 zwei unterschiedliche Codierungen, nämlich einmal +0 (0000) und -0 (1000) gibt. Dies wirkt sich auch auf den Wertebereich der darstellbaren Zahlen aus. Mit 4 Bit können wir wie gesehen nur die Zahlen -7 bis +7, also 15 verschiedene Werte darstellen, statt der eigentlich möglichen 16 Kombinationen mit 4 Bits ($2^4 = 16$). Eine Zahl geht also durch die redundanten doppelte Kodierung der 0 verloren. Noch schwerwiegender ist aber der Nachteil, dass die im vorigen Kapitel beschriebenen Rechenregeln zur Addition und Subtraktion hier nicht mehr verwendet werden können, siehe folgendes Beispiel.

Wir wollen $-3 + 5 = 2_{(10)}$ im Binärsystem rechnen:

$$\begin{array}{r}
 1011 \\
 + \quad 0101 \\
 \hline
 10000 \text{ ???}
 \end{array}$$

Unsere Rechenregeln mit Übertrag führen anscheinend nicht zum gewünschten Ergebnis $10_{(2)}$.

Um diese beiden Probleme zu beseitigen, hat sich als gebräuchlichste Darstellung negativer Zahlen die sogenannte **Zweierkomplementdarstellung** durchgesetzt.

Mit 4 Bit kann folgende Kodierung verwendet werden, um den Wertebereich von -8 bis +7 abzudecken.

1000 = -8	1100 = -4	0000 = 0	0100 = +4
1001 = -7	1101 = -3	0001 = +1	0101 = +5
1010 = -6	1110 = -2	0010 = +2	0110 = +6
1011 = -5	1111 = -1	0011 = +3	0111 = +7

Vergleichen wir die Darstellung mit ersterer Variante, fällt folgendes auf. Nach wie vor kann mit Hilfe des ersten Bits das Vorzeichen unterschieden werden, positive Zahlen haben wieder eine 0, negative eine 1 an erster Stelle. Nun gibt es jedoch nur eine Darstellung der 0, nämlich 0000 und es können tatsächlich $2^4 = 16$ unterschiedliche Werte kodiert werden. Positive Zahlen werden wie bisher kodiert und nur mit einer führenden 0 versehen, danach werden aufsteigend die negativen Zahlen -8 bis -1 kodiert. -1 hat somit immer die Darstellung alle Bits auf 1 gesetzt.

Diese Form der Kodierung lässt sich auch auf der Zahlengerade wie folgt darstellen,

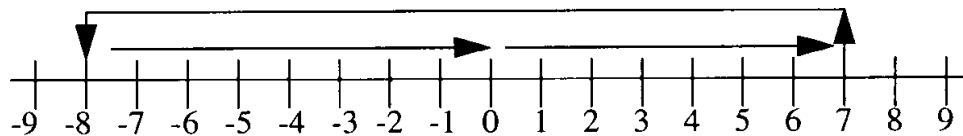


Abbildung 2-5 Zahlengerade

oder auch in Form eines Zahlenkreises.

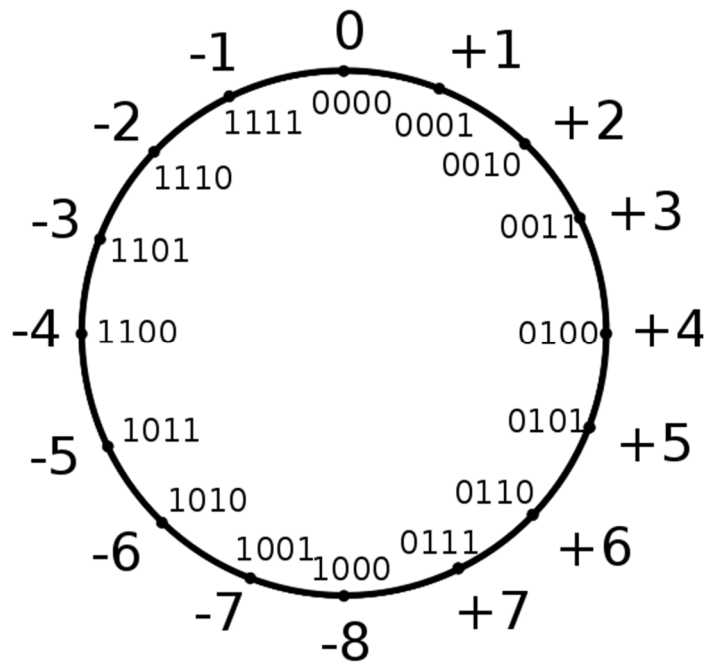


Abbildung 2-6 Zahlenkreis Zweierkomplement

Allgemein gilt: Mit N Bits kann der Bereich der Zahlen von -2^{N-1} bis $+2^{N-1}-1$ abgebildet werden. Die Ziffernfolge $b_nb_{n-1}...b_1b_0$ bezeichnet dabei die Zahl

$$z = b_n \cdot (-2^n) + b_{n-1} \cdot 2^{n-1} + \dots + b_1 \cdot 2^1 + b_0$$

z.B. $1101 = 1 \cdot (-2^3) + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 = -8 + 4 + 0 + 1 = -3$

Zur Berechnung des Zweierkomplement kann nun wie folgt vorgegangen werden, wobei hier sehr wichtig ist, eine vorgegebene Bitanzahl zu definieren. Zuerst bildet man das Komplement der Zahl (bitweise Negation) und danach addiert man 1 dazu.

Zum Beispiel ist die Darstellung von -5 mit 4 Bits 1011:

Zahl (+5)	0101
Negation	1010
1 addieren	+0001
Zer Komplement	1011

Mit 8 Bit Kodierung wäre -5 allerdings 11111011:

Zahl (+5)	00000101
Negation	11111010
1 addieren	+00000001
2er Komplement	11111011

Eine weitere Möglichkeit das Zweierkomplement zu bestimmen, ist die Ziffern der positiven Zahl von rechts nach links bis inklusive zur ersten 1 zu kopieren und die restlichen Ziffern zu negieren.

Mit dieser Zweierkomplementdarstellung können Addition und Subtraktion nun wie gewohnt durchgeführt werden, etwaige Überläufe über die maximale Bitanzahl können dabei ignoriert werden.

Die Subtraktion kann nun auch als Addition des Zweierkomplements dargestellt werden. So wäre $12 - 4 = 8_{(10)}$ mit 8 Bit nun wie folgt zu berechnen:

Zweierkomplementdarstellung von -4:

4	00000100
neg	11111011
+1	+00000001
2er	11111100


$$\begin{array}{r}
 00001100 \\
 + 11111100 \\
 \hline
 00001000
 \end{array}$$

Gleitkommadarstellung reeller Zahlen

Bis jetzt haben wir nur ganze Zahlen betrachtet, wie können reelle Zahlen mit Nachkommastellen ins Binärsystem umgewandelt werden?

Eine gängige Methode für den Nachkommateil ist die fortlaufende Multiplikation mit der Basis 2. Hierbei notiert man den ganzzahligen Teil (also die Vorkommastelle 0 oder 1) und spaltet im nächsten Schritt diese Stelle ab und rechnet solange weiter, bis die Multiplikation 1 ergibt oder die gewünschte Genauigkeit erreicht ist.

Als Beispiel betrachten wir die Umwandlung von $0,78125_{(10)}$ ins Binärsystem:

$0,78125 \cdot 2$	$= 1,5625$		1 abspalten
$0,5625 \cdot 2$	$= 1,125$		1 abspalten
$0,125 \cdot 2$	$= 0,25$		0 abspalten
$0,25 \cdot 2$	$= 0,5$		0 abspalten
$0,5 \cdot 2$	$= 1,0$		1 abspalten -> Ende, da 1 erreicht.

Für das Ergebnis lesen wir nun die binären Nachkommastellen von oben nach unten ab, somit ist $0,78125_{(10)} = 0,11001_{(2)}$


Umgekehrt können wir die Binärzahl mittels

$$z = 0.a_1a_2\dots a_n_{(2)} = a_12^{-1} + a_22^{-2} + \dots + a_n2^{-n}_{(10)}$$

wieder ins Dezimalsystem umrechnen, die Probe zu oberem Beispiel lautet also:

$$0,11001_{(2)} = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} = \frac{1}{2} + \frac{1}{4} + \frac{1}{32} = 0,5 + 0,25 + 0,03125 = 0,78125_{(10)}$$

Zu beachten ist allerdings, dass eine exakte Umwandlung von Dezimalbrüchen in Binärbrüche nicht immer möglich ist. Viele gebrochene Zahlen, die sich im Dezimalsystem exakt darstellen lassen, werden im Binärsystem periodisch unendlich lang, wie das Beispiel von $0,1_{(10)}$ zeigt:

$0,1 \cdot 2$	$= 0,2$		0 abspalten
$0,2 \cdot 2$	$= 0,4$		0 abspalten
$0,4 \cdot 2$	$= 0,8$		0 abspalten
$0,8 \cdot 2$	$= 1,6$		1 abspalten
$0,6 \cdot 2$	$= 1,2$		1 abspalten -> ab hier periodisch

$0,1_{(10)}$ kann also nicht exakt im Binärsystem dargestellt werden, sondern muss als $0,0\ 0011\ 0011\ 0011\dots_{(2)}$ genähert werden.

Diese Erkenntnis hat natürlich massive Konsequenzen auf die gesamte Informatik. Es entstehen bereits ohne etwas zu rechnen, Rundungsfehler nur durch die interne binäre Darstellung von Zahlen, da ja nur eine begrenzte Anzahl an Nachkommastellen zur Speicherung zur Verfügung stehen!

Bei reellen Zahlen können nun mit den bisher vorgestellten Methoden der ganzzahlige Anteil und die Nachkommastellen getrennt voneinander umgewandelt werden.

So ist z.B. $5,75_{(10)} = 101,11_{(2)}$

da $5_{(10)} = 101_{(2)}$ und $0,75 = \frac{1}{2} + \frac{1}{4} = 0,11_{(2)}$ ist.

Wie soll eine binäre Zahl mit Nachkommastellen aber nun intern abgespeichert werden? Eine Möglichkeit wäre die sogenannte **Festkommadarstellung** (fixed point representation), wo neben dem VZ Bit eine fixe Bitanzahl für die Stellen vor dem Komma und eine fixe Bitanzahl für die Stellen hinter dem Komma reserviert werden.

Da aufgrund von begrenzter Speicheranzahl natürlich nur ein sehr geringer Teil der reellen Zahlen abbildbar ist, viele Anwendungen aber sehr unterschiedlich große Zahlen repräsentieren müssen, z.B. sehr große (10^{14}) oder sehr kleine (10^{-29}) Zahlen oder Mischformen ($103,4 \cdot 10^{17}$, $1,45 \cdot 10^{-29}$), ist es vorteilhaft, die Position des Kommas separat zu speichern, was zur **Gleitkommadarstellung** (floating point representation) führt.

Im Dezimalsystem kann z.B. die Zahl 0,000456 in Exponentialschreibweise als $456 \cdot 10^{-6}$ dargestellt werden. 456 wird hierbei als **Mantisse** bezeichnet, und -6 als **Exponent**.

Allgemein kann die Zahl $z_{(b)}$ als $m \cdot b^E$ dargestellt werden, wobei b die Basis, m die Mantisse und E den Exponenten bezeichnet. Die Zahl 0,000456 könnte aber genauso gut als $4,56 \cdot 10^{-4}$ oder als $0,0456 \cdot 10^{-8}$ dargestellt werden.

Um diese Mehrdeutigkeit zu vermeiden, verwendet man normalerweise die normalisierte Darstellung einer Gleitkommazahl und schreibt die Mantisse m so, dass genau eine Stelle vor dem Dezimalpunkt eine von Null verschiedene Ziffer ist. [Ernst, 2016]

Eine **normalisierte Gleitkommazahl** zur Basis 2 kann nun wie folgt definiert werden
 $\pm 1.m_1m_2\dots m_n \cdot 2^E$

Dies hat den Vorteil, dass die 1 vor dem Komma nicht abgespeichert werden muss, da die Zahl ja immer in dieser Form dargestellt wird. Abgespeichert werden ein Vorzeichen Bit (VZ), danach eine gewisse Anzahl an Bits für den Exponenten (E) und abschließend die Mantissenbits (m).

Betrachten wir als Beispiel wieder die Zahl $5,75_{(10)}$, die wir bereits als $101,11_{(2)}$ ins Binärsystem umgerechnet haben. Für die Darstellung als Gleitkommazahl sollen 1 Bit für das VZ, 3 Bit für den Exponenten und 8 Bit für die Mantisse verwendet werden, dies ist nur ein simples Beispiel, die in der Praxis verwendeten Datentypen haben natürlich eine höhere Genauigkeit, siehe unten.

Zuerst muss die Zahl normalisiert werden. Aus $101,11_{(2)}$ wird gemäß obiger Definition $1,0111 \cdot 2^2$. Das Komma muss also um 2 Stellen nach links verschoben werden, was im Binärsystem einer Multiplikation mit $2^2 = 4$ gleichkommt. Nun können wir die Kodierung der Gleitkommazahl angeben. Da es sich um eine positive Zahl handelt, ist das VZ Bit 0, der Exponent 2 muss natürlich noch ins Binärsystem umgewandelt werden und entspricht in 3 Bit Kodierung 010, die Mantisse in 8 Bit Darstellung entspricht 01110000. Achtung, hier werden die Nullen hinterm Komma natürlich rechts aufgefüllt!

Somit ist die Darstellung wie folgt:

VZ	Exponent	Mantisse
0	010	01110000

Die auch in der Programmiersprache C verwendeten Datentypen **float** (einfache Genauigkeit) und **double** (doppelte Genauigkeit) sind nach dem **IEEE Standard 754** (Institute of Electrical and Electronics Engineers) wie folgt definiert:

Länge	VZ Bits	Exponent Bits	Mantisse Bits
float 32 Bit	1	8	23
double 64 Bit	1	11	52

Der Exponent wird hierbei allerdings nicht mit Vorzeichenbit bzw. Zweierkomplementdarstellung abgespeichert, stattdessen wird mit einer Verschiebung (einem sogenannten bias) gearbeitet, um negative Exponenten zu vermeiden.

Bei einfacher Genauigkeit (float) stehen laut obiger Tabelle 8 Bit für den Exponenten zur Verfügung, damit ist ein Wertebereich von 2^8 also 256 verschiedene Werte möglich. Exponenten von -127 bis +128 werden mit einem Bias von +127 verschoben, um einen positiven Wert zwischen 0 und 255 kodieren zu können. Somit kann der tatsächliche Exponent mittels

$E = e - \text{bias}$

berechnet werden, wenn e der verschobene Exponent ist.

Bei doppelter Genauigkeit (double) und 11 Bit für den Exponenten wird 1023 als Bias verwendet und der Wertebereich der Exponenten E ergibt sich von -1023 bis +1024. Allgemein kann der Bias bei N Bit für den Exponenten mit $2^{N-1}-1$ berechnet werden.

Betrachten wir nochmal das simple Beispiel von oben mit 3 Bits für den Exponenten. Es wäre in diesem Fall der Bias 2^2-1 also 3. Dies würde bedeuten, dass der Exponent 2 nicht als 010 sondern addiert mit dem Bias als $2+3 = 5 = 101$ gespeichert wäre.

Gleitkommaarithmetik und Genauigkeit

Wie bereits erwähnt, treten aufgrund von limitierter Stellenanzahl bei der Speicherung von Zahlen in der Informatik Rundungsfehler auf. Selbst ohne zu rechnen, lassen sich Dezimalbrüche wie 0.1 im Binärsystem nicht exakt darstellen. Weiters ist klar, dass nur ein sehr begrenzter Wertebereich der reellen Zahlen überhaupt darstellbar ist und unendlich große Lücken zwischen zwei darstellbaren Zahlen in Kauf genommen werden müssen.

Wie sieht es aber mit grundlegenden Gesetzen der Algebra aus? Das **Kommutativgesetz** $a+b = b+a$ bleibt bei der Addition und Multiplikation von Gleitkommazahlen erhalten, anders sieht es jedoch beim **Assoziativ-** und beim **Distributivgesetz** aus.

Als Beispiel zum besseren Verständnis verwenden wir hierfür statt dem Binärsystem eine dezimale Arithmetik mit 7 Stellen:

$$1234.567 + 45.67844 = 1280.245$$

$$1280.245 + 0.0004 = 1280.245$$

$$\text{Aber } 45.67844 + 0.0004 = 45.67884$$

$$45.67884 + 1234.567 = 1280.246$$

$$1234.567 \times 3.333333 = 4115.223$$

$$1.234567 \times 3.333333 = 4.115223$$

$$4115.223 + 4.115223 = 4119.338$$

$$\text{Aber } 1234.567 + 1.234567 = 1235.802$$

$$1235.802 \times 3.333333 = 4119.340$$

Außerdem treten öfter Effekte der **Auslöschung** (Cancellation) auf, wobei die Subtraktion von beinahe gleich großen Operanden hohen Genauigkeitsverlust bewirken kann.

Rundungsfehler entstehen z.B. auch bei der Umwandlung von (63.0/9.0) in Integer, die 7 ergibt, aber die Umwandlung von (0.63/0.09) kann 6 ergeben.

Dadurch sind in Programmen der Test auf Division durch 0 sowie allgemein Gleichheitstests problematisch! Anstatt eines exakten Vergleichs, der durch Rundungsfehler dann oft fehlschlagen könnte, ist es hier ratsam, zu überprüfen, ob die Differenz der beiden Werte kleiner oder gleich ein definiertes Epsilon ist. Generell gilt natürlich, für höhere Genauigkeit double statt float verwenden!

Nachdem wir nun die wichtigsten Aspekte der Zahlendarstellung in der Informatik für ganze und reelle Zahlen gehört haben, sei zum Abschluss dieses Kapitel und zur Auflockerung auf einen klassischen Informatikwitz verwiesen.

"There are only 10 types of people in the world: Those who understand binary and those who don't."

Wem die Pointe einstweilen noch verschlossen bleibt, dem sei empfohlen, das aktuelle Kapitel noch einmal zu studieren...

2.5 Informationstheorie

Zum Abschluss des Grundlagenkapitels soll noch ein kurzer Überblick über die Informationstheorie und als Beispiel ein Verfahren für eine effiziente Kodierung mit möglichst wenig Redundanz in der Datenübertragung gegeben werden.

Folgende Abbildung zeigt den schematischen Ablauf einer Übertragung einer Nachricht von einem Sender (Quelle) über einen Kanal zu einem Empfänger (Senke). Meist wird jedoch nicht die Originalnachricht übertragen, sondern diese mittels Codierer in ein geeignetes Format für die Übertragung umgewandelt. Der Decodierer auf der Empfängerseite muss die Nachricht dann wieder in eine leserliche Form bringen, natürlich sind bei der Übertragung auch diverse Störsignale möglich.

Allgemein kann eine Nachricht wie folgt definiert werden:

„Eine Nachricht ist eine aus den Zeichen eines Alphabets gebildete Zeichenfolge. Diese Zeichenfolge muss nicht endlich sein, aber abzählbar (d. h. man muss die einzelnen Zeichen durch Abbildung auf die natürlichen Zahlen durchnummerieren können), damit die Identifizierbarkeit der Zeichen sichergestellt ist.“ [Ernst, 2016]

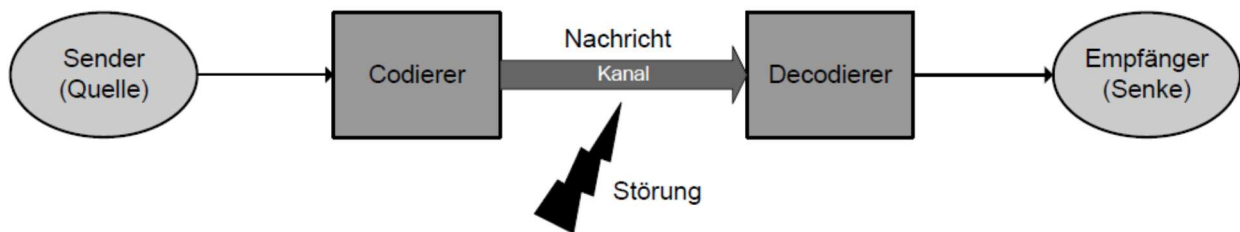


Abbildung 2-7: Nachrichtenübertragung [Ernst, 2016]

Shannon Entropie

Gegründet von Shannon 1948 in seinem Artikel "A Mathematical Theory of Communication" wird versucht, ein **Maß für Informationsgehalt** bei der Übertragung einer Nachricht von einem Sender zu einem Empfänger, zu finden. Die Nachricht besteht aus einzelnen Zeichen, die eine bestimmte Auftrittswahrscheinlichkeit aufweisen. Die zu ermittelnde Kenngröße **mittlerer Informationsgehalt H** einer Nachricht wird auch **information entropy** (bzw. Shannon entropy) genannt und in bits angegeben. Daraus kann dann die durchschnittliche Anzahl Bits, die zur Speicherung oder Kommunikation der einzelnen Zeichen der Nachricht benötigt wird, abgeleitet werden.

Der Begriff Entropie beschreibt in diesem Zusammenhang die Unsicherheit der Messung einer Zufallsvariable, z.B. hat ein Münzwurf weniger Entropie als das Würfeln eines 6-seitigen Würfels, da ja bei der Münze das Ereignis Kopf oder das Ereignis Zahl mit jeweils $p=0.5$ Auftrittswahrscheinlichkeit stattfindet, beim Würfel jede der 6 Zahlen jedoch Auftrittswahrscheinlichkeit $p=1/6$ aufweist.

Der Shannon Informationsgehalt eines Zeichens x messbar in bits wird nun wie folgt definiert [Blieberger, 2005]:

$$h = \text{Id}(1/p) = -\text{Id } p$$

wobei p die Auftrittswahrscheinlichkeit von x und mit Id der Logarithmus zur Basis 2 (logarithmus dualis, also die Umkehrfunktion zur Potenzfunktion $y=2^x$) bezeichnet. Da ja die Wahrscheinlichkeit p

kleiner oder gleich 1 sein muss, ist trotz negativem Vorzeichen der Informationsgehalt natürlich positiv, wenn man die Logarithmusfunktion betrachtet:

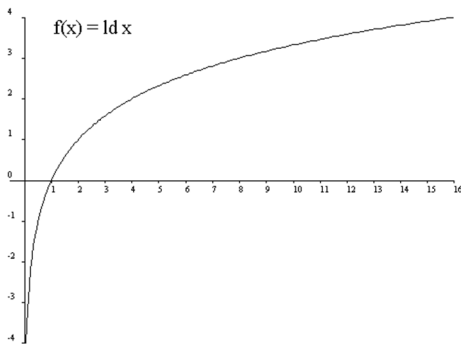


Abbildung 2-8 Logarithmusfunktion zur Basis 2

Weiters kann man sehr leicht die Äquivalenz der beiden Formeln mit den Rechenregeln für Logarithmusfunktionen herleiten, da $\lg(1/p) = \lg 1 - \lg p$ gilt und $\lg 1$ gleich 0 ist.

Beim Münzwurf ergibt sich nun also

$$h = \lg(1/0.5) = \lg 2 = 1 \text{ bit}$$

Zufällig gewählte Zeichen aus dem Standard ASCII Code (7 Bit Kodierung) haben eine Entropie von genau 7 bits pro Zeichen, da ja $p = 1/2^7 = 1/128$ ist und damit $h = \lg 128 = 7$ ist.

In der Praxis werden allerdings selten zufällige ASCII Texte übertragen, einige Zeichen in der deutschen bzw. englischen Sprache sind normalerweise häufiger als andere (z.B. Vokale), damit ist die Unsicherheit und damit auch die Entropie natürlich niedriger. So liegt die Entropie von englischem Text nach Erfahrungswerten zwischen 1.0 und 1.5 bits pro Zeichen.

Als Extrembeispiel könnten wir eine lange Nachricht mit wiederholt demselben Zeichen übertragen, dann wäre die Auftrittswahrscheinlichkeit $p=1$ und damit $h = \lg(1/1) = 0$, also hätte diese Nachricht Entropie 0 und wäre komplett vorhersagbar.

Die Entropie einer zufälligen n-stelligen Dezimalzahl ist demnach $h=n*\lg 10$, da eine Ziffer Auftrittswahrscheinlichkeit $p=1/10$ hat.

Der Mittelwert der Informationsgehalte aller Zeichen des zugrundeliegenden Alphabets Σ , also der Erwartungswert wird nun als **mittlerer Informationsgehalt H** bzw. Shannon Entropie wie folgt definiert [Blieberger, 2005]:

$$H = \sum_i p_i h_i$$

Betrachten wir dazu ein einfaches Beispiel mit dem gegebenen Alphabet $Z=\{a,b,c\}$ samt Auftrittswahrscheinlichkeiten: $p(a)=0.5$, $p(b)=0.25$, $p(c)=0.25$

Daher gilt:

$$h(a)=\lg 2=1$$

$$h(b)=\lg 4=2$$

$$h(c)=\lg 4=2$$

$$\text{Und damit: } H = 0.5*1+0.25*2+0.25*2 = 1.5 \text{ bit}$$

Was bedeutet das Ergebnis nun für eine effiziente Kodierung?

Ein optimaler Code für dieses Beispiel wäre ein Code mit variabler Länge mit $a=1$, $b=01$, $c=00$.

Diese Kodierung ist umkehrbar eindeutig, so kann z.B. aus dem Datenstrom 00110100011 eindeutig die zugrunde liegende Nachricht caabcba rekonstruiert werden.

Eine klassische Kodierung fixer Länge, die die Auftrittswahrscheinlichkeit nicht berücksichtigt, würde z.B. a=00, b=01, c=10 kodieren, somit wäre die Nachricht caabcba hierbei 10000001100100

Die unterschiedliche Nachrichtenlänge ist deutlich erkennbar. Wie können diese Erkenntnisse nun in einen effizienten Algorithmus zur optimalen Codegenerierung benutzt werden? Das nächste Kapitel gibt hier die Antwort und zeigt, wie mit einem sehr einfachen Methodik sogar ohne die Entropieformeln verwenden zu müssen, ein besonders redundanzarmer Code generiert werden kann.

Huffman Codierung

Ein Code der besonders redundanzfrei ist wird nach D. A. Huffman benannt. Beschränkt man sich auf die Codierung von einzelnen Zeichen ist dieses Verfahren optimal. [Blieberger, 2005]

Beim Verfahren zur Erzeugung einer Huffman-Codes wird schrittweise ein Codebaum aufgebaut. Man fasst dazu die beiden Zeichen mit der geringsten Auftrittswahrscheinlichkeit zusammen und behandelt sie im weiteren Verlauf wie ein einzelnes Zeichen, dessen Auftrittswahrscheinlichkeit gleich der Summe der beiden Einzelwahrscheinlichkeiten ist. Dieses Verfahren wird solange fortgesetzt bis der Codebaum fertig aufgebaut ist. Diese Vorgehensweise wird am besten an einem konkreten Beispiel verdeutlicht:

Wir entwickeln nun einen Huffman-Code für ein Alphabet Σ , das aus folgenden Zeichen besteht, modifiziert nach [Blieberger, 2005]:

$\Sigma = \{a, b, c, d, e\}$ mit den Auftrittswahrscheinlichkeiten p :

Σ	P	h	Code	l	$p \cdot l$	$p \cdot h$
A	0.40	1.32	0001	4	1.60	0.52
B	0.20	2.32	0010	4	0.80	0.46
C	0.18	2.47	0100	4	0.72	0.44
D	0.11	3.18	1000	4	0.44	0.35
E	0.11	3.18	0101	4	0.44	0.35

Folgende Definitionen werden in der Tabelle verwendet:

- p ist die Auftrittswahrscheinlichkeit eines Zeichens
- l ist die Länge eines Zeichens in Bit
- h ist der Informationsgehalt eines Zeichens in bit.
- $h = \log_2(1/p) = -\log_2 p$, wobei \log_2 der logarithmus dualis (der Logarithmus zur Basis 2) ist.
- Die mittlere Wortlänge ist die Summe der mit der Auftrittswahrscheinlichkeit gewichteten Wortlängen $L = \sum p_i l_i$ in Bit.

- Analog der mittlere Informationsgehalt $H = \sum p_i h_i = \sum p_i \lg(1 / p_i) = - \sum p_i \lg p_i$ in bit.
- Die Redundanz ist: $R = L - H$ mit $R \geq 0$ in Bit.
- Die relative Redundanz: $r = R / L$

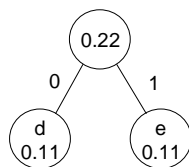
In diesem Beispiel mit einer Standardcodierung von 0000 bis 0101 unabhängig von den Auftretswahrscheinlichkeiten gilt daher:

- Die mittlere Wortlänge $L = 4$ Bit
- Der mittlere Informationsgehalt $H = 2.12$ bit
- Die Redundanz $R = 1.88$ Bit
- Die relative Redundanz $r = 0.47$

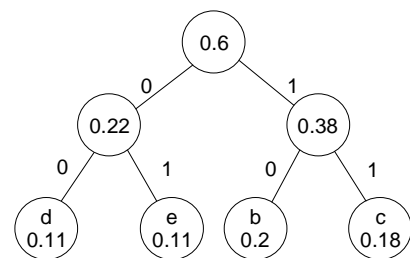
Die Huffman Codierung kann nun hoffentlich die mittlere Wortlänge und die Redundanz reduzieren.

Die Vorgehensweise ist in folgenden Tabellen und Grafiken zusammengefasst:

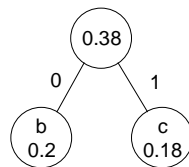
Σ	p
a	0.40
de	0.22
b	0.20
c	0.18



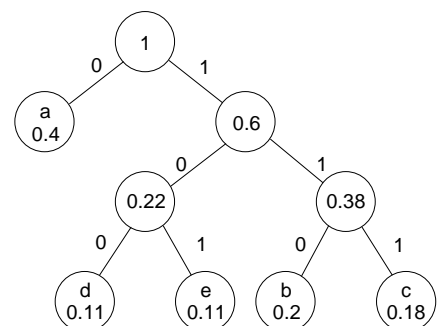
Σ	P
bcde	0.60
a	0.40



Σ	P
a	0.40
bc	0.38
de	0.22



Σ	P
abcd	1.00
e	



Σ	p	h	Code	L	p l	p h
A	0.40	1.32	0	1	0.40	0.52
B	0.20	2.32	110	3	0.60	0.46
C	0.18	2.47	111	3	0.54	0.44
D	0.11	3.18	100	3	0.33	0.35
E	0.11	3.18	101	3	0.33	0.35

Nun gilt also:

- Die mittlere Wortlänge $L = 2.2$ Bit
- Der mittlere Informationsgehalt $H = 2.12$ bit
- Die Redundanz $R = 0.08$ Bit

- Die relative Redundanz $r = 0.0364$

Zu beachten ist im Vergleich zu vorhin die Abnahme der Redundanz!

Zum Abschluss dieses Kapitels soll ein wesentlicher Nachteil dieses Verfahrens nicht unerwähnt bleiben. Der Huffman-Code setzt natürlich voraus, dass alle Auftretswahrscheinlichkeiten der verwendeten Zeichen des Alphabets bereits bekannt sind. Nur so kann ein optimaler Code generiert werden. Oft ist diese Information in der Praxis natürlich vor dem Nachrichtenaustausch zwischen beliebigen Sendern und Empfängern natürlich noch nicht verfügbar. Eine mögliche Alternative ist es, sogenannte **adaptive Huffman-Codes** zu verwenden. Sowohl Sender als auch Empfänger zählen die relativen Häufigkeiten der übertragenen Zeichen und adaptieren den Code periodisch, wenn sich dadurch die Auftretswahrscheinlichkeiten ändern. Somit wird der Code dynamisch an die konkreten Daten angepasst.

2.6 Selbstlernkontrolle

Fragen

Wieviele Bit werden benötigt, um alle Kleinbuchstaben als Bitstrings von Nullen und Einsen zu kodieren?

- (a) 4
- (b) 5
- (c) 6
- (d) 7

Welche der folgenden Aussagen sind wahr?

- (a) Mit dem ASCII Code können alle Symbole der weltweit verwendeten Sprachen codiert werden.
- (b) Sind die Auftretenswahrscheinlichkeiten der Symbole bekannt, liefert der Huffman Code die optimale Kodierung.
- (c) Die ersten 128 Zeichen sind in der UTF-8 Kodierung ident mit dem ASCII Code.
- (d) Die Entropie ist ein Maß für die Ordnung. Je höher die Entropie, umso höher ist die Ordnung.

Welcher Wertebereich von Zahlen kann in der Zweierkomplement Darstellung mit 7 Bit dargestellt werden?

- (a) Natürliche Zahlen von 0 bis 127
- (b) Ganze Zahlen von -128 bis 127
- (c) Ganze Zahlen von -127 bis 128
- (d) Ganze Zahlen von -127 bis 127

Welche der folgenden Aussagen sind wahr?

- (a) $0.1_{(10)}$ ist im Binärsystem nur als periodischer Bruch darstellbar.
- (b) Alle reellen Zahlen sind mit double Genauigkeit als Gleitkommazahlen darstellbar.
- (c) Gleitkommaarithmetik ist nicht assoziativ und distributiv.
- (d) Binärzahlen können in Gruppen zu 3 Bits einzeln ins Hexadezimalsystem umgewandelt werden.

In der booleschen Algebra gilt:

- (a) $a + a = a$
- (b) $a \cdot 0 = a$
- (c) $a + b = b + a$
- (d) $\overline{a \cdot b} = \overline{a} \cdot \overline{b}$

Lösungen

Wieviele Bit werden benötigt, um alle Kleinbuchstaben als Bitstrings von Nullen und Einsen zu kodieren?

- (a) 4
- (b) 5**
- (c) 6
- (d) 7

Welche der folgenden Aussagen sind wahr?

- (a) Mit dem ASCII Code können alle Symbole der weltweit verwendeten Sprachen codiert werden.
- (b) Sind die Auftrittswahrscheinlichkeiten der Symbole bekannt, liefert der Huffman Code die optimale Kodierung.**
- (c) Die ersten 128 Zeichen sind in der UTF-8 Kodierung ident mit dem ASCII Code.**
- (d) Die Entropie ist ein Maß für die Ordnung. Je höher die Entropie, umso höher ist die Ordnung.

Welcher Wertebereich von Zahlen kann in der Zweierkomplement Darstellung mit 7 Bit dargestellt werden?

- (a) Natürliche Zahlen von 0 bis 127
- (b) Ganze Zahlen von -128 bis 127**
- (c) Ganze Zahlen von -127 bis 128
- (d) Ganze Zahlen von -127 bis 127

Welche der folgenden Aussagen sind wahr?

- (a) $0.1_{(10)}$ ist im Binärsystem nur als periodischer Bruch darstellbar.**
- (b) Alle reellen Zahlen sind mit double Genauigkeit als Gleitkommazahlen darstellbar.
- (c) Gleitkommaarithmetik ist nicht assoziativ und distributiv.**
- (d) Binärzahlen können in Gruppen zu 3 Bits einzeln ins Hexadezimalsystem umgewandelt werden.

In der booleschen Algebra gilt:

- (a) $a + a = a$**
- (b) $a \cdot 0 = a$
- (c) $a + b = b + a$**
- (d) $\overline{a \cdot b} = \overline{a} \cdot \overline{b}$

2.7 Übungsteil

Übung 1: Boolesche Algebra

Zeigen Sie die Gültigkeit der deMorgan Gesetze mittels Wahrheitstabellen.

Übung 2: Zahlensysteme

Wandeln Sie 4137 ins Binär und Hexadezimalsystem um. Welche Reihenfolge der Berechnung ist am effizientesten, begründen Sie ihre Antwort. Geben Sie alle Zwischenschritte der Berechnung an.

Übung 3: 2 Komplement

Berechnen Sie die 2er Komplement Darstellung von -53 mit 8 Bit als Kodierung. Welcher Wertebereich ist mit 8 Bit kodierbar?

Übung 4: Gleitpunktdarstellung

Berechnen Sie die normalisierte Gleitpunktdarstellung von -19.125. Verwenden Sie ein Bit für das Vorzeichen, 3 für den Exponenten und 12 für die Mantisse.

Übung 5: Huffman Kodierung

Erzeugen Sie eine Huffman Kodierung für folgendes Alphabet. Skizzieren Sie auch den Kodierungsbaum.

Symbol	a	b	c	d	e
Probability	0.3	0.1	0.15	0.2	0.25

3 Einführung in die Automatentheorie

Nachdem wir im vorigen Kapitel die wichtigsten Grundlagen der Informationsverarbeitung gezeigt haben, soll nun in Kapitel 3 der Fokus auf endliche Automaten als Basis für die Entwicklung abstrakter Rechenmodelle und des Algorithmusbegriffes gelegt werden.

3.1 Definition und informelle Einführung

Automat

Um den Begriff Automatentheorie zu definieren, müssen wir zuerst die Definition des Begriffs Automat angeben. Sucht man im Duden nach dem Wort Automat, erfährt man, dass das Wort aus dem griechischen Wort *autómatos* abstammt, das „sich selbst bewegend“ oder „aus eigenem Antrieb“ bedeutet. Daraus können wir leicht Beispiele für Automaten aus dem alltäglichen Leben ableiten.

- ♦ ein Apparat, der nach Münzeinwurf selbsttätig Waren abgibt oder eine Dienst- oder Bearbeitungsleistung erbringt wie z.B. ein Getränkeautomat
- ♦ eine Werkzeugmaschine, die Arbeitsvorgänge nach einem vorgegebenen Programm selbsttätig ausführt
- ♦ ein elektronisch gesteuertes System, das Information an einem Eingang aufnimmt, selbstständig verarbeitet und an einem Ausgang abgibt

Wir werden uns im Weiteren speziell mit endlichen Automaten beschäftigen. Als informelle Einführung genügt anfangs einmal folgende Definition:

Ein **endlicher Automat** ist ein System, das sich zu jedem gegebenen Zeitpunkt in einem von einer endlichen Anzahl von Zuständen befindet. Ein Zustand kann als Gedächtnis des Automaten gesehen werden, das die relevante Information und die Änderungen der Eingabe seit dem Systemstart des Automaten widerspiegelt.

Beispiel: Ein Kippschalter als endlicher Automat

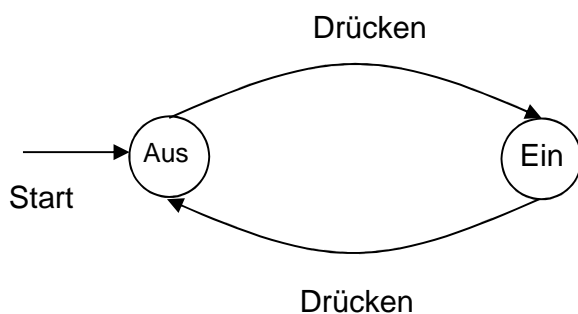


Abbildung 3-1 Kippschalter als endlicher Automat

Ein Kippschalter kann als endlicher Automat mit 2 Zuständen „Ein“ und „Aus“ modelliert werden. Der Benutzer kann einen Schalter drücken, der abhängig vom Zustand des Kippschalters eine unterschiedliche Wirkung hat. Das Gerät selbst weiß, wann es sich im Zustand „Ein“ oder „Aus“ befindet. Diese zwei Zustände bilden also das Gedächtnis des Automaten. Beim Systemstart befindet sich der Kippschalter im Zustand „Aus“. Drücken des Schalters bewirkt nun einen so

genannten Zustandsübergang in den Zustand „Ein“. Wird der Schalter nochmals gedrückt, wechselt der Automat wieder in den Zustand „Aus“. In Abbildung 3-1 ist eine graphische Darstellung des endlichen Automaten zu sehen. Die beiden Zustände „Ein“ und „Aus“ werden durch Kreise symbolisiert. Die Pfeile zeigen Zustandsübergänge an, die durch die Eingabe „Drücken“ ausgelöst werden. Der Zustand „Aus“ wird mit dem eingehenden Pfeil als Startzustand gekennzeichnet. Genauer zur graphischen Beschreibung von endlichen Automaten folgt in Kapitel 3.4.

Automatentheorie

Mit dem Begriff Automatentheorie ist nun die Theorie, die sich mit mathematischen Modellen von Automaten befasst, gemeint. Die Automatentheorie ist heute ein Teilgebiet der theoretischen Informatik, geht aber bis in den Anfang des 20. Jahrhunderts zurück, als es noch keine Computer gab. In den 30-er Jahren des vorigen Jahrhunderts beschäftigte sich ein gewisser A. Turing mit abstrakten Rechenmaschinen, den nach ihm benannten Turing Maschinen, die von der Rechenleistung her über sämtliche Fähigkeiten heutiger Computer verfügen. Es ging hauptsächlich um die Frage, was durch eine Rechenmaschine berechnet werden kann und was nicht. Heute werden diese Fragestellungen nach wie vor in der Berechenbarkeitstheorie und Komplexitätstheorie behandelt. Wir werden uns in Kapitel 4.5 noch genauer mit dieser Thematik beschäftigen. Momentan interessiert aber sicherlich viel mehr die Frage, warum sich ein Informatiker oder gerade ein Wirtschaftsinformatiker heutzutage noch mit solchen abstrakten Modellen beschäftigen sollte?

Lernergebnisse

Nach diesem Kapitel sind Studierenden in der Lage,

- Die Funktionsweise von endlichen Automaten zu verstehen und die erzeugte Sprache zu definieren.
- einfache abstrakte Aufgabenstellungen der formalen Sprachen als endliche Automaten zu modellieren.

3.2 Motivation

Automaten sind ein wichtiges Modell für viele Bereiche der Hard- und Software. Im Folgenden werden Beispiele aus allen Bereichen der Informatik gegeben, hier kann schon die Vielfaltigkeit dieser Modelle erkannt werden, nach [Hopcroft, 2002].

Hardware und Betriebssysteme

- ◆ Modellierung eines Rechnersystems (Schaltwerk) als Automat
- ◆ Software zum Entwurf und zur Überprüfung des Verhaltens digitaler Schaltkreise
- ◆ Modellierung von Prozesszuständen in modernen Multiprocessing Betriebssystemen
- ◆ Software zum Verifizieren aller Arten von Systemen, die eine endliche Anzahl verschiedener Zustände haben, wie z.B. Kommunikationsprotokolle (z.B. TCP/IP Protokoll)

Pattern Matching

- ◆ Textsuche/Ersetzung in Editoren, Textverarbeitungssystemen
- ◆ Suchmaschinen im Internet
- ◆ Die lexikalische Analysekomponente von Compilern. D.h. die Compilerkomponente, die den Eingabetext in logische Einheiten aufschlüsselt, wie z.B. Bezeichner, Schlüsselwörter und Satzzeichen (Parser)

Software Engineering

- ◆ Objektorientierte Modellierung (UML)
- ◆ Aktivitäts- und Interaktionsdiagramme

3.3 Begriffe der Automatentheorie

Bevor wir uns mit endlichen Automaten genauer beschäftigen können, müssen einige Grundbegriffe der Automatentheorie definiert werden. Die Bedeutung der Begriffe „Alphabet“, „Zeichenreihe“ und „Sprache“ sollten ja umgangssprachlich bekannt sein, für die Automatentheorie benötigen wir aber natürlich eine exakte formale mathematische Definition. Voraussetzung dafür sind Grundkenntnisse der Mengenlehre, die jedoch hier nicht noch mal wiederholt werden sollen.

Alphabet

Ein Alphabet ist eine endliche, nicht leere Menge von Symbolen. Ein Alphabet wird durch das Symbol Σ (großes Sigma) dargestellt.

Beispiele für häufig verwendete Alphabete in der Automatentheorie:

$\Sigma = \{0, 1\}$ das binäre Alphabet

$\Sigma = \{a, b, c, \dots, z\}$ die Menge aller Kleinbuchstaben
oder die Menge aller ASCII-Zeichen

Kein Alphabet hingegen wären die Menge aller natürlichen oder reellen Zahlen, da diese bekanntlich nicht endlich sind.

Zeichenreihe

Eine Zeichenreihe oder in der Literatur auch oft Wort oder String genannt, ist eine endliche Folge von Symbolen eines bestimmten Alphabets.

Beispielsweise ist 10011 eine Zeichenreihe des binären Alphabets $\Sigma = \{0, 1\}$, abc wäre allerdings keine Zeichenreihe des binären Alphabets jedoch z.B. eine Zeichenreihe des Alphabets der Kleinbuchstaben oder auch der Menge aller ASCII-Zeichen.

Eine Sonderstellung nimmt die **leere Zeichenreihe** ein, eine Zeichenreihe, die keine Symbole enthält, trotzdem jedoch in der Automatentheorie des Öfteren vorkommt und auch irgendwie beschrieben werden muss. Hierfür hat man sich auf das Symbol ε (kleines Epsilon) geeinigt. Die leere Zeichenreihe ε kann natürlich aus jedem beliebigen Alphabet stammen.

Häufig benötigt wird auch die **Länge einer Zeichenreihe**, definiert als die Anzahl der für Symbole verfügbaren Positionen (Anzahl der Buchstaben des Wortes). Beispielsweise hat die Zeichenreihe 10011 die Länge 5. Oft wird die Länge einer Zeichenreihe umgangssprachlich auch einfach als Anzahl der Symbole definiert, was streng genommen problematisch ist, da bei der Zeichenreihe 10011 die Anzahl der (unterschiedlichen) Symbole ja nur zwei beträgt, nämlich 0 und 1, hier jedoch die Länge des Wortes, also 5 gemeint ist.

Die Standardnotation für die Länge einer Zeichenreihe w lautet $|w|$.

Beispiel: $|101| = 3$ und $|\varepsilon| = 0$

D.h. die Länge der leeren Zeichenreihe ε wird mit 0 definiert.

Potenzen eines Alphabets

Eine Potenz eines Alphabets definiert die Menge aller Zeichenreihen einer bestimmten Länge über dem Alphabet Σ .

Σ^k ist die Menge aller möglichen Zeichenreihen mit der Länge k , deren Symbole aus dem Alphabet Σ stammen. Es gilt $\Sigma^0 = \{\varepsilon\}$ ungeachtet dessen, welches Alphabet Σ bezeichnet.

Beispiel: Wenn $\Sigma = \{0, 1\}$, dann ist

$$\Sigma^1 = \{0, 1\}$$

$$\Sigma^2 = \{00, 01, 10, 11\}$$

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Σ^* ist die Menge aller Zeichenreihen über einem Alphabet Σ .

D.h. $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$ (die Vereinigungsmenge aller Potenzen)

Beispiel: $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$

Manchmal möchte man die leere Zeichenreihe aus der Menge von Zeichenreihen ausschließen und definiert somit Σ^+ als die Menge aller nichtleeren Zeichenreihen des Alphabets Σ .

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$$

$$\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$$

Sprache

Eine Sprache definiert eine Menge von Zeichenreihen aus Σ^* , wobei Σ ein bestimmtes Alphabet darstellt. D.h. wenn Σ ein Alphabet ist und $L \subseteq \Sigma^*$, dann ist L eine Sprache über dem Alphabet Σ . [Hopcroft, 2002]

Eine Sprache ist immer auf einem festen endlichen Alphabet definiert, kann jedoch eine unendliche Anzahl von Zeichenreihen aus Σ^* enthalten, wobei natürlich nicht alle Symbole aus Σ vorkommen müssen.

Umgangssprachliche Beispiele

- ♦ Englisch ist eine Sammlung zulässiger englischer Wörter (eine Menge von Zeichenreihen) über dem normalen lateinischen Alphabet
- ♦ Die Programmiersprache C ist eine Sprache, die auf einem Alphabet, das eine Teilmenge des ASCII-Zeichensatzes ist, basiert. Die Sprachelemente von C bilden eine Teilmenge der möglichen Zeichenreihen dieses Alphabets.

Abstraktes Beispiel

- ♦ Die Sprache aller Zeichenreihen die aus n Nullen gefolgt von n Einsen bestehen, wobei $n \geq 0$ ist: $L = \{ \varepsilon, 01, 0011, 000111, \dots \}$

Operationen auf Sprachen

Neben den Mengenoperationen Vereinigung, Durchschnitt, Differenz gibt es spezielle Operationen, die für die weiteren Betrachtungen von formalen Sprachen für uns relevant sind.

Konkatenation

Der Konkatenationsoperator \circ bezeichnet das Hintereinanderschreiben von Zeichenreihen und ist wie folgt definiert:

$$v \circ w = vw$$

Der Konkatenationsoperator ist allerdings auch für Sprachen definiert. Gegeben sind zwei Sprachen L_1 und L_2 über dem Alphabet Σ . Dann gilt:

$$L_1 \circ L_2 = \{ vw \mid v \in L_1, w \in L_2 \}$$

Es werden also alle Wörter aus L_1 mit allen Wörtern aus L_2 verknüpft. Der Operator \circ kann dabei auch weggelassen werden: $L_1 \circ L_2 = L_1 L_2$.

Beispiel: $L_1 = \{\varepsilon, ab, abb\}$ $L_2 = \{b, ba\}$

$$L_1 \circ L_2 = \{ b, ba, abb, abba, abbb, abbbba \}$$

$$L_2 \circ L_1 = \{ b, bab, babb, ba, baab, baabb \}$$

Potenz einer Sprache

Die n-te Potenz einer Sprache L über dem Alphabet Σ ist definiert durch:

$$L^0 = \{\varepsilon\}$$

$$L^{n+1} = L^n \circ L$$

Verdeutlichen wir diese Definition wieder durch ein Beispiel, gegeben ist die Sprache $L = \{a, ab\}$, dann gilt:

$L^0 = \{\epsilon\}$
 $L^1 = L^0 \circ L = \{\epsilon\} \circ \{a, ab\} = \{a, ab\}$
 $L^2 = L^1 \circ L = \{a, ab\} \circ \{a, ab\} = \{aa, aab, aba, abab\}$
...

Kleene-Stern-Produkt (oder Abschluss bzw. Hülle)

Das Kleene-Stern-Produkt L^* einer Sprache ist definiert als die Vereinigung aller Potenzen L^n , $n \geq 0$:

$$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$$

Somit beschreibt L^* die Menge aller Zeichenreihen, die durch die Verkettung einer beliebigen Anzahl von Zeichenreihen aus L (wobei Wiederholungen zulässig sind, d.h. dieselbe Zeichenreihe kann mehrmals verwendet werden) gebildet werden. [Hopcroft, 2002]

Nach diesem kurzen Ausflug in die Mathematik haben wir das nötige Rüstzeug, um endliche Automaten und formale Sprachen analysieren zu können.

3.4 Deterministische endliche Automaten

Nachdem in Kapitel 3.1 ein Kippschalter als erster einfacher endlicher Automat vorgestellt wurde und im vorigen Kapitel notwendige Grundbegriffe der Automatentheorie erläutert wurden, ist es nun an der Zeit, das formale Konzept eines Automaten zu beschreiben. Wir beschränken uns dabei auf die einfachste Form von Automaten, **die Deterministischen Endlichen Automaten (DEA)**, oder in der englischen Fachliteratur auch **deterministic finite automaton (DFA)** genannt.

Deterministisch bedeutet dabei, dass es in jedem Zustand für jedes Eingabesymbol nur höchstens einen Folgezustand gibt, d.h. bei Eingabe eines Symbols aus dem Alphabet kann ausgehend vom aktuellen Zustand nur in einen Folgezustand gewechselt werden. Anders ausgedrückt beschreibt die Eigenschaft deterministisch, dass in einem deterministischen Automaten nur ein Zustand gleichzeitig eingenommen werden kann.

Im Gegensatz dazu kann sich ein **nichtdeterministischer Automat** gleichzeitig in mehreren Zuständen befinden. D.h. ausgehend von einem Zustand und einem Eingabesymbol können mehrere Folgezustände existieren. Wir beschäftigen uns jedoch in dieser Lehrveranstaltung nur mit deterministischen endlichen Automaten, was in der Mächtigkeit der Automaten auch keine Einschränkung bedeutet, da gezeigt werden kann, dass deterministische und nichtdeterministische Automaten von den definierten Sprachen her äquivalent sind, auch wenn mit nichtdeterministischen Automaten gewisse Anwendungen effizienter beschrieben werden können. Genauer dazu findet sich beispielsweise in [Hopcroft, 2002]. Für uns wird jedoch der Begriff Determinismus auch in Kapitel 5, Eigenschaften von Algorithmen noch eine Rolle spielen, und wir können dann die gewonnenen Erkenntnisse von der Automatentheorie weiter verwenden.

Der Begriff **endlich** wurde bereits in der Einleitung in Kapitel 3.1 verwendet und bedeutet, dass der Automat nur eine endliche Anzahl an Zuständen besitzt. Diese Limitierung ist natürlich für einen Informatiker äußerst relevant, da nur ein Automat mit einer endlichen Menge von Zuständen ohne Probleme mit einer fixen Ressourcenmenge z.B. als Schaltkreis in Hardware oder als einfaches Computerprogramm implementiert werden kann. Auf der anderen Seite bedeutet diese Einschränkung natürlich auch, dass das System des endlichen Automaten sorgfältig spezifiziert werden muss, da das Gedächtnis des Automaten in Form der Zustände nur endlich ist und unter Umständen nicht die gesamte Historie des Systems beschrieben werden kann.

Definition

Formal besteht nun ein Deterministischer Endlicher Automat (DEA) aus folgenden Komponenten, nach [Hopcroft, 2002]:

- ◆ einer endlichen Menge von **Zuständen** Q .
- ◆ einer endlichen Menge von **Eingabesymbolen** Σ (auch Alphabet des Automaten genannt).
- ◆ einer **Übergangsfunktion** δ (kleines delta), der ein Zustand und ein Eingabesymbol als Argumente übergeben werden und die einen Zustand zurückgibt $Q \times \Sigma \rightarrow Q$. D.h. die Funktion δ bestimmt ausgehend von einem Zustand q und einem Eingabesymbol a einen Folgezustand p : $\delta(q,a)=p$. In der grafischen Darstellung des Automaten eines Kippschalters in Abbildung 1 wurde die Übergangsfunktion durch gerichtete Pfeile mit Beschriftung dargestellt.
- ◆ **einem Startzustand** q_0 , bei dem es sich um einen der in Q enthaltenen Zustände handelt.
- ◆ einer Menge **F finaler** oder **akzeptierender Zustände**, die Menge F ist eine Teilmenge von Q . D.h. F kann mehrere Zustände aus Q beinhalten aber auch eine leere Menge sein, sodass ein DEA nicht unbedingt einen finalen Zustand haben muss.

Ein DEA wird also definiert durch das **Quintupel** $A = (Q, \Sigma, \delta, q_0, F)$.

Darstellungsmöglichkeiten

Da die formale Quintupel Notation eines DEA mit der detaillierten Auflistung der Übergangsfunktion δ aufwendig und schwer lesbar ist, haben sich zwei Darstellungsmöglichkeiten zur Beschreibung von deterministischen endlichen Automaten durchgesetzt, die beide ihre Vor- und Nachteile besitzen:

1. Zustandsdiagramme (state diagrams) bzw. Übergangsdiagramme als Graphenrepräsentation
2. Übergangstabellen mit einer tabellarischen Auflistung der Übergangsfunktion δ

Zustandsdiagramme

Ein Zustands- oder Übergangsdiagramm ist ein bewerteter gerichteter Graph der die Übergangsfunktion und damit den gesamten Automaten wie folgt beschreibt:

- ◆ Jeder Zustand q aus Q entspricht einem Knoten im Graphen und wird durch einen Kreis dargestellt.
- ◆ Jeder Zustandsübergang der Übergangsfunktion δ entspricht einem Pfeil (gerichtete Kante) in dem Graphen.
- ◆ Die Bewertung einer Kante ist das Eingabesymbol a , das diese Zustandsänderung bewirkt. Somit wird z.B. $\delta(q,a)=p$ durch einen Pfeil von Knoten q nach Knoten p mit der Beschriftung a dargestellt.
- ◆ Von jedem Knoten gehen so viele Kanten aus, wie es Eingabesymbole in Σ gibt.
- ◆ Der Startzustand q_0 wird durch einen eingehenden Pfeil markiert und optional mit „Start“ beschriftet.
- ◆ Finale bzw. akzeptierende Zustände aus der Menge F werden durch Doppelkreise gekennzeichnet.

- ♦ Zu Vereinfachung der Diagramme ist es möglich mehrere Beschriftungen an einer Kante anzubringen, falls es mehrere Eingabesymbole gibt, die einen Übergang von q nach p bewirken. Diese Symbole werden dann auf der Kante durch einen Schrägstrich oder Beistrich voneinander getrennt.
- ♦ Ist $\delta(q,a)=q$, d.h. bewirkt das Eingabesymbol a im Zustand q einen Zustandsübergang auf sich selbst, wird im Graph der Pfeil als eine Schleife von q nach q mit der Beschriftung a gezeichnet.

Zusammengefasst sind die verschiedenen Darstellungsmöglichkeiten des Übergangsdiagrammes eines endlichen Automaten in Abbildung 3-2 ersichtlich.

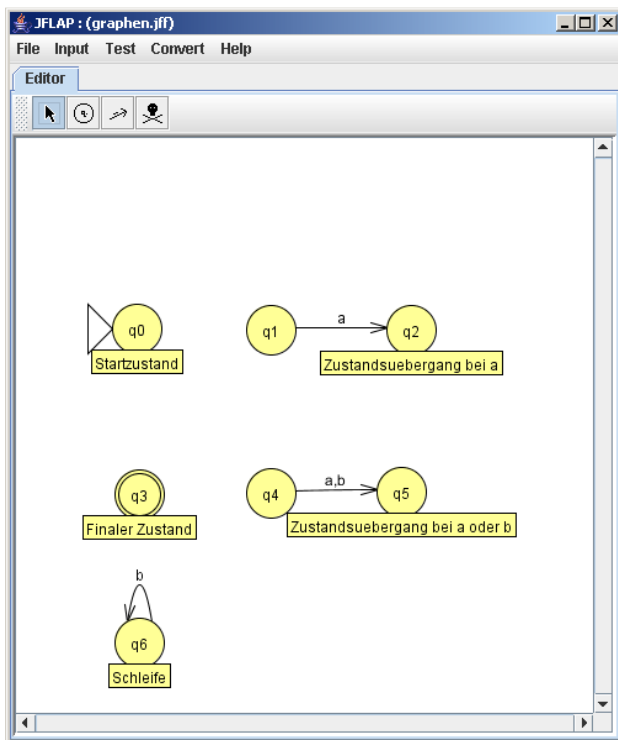


Abbildung 3-2 Graphische Notation eines Automaten

Übergangstabelle

Eine Übergangstabelle ist eine tabellarische Darstellung einer Funktion wie δ , die zwei Argumente verarbeitet und einen Wert zurückgibt. Die Zeilen der Tabelle entsprechen den Zuständen und die Spalten den Eingabesymbolen des Automaten. Der Eintrag in der Zeile für den Zustand q_i und der Spalte für die Eingabe a_j entspricht dem Folgezustand $\delta(q_i, a_j)$.

Nachdem wir nun beide Möglichkeiten, einen DEA zu definieren, vorgestellt haben, ist es an der Zeit anhand eines ausführlichen Beispiels die gewonnenen Erkenntnisse in die Praxis umzusetzen und unseren ersten DEA zu definieren und zu analysieren.

Beispiel für einen DEA

Gegeben ist folgender deterministischer endliche Automat $A = (Q, \Sigma, \delta, s_0, F)$ durch

$$Q = \{s_0, s_1, s_2\}$$

$$\Sigma = \{a, b\}$$

$$F = \{s_2\}$$

und δ beschrieben durch folgende Übergangstabelle:

	a	b
s ₀	s ₀	s ₁
s ₁	s ₀	s ₂
s ₂	s ₀	s ₂

Der DEA ist durch diese Angabe des Quintupels A und der Übergangstabelle für δ vollständig beschrieben. Der Automat besitzt die 3 Zustände der Menge Q (s_0 , s_1 und s_2). Zustände werden nicht immer mit q sondern wie in diesem Beispiel auch manchmal mit s für engl. state beschriftet. Das Eingabealphabet des Automaten Σ beinhaltet die beiden Symbole a und b . Der Startzustand ist s_0 , er wird entsprechend der Definition in Kapitel 2.4 an vierter Stelle im Quintupel angegeben und muss natürlich nicht immer zwangsläufig der erste Zustand aus Q sein. Die Menge der finalen oder akzeptierenden Zustände F beinhaltet in diesem Beispiel den Zustand s_2 .

Mit diesen Informationen ist es nun sehr leicht, das zu diesem endlichen Automaten zugehörige Zustandsdiagramm zu zeichnen. Das Diagramm ist in Abbildung 3-3 ersichtlich.

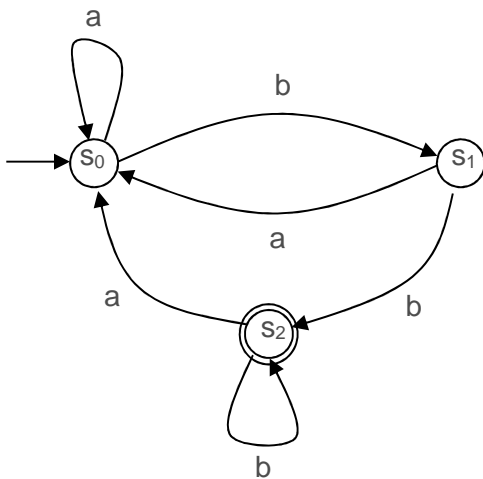


Abbildung 3-3 Zustandsdiagramm eines DEA

Beide Darstellungsformen sind natürlich äquivalent. Aus der Quintupel Notation zusammen mit der Übergangstabelle für δ kann sehr leicht das Zustandsdiagramm generiert werden. Umgekehrt ist ein endlicher Automat auch vollständig durch ein Zustandsdiagramm definiert, und es kann daraus ohne Probleme die Übergangstabelle des Automaten angegeben werden.

Was sind nun die Vor- und Nachteile beider Darstellungsformen? Das Zustandsdiagramm als graphische Repräsentation bietet sicherlich für den Menschen eine anschaulichere Notation der Zustandsübergänge und damit des Verhaltens des Automaten als eine Tabelle. Dies gilt natürlich auch nur für eine überschaubare Anzahl von Zuständen, da ansonsten der Graph mit evtl. vielen überkreuzten Pfeilen rasch unübersichtlich werden kann. Auf der anderen Seite ist die Tabelle sicherlich leichter in einer computergesteuerten Verarbeitung weiter zu verwenden.

Doch die Übergangstabelle bietet einen weiteren Vorteil bei der Überprüfung, ob die Übergangsfunktion δ des DEA vollständig definiert ist. Im Zustandsdiagramm müssen von jedem Knoten so viele Kanten ausgehen, wie es verschiedene Symbole im Eingabealphabet gibt. In unserem Beispiel mit dem Eingabealphabet $\Sigma = \{a, b\}$ heißt das also, dass von jedem der drei Knoten genau zwei Pfeile ausgehen. Dies ist selbst in dem einfachen Beispiel nicht sofort ersichtlich. In der Übergangstabelle jedoch ist die Vollständigkeit auf den ersten Blick klar ersichtlich, nämlich genau dann, wenn die Tabelle in allen Zellen vollständig ausgefüllt ist.

Verhalten und Sprache eines DEA

Nachdem wir nun ausführlich die Darstellungsmöglichkeiten eines DEA diskutiert haben, können wir nun das Verhalten eines deterministischen endlichen Automaten definieren. Am besten können wir das Verhalten anhand des konkreten DEA aus dem vorigen Kapitel, Abbildung 3-3 beschreiben.

Der DEA befindet sich zu Systemstart im Startzustand s_0 . Nun werden Eingabesymbole aus dem Alphabet des Automaten, in unserem Fall a oder b eingelesen. Das führt den DEA mit Hilfe der Übergangsfunktion in einen anderen (oder denselben) Zustand. Die Funktion bestimmt ausgehend von einem Zustand und einem Eingabezeichen genau einen Folgezustand. So wechselt der DEA in unserem Beispiel ausgehend vom Startzustand beim Einlesen eines b in den Folgezustand s_1 , beim Einlesen eines a wird der Automat allerdings im Startzustand s_0 bleiben. Der Automat liest nun Symbol für Symbol einer kompletten Zeichenreihe ein und verändert jeweils seinen Zustand abhängig vom aktuellen Eingabesymbol und dem aktuellen Zustand.

Wenn sich der DEA nach Eingabe einer kompletten Zeichenreihe in einen finalen bzw. akzeptierenden Zustand befindet, so bezeichnet man diese Zeichenreihe als ein Wort aus der Sprache des DEA. Dieser Sachverhalt wird aber noch genauer definiert. Betrachten wir zuerst einige Zeichenreihen als Eingabebeispiele für unseren DEA und markieren wir jeweils den erreichten Zustand nach Verarbeitung der Zeichenreihe:

	Eingabe	erreichter Zustand
1	aab	$\delta(s_0, aab) = s_1$
2	baba	$\delta(s_0, baba) = s_0$
3	abb	$\delta(s_0, abb) = s_2$
4	abbabb	$\delta(s_0, abbabb) = s_2$
5	bba	$\delta(s_0, bba) = s_0$

Von diesen fünf Zeichenreihen führen nur die dritte und vierte Zeichenreihe (abb und abbabb) in den finalen Zustand s_2 . Interessant ist nun die Fragestellung, welche Zeichenreihen allgemein den DEA in einen finalen Zustand führen. Nach ein paar weiteren Testeingaben und mit etwas Gespür für endliche Automaten sollte man rasch zur Erkenntnis gelangen, dass genau alle Eingabewörter, die auf „bb“ enden, in einen finalen bzw. akzeptierenden Zustand führen.

Allgemein akzeptiert ein Automat A eine Zeichenreihe $w \in \Sigma^*$, falls $\delta(s_0, w) \in F$ gilt d.h. wenn die Verarbeitung des Wortes in einen finalen Zustand führt.

Nun kann die **Sprache eines Automaten** wie folgt definiert werden:

Unter der Sprache L eines Automaten A verstehen wir die Menge aller Worte die der Automat akzeptiert.

Also $L(A) ::= \{ w \in \Sigma^* \mid \delta(s_0, w) \in F \}$

L(A) heißt die von A akzeptierte Sprache

In unserem Beispiel wäre also L(A) die Menge aller Wörter aus $\{a,b\}^*$, die auf „bb“ enden.

Aufgabenstellungen in der Praxis

Nachdem wir definiert haben, dass zu jedem deterministischen Automaten eine zugehörige Sprache existiert bieten sich nun zwei Aufgabenstellungen an, die diesen Zusammenhang ausnutzen. Auf der einen Seite gibt es Aufgabenstellungen, in denen ausgehend von einem endlichen Automaten die akzeptierte Sprache des Automaten gesucht wird, so wie wir das im vorigen Beispiel durchgeführt haben. Häufiger in der Praxis ist allerdings die umgekehrte Aufgabenstellung. Ausgehend von einer Sprache wird ein endlicher Automat gesucht, der diese Sprache akzeptiert. So werden z.B. endliche Automaten benötigt, die einen gegebenen Suchtext, ein Muster oder syntaktisch korrekte Statements einer Programmiersprache erkennen und akzeptieren können. Solche Automaten finden dann in Textverarbeitungssystemen und Compilern ihre Anwendung. Diese Aufgabenstellung ist deutlich komplexer, da hier ja ausgehend von einer Sprache ein kompletter Automat entworfen werden muss und von vorn herein oft nicht klar ist, wie viele Zustände der Automat benötigen wird und welche Zustandsübergänge modelliert werden müssen.

Folgendes Beispiel soll diese Aufgabenstellung verdeutlichen:

Beispiel Automatendesign

Gegeben ist das Eingabealphabet $\Sigma = \{a, b\}$ und die Sprache L , wobei L als die Menge aller Worte aus Σ^* definiert ist, in denen mindestens zwei aufeinander folgende a oder zwei aufeinander folgende b vorkommen. So wären z.B. die Zeichenreihen „aa“, „abb“ oder „babbbba“ Elemente von L , nicht aber z.B. „aba“.

Gesucht ist nun ein deterministischer endlicher Automat, der die Sprache L akzeptiert.

Es ist klar, dass die Zustände des DEA in irgendeiner Form die Anzahl der a bzw. b speichern werden, um zu erkennen, ob bereits zwei a oder b eingelesen wurden, aber wie viele Zustände werden dafür benötigt? Ziel bei der Entwicklung eines DEA sollte es immer sein, mit so wenigen Zuständen wie möglich auszukommen, um die Komplexität des Automaten nicht unnötig zu erhöhen. Betrachten wir einmal nur das Eingabesymbol a . Hier genügt es zu unterscheiden, ob kein a , ein a oder mindestens zwei a eingelesen wurden. D.h. wir haben einmal 3 Zustände identifiziert. Der Startzustand s_0 bedeutet, dass noch kein a eingelesen wurde. In den Zustand s_1 wird gewechselt, wenn ein a eingelesen wurde. Bei einem weiteren a haben wir bereits ein Wort der Sprache des Automaten erreicht und wir wechseln in den akzeptierenden Zustand s_2 . Jedes weitere a verursacht keine Zustandsänderung, da ja bereits mindestens 2 a eingelesen wurden, bleibt der Automat im finalen Zustand s_2 .

Was passiert nun allerdings beim Einlesen von einem b ? Reichen die drei Zustände aus oder benötigen wir neue Zustände zur Abarbeitung von b ? Was würde passieren wenn wir den Zustand s_1 dahingehend erweitern, dass er entweder ein a oder ein b speichert, d.h. es gebe einen Zustandsübergang von s_0 nach s_1 sowohl für a als auch für b ? Der Automat weiß dann allerdings in s_1 nicht mehr, ob als letztes Zeichen ein a oder b eingelesen wurde und würde in weiterer Folge auch „ba“ akzeptieren und in den finalen Zustand s_2 übergehen. D.h. wir benötigen auf jeden Fall noch einen vierten Zustand s_3 der analog zu s_1 jedoch ein b speichert. Wird nun ein zweites b eingelesen, kann der Automat wieder in den finalen Zustand s_2 wechseln. Damit geht zwar die Information verloren, ob zwei a oder zwei b hintereinander eingelesen wurden, diese Information ist allerdings auch nicht relevant, da beide Varianten akzeptiert werden sollen und keine Weiterverarbeitung mehr stattfinden muss.

Zusammengefasst sind unsere Überlegungen im Zustandsdiagramm des DEA in Abbildung 3-4 auf der folgenden Seite. Interessant sind noch die Zustandsübergänge von s_1 nach s_3 . Wird vom Automaten beispielsweise die Zeichenreihe „ab“ eingelesen, wechselt der DEA zuerst in Zustand s_1 und danach in Zustand s_3 , verwirft damit praktisch das erste a , speichert ein b und wartet auf ein zweites b . Folgt allerdings wieder ein a , wechselt der Automat zurück zu Zustand s_1 usw.

Vollständig definiert in Quintupel Notation ist unser Automat nun wie folgt definiert:

$A = (Q, \Sigma, \delta, s_0, F)$ mit

$Q = \{s_0, s_1, s_2, s_3\}$

$\Sigma = \{a, b\}$

$F = \{s_2\}$

und δ beschrieben durch folgende Übergangstabelle:

	a	b
s ₀	s ₁	s ₃
s ₁	s ₂	s ₃
s ₂	s ₂	s ₂
s ₃	s ₁	s ₂

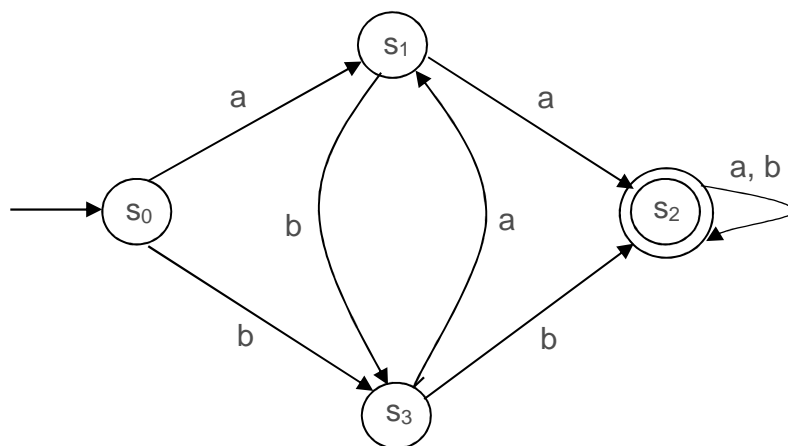


Abbildung 3-4 Zustandsdiagramm für DEA aus dem Beispiel Automatendesign

3.5 Selbstlernkontrolle

Fragen

Welche der folgenden drei Mengen sind Alphabete?

$B = \{*, \#, <\}$

$N = \{1, 2, 3, 4, 5, \dots\}$

$Q = \{\}$

- (a) B, Q
- (b) nur N
- (c) nur B
- (d) B, N

Gegeben sei das binäre Alphabet $A = \{0, 1\}$. Welche der Aussagen bezüglich der Potenz des Alphabetes ist wahr?

- (a) $A^3 = \{111, 110, 101, 100, 011, 010, 001, 000\}$
- (b) $A^0 = \{1\}$, da alles hoch Null gleich 1 ist
- (c) Die Menge A^* hat unendlich viele Elemente.
- (d) A^n ist die Menge aller Zeichenreihen mit der Länge 2 mal n.

Sei L eine Sprache über dem Alphabet Σ . Welche der Aussagen bezüglich der Sprache L ist wahr?

- (a) L ist eine Menge von Zeichenreihen aus Σ^* .
- (b) $\Sigma^* \subseteq L$.
- (c) Eine Sprache L kann nur endlich viele Zeichenreihen enthalten.

Durch welches Quintupel wird ein endlicher deterministischer Automat definiert?

- (a) 1. der Menge von Zuständen
2. dem Bandalphabet
3. einer Übergangsfunktion
4. einem Startzustand
5. der Menge der finalen Zustände
- (b) 1. der Menge von Zuständen
2. der Menge von Eingabesymbolen
3. einer Übergangsfunktion
4. einem Startzustand
5. der Menge der akzeptierenden Zustände
- (c) 1. der Menge von Zuständen
2. der Menge der Nonterminalen
3. der Menge der Terminalen
4. einem Startzustand
5. der Menge der akzeptierenden Zustände

Was bedeutet der Begriff „deterministisch“ für einen deterministischen endlichen Automaten?

- (a) Da es nur endlich viele Eingabesymbole gibt, ist der Automat deterministisch.
- (b) Der Automat erreicht nach einer endlichen Anzahl von Eingabesymbolen auf jeden Fall den akzeptierenden Zustand.
- (c) In jedem Zustand gibt es für jedes Eingabesymbol höchstens einen Folgezustand.
- (d) Sowohl die Anzahl der Zustände als auch die Anzahl der Eingabesymbole ist endlich.

Worauf bezieht sich die Bezeichnung „endlich“ bei einem endlichen Automaten?

- (a) Der Automat baut auf einem endlichen Alphabet auf.
- (b) Der Automat besitzt eine endliche Menge an Zuständen.
- (c) Der Automat kommt nach endlich vielen Schritten zum akzeptierenden Zustand.
- (d) Der Automat benötigt eine endliche Zeit um zum akzeptierenden Zustand zu kommen.

Lösungen

Welche der folgenden drei Mengen sind Alphabete?

$B = \{*, \#, <\}$

$N = \{1, 2, 3, 4, 5, \dots\}$

$Q = \{\}$

- (a) B, Q
- (b) nur N
- (c) nur B**
- (d) B, N

Gegeben sei das binäre Alphabet $A = \{0, 1\}$. Welche der Aussagen bezüglich der Potenz des Alphabetes ist wahr?

- (a) $A^3 = \{111, 110, 101, 100, 011, 010, 001, 000\}$
- (b) $A^0 = \{1\}$, da alles hoch Null gleich 1 ist
- (c) Die Menge A^* hat unendlich viele Elemente.**
- (d) A^n ist die Menge aller Zeichenreihen mit der Länge 2 mal n.

Sei L eine Sprache über dem Alphabet Σ . Welche der Aussagen bezüglich der Sprache L ist wahr?

- (a) L ist eine Menge von Zeichenreihen aus Σ^* .**
- (b) $\Sigma^* \subseteq L$.
- (c) Eine Sprache L kann nur endlich viele Zeichenreihen enthalten.

Durch welches Quintupel wird ein endlicher deterministischer Automat definiert?

- (a) 1. der Menge von Zuständen
2. dem Bandalphabet
3. einer Übergangsfunktion
4. einem Startzustand
5. der Menge der finalen Zustände
- (b) 1. der Menge von Zuständen
2. der Menge von Eingabesymbolen
3. einer Übergangsfunktion
4. einem Startzustand
5. der Menge der akzeptierenden Zustände**
- (c) 1. der Menge von Zuständen
2. der Menge der Nonterminalen
3. der Menge der Terminalen
4. einem Startzustand
5. der Menge der akzeptierenden Zustände

Was bedeutet der Begriff „deterministisch“ für einen deterministischen endlichen Automaten?

- (a) Da es nur endlich viele Eingabesymbole gibt, ist der Automat deterministisch.
- (b) Der Automat erreicht nach einer endlichen Anzahl von Eingabesymbolen auf jeden Fall den akzeptierenden Zustand.
- (c) In jedem Zustand gibt es für jedes Eingabesymbol höchstens einen Folgezustand.**
- (d) Sowohl die Anzahl der Zustände als auch die Anzahl der Eingabesymbole ist endlich.

Worauf bezieht sich die Bezeichnung „endlich“ bei einem endlichen Automaten?

- (a) Der Automat baut auf einem endlichen Alphabet auf.
- (b) Der Automat besitzt eine endliche Menge an Zuständen.**
- (c) Der Automat kommt nach endlich vielen Schritten zum akzeptierenden Zustand.
- (d) Der Automat benötigt eine endliche Zeit um zum akzeptierenden Zustand zu kommen.

3.6 Übungsteil

Übungstool JFLAP

Zum Zeichnen und Testen der Zustandsdiagramme kann das Tool JFLAP verwendet werden.

<http://www.jflap.org/>

Machen Sie sich mit dem Tool vertraut.

Übung 1: Mausefalle

Zeichnen Sie ein Zustandsdiagramm einer Mausefalle. Die Falle kann gespannt oder nicht gespannt sein. Die „Eingabe“ des Automaten kann „Maus kommt“ und „Maus kommt nicht“ sein. Stellen Sie die möglichen Übergänge ohne Startposition graphisch dar.

Übung 2

Gegeben ist der endliche Automat $A = (Q, \Sigma, \delta, s_0, F)$ durch $Q = \{s_0, s_1\}$, $\Sigma = \{0, 1\}$, $F = \{s_1\}$ und der durch die folgende Tabelle beschriebenen Übergangsfunktion δ .

	0	1
s_0	s_0	s_1
s_1	s_1	s_0

- 1) Zeichnen Sie das Zustandsdiagramm dieses Automaten.
- 2) Probieren Sie einige Zeichenreihen als Eingaben aus.
- 3) Welche Zeichenreihen bringen diesen Automaten in den akzeptierenden Zustand?
- 4) Wie ist die Sprache dieses Automaten definiert?

Übung 3: Getränkeautomat

Entwerfen Sie einen deterministischen endlichen Automaten für einen Getränkeautomaten. Getränke kosten 2 Euro, der Automat akzeptiert 50 Cent, 1 Euro und 2 Euro Münzen. Werden also mindestens 2 Euro eingeworfen, geht der Automat in den akzeptierenden Zustand und wirft das Getränk aus. Retourgeld wird nicht berücksichtigt! Zeichnen Sie das Zustandsdiagramm dieses Automaten.

Übung 4

Entwerfen Sie einen deterministischen endlichen Automaten der die Sprache L akzeptiert, wobei für L gilt:

$L = \{w \mid w \text{ enthält eine gerade Anzahl von Nullen und eine gerade Anzahl von Einsen}\}$

Anmerkung: Auch 0 gilt als gerade Anzahl, damit ist auch ϵ ein Element von L !

- 1) Zeichnen Sie das Zustandsdiagramm dieses Automaten.
- 2) Schreiben Sie die Übergangstabelle des Automaten nieder.
- 3) Überprüfen Sie mittels einiger Zeichenreihen den Automaten.

Übung 5

Erstellen Sie einen Automaten, der in den akzeptierenden Zustand wechselt, wenn er eine binäre Zahl, die dezimal durch 4 teilbar ist, verarbeitet.

- 1) Zeichnen Sie das Zustandsdiagramm des Automaten.
- 2) Erstellen Sie die Übergangstabelle des Automaten.

Anmerkung: Der Automaten soll die binäre Zahl von rechts nach links abarbeiten, also vom Startzustand ausgehend wird zuerst die rechte Ziffer der binären Zahl eingelesen und die Anzahl der Stellen kann beliebig groß werden.

Übung 6: Marmelspiel

Gegeben ist ein Marmelspiel mit zwei Eingängen (A, B) und zwei Ausgängen (C, D). Eine Marmel wird bei A oder B in die Spielbahn fallen gelassen. Die Hebel x1, x2 und x3 an den Verzweigungen bewirken, dass die Marmel nach links oder rechts rollt. Sobald eine Marmel auf einen dieser Hebel trifft, wird dieser Hebel nach dem Passieren der Marmel umgestellt, sodass die nächste Marmel in die andere Richtung rollt.

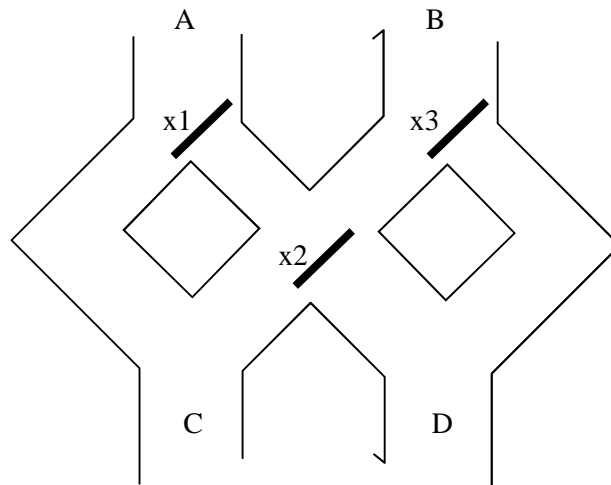


Abbildung 3-5 Marmelspiel als endlicher Automat

Modellieren Sie das Spiel als endlichen Automaten. Die Eingaben A und B sollen die Öffnungen repräsentieren, in die man die Marmel fallen lässt. Wenn die Marmel durch Öffnung D aus dem Spiel rollt, dann soll dies als akzeptierender Zustand gelten. Als nicht akzeptierender Zustand soll gelten, wenn die Marmel durch C aus dem Spiel rollt.

Beschreiben Sie die Sprache des Automaten informell.

Übung 7: Marmelspiel

Recherchieren Sie im Internet die beiden Automatentypen Moore- und Mealyautomaten. Ist es mit diesen Automaten einfacher das Marmelspiel zu implementieren?

4 Formale Sprachen und Grammatiken

4.1 Reguläre Sprachen

Wir haben in Kapitel 3.4 bereits die Sprache eines DEA als die Menge aller Zeichenreihen, die der Automat akzeptiert, definiert. Nun wollen wir die Eigenschaften dieser Sprachen genauer betrachten und eine neue Notation namens reguläre Ausdrücke einführen, die solche Sprachen kompakt beschreiben kann.

Beginnen wir mit der Definition von **regulären Sprachen**.

Eine Sprache $L \subseteq \Sigma^*$ heißt regulär, falls es einen endlichen Automaten A gibt, der L akzeptiert d.h. für den $L = L(A)$ gilt [Vossen, 2004].

Somit heißen von endlichen Automaten akzeptierte Sprachen reguläre Sprachen. Wir werden später noch untersuchen, ob alle formalen Sprachen in der Informatik reguläre Sprachen sind, d.h. anders ausgedrückt, ob zu jeder Sprache ein endlicher Automat gefunden werden kann, der diese akzeptiert, oder ob es auch komplexere Sprachen gibt, für die das nicht gelingt. Zuvor betrachten wir aber noch ein weiteres Konzept, um reguläre Sprachen zu definieren.

Lernergebnisse

Nach diesem Kapitel sind die Studierenden in der Lage,

- Den Zusammenhang zwischen Automaten, regulären Ausdrücken und Grammatiken zu erläutern.
- einfache abstrakte Aufgabenstellungen der formalen Sprachen als regulären Ausdruck bzw. kontextfreie Grammatik darzustellen.

4.2 Reguläre Ausdrücke

Informell betrachtet ist ein regulärer Ausdruck (engl.: regular expression) nichts anderes als **ein Muster für die Wörter** einer regulären Sprache und stellt somit also ein beschreibendes Konzept für reguläre Sprachen dar.

Motivation

Reguläre Ausdrücke besitzen eine Vielzahl von Anwendungsmöglichkeiten in der praktischen und angewandten Informatik. Die meisten Texteditoren und Textverarbeitungsprogramme unterstützen z.B. bei der Suchen/Ersetzen Funktion die Angabe eines regulären Ausdrucks um nach Mustern in Texten zu suchen. Weiters werden reguläre Ausdrücke zur formalen Beschreibung von zulässigen Eingaben in Webformularen oder anderen Dialogmasken verwendet. Wie muss z.B. eine gültige Email-Adresse aufgebaut sein oder wann ist die Eingabe einer Telefonnummer mit Vorwahl korrekt und wann nicht? Es gibt sogar bereits Werkzeuge, die aus der formalen Beschreibung der regulären Ausdrücke automatisch so genannte Scanner generieren, also Software, die zur Überprüfung von Benutzereingaben auf syntaktische Korrektheit eingesetzt werden kann. Ein Beispiel dafür wäre das UNIX Tool `lex`, dass ausgehend von regulären Ausdrücken automatisch C-Code generiert. Varianten davon gibt es natürlich für jede moderne Programmiersprache. Auch in Compilern selbst werden

solche Verfahren eingesetzt. Ein weiteres bekanntes Tool ist das UNIX Kommando `grep`², das in Dateien nach Mustern, angegeben durch reguläre Ausdrücke, suchen kann.

Zusammenhang mit endlichen Automaten

Endliche Automaten und reguläre Ausdrücke sind äquivalent, der Beweis findet sich in der Literatur z.B. in [Vossen, 2004]. D.h. zu jedem endlichen Automaten existiert ein entsprechender regulärer Ausdruck der die Sprache des Automaten beschreibt, und zu jedem regulären Ausdruck existiert ein entsprechender endlicher Automat der die beschriebene Sprache akzeptiert.

Versuchen wir nun diesen Zusammenhang auch bildlich darzustellen. Was passiert, wenn wir durch einen Automaten beginnend beim Startzustand bis zu einem finalen Zustand wandern und die Eingabesymbole der aufgetretenen Zustandsübergänge notieren?

Wir erzeugen damit ein Wort $w \in L(A)$, also der Sprache des Automaten. Bei genauerer Betrachtung werden dabei aber immer wieder folgende Operationen durchgeführt, denen Operatoren zugeordnet werden können, die wir bereits in Kapitel 3.4 definiert haben:

- ◆ Konkatination (Operator \circ): Buchstaben hintereinander schreiben.
- ◆ Selektion (Oder-Operator $|$): Auswahl eines Zustandes, wenn von einem Zustand mehrere Übergänge ausgehen, entspricht dem Vereinigungsoperator \cup .
- ◆ Wiederholung (Sternoperator $*$): Rückkehr zu einem bereits besuchten Zustand und Wiederholung eines bereits erzeugten Teilwortes. Durch Wiederholung kann also ein Teilwort beliebig oft erzeugt werden.

Definition

Ein regulärer Ausdruck über einem Alphabet Σ ist ein Muster für eine Menge von Wörtern, die sich durch Anwendung der Operatoren auf die Buchstaben von Σ ergeben, formal lautet die Definition wie folgt:

- ◆ \emptyset ist ein regulärer Ausdruck der die leere Sprache $\{\}$ definiert.
- ◆ ε ist ein regulärer Ausdruck der die Sprache definiert, die nur das leere Wort enthält $\{\varepsilon\}$.
- ◆ Jeder Buchstabe $a \in \Sigma$ ist ein regulärer Ausdruck, der jeweils die Sprache definiert, die nur ihn selbst als einziges Wort enthält, $L(a)=\{a\}$.
- ◆ Sind A und B reguläre Ausdrücke dann ist auch:
 - $A \circ B$ ein regulärer Ausdruck, wobei der Konkatination der Ausdrücke A und B die Konkatination ihrer Sprachen zugeordnet wird, also $L(A) \circ L(B)$.
 - $A | B$ ein regulärer Ausdruck, wobei der Selektion der Ausdrücke die Vereinigung ihrer Sprachen zugeordnet wird, also $L(A) \cup L(B)$.
 - A^* ein regulärer Ausdruck, wobei dem Wiederholungsoperator das Kleene-Stern-Produkt $(L(A))^*$ also die Vereinigung aller Potenzen der von A definierten Sprache $L(A)$ entspricht.

Operanden von regulären Ausdrücken können zusätzlich noch mit runden Klammern gruppiert werden, fehlen Klammern gelten Vorrangeregeln, die aus gewöhnlichen arithmetischen Ausdrücken bekannt sein sollten. Der Sternoperator $*$ hat die höchste Priorität gefolgt vom Konkatinationsoperator \circ und dem Selektionsoperator $|$, der somit die niedrigste Priorität aufweist.

² `grep` steht für globally search for regular expressions and print.

Beispiel

Folgendes Beispiel soll diese kompliziert erscheinende Definition anschaulich erläutern:
Gegeben ist das binäre Alphabet $\Sigma=\{0,1\}$ und der reguläre Ausdruck REXP^3 : $(0(0|1)^*)$.

Exakt würde der Ausdruck $(0 \circ (0|1)^*)$ lauten, aber es ist in der Praxis üblich, den Konkatenationsoperator weg zu lassen.

Welches Muster beschreibt nun dieser reguläre Ausdruck, d.h. wie ist die Sprache $L(\text{REXP})$ definiert?

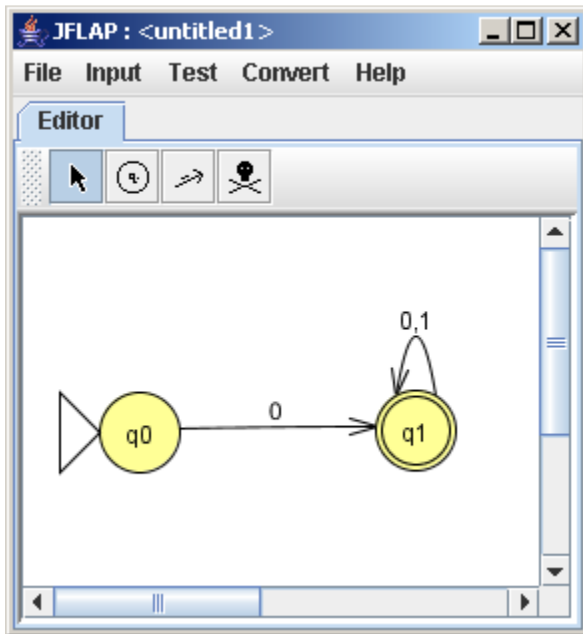
Analysiert man die Klammerung, so erkennt man die beiden Teilausdrücke 0 und $(0|1)^*$ die konkateniert werden. D.h. alle erzeugten Wörter bestehen aus einer 0 gefolgt vom Ausdruck $(0|1)^*$. Dieser Ausdruck wiederum bedeutet 0 oder 1 und zwar bedingt durch den Sternoperator beliebig oft wiederholt. Wichtig dabei ist, dass beliebig oft auch kein Vorkommen inkludiert, d.h. es handelt sich um null bis unendlich viele Wiederholungen.

Der reguläre Ausdruck $(0(0|1)^*)$ beschreibt nun also alle Bitfolgen die mit 0 beginnen, gefolgt von einer beliebigen Anzahl von 0 oder 1 (auch kein Vorkommen).
Die Sprache $L(\text{REXP}) = \{0, 00, 01, 000, 001, 010, 011, \dots\}$.

Wichtig bei der Auswertung von regulären Ausdrücken ist auf die Klammerung und speziell auf die Vorrangregeln zu achten. Lässt man im vorigen Beispiel die Klammern weg, erhält man folgenden Ausdruck REXP2 $(00|1)^*$ der eine komplett andere Sprache definiert. Da der Oder-Operator die niedrigste Priorität aufweist, bedeutet dieser Ausdruck nun 00 oder 1^* also das Wort 00 oder eine beliebige (auch keine) Wiederholung von 1 .
Somit wäre die Sprache $L(\text{REXP2}) = \{00, \varepsilon, 1, 11, 111, 1111, \dots\}$.

In diesem Kapitel haben wir festgestellt, dass reguläre Ausdrücke und endliche Automaten äquivalent sind, d.h. es müsste uns ja ohne Probleme gelingen, einen DEA zu entwerfen, der ausgehend von der REXP $(0(0|1)^*)$ die Sprache $L(\text{REXP})$ akzeptiert. Für dieses einfache Beispiel ist dies auch wirklich ohne viel Aufwand möglich. Im Allgemeinen kann die Transformation zwischen DEA und regulären Ausdrücken jedoch komplexer sein. Details dazu und genaue Verfahren finden sich z.B. in [Hopcroft, 2002]. In unserem Beispiel reicht jedoch ein einfacher Automat mit zwei Zuständen, der in folgenden Abbildung 4-1 dargestellt ist.

³ REXP steht für Regular Expression und findet sich oft in der Literatur als Abkürzung für einen regulären Ausdruck

Abbildung 4-1 DEA für REXP $0(0|1)^*$

Erweiterte Notationen

In der Praxis werden häufig erweiterte Notationen für reguläre Ausdrücke verwendet, d.h. allerdings nicht dass die Mächtigkeit dieser Notationen größer ist und damit komplexere Sprachen erzeugt werden können, es handelt sich einfach um eine einfachere und abkürzende Schreibweise. Es folgt eine Auflistung der wichtigsten regulären Ausdrücke, wie sie z.B. auch im UNIX Tool grep verwendet werden:

x	Eine Instanz von x
.	Ein beliebiges Zeichen (außer newline)
^x	Jedes Zeichen außer x
\x	Metazeichen (wie Punkt, Klammer, ...) müssen mit einem \ vorangestellt angegeben werden, wenn sie im Muster selbst vorkommen sollen
[x]	Alle Zeichen in diesem Bereich (z. B. [abuv] die Buchstaben a, b, u oder v, [a-z] alle Kleinbuchstaben)
()	runde Klammern dienen für die Gruppierung
	der OR Operator (Auswahl) (a A)
{x}	Der Ausdruck muss genau x mal vorkommen
{x,}	Der Ausdruck muss mindestens x mal vorkommen
{x,y}	Der Ausdruck kommt mindestens x mal und höchstens y mal vor
?	Abkürzung für {0, 1}
*	Abkürzung für {0, }
+	Abkürzung für {1, }
^	Start einer neuen Zeile
\$	Ende der Zeile

Eine gute Möglichkeit zur Überprüfung und zum Debugging von regulären Ausdrücken bieten diverse Onlinetools wie z.B. <https://regex101.com/> oder <http://regexr.com/>.

4.3 Grammatiken

Wiederholen wir noch einmal die uns bisher bekannten Möglichkeiten, formale Sprachen zu definieren. Wir haben auf der einen Seite endliche Automaten kennen gelernt, die eine reguläre Sprache durch Akzeptieren ihrer Wörter spezifizieren. Ein regulärer Ausdruck (regular expression) auf der anderen Seite ist ein Muster für die Wörter einer regulären Sprache und somit ein beschreibendes Konzept für reguläre Sprachen. Mit Grammatiken gibt es jedoch noch eine weitere Variante:

Eine Grammatik gibt Regeln an, mit denen aus einem Startwort alle Wörter der Sprache hergeleitet werden können, somit ist eine Grammatik als ein erzeugendes Konzept für formale Sprachen zu sehen. Ein spezieller Typ von Grammatiken, mit denen wir uns vorerst beschäftigen, sind so genannte Typ 3 bzw. reguläre Grammatiken, die genau reguläre Sprachen erzeugen können und somit äquivalent zu endlichen Automaten und regulären Ausdrücken sind. Der Beweis dazu findet sich z.B. wieder in [Vossen, 2004]. Abbildung 4-2 fasst die Darstellungsformen für reguläre Sprachen zusammen.

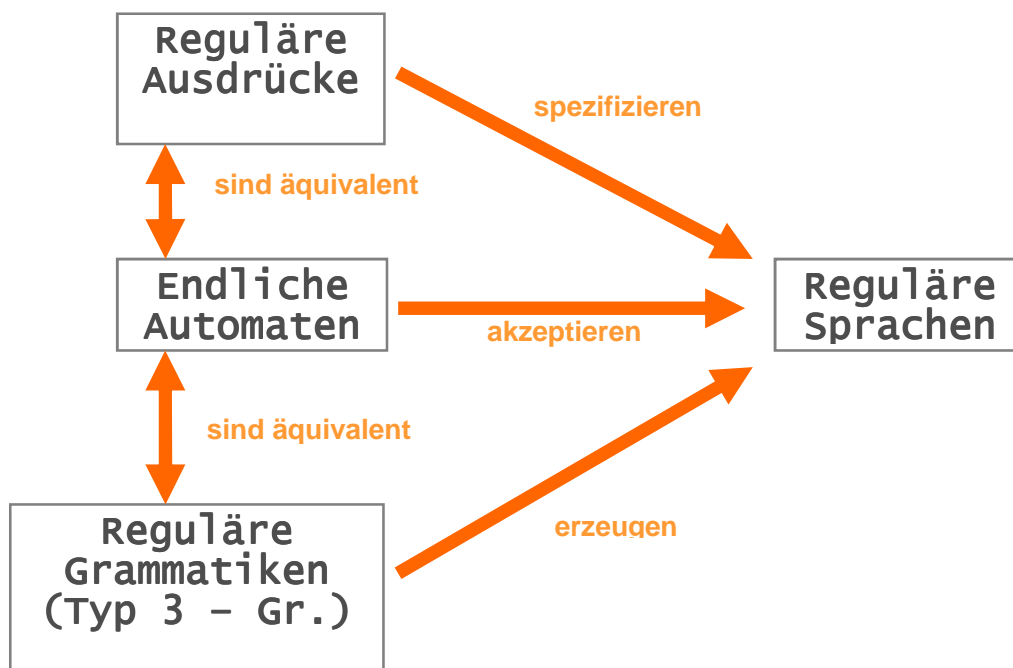


Abbildung 4-2 Beschreibungsformen für reguläre Sprachen

Definition

Eine **Grammatik G** ist formal wie folgt definiert:

$G = (\Sigma, N, P, S)$ bestehend aus

- ♦ einem **Terminalalphabet** Σ .
- ♦ einem von Σ disjunkten⁴ Alphabet von **Nonterminalen** bzw. Variablen N .
- ♦ einer Menge P von **Produktionen** bzw. Ersetzungsregeln.
- ♦ einem **Startsymbol** aus der Menge der Nonterminalen $S \in N$.

⁴ Zwei Mengen heißen disjunkt, wenn ihre Schnittmenge leer ist, in unserem Fall ist $\Sigma \cap N = \emptyset$.

Die Elemente $p = (l, r) \in P$ heißen Produktionen oder Regeln, wobei l die linke Seite von p und r die rechte Seite definiert. Normalerweise wird eine Regel in der Form $l \rightarrow r$ notiert und bedeutet umgangssprachlich „ l geht über in r “ oder „ l wird ersetzt durch r “.

Betrachten wir die Bedeutung der einzelnen Teile einer Grammatik anhand eines Beispiels:

Gegeben ist die Grammatik $G = (\{0, 1\}, \{S, A, B\}, P, S)$ mit

$$P = \{ \begin{array}{l} S \rightarrow 0S, S \rightarrow 1S, S \rightarrow 0A \\ A \rightarrow 0B, A \rightarrow 1B, \\ B \rightarrow 0, B \rightarrow 1 \end{array} \}$$

Entsprechend der Definition besteht diese Grammatik also aus den Terminalsymbolen 0 und 1, aus der disjunkten Menge von Nonterminalsymbolen bzw. Variablen S , A und B , der Regelmengengruppe P und dem Startsymbol S aus der Menge der Nonterminale.

Ableitung und Sprache

Da eine Grammatik ein erzeugendes Konzept für eine formale Sprache darstellt, stellt sich nun die Frage, wie genau nun Wörter mit ihrer Hilfe erzeugt werden. Ausgehend vom Startsymbol wird ein Wort durch eine Folge von Regelanwendungen erzeugt. Diesen Vorgang nennt man **Ableitung**. Pro **Ableitungsschritt** wird ein Nonterminalsymbol durch Anwenden einer Produktionsregel, in der dieses Nonterminal auf der linken Seite vorkommt, durch die rechte Seite dieser Regel ersetzt. Ein Ableitungsschritt wird dabei mit \Rightarrow bezeichnet.

In unserem Beispiel beginnen wir mit dem Startsymbol S und suchen eine Produktionsregel, in der S auf der linken Seite vorkommt. Drei solche Regeln existieren und wir wählen z.B. die Regel $S \rightarrow 0S$ aus. Wenden wir diese Regel an, so müssen wir S durch $0S$ ersetzen und erhalten damit das abgeleitete Wort $0S$. In diesem Wort kommt allerdings wieder das Nonterminalsymbol S vor und wir können diesmal z.B. die Regel $S \rightarrow 1S$ anwenden und ersetzen somit S durch $1S$. Nochmaliges Anwenden der Regel erzeugt insgesamt das Wort $011S$. Wenden wir nun die Regel $S \rightarrow 0A$ an, entsteht das Wort $0110A$. Jetzt benötigen wir eine Regel, die das Nonterminal A auf der linken Seite aufweist, z.B. die Regel $A \rightarrow 1B$ und erhalten dadurch das abgeleitete Wort $01101B$. Wenden wir abschließend die Regel $B \rightarrow 0$ an, entsteht das sogenannte Terminalwort 011010 , das kein Nonterminalsymbol mehr enthält und damit auch keine Regel mehr anwendbar ist. Der gesamte Ableitungsprozess wird wie folgt zusammengefasst:

S	$\Rightarrow 0S$	mit Regel $S \rightarrow 0S$
	$\Rightarrow 01S$	mit Regel $S \rightarrow 1S$
	$\Rightarrow 011S$	mit Regel $S \rightarrow 1S$
	$\Rightarrow 0110A$	mit Regel $S \rightarrow 0A$
	$\Rightarrow 01101B$	mit Regel $A \rightarrow 1B$
	$\Rightarrow 011010$	mit Regel $B \rightarrow 0$

Der Ableitungsprozess lässt sich also solange fortsetzen, bis das bis dahin abgeleitete Wort kein Nonterminalsymbol mehr enthält und damit keine Regel mehr anwendbar ist wie in obigen Beispiel. Dann gehört das abgeleitete Wort zur von G definierten Sprache $L(G)$. Es kann aber auch der Fall sein, dass das bis dahin abgeleitete Wort ein Nonterminalsymbol enthält, zu dem keine Regel mehr existiert, wo dieses Symbol auf der linken Seite vorkommt, dann muss der Ableitungsprozess abgebrochen werden und das abgeleitete Wort ist kein Element von $L(G)$.

Die **Sprache $L(G)$** der Grammatik G über dem Terminalalphabet Σ ist nun definiert als die Menge aller **Terminalwörter**, die aus dem Startsymbol in **endlich vielen Schritten abgeleitet** werden können.

Reguläre Grammatiken

Eine Grammatik G , die nur Produktionsregeln der Form $A \rightarrow aB$ aufweist, heißt **rechtslinear**, weil die Wörter von links nach rechts buchstabenweise erzeugt werden, wie aus dem vorigen Beispiel ersichtlich ist. In jedem Ableitungsschritt kommt höchstens ein Terminal dazu, und das auch nur am rechten Ende des abgeleiteten Wortes. Es existiert jedoch auch der umgekehrte Fall, nämlich die linkslineare Grammatik.

Eine Grammatik G , die nur Produktionsregeln der Form $A \rightarrow Ba$ aufweist, heißt **linkslinear**, weil die Wörter von rechts nach links buchstabenweise erzeugt werden.

Rechts- und linkslineare Grammatiken sind allerdings äquivalent, der Beweis findet sich z.B. in [Vossen, 2004]. Beide Arten gemeinsam heißen Typ-3 oder reguläre Grammatiken. Pro Ableitungsschritt wird höchstens ein Terminalsymbol erzeugt.

Es lässt sich weiters beweisen, dass Typ-3 Grammatiken, endliche Automaten und reguläre Ausdrücke äquivalent sind und alle drei Methoden reguläre Sprachen definieren. Es kommt in der Praxis einfach auf die Aufgabenstellung an, welche Darstellungsform geeignet ist. Am anschaulichsten und am besten für den Menschen nachvollziehbar ist sicherlich die graphische Darstellung eines endlichen Automaten, die kompakteste Notation bietet jedoch eindeutig ein regulärer Ausdruck.

Uns interessiert aber jetzt die Fragestellung ob es auch Sprachen gibt, die nicht regulär sind. Oder anders ausgedrückt, gibt es Sprachen, die nicht von endlichen Automaten akzeptiert werden können oder durch reguläre Ausdrücke bzw. reguläre Grammatiken definiert werden können? Die Antwort wird das folgende Kapitel liefern.

4.4 Kontextfreie Sprachen

Grenzen regulärer Sprachen

Rufen wir uns noch einmal die wesentlichen Eigenschaften von endlichen Automaten in Erinnerung. Wie der Name schon deutlich macht, besitzt ein endlicher Automat nur endlich viele Zustände (hat ein endliches Gedächtnis) und kann sich also nur endlich viele Situationen merken. Ob diese Einschränkung relevant ist, wollen wir anhand folgender Sprache untersuchen:

Gegeben ist die Sprache $L = \{0^n 1^n \mid n \geq 0\}$ über dem binären Alphabet $\Sigma = \{0, 1\}$.

L besteht aus allen Wörtern mit einer beliebigen Folge von Nullen gefolgt von der gleichen Anzahl an Einsen.

Ein endlicher Automat, der L akzeptiert, müsste sich die Anzahl der Nullen merken um danach zu überprüfen, ob dieselbe Anzahl an Einsen existiert. Da diese Anzahl aber unendlich groß werden kann, ist es unmöglich diesen Sachverhalt auf endlich viele Zustände abzubilden. Es gelingt also nicht, einen endlichen Automaten zu definieren, der L akzeptiert. Aus der Tatsache, dass endliche Automaten, reguläre Ausdrücke und reguläre Grammatiken äquivalent sind, lässt sich auch zeigen,

dass für L weder ein regulärer Ausdruck noch eine reguläre Grammatik angegeben werden können. Somit ist diese Sprache nicht regulär! Ein formaler Beweis findet sich z.B. in [Vossen, 2004].

Auf den ersten Blick ist diese Erkenntnis nicht weiter dramatisch, und die Sprache $L = \{0^n 1^n \mid n \geq 0\}$ wirkt auch nicht besonders interessant. Bei genauerer Betrachtung hat die Sprache L jedoch einige Anwendungen in der Informatik. Ersetzt man 0 durch eine geöffnete Klammer und 1 durch eine geschlossene Klammer, können durch L geklammerte arithmetische Ausdrücke oder Blockstrukturen in Programmiersprachen repräsentiert werden. So müssen in einem syntaktisch korrekten C Programm z.B. gleich viele geöffnete geschwungene Klammern $\{$ wie geschlossene geschwungene Klammern $\}$ vorkommen und der C-Compiler muss dies überprüfen. Wir wissen aber, dass L nicht regulär ist und daher nicht von einem endlichen Automaten überprüft werden kann. Daraus können wir folgern, dass ein Compiler mächtiger als endliche Automaten bzw. reguläre Sprachen sein muss! Wie können solche Sprachen aber nun repräsentiert werden? Die Antwort liefern die folgenden Kapitel.

Kontextfreie Grammatiken

Hebt man die Einschränkung von Typ-3 Grammatiken auf, dass eine Regel nur ein Terminalsymbol erzeugen darf, so erhält man **kontextfreie Grammatiken** (Typ-2 Grammatiken). Die linke Seite der Regeln besteht aber nach wie vor aus genau einem Nonterminalsymbol. In einem Ableitungsschritt wird daher ein Nonterminal ohne Rücksicht auf die benachbarten Zeichen, daher der Name kontextfrei, durch die rechte Seite der Regel ersetzt. Mit dieser Grammatik gelingt es uns nun, eine Regel zu definieren, die die Sprache L aus dem vorigen Kapitel erzeugen kann, da wir gleichzeitig eine 0 und eine 1 erzeugen dürfen.

Gegeben ist die folgende Grammatik:

$G = (\{0,1\}, \{S\}, \{S \rightarrow 0S1, S \rightarrow \varepsilon\}, S)$

Mit der Regel $S \rightarrow 0S1$ erzeugen wir pro Ableitungsschritt genau eine 0 und eine 1 und das abgeleitete Wort wächst nach mehrmaliger Anwendung der Regel nach links und nach rechts, wobei die Nullen immer vor den Einsen bleiben und die Anzahl der Nullen und der Einsen immer identisch bleibt. Im Folgenden ist der Ableitungsprozess dargestellt:

$S \Rightarrow 0S1$
 $\Rightarrow 00S11$
 $\Rightarrow 000S111$
...

Um ein Wort abzuleiten benötigen wir noch die zweite Regel $S \rightarrow \varepsilon$ zur Terminierung, um das Nonterminal S in der Mitte zu entfernen. Es ist leicht zu erkennen, dass die erzeugte Sprache dieser Grammatik $L(G)$ genau unserer gesuchten Sprache $\{0^n 1^n \mid n \geq 0\}$ entspricht.

Eine Sprache L heißt **kontextfrei**, wenn es eine zugehörige kontextfreie Grammatik gibt.

Um die Mächtigkeit von kontextfreien Sprachen zu verdeutlichen, schauen wir uns noch ein weiteres praxisrelevantes Beispiel an. Gesucht ist eine Grammatik für eine Sprache von arithmetischen Ausdrücken. Um die Sprache einfach zu halten verwenden wir nur folgende Terminalsymbole:

- ♦ a (stellvertretend für alle Variablen und Zahlen)
- ♦ Operatoren $+, -, *, /$
- ♦ Klammern $($ und $)$

Wörter dieser Sprache wären zum Beispiel:

- ◆ a
- ◆ a+a
- ◆ a*(a-a)
- ◆ (a+a)*(a-a)
- ◆ (a+a)/(a*(a-a))
- ◆ ...

Als Nonterminale verwenden wir E (für Expression) und O (für Operator). Die Grammatik kann nun wie folgt definiert werden (nach [Vossen, 2004]):

Grammatik $G = (\{a, +, -, *, /, (,)\}, \{E, O\}, P, E)$

$$P = \left\{ \begin{array}{l} E \rightarrow a \mid E O E \mid (E), \\ O \rightarrow + \mid - \mid * \mid / \end{array} \right\}$$

Der senkrechte Strich | in den Regeln bedeutet lediglich eine Kurzschreibweise und entspricht dem Oder in regulären Ausdrücken. So bedeutet die Regel $O \rightarrow + \mid - \mid * \mid /$ z.B. O geht über in +, -, * oder / und dient somit zum Erzeugen der Operatoren. Die Regel $E \rightarrow a$ erzeugt genau einen Bezeichner a, die Regel $E \rightarrow E O E$ verknüpft zwei Expressions mit einem Operator und die Regel $E \rightarrow (E)$ klammert eine Expression.

Betrachten wir z.B. die Ableitung von $(a+a)*(a-a)$ mit dieser Grammatik:

E	=> E O E	mit Regel $E \rightarrow E O E$
	=> E * E	mit Regel $O \rightarrow *$
	=> (E) * E	mit Regel $E \rightarrow (E)$
	=> (E) * (E)	mit Regel $E \rightarrow (E)$
	=> (E O E) * (E)	mit Regel $E \rightarrow E O E$
	=> (E O E) * (E O E)	mit Regel $E \rightarrow E O E$
	=> (E + E) * (E O E)	mit Regel $O \rightarrow +$
	=> (E + E) * (E - E)	mit Regel $O \rightarrow -$
	=> (a + a) * (a - a)	mit mehrmaliger Anwendung der Regel $E \rightarrow a$

Als grafische Veranschaulichung einer Ableitung dienen so genannte **Ableitungsbäume** (Parse Trees). Der Ableitungsbaum zu $(a+a)*(a-a)$ ist in Abbildung 4-3 auf der folgenden Seite ersichtlich. Hier werden beginnend beim Startsymbol E die angewandten Ersetzungsregeln schrittweise dargestellt.

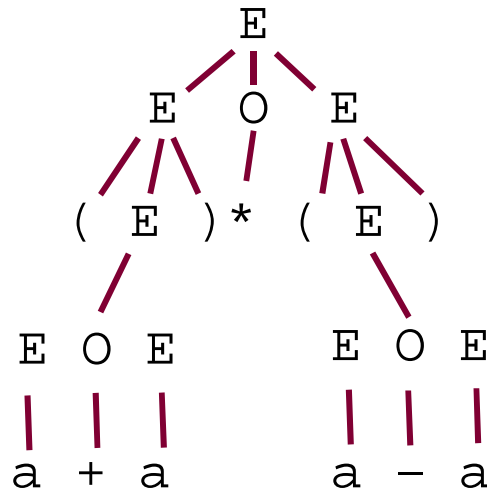


Abbildung 4-3 Ableitungsbaum

Mit Hilfe von Ableitungsbäumen lässt sich auch noch eine weitere interessante Eigenschaft von kontextfreien Grammatiken visualisieren.

Eine Grammatik heißt **mehrdeutig**, falls mehrere Ableitungen zum selben Ergebnis führen bzw. wenn es für mindestens ein Wort mindestens zwei verschiedene Ableitungsbäume gibt.

Betrachten wir das Wort $a * a + a$. In Abbildung 4-4 sehen wir zwei verschiedene Ableitungsbäume für dieses Wort. Unsere Grammatik für arithmetische Ausdrücke ist daher mehrdeutig, was in der Praxis zu Problemen führt und vermieden werden sollte. Entwirft man z.B. eine Grammatik für eine Programmiersprache, so sollte diese eindeutig sein, da der Compiler eine eindeutige Syntax benötigt um Ausdrücke weiter zu verarbeiten. Im Beispiel $a * a + a$ sollte immer zuerst die Multiplikation und erst danach die Addition ausgewertet werden, um die bekannte Regel „Punkt- vor Strichrechnung“ einzuhalten.

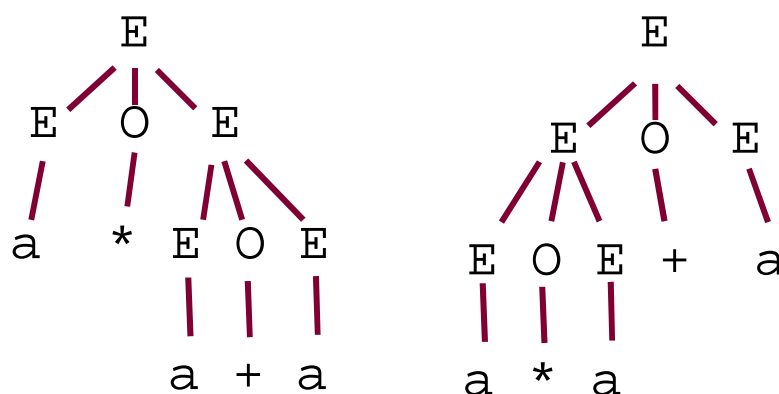


Abbildung 4-4 Mehrdeutige Grammatik

Kellerautomaten

Analog zu regulären Sprachen, die durch endliche Automaten akzeptiert werden gibt es auch Automaten, die kontextfreie Sprachen akzeptieren, die so genannten Kellerautomaten.

Ein Kellerautomat funktioniert grundlegend analog zu endlichen Automaten, besitzt aber einen weiteren Speicher, den Keller bzw. Stack mit beliebig großer Kapazität. Der Zugriff auf diesen Speicher ist allerdings beschränkt. Nur das zuletzt eingefügte Symbol (das oberste) kann gelesen oder verändert werden. Dieses Zugriffsprinzip heißt LIFO Prinzip (Last In-First Out) und wird uns bei den Datenstrukturen noch genauer beschäftigen. Weitere Details und Beispiele zu Kellerautomaten finden sich in [Vossen, 2004], für uns reicht das Wissen von der Existenz und eine genauere Analyse würde den Rahmen der Lehrveranstaltung und dieses Skriptums sprengen. Interessanter sind da schon mögliche Anwendungen und weitere Darstellungsformen von kontextfreien Grammatiken, die in den nächsten beiden Kapiteln behandelt werden.

Anwendungen

Wie bereits mehrmals erwähnt, ist ein Hauptanwendungsgebiet von formalen Sprachen und Grammatiken der **Compilerbau**. Ableitungsbäume, wie zuvor beschrieben, dienen hierbei als interne Repräsentation eines Programms. Daraus kann ein Compiler den zugehörigen Object Code generieren. Kontextfreie Grammatiken beschreiben somit die Syntax von gängigen Programmiersprachen.

Eine weitere Anwendung kontextfreier Grammatiken findet sich in der **Extensible Markup Language** (XML), einer Sprache für Datenaustausch. Zulässige Inhalte eines XML Dokuments, also die Zusammensetzung und Schachtelung der Tags, beschreibt eine so genannte Document Type Definition (DTD), die im Wesentlichen einer kontextfreien Grammatik entspricht. Weitere Details zu XML und DTDs werden in der Lehrveranstaltung Webtechnologien erläutert.

Eine interessante Anwendung von kontextfreien Grammatiken in einem ganz anderen Bereich sind die so genannten **L-Systeme**. Diese Grammatiken wurden von A. Lindenmayer ursprünglich entwickelt zur Modellierung von Wachstumsvorgängen in biologischen Organismen (Pflanzen, Bäume, ...). Im Gegensatz zu herkömmlichen Grammatiken handelt es sich bei einem L-System um eine Grammatik mit parallelem Ersetzungssystem, d.h. in einem Schritt werden gleichzeitig alle anwendbaren Regeln verwendet. Zusätzlich können Regeln z.B. durch Zufallszahlen beeinflusst werden um Umwelteinflüsse auf benachbarte Pflanzen zu modellieren und Variationen zu erzeugen. Anwendungen finden sich z.B. in der 3D-Computergrafik, wo computergenerierte Pflanzen und Waldszenen nicht händisch modelliert sondern mit L-Systemen definiert und erzeugt werden. Ein Beispiel hierfür ist in Abbildung 4-5 auf der folgenden Seite zu sehen.

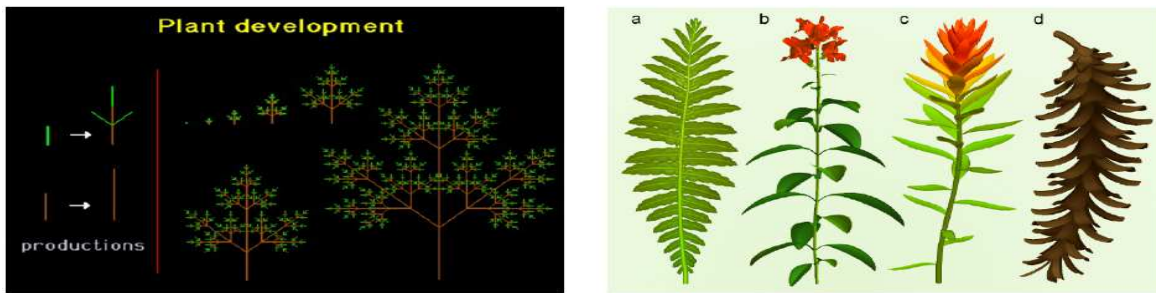


Abbildung 6 Die Ersetzungsregeln (productions) sind bei L-Systemen nicht einfach nur Symbole, sondern werden als Teile von Pflanzen interpretiert.



Abbildung 7 Ein computergenerierter Wald, der nicht händisch modelliert wurde, sondern mit L-Systemen definiert und erzeugt wurde.

Abbildung 4-5 Anwendung von L-Systemen

Erweiterte Backus-Naur-Form und Syntaxdiagramme

Eine effiziente Darstellungsform für kontextfreie Grammatiken, die in der Praxis häufig zur Definition von Programmiersprachen verwendet wird, bildet die erweiterte Backus-Naur-Form (EBNF). Die EBNF wurde von J. Backus und P. Naur, zwei Informatikern die in den fünfziger und sechziger Jahren Programmiersprachen wie FORTRAN und ALGOL definiert haben, entwickelt.

In der EBNF Notation werden Produktionsregeln der Grammatik um Elemente der regulären Ausdrücke erweitert, um dadurch eine kompaktere Schreibweise zu erhalten. Folgende Elemente sind dabei möglich:

- ◆ Alternativen: $a|b$
- ◆ Klammerung: (a)
- ◆ Wiederholung: $\{a\}$ oder a^*
- ◆ Optional: $[a]$

Weiters werden EBNF Regeln mit $l ::= r$ statt der bekannten Schreibweise $l \rightarrow r$ notiert.

Eine Grammatik für Integer Zahlen in EBNF Notation könnte wie folgt definiert sein, nach [Vossen, 2004]:

$G = (\{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{\text{int}, \text{vz}, \text{zf}, \text{z1}, \text{z0}\}, P, \text{int})$

$P = \{$

int	::=	vz zf,
vz	::=	[+ -],
zf	::=	(0 z1 z0*),
z1	::=	(1 2 3 4 5 6 7 8 9),
z0	::=	(0 z1)

$\}$

Ein Integer (int) ist also ein Vorzeichen (vz) gefolgt von einer Ziffernfolge (zf). Ein Vorzeichen ist entweder + oder - oder keins von beiden, eckige Klammern in der Regel bedeuten ja optional. Eine

Ziffernfolge ist 0 oder z1 gefolgt von beliebig vielen z0. z1 definiert Ziffern von 1 bis 9, also ohne 0. z0 definiert Ziffern von 0 bis 9. Diese Unterscheidung ist notwendig, da wir führende Nullen nicht akzeptieren wollen, d.h. das Wort 0123 wäre nicht Element von $L(G)$, das Wort 0 jedoch schon.

Syntaxdiagramme bieten eine grafische Darstellung von kontextfreien Grammatiken und EBNF Elementen. Dabei entsprechen Rechtecke Nonterminalen und Ellipsen bzw. Kreise sind Terminalsymbole. Jeder endliche Durchlauf entlang der Kanten des Diagramms (den verbindenden Geraden) ergibt eine endliche Zeichenfolge die der Syntax der spezifizierten Sprache entspricht. Abbildung 4-6 zeigt das Syntaxdiagramm für die oben definierte Grammatik für Integer Zahlen.

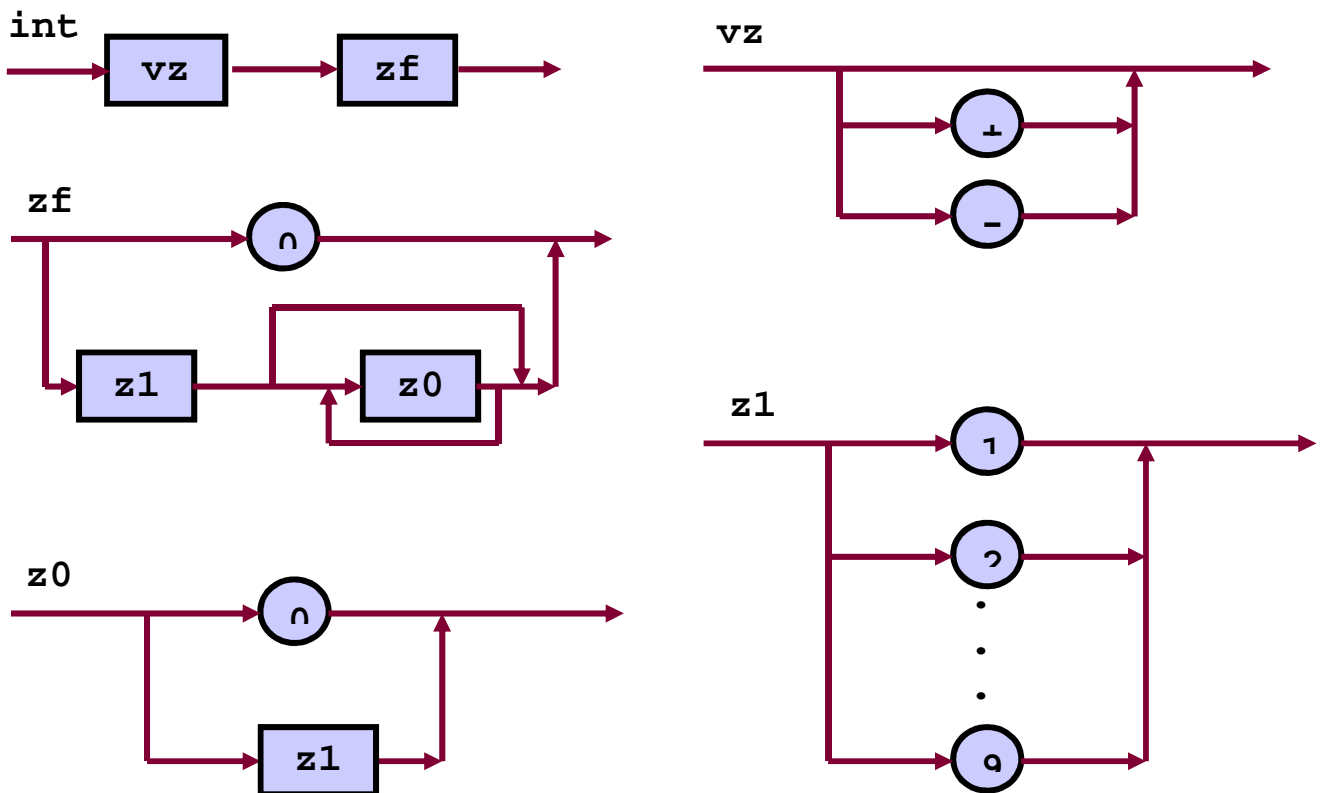


Abbildung 4-6 Syntaxdiagramm für eine EBNF Grammatik

4.5 Sprachklassen

Wir haben uns bis jetzt mit regulären und kontextfreien Sprachen beschäftigt, die jeweils durch reguläre (Typ-3) und kontextfreie (Typ-2) Grammatiken definiert werden können. Die Vermutung liegt nahe, dass es noch weitere Sprachklassen gibt und nicht alle formalen Sprachen regulär oder kontextfrei sind.

Hebt man die Beschränkung von kontextfreien Grammatiken auf, dass die linken Seiten von Produktionsregeln nur aus genau einem Nonterminalsymbol bestehen dürfen, so erhält man die noch mächtigeren Sprachklassen der **kontextsensitiven** (Typ-1) und der **rekursiv-aufzählbaren** (Typ-0) Sprachen. In solchen Grammatiken dürfen also die linken Seiten einer Regel aus Terminal- und Nonterminalsymbolen bestehen und das Nonterminal darf nur durch die rechte Seite ersetzt werden, wenn es im entsprechend der linken Seite vorgegebenen Kontext auftritt. Wir werden uns aber nicht näher mit diesen Grammatiken beschäftigen sondern interessieren uns für die Auswirkungen auf unsere bekannten Automaten, die die Sprachen akzeptieren. Typ-1 und Typ-0 Sprachen können nicht mehr durch Kellerautomaten akzeptiert werden und wir müssen deren Beschränkung aufheben, nur auf das oberste Element des Kellerspeichers zugreifen zu dürfen. Erlaubt man Zugriff auf jede Zelle des unendlichen Speichers, erhält man eine neue, noch

mächtigere Klasse von Automaten, die **Turing Maschine**. Da dieser Automat einen besonderen Stellenwert in der Informatik hat und auch eine gute Überleitung zum Begriff Algorithmus bietet, ist den Turing Maschinen mit Kapitel 5 ein eigenes Kapitel gewidmet.

Chomsky-Hierarchie

Zuvor sollen aber noch einmal alle Sprachklassen zusammengefasst werden. 1958 hat der amerikanische Linguist Noam Chomsky vier Einschränkungen von Regelstrukturen von Grammatiken vorgeschlagen und damit die nach ihm benannte Chomsky-Hierarchie der Typ-0, Typ-1, Typ-2 und Typ-3 Sprachen definiert. Jede Sprache höheren Typs ist in der Hierarchie Teilmenge der mächtigeren niedriger bezifferten Typen. So ist z.B. jede reguläre (Typ-3) Sprache auch eine kontextfreie (Typ-2) Sprache, nicht aber umgekehrt. Weiters ist jedem Sprachtyp auch ein entsprechender Automatentyp zugeordnet, wie in der folgenden Tabelle ersichtlich ist, die die Chomsky Hierarchie zusammenfasst.

Typ	Automat	Sprache	Grammatik
3	Endlicher Automat	Reguläre Sprachen	Reguläre Grammatik
2	Kellerautomat	Kontextfreie Sprachen	Kontextfreie Grammatik
1	Turing Maschine	Kontextsensitive Sprachen	Kontextsensitive Grammatik
0	Turing Maschine	Rekursiv aufzählbare Sprachen	Typ 0 Grammatik

4.6 Selbstlernkontrolle

Fragen

Durch welches geordnete Paar wird eine Grammatik definiert?

- (a) 1. ein Bandalphabet
2. ein Eingabealphabet
3. eine Menge von Isomorphismen
4. ein Startsymbol
- (b) 1. ein Terminalalphabet
2. ein Nonterminalalphabet
3. eine Menge von Isomorphismen
4. ein Startsymbol
- (c) 1. ein Terminalalphabet
2. ein Nonterminalalphabet
3. eine Menge von Produktionen
4. ein Startsymbol

Welche der folgenden Aussagen sind wahr?

- (a) Zu jedem endlichen Automaten gibt es auch eine reguläre Grammatik.
- (b) Alle Sprachen in der Informatik sind regulär.
- (c) Zu jedem regulären Ausdruck kann ein endlicher Automat definiert werden.
- (d) Reguläre Sprachen sind mächtiger als kontextfreie Sprachen.
- (e) Zur Überprüfung der Syntax einer Programmiersprache reichen reguläre Ausdrücke nicht aus.
- (f) Syntaxdiagramme bieten eine grafische Darstellung von kontextfreien Grammatiken.

Lösungen

Durch welches geordnete Paar wird eine Grammatik definiert?

- (a) 1. ein Bandalphabet
2. ein Eingabealphabet
3. eine Menge von Isomorphismen
4. ein Startsymbol
- (b) 1. ein Terminalalphabet
2. ein Nonterminalalphabet
3. eine Menge von Isomorphismen
4. ein Startsymbol
- (c) **1. ein Terminalalphabet**
2. ein Nonterminalalphabet
3. eine Menge von Produktionen
4. ein Startsymbol

Welche der folgenden Aussagen sind wahr?

- (a) **Zu jedem endlichen Automaten gibt es auch eine reguläre Grammatik.**
- (b) Alle Sprachen in der Informatik sind regulär.
- (c) **Zu jedem regulären Ausdruck kann ein endlicher Automat definiert werden.**
- (d) Reguläre Sprachen sind mächtiger als kontextfreie Sprachen.
- (e) **Zur Überprüfung der Syntax einer Programmiersprache reichen reguläre Ausdrücke nicht aus.**
- (f) **Syntaxdiagramme bieten eine grafische Darstellung von kontextfreien Grammatiken.**

4.7 Übungsteil

Übung 1: Reguläre Ausdrücke

Zwei Beispiele für Reguläre Ausdrücke aus dem täglichen Leben sind das Autokennzeichen und die Email-Adresse. Für beide gibt es Regeln, denen die einzelnen Nummern bzw. Adressen genügen müssen.

Bringen Sie mindestens zwei weitere Beispiele für reguläre Ausdrücke und stellen Sie diese als REXP dar.

Beispiel: E-mail-Adresse

[A-Za-z0-9]{1,} @ [A-Za-z0-9]{1,} \. [a-z]{1,}

weitere Beispiele:

[a-zA-Z]* Beliebige Zeichen aus Buchstaben (z. B. ugHrB).

[A-Z0-9]{8} Acht Zeichen aus A bis Z und 0 bis 9, (z. B. RX6Z45UB).

[A-Z]([a-z])+ Ein Grossbuchstabe gefolgt von mindestens 1 Kleinbuchstaben (z.B. Stu).

([0-9]-){2}[0-9] Drei Zahlen, durch Striche getrennt (z. B. 2-1-8).

[A-G]{2,} Mindestens zwei Grossbuchstaben aus A bis G (z. B. BGA).

[bRxv]{3} Drei Buchstaben aus b, R, x und v (z. B. xxv).

Übung 2

Gegeben ist das Alphabet $\Sigma = \{0, 1\}$ und folgender reguläre Ausdruck
REXP: $((0|1)^1)^*$.

- 1) Nennen Sie drei Wörter dieser Sprache.
- 2) Zeichnen Sie das Zustandsdiagramm eines deterministischen endlichen Automaten, der diese Sprache akzeptiert.
- 3) Erstellen Sie die Übergangstabelle des Automaten
- 4) Aus wie vielen Wörtern besteht die Sprache dieses Automaten?

Übung 3: Grammatiken

In der Programmiersprache C dürfen Bezeichner (z.B. Variablennamen und Funktionsnamen) an der ersten Stelle keine Ziffer haben. Weiters ist es üblich, dass Bezeichner durchgehend klein geschrieben werden.

Erstellen Sie eine Grammatik $G(L)$ für $L :=$ Menge aller Folgen von Kleinbuchstaben und Ziffern, die mit einem Buchstaben anfangen

Hinweis:

$\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$

Verwenden sie Abkürzungen wie z.B. $\text{letter} := a | b | c | \dots | z$ und $\text{digit} := 0 | 1 | 2 | \dots | 9$

Übung 4

Ein wohlgeformter Klammerterm besteht aus gleich viel öffnenden und schließenden Klammern. An keiner Stelle darf die Zahl der schließenden Klammern die der bisher geöffneten Klammern überschreiten. z.B. $(())$, $()()$, $((())())$

Erstellen Sie eine Grammatik $G(L)$ für $L :=$ Menge der wohlgeformten Klammerterme.

Hinweis:

$\Sigma = \{ (,) \}$ und $N = \{ S \}$

Übung 5

a) Welche Sprache kann von der folgenden Grammatik erzeugt werden?

$G = \{ \{a,b\}, \{S,A,B\}, P, S \}$
 $P = \{ p_1, p_2, p_3, p_4, p_5 \}$
mit $P = \{$ $p_1 = S \rightarrow AB,$
 $p_2 = A \rightarrow aA, p_3 = A \rightarrow \epsilon,$
 $p_4 = B \rightarrow Bb, p_5 = B \rightarrow \epsilon \}$

b) Wie könnte man die Produktionsregeln vereinfachen?

5 Turing Maschinen und Berechenbarkeit (Optional)

Dieses Kapitel dient nur als optionaler Lernstoff zur Vertiefung der Kenntnisse und ist nicht prüfungsrelevant für die Lehrveranstaltung „Grundlagen der Informatik“ im Studiengang Verkehr und Umwelt.

5.1 Einleitung

Eine grundlegende Frage in der Mathematik und der theoretischen Informatik ist, ob es Problemstellungen gibt, die prinzipiell unlösbar sind, d.h. für deren Lösung man keine Formel und kein Computerprogramm angeben kann? Um solche Fragestellungen formal zu untersuchen entwickelte der britische Mathematiker Alan Turing 1936 die Turing Maschine, ein mathematisches Modell einer Maschine für mögliche Berechnungen. Eine solche Turing Maschine ähnelt einem heutigen Computer von der Grundstruktur und ist ein Gerät bestehend aus

- ◆ einer **Kontrolleinheit**,
- ◆ einem **Schreib-/Lesekopf** und
- ◆ einem **unbegrenztem Datenband**

wobei im Gegensatz zu den Kellerautomaten aus Kapitel 4.4 der Zugriff auf jede Speicherzelle erlaubt ist. Entsprechend der Chomsky-Hierarchie können dadurch Typ-0 Sprachen akzeptiert werden.

Beeindruckend daran ist, dass eine Turing Maschine mit den drei Operationen (Lesen, Schreiben und Schreib-/Lesekopf bewegen) die Probleme lösen kann, die auch von einem Computer gelöst werden könnten. Man kann sogar eine programmierähnliche Notation verwenden, um Programme für eine Turing Maschine anzugeben, die diese abarbeiten kann. Sämtliche mathematische Grundfunktionen, wie Addition und Multiplikation lassen sich mit diesen drei Operationen simulieren und aufbauend können natürlich komplexere Problemstellungen abgebildet werden. Wenn wir uns also mit Turing Maschinen beschäftigen, hilft uns das auch, die grundlegenden Konzepte des Programmierens und die Mächtigkeit von Programmiersprachen zu verstehen und das Thema bietet eine ideale Ausgangsbasis zu Algorithmen, dem Kernstoff dieser Lehrveranstaltung. In Folge werden wir Turing Maschinen formal definieren und ein Beispiel betrachten. Weiters werden der Zusammenhang zwischen Turing Maschinen und Algorithmen erläutert sowie schlussendlich die Grenzen der Berechenbarkeit diskutiert.

5.2 Definition und Funktionsweise

Bevor wir eine Turing Maschine formal definieren, betrachten wir die in der Einleitung erwähnten Elemente einer solchen Maschine genauer.

Die **Kontrolleinheit** kann, ähnlich einem endlichen Automaten, endlich viele interne Zustände $s_0, s_1, s_2, s_3, \dots, s_n$ annehmen. Das Band dient als **Eingabe- und Speicherband**. Es ist in einzelne Felder unterteilt, die genau ein Zeichen speichern. Mit dem **Schreib-/Lesekopf** kann die Maschine genau ein Feld lesen bzw. überschreiben. Ferner kann der Schreib-/Lesekopf auf dem Band um eine Position nach links oder rechts gerückt werden. Das Band ist also mit dem Hauptspeicher eines modernen Rechners vergleichbar, wobei dieser natürlich nur eine endliche Größe aufweist. Das ist aber für die praktische Umsetzung einer Turing Maschine kein Hindernis, da das Band durchaus auch endlich sein kann. Es muss einfach nur lang genug sein, um die aktuelle Berechnung ungehindert ausführen zu können, d.h. der Lese- und Schreibkopf darf nicht an das Ende stoßen. Abbildung 5-1 fasst das Maschinenmodell einer Turing Maschine zusammen.

Bei Turing-Maschinen unterscheiden wir im Gegensatz zu endlichen Automaten zwischen einem **Eingabealphabet** und einem **Bandalphabet**. Das Eingabealphabet umfasst die Zeichen, die von „außen“ auf das Band geschrieben wurden, also vor dem Start der Maschine. Das Bandalphabet beinhaltet alle Zeichen die die Maschine lesen und verarbeiten kann. Diese Menge umfasst neben dem Eingabealphabet in jedem Fall noch das Leersymbol, das anzeigt, dass ein Feldinhalt leer, d.h. undefiniert ist. Das Leersymbol wird in der Literatur meist als $*$ oder als $\#$ definiert. Beide Mengen sind natürlich endlich und das Eingabealphabet ist eine Teilmenge des Bandalphabets.

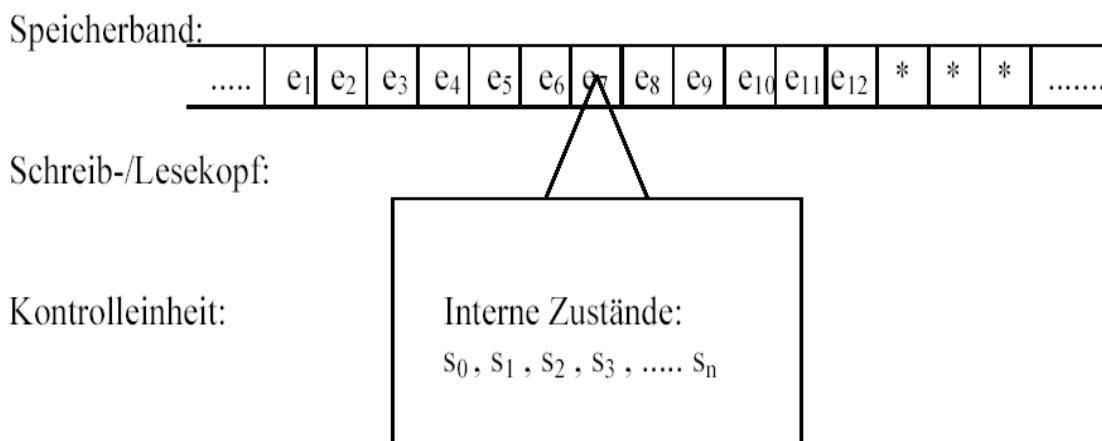


Abbildung 5-1 Maschinenmodell einer Turing Maschine

Nun können wir die formale Definition einer Turing Maschine als Sechstupel $T = (E, B, S, \delta, s_0, F)$ angeben, wobei die einzelnen Komponenten wie folgt definiert sind:

- ◆ Eingabealphabet $E = \{e_1, e_2, e_3, \dots, e_r\}$
- ◆ Bandalphabet $B = \{b_1, b_2, b_3, \dots, b_m\}$, wobei gilt: $E \cup \{*\} \subseteq B$
- ◆ Menge der Zustände $S = \{s_0, s_1, s_2, \dots, s_n\}$
- ◆ Überföhrungsfunktion $\delta: S \times B \rightarrow S \times B \times \{L, R, N\}$
- ◆ Startzustand $s_0 \in S$
- ◆ Menge der finalen Zustände $F \subseteq S$

Die Turing Maschine führt eine Berechnung aus, indem sie schrittweise eine Eingabe in eine Ausgabe umwandelt, wobei alle Ein- und Ausgaben sowie etwaige Zwischenergebnisse auf dem Band gespeichert werden. Zu Beginn steht ein Wort als Eingabe auf dem Band und der Schreib-/Lesekopf befindet sich über dem ersten Zeichen der Eingabe. Der Rest des Bandes ist mit dem

leeren Feld * bzw. # formatiert. Weiters befindet sich die Turing Maschine zu Beginn der Berechnung natürlich im Startzustand s_0 .

Die Funktionsweise der Turing-Maschine wird im Wesentlichen durch die Funktion δ beschrieben.

Dabei bedeutet $\delta : S \times B \rightarrow S \times B \times \{L,R,N\}$ folgendes:

1. Die Maschine befindet sich in einem Zustand s_x und liest ein Bandsymbol e_x
2. Dann wechselt die Maschine in den Zustand s_y ,
3. schreibt ein Bandsymbol e_y
4. und rückt mit dem Schreib/Lesekopf nach links (L), rechts (R) oder bleibt an der aktuellen Stelle (N)

D.h. die Funktion δ hat als Argumente den aktuellen Zustand s_x und das Bandsymbol e_x , das sich gerade unter der aktuellen Position des Schreib/Lesekopfes befindet. Als Ergebnis liefert die Funktion den Folgezustand s_y , das Symbol e_y , das alte Bandsymbol e_x überschreibt und die nachfolgende Bewegung des Schreib/Lesekopfs als Symbol L, R oder N.

Am konkreten Beispiel $\delta(q,a)=(r,b,L)$ sollte diese Funktionsweise noch klarer werden:

Die Maschine befindet sich in Zustand q , liest das Symbol a , wechselt in den Folgezustand r , schreibt das Symbol b , überschreibt damit das alte Bandsymbol a und bewegt den Schreib/Lesekopf nach links. Damit hat die Turing Maschine einen Einzelschritt ihres Arbeitszyklus durchlaufen. Natürlich kann das neue Symbol b auch mit a identisch sein, es muss nicht zwangsläufig ein neues Symbol auf das Band geschrieben werden. Weiters können der neue und der alte Zustand auch identisch sein, was wir ja bereits in Form von Schleifen bei endlichen Automaten kennen.

Wie bei endlichen Automaten kann auch hier die Überföhrungsfunktion δ mittels Zustandstabelle angegeben werden. Es wird allerdings jedem Paar(Zustand s_x , Bandsymbol e_x) ein Tripel (Zustand s_y , Bandsymbol e_y , (L,R oder N)) zugewiesen.

Erreicht die Turingmaschine einen finalen Zustand, also einen Zustand der Menge F , ist die Berechnung beendet. Die Ausgabe ist dann der Inhalt des Bandes und das Eingabewort ist Element der von der Turing Maschine akzeptierten Sprache.

Meistens wird die Turing Maschine auch so definiert, dass die Überföhrungsfunktion δ nicht vollständig ist und die Maschine anhält, falls δ für das aktuelle Argumentpaar (s_x, e_x) nicht definiert ist. Die Turing Maschine akzeptiert in diesem Fall das Eingabewort nicht und bricht die Berechnung ab.

5.3 Beispiel

Anhand eines einfachen Beispiels soll die Funktionsweise einer Turing Maschine noch einmal deutlich gemacht werden. Gesucht ist eine Turing Maschine, deren Schreib-/Lesekopf nach rechts zum nächsten Leerzeichen läuft und dabei alle Felder löscht.

Wir definieren als Eingabealphabet das binäre Alphabet $E=\{0,1\}$ und als Bandalphabet $B=\{0,1,*\}$, bestehend aus dem Eingabealphabet und dem Leerzeichen *. Weiters benötigen wir für diese einfache Aufgabe nur zwei Zustände und wir definieren die Zustandsmenge $S=\{s_0,s_1\}$ und die Menge der finalen Zustände $F=\{s_1\}$. Der Startzustand ist s_0 . Wie sieht nun die zur Überföhrungsfunktion δ gehörige Zustandstabelle aus? Die Antwort liefert die folgende Tabelle:

	0	1	*
s_0	$(s_0, *, R)$	$(s_0, *, R)$	$(s_1, *, N)$
s_1	-	-	-

Die Funktion ist nicht vollständig definiert und ein Stopp wird durch einen - markiert.

Ausgehend vom Startzustand s_0 und der aktuellen Position des Schreib/Lesekopfes bleibt die Turing Maschine im Zustand s_0 solange eine 0 oder 1 eingelesen wird. Das eingelesene Symbol wird dabei durch ein Leerzeichen * ersetzt und somit gelöscht, danach bewegt sich der Schreib/Lesekopf eine Position nach rechts. Erst wenn an der aktuellen Position ein Leerzeichen eingelesen wird und somit das auf dem Band befindliche Wort vollständig gelöscht wurde, wechselt die Maschine in den finalen Zustand s_1 und beendet die Abarbeitung. In Abbildung 5-2 ist zum Beispiel die resultierende Bandbeschriftung beim Eingabewort 010 dargestellt.

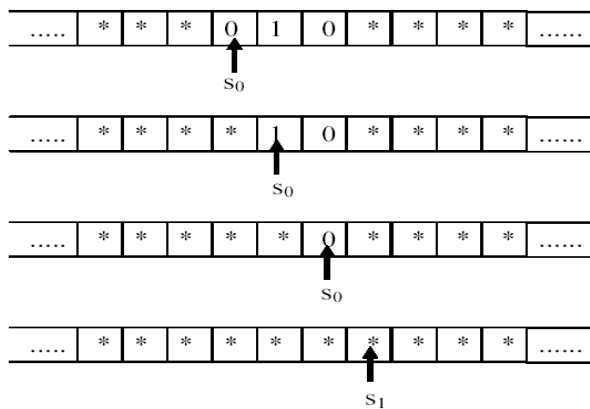


Abbildung 5-2 Beispiel Turing Maschine zum Löschen

5.4 Der Begriff Algorithmus

Es ist nun an der Zeit den zentralen Begriff Algorithmus formal zu definieren und den Zusammenhang mit Turing Maschinen zu erläutern. Das Wort Algorithmus geht zurück auf den arabischen Autor Al-Khowarizmi (ca. 825 n. Chr.), der ein Lehrbuch über das Rechnen mit indischen Ziffern geschrieben hat. Das heißt Algorithmen sind entstanden, um mathematische Berechnungsvorschriften zu definieren. Man kann aber auch allgemeiner Algorithmen einfach als Vorschriften für die Ausführung bestimmter Tätigkeiten sehen und folgende intuitive Beispiele für den Begriff Algorithmus im täglichen Leben angeben:

- ◆ Kochrezepte
- ◆ Bedienungsanleitungen
- ◆ Wegbeschreibungen
- ◆ ...

In diesem intuitiven Sinne ist ein Algorithmus also einfach eine Art Anleitung oder Vorschrift, wie man zu einem Problem eine Lösung findet.

In der Informatik definieren wir nun den Begriff Algorithmus wie folgt:

Ein **Algorithmus** ist eine präzise und eindeutige Beschreibung eines allgemeinen Schemas, das unter Verwendung von endlich vielen, effektiv ausführbaren, elementaren Arbeitsschritten (Aktionen) zur Lösung einer Klasse gleichartiger Probleme von einer mechanisch oder elektronisch arbeitenden Maschine schrittweise abgearbeitet werden kann.

Die mangelnde mathematische Genauigkeit des Begriffs Algorithmus störte viele Mathematiker im 19. und 20. Jahrhundert und es wurden einige Ansätze entwickelt, um den Begriff formaler und sprachlich exakter zu beschreiben. Einer dieser Ansätze sind die nun bereits bekannten Turing Maschinen, mit dessen Hilfe wir den Begriff Algorithmus formal wie folgt definieren können:

Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.

5.5 Berechenbarkeit

Wir werden uns mit Eigenschaften von Algorithmen im Kapitel 6 noch genauer beschäftigen, den Abschluss dieses Kapitels bildet aber die bereits in der Einleitung gestellte Frage, die wir jetzt so formulieren können: Gibt es Problemstellungen, die prinzipiell unlösbar sind, d.h. für deren Lösung man keinen Algorithmus angeben kann?

Die **Church-Turing-These**, benannt nach Alonzo Church und Alan Turing, stellt die Behauptung auf, dass eine Turingmaschine alle von Menschen berechenbaren mathematischen Funktionen lösen kann. Diese These ist mathematisch nicht beweisbar, in der Informatik wird allerdings angenommen, dass sie stimmt.

Für uns Informatiker bedeutet diese These also, dass alles was mit einem Algorithmus berechenbar ist auch mit einer Turing Maschine berechenbar ist. Andererseits, alles was nicht mit einer Turing Maschine berechnet werden kann, ist überhaupt nicht berechenbar! D.h. ein Problem, das man nicht mit einer Turing Maschine lösen kann, gilt auch allgemein als unlösbar! Spätestens jetzt sollte die fundamentale Bedeutung und Mächtigkeit der Turing Maschinen einleuchten.

Nicht alle bekannten Problemstellungen aus der Mathematik bzw. Informatik können gelöst werden. Ein **unentscheidbares** Problem ist eines, das nicht durch einen Algorithmus bzw. eine Turing Maschine gelöst werden kann, auch wenn unbeschränkt Zeit und Geld zur Verfügung steht. Diese Erkenntnis erschüttert vielleicht viele angehende Informatiker bzw. Wirtschaftsinformatiker, da ja oft in Programmiererkreisen die Mentalität gilt, alles lässt sich durch Computer lösen, vorausgesetzt es ist genug Hauptspeicher und CPU-Power verfügbar.

Ein Beispiel eines solchen unentscheidbaren, also nicht lösbaren Problems ist Hilberts⁵ 10. Problem:

Gesucht ist ein Verfahren, das für ein beliebiges Polynom p mit ganzzahligen Koeffizienten in einer endlichen Anzahl von Verfahrensschritten feststellen kann, ob die Gleichung $p = 0$ in ganzen Zahlen lösbar ist.

Ein Beispiel ist das Polynom $6x^3yz^2+3xy^2-x^3-10$. Dieses Polynom hat die ganzzahlige Lösung $x=5, y=3, z=0$ ($6 \cdot 5^3 \cdot 3 \cdot 0^2 + 3 \cdot 5 \cdot 3^2 - 5^3 - 10 = 0$). 1970 konnte allerdings gezeigt werden, dass kein Algorithmus existiert, der dieses Problem für beliebige Polynome löst.

⁵ Im Jahr 1900 präsentierte der deutsche Mathematiker David Hilbert eine Liste von 23 bis dahin ungelösten Problemen der Mathematik

Weitere Fragestellungen der Berechenbarkeitstheorie wie z.B. das Halteproblem würden den Rahmen der Lehrveranstaltung sprengen, finden sich aber z.B. in [Vossen,2004].

Wir verlassen damit das Themengebiet der theoretischen Informatik und widmen uns ab dem nächsten Kapitel dem Kernthema Algorithmen und Datenstrukturen.

5.6 Selbstlernkontrolle

Fragen

Durch welches Sechstupel wird eine Turing-Maschine definiert?

- (a)
 1. ein Eingabealphabet
 2. ein Bandalphabet
 3. die Menge der Zustände
 4. die Überföhrungsfunktion
 5. der Startzustand
 6. die Menge der finalen Zustände
- (b)
 1. ein Nonterminalalphabet
 2. ein Turingalphabet
 3. die Menge der Zustände
 4. die Überföhrungsfunktion
 5. der Startzustand
 6. dem Schreib/Lesekopf
- (c)
 1. ein Nonterminalalphabet
 2. ein Bandalphabet
 3. die Menge der Zustände
 4. die Überföhrungsfunktion
 5. der Startzustand
 6. dem Schreib/Lesekopf

Aus welchen Teilen besteht das Modell einer Turing-Maschine?

- (a) der Kontrolleinheit, dem Speicherband, dem Schreib/Lesekopf
- (b) dem Speicherband, dem Ableitungsbaum, dem Schreib/Lesekopf
- (c) dem Speicherband, dem Stack, der Kontrolleinheit der Kontrolleinheit, dem Speicherband, dem Stack, dem Schreib/Lesekopf

Was beschreibt die Überföhrungsfunktion einer Turing-Maschine?

- (a) jedem Tripel (Zustand, Bandsymbol, Eingabesymbol) wird ein Paar (Zustand, Bandsymbol) zugewiesen.
- (b) jedem Tripel (Zustand, Bandsymbol, links/rechts) wird ein Paar (Zustand, gehe links/rechts) zugewiesen.
- (c) jedem Paar (Zustand, Bandsymbol) wird ein Tripel (Zustand, Bandsymbol, gehe links/rechts) zugewiesen.

Lösungen

Durch welches Sechstupel wird eine Turing-Maschine definiert?

- (a) **1. ein Eingabealphabet**
2. ein Bandalphabet
3. die Menge der Zustände
4. die Überföhrungsfunktion
5. der Startzustand
6. die Menge der finalen Zustände
- (b) 1. ein Nonterminalalphabet
2. ein Turingalphabet
3. die Menge der Zustände
4. die Überföhrungsfunktion
5. der Startzustand
6. dem Schreib/Lesekopf
- (c) 1. ein Nonterminalalphabet
2. ein Bandalphabet
3. die Menge der Zustände
4. die Überföhrungsfunktion
5. der Startzustand
6. dem Schreib/Lesekopf

Aus welchen Teilen besteht das Modell einer Turing-Maschine?

- (a) der Kontrolleinheit, dem Speicherband, dem Schreib/Lesekopf**
- (b) dem Speicherband, dem Ableitungsbaum, dem Schreib/Lesekopf
- (c) dem Speicherband, dem Stack, der Kontrolleinheit der Kontrolleinheit, dem Speicherband, dem Stack, dem Schreib/Lesekopf

Was beschreibt die Überföhrungsfunktion einer Turing-Maschine?

- (a) jedem Tripel (Zustand, Bandsymbol, Eingabesymbol) wird ein Paar (Zustand, Bandsymbol) zugewiesen.
- (b) jedem Tripel (Zustand, Bandsymbol, links/rechts) wird ein Paar (Zustand, gehe links/rechts) zugewiesen.
- (c) jedem Paar (Zustand, Bandsymbol) wird ein Tripel (Zustand, Bandsymbol, gehe links/rechts) zugewiesen.**

5.7 Übungsteil

Übung 1: Einführung Visual Turing

Machen Sie sich mit dem Programm Visual Turing vertraut (Programm im Download-Verzeichnis verfügbar oder unter <http://cheransoft.com/vturing/>) und erstellen Sie eine Turing Maschine, die alle Symbole am Band bis zum nächsten Leerzeichen löscht.

Übung 2: Addition von 1

Entwickeln Sie eine Turing-Maschine, die eine Binärzahl um 1 erhöht, das heißt es soll binär 1 dazu addiert werden. Auf dem Speicherband befindet sich eine nichtleere Folge von 0 und 1 (Binärdarstellung einer natürlichen Zahl). Vor und nach dieser Zahl sind Leerzeichen auf dem Band. Zu Beginn steht die Turing-Maschine am linken Ende der Zahl.

Implementieren Sie diese Turing-Maschine mittels Visual Turing. Da das Programm nur Buchstaben als Eingabesymbole akzeptiert, verwenden Sie anstelle von 0 und 1 die Symbole „O“ und „I“.

Übung 3: Sortierer

Entwickeln Sie mittels Visual Turing eine Turing-Maschine, die als Sortierer arbeitet. Eine aus *a* und *b* bestehende ununterbrochene Bandbeschriftung soll so sortiert werden, dass im linken Teil der Bandbeschriftung nur *a* stehen und im rechten Teil nur *b*.

Beispiel Input: ###ababaab### Resultat: ###aaaabbb###

6 Algorithmen

6.1 Grundlegende Begriffe

Nachdem wir in Kapitel 5.4 bereits den Begriff Algorithmus sowohl intuitiv als auch mathematisch exakt mit Hilfe von Turing Maschinen definiert haben und die Grenzen der Berechenbarkeit aufgezeigt haben, interessieren uns nun wichtige Eigenschaften und grundlegende Begriffe im Zusammenhang mit Algorithmen, die für die Programmierpraxis relevant sind. Betrachten wir noch einmal die Definition aus Kapitel 5.4 und analysieren wir die Bedeutung:

Ein Algorithmus ist eine präzise und eindeutige Beschreibung eines allgemeinen Schemas, das unter Verwendung von endlich vielen, effektiv ausführbaren, elementaren Arbeitsschritten (Aktionen) zur Lösung einer Klasse gleichartiger Probleme von einer mechanisch oder elektronisch arbeitenden Maschine schrittweise abgearbeitet werden kann.

Als erstes wird eine präzise und eindeutige Beschreibung gefordert. Ein Algorithmus kann daher nicht umgangssprachlich in einer natürlichen Sprache wie Deutsch oder Englisch formuliert werden, da hier zwangsläufig Mehrdeutigkeiten entstehen können. Abhilfe schafft hier nur eine exakte mathematische Formulierung, eine Pseudo-Code Notation, eine grafische Notation z.B. mit Flussdiagrammen oder die Formulierung mittels einer Programmiersprache.

Weiters soll ein allgemeines Schema zur Lösung einer Klasse gleichartiger Probleme gefunden werden. Ein Algorithmus muss also ein Lösungsverfahren für eine ganze Klasse von Problemen definieren und nicht nur für ein isoliertes Einzelproblem, d.h. allgemeingültig sein. Eine Vorschrift zur Addition der drei Zahlen 5, 7 und 12 wird daher dieser Forderung nicht genügen, eine Lösungsvorschrift zur Addition von N beliebigen ganzen Zahlen, wobei $N \geq 1$ ist, sehr wohl. Ziel dieses Kriteriums ist es also möglichst wenige mächtige Algorithmen, die universell einsetzbar sind, zu definieren und nicht für jeden Spezialfall einen eigenen Algorithmus angeben zu müssen.

Lernergebnisse

Nach diesem Kapitel sind die Studierenden in der Lage,

- eine Problemstellung mittels Flussdiagramm oder Struktogramm zu modellieren und damit den Begriff Algorithmus zu erklären.
- Den Aufwand nach O-Notation für einen einfachen Algorithmus zu erklären.

Bestandteile eines Algorithmus

Ein Algorithmus darf nur aus endlich vielen elementaren Arbeitsschritten bestehen. D.h. die Beschreibung des Algorithmus darf nicht unendlich groß sein. Diese Eigenschaft wird auch als **statische Finitheit** bezeichnet. Außerdem müssen die für die Ausführung benötigten Ressourcen, insbesondere der Speicherplatz zu jedem Zeitpunkt endlich sein, keinem Algorithmus kann ein unendlich großer Speicherplatz zugeteilt werden. Diese Forderung wird im Gegensatz zur statischen Finitheit als **dynamische Finitheit** bezeichnet.

Die Problemlösung in einem Algorithmus beruht auf einer Folge von elementaren Aktionen eines technischen Systems, siehe Zustandsübergänge in endlichen Automaten oder Turing Maschinen. Die Zustände werden dabei durch Werte von Variablen beschrieben. Was sind nun solche elementaren Arbeitsschritte oder Aktionen? Elementare Aktionen, die in weiterer Folge in Programmiersprachen durch Anweisungen kodiert werden sind z.B. Eingabe, Wertzuweisung und Ausgabe. Die Gesamtheit dieser Anweisungen und damit die Umsetzung des Algorithmus in eine konkrete Programmiersprache werden als **Programm** bezeichnet. Für den Ablauf eines Algorithmus ist es notwendig, die Reihenfolge der Arbeitsschritte festzulegen und bedingungsabhängig zu steuern. Diese Ablaufsteuerung wird mit Hilfe von Kontrollstrukturen realisiert, die die Reihenfolge und Häufigkeit der Ausführung der Aktionen bestimmen. Kontrollstrukturen in prozeduralen Algorithmen beruhen auf mindestens einem der folgenden Kontrollelemente:

- ◆ Folge (Sequenz)
- ◆ Auswahl (Selektion)
- ◆ Wiederholung (Iteration)
- ◆ Unterprogramme (Blöcke)

Die konkrete Umsetzung dieser Elemente z.B. in der Programmiersprache C durch if statements oder for() bzw. while() Schleifen soll nicht Thema dieser Lehrveranstaltung sein sondern wird parallel in der Programmierlehrveranstaltung vermittelt.

6.2 Flussdiagramme und Struktogramme

Als Beschreibungsmittel für Kontrollstrukturen können einerseits grafische Methoden wie **Flussdiagramme** (Ablaufdiagramm) oder **Struktogramme** (nach Nassi-Shneidermann) oder beschreibende Methoden wie **Pseudocode** dienen.

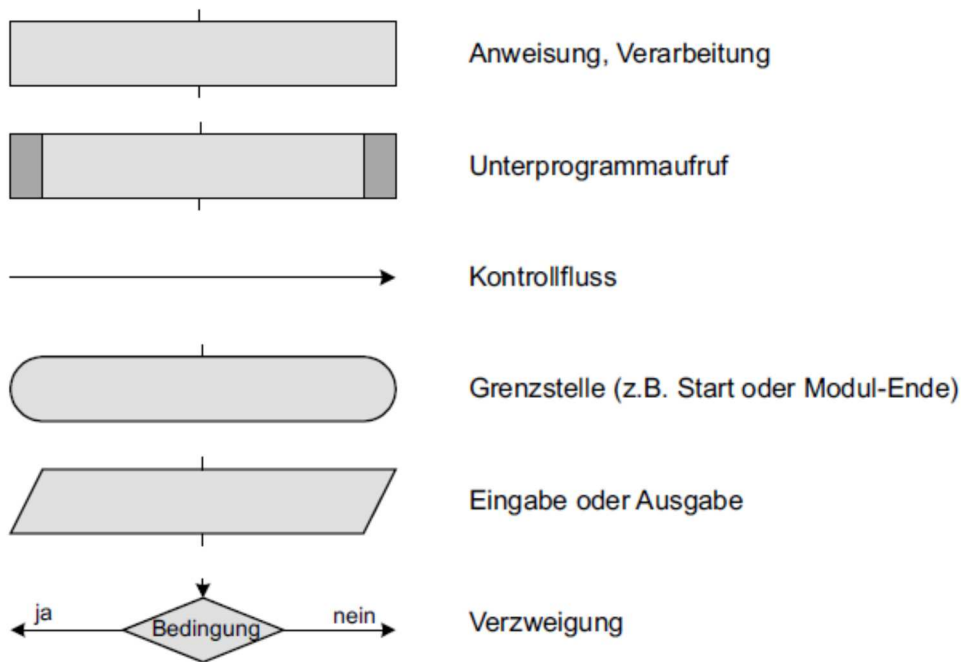
Der Vorteil all dieser Verfahren ist die Loslösung von der konkreten Syntax einer Programmiersprache und somit die Fokussierung auf die wesentliche Elemente des Algorithmus selbst.

„Pseudocode ist eine Kunstsprache, die der natürlichen Sprache ähnelt. Sie ist reduziert und in einer dem Problem angepassten Weise formalisiert. Algorithmen lassen sich damit prägnant und verständlich formulieren“ [Ernst, 2016].

Pseudocode hat also den Vorteil, dass er technologieunabhängig und leichter verständlich auch für NichttechnikerInnen ist. Oft werden vereinfachte Elemente höherer Sprachen kombiniert mit mathematischen Notationen verwendet, um die Ungenauigkeiten der natürlichen Sprache zu reduzieren.

Flussdiagramme bzw. Ablaufdiagramme, deren Symbole in DIN 66001 und DIN 66262 normiert sind, dienen zur übersichtlichen grafischen Veranschaulichung. Auch Struktogramme oder Nassi-Shneiderman-Diagramme (DIN 66261) bieten eine grafische Aufbereitung von Algorithmen und sind oft übersichtlicher als Flussdiagramme, da gerade in komplexeren Flussdiagrammen viele Pfeile und verwirrend wirken können. [Ernst, 2016]

Abbildungen 6.1 und 6.2 zeigen die wichtigsten Elemente und Verzweigungsstrukturen in Flussdiagrammen. Abbildung 6.3 im Gegensatz dazu die wichtigsten Elemente in Struktogrammen.



6-1 Wichtigste Elemente von Flussdiagrammen [Ernst, 2016]

Abbildung

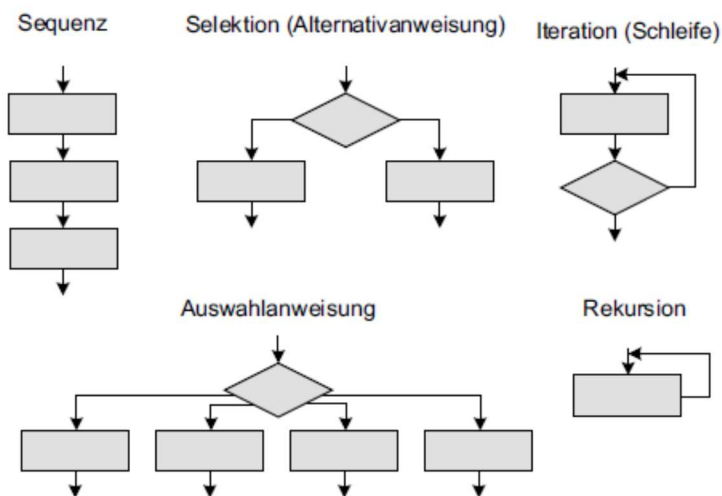


Abbildung 6-2 Verzweigungen in Flussdiagrammen [Ernst, 2016]

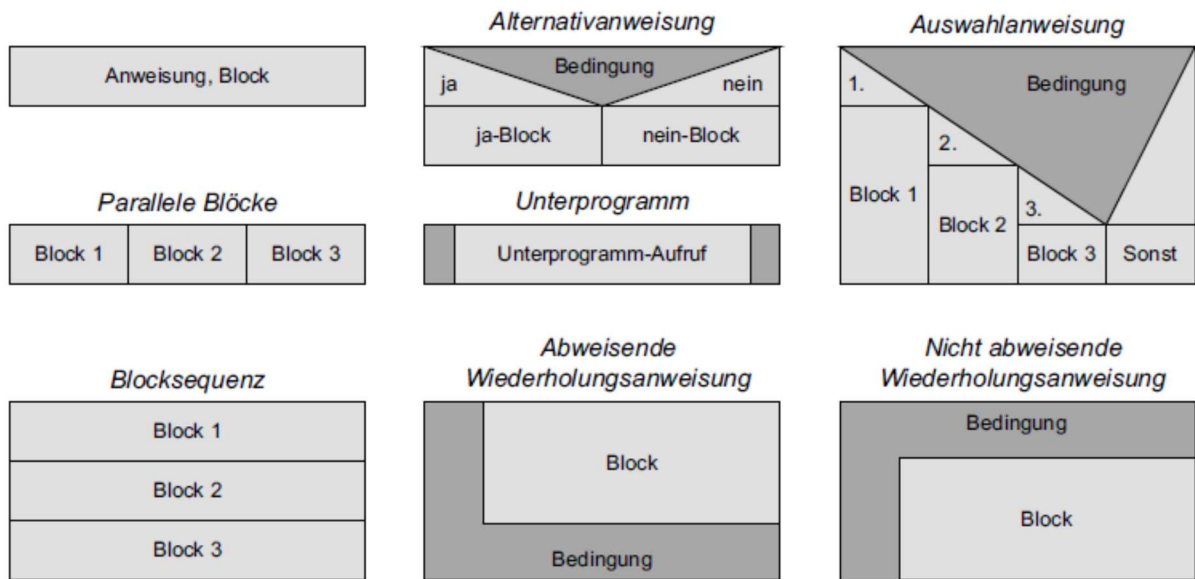


Abbildung 6-3 Wichtigste Elemente von Struktogrammen [Ernst, 2016]

Wir wollen zusätzlich die Kontrollelemente aus Kapitel 6.1. der Reihe nach in den 3 Darstellungsformen gegenüberstellen. Grundlegendes Element der grafischen Methoden ist die Befehlsfolge (Verarbeitung) zur Darstellung einer oder mehrerer Aktionen in einem Algorithmus. Im Flussdiagramm entsteht die Darstellung der Befehle durch Kästchen und des Flusses durch Linien und Pfeile, im Struktogramm durch ineinander geschachtelte Kästchen.

Folgen bestimmen die lineare Reihenfolge von Aktionen in Algorithmen und können wie folgt dargestellt werden:

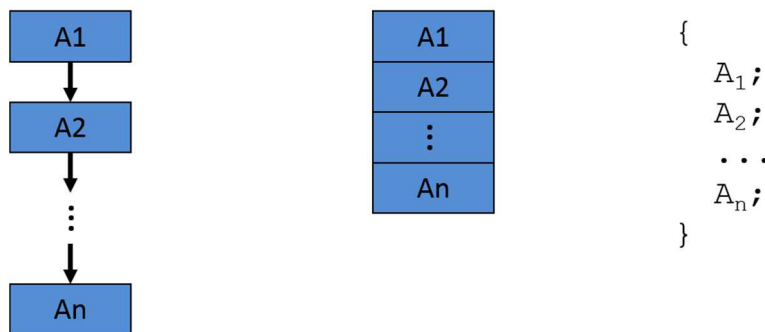


Abbildung 6-4 Folge als Flussdiagramm, Struktogramm und Pseudocode

Bei der Auswahanweisung (sic) gibt es die bedingte Verarbeitung, also eine Aktion, die in Abhängigkeit einer booleschen Bedingung entweder ausgeführt wird oder nicht:

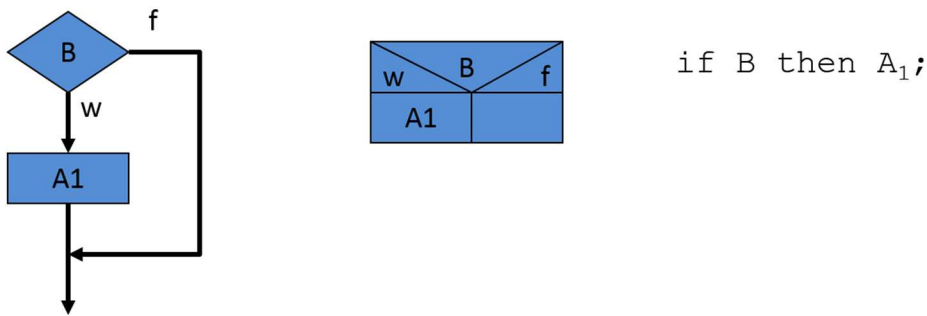


Abbildung 6-5 Bedingte Anweisung

Die einfache Alternative beschreibt eine Auswahlanweisung, wo in Abhängigkeit einer booleschen Bedingung entweder eine Aktion oder eine andere Aktion ausgeführt wird:

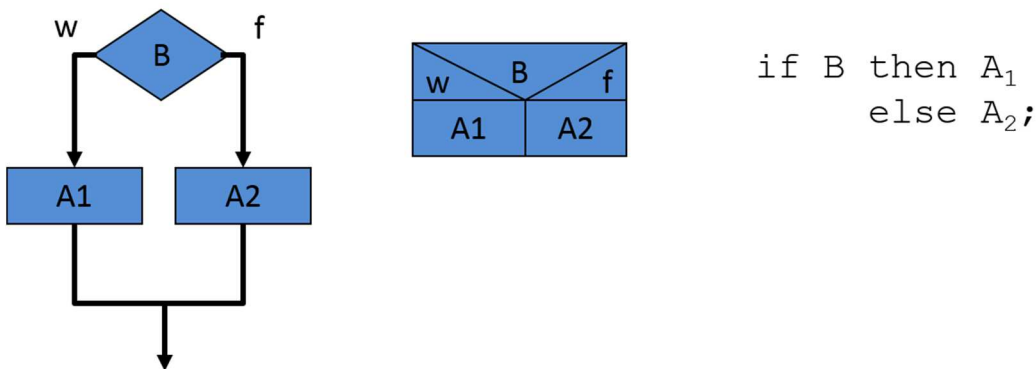


Abbildung 6-6 einfache Alternative

Die mehrfache Alternative beschreibt wie der Name vermuten lässt eine Mehrfachauswahl. In Abhängigkeit einer Bedingung (mit mehreren möglichen Werten w_1, w_2, \dots, w_n) wird eine Aktion aus einer Menge möglicher Aktionen ausgewählt und ausgeführt:

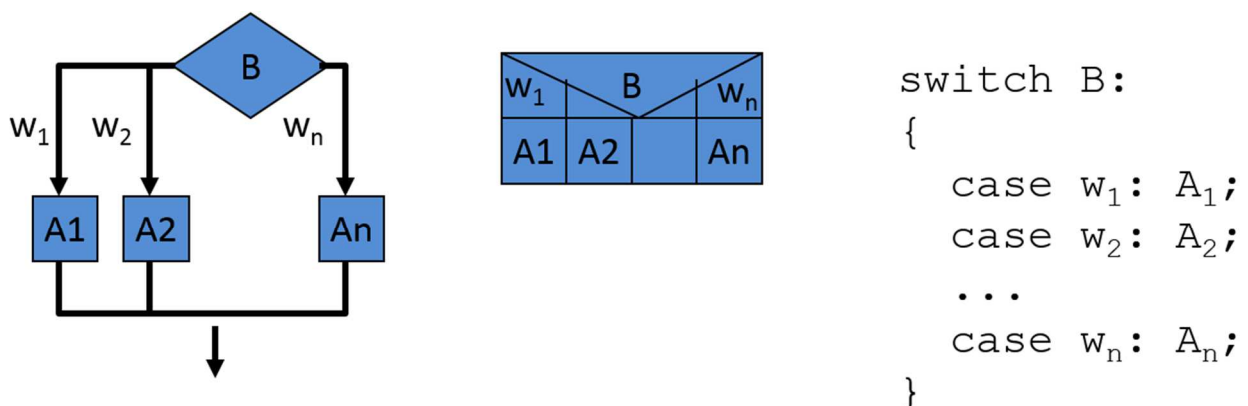


Abbildung 6-7 mehrfache Alternative

Eine Schleife mit vorausgehender Überprüfung ist folgendermaßen definiert: Solange eine boolesche Bedingung erfüllt ist, wird eine Aktion ausgeführt. Die Bedingung wird vor der ersten Ausführung der Aktion geprüft, diese Form heißt auch abweisende Schleife (bzw. While-Schleife).

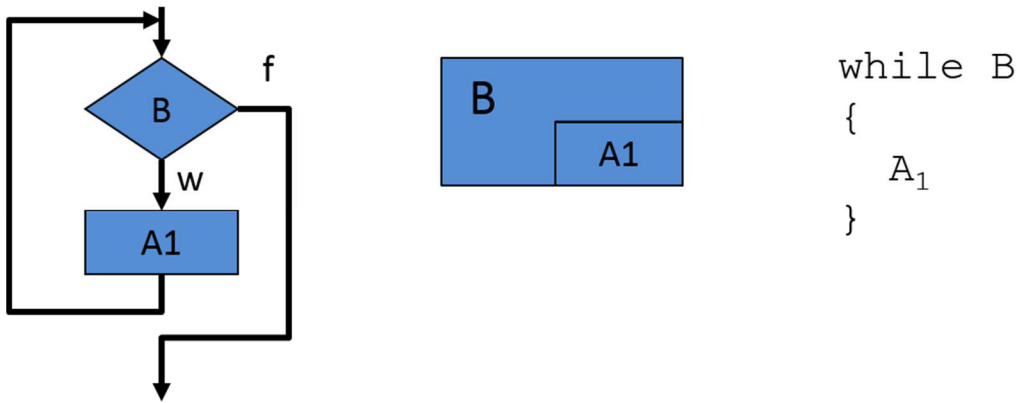


Abbildung 6-8 Schleife mit vorausgehender Überprüfung

Im Gegensatz dazu ist eine Schleife mit nachfolgender Überprüfung wie folgt definiert: Solange bis eine boolesche Bedingung erfüllt ist, wird eine Aktion ausgeführt. Die Bedingung wird (erst) nach der ersten Ausführung der Aktion zum ersten Mal geprüft, d.h. ein Schleifendurchlauf findet auf jeden Fall statt im Gegensatz zur While-Schleife. Diese Form heißt auch Repeat-Schleife.

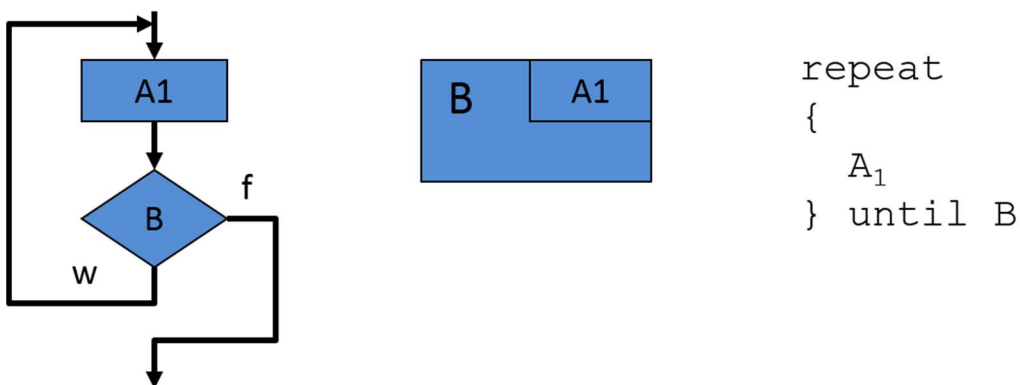


Abbildung 6-9 Schleife mit nachfolgender Überprüfung

6.3 Eigenschaften von Algorithmen

Beispiel Primzahlalgorithmus

Um die Bestandteile eines Algorithmus und die weiteren Eigenschaften besser verstehen zu können, werden wir nun ein praktisches Beispiel näher erläutern. Gesucht wird ein Algorithmus, der feststellt ob eine gegebene Zahl eine Primzahl ist. Es gibt hierzu in der Literatur sehr viele verschiedene Algorithmen, da Primzahlen in vielen Berechnungen benötigt werden. Wir wollen uns aber vorerst auf einen ganz einfachen Algorithmus beschränken, der ohne mathematische Feinheiten auskommt. Der erste Schritt bei der Konstruktion eines Algorithmus ist immer das Nachdenken über das Problem selbst, um möglichst genau die Aufgabenstellung beschreiben zu können. Wann ist eine positive ganze Zahl x eine Primzahl? Aus der Mathematik kennen wir die Definition einer Primzahl. Eine Zahl x ist genau dann eine Primzahl, wenn sie nur durch sich selbst und durch 1 teilbar ist. Diese Eigenschaft hilft uns schon sehr bei der Suche nach einer ersten Lösungsidee für diese Problemstellung. Wir können einfach durch alle möglichen Teiler von 2 bis $x-1$ dividieren. Falls eine dieser Divisionen ohne Rest durchführbar ist, wissen wir dass x keine Primzahl sein kann, auf der anderen Seite haben wir in x eine Primzahl gefunden, wenn keine Division ohne Rest möglich

ist. Nun können wir diesen Algorithmus in einer Pseudo-Code Notation und unter der Zuhilfenahme der prozeduralen Kontrollelemente wie Selektion und Iteration angeben.

1. Setze den Teiler T auf 2.
2. ist x ohne Rest durch T teilbar, gib "x ist nicht prim" aus und beende.
3. erhöhe T um 1.
4. ist $T < x$, fahre fort bei Schritt 2.
5. gib "x ist prim" aus und beende.

Folgende Abbildung zeigt den Algorithmus visualisiert als Flussdiagramm:

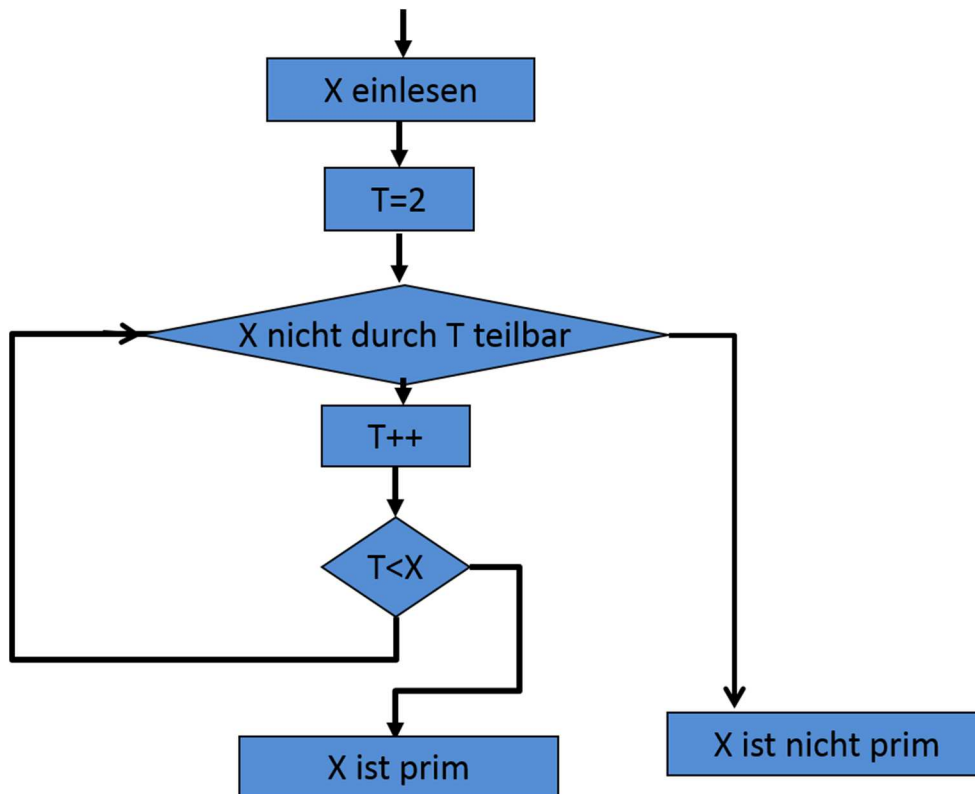


Abbildung 6-10 Flussdiagramm Primzahlalgorithmus

Die Umsetzung in C könnte wie folgt aussehen, eine genaue Analyse des C-Programms folgt dann in Kapitel 6.4:

```

void prim(int x)
{
    int T = 2; /* Schritt 1 */
    do
    {
        if ((x % T) == 0) /* Schritt 2 */
        {
            printf("%d ist nicht prim\n", x);
            return;
        }
        T++; /* Schritt 3 */
    } while (T < x); /* Schritt 4 */
    printf("%d ist prim\n", x); /* Schritt 5 */
}
  
```

Korrektheit von Algorithmen

Eine weitere Eigenschaft eines Algorithmus, die natürlich im Allgemeinen auch wünschenswert ist, ist die **Korrektheit** eines solchen. Für jede korrekte Eingabe wird das nach dem gewählten Algorithmus richtige Ergebnis abgeliefert. Die Wichtigkeit der Korrektheit leuchtet bei kritischen Anwendungen wie z.B. bei Steueranlagen für Atomkraftwerke oder in der medizinischen Informatik z.B. bei der Gehirnchirurgie besonders ein, ist aber natürlich auch in weniger sensiblen Bereichen wünschenswert. Die Frage die sich nun aber aufdrängt lautet, wie kann die Korrektheit eines Algorithmus formal nachgewiesen werden? Idealerweise könnten wir zur Lösung dieses Problems selbst wieder einen Algorithmus angeben, der die Korrektheit von allgemeinen Programmen beweist. Die theoretische Informatik beschäftigt sich mit dieser Fragestellung und kann beweisen, dass es einen solchen Algorithmus nicht gibt und damit dieses Problem unentscheidbar ist, siehe Kapitel 5.5. Man kann also nur im Einzelfall versuchen mit einer mathematischen Beweisführung die Korrektheit eines Algorithmus nachzuweisen, was sehr aufwendig werden kann. Oft begnügt man sich daher mit einem pragmatischeren Ansatz aus dem Bereich des Software-Testens, wie z.B. Black-Box oder White-Box Tests. Wir wollen gar nicht näher auf Teststrategien eingehen, da hiermit eine eigene Vorlesung gefüllt werden könnte. Interessant ist allerdings noch der Begriff **relative Korrektheit**. Ein Algorithmus kann nicht als korrekt an sich bewiesen werden, sondern immer nur in Bezug auf dessen **Spezifikation**. Eine Spezifikation legt eindeutig die berechnete Funktion und das beabsichtige Verhalten eines Algorithmus fest, definiert also was der Algorithmus tun soll.

Die zwei Möglichkeiten, die relative Korrektheit zu überprüfen werden nun **Verifikation** und **Validation** genannt und in [Saake, 2002] wie folgt zusammengefasst:

- ◆ Verifikation: formaler Beweis der Korrektheit bezüglich einer formalen Spezifikation
- ◆ Validation: (nichtformaler) Nachweis der Korrektheit bezüglich einer informellen oder formalen Spezifikation (etwa systematischen Testen)

Die Validation mittels Software-Test Methoden kann im Allgemeinen keine hundertprozentige Sicherheit geben, ob der Algorithmus formal korrekt ist. In der Praxis begnügt man sich aber oft mit einer hinreichenden Sicherheit, oft genug ist sogar erst der Kunde nach dem Kauf der Beta-Tester der Software.

Interessant zum Abschluss der Korrektheitsproblematik ist auch noch die Fragestellung, ob es auch nicht korrekte Algorithmen gibt, die sinnvoll sind? Die Frage verursacht beim Leser zuerst vielleicht Kopfschütteln, was soll ein Algorithmus der ein falsches Ergebnis liefert für einen Sinn haben? An sich wünscht man sich natürlich einen korrekten Algorithmus, der für jede Eingabe das richtige Ergebnis liefert, nur ist das in der Praxis aus Effizienzgründen nicht immer möglich. Wir haben bei unseren Betrachtungen bis jetzt das Thema Komplexität komplett außer Acht gelassen. Wir werden uns noch mit Aufgabenstellungen z.B. bei Graphenalgorithmen beschäftigen, die so komplex sind, dass die exakte Berechnung zu viel Rechenzeit in Anspruch nimmt oder überhaupt unmöglich durchzuführen ist. In solchen Fällen kommen so genannte **approximative Algorithmen** zum Einsatz, die nur eine Näherungslösung mit begrenzter Genauigkeit berechnen, dies dafür aber in einer möglichst kurzen Zeit. Eine andere Möglichkeit bieten z.B. **Monte-Carlo Algorithmen**, die mit einer beschränkten Wahrscheinlichkeit ein falsches Ergebnis liefern dürfen, dafür aber sehr effizient sind. Ein Beispiel dafür ist der Primzahltest von Rabin, der im Gegensatz zu unserem primitiven Primzahlalgorithmus aus dem vorigen Kapitel für große Zahlen sehr schnell bestimmen kann, ob es sich dabei um eine Primzahl handelt oder nicht. Jedoch besteht eine gewisse Irrtumswahrscheinlichkeit. Da die exakte Bestimmung bei großen Zahlen aber sehr viel Zeit in Anspruch nimmt, wird diese Irrtumswahrscheinlichkeit z.B. in der Kryptographie bei asymmetrischen Verschlüsselungsverfahren wie RSA in Kauf genommen, da hier rasch Primzahlen mit tausenden Stellen benötigt werden. Das heißt Korrektheit eines Algorithmus ist ein Kriterium das in bestimmten Ausnahmefällen durchaus aufgelockert werden kann, wenn z.B. die Performance wichtiger erscheint. Wie so oft in der Informatik muss hier ein sinnvoller Kompromiss gefunden werden.

Terminierung

Neben der Korrektheit eines Algorithmus interessiert uns auch dessen Terminierungsverhalten.

Ein Algorithmus heißt **terminierend**, wenn er für jede zulässige Eingabe in endlich vielen Schritten ein Ergebnis liefert.

Aus Effizienzgründen muss es das Ziel sein, hinreichend schnell terminierende Algorithmen zu finden, die eine konkrete Problemstellung lösen können. Ein an sich korrekter Algorithmus, der allerdings nicht terminiert, d.h. unendlich lange für die Berechnung benötigen würde weist in der Praxis verständlicherweise wenig Nutzen auf. Es macht praktisch oft auch keinen Unterschied, ob ein Algorithmus nur sehr lange zur Berechnung benötigt oder nie terminiert, der Benutzer wird in beiden Fällen nach einer gewissen Zeit das Programm abbrechen.

Da es aber bei Algorithmen und deren Umsetzung in Programmen nicht immer nur um Berechnungen geht, haben sehr wohl auch nicht terminierende Algorithmen ihre Berechtigung. Steuerungs- und Überwachungssysteme sind nicht terminierend, sondern sollen wenn möglich endlos weiter laufen und 24 Stunden am Tag, 7 Tage die Woche verfügbar sein, genauso beispielsweise ein Webserver. Auch ein Betriebssystem oder andere Programme, die auf Interaktion des Benutzers warten terminieren nur, wenn der Benutzer einen speziellen Befehl zum Beenden ausführt.

Ob nun ein beliebiger Algorithmus terminiert oder nicht ist übrigens im allgemeinen Fall nicht zu bestimmen. Diese Fragestellung ist wieder ein Thema in der theoretischen Informatik und wird als **Halteproblem** bezeichnet. Alan Turing konnte mit den Turing Maschinen beweisen, dass das Halteproblem nicht entscheidbar ist.

Determinismus und Determiniertheit

Um die Verwirrung komplett zu machen, gibt es neben dem Begriff terminierend noch die Begriffe deterministisch und determiniert. Der Begriff Determinismus ist uns ja schon bereits von den endlichen Automaten ein Begriff. Noch einmal zur Wiederholung: Deterministisch bedeutet dabei, dass es in jedem Zustand für jedes Eingabesymbol nur höchstens einen Folgezustand gibt. Im Kontext von Algorithmen ist die Bedeutung wie folgt: Existiert in einem Algorithmus zu jeder Aktion genau eine Folgeaktion so heißt der Algorithmus **deterministisch**. Somit ist die Schrittfolge der Aktionen eindeutig vorgegeben und es ist pro Teilschritt immer nur eine Möglichkeit zur Programmfortsetzung vorhanden.

Ein Algorithmus ist hingegen **nicht deterministisch**, wenn an einer Stelle im Algorithmus eine willkürliche Auswahl (unter einer gewissen Anzahl von Varianten) möglich ist. Ein Beispiel für einen nichtdeterministischen Algorithmus ist die Vermittlungsfunktion in Datennetzen, wie sie z.B. bei Routing Protokollen im Internet verwendet wird. Die Vermittlungsrechner in den Netzknoten haben mitunter die Wahl zwischen mehreren Übertragungswegen. Der so genannte Next Hop, der nächste Rechner, an den das Paket weitergeleitet wird, kann also nicht deterministisch bestimmt werden.

Können den Varianten Wahrscheinlichkeiten zugeordnet werden, so spricht man von **stochastischen Algorithmen**. Werden die Varianten durch Zufallszahlen gesteuert, spricht man von **randomisierten** oder **probabilistischen Algorithmen**, wie z.B. die in zuvor erwähnten Monte Carlo Methoden.

Sind hingegen die Ausgabewerte eindeutig durch die Eingabewerte bestimmt spricht man von einem **determinierten Algorithmus**. D.h. bei jeder Ausführung mit den gleichen Eingabewerten muss auch wieder das gleiche Resultat geliefert werden. In der Regel wird diese Eigenschaft auch erfüllt

und gewünscht. Randomisierte Algorithmen, die mit Zufallszahlen gesteuert werden, sind oft allerdings nicht determiniert. Eine Simulation eines Roulette Tisches z.B. soll ja auch nicht bei jedem Durchlauf das gleiche Resultat liefern, sondern eben eine zufällige Zahl zwischen 0 und 36.

Wie hängen nun die Begriffe Determinismus und Determiniertheit zusammen? Ein deterministischer Algorithmus ist zwangsläufig auch determiniert. In jedem Einzelschritt eines deterministischen Algorithmus ist ja die Folgeaktion eindeutig definiert, dadurch ist insgesamt ausgehend von der Eingabe auch die Ausgabe eindeutig vorgegeben. Die Umkehrung gilt allerdings nicht! Nicht jeder determinierte Algorithmus muss auch deterministisch sein. Als Beispiel dient wieder das Routing im Internet. Wir haben gesehen, dass der Weg eines Pakets vom Quell- zum Zielrechner nicht immer deterministisch bestimmt sein muss, sehr wohl ist das Ergebnis allerdings determiniert. Egal welchen Weg das Paket nimmt, das Ziel und somit die Ausgabe des Algorithmus ist eindeutig bestimmt. Ein weiteres Beispiel für einen determinierten aber nicht deterministischen Algorithmus ist die folgende Vorschrift zum Sortieren eines Stapels von Karteikarten:

- ◆ Wähle eine zufällige Karte
- ◆ Bilde zwei Stapel, einen mit allen Karten, die alphabetisch vor der gewählten Karte liegen und einen mit allen Karten, die alphabetisch danach gereiht sind
- ◆ Sortiere die beiden kleineren Stapel
- ◆ Füge die sortierten Stapel und die zufällige Karte wieder zusammen

Das Ergebnis dieses Algorithmus ist sicherlich ein korrekt sortierter Stapel von Karteikarten. Da diese Sortierreihenfolge bei jeder Eingabe eindeutig bestimmt ist, ist dieses Verfahren determiniert. Durch die Wahl einer zufälligen Karte ist der Algorithmus aber sicherlich nicht deterministisch.

Zusammenfassend gilt also folgende Regel: **Determinismus impliziert Determiniertheit** aber nicht umgekehrt!

Nun haben wir alle wichtigen Eigenschaften von Algorithmen behandelt, bis auf das Thema Effizienz und Komplexität, das bis jetzt nur kurz beim Thema Korrektheit angesprochen wurde. Da die Komplexität von Algorithmen jedoch ein besonders umfangreiches Forschungsgebiet der theoretischen Informatik ist und auch viele praktische Auswirkungen hat, ist diesem Thema im Folgenden ein eigenes Unterkapitel gewidmet.

6.4 Komplexität von Algorithmen

Die Komplexitätstheorie beschäftigt sich mit dem Abschätzen des Aufwands von gegebenen Algorithmen mit formalen Methoden, wobei sich die Komplexität auf die benötigten Ressourcen, im Allgemeinen Rechenzeit und Speicherplatzbedarf, bezieht.

Es ist in den meisten Fällen klar, dass ein korrekter Algorithmus benötigt wird, der eine gegebene Problemstellung löst, jedoch ist es in der Praxis oft auch wünschenswert, dass ein Algorithmus gefunden wird, der das Problem mit möglichst geringen Aufwand löst. Auch diese Fragestellung wird in der Komplexitätstheorie untersucht, was ist der theoretische Mindestaufwand für eine gegebene Problemklasse?

Betrachtet man einen Algorithmus genauer, lassen sich auch verschiedene Verhaltensweisen bei unterschiedlichen Eingabewerten feststellen. Ein Algorithmus wird in den meisten Fällen nicht für alle Eingabemöglichkeiten dieselbe Effizienz aufweisen. Man unterscheidet daher in der Praxis drei Fälle bei der Betrachtung des Aufwands eines Algorithmus:

- ◆ Bester Fall (**best case**)

- ◆ schlechtester Fall (**worst case**)
- ◆ Mittlerer Fall (**average case**)

Best case und worst case Verhalten sind klar und können mit entsprechender Übung ermittelt werden, der average case ist schwieriger zu bestimmen. Denn hierzu sind auch statistische Mittel notwendig, da ja die Verteilung und die Häufigkeiten der Eingabedaten nicht immer klar sind und es daher oft schwer ist, Durchschnittswerte zu ermitteln.

Aufwandsabschätzung

Um die Fragestellungen der Komplexitätstheorie beantworten zu können, müssen wir ein Maß für den Aufwand definieren, wobei wir uns in diesem Kapitel nur mit der Abschätzung der Laufzeit eines Algorithmus beschäftigen wollen und andere Komplexitätskriterien wie z.B. Speicherplatzbedarf außer Acht lassen wollen.

Unser Aufwandsmaß soll eine mathematische Funktion f sein, wobei $f(N)=a$ folgende Bedeutung hat: Der Algorithmus hat den Aufwand a bei einer Problemgröße bzw. Datenmenge N . Die Problemgröße N ist ein Maß für die Größe der Eingabe, in einem Sortieralgorithmus wäre N z.B. die Anzahl der Elemente, die sortiert werden sollen. Der Aufwand a ist ein grobes Maß für die Rechenzeit, die der Algorithmus benötigt, wobei es nicht darum geht eine exakte Zeit in Sekunden auf einer bestimmten CPU zu bestimmen, sondern die Ausführungszeit maschinenunabhängig abzuschätzen, in dem einfach die Anzahl der Grundoperationen wie Speicherzugriffe, arithmetische Operationen, Vergleiche usw. gezählt werden.

So würde z.B. die einfache C Anweisung

```
x=x+1;
```

als eine Operation zählen. Eine for Schleife von 1 bis n , die eine Berechnung durchführt, z.B.

```
int summe=0;
for(int i=1; i<=n; i++)
{
    summe=summe+i;
}
```

hingegen zählt n Operationen (eine für jeden Schleifendurchlauf) und eine doppelt verschachtelte for Schleife, wie z.B.

```
int summe=0;
for(int i=1;i<=n;i++)
{
    for(int j=1;j<=n;j++)
    {
        summe=summe+i*j;
    }
}
```

bereits n^2 Operationen. Primitive Operationen wie Erhöhen der Zählvariablen, Indexberechnungen werden dabei nicht berücksichtigt. Diese Details können insofern vernachlässigt werden, da selten der exakte Aufwand sondern nur eine ungefähre Abschätzung der Größenordnung relevant ist.

Betrachtet wird das Verhalten des Aufwandes bei immer größer werdenden Problemgrößen N . Mathematisch interessiert und also der asymptotische Verlauf der Funktion f . Die Aufwandsabschätzung gilt also nur für große Datenmengen und analysiert wird die Ordnung von Algorithmen. Ein Beispiel verdeutlicht diesen Sachverhalt: Benötigt Algorithmus A für doppelt so viele Daten doppelt so lange und Algorithmus B hingegen viermal solange, so besitzen Algorithmus

A und B eine unterschiedliche Ordnung. Ist Algorithmus B jedoch immer nur um eine Konstante langsamer, z.B. 50%, haben beide trotzdem die gleiche Ordnung.

Die O-Notation

Die Ordnung von Algorithmen lässt sich mathematisch mit Hilfe der O-Notation nach den Zahlentheoretikern Landau und Bachmann beschreiben. Das große O steht dabei für Ordnung und ist eins der Landau Symbole zum Beschreiben des asymptotischen Verhaltens von Funktionen und Folgen.

Die Definition der O-Notation lautet wie folgt:

Ein Algorithmus $g(N)$ (Funktion über der Datenmenge N) wird als **$O(f(N))$** bezeichnet, wenn Konstanten c_0 und N_0 (natürliche Zahlen) existieren, sodass: $g(N) < c_0 \cdot f(N)$ für alle $N > N_0$

Man spricht dann: $g(N)$ hat die Ordnung $f(N)$, wobei $f(N)$ die obere Schranke für den asymptotischen Verlauf des Aufwands darstellt. Die Asymptote ist ja eine Gerade, der sich eine Kurve bei immer größerer Entfernung vom Ursprung unbegrenzt annähert. Der Algorithmus g wächst damit asymptotisch nicht schneller als die Funktion f und g gehört zur Klasse der Funktionen, deren Wachstum durch f beschränkt ist. c_0 und N_0 sind unbekannt und können beliebig groß sein, auch daran erkennen wir schon, dass diese Notation keinerlei Bedeutung für kleine N und den exakten Aufwand haben kann, sondern nur zum Vergleich von Algorithmen dient und die Größenordnung des Aufwands angibt. Die wichtigsten Vergleichsfunktionen für $f(N)$ die in der Praxis bei der Aufwandsabschätzung von Algorithmen auftreten, sind in der folgenden Tabelle zusammengefasst. Man spricht in diesem Zusammenhang auch von Komplexitätsklassen.

Aufwand	Bezeichnung	Einstufung
$O(1)$	konstanter Aufwand	optimal
$O(\lg N)$	logarithmischer Aufwand	
$O(\sqrt{N})$	Aufwand mit Wurzel N	
$O(N)$	linearer Aufwand	
$O(N \cdot \lg N)$	quasilinearer Aufwand	
$O(N^2)$	quadratischer Aufwand	
$O(N^k)$	polynomieller Aufwand	
$O(2^N)$	exponentieller Aufwand	
$O(N!)$	faktorieller Aufwand	am schlechtesten

Die Funktionen in der Tabelle sind nach Rechenaufwand sortiert, $O(1)$, der konstante Aufwand wäre optimal, ein exponentieller Aufwand $O(2^N)$ oder ein faktorieller Aufwand $O(N!)$ sind in der Praxis katastrophal von der Rechenzeit, wie die folgende Wachstumstabelle für den jeweiligen Komplexitätsbereich eindrucksvoll unter Beweis stellt. Bereits bei einer ja relativ kleinen

Datenmenge von $N=32$ ist der Unterschied zwischen logarithmischem Aufwand (5 Operationen) und exponentiellem Aufwand (4 Milliarden Operationen) dramatisch.

N	ld N	$N * \log N$	N^2	2^N
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65536
32	5	160	1024	4294967296

Da die O-Notation nur eine obere Schranke angibt, kann die Funktion $f(N)$ mit folgenden Rechenregeln vereinfacht werden:

- ◆ $O(k \cdot f(N)) = O(f(N))$
konstante Faktoren spielen keine Rolle und können weggelassen werden, so ist z.B. $O(100 \cdot N) = O(N)$, was auf den ersten Blick vielleicht seltsam aussehen mag, da in der Praxis ein Faktor 100 ja sehr wohl den Aufwand spürbar vergrößern würde. Da uns aber nur das asymptotische Verhalten bei großen Datenmengen N interessiert, können wir diese Faktoren trotzdem außer Acht lassen.
- ◆ $O(\log_a N) = O(\lg N)$
Die Basis des Logarithmus spielt keine Rolle, da es sich hierbei wieder nur um einen konstanten Faktor handelt.
- ◆ $O(f(N) + g(N)) = O(\max\{f(N), g(N)\})$
In einer Summe von Termen ist nur der aufwendigste Teil relevant, also der höchste Exponent, die anderen Teile können vernachlässigt werden. So entspricht z.B. $O(N + N^2)$ dem Aufwand $O(N^2)$.

Ein Beispiel soll diese Rechenregeln noch einmal verdeutlichen:

Wie kann die Aufwandsfunktion $O(17N^2 + 3N + 100)$ vereinfacht werden?

Nach den obigen Regeln, müssen wir in einer Summe von Termen nur den aufwendigsten Teil betrachten, in unserem Fall $17N^2$, da N^2 der höchste Exponent ist. 17 ist nur ein konstanter Faktor und kann ebenfalls ignoriert werden, somit bleibt der Aufwand $O(N^2)$ und $O(17N^2 + 3N + 100)$ kann also zu $O(N^2)$ vereinfacht werden.

Aufwandsabschätzung Beispiel Primzahlalgorithmus

Betrachten wir noch einmal unseren primitiven Primzahlalgorithmus aus Kapitel 6.3 und versuchen eine Aufwandsabschätzung mit Hilfe der O-Notation anzugeben. Zur Erinnerung noch einmal der Algorithmus in C:

```
void prim(int x)
{
```

```

int T = 2; /* Schritt 1 */
do
{
if ((x % T) == 0) /* Schritt 2 */
{
    printf("%d ist nicht prim\n", x);
    return;
}
T++; /* Schritt 3 */
}
while (T < x); /* Schritt 4 */
printf("%d ist prim\n",x); /* Schritt 5 */
}

```

Analysieren wir Zeile für Zeile des Algorithmus und versuchen wir die Aufwandsabschätzung für den best case und für den worst case in Abhängigkeit der Eingabe x anzugeben. Schritt 1, die Initialisierung und Wertzuweisung der Variable T wird auf jeden Fall ausgeführt, kann aber als primitive Operation vernachlässigt werden. Der Aufwand ist somit für diese Anweisung $O(1)$, konstant. Den Hauptteil des Algorithmus bildet die do-while Schleife $\text{while } (T < x)$. Um deren Aufwand angeben zu können, müssen wir die Schleife genauer analysieren. Sie wird auf jeden Fall mindestens einmal durchlaufen, da ja die Bedingung erst am Ende der Schleife überprüft wird. Innerhalb der Schleife findet mit Schritt 2 eine Selektion in Form der Anweisung $\text{if } ((x \% T) == 0)$ statt. Die Überprüfung der Bedingung hat wieder konstanten Aufwand $O(1)$ und kann vernachlässigt werden. Ist die Bedingung erfüllt, wird ausgegeben, dass es sich bei x um keine Primzahl handelt und der Algorithmus ist zu Ende. Wir haben somit bereits den best case des Algorithmus gefunden, den kürzest möglichen Kontrollfluss durch das Programm. Im ersten Schleifendurchlauf ist T gleich 2, das heißt wenn x modulo 2 gleich 0 ist, also durch 2 ohne Rest dividierbar ist oder noch anders formuliert, wenn x gerade ist, wird der Algorithmus sofort wieder beendet. Der best case des Algorithmus gilt also für gerade x und hat den Gesamtaufwand $O(1)$, also konstanten Aufwand. Dieser Aufwand setzt sich nur aus der Initialisierung der Variable T (Schritt 1), der einmaligen Überprüfung der Bedingung (Schritt 2) und der Ausgabe, dass x nicht prim ist zusammen. Alle drei Teilschritte haben konstanten Aufwand $O(1)$. Aus den Rechenregeln der O -Notation wissen wir, dass $O(1)+O(1)+O(1)$ gleich $O(1)$ ist.

Was passiert allerdings wenn x nicht durch 2 teilbar ist und wie schaut der worst case des Algorithmus aus? Ist die Bedingung in Schritt 2 nicht erfüllt, so wird in Schritt 3 mit der Anweisung $T++$; T um 1 erhöht und in Schritt 4 die Bedingung der do-while Schleife überprüft. Solange $T < x$ gilt, wird die Schleife weiter durchlaufen. Im nächsten Schleifendurchlauf wird natürlich wieder in Schritt 2 überprüft ob x durch T ohne Rest teilbar ist usw. Diese Bedingung wird genau für eine Primzahl nie zutreffen und wir haben somit den worst case des Algorithmus gefunden. Im Falle einer Primzahl wird Schritt 2 nie zum Abbruch des Algorithmus führen sondern die do-while Schleife solange ausgeführt, bis schlussendlich T gleich x gilt. Dann bricht die do-while Schleife ab und es folgt in Schritt 5 die Ausgabe, dass es sich bei x um eine Primzahl handelt. Wie oft wird nun in diesem Fall die Schleife durchlaufen? T ist mit 2 initialisiert und wird pro Schleifendurchlauf um 1 erhöht. Die Schleife wird also von T gleich 2 bis T gleich $x-1$ ausgeführt, wir erhalten daher $x-2$ Iterationen. Der Aufwand der Schleife kann dann mit der Formel Anzahl der Iterationen * Aufwand der Anweisungen innerhalb der Schleife berechnet werden und ist daher in unserem Fall $(x-2) \cdot O(1)$, da alle Anweisungen innerhalb der Schleife nur konstanten Aufwand haben. Da in der O -Notation immer n und nicht x als Eingabegröße verwendet wird, erhalten wir einen linearen Aufwand von $O(n)$. Die Konstanten -2 sowie der konstanten Aufwand für Schritt 1 und Schritt 5 kann aufgrund der Rechenregeln ebenfalls vernachlässigt werden und wir erhalten den worst case Aufwand $O(n)$ für unseren Primzahlalgorithmus, der genau dann eintritt wenn es sich bei x wirklich um eine Primzahl handelt.

Zum Abschluss dieser Aufwandsanalyse drängt sich natürlich die Frage auf, ob dieser Algorithmus wirklich effizient ist und ob es nicht weitaus bessere Primzahlalgorithmen gibt. In der Literatur findet sich eine Vielzahl von interessanten Algorithmen zu diesem Thema, die bei weiten den Rahmen dieses Skriptums sprengen würden, einige kleine Optimierungen können wir allerdings sehr wohl einbauen.

Aus der Mathematik können wir herleiten, dass der kleinste Teiler einer Nicht-Primzahl sicher kleiner gleich als deren Quadratwurzel sein muss. Wir müssen in der do-while Schleife also nicht bis $x-1$ testen sondern können bereits bei der Wurzel von x abbrechen. Weiters ist es unnötig jede Zahl von 2 bis Wurzel x als Teiler zu testen, es genügt vielmehr den Teiler 2, dann den Teiler 3 zu testen und danach den Teiler in jedem Schleifendurchlauf um 2 zu erhöhen, da 2 die einzige gerade Primzahl ist. Diese Änderungen sind im folgenden Algorithmus zusammengefasst:

```
void prim (int x)
{
    if ((x % 2) == 0) {
        printf("%d ist nicht prim\n",x);
        return;
    }
    int T = 3;
    while (T*T<=x)
    {
        if ((x % T) == 0)
        {
            printf("%d ist nicht prim\n",x);
            return;
        }
        T = T + 2;
    }
    printf("%d ist prim\n",x);
}
```

Im best case hat sich gegenüber dem ersten Algorithmus nichts geändert. Bei einer geraden Zahl bricht der Algorithmus wieder sofort nach Überprüfung der ersten Bedingung ab und wir erhalten wieder konstanten Aufwand $O(1)$. Im worst case allerdings, wenn es sich bei x um eine Primzahl handelt, sind statt $x-2$ nur mehr $(\sqrt{x}-3)/2$ Iterationen notwendig, da die do-while Schleife jetzt nur von 3 bis \sqrt{x} geht und T in jedem Schleifendurchlauf um 2 erhöht wird. Wir erhalten somit also worst case Aufwand $O(\sqrt{n})$ da die konstanten Faktoren keine Rolle spielen.

Ausblick Traveling Salesman Problem

Zum Abschluss des Kapitels Komplexität und damit als krönender Abschluss und Ausblick der gesamten Lehrveranstaltung soll nun ein besonders komplexes und berühmtes Problem der Informatik kurz vorgestellt werden, dass auch speziell im Kontext des Studiengangs Verkehr und Umwelt im Logistikbereich eine große Bedeutung hat.

Das Problem des Handlungsreisenden bzw. das Traveling Salesman Problem (TSP) ist in der einfachsten Form für einen ungerichteten, vollständigen, zusammenhängenden und gewichteten Graphen $G = (V,E)$ definiert. Die Knoten können dabei als Städte und die Kanten als Verbindungen zwischen den Städten mit der Entfernung als Kantengewicht interpretiert werden.

Gesucht ist nun eine **Rundreise durch alle Städte**, die jede Stadt genau einmal besucht und **minimale Gesamtkosten** aufweist.

Ein Beispiel einer solchen Rundreise mit minimalen Kosten zeigt folgende Abbildung 6-11 rechts. Links sind die Knoten des Graphen dargestellt. Da es sich um einen vollständigen Graphen handelt, in dem sowieso von jedem Knoten zu jedem anderen Knoten im Graph eine Kante existiert, müssen diese der Übersichtlichkeit wegen nicht gesondert eingezeichnet werden. Auch das Kantengewicht ergibt sich automatisch durch die Entfernung der beiden beteiligten Knoten. Selbst bei dieser geringen Knotenanzahl gibt es unzählige mögliche Rundreisen und es ist auf den ersten Blick sicherlich nicht erkennbar, ob die Rundreise in der rechten Abbildung wirklich die optimale Lösung darstellt.

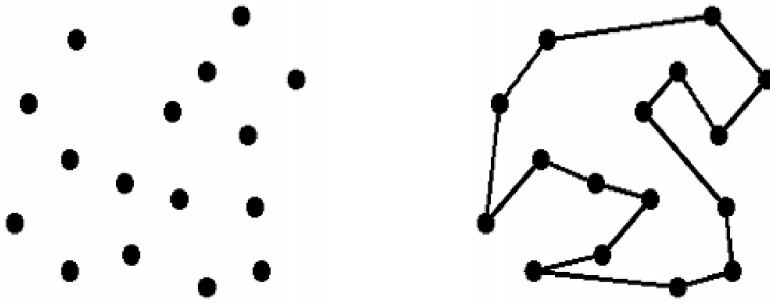


Abbildung 6-11 TSP Rundreise

Anwendungen des TSP in der klassischen Form finden sich in der Routenplanung eines Kundendienstes oder allgemein in Logistik Unternehmen, sowie bei der Verteilung von Waren. Die Anwendungen des TSP sind jedoch noch weit vielschichtiger, da die Begriffe Städte und Entfernungen ja nicht immer wörtlich zu nehmen sind. Beim Bohren von Leiterplatten, muss ein Bohrer eine vorgegebene Menge von Löchern in möglichst kurzer Zeit abarbeiten, eine minimale „Rundreise“ des Bohrers über alle Löcher kann dies garantieren. Auch die Planung von Layouts für integrierte Schaltkreise bedient sich beim TSP. Schlussendlich kann das TSP sogar in der Genetik bei der Genom-Sequenzierung eingesetzt werden, wobei hier die Entfernung für den Grad der Übereinstimmung zweier DNA-Stränge steht.

Wie kann das TSP nun exakt gelöst werden?

Eine primitive Lösungsidee wäre ein **Enumerationsverfahren**, also das einfache Auflisten aller möglichen Rundreisen (Touren) und die Bestimmung der Tour mit minimalen Kosten. Hierbei drängt sich die Frage auf: Wie viele Touren gibt es?

Ein vollständiger ungerichteter Graph mit n Knoten besitzt $n \cdot (n-1) / 2$ Kanten. Wird die Ausgangsstadt fixiert, da die Rundreise ja in einer beliebigen Stadt starten kann, so erhält man aus der Kombinatorik insgesamt **$(n-1)! / 2$ Touren**. Der Faktor $1/2$ entsteht dadurch, dass die Richtung der Tour, also ob die Städte im oder gegen den Uhrzeigersinn besucht werden, keine Rolle spielt.

Um ein Gefühl für die Größenordnung dieser Formel zu bekommen und die Auswirkungen auf die Rechenzeit abschätzen zu können, treffen wir folgende Annahme: ein Rechner kann 40 Millionen Hamiltonsche Kreise (HC) pro Sekunde berechnen. Die unten stehende Tabelle gibt für N Städte die Anzahl der Hamiltonschen Kreise und die benötigte Rechenzeit an.

N	Anzahl HC	Zeit
10	181440	0,00045 Sekunden
17	ca. 10^{13}	3 Tage
19	ca. 10^{15}	2,5 Jahre
20	ca. 10^{17}	48 Jahre
25	ca. 10^{23}	10^8 Jahre
60	ca. 10^{80}	10^{64} Jahre

Spätestens jetzt sehen wir, dass die Enumeration aller Möglichkeiten wohl keine vernünftige Alternative darstellt. Schon bei einer vergleichsweise geringen Problemgröße von 20 Städten gibt es 10^{17} Möglichkeiten, für die auch ein Rechner der nächsten Generation noch sehr lange rechnen würde, gar nicht zu reden von Problemstellungen mit 100en oder 1000en Knoten, die in der Praxis durchaus vorkommen können, wo wir auf Zeitdimensionen stoßen, die die Entstehungsgeschichte unseres Universums sprengen würden. Die gewonnenen Erkenntnisse aus der Komplexitätstheorie

zeigen aber, dass kein exakter Algorithmus bekannt ist, der das Problem allgemein in polynomieller Zeit und damit effizient lösen kann. Nur für kleine Städteanzahlen oder für Spezialfälle ist das TSP exakt lösbar.

Die folgende Abbildung zeigt eine exakte Lösung für eines der momentan größten gelösten TSP Instanzen, 15112 Städte Deutschlands. Das Problem wurde an der Rice und Princeton University in den USA mit Hilfe von 110 Prozessoren und 22.6 Jahren Gesamtrechenzeit gelöst. Die Beschreibung eines exakten Lösungsverfahrens würde den Rahmen dieses Skriptums sprengen, nähere Infos finden sich aber auf der inoffiziellen TSP Homepage <http://www.math.princeton.edu/tsp/>.



Abbildung 6-12 exakte Lösung eines TSPs für 15112 Städte

Wir verwenden daher nur **approximative Algorithmen**, die die optimale Lösung nur annähern, solche Verfahren werden auch **Heuristiken** genannt.

Für das TSP gibt es eine Vielzahl von Heuristiken. Eine Kategorie von solchen Verfahren sind dabei **Konstruktionsheuristiken**, die von Grund auf eine Näherungslösung erzeugen, wie z.B. die Nearest Neighbour Heuristik, diverse Insert Heuristiken oder die Spanning Tree Heuristik, die eine Rundreise ausgehend von einem minimalen Spannbaum erzeugt. Weiters gibt es **Verbesserungsheuristiken**, die ausgehend von einer zulässigen Lösung versuchen, diese durch

lokale Änderungen zu verbessern. Auch **genetische bzw. evolutionäre Algorithmen** können zur Approximation eines TSP verwendet werden. Hier werden aus dem Vorbild der biologischen Evolution Optimierungsprobleme durch die natürlichen Operationen wie Mutation, Selektion und Rekombination gelöst.

Applets zur Visualisierung der wichtigsten Heuristiken und weiterführende Informationen finden sich unter <http://www-e.uni-magdeburg.de/mertens/TSP/index.html>.

6.5 Selbstlernkontrolle

Fragen

Welche der folgenden Aussagen sind wahr?

- (a) Jeder deterministische Algorithmus ist auch determiniert.
- (b) Jeder determinierte Algorithmus ist korrekt.
- (c) Pseudocode ist unabhängig von einer konkreten Programmiersprache.
- (d) Die O-Notation hat keine Aussagekraft für sehr große Datenmengen.

Was ergibt die Vereinfachung des folgenden Ausdrucks mit Hilfe der Rechenregeln zur O-Notation?

$$O(2000 \cdot (30N^9 + N^3 + 10N^2) / (50N^3 + N))$$

- (a) $2000 \cdot N^3$
- (b) N^3
- (c) $2000 \cdot N^6$
- (d) N^6

Was ergibt die Vereinfachung des folgenden Ausdrucks mit Hilfe der Rechenregeln zur O-Notation?

$$O((N^9 + N^3) \cdot (5N^2) / (N^3 + N))$$

- (a) $5 \cdot N^8$
- (b) N^8
- (c) $5 \cdot N^6$
- (d) N^6

Welche Aussagen zum TSP treffen zu?

- (a) Es gibt eine Vielzahl von effizienten Algorithmen zur exakten Bestimmung der optimalen Rundreise.
- (b) Für 5 Städte gibt es 12 unterschiedliche Rundreisen.
- (c) Der exakte Algorithmus zur Bestimmung der optimalen Rundreise von N Städten mittels vollständiger Enumeration hat Aufwand $O(N!)$.
- (d) Mit Heuristiken kann garantiert immer die exakte Lösung gefunden werden.

Lösungen

Welche der folgenden Aussagen sind wahr?

- (a) Jeder deterministische Algorithmus ist auch determiniert.**
- (b) Jeder determinierte Algorithmus ist korrekt.
- (c) Pseudocode ist unabhängig von einer konkreten Programmiersprache.**
- (d) Die O-Notation hat keine Aussagekraft für sehr große Datenmengen.

Was ergibt die Vereinfachung des folgenden Ausdrucks mit Hilfe der Rechenregeln zur O-Notation?

$$O(2000 \cdot (30N^9 + N^3 + 10N^2) / (50N^3 + N))$$

- (a) $2000 \cdot N^3$
- (b) N^3
- (c) $2000 \cdot N^6$
- (d) N^6**

Was ergibt die Vereinfachung des folgenden Ausdrucks mit Hilfe der Rechenregeln zur O-Notation?

$$O((N^9 + N^3) \cdot (5N^2) / (N^3 + N))$$

- (a) $5 \cdot N^8$
- (b) N^8**
- (c) $5 \cdot N^6$
- (d) N^6

Welche Aussagen zum TSP treffen zu?

- (a) Es gibt eine Vielzahl von effizienten Algorithmen zur exakten Bestimmung der optimalen Rundreise.
- (b) Für 5 Städte gibt es 12 unterschiedliche Rundreisen.**
- (c) Der exakte Algorithmus zur Bestimmung der optimalen Rundreise von N Städten mittels vollständiger Enumeration hat Aufwand $O(N!)$.**
- (d) Mit Heuristiken kann garantiert immer die exakte Lösung gefunden werden.

6.6 Übungsteil

Übung 1: Quersumme

Zeichnen Sie ein Flussdiagramm und Struktogramm für folgenden Algorithmus zur Berechnung der Quersumme von beliebigen Zahlen (=Summe der einzelnen Ziffern).

Eingabe:

Eine beliebige (positive ganze) Zahl. Mit „Enter“ wird eine Eingabe beendet. Andere Zeichen (Buchstaben, Sonderzeichen) sind nicht zulässig (in diesem Fall soll das Programm das eingegebene Zeichen ignorieren).

Ausgabe:

Quersumme.

Beispiel:

145c81 Quersumme: 19

Beispiel-Algorithmus:

1. Quersumme wird mit 0 initialisiert.
2. Einlesen der nächsten Ziffer (bis „enter“ gedrückt wird).
3. Wenn Eingabe eine gültige Ziffer:

Ziffer zur Quersumme hinzufügen.

4. Wenn "Enter" eingegeben wurde:

Quersumme ausgeben und auf 0 zurücksetzen.

5. Gehe wieder zu Punkt 2.

Übung 2: Druckeinheiten

Fahrradpumpen und Luftentnahmestationen bei Tankstellen zeigen Reifendruck oft in verschiedenen Einheiten an, was gerade bei Auslandsreisen problematisch sein kann. Hier wäre eine Umrechnungstabelle praktisch. Erstellen Sie daher ein Flussdiagramm und ein Struktogramm zur Umrechnung von Bar in andere Einheiten.

Eingabe:

Keine Eingabe.

Ausgabe:

Eine Tabelle, die für Werte zwischen 0 und 15 Bar (in Schritten von 1 Bar) in einer Zeile die Werte in Bar, Pascal, technischer Atmosphäre, Torr und Pfund pro Quadratzoll (in dieser Reihenfolge), getrennt durch Tabulatoren, angibt.

Beispiel:

Bar	Pascal	At	Torr	psi
0	0	0	0	0

(...)				
4	400000	4.08	3000	58.016

Hinweise:

Verwenden Sie für die Umrechnung folgende Tabelle (siehe auch [http://de.wikipedia.org/wiki/Druck_\(Physik\)](http://de.wikipedia.org/wiki/Druck_(Physik))):

Bar	Pascal	Techn. Atmosphäre	Torr	Pfund pro Quadratzoll
1 bar	100000 Pa	1.02 at	750 torr	14.504 psi

Übung 3: Reihenentwicklung von π

Die Kreiszahl π kann durch die Euler-Reihe recht gut approximiert werden:

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 \dots$$

Entwerfen Sie einen Algorithmus, mit dem diese Reihe für eine eingegebene Anzahl von Schritten berechnet werden kann (ein Schritt ist die Addition oder Subtraktion eines weiteren Werts). Nach erfolgter Ausgabe soll ein neuer Wert eingegeben werden können. Wird 0 eingegeben, wird das Programm beendet. Beschreiben Sie Ihren Algorithmus mittels Struktogramm und Pseudocode.

Eingabe:

Anzahl der Schritte der Reihenentwicklung.

Ausgabe:

Approximation von Pi mit 8 Nachkommastellen.

Beispiel:

```
Anzahl der Iterationen: 100
Pi = 3.13159290
Anzahl der Iterationen: 5000
Pi = 3.14139265
Anzahl der Iterationen: 100000
Pi = 3.14158265
Anzahl der Iterationen: 0
Press any key to continue
```

Übung 4: Determinismus und Determiniertheit

Geben Sie ein eigenes Beispiel für einen Algorithmus an, der determiniert aber nicht deterministisch ist.

Übung 5: Bonbonglas

Ein Bonbonglas ist mit vielen Lakritzeschnecken und Gummibärchen gefüllt. Außerdem liegt neben dem Glas eine "Wundertüte", die unerschöpflich viele Lakritzeschnecken enthält. Der folgende Vorgang soll nun solange ausgeführt werden, bis er sich nicht mehr wiederholen lässt:

1. Entnehmen Sie zwei beliebige Süßigkeiten aus dem Glas.

2. Falls beide die gleiche Geschmacksrichtung haben, essen Sie sie auf und füllen eine Lakritzeschnecke aus der nebenstehenden Tüte in das Glas.
3. Haben Sie eine Lakritzeschnecke und ein Gummibärchen gezogen, dürfen Sie nur die Lakritzeschnecke naschen. Das Gummibärchen müssen Sie wieder in das Glas zurücklegen.

Folgende Fragen drängen sich auf:

1. **Terminiert** dieser Vorgang?
2. Wenn ja, wie viele Süßigkeiten verbleiben im Glas?
3. Ist der Algorithmus **determiniert**?
4. Ist der Algorithmus **deterministisch**?

Hinweis: Wir empfehlen Ihnen, bei praktischen Untersuchungen des Problems die Werte für n (Anzahl der Gummibärchen) und m (Anzahl der Lakritzeschnecken) nicht zu groß zu wählen, da eine Magenverstimmung das Lösen der Aufgabe erheblich beeinträchtigt.

7 Anlagen

7.1 Literaturverzeichnis

- [Blieberger, 2005] J. Blieberger, B. Burgstaller, G.-H. Schildt; Informatik: Grundlagen (Springers Lehrbücher der Informatik); Springer Verlag; 5.Auflage, 2005.
- [Claus, 2006] V. Claus; Duden Informatik A–Z. Fachlexikon für Studium, Ausbildung und Beruf; Bibliographisches Institut und F.A. Brockhaus AG, 2006.
- [Ernst, 2016] Hartmut Ernst, Jochen Schmidt, Gerd Beneken; Grundkurs Informatik; Springer Verlag; 6.Auflage, 2016.
- [Knuth, 1997] Knuth D. E. The Art of Computer Programming. Volume 1. Fundamental Algorithms. Third Edition; Addison Wesley; 1997.
Knuth D. E. The Art of Computer Programming. Volume 2. Seminumerical Algorithms. Third Edition; Addison Wesley; 1997.
Knuth D. E. The Art of Computer Programming. Volume 3. Sorting and Searching. Third Edition; Addison Wesley; 1997.
- [Hopcroft, 2002] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman; Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie; Addison-Wesley; 2.Auflage; 2002.
- [Saake, 2002] Gunter Saake, Kai-Uwe Sattler; Algorithmen&Datenstrukturen, Eine Einführung mit Java; dpunkt Verlag; 2002.
- [Vossen, 2004] Gottfried Vossen, Kurt-Ulrich Witt; Grundkurs Theoretische Informatik; Vieweg Verlag; 3.Auflage; 2004.

7.2 Internetquellen

Automatenvisualisierungstool JFLAP

- <http://www.iflap.org/>

Onlinetools zum Debugging von regulären Ausdrücken

- <https://regex101.com/>
- <http://regexr.com/>

Tool zur Visualisierung von Turing Maschinen

- <http://sourceforge.net/projects/visualturing>

Links und Applets zum Traveling Salesman Problem

- <http://www.math.princeton.edu/tsp/>
- <http://www-e.uni-magdeburg.de/mertens/TSP/index.html>

7.3 Abbildungsverzeichnis

Abbildung 2-1 Zusammenhang Information und Daten	2—3
Abbildung 2-2 ASCII Table [http://www.asciitable.com/]	2—6
Abbildung 2-3 Unicode Symbolauswahl in Microsoft Office 2013	2—7
Abbildung 2-4 Logische Operatoren [Ernst, 2016]	2—8
Abbildung 2-5 Zahlengerade	2—15
Abbildung 2-6 Zahlenkreis Zweierkomplement	2—16
Abbildung 2-7: Nachrichtenübertragung [Ernst, 2016]	2—21
Abbildung 2-8 Logarithmusfunktion zur Basis 2	2—22
Abbildung 3-1 Kippschalter als endlicher Automat	3—29
Abbildung 3-2 Graphische Notation eines Automaten	3—36
Abbildung 3-3 Zustandsdiagramm eines DEA	3—37
Abbildung 3-4 Zustandsdiagramm für DEA aus dem Beispiel Automatendesign	3—40
Abbildung 3-5 Murmespiel als endlicher Automat	3—46
Abbildung 4-1 DEA für REXP $(0(0 1)^*)$	4—4
Abbildung 4-2 Beschreibungsformen für reguläre Sprachen	4—5
Abbildung 4-3 Ableitungsbaum	4—10
Abbildung 4-4 Mehrdeutige Grammatik	4—10
Abbildung 4-5 Anwendung von L-Systemen	4—12
Abbildung 4-6 Syntaxdiagramm für eine EBNF Grammatik	4—13
Abbildung 5-1 Maschinenmodell einer Turing Maschine	5—2
Abbildung 5-2 Beispiel Turing Maschine zum Löschen	5—4
Abbildung 6-1 Wichtigste Elemente von Flussdiagrammen [Ernst, 2016]	6—3
Abbildung 6-2 Verzweigungen in Flussdiagrammen [Ernst, 2016]	6—3
Abbildung 6-3 Wichtigste Elemente von Struktogrammen [Ernst, 2016]	6—4
Abbildung 6-4 Folge als Flussdiagramm, Struktogramm und Pseudocode	6—4
Abbildung 6-5 Bedingte Anweisung	6—5
Abbildung 6-6 einfache Alternative	6—5
Abbildung 6-7 mehrfache Alternative	6—5
Abbildung 6-8 Schleife mit vorausgehender Überprüfung	6—6
Abbildung 6-9 Schleife mit nachfolgender Überprüfung	6—6
Abbildung 6-10 Flussdiagramm Primzahlalgorithmus	6—7
Abbildung 6-11 TSP Rundreise	6—16
Abbildung 6-12 exakte Lösung eines TSPs für 15112 Städte	6—17