Grundlagen der Informatik

Lukas Prokop

12. August 2014

Inhaltsverzeichnis

Fehler bitte an `admin@lukas-prokop.at` melden. Danke!

Dieses Dokument wurde für die Lehrveranstaltung „Grundlagen der Informatik“ (VO, 716.232) (Technische Universität Graz, Prof. Wolfgang Slany) konzipiert und deckt die wesentlichen Inhalte ab. Es soll einerseits als Leitfaden zu den Inhalten und Themen dienen und andererseits an Beispielen Konzepte und Werkzeuge der Informatik verständlich machen. Es sei zu beachten, dass die Inhalte keineswegs die Themen so vollständig und tief erfassen wie sie für Wirtschaft und Forschung notwendig sind. Es sei ein Fokus auf die *Grundlagen der Informatik* gegeben.

Ich danke folgenden Personen, die durch Feedback zur Verbesserung des Dokuments beigetragen haben:

criscom, Frédéric Gierlinger, Peter Grassberger, Gernot Lecaks, Georg Regitnig, Wolfgang Slany, Karl Voit, Helmut Zöhrer

Kapitel 1

Grundlagen der Informatik

Informatik wird als Kombination aus zwei Wörtern beschrieben. Der *Information* und der *Automatik*. Der Wortstamm wurde in Sprachen wie Holländisch (informatica), Italienisch (informatica), Französisch (informatique) und Polnisch (informa-tyka) etabliert [Bal04, p. 21]. Im Englischen wird Informatik jedoch hauptsächlich mit dem Term *computer science* bezeichnet, welcher einen Fokus auf das Verb *to compute* (berechnen) und die Wissenschaft als solches legt. Wir möchten kurz erörtern in welcher Beziehung Informatik zur Automatik, Berechnung, Wissenschaft und der Mathematik steht.

Informatik ordnet sich ähnlich der Mathematik als Ideal- oder Formalwissenschaft ein. Bei dieser Form von Wissenschaft werden nicht wie bei Naturwissenschaften reale Vorgänge gemessen, analysiert und das zugrunde liegende Modell zur Erklärung des Vorgangs geschaffen. Es werden Anforderungen an die Informatik herangetragen, wobei die Modelle und Konzepte geschaffen werden, um diesen Anforderungen gerecht zu werden. Informatiker benötigen Fähigkeiten aus den Bereichen des systematischen Denkens, Modellieren, analytischen Lösens von Problemen und natürlich Kreativität, um diese Konzepte entwickeln zu können.

Als Grundbaustein der Informatik dient Information. Information wird als Summe von Daten (Sequenzen aus einem Alphabet) und Semantik (Bedeutung der Sequenzen durch seine Relationen) betrachtet, welche mit Maschinen automatisiert analysiert, bearbeitet und dargestellt wird, wobei sie wieder neue Information schafft. Die Automatik bzw. algorithmische Erfassung der notwendigen Vorgänge ermöglicht es uns die entwickelten Konzepte wiederzuverwenden und abstraktere Konzepte zu entwickeln.

Historisch betrachtet ist die Informatik eine junge Wissenschaft. Viele eigenständige Wissenschaften haben sich erst in den letzten Jahrhunderten etabliert. Der Einzug der Informatik in den Alltag datiert sich—je nach Auslegung—keine 100 Jahre zurück. Die Informatik nutzt dabei als Grundlage

viele Konzepte der Mathematik. Die zahlentheoretischen Zusammenhänge, das Transformieren von Zahlen mithilfe schrittweiser Anleitungen und das Berechnen eines Gesamtsystems basierend auf ein paar Grunddaten sind Konzepte, welche von der Mathematik übernommen wurden und daher finden sich unter den frühen Advokaten der Informatik einige bekannte Mathematiker.

Was versteht man jetzt als *Grundlagen* der Informatik? Wie jede Wissenschaft besitzt auch die Informatik Grundannahmen über die Welt, welche verwendet werden. Diese Grundannahmen werden als *Axiome* bezeichnet. Sie erlauben es komplexere Zusammenhänge abzuleiten. Als Beispiel nimmt die Mathematik (bzw. Arithmetik) etwa das Axiom `1 + 1 = 2` an. Die Informatik setzt etwa voraus, dass Zahlen in einer sequentiellen Datenstruktur persistent gespeichert werden können. Erst dadurch wird die Informatik praxisrelevant und erlaubt uns Berechnungen zu verwirklichen. Es sind die Axiome, die wir in diesem Dokument betrachten wollen.

Die Informatik als Wissenschaft wird gerne in 4 Bereiche gespaltet:

- Die *theoretische Informatik* befasst sich mit den theoretischen Grundlagen und Grenzen der Berechenbarkeit, versucht Probleme zu klassifizieren und besitzt Schnittstellen zur Logik, Linguistik und Algorithmentheorie (Komplexitätstheorie, Automatentheorie, Formale Sprachen, ...).

- Die *technische Informatik* versucht reale Umsetzungen zur Verfügung zu stellen, um Berechnungen zu ermöglichen und konzipiert die dafür notwendigen Maschinen. Sie besitzt insbesondere Schnittstellen zur Elektrotechnik bzw. Elektronik. Teilbereiche der technischen Informatik sind Prozessorarchitektur, Netzwerktechnik, Signalverarbeitung und hardwarenahe Systeme.

- Die *angewandte Informatik* versucht all jene Tätigkeiten zu automatisieren und zu verbessern, die ein Informatiker selbst im Alltag benötigt. Große Themenfelder dieses Bereiches sind etwa Programmiersprachen, Betriebssysteme oder Datenbanken. Da sie die Informatik selbst bedient, besitzt sie keine Schnittstellen zu anderen Wissenschaften; verwendet jedoch Elemente der Logik, Mengenlehre und Architektur von Maschinen.

- Die *praktische Informatik* versucht die Konzepte der Informatik in anderen Wissenschaften (etwa Wirtschaft, Medizin, Biologie) anzuwenden. Sie nimmt sich explizit der resultierenden Werkzeuge aus anderen Gebieten der Informatik an, um sie auf die Modelle anderer Wissenschaften anzupassen.

Kapitel 2

Bits und Bytes

10 is always the base

2.1 Zahlensysteme

Eine der bekannteren Grundlagen der Informatik ist, dass in der Informatik auch andere Zahlensysteme als das übliche Dezimalsystem zur Anwendung kommen. Vorgänge in der Software und Hardware lassen sich in jedem beliebigen Zahlensystem abbilden, da die Zahl als mathematisches Objekt verwendet wird und nicht über ihre Repräsentation. Es hat sich jedoch herausgestellt, dass das Dezimalsystem nicht für jeden Anwendungsbereich eignet ist.

Bei der Verwendung von mehreren Zahlensystemen hat es sich etabliert, die Basis (*Radix*) als Index neben die Repräsentation zu schreiben. Die Zahl 42 in Dezimaldarstellung wäre also mit 42_{10} und die Zahl 4 in Binärdarstellung wäre mit 100_2 zu notieren.

2.1.1 Terminologie

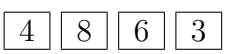


Abbildung 2.1: Zahlen als Sequenz von Ziffern.

Zahlensysteme sind so aufgebaut, dass eine Zahl als eine Sequenz von Ziffern beliebiger Länge dargestellt werden kann. Die Menge der Ziffern bildet dabei das sogenannte *Alphabet*.

2.1.2 Basis 10

Der Grund für das Entstehen des Dezimalsystems (Abk. `dec`) liegt in der Anzahl der Finger an beiden Händen eines Menschen. Bereits in der Volkschule lernen Kinder mithilfe der Finger die arithmetischen Operationen kennen und setzen sich dabei mit dem Dezimalsystem auseinander. Durch Wörter wie „zehn“ haben wir uns auch eigene Wörter geschaffen, die Zahlen in ihrer dezimalen Repräsentation bezeichnen.[1]

Das Dezimalsystem umfasst 10 verschiedene Ziffern.

$$\begin{aligned} \text{Alphabet}_{\text{dec}} &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ |\text{Alphabet}_{\text{dec}}| &= 10 \end{aligned}$$

Fraglich ist, welche die erste natürliche Zahl ist. Einerseits wird mit 1 zum Zählen begonnen und andererseits ist 0 die Ziffer mit dem niedrigsten Wert. Jedenfalls hat es sich seit der C-Programmiersprache in der überwiegenden Mehrheit der Programmiersprachen durchgesetzt *Indexing* (das Referenzieren eines Eintrags in einer Sequenz) mit Null beginnend durchzuführen („null-basierte Nummerierung“).[2]

2.1.3 Basis 2

Das Zahlensystem mit der Basis 2 wird als Binärsystem (Abk. `bin`) bezeichnet. Binärzahlen haben ihre starke Verbreitung durch die einfache technische Realisierung in der Elektronik erlangt. „Strom ein“ und „Strom aus“ sind zwei Zustände, die im Binärsystem abgebildet werden können. Die überwiegende Mehrheit der heutigen Maschinen implementiert binäre Operationen.

2.1.4 Basis 16 und 8

Das Hexadezimalsystem (Abk. `hex`) setzt 16 als Radix voraus. Das Besondere an 16 ist der Zusammenhang zum Binärsystem ($2^4 = 16$). Selbiges gilt für das Oktalsystem (Abk. `oct`, Basis 8), welches die Zahlen 0 bis 7 verwendet:

$$\begin{aligned} \text{Alphabet}_{\text{hex}} &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\} \\ |\text{Alphabet}_{\text{hex}}| &= 16 \end{aligned}$$

[1] Für die Aussprache einer Zahl in einem anderen Zahlensystem als dem üblichen Dezimalsystem gilt als Konvention, dass sie gleich der Aussprache der einzelnen Ziffern ist. So wird 100_2 etwa als „eins-null-null“ ausgesprochen.

[2] Ausnahmen (chronologisch betrachtet nach C) bilden etwa die Sprachen Lua, Mathematica, MATLAB und SQL

$$\begin{aligned} \text{Alphabet}_{\text{oct}} &= \{0, 1, 2, 3, 4, 5, 6, 7\} \\ |\text{Alphabet}_{\text{oct}}| &= 8 \end{aligned}$$

2.1.5 Stellenwertsystem

Die vorgestellte Definition, dass eine „Zahl eine Sequenz von Ziffern“ ist, basiert auf unserer Auffassung, dass Zahlen in einem Stellenwertsystem dargestellt werden. Beim Zählen reservieren wir eine *Stelle* und positionieren der Reihe nach die Ziffern mit ansteigendem Wert. Hat die Stelle den höchsten Wert des Alphabets erreicht, kommt es zum *Übertrag*. Dabei wird die nächste Stelle inkrementiert und die vorigen Stellen auf den niedrigsten Wert gesetzt. Mit diesem Konzept von Ersetzen & Übertragen können wir uns eine abzählbar unendliche Struktur bilden.

$$007 \xrightarrow{\text{Ersetzen}} 008 \xrightarrow{\text{Ersetzen}} 009 \xrightarrow{\text{Übertrag}} 010 \xrightarrow{\text{Ersetzen}} 011$$

Abbildung 2.2: Inkrementieren mit natürlichen Dezimalzahlen.

Die Dezimalzahlen seien in einer sequentiellen Datenstruktur gespeichert. Indexing innerhalb dieser Struktur n mit dem Index i erfolgt mit der Notation $n[i]$. Mit $n = 100_2 = \{1, 0, 0\}$ referenziert $n[0]$ die 1 und $n[2]$ die erste 0. Eine Zahl z lässt sich dann wie folgt als eine Summe darstellen:

$$z = \sum_{i=0}^{d} n[i] \cdot 10^i$$

Die Zahl 10 muss in der Formel entsprechend dem Radix für andere Zahlensysteme angepasst werden. Die Zahl aus Abbildung 2.1 kann somit auf folgende Weise aufgelöst werden.

$$4 \cdot 10^3 + 8 \cdot 10^2 + 6 \cdot 10^1 + 3 \cdot 10^0$$

$$4000 + 800 + 60 + 3$$

$$4863$$

2.1.6 Das Bit

Ein Bit (Abk. *BInary digiT*) bezeichnet eine Stelle in einem binären Zahlensystem. In einem Bit können zwei Zustände unterschieden werden. Die Anzahl der möglichen Zustände wächst logarithmisch zur Basis 2 (*logarithmus dualis*); wie in Tabelle 2.1 dargestellt. Der Zusammenhang zwischen der Anzahl der

Stellen (d)	mögliche Zustände (s)
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256

Tabelle 2.1: Entwicklung der Stellen im Vergleich zur Anzahl der Zustände.

abbildbaren Zustände s und der Anzahl der Ziffern d erklärt sich durch $s = 2^d$ bzw. logarithmisch mit $\log_2 s = d$.

Folgende Konvention hat sich etabliert: Eine Menge von 8 Bits wird als ein *Byte* bezeichnet. Ein Byte ist meist die kleinste Speichergröße, die von einer CPU in einem modernen Rechner angesprochen werden kann. Bekannt ist das Byte aus Einheiten wie Megabyte und Gigabyte, um die Speichergröße von Medien wie Caches, Festplatten und USB-Sticks anzugeben. In Tabelle 2.2 wird die Umrechnung der Speichergrößen in Bytes dargestellt.

Einheit	Bytes
1 KB (*Kilobyte*)	1000 Bytes
1 MB (*Megabyte*)	10^6 Bytes
1 GB (*Gigabyte*)	10^9 Bytes
1 TB (*Terabyte*)	10^{12} Bytes

Tabelle 2.2: Einheiten mit SI-Präfixen.

Leider sind Begriffe wie „Kilobyte“ in der Praxis nicht so eindeutig definiert, wie es wünschenswert wäre. Ein Kilo (2^{10}) weicht bei dieser Definition von Tabelle 2.2 um 24 Werte vom üblichen Kilo (10^3) ab. Es gab mehrere Versuche die Präfixe auf das Binärsystem anzupassen, allerdings spiegelt die genannten Tabelle die übliche Konvention wider.[3]

[3]Zur Unterscheidung wurde von der IEC der Präfix Kibi/Mebi/Gibi für die Basis 2 eingeführt.

2.1.7 Overflows und Underflows

Wir sprachen bei einer Zahl immer vom mathematischen Objekt Zahl. Dieses besitzt jedoch eine wichtige Einschränkung nicht: Länge. Unendliche Länge lässt sich nicht in der echten Welt realisieren und beschränkt uns in unserer Verwendung von Zahlen.

Daher müssen wir Zahlen auf eine Länge beschränken. Die Zahlen liegen dann an einem beliebigen Ort im Speicher. Eine Ganzzahl (engl. „Integer“) wird üblicherweise in einer Größe von 32bit[4] gespeichert.

1	1	0	1	+1 =
1	1	1	0	+1 =
1	1	1	1	+1 =
0	0	0	0	

Abbildung 2.3: Illustration eines Overflow in der letzten Zeile.

Aufgrund der beschränkten Länge tritt das Problem des Overflows bzw. Underflows auf. Ein Integer bestehend aus nur 4 Bits, kann 16 ($2^4 = 16$) Zustände annehmen. Der zugehörige Wertebereich reicht von 0 bis 15. Wird nun dieser Wert inkrementiert, kommt es zum Übertrag, wobei keine neue Stelle zur Verfügung steht. In Folge dessen geht der neue Wert der zusätzlichen Stelle verloren. Die restlichen Stellen werden entsprechend dem Übertrag auf die niedrigste Zahl zurückgesetzt. Statt 16 erhalten wir 0 (siehe Abbildung 2.3).

Der Underflow beschreibt das Äquivalent für das Dekrementieren. Der Over- und Underflow kann durch Techniken wie Ganzzahlarithmetik vermieden werden, doch aufgrund des Geschwindigkeitsverlustes und der Komplexität verzichtet man in maschinennahen Umgebungen darauf. Dynamische Programmiersprachen (etwa Python und ruby) bieten oft Ganzzahlen mit unendlicher Genauigkeit an[5] und verstecken die Umsetzung auf maschinennahe Zahlen mit beschränkter Länge.

2.2 Konvertierung zwischen Zahlensystemen

Liegt eine Repräsentation einer Zahl vor, so lässt sie sich in ein beliebiges anderes Zahlensystem mit Stellenwertordnung umrechnen, indem man eines der hier vorgestellten Verfahren verwendet.

[4] Wie wir jetzt wissen, sind das 4 Bytes. Diese können 4294967296 (ca. 4.3 Milliarden) verschiedene Zustände annehmen.

[5] Nur beschränkt durch die Größe des Hauptspeichers.

Moduloansatz über die Basis 10

Die einfachste Variante besteht darin die Zahl pro Stelle aufzulösen, sie in unsere Basis 10 zu bringen und dann wieder entsprechend der Potenzen aufzubauen.

$$42_{16}$$

Wir spalten die Zahl in ihre Potenzdarstellung[6] auf:

$$4 \cdot 16^1_{10} + 2 \cdot 16^0_{10}$$

Wir rechnen diese Zahl aus:

$$64_{10} + 2_{10} = 66_{10}$$

Wir legen für 66_{10} eine Zahl 8^0 $(= 1)$ an und rechnen mit der Basis 10 weiter. Der Faktor ist das Ergebnis der Operation $\frac{(66)}{8^0}$ mod 8^1. Bei uns also 2.

$$66 = \ldots + 2 \cdot 8^0$$

Wir legen 8^1 $(= 8)$ an. Das Ergebnis der Operation $\frac{(66-2\cdot 8^0)}{8^1}$ mod 8^1 ist 0.

$$66 = \ldots + 0 \cdot 8^1 + 2 \cdot 8^0$$

Wir legen 8^2 $(= 64)$ an. Das Ergebnis der Operation $\frac{(66-2\cdot 8^0-0\cdot 8^1)}{8^2}$ mod 8^2 ist 1.

$$66 = 1 \cdot 8^2 + 0 \cdot 8^2 + 2 \cdot 8^0$$

Da $66 - 2 \cdot 8^0 - 0 \cdot 8^1 - 0 \cdot 8^2 = 0$ sind wir am Ziel angekommen:

$$102_8$$

Der Kettenansatz

Zahlen lassen sich als verschachtelte Kette darstellen. Wir spalten dabei immer einen Restwert in einer alternierenden Sequenz von Addition und Multiplikation ab.

$$59_{10}$$

$$(58 + 1)$$

$$((29) \cdot 2 + 1)$$

$$(((14) \cdot 2 + 1) \cdot 2 + 1)$$

[6] auch „Wissenschaftliche Schreibweise“ genannt

$$((((7) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1)$$
$$(((((3) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1)$$
$$((((((1 \cdot 2 + 1) \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1) \cdot 2 + 1)$$
$$\Rightarrow 111011_2$$

Der Tabellenansatz

Der Tabellenansatz erlaubt einen einfachen Umgang und stellt die explizite Frage „Brauche ich diesen Wert, um meinen Zielwert zu erreichen?“. Erfolgt die Antwort mit „Ja“, wird eine 1 notiert; sonst 0. Das Funktionsprinzip wird in Tabelle 2.3 dargestellt. Wir konvertieren die Zahl 59_{10} ins Binärsystem. Dazu tragen wir auf einer Linie die 2er-Potenzen absteigend auf (die höchste Potenz muss die größte Potenz sein, die kleiner-gleich der gesuchten Zahl ist). Wir bearbeiten die Zahlen von links nach rechts. Wir fragen uns, ob die Zahl 32 kleiner oder gleich 59 ist (d.h. ob sie „gebraucht“ wird). Wenn ja, tragen wir darunter eine 1 ein. Wenn nein, tragen wir darunter eine 0 ein. Wir fragen uns, ob die Zahl $32 + 16$ kleiner 59 ist. Ja, daher 1. $32 + 16 + 8$ ist auch kleiner 59 und daher tragen wir eine 1 ein. $32 + 16 + 8 + 4$ wäre jedoch 60 und überschreitet unseren Wert 59. Wir lehnen diese 2er-Potenz mit einer 0 ab. Mit $32 + 16 + 8 + 2 + 1$ erreicht die Summe genau unsere Zahl 59; in binär 111011_2.

59	32	16	8	4	2	1
	1	1	1	0	1	1

Tabelle 2.3: Tabellenansatz.

Stellenweise Methode

Es sei d der Zielradix und s der Ausgangsradix. Ist $s > d$ und ist s ein Vielfaches von d können wir die Zahl pro Stelle konvertieren. Dabei wird eine Stelle durch $\frac{\log s}{\log d}$ Stellen ersetzt.

Wir beschreiben die Bedingung informal: Erreicht man den Zielradix durch Potenzieren des Ausgangsradix (etwa $2^n = 16$ mit $n = 4$ oder $3^n = 9$ mit $n = 2$), können wir jede Ziffer der Ausgangszahl verwenden und mit n Stellen in die Zielbasis bringen und schlussendlich die Einzelzahlen aneinander fügen (*konkatenieren*).

Als Beispiel betrachten wir in 2.4 die Konvertierung von FACE_{16} ins Binärsystem. So wird etwa F mit $n = 4$ binär durch 1110 dargestellt.

0xFACE			
F	A	C	E
1110	1010	1100	1110

Tabelle 2.4: Stellenweise Methode.

2.3 Arithmetische Operationen

Moderne Rechenmaschinen nutzen ALUs („arithmetic logical unit“), um Rechenoperationen mit Bytes durchzuführen. Diese umfassen etwa:

- Addition
- Subtraktion
- Multiplikation
- Division

Die arithmetischen Operationen lassen sich mit den bekannten Methoden in allen Zahlensystemen rechnen und werden daher hier nicht näher erörtert. Es sei nur erwähnt, dass es auf einer Maschine zu Fehlerfällen kommen kann wie etwa eine Division durch Null.

2.4 Bitweise Operationen

Eine ALU kann auch Operationen durchführen, die auf die einzelnen Bits angewandt werden:

- Und-Verknüpfung
- Oder-Verknüpfung
- Negation
- Exklusive Oder-Verknüpfung

Die Operationen werden im Kapitel 3 behandelt.

Beachte, dass Bits genau die beiden aussagenlogischen Ausdrücke *wahr* und *falsch* repräsentieren können, jedoch diese Werte praktisch nie auf modernen Maschinen durch wirkliche Bits gespeichert werden, da (wie erwähnt) die kleinste adressierbare Speichereinheit ein Byte ist.

Kapitel 3

Aussagenlogik

Die Aussagenlogik stellt die Basis jeder Form von Logik dar, wie sie in der Informatik, Mathematik, Philosophie und weiteren Wissenschaften zur Anwendung kommt. Im folgenden Kapitel werden die Grundbegriffe und Grundkonzepte beschrieben mit denen die Aussagenlogik formalisiert wird. In den Lehrveranstaltungen „Diskrete Mathematik“ (503.007) und „Logik und Berechenbarkeit“ (705.033, Informatikstudium)[1] wird die Aussagenlogik tiefgehender behandelt und als Basis für weitere Logikformen wie der Prädikatenlogik verwendet. Die übergeordnete first-order logic (FOL) findet in der Praxis am meisten Verwendung.

Eine Aussage ist ein aussagenlogischer Ausdruck, der wahr oder falsch ist. Diese Ausdrücke lassen sich beliebig verketten, wodurch wie in der Arithmetik beliebig lange Ausdrücke entstehen. Verkettete Aussagen kommen durch die Verknüpfung von Aussagen mithilfe von Operatoren zustande[2]. Eine Aussage kann aber auch ein Wahrheitswert (wahr oder falsch) oder eine Variable (Literal, frei zuweisbarer Wert) sein. Weitere Notation für die Werte wahr und falsch sind $\{1, 0\}$, $\{W, F\}$ (Deutsch), $\{T, F\}$ (Englisch) oder $\{\top, \bot\}$. In diesem Dokument wird bevorzugt 1 und 0 verwendet.

3.1 Wahrheitstabellen und Verknüpfungen

Es gibt 3 Grundverknüpfungen, auf die aussagenlogischen Ausdrücke oft reduziert werden: Das *Oder* ($\vee$, „Disjunktion“, Tabelle 3.1) wird wahr, wenn eines oder beide Argumente wahr sind. Das *Und* ($\wedge$, „Konjunktion“, Tabelle 3.2) wird wahr, wenn beide Argumente einzeln wahr sind. Die *Negation* ($\neg$) ist eine unäre Operation (d.h. sie bezieht sich nur auf 1 Argument) und gibt

[1] Lehrveranstaltungen an der Technischen Universität, Graz

[2] Aussagenlogische Ausdrücke sind somit rekursive Strukturen

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 3.1: Wahrheitstabelle einer Oder-Verknüpfung.

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 3.2: Wahrheitstabelle einer Und-Verknüpfung.

A	$\neg A$
0	1
1	0

Tabelle 3.3: Wahrheitstabelle einer Nicht-Verknüpfung.

statt einem wahr falsch zurück und vice versa (Tabelle 3.3).

Die Tabellen stellen Wahrheitstabellen dar; Tabellen, die die einzelnen Variablenbelegungen angeben sowie das Ergebnis (und eventuell auch Teilergebnisse) des aussagenlogischen Ausdrucks. Mit diesen drei Operatoren lassen sich alle aussagenlogische Ausdrücke formulieren (d.h. jede beliebige Kombination von Wahrheitswerten in einer Wahrheitstabelle).

Zwei aussagenlogische Ausdrücke sind äquivalent, wenn deren Wahrheitstabellen äquivalent sind.

3.2 Weitere Operatoren

3.2.1 Exklusives Oder (XOR)

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 3.4: Wahrheitstabelle einer XOR-Verknüpfung.

XOR ($\oplus$, exklusives Oder, Kontravalenz, Abb. 3.4) ist genau dann wahr, wenn nur genau einer der Werte wahr ist. Dies entspricht dem natürlichsprachlichen „Entweder... oder...“.

$$A \oplus B \quad \leftrightarrow \quad (A \vee B) \wedge (\neg A \vee \neg B)$$

Das Besondere an XOR ist seine Bijektivität. Durch zweifache Anwendung des XORs mit einem Schlüssel wird der selbe Wert wieder retourniert. Deshalb kommt XOR etwa in der Kryptographie sehr oft zum Einsatz. Aspekte der Kryptographie (etwa der AES-Algorithmus, der im AddRoundKey Schritt eine XOR-Verknüpfung durchführt) werden in der Lehrveranstaltung „Einführung in die Informationssicherheit“ (705.026) behandelt.

Interessant ist weiters, dass ein 3-stelliges XOR diese Eigenschaft verliert. Ein 3-stelliges XOR ist nicht nur dann wahr, wenn eine der Variablen wahr

0	1	1	0	1	1	Wert
1	1	0	0	0	1	Schlüssel
1	0	1	0	1	0	XOR-Ergebnis
1	1	0	0	0	1	Schlüssel
0	1	1	0	1	1	XOR-Ergebnis = Wert

ist, sondern auch wenn alle Variablen wahr werden.

$$0 \oplus 0 \oplus 0 = 0$$
$$0 \oplus 0 \oplus 1 = 1$$
$$0 \oplus 1 \oplus 0 = 1$$
$$1 \oplus 0 \oplus 0 = 1$$
$$0 \oplus 1 \oplus 1 = 0$$
$$1 \oplus 1 \oplus 0 = 0$$
$$1 \oplus 0 \oplus 1 = 0$$
$$1 \oplus 1 \oplus 1 = 1$$

3.2.2 Implikation

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Tabelle 3.5: Wahrheitstabelle einer Implikation.

Eine *Implikation* ($\rightarrow$ oder $\Longrightarrow$, Subjunktion, Abb. 3.5) wird so interpretiert, dass der zweite Ausdruck verwendet wird, wenn der erste Ausdruck wahr ist. Dies entspricht dem natürlichsprachlichen „Wenn... dann...“.

$$A \rightarrow B \quad \leftrightarrow \quad \neg A \vee B \tag{3.1}$$

Ist die Vorbedingung A *nicht* erfüllt, so ist die Implikation bedingungslos wahr. Dies kann als unintuitiv betrachtet werden, stellt sich jedoch als sehr praktikabel heraus, wenn man logische Systeme modelliert. Bei einem Pfeil in die linke Richtung ($\leftarrow$) handelt es sich auch um eine Implikation, jedoch wird die Position der Argumente im Vergleich zum rechten Pfeil vertauscht.

3.2.3 NAND

A	B	$A \mid B$
0	0	1
0	1	1
1	0	1
1	1	0

Tabelle 3.6: Wahrheitstabelle einer NAND-Verknüpfung.

Das NAND ($\mid$ oder $\uparrow$, Shefferscher Strich) ist die Negation des Und-Operators. Als Besonderheit lassen sich aus diesem Operator alle aussagenlogische Ausdrücke ableiten. Somit können sie darauf verzichten die aussagenlogischen Ausdrücke auf die Operatoren $\wedge$, $\vee$ und $\neg$ zu reduzieren, sondern können direkt nur das NAND nutzen, um alle Ausdrücke abzubilden. So wird als Beispiel in Formel 3.2 das NAND verwendet, um den Wahrheitswert falsch abzuleiten.

$$(A \mid A) \mid B \leftrightarrow \bot \tag{3.2}$$

3.2.4 Bijunktion

A	B	$A \leftrightarrow B$
0	0	1
0	1	0
1	0	0
1	1	1

Tabelle 3.7: Wahrheitstabelle einer Bijunktion.

Die Bijunktion ($\leftrightarrow$, Äquivalenz) wird genau dann wahr, wenn beide Argumente äquivalent sind. Sie entspricht dem natürlichsprachlichen „sein“ und wird verwendet, um die Äquivalenz zweier aussagenlogischer Ausdrücke zu notieren. Interessant ist, dass es sich um die Negation einer XOR-Verknüpfung handelt.

$$A \leftrightarrow B \quad \leftrightarrow \quad (A \rightarrow B) \wedge (B \rightarrow A) \tag{3.3}$$

3.3 Tautologie

Unter einer Tautologie ($\top$) versteht man eine boolsche Funktion, die unter einer beliebigen Variablenkonfiguration zum Ausdruck Wahr evaluiert. Sie ist damit bedingungslos. Da der Wahrheitswert selbst als boolsche Funktion interpretiert werden kann, wird das Symbol $\top$ oft synonym verwendet.

$$((A \vee B) \vee (B \wedge A)) \vee (\neg A \wedge \neg B) \leftrightarrow \top \tag{3.4}$$

3.4 Widerspruch

Unter einem Widerspruch ($\bot$, Kontradiktion) versteht man eine boolsche Funktion, die unter einer beliebigen Variablenkonfiguration zum Ausdruck falsch evaluiert. Sie ist bedingungslos. Sie stellt ein wichtiges Werkzeug bei der mathematischen Beweisform *Beweis durch Widerspruch* dar. In einigen formalen Systemen ist auch der Spruch „ex falso quodlibet“ relevant, welcher repräsentiert, dass aus einem Widerspruch Beliebiges geschlossen werden kann.

$$((A \vee B) \vee (B \wedge A)) \wedge (\neg A \wedge \neg B) \leftrightarrow \bot \tag{3.5}$$

3.5 Operatorenpräzedenz

Ausdrücke könnten mehrdeutig sein, wenn die Klammerung die Reihenfolge der Evaluierung der Operatoren nicht klar vorgibt.

$$A \vee B \wedge \neg C \rightarrow D$$

Die Operatorenpräzedenz (engl. „operator precedence“) sind semantische Regeln, welche Operatoren am stärksten binden. Je stärker ein Operator bindet, desto früher wird er evaluiert. Es gibt unterschiedliche Auffassungen und sie werden nicht so konsistent verwendet, wie hier dargestellt, doch stellt die Tabelle 3.8 eine Zusammenfassung des allgemeinen Konsens dar. Allgemein binden binäre (= zweistellige) Operatoren schwächer als unäre (= einstellige) und typischerweise bindet das Und stärker als das Oder. Während alle Operatoren links-assoziativ sind, bildet die Implikation mit einer Rechtsassoziativität eine Ausnahme:

$$(a \rightarrow b \rightarrow c) \quad \leftrightarrow \quad (a \rightarrow (b \rightarrow c)) \tag{3.6}$$

Klammern sind ein Ausweg, um Unklarheiten in der Präzedenz zu deklarieren und werden auf dem selben Weg in der Mathematik verwendet. Die

$\neg$	Negation
$\wedge$	Konjunktion
$\vee$	Disjunktion
$\rightarrow$	Implikation

Tabelle 3.8: Operatorenpräzedenz für aussagenlogische Operatoren (Operatoren weiter oben binden stärker).

aussagenlogische Formel von vorhin besitzt folgende äquivalente Klammerung:

$$((A \vee (B \wedge (\neg C))) \rightarrow D) \tag{3.7}$$

3.6 Spezielle Strukturen

3.6.1 Klauseln und Monome

Unter einer *Klausel* (Disjunktionsterm) versteht man eine Oder-verknüpfte Liste von Literalen. Das konjunktive Äquivalent bezeichnet man als Monom (Konjunktionsterm). Eine n-Klausel ist eine Disjunktion aus n Literalen.

Hornklauseln finden in der logischen Programmierung Anwendung und seien deshalb hier auch kurz erwähnt: Hornklauseln sind Klauseln, die maximal einen negationsfreien Literal besitzen.

Typ	Beispiel (jeweils $\{A, B, C, D, E\} \in \{1, 0\}$)
Klausel	$A \vee B \vee C \vee D \vee E$
3-Klausel	$A \vee B \vee C$
Monom	$A \wedge B \wedge C \wedge D$
Hornklausel	$A \vee \neg B \vee \neg C \vee \neg D$

Tabelle 3.9: Beispiele für Klauseln und Monome.

3.6.2 Konjunktive Normalform (KNF)

Unter der konjunktiven Normalform (engl. „conjunctive normal form“, CNF) versteht man einen aussagenlogischen Ausdruck, der aus einer beliebigen Anzahl von UND-verknüpften Klauseln gebildet wird. Ein Beispiel ist in

Formel 3.8 gegeben.

$$\bigwedge_i \bigvee_j x_{ij} \qquad \text{mit } x_{ij} \text{ als positives oder negatives Literal}$$

$$(a \vee b) \wedge (\neg a \vee c \vee d) \wedge (b \vee c) \tag{3.8}$$

Jede beliebige boolsche Funktion lässt sich als KNF darstellen. Wir befassen uns mit dieser Aussage, indem wir eine Funktion hernehmen und sie in eine KNF transformieren. In Tabelle 3.10 wird die boolsche Funktion $f_0 : ((A \vee B) \wedge C) \vee (A \wedge B)$ in eine KNF transformiert. Dass die Funktion f_0 mit der korrespondierenden KNF übereinstimmt (und wir daher keinen Fehler gemacht haben) lässt sich überprüfen indem wir von beiden Funktionen die Wahrheitstabellen bilden und vergleichen. Wir stellen fest, dass die Wahrheitstabellen beide jener in Tabelle 3.11 entspricht.

Es lässt sich auch ein allgemeiner Algorithmus angeben aus dem die KNF eines beliebigen Ausdrucks gebildet werden kann:

1. Erstelle die Wahrheitstabelle.
2. Wähle alle Konfigurationen unter denen die Formel zu falsch evaluiert.
3. Negiere alle Variablen.
4. Verbinde alle Variablen einer Konfiguration mit ODERs.
5. Verbinde alle Konfigurationen mit mehreren UNDs.

Dieser Algorithmus wurde in Tabelle 3.12 auf Basis der Wahrheitstabelle 3.11 angewandt.

3.6.3 Disjunktive Normalform

Unter der disjunktiven Normalform versteht man einen aussagenlogischen Ausdruck, der aus eine beliebigen Anzahl von Oder-verknüpften Monomen gebildet wird.

$$\bigvee_i \bigwedge_j x_{ij} \qquad \text{mit } x_{ij} \in \{1, 0\} \text{ oder } \neg x_{ij}$$

Äquivalent zur KNF kann die DNF gebildet werden, indem Variablenbelegungen (die zu wahr evaluieren) disjunktiert werden; allerdings nicht negiert.

$$((A \vee B) \wedge C) \vee (A \wedge B)$$

$$[X \vee (Y \wedge Z) \Leftrightarrow (X \vee Y) \wedge (X \vee Z)]$$

$$((A \vee B) \vee (A \wedge B)) \wedge ((A \wedge B) \vee C)$$

[Tauschen]

$$((A \wedge B) \vee C) \wedge ((A \wedge B) \vee (A \vee B))$$

$$[X \vee (Y \wedge Z) \Leftrightarrow (X \vee Y) \wedge (X \vee Z)]$$

$$((A \vee C) \wedge (B \vee C)) \wedge ((A \wedge B) \vee (A \vee B))$$

$$[((X \wedge Y) \vee (X \vee Y)) \Leftrightarrow (X \vee Y)]$$

$$(A \vee C) \wedge (B \vee C) \wedge (A \vee B)$$

Tabelle 3.10: Transformation von f_0 in eine KNF.

A	B	C	$((A \vee B) \wedge C) \vee (A \wedge B)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabelle 3.11: Wahrheitstabelle der Funktion f_0.

Funktion	Typ
$((A \vee B) \wedge C) \vee (A \wedge B)$	f_0
$(A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee B \vee C)$	KNF
$(\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C)$	DNF

Tabelle 3.12: Konstruktion der KNF/DNF durch Disjunktion/Konjunktion der Variablenbelegungen, die zu wahr/falsch führen.

3.7 Gödelscher Unvollständigkeitssatz

> In jedem formalen und logischen System gibt es Aussagen, die unentscheidbar sind.

Kurt Gödel (* 1906 † 1978) beschrieb die sogenannten Unvollständigkeitssätze, die die Grenzen von formalen Systemen aufzeigen [Gö31]. In der Aussagenlogik möchten wir uns mit Unvollständigkeit beschäftigen, indem wir das Lügnerparadoxon betrachten:

> Diese Aussage ist falsch.

Handelt es sich um einen Ausdruck, der gesamtheitlich als 1 angenommen wird, so muss die Summe aller Teilaussagen als wahrheitsgemäß angesehen werden d.h. „Diese Aussage ist falsch“ wird zu 1 evaluieren. Da sich die Aussage auf den Gesamtausdruck bezieht, müsste dieser Ausdruck zu 0 evaluieren, was sich mit unserer Annahme widerspricht. Das vorliegende Problem kann durch die Selbstreferenzierung nicht gelöst werden. Es ist somit nicht entscheidbar.

Selbige Überlegung lässt sich mit 0 durchführen. Im Gesamten kann somit die Aussage nicht entschieden werden. Das Lügner-Paradoxon bildet ein Beispiel des Unvollständigkeitssatzes in der Logik.

Kapitel 4

Mengenlehre

Die Mengenlehre ist ein Teilgebiet der Mathematik und beschäftigt sich mit Sammlungen von unterscheidbaren Objekten. Da in der Informatik das Konzept der Mengen oft wiederverwendet wird, sei eine oberflächliche, aber präzise Betrachtungsweise gegeben.

Eine *Menge* (engl. Set) ist

- ungeordnet
- jedes Element unterscheidet sich von anderen Elementen in dieser Menge und kommt daher nur einmal vor
- besitzt eine beliebige Anzahl von Elementen

Die *leere Menge* wird mit $\varnothing$ oder $\{\}$ bezeichnet. Die Anzahl der Elemente einer Menge nennt man auch *Kardinalität einer Menge*. Zur Notation der Kardinalität werden vertikale Balken um die Menge verwendet. So haben wir etwa in Abschnitt 2.1.2 von $|\text{Alphabet}_{\text{dec}}|$ gesprochen, womit die Anzahl der Elemente der Menge Alphabet im Dezimalsystem gemeint ist. Die Definition der Mengen erlaubt sowohl endliche als auch unendliche[1] Mengen. Mengen werden wie Matrizen per Konvention mit einem Großbuchstaben bezeichnet.

$$A = \{1, 2, 3, 4\}$$

$$B = \{\}$$

[1] Die Diskussion um die Definition eines Unendlichkeitsbegriffs und der Abgrenzung zum Begriff „endlich abzählbar“ entstammt direkt der Mengenlehre. Als guter Einstieg in die Thematik dient der Satz von Cantor.

4.1 Mengennotation

Wir stellen fest dass geschwungene Klammern genutzt werden, um Mengen zu definieren. Innerhalb dieser Klammern sind die Elemente dieser Menge kommagetrennt aufgelistet. Diese Notation reicht jedoch nicht immer aus, um beliebige Mengen zu spezifizieren.

Die Mengennotation (engl. „set-builder notation“, „set comprehension“) erlaubt es uns zwischen den geschwungenen Klammern einen vertikalen Balken (oder einen Doppelpunkt) zu verwenden. Wenn die Kriterien auf der rechten Seite des Balkens erfüllt sind, wird das Element auf der linken Seite aufgenommen. Domänenkriterien (zB. $t \in \mathbb{N}$) dürfen jedoch auch auf der linken Seite genannt werden, wenn sie sich nur auf das auszuwählende Element beziehen. Auf diesem Weg können wir auch unendliche Mengen abbilden.

$$C = \{x^2 | x \in \mathbb{N}\}$$

$$D = \{x | x^2 \mod p = 0, x \in \mathbb{N}\}$$

$$E = \{x \in \mathbb{N} | x^2 \mod p = 0\}$$

4.2 Abgrenzungen zu ähnlichen Begriffen

Entsprechend der genannten Definition darf jedes Element genau nur einmal vorkommen. Obwohl diese Definition oft respektiert wird, wird sie im Allgemeinen inkonsistent verwendet. So wird auch die Sequenz $(\{1, 2, 3, 2\})$ als Menge bezeichnet. Die Frage ist, wie sich die Begriffe Liste, Sequenz, Tupel und Menge voneinander abgrenzen. Wir möchten hier einen abstrakten Überblick geben, da die Unterschiede zur Definition von Datenstrukturen in der Programmierung relevant sind.

Liste Eine Liste ist eine geordnete Sammlung von beliebig vielen Elementen.

Sequenz Eine Sequenz ist eine geordnete Sammlung von beliebig vielen Elementen, die meistens auf eine endliche Größe beschränkt sind.

Tupel Eine geordnete Sammlung von homogenen Elementen endlicher Anzahl.

Menge Ungeordnete Sammlung von unterschiedlichen Objekten.

Alle Begriffe erlauben grundsätzlich Verschachtelungen (zB Liste in Liste), werden jedoch oft nur eindimensional verwendet.

4.3 Elementare Mengenoperationen

4.3.1 Vereinigung

Gegeben seien 2 Mengen A und B. Die *Vereinigung* (engl. „union“) der Mengen A und B ist die Menge der Elemente, die in A *oder* B vorkommen. Dies bedeutet wir können den logischen Operator Oder verwenden, um aufgenommene bzw. abgelehnte Elemente der Vereinigungsmenge zu spezifizieren. Hier ergibt sich eine Schnittstelle zwischen Logik und Mengenlehre:

$$A \cup B = \{x \mid (x \in A) \vee (x \in B)\}$$

4.3.2 Durchschnitt oder Schnittmenge

Gegeben seien 2 Mengen A und B. Der *Durchschnitt* (engl. „intersection“) der Menge A und B ist die Menge jener Elemente, die in A *und* B vorkommen:

$$A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}$$

4.3.3 Komplement oder Differenz

Gegeben seien 2 Mengen A und B. Das *relative Komplement* der Menge A und B ist die Menge jener Elemente, die in A, aber nicht in B, vorkommen.

$$A \setminus B = \{x \mid (x \in A) \wedge (x \notin B)\}$$

4.3.4 Teilmenge

Die Teilmenge beschreibt eine Relation und erzeugt keine neue Menge. Ihr Ergebnis ist also eine „Ja“ oder „Nein“ Antwort. Eine Menge ist eine Teilmenge einer anderen Menge genau dann wenn alle Elemente der Teilmenge in ihr enthalten sind. A ist eine Teilmenge von B genau dann wenn $A \subseteq B$:

$$A \subseteq B \text{ iff } \forall x \in A, x \in B$$

Wir grenzen hierbei „echte Teilmengen“ ($A \neq B$, Operator $\subset$) von „Teilmengen“ ($A = B \vee A \subset B$, Operator $\subseteq$) ab.

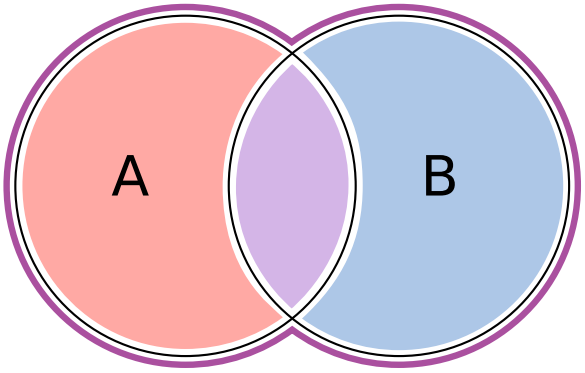


Abbildung 4.1: Illustration der Vereinigungsmenge (rote Menge A, blaue Menge B, Vereinigungsmenge $A \cup B$ als violette Umrandung).

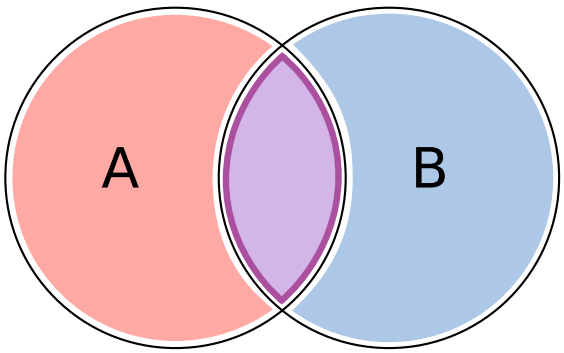


Abbildung 4.2: Illustration des Durchschnitts (rote Menge A, blaue Menge B, violetter Durchschnitt $A \cap B$).

4.3.5 Mitgliedschaft

Eine weitere Relation ist die Mitgliedschaft. Für jede Menge kann getestet werden, ob ein Element enthalten ist. Als Beispiel sei die Menge $C = \{x^2 | x \in \mathbb{N}\}$ gegeben. Darin sind etwa die Elemente $1, 4$ und 81 enthalten. Die Frage der Mitgliedschaft lautet: $x \in C$ für ein bestimmtes x?

$$4 \in C \qquad 42 \notin C \qquad 441 \in C$$

4.4 Darstellung von Mengen

Möchte man Mengen und deren Relationen darstellen, bieten sich Venn-Diagramme an, die eine intuitive Übersicht bieten. In Darstellung 4.4 können wir auch ein Beispiel des Inklusion-Exklusion Prinzips sehen. Dieses besagt, dass wir für die Beschreibung der Kardinalität von A, B *und* C die einzelnen Kardinalitäten der Mengen A, B und C betrachten müssen, die paarweisen Schnittmenge zu exkludieren sind und anschließend die Schnittmenge aller drei Mengen wieder zu inkludieren ist. Formal:

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

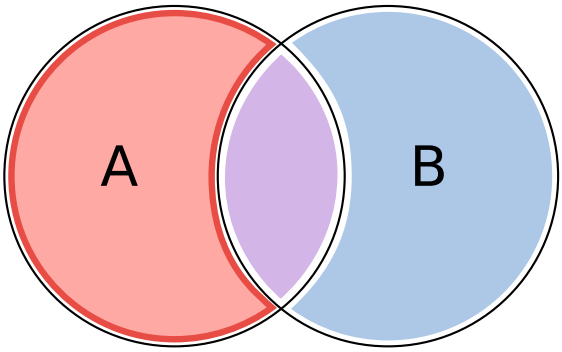


Abbildung 4.3: Illustration des Komplements (rote Menge A, blaue Menge B, rotes Komplements $A \setminus B$).

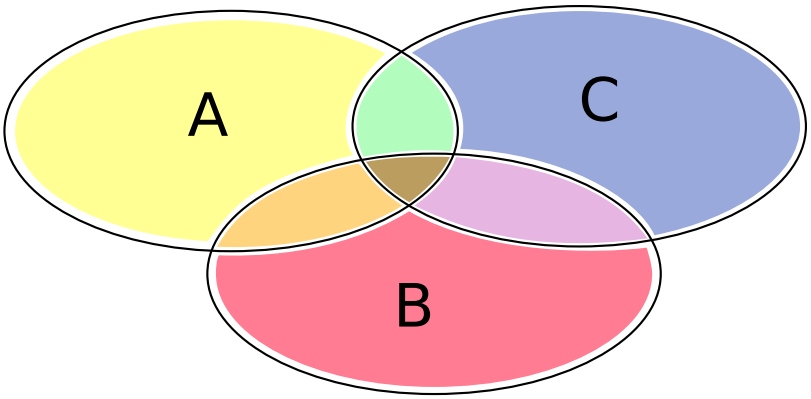


Abbildung 4.4: Venn Diagramm Beispiel.

Kapitel 5

Turingmaschinen

Alan Turing (* 1912 † 1954) machte sich viele Gedanken rund um mathematische Beweise und mechanische Prozesse. Er fragte sich, ob eine automatisierte Beweisführung von mathematischen Formeln möglich ist und konzipierte hierfür ein Maschinenkonzept. Die *a-machine* [Tur37] (oder heutzutage als *Turingmaschine* bezeichnet) wurde zum Referenzmodell der Theoretischen Informatik, welche auf einem praktikablen Weg zeigt, wo die Grenzen von Berechenbarkeit liegen und was automatisiert werden kann. Sie veranlasste auch John von Neumann zur Konzeption der noch heute maßgeblichen Computerarchitektur. Die Church-Turing-These behauptet, dass auch andere Konzepte wie das λ-Kalkül von Alonzo Church (welches zur selben Zeit entstanden ist) bezüglich Berechenbarkeit äquivalent sind:

> Die Klasse der intuitiv berechenbaren Funktionen ist genau die Klasse der Turing-berechenbaren (d.h. durch eine Turingmaschine berechenbare) Funktionen.
> —Die Church-Turing-These

5.1 Definition

Eine Turingmaschine ist formal betrachtet ein 7-Tupel.

$$\mathrm{TM} = (Q, \Gamma, b, \Sigma, \delta, q_0, F) \tag{5.1}$$

Q Eine endliche, nicht-leere Menge an Zuständen, die die Turingmaschine annehmen kann.

Γ Eine endliche, nicht-leere Menge an Zeichen, die auf dem Band verwendet werden können.

b Das Blanksymbol, welches den Initialzustand unbeschriebener Stellen beschreibt.

Σ Eine endliche Menge an Zeichen, welche auf dem Band vorhanden sind.

δ Die Übergangsfunktion.

q_0 Ein Anfangszustand der TM.

F Eine Menge von Endzuständen. Wird einer dieser Zustände erreicht, hält die Turingmaschine an.

Dabei gelten folgende Relationen zwischen den Elementen:

$$b \in \Gamma \setminus \Sigma \qquad \Sigma \subset \Gamma \qquad b \in \Gamma$$

$$q_0 \in Q \setminus F \qquad F \subset Q$$

5.2 Funktionsweise

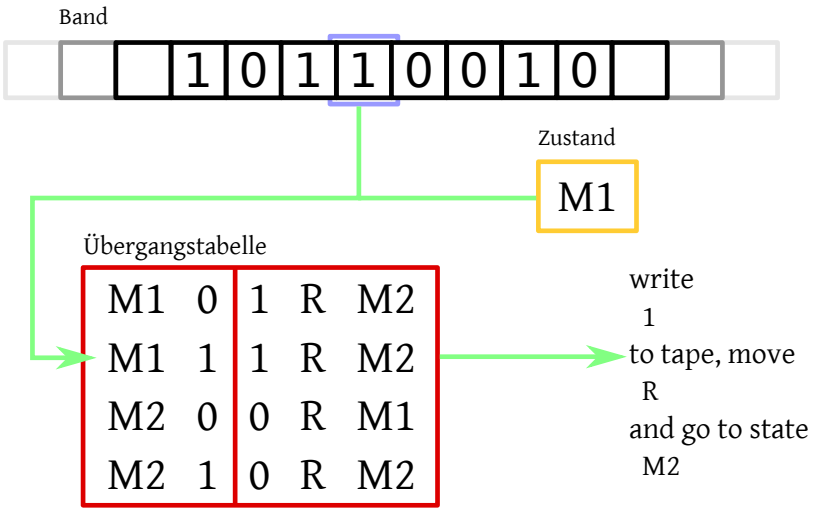


Abbildung 5.1: Die Funktionsweise einer Turingmaschine visualisiert.

In Abbildung 5.1 ist der Aufbau einer Turingmaschine visualisiert. Die Turingmaschine besitzt ein unendlich langes *Band* (schwarz) auf welchem

Zeichen aus dem Alphabet Γ stehen. Der *Cursor* (blau) befindet sich an einer bestimmten Stelle auf diesem Band und liest und schreibt Zeichen. Basierend auf der Information in welchem *Zustand* (orange) sich die Turingmaschine befindet und welches *Zeichen gelesen* wurde, wird eine Zeile in der *Übergangstabelle* (rot) ausgewählt, die dann den Zustandsübergang definiert.

Die Turingmaschine besitzt eine initiale Konfiguration. Als Konfiguration bezeichnet man das Tupel (Zustand, Band, Cursor). Die abgebildeten Turingmaschine befindet sich in einem Zustand M1, auf dem Band befinden sich die Zeichen 10110010 (aus dem Alphabet Σ) und der Cursor ist links der Mitte positioniert. Dies sei unsere initiale Konfiguration. Die weiteren Konfigurationen lassen sich durch schrittweise Ausführung der Instruktionen ableiten.
Aufgrund der Konfiguration können wir jetzt in der Übergangstabelle nachschlagen, welche Instruktion ausgeführt werden soll. Eine Zeile der Spalte besteht dabei aus 5 Werten: Ausgangszustand, gelesenes Zeichen, geschriebenes Zeichen, Bewegung und nächster Zustand. Dies bedeutet in Zeile 2 stimmt die Konfiguration mit den Werten überein: Wir befinden uns im Zustand M1 (ja, in diesem befinden wir uns gerade) und lesen das Zeichen „1“ (ja, wurde vom Cursor gelesen). Dementsprechend ist eine „1“ an diese Position am Band zu schreiben, nach rechts („R“) zu fahren und in den Zustand „M2“ zu wechseln. Wenn wir den nachfolgenden Schritt beobachten, wird die Zeile 3 ausgeführt: Im Zustand „M2“ wurde das Zeichen „0“ gelesen, wird das Zeichen „0“ geschrieben, der Cursor nach rechts („R“) bewegt und in den Zustand „M1“ gewechselt.

Erreicht die Turingmaschine einen der definierten Endzustände, wird die Eingabe als „akzeptiert“ (Ja) interpretiert. Kommt die Turingmaschine in einen undefinierten Zustand, wird die Eingabe „abgelehnt“ (Nein).

Durch das Wechseln von Zuständen und das Speichern von Zeichen auf dem Band können Daten verarbeitet werden. Der Input und Output eines ablaufenden Algorithmus' bildet die Initial- und Endkonfiguration der Turingmaschine. Die Logik bzw. das Programm ist in der Übergangstabelle kodiert. Es ist im Allgemeinen nicht möglich aus der Problembeschreibung den entsprechenden Algorithmus zu generieren, der das Problem entscheidet. Dies begründet das Tätigkeitsfeld eines Programmierers und erfordert seine menschliche Kreativität.

Eine Turingmaschine, die eine Bewegung „Stop“ unterstützt ist gleich mächtig wie eine Turingmaschine, die es nicht unterstützt. Hierfür muss die Anzahl der Zustände (in denen ein Stop gebraucht wird) verdoppelt werden. In den folgenden Programmen verwenden wir die Bewegung „Stop“.

Zustand	Lesen	Schreiben	Bewegung	neuer Zustand
Start	□	□	Stop	Even
Start	0	0	Rechts	Odd
Start	1	1	Rechts	Odd
Even	□	□	Stop	Even
Even	0	0	Rechts	Odd
Even	1	1	Rechts	Odd
Odd	□	□	Stop	Odd
Odd	0	0	Rechts	Even
Odd	1	1	Rechts	Even

Tabelle 5.1: Eine Übergangstabelle für „Gerade oder ungerade Anzahl“.

5.3 Algorithmenbeispiele auf der Turingmaschine

Im Folgenden werden 3 Algorithmen für 3 verschiedene Probleme vorgestellt, die wir auf einer Turingmaschine entscheiden werden. Sie folgen jeweils unterschiedlichen Ideen und sollen dem Leser mögliche Lösungsansätze für andere äquivalente Probleme illustrieren.

5.3.1 Algorithmus: Gerade oder ungerade Anzahl

Gegeben sei eine Turingmaschine. Auf dem Band befindet sich eine beliebige Sequenz von Nullen (0) und Einsen (1). Der Cursor befindet sich auf der linkesten Ziffer. Gehe in den Endzustand „Even“ wenn sich eine gerade Anzahl an Ziffern auf dem Band befinden. Gehe in den Endzustand „Odd“ wenn sich eine ungerade Anzahl an Ziffern auf dem Band befinden. Die Position des Cursors in den Endzuständen ist nicht spezifiziert. Das Band muss im Endzustand gleich wie am Anfang aussehen. Schreibe das Programm der Turingmaschine, um dieses Problem zu entscheiden.

Lösung (Tabelle 5.1). Der Lösungsansatz beruht darauf zwischen den zwei Zuständen „Even“ und „Odd“ zu alternieren, wobei der entsprechende Zustand repräsentiert, ob eine gerade oder ungerade Anzahl von Zeichen bisher gelesen wurde. Zu beachten ist weiters, dass die Sequenz auch die Länge 0 umfasst und daher der Fall abgedeckt werden muss, falls sich kein Zeichen (bzw. das Blanksymbol) unter dem Startzustand befindet.

5.3.2 Algorithmus: 1en zählen

Gegeben sei eine Turingmaschine. Auf dem Band befindet sich eine beliebige Sequenz von Nullen und Einsen (der Mindestlänge 1). Der Cursor befindet sich auf der linkesten Ziffer. Gesucht ist ein Algorithmus, welcher das Band mit Blanksymbolen überschreibt und genau so viele Einsen nebeneinander angeordnet schreiben soll, wie im String anfangs enthalten waren. Als Beispiel soll der Eingabestring „00101101“ zur Ausgabe „1111“ führen. Als weiteres Beispiel soll „10“ zu „1“ führen. Im Endzustand „End“ soll der Cursor auf der rechtesten 1 stehen.

Lösung (Tabelle 5.2). Der Algorithmus folgt diesem Ablauf:

1. Markiere Anfang (ˆ) und Ende ($) mit zwei Hilfssymbolen (diese Markung wird in den Zuständen Start und Mark vorgenommen).

2. Im Zustand *Find* gehe nach rechts und suche die nächste 1. Ersetze sie mit einer 0 und gehe in Zustand *Found* oder falls du keine 1 findest, finalisiere das Band.

3. In den Zuständen *Found*, *Write* und *Return* wird das Symbol auf der rechten Seite des Hilfssymbols $ hinzugefügt und es wird zurück zur linken Seite gegangen.

4. Beim Finalisieren lösche die Hilfssymbole ˆ und $ und die Anfangssequenz von Nullen und Einsen vom Band. Gehe an die rechteste Position des Ergebnisses.

5.3.3 Algorithmus: Minimum von 3 2-bit Zahlen

Gegeben sei eine Turingmaschine. Auf dem Band befinden sich 6 Nullen oder Einsen. Diese sind als 3 2-bit Zahlen zu interpretieren. Schreibe rechts von diesen Zahlen jenen Wert, der das Minimum der 3 Zahlen repräsentiert und lasse das restliche Band im Anfangszustand. Der Startzustand sei „Start“ und der Endzustand sei „End“. Die initiale Cursorposition sei die linkeste Ziffer. Im Endzustand soll der Cursor ganz rechts sein.

Lösung (Tabelle 5.3). Unser Lösungsansatz beruht auf der Idee, dass wir sämtliche relevanten Informationen des Inputs in den Zuständen speichern. Wir modifizieren das Band nicht (d.h. gelesene und geschriebene Zeichen sind stets ident außer wir schreiben das Ergebnis) und die Bewegung geht stets nach rechts (außer wir terminieren). Wir lesen Ziffer für Ziffer ein und wechseln in einen entsprechenden Zustand. So wird etwa in dem Startzustand bei einem gelesenen 1 in den Zustand 001 gewechselt, um uns zu merken

Zustand	Lesen	Schreiben	Bewegung	neuer Zustand
Start	□	$	Links	Mark
Start	0	0	Rechts	Start
Start	1	1	Rechts	Start
Mark	□	^	Rechts	Find
Mark	0	0	Links	Mark
Mark	1	1	Links	Mark
Find	0	0	Rechts	Find
Find	1	0	Rechts	Found
Find	$	$	Links	Finalize
Found	0	0	Rechts	Found
Found	1	1	Rechts	Found
Found	$	$	Rechts	Write
Write	□	1	Links	Return
Write	1	1	Rechts	Write
Return	0	0	Links	Return
Return	1	1	Links	Return
Return	$	$	Links	Return
Return	^	^	Rechts	Find
Finalize	0	0	Links	Finalize
Finalize	1	1	Links	Finalize
Finalize	^	□	Rechts	Delete
Delete	0	□	Rechts	Delete
Delete	1	□	Rechts	Delete
Delete	$	□	Rechts	Result
Result	□	□	Stop	End
Result	1	1	Rechts	Result

Tabelle 5.2: Eine Zustandstabelle für „Einsen zählen“.

dass das kleinste Minimum 00 war (wir haben bisher noch keine Zahl gelesen und daher wird die niedrigste Zahl 00 verwendet) und mit der zusätzlichen 1 merken wir uns das letzte gelesene Symbol. Dies ist bei allen Zuständen, die aus 3 Ziffern bestehen gleichartig. Wir müssen dabei in jenen Zustand wechseln, welcher das Maximum der zwei Zustände repräsentiert, die wir durch den aktuellen Zustand und das gelesene Zeichen kodiert haben. Haben wir eine gerade Anzahl an Zeichen gelesen, befinden wir uns in einem Zustand aus 2 Ziffern und sonst in einem Zustand aus 3 Ziffern. Kommen wir auf ein Blanksymbol, schreiben wir jene Zahl, die wir als Minimum im Namen des Zustands gespeichert haben.

5.4 Theoretische Erkenntnisse

5.4.1 Universelle Turingmaschine

Unter einer *universellen Turingmaschine* versteht man eine Turingmaschine, welche als Parameter[1] eine andere Turingmaschine entgegen nimmt und nachfolgend die einzelnen Schritte der Turingmaschine berechnet. Dabei befinden sich alle Informationen wie der aktuelle Zustand, das Band und das Programm auf dem Band der universellen Turingmaschine. Wird ein Befehl der simulierten Turingmaschine ausgeführt, wird am Band nach der entsprechenden Instruktion nachgeschlagen und der Befehl ausgeführt.

Die genaue Implementierung wird hier nicht weiter präzisiert, da es sich nur um Überlegungen hinsichtlich Kodierungen handelt.

5.4.2 Mehrbandturingmaschinen

Eine Erweiterung der Turingmaschine besteht darin mehr als 1 Band zu verwenden. Dies hilft insbesondere bei der Organisation der Daten bei der Implementierung von Algorithmen. Betrachten wir etwa ein Graphenproblem, hilft es beispielweise auf einem Band den aktuell betrachteten Knoten zu speichern und auf den weiteren Bändern zwei anliegende Kanten. In dem Fall müssen wir die Wanderungen nicht ausformulieren, wenn wir die Bänderdaten von einem anderen Band lesen wollen und diese Bänder nebeneinander angeordnet werden.

Jede Mehrbandturingmaschine kann durch eine Turingmaschine mit 1 Band simuliert werden.

[1] Unter Parameter im Kontext einer Turingmaschine versteht man einen Wert, welcher bereits auf das Band kodiert wurde, bevor die Turingmaschine gestartet wurde.

Zustand	Lesen	Schreiben	Bewegung	neuer Zustand
Start	0	0	Rechts	000
Start	1	1	Rechts	001
000	0	0	Rechts	00
000	1	1	Rechts	01
001	0	0	Rechts	10
001	1	1	Rechts	11
010	0	0	Rechts	01
010	1	1	Rechts	01
011	0	0	Rechts	10
011	1	1	Rechts	11
100	0	0	Rechts	10
100	1	1	Rechts	10
101	0	0	Rechts	10
101	1	1	Rechts	11
110	0	0	Rechts	11
110	1	1	Rechts	11
111	0	0	Rechts	11
111	1	1	Rechts	11
00	□	0	Rechts	0
00	0	0	Rechts	000
00	1	0	Rechts	001
01	□	0	Rechts	0
01	0	0	Rechts	010
01	1	1	Rechts	011
10	□	1	Rechts	0
10	0	0	Rechts	100
10	1	1	Rechts	101
11	□	1	Rechts	1
11	0	0	Rechts	110
11	1	1	Rechts	111
0	□	0	Stop	End
1	□	1	Stop	End

Tabelle 5.3: Eine Zustandstabelle für „Minimum von 3 2-bit Zahlen“.

Die Frage, die sich jedoch stellt, ist: Hat diese Erweiterung Auswirkungen auf die Komplexität der Probleme? Wie es sich herausstellt, handelt es sich nur um einen polynomiallen Mehraufwand.

5.4.3 Nicht-Determinismus

Der Nichtdeterminismus ist ein wichtiges Konzept für die theoretische Informatik. Grundsätzlich handelt es sich bei der Übergangsfunktion der Turingmaschine um eine Abbildung des gelesenen Zeichens und des aktuellen Zustands auf den neuen Zustand, das zu schreibende Zeichen und eine Bewegung. Nichtdeterminismus beschreibt den Zustand eines Systems in dem der Übergang von einem Zustand der Maschine in den nächsten Zustand nicht wohldefiniert und eindeutig ist. In anderen Worten auch: Für eine gegebene Konfiguration der Turingmaschine gibt es mehrere Möglichkeiten in den nächsten Zustand zu kommen.

Im Kontext der Komplexitätstheorie spricht man bei Nichtdeterminismus von den sogenannten *Orakelturingmaschinen*. Liegen mehrere Zustandübergänge für eine Konfiguration vor, befragt die Turingmaschine die (zur Verfügung stehende) Orakelturingmaschine. Diese antwortet stets mit jenem Übergang, welcher direkt zur richtigen Lösung führt. Diese Hervorsagefähigkeit begründet ihren Namen „Orakel“.

5.4.4 Unentscheidbarkeit des Halteproblems

Alan Turing konnte 1936 [Tur37] überraschenderweise beweisen, dass es einfache, nur mit Ja oder Nein beantwortbare Fragestellungen (auch Entscheidungsprobleme genannt) gibt, die sich mit Turingmaschinen *nicht* vollständig beantworten lassen, obwohl eine klar definierte Antwort offensichtlich existieren muss. Dabei ist es für die eigentliche Fragestellung selbstverständlich rein logisch gesehen unwichtig, *wie* die Antwort herausgefunden wird, sie muss nur der Wahrheit entsprechen und liegt sozusagen unmittelbar vor, während eine Turingmaschine die korrekte Antwort erst Schritt-für-Schritt berechnen muss.

Turing bewies nun, dass es keine Turingmaschine H geben *kann*, die in jedem Fall nach endlich vielen Schritten die richtige Antwort auf die Fragestellung liefert, ob eine beliebige andere Turingmaschine M bei anfänglichen Bandinhalt I_M irgendwann anhält oder stattdessen für immer weiterrechnet (das sogenannte *Halteproblem*).

Unter Annahme der Church-Turing These lassen sich solche Fragestellungen daher mittels bekannter Methoden nicht eindeutig entscheiden, und man bezeichnet sie entsprechend als *unentscheidbar* (ohne Annahme der Church-Turing These kann man nur sagen, dass das Halteproblem nicht

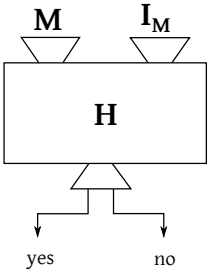
Turing-berechenbar ist, allerdings ist die Church-Turing These allgemein akzeptiert).

Beweis der Unentscheidbarkeit des Halteproblems

Der Beweis der Unentscheidbarkeit des Halteproblems folgt durch einen Widerspruch, welcher nur beseitigt werden kann, wenn folgende Annahme zurückgenommen wird (d.h. die Turingmaschine H kann es nicht geben).

Annahme: Es gibt eine Turingmaschine H, die für eine beliebige Turingmaschine M und einen beliebigen Input I_M in endlich vielen Schritten herausfindet, ob M mit anfänglichen Bandinhalt I_M irgendwann anhält oder nicht.

Konstruktion: Die Turingmaschine H schreibt als Antwort aufs Band *Ja*, falls M mit I_M irgendwann anhält, und *Nein*, falls M mit Input I_M nicht anhält, und bleibt sofort danach stehen (siehe Abbildung 5.2).



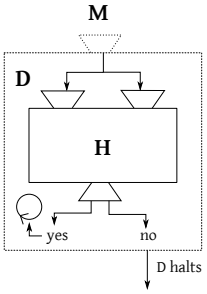


Abbildung 5.2: Turingmaschine H. Abbildung 5.3: Turingmaschine D.

Für den Beweis wird eine weitere Turingmaschine D (siehe Erklärung weiter unten, warum hier der Buchstabe D verwendet wird) definiert, die sich wie folgt verhält (siehe Abbildung 5.3):

- Bei der Eingabe M (in kodierter Form) simuliert D die Turingmaschine H mit Eingabe $I_M = M$.
- H antwortet hier also mit „Ja“ oder „Nein“, je nachdem ob M bei Verwendung des eigenen kodierten Programmcodes (also von M) als Input anhält oder nicht.
- Sobald die Simulation von H fertig ist (was nach endlich vielen Schritten sicher passiert, so die Annahme), so verhält sich D zusätzlich noch wie folgt:

  - Falls H als Resultat „Ja“ ausgegeben hat, geht D in eine Endlosschleife.
  - Falls H als Resultat „Nein“ ausgegeben hat, hält D an.

Folgerung: Nun wird der kodierte Programmcode von D in die Turingmaschine D als Input eingegeben, d.h. D wird mit Programminput D ausgeführt. Es gibt dann nur folgende zwei Fälle:

- Falls D mit der Eingabe der kodierten Form von D anhielte, dann müsste die Simulation von H, sobald diese fertig ist, also knapp *vor* dem Ende der Ausführung von D, ein „Nein“ ausgegeben haben. Dies entspricht aber der Evaluierung des Anhaltens von D mit der Eingabe D, d.h. D mit der Eingabe D dürfte dementsprechend *nicht* angehalten haben. Da D mit der Eingabe D also sowohl halten als auch nicht halten müsste, ergibt sich in diesem Fall ein Widerspruch.

- Falls D mit der Eingabe der kodierten Form von D in eine Endlosschleife ginge, also *nicht* anhielte, dann müsste die Simulation von H vor dem übergang in die Endlosschleife bei der Ausführung von D ein „Ja“ ausgegeben haben. Letzteres entspricht aber der Evaluierung des Anhaltens von D mit der Eingabe D, d.h. D mit der Eingabe D müsste dementsprechend angehalten haben. Da D mit der Eingabe D also sowohl nicht halten als auch halten müsste, ergibt sich auch in diesem Fall ein Widerspruch.

Beide Fälle führen also zu einem Widerspruch. Außer der Annahme, dass H existiert, ist dieser Beweis nachvollziehbar und korrekt. Um diesen Widerspruch aufzulösen, können wir also nur die Annahme verwerfen. Q.E.D.

Es *kann* also keine solche Turingmaschine geben, die immer entscheiden kann, ob eine beliebige andere Turingmaschine irgendwann in ihren Berechnungen anhält. Über die Church-Turing-These gilt dies auch für jegliche Algorithmen (und damit für alle Arten rationaler Gedankengänge).

Alternativer Beweis

Gegeben sei die selbe Annahme und Konstruktion.

Der Buchstabe D steht für Diagonalisierungsargument. Und zwar betrachtet man eine Tabelle, deren Zeilen und Spalten mit allen überhaupt möglichen Kodierungen von Turingmaschinen und allen ihren möglichen Inputs für H, jeweils aufsteigend geordnetet, beschriftet sind. Dies lässt sich z.B. mittels einer binären Kodierung realisieren. Im Feld der Tabelle in der Zeile, deren

Beschriftung M kodiert, und der Spalte, deren Beschriftung I_M kodiert, steht 0 falls die Turingmaschine M mit Input I_M anhält, und 1 falls sie nicht anhält. Dann entsprechen die Werte in der *Diagonale* in der Tabelle genau dem Verhalten von D, wenn der Wert 1 bedeutet, dass D *anhält* (also genau *umgekehrt* wie für M!), und 0 dass sie nicht anhält. Dann entsprechen die Werte in Zeile (und Spalte) mit der Kodierung von M in der *Diagonale* in der Tabelle genau dem Verhalten von D mit Input M, wenn der Wert 1 bedeutet, dass D mit Input M *anhält* (also genau *umgekehrt* wie für M!), und 0 dass sie nicht anhält.

Der Widerspruch ergibt sich bei dieser Sichtweise daraus, dass ja D ebenfalls eine Turingmaschine ist, die auch selbst mit der gleichen Methode wie M und I_M kodiert werden kann. Ihre Ausführung auf sich selbst als Input entspricht also ebenfalls einem Feld in der Diagonale, das dann allerdings gleichzeitig sowohl den Wert 0 als auch den Wert 1 enthalten müsste, was natürlich nicht sein kann — dieses Feld in der Diagonale und damit auch die Kodierung von D sowie in weiterer Folge H können daher nicht existieren. Gemäß diesem Beweis ist das Halteproblem also ebenfalls unentscheidbar. Q.E.D.

Das 3n + 1 Problem: Dieses Problem, auch bekannt unter der Bezeichnung Collatz-Problem, zählt zu den bekanntesten ungelösten mathematischen Problemen. Sei n eine natürliche Zahl. Falls n gerade ist, dann fahre mit $n/2$, sonst mit $3n+1$ fort, bis durch wiederholte Anwendung dieser beiden Formeln die Zahl 1 erreicht wird. Die Frage ist, ob der Algorithmus bei *jedem* ganzahligen Anfangswert größer als 0 die Zahl 1 erreicht und damit anhält. Trotz größter Anstrengungen ist diese Frage bisher unbeantwortet, und für mehrere Verallgemeinerungen konnte bewiesen werden, dass sie unentscheidbar sind.

5.4.5 Unentscheidbarkeit des Korrektheitsproblems

Es gibt unzählige andere unentscheidbare Probleme[2], zum Beispiel das für die Informatik bedeutsame sogenannte *Korrektheitsproblem*, bei dem es darum geht herauszufinden ob ein Algorithmus einer Spezifikation entspricht oder nicht. Klarerweise wäre es sehr wünschenswert, automatisch die Einhaltung einer Spezifikation überprüfen zu können. Leider lässt sich die Unentscheidbarkeit des Korrektheitsproblems leicht mittels eines sogenannten Reduktionsbeweises zeigen, wodurch sich dieser Wunsch als prinzipiell unerfüllbar erweist.

[2]Laut dem Satz von Rice ist *jede* nicht-triviale Fragestellung die Turingmaschinen betrifft unentscheidbar.

Beweis der Unentscheidbarkeit des Korrektheitsproblems

In dem Beweis wird das Halteproblem mittels einer sogenannten Reduktion[3] auf eine bestimmte Form des Korrektheitsproblems reduziert. Wäre nun das allgemeine Korrektheitsproblem entscheidbar, würde sich aus der Reduktion ein funktionierender Algorithmus für das Halteproblem ergeben, was im Widerspruch zur bereits gezeigten Unentscheidbarkeit des Halteproblems stünde. Deshalb *muss*, damit dieser Widerspruch umgangen werden kann, auch das Korrektheitsproblem unentscheidbar sein.

Im Falle des Korrektheitsproblems müssen wir also bloß eine Reduktion angeben, die zu jeder beliebigen Instanz des Halteproblems eine entsprechende Instanz des Korrektheitsproblems mit gleicher Ja/Nein Antwort konstruiert. Sei M eine beliebige Turingmaschine mit einem beliebigen Input I_M. Die Reduktion besteht nun darin, eine neue Turingmaschine M' zu konstruieren, indem wir Übergangsregeln zu M hinzuzufügen, sodass nach dem Halten des ursprünglichen Programms in mehreren zusätzlichen Schritten das gesamte Band mit Nullen überschrieben wird, eine einzelne 1 darauf geschrieben wird und M' unmittelbar danach stehen bleibt. Dies ist möglich, da wir ohne Beschränkung der Allgemeinheit erzwingen können, dass eine Turingmaschine, hier M, immer nur in *einem* speziellen Endzustand anhalten kann.

Die Instanz des Korrektheitsproblem wird nun formuliert als die Frage, ob das Programm M' mit der gegebenen Spezifikation „Bei Input I_M liefere den Wert 1 zurück“ übereinstimmt. Klarerweise wird das *genau dann und nur dann* passieren, wenn M mit Input I_M anhält.

Unter der Annahme dass es nun eine Turingmaschine K gäbe, die diese einfache Form des Korrektheitsproblems entscheiden könnte, würde durch obige Reduktion auch ein Algorithmus für das Halteproblem vorliegen. Er würde zuerst M' konstruieren und dann K auf M' mit Input I_M anwenden und das Ja/Nein Ergebnis der Berechnung von K als Ergebnis von H zurückgeben (siehe Abbildung 5.4).

Dies steht allerdings im Widerspruch zur Unentscheidbarkeit des Halteproblems, demgemäß es ja keinen solchen Algorithmus geben kann. Daher müssen wir die oben genannte Annahme, dass eine solche Turingmaschine K existieren kann, zurücknehmen, und damit ist natürlich auch das allgemeine Korrektheitsproblem unentscheidbar. Q.E.D.

Für die Softwareentwicklung bedeutet dies, dass im Allgemeinen leider *weder* eine automatisierte Überprüfung (auch *formale Verifikation* genannt) einer Übereinstimmung zwischen einem Programm (bzw. seines Programm-

[3]Eine einfache Transformation, die problemlos ausprogrammiert werden kann. Der Begriff ergibt sich daraus, dass sich das Halteproblem sozusagen als eine vereinfachte, also eingeschränkte, sprich reduzierte Version des Korrektheitsproblems herausstellt.

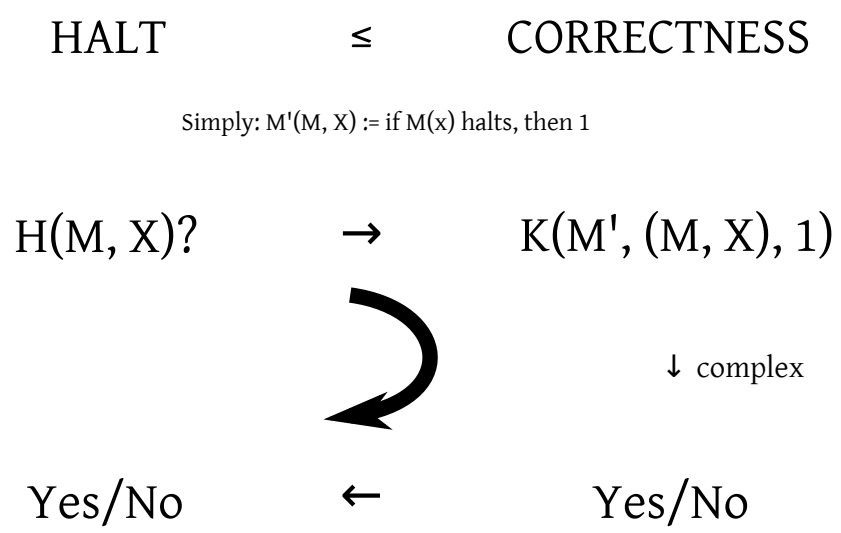


Abbildung 5.4: Reduktion des Halteproblems auf das Korrektheitsproblem.

codes) und seiner Spezifikation *noch* eine händische durch einen rational vorgehenden Menschen möglich ist.

Kapitel 6

Graphentheorie

Ein Graph ist eine Menge (V, E) von Knoten (engl. vertex) und Kanten (engl. edge). Graphen unterscheiden sich aufgrund ihrer Eigenschaften in Form und Anzahl an Knoten und Kanten. Da Graphen so vielfältig eingesetzt werden, haben sich viele Begriffe etabliert, die teilweise mehrdeutig verwendet werden. Es sei hier eine möglichst genaue Definition der Begriffe im Kontext der Informatik gegeben.

6.1 Terminologie

(un)gerichteter Graph Ein gerichteter Graph unterscheidet zwischen einer Kante (u, v) und der Kante (v, u), während diese Unterscheidung beim ungerichteten Graphen wegfällt. Bei einer Visualisierung werden Pfeile genutzt, um die Richtung einer Kante anzugeben.

Pfad / Weg Unter einem Pfad (oder Weg) versteht man eine Sequenz von Kanten, die einer Wanderung durch den Graphen entspricht (Endknoten sind Startknoten der nächsten Kante).

Zyklus Ein Pfad dessen Startknoten gleich dem Endknoten ist.

Schleife Eine Kante, die den Knoten mit sich selbst verbindet. Sie ist auch damit ein Zyklus der Länge 1.

Multigraph Graph, welcher mehrere Kanten zwischen zwei Knoten und Schleifen erlaubt.

einfacher/simpler Graph Ungerichteter Graph, welcher kein Multigraph ist.

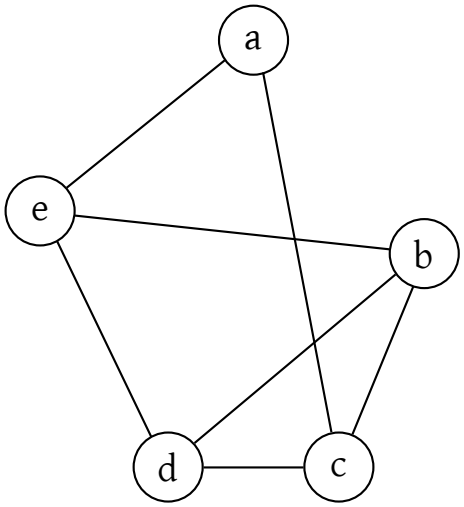


Abbildung 6.1: Visualisierung eines ungerichteten Beispielgraphs $V = \{a, b, c, d, e\}, E = \{(a, c), (a, e), (b, c), (b, d), (b, e), (c, d), (d, e)\}$. Der Graph besitzt 7 von maximal 10 ($= \frac{|V| \cdot (|V|-1)}{2}$) Kanten. Es handelt sich um einen einfachen Graphen, der 4 Zyklen besitzt und planar dargestellt werden kann.

Baum Ein Baum wird als zusammenhängender, zyklusfreier Graph verstanden. Er ermöglicht es Daten einer hierarchischen Struktur abzubilden.

Brücke Unter einer Brücke versteht man eine Kante deren Entfernung zum Verlust der Eigenschaft „zusammenhängend“ des Graphens führt.

Vollständiger Graph Unter einem vollständigen Graph versteht man einen Graph, in dem jeder Knoten mit jedem Knoten direkt verbunden ist.

Untergraph Ein Graph G', welcher eine Untermenge der Knoten von G verwendet und nur jene Kanten verwendet, deren zugehörige Knoten in G' enthalten sind.

Clique Ein vollständiger Untergraph.

Eulerkreis Der Eulerkreis ist ein spezieller Zyklus, der alle Knoten des Graphen nutzt und jede Kante genau einmal verwendet.

Grad eines Knoten Der Grad eines Knotens ist definiert als die Anzahl an ein- und ausgehenden Kanten.

Hamiltonpfad Unter einem Hamiltonpfad versteht man einen Pfad im Graphen, der jeden Knoten genau einmal besucht.

Planarität Ein Graph ist planar, wenn er ohne Überschneidungen von Kanten auf einer 2D-Fläche gezeichnet werden kann.

Quelle/Senke Unter der Quelle versteht man bei einem gerichteten Graphen einen Knoten, der nur nach außen führende Kanten besitzt. Eine Senke ist ein Knoten, der nur eingehende Kanten auf diesen Knoten besitzt.

6.2 Datenstrukturen zur Speicherung von Graphen

Für die Speicherung eines Graphen ist eine Datenstruktur erforderlich, die Knoten und Kanten abbildet. Die Knoten werden typischerweise durchnummeriert; deren Namen separat assoziativ gespeichert. Im Fokus stehen zwei verschiedene Konzepte um Kanten zu speichern:

Inzidenzliste Die Kanten werden als Paar von Knoten in einer Liste gespeichert.

Adjazenzmatrix In einer Matrix der Größe $|V| \times |V|$ wird ein Eintrag $m_{i,j}$ mit einer 1 (oder einem Gewicht) besetzt, wenn eine Kante zwischen den Knoten i und j besteht.

Kapitel 7

Abstraktion

Die Informatik kombiniert all die genannten Grundlagen um höhere Konzepte zu bauen. Dieser Ansatz nennt sich Abstraktion. Durch das Auslassen kleiner Details auf der unteren Abstraktionsebene, können wir ein vereinfachtes Bild in der aktuellen Abstraktion annehmen. Ein Beispiel ist etwa das Schienennetz Österreichs. Möchte ein Informatiker alle freigegebenen und besetzen Schienenblöcke visualisieren, so interessiert ihn das spezifische Material des Gleises auf einem gewissen Streckenabschnitt wenig. Ein wesentlich besserer Ansatz (um die Schienenblöcke auf dem Schienennetz zu erfassen) ist es das Netz als einen Graphen darzustellen, um die Details wegzulassen, die für ein bestimmtes Problem irrelevant sind. Die Visualisierung des Graphen ist eine abstrakte Sichtweise auf das Netz.

7.1 In der Programmierung

Auch in der Programmierung benötigen wir durchgehend Methoden zur Abstraktion. Dabei werden einzelne Instruktionssequenzen zu einem Block zusammengefasst, welcher eine gewisse semantische Funktionalität repräsentiert. So kann etwa eine kleine Unterfunktion (siehe 1) eine Konvertierung der ASCII-Kleinbuchstaben in Großbuchstaben vornehmen. Und diese Funktionen können mehrfach in verschiedenen Kontexten wiederverwendet werden, um größere Konzepte zu bauen.

7.2 In Bezug auf die theoretische Informatik

Das selbe trifft auf Turingmaschinen zu. So sprechen wir bei universellen Turingmaschinen davon, dass eine Turingmaschine eine andere Turingmaschine simuliert. Wir haben dabei betont, dass wir auf die konkrete Kodierung der

simulierten Turingmaschine nicht eingehen. Dem liegt zugrunde, dass diese Kodierungsform gewisse Details festlegen würde, die für das Verständnis der universellen Turingmaschine unerheblich sind; äquivalent zur Abstraktion, die Details versteckt.

Abstraktion ist ein fundamentales Konzept der Informatik. Ironisch meint Kevlin Henney zu einem Zitat von David Wheeler,

> „All problems in computer science can be solved by another level of indirection except for the problem of too many layers of indirection“
> —Kevlin Henney

> „Alle Probleme der Informatik können durch eine weitere Ebene von Umwegen gelöst werden; außer das Problem von zu vielen Umwegen.“
> —frei ins Deutsche übersetzt

Algorithm 1 Klein- in Großbuchstabenkonvertierung für ASCII in C.

```
int uppercase(char *string)
{
    int index;
    for (index=0; index<strlen(string); index++) {
        if (string[index] >= 97 && string[index] <= 122)
            string[index] = string[index] & (~32);
    }
    return 0;
}
```

Kapitel 8

Landau Notation

Die Informatik ist daran interessiert, Algorithmen (so wie sie im Alltag zur Anwendung kommen) zu optimieren, um Rechenzeit, Speicherplatz und Energie zu sparen. Effizientere Algorithmen erlauben ein wirtschaftlicheres Handeln. Wir benötigen daher ein Werkzeug, um Algorithmen bezüglich ihrer Eigenschaften wie Speicherverbrauch und Laufzeit vergleichen zu können. Obwohl wir nachfolgend von der Laufzeit sprechen, kann die dahinterstehende Theorie genauso auf andere Ressourcen angewandt werden.

Wir bedienen uns einer speziellen mathematischen Notation, welcher die Betrachtung des asymptotischen Verhaltens einer Funktion zugrunde liegt. Diese nennt sich „Landau-Notation“, „$\mathcal{O}$-Notation“[1] oder engl. “Big O-Notation“. Der Ressourcenverbrauch eines Algorithmus kann durch eine mathematische Funktion beschrieben werden und die $\mathcal{O}$-Notation gibt uns ein relatives Maß, um Algorithmen vergleichen zu können.

Bei den nachfolgenden Beispielen sprechen wir von einer Variable n. Es ist jene Variable, die signifikanten Einfluß auf die Effizienz des Algorithmus hat. Bei einem Sortieralgorithmus wird man mit n die Anzahl der Vergleichsoperationen zwischen Elementen beschreiben. Bei der Multiplikation zweier Zahlen wird man die maximale Anzahl der Stellen der beiden Zahlen mit n bezeichnen. Es ist wichtig zu wissen, wofür dieses n konkret steht, doch ergibt es sich aus dem Kontext oft implizit. Im allgemeinen Fall handelt es sich um die Größe der Eingabe.

[1] In LaTeX `\mathcal{O}` und in Unicode mit U+039F „GREEK CAPITAL LETTER OMICRON“ notiert.

8.1 2 grundlegende Klassen

8.1.1 Konstante Laufzeiten

Der Algorithmus 2 (siehe Seite 52) nimmt einen Parameter `n` an und gibt einen leeren Wert zurück. Wir suchen jetzt eine mathematische Funktion, welche uns repräsentiert wieviele Schritte der Algorithmus für eine Eingabe ausführen wird. Für Algorithmus 2 gilt: Unabhängig von der Größe von n wird 1 Instruktion ausgeführt[2]. Wir setzen daher die mathematische Funktion

$$f_1(x) = 1$$

an. Relevant ist hier, dass es sich bei der Zahl 1 um eine Konstante handelt. Alle Algorithmen deren Effizienz mit einer mathematischen Funktion beschrieben werden können, die nur Konstanten verwendet, fallen in die Klasse „konstant“. Also etwa $f_2(x) = 0$ oder $f_3(x) = 365$ als auch f_1 haben konstante Laufzeiten. Ganz allgemein: Ein Algorithmus der Ressourcen unabhängig von der Eingabegröße verwendet, besitzt eine konstante Schranke bezüglich dieser Ressource.

Wir haben die Formulierung „in eine Klasse fallen“ genutzt. Es bedeutet, dass eine Klasse (wie $\mathcal{O}(1)$) eine Menge von Funktionen ist und unsere gegebene Funktion f_1 ist Teil dieser Menge. Alle konstanten Algorithmen sind Elemente von $\mathcal{O}(1)$. Wir schreiben „ist Teil von“ mit dem Gleichheitszeichen an[3]:

$$f_1 = \mathcal{O}(1) \qquad f_2 = \mathcal{O}(1) \qquad f_3 = \mathcal{O}(1)$$

8.1.2 Lineare Laufzeiten

Der Algorithmus 3 benötigt eine lineare Laufzeit. Der Ablauf verrät, dass er über alle Elemente der übergebenen Liste iteriert und damit genau n Schritte benötigt, wenn n die Anzahl der Listenelemente beschreibt. Wird ein n hinzugefügt, benötigt der Algorithmus 1 Instruktion mehr. Es besteht eine *direkte Proportionalität* zwischen n und der Anzahl von Schritten.

8.2 Wozu $\mathcal{O}$ und wie es definiert ist

Angenommen wir hätten zwei Algorithmen. Der erste Algorithmus löst ein gegebenes Problem in konstanter Zeit. Der zweite Algorithmus behandelt

[2]Die Rückgabe eines Wertes ist stets Teil einer Funktionsverarbeitung und zählen wir daher nicht dazu.

[3]Das Zeichen $\in$ wird auch genutzt, aber kommt in Fachliteratur selten vor. Wir möchten daher in diesem Dokument der Gleichheitszeichen-Konvention folgen.

das selbe Problem in linearer Zeit. In diesem Fall werden wir den ersten Algorithmus bevorzugen, da dieser Algorithmus bei größeren Eingabedaten trotzdem noch schnell zu einem Ergebnis kommt. Der erste Algorithmus ist „effizienter“ (vergleiche mit Tabelle 8.1).

Wir führen nun die $\mathcal{O}$-Notation ein und untersuchen, ob eine der beiden Funktionen „größer“ ist als die andere. „Größer“ bedeutet mathematisch wir finden eine Schranke $g(n)$ für eine Funktion $f(n)$, die stets größere Werte liefert als $f(n)$. In diesem Fall sprechen wir von einer „oberen Schranke“ und uns interessiert das asymptotische Verhalten.

Wir überlegen uns intuitiv folgende Definition: Eine Funktion $g(n)$ ist die obere Schranke von $f(n)$, wenn...

$$\forall n \in \mathbb{N} : f(n) \leq g(n)$$

Möchten wir $f_4(n) = 1$ und $f_5(n) = n$ vergleichen, können wir f_4 als $f(n)$ und f_5 als $g(n)$ einsetzen. Wir stellen fest, dass für ein beliebiges, positives n das Ergebnis der Funktion $g(n)$ ein größeres Ergebnis liefert. f_5 ist eine obere Schranke von f_4.

Wir vergleichen noch zwei andere Funktionen:

$$f_6(n) = 42 \qquad f_7(n) = \pi \cdot n$$

Wir erhalten für $n = 2$ bei $f_6(2) = 42$ und bei $f_7(2) \approx 6,283$. Es ist irgendwie unbefriedigend, dass f_7 *keine* obere Schranke von f_6 ist, da die Werte von f_6 größer als jene von f_7 für niedrige n sind. Es ist doch erkennbar, dass ab einem gewissen n die Funktion f_7 wesentlich stärker wächst. Daher erweitern wir unsere Definition um einen Mindestwert, ab dem diese Kleiner-Gleich-Relation erst gelten muss.

Wir betrachten eine Funktion (die zB eine Laufzeit beschreibt). Es sei folgende Definition gegeben:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists n_0 \in \mathbb{N},\ \forall n \geq n_0 : f(n) \leq g(n)\}$$

Diese Definition erfüllt, was wir erreichen wollen. Wir definieren uns ein sehr großes n für welchen wir den Vergleich der Werte beider Algorithmen (bzw. deren Funktionen) ansetzen und beobachten welche Funktion schneller wächst.

Wir treffen jedoch noch eine Entscheidung: Faktoren sind nicht sehr attraktiv. Die Information „die Funktion wächst linear“ ist wichtig, aber ob sie mit dem Faktor 2 oder 3 wächst, definieren wir als irrelevant. Auch wenn $f_8(n) = 2n$ langsamer wächst als $f_9(n) = 3n$ so sind sie doch sehr ähnlich. Die Entscheidung lässt sich auch mit der Praxis begründen: Moore's Law

Algorithm 2 Subroutine with constant runtime.

```
def function(n):
    print("Hello World")
    return
```

Algorithm 3 Subroutine with linear runtime.

```
def function(lst):
    for i in lst:
        print(i)
    return
```

n	Schritte für f_1	Schritte für f_2
1	1	1
2	1	2
3	1	3
100	1	100
10^6	1	10^6

Tabelle 8.1: Zwei Algorithmen, deren Laufzeit mit $f_1(n) = 1$ und $f_2(n) = n$ beschrieben werden können, im Vergleich.

zeigt uns, dass die Geschwindigkeit von Rechnern exponentiell zunimmt und damit Unterschiede durch Faktoren locker wettmacht. Wir möchten definieren, dass Funktionen mit konstanten Faktoren in der selben Klasse liegen. Wir erweitern unsere Definition ein letztes Mal und führen dazu einen Faktor c ein:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

Diese Definition ist unser formaler Unterbau zur Betrachtung der Landau-Notation.

8.3 Kombination von Algorithmen

Algorithm 4 Combination of algorithms.

```
def function(lst):
    for i in lst:
        print(i)
    print("Hello World")
    return
```

Algorithm 5 Algorithm with nested algorithms.

```
def function(lst):
    print("Hello World")
    return

def function2(lst):
    for i in lst:
        function(lst)
        print(i)
    return
```

Gegeben sei Algorithmus 4, welcher in zwei Teilbereiche geteilt werden kann. Einerseits haben wir eine Iteration, welche auf eine lineare Laufzeitfunktion in $\mathcal{O}(n)$ hindeutet. Nachfolgend haben wir einen Algorithmus, welcher aus einer Instruktion besteht (konstante Laufzeit in $\mathcal{O}(1)$). Wie verhält sich

jetzt die gesamte Laufzeit des Algorithmus? Wir verwenden die Rechenregeln der $\mathcal{O}$-Notation:

$$f_1(n) = \mathcal{O}(g_1(n)) \qquad f_2(n) = \mathcal{O}(g_2(n))$$

$$\Rightarrow f_1(n) + f_2(n) = \mathcal{O}(\max(g_1(n), g_2(n)))$$

$$\Rightarrow f_1(n) \cdot f_2(n) = \mathcal{O}(g_1(n) \cdot g_2(n))$$

Die erste Regel besagt, dass bei sequentieller Betrachtung der Algorithmen die am stärksten wachsende Funktion die asymptotische Schranke des gesamten Algorithmus angibt. Die zweite Regel beschreibt eine verschachtelte Ausführung wie in Algorithmus 5. Wenn ein Algorithmus einen anderen Algorithmus verwendet, entspricht dies der Multiplikation der Funktionen.

Für Algorithmus 4 berechnen wir:

$$n + 1 = \mathcal{O}(\max(n, 1))$$

$$n + 1 = \mathcal{O}(n)$$

Für Algorithmus 5 gilt:

$$n \cdot 1 = \mathcal{O}(n \cdot 1)$$

$$n \cdot 1 = \mathcal{O}(n)$$

Wir merken uns: Wir müssen immer alle Elemente des Algorithmus betrachten, um die Gesamtlaufzeit angeben zu können.

8.4 Verallgemeinerung der $\mathcal{O}$-Notation

Wobei wir hier in unserem Fall von Laufzeiten sprechen, behandelt die Landau-Notation allgemein asymptotisches Verhalten von Funktionen. Wir können für beliebige Ressourcenverwendung eine mathematische Funktion zur Beschreibung heranziehen und durch die $\mathcal{O}$-Notation entsprechende Klassen bilden, um Ressourcenverwendungen zu vergleichen. Liegen zwei Funktionen in der selben Klasse und ein genauer Vergleich ist notwendig, ist eine genaue mathematische Analyse der ignorierten Terme notwendig. Wir haben jedoch vorhin eine Begründung geliefert, dass diese Terme im Allgemeinen nicht relevant sind.

Abseits von der oberen Schranke $\mathcal{O}$ gibt es auch untere Schranken Ω und exakte Schranken Θ. Um den Umfang dieses Dokuments zu beschränken wird hierauf nicht mehr eingegangen und diese Elemente werden in Lehrveranstaltungen wie „Datenstrukturen und Algorithmen“ (708.031) und „Entwurf und Analyse von Algorithmen“ (716.033) behandelt.

Weiters möchten wir hier noch folgenden Aspekt diskutieren: Im Algorithmus 2 wird von einer Funktion `print` gesprochen. Doch welche Laufzeit besitzt diese Funktion eigentlich? Über mehrere Abstraktionsebenen hinweg wird es sich letztendlich um eine Sequenz von Maschinenbefehlen handeln. Doch dazu müssten wir die gesamte Maschine analysieren. Wir einigen uns aber in der Theoretischen Informatik darauf, dass die Architektur der Maschine vernachlässigt wird indem stets die Turingmaschine als Referenzmodell herangezogen wird. Die Turingmaschine bietet uns ein universelles Konzept, um Berechenbarkeit erfassbar zu machen. Ein Schritt auf der Turingmaschine ist unsere elementare Einheit. Und unter anderem deshalb ist die Turingmaschine heute genauso wichtig wie zu Zeiten als Rechenmaschinen noch nicht gebaut werden konnten.

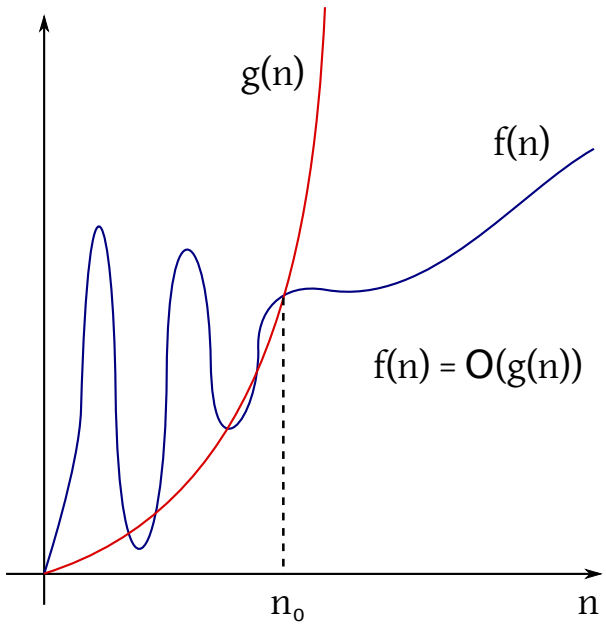


Abbildung 8.1: $g(n)$ bildet eine obere Schranke von $f(n)$.

Abbildung 8.1 Gegeben sei eine beliebige Funktion $f(n)$ mit der oberen Schranke $g(n)$. Wir können ein n_0 finden, wobei ab diesem n_0 jeder Wert von $g(n)$ größer ist als der Wert von $f(n)$.

8.5 Weitere Klassen von Funktionen

8.5.1 Logarithmische Funktionen

Algorithmen mit konstanter Laufzeit werden durch Unabhängigkeit von der Eingabemenge begründet. Die Iteration in proportionaler Abhängigkeit der Eingabemenge definiert die Klasse der Algorithmen mit linearer Laufzeit. Eine weitere Klasse wird durch logarithmisches Verhalten begründet, welche zwischen der konstanten und linearen Klasse einzuordnen ist.

Für Algorithmen mit logarithmischer Laufzeit gibt es ein bekanntes Beispiel. Die Binärsuche ist ein Algorithmus, welcher ein Element in einer geordneten Liste findet. Dazu betrachtet der Algorithmus das mittlere Element der Liste und vergleicht es mit dem gesuchten Wert. Ist es größer, wird die Liste auf die linke Hälfte der Liste reduziert. Ist es kleiner, wird die Liste auf die rechte Hälfte reduziert. Stimmt es überein, haben wir das gesuchte Element gefunden. Dieser Algorithmus setzt sich so lange fort bis das Element gefunden wurde. Die ständige Halbierung der Eingabegröße n beschreibt eine logarithmische Funktion. Daher ist die Binärsuche ein Vertreter der Algorithmen mit logarithmischer Laufzeit.

8.5.2 Polynomielle Funktionen der Form n^k

Funktionen der Form $f(n) = n^k$ mit $k \in \mathbb{R}^+$ treten sehr häufig auf. Ein typischer Algorithmus mit solch einer Struktur ist eine verschachtelte Schleife, wobei die Anzahl der Iterationen proportional zu n sein muss. Die Anzahl der Verschachtelungen entspricht k.

Liegt in einer Schleife über alle Inputwerte eine Schleife über alle Inputwerte vor, wird pro Wert die gesamte Liste iteriert. Dies entspricht einem Algorithmus dessen Laufzeit mit der Funktion n^2 beschrieben werden kann.

Eine offene Frage ist, ob n^2 eine andere Klasse als n^3 begründet. Da der formale Beweis für den Umfang des Dokuments unangebracht wäre, wird auf diesen verzichtet[4] und notiert, dass n^k eine andere Klasse als n^l begründet für beliebige k und l mit $k \neq l$.

$$n^3 \notin \mathcal{O}\left(n^2\right)$$

8.5.3 Exponentielle Funktionen

Exponentialfunktionen besitzen eine Struktur k^n mit $k \in \mathbb{R}^+$. Algorithmen, die Untermengen einer Menge mit n Elementen verarbeiten, liegen oft in

[4] Für formale Beweise ist stets die Ungleichheit der Funktionen in der formalen Definition der $\mathcal{O}$-Notation zu zeigen.

dieser Klasse, da die Anzahl der Untermengen für eine gegebene Menge 2^n ist. Ein anderes Beispiel sind die möglichen Variablenbelegungen für eine boolsche Formel. Sei eine boolsche Formel mit n Variablen gegeben, müssen 2^n Variablenbelegungen getestet werden um festzustellen, ob Variablenbelegungen existieren, die diese Formel wahr machen.

Wieder ist es interessant, ob alle Exponentialfunktionen unterschiedlicher Basis in eine Klasse von Funktionen gehören. Auch an dieser Stelle sparen wir uns den formalen Beweis, aber notieren auch hier, dass jede Basis eine eigene Klasse bildet.

$$3^n \notin \mathcal{O}(2^n)$$

8.6 Die Suche nach der zugehörigen oberen Schranke

Wir sprechen von „möglichst großem n“ und von „Wachstum“ sowie „oberen Schranken“. Bei der Angabe von Beispielen zu diesem Thema findet man jedoch die Frage nach der „langsamst wachsenden Funktion“. Wie passen diese Dinge zusammen?

Wir haben eine Klasse als Menge von Funktionen definiert. Die Effizienz eines Algorithmus kann durch eine mathematische Funktion beschrieben werden. Daher wissen wir in welcher Klasse ein Algorithmus liegt. Weiters bilden diese Mengen Untermengen von anderen Klassen. Wir sprechen hier von „oberen Schranken“, die nach unten hin offen sind. Ein Algorithmus mit linearem Laufzeitverhalten ist gleich effizient wie ein anderer linearer Algorithmus und ineffizienter als konstante oder logarithmische Algorithmen. Es bildet sich also eine Hierarchie:

$$A \to B \qquad \text{„A ist effizienter als B“}$$

$$\text{konstant} \to \text{logarithmisch} \to \text{linear} \to$$
$$\text{polynomiell mit } n^k \to \text{exponentiell mit } k^n$$

$$k \in \mathbb{R}^+, n \in \mathbb{N}$$

Es können noch weitere Klassen definiert werden, um Algorithmen noch besser zu klassifizieren. Allerdings handelt es sich bei diesen Klassen um einen guten Überblick. Wir notieren weiters, dass alle Klassen kleiner oder gleich „polynomiell mit n^k“ durch einen Polynom[5] beschrieben werden können und daher mit „Algorithmen polynomieller Laufzeit“ bezeichnet werden. Es ist

[5]Mathematische Struktur der Form $y_0x^0 + y_1x^1 + y_2x^2 + \ldots + y_nx^n$.

gängige Konvention polynomielle Algorithmen als „effizient“ zu bezeichnen und andere (insbesondere exponentielle Algorithmen) als „ineffizient“.

Wir könnten nun alle Funktionen mit $\mathcal{O}(n^n)$ abschätzen und stellen fest, dass diese Klasse alle praxisrelevanten Algorithmen umfasst. Allerdings würde es uns nicht weiterhelfen die Effizienz eines Algorithmus zu erfassen. Wir suchen daher für einen Algorithmus jene mathematische Funktion, die den Algorithmus möglichst genau beschreibt. Wir wählen keine zu große Schranke, um gute Klassifikation zu erreichen. Wir wählen keine zu kleine Funktion, da dies keine obere Schranke der Funktion wäre. Mit „langsamst wachsende Funktion“ ist also die kleinstmögliche (aber formal korrekte) „obere Schranke“ gemeint.

8.7 Zusammenfassung

In den vorigen Kapiteln haben wir uns einige Formalismen mit Hinweis auf andere Lehrveranstaltungen erspart. Zugleich haben wir kein intuitives Verständnis der $\mathcal{O}$-Notation entwickelt. Zweiteres wollen wir in diesem Abschnitt nachholen.

Gegeben sei ein Algorithmus. Wir untersuchen wie stark die Laufzeit des Algorithmus von der Eingabe abhängt. Auf Basis dieser Information bilden wir eine mathematische Funktion, die dieses Verhalten beschreibt. Wir reduzieren dann die Funktion auf eine Repräsentation, welche für die jeweilige Klasse repräsentativ ist. Dies können wir mit folgenden Regeln durchführen (beachte dass dieses Modell nicht vollständig ist, aber für die Betrachtung einfacher Polynome ausreicht):

1. Sind zwei Funktionen durch eine Addition verbunden (Exponenten dürfen nicht als eigenständige Funktion betrachtet werden), reduzieren wir die Funktion auf jenen Summanden, der die größte obere Schranke besitzt.

2. Eine Funktion der Struktur an^b (mit a und b als Konstanten) reduzieren wir auf n^b.

3. Eine Funktion der Struktur ab^{cn+d} (mit a, b, c und d als Konstanten) reduzieren wir auf e^n mit $e = b^c$. Schließlich ist $ab^{cn+d} = ab^d b^{cn}$ wobei ab^d konstant ist. Weiters gilt die Rechenregel $n^{km} = \left(n^k\right)^m$ womit sich e^n ergibt.

Wir möchten ein paar Beispiele durchgehen. Die Funktion $f(n) = 3n^2 + n$ können wir zuerst mit Regel (1) bearbeiten. Wir sehen, dass die Funktion aus einer quadratischen Teilfunktion $3n^2$ und einer linearen Funktion n besteht.

Wir wissen aufgrund der Klassifikationen, dass die quadratische Funktion schnell wächst. Daher möchten wir die Funktion auf $f(n) = 3n^2$ reduzieren. Nun wird unsere Struktur durch Regel (2) beschrieben und wir ersetzen sie dementsprechend mit n^2. Das heißt unsere finale Klasse in der f liegt, nennt sich $\mathcal{O}(n^2)$.

Für das Beispiel $f(n) = 5^n + n^2 - 5$ gilt: $5^n + n^2 - 5 \overset{(1)}{\Rightarrow} 5^n \Rightarrow \mathcal{O}(5^n)$. In Tabelle 8.2 wird nochmals ausgeführt welche Relationen zwischen den Klassen besteht.

Ich möchte allerdings auch nicht verneinen, dass wir Themen offen gelassen haben. Ich möchte wieder auf weiterführende Lehrveranstaltungen verweisen, aber trotzdem einen Überblick geben:

- Was ist wenn wir eine beschränkte Problemgröße haben? n kann nicht ins Unendliche wachsen. In diesem Fall eignet sich die $\mathcal{O}$-Notation nicht gut.
- Kann ich mehrere Input-Variablen modellieren?
- Wie analysiert man rekursive Funktionen?
- Wie stehen obere, untere und exakte Schranken zueinander? Welche Rechenregeln ergeben sich?

Klasse	Beschreibung
$\mathcal{O}(1)$	konstant
$\mathcal{O}(\log n)$	logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}\left(n^k\right)$	polynomiell k-ten Grades (mit $k > 3$)
$\mathcal{O}(2^n)$	exponentiell zur Basis 2
$\mathcal{O}(3^n)$	exponentiell zur Basis 3
$\mathcal{O}(k^n)$	exponentiell zur Basis k

Tabelle 8.2: Klassen nach $\mathcal{O}$-Notation.

Beispielfunktionen und deren Klassenzugehörigkeit sind in Tabelle 8.3 gegeben.

$$f_1(n) = 1 \qquad f_2(n) = 42 \qquad f_3(n) = \log n \qquad f_4(n) = \log_2 n$$
$$f_5(n) = n \qquad f_6(n) = 2n \qquad f_7(n) = n^2$$
$$f_8(n) = n^3 \qquad f_9(n) = 3^n \qquad f_{10}(n) = 3^{n+1} \qquad f_{11}(n) = 3^{n-1337}$$

$=$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(3^n)$
f_1	✓	✓	✓	✓	✓	✓	✓
f_2	✓	✓	✓	✓	✓	✓	✓
f_3	✗	✓	✓	✓	✓	✓	✓
f_4	✗	✓	✓	✓	✓	✓	✓
f_5	✗	✗	✓	✓	✓	✓	✓
f_6	✗	✗	✓	✓	✓	✓	✓
f_7	✗	✗	✗	✓	✓	✓	✓
f_8	✗	✗	✗	✗	✓	✓	✓
f_9	✗	✗	✗	✗	✗	✗	✓
f_{10}	✗	✗	✗	✗	✗	✗	✓
f_{11}	✗	✗	✗	✗	✗	✗	✓

Tabelle 8.3: Klassifikation der Funktionen f_1 bis f_{11}. Wähle eine Funktion (Zeile) und schaue nach ob sie in der Klasse (Spalte) enthalten ist. ✓ für „Ja“ und ✗ für „Nein“.

Kapitel 9

Algorithmen

Quod erat faciendum

lat. „Was zu machen war“

Algorithmen sind schrittweise Anleitungen zur Lösung eines Problems. Sie bilden damit eine Sequenz von Instruktionen, welche es uns ermöglicht Datenmodifikationen systematisch vorzunehmen und neue Daten zu produzieren. Sie können dabei beliebig aufgeteilt werden, sodass typische Programme aus einer Reihe von Algorithmen zur Lösung von Teilproblemen bestehen. Es ist Aufgabe des Entwicklers die Logik des Algorithmus entsprechend seiner Anforderungen korrekt zu implementieren und für variable Daten alle Spezialfälle abzudecken.

Der Algorithmusbegriff wurde im 9. Jahrhundert durch Muhammed al-Chwarizmi geprägt. Wie (viele nachfolgende) Mathematiker beschrieb er Verfahren mit denen Zahlen nach einer Vorschrift schrittweise transformiert wurden. Algorithmen wurden zum Instrument um Berechnungen durchzuführen und mit der industriellen Revolution stieg auch das Verlangen diese Berechnungen automatisiert auf Maschinen durchzuführen.

Ein wichtiger Lernprozess bei der Entwicklung von Algorithmen ist das *algorithmische Denken*. Eine allgemeine Datentransformation in Worten zu fassen scheint leichter als die schrittweise Formulierung der Logik. Erst durch die Aneignung von algorithmischem Denken können Prozesse in Programmiersprachen formalisiert werden.

Ein kleines Beispiel für algorithmisches Denken ist etwa das Tauschen zweier Zahlen. Stellt man einer Person die Frage, wie sie zwei auf einer Tafel dargestellte Zahlen tauschen würde, wählen die meisten Personen eine visuellen Ansatz und deuten mit Fingern an, wie die Position der einzelnen Zahlen zu wechseln ist. Auf einer Rechenmaschine gibt es das Konzept der

Positionen auf einer zweidimensionalen Fläche nicht und auch das simultane Austauschen ist nicht möglich. Die 2 Zahlen liegen in einer Maschine auf 2 verschiedenen Speicherbereichen. Der algorithmische Ansatz verwendet zum Tauschen einen dritten Speicherbereich als Hilfsvariable (hier: h). Wir können nun folgenden Algorithmus formulieren, welcher pro Schritt eine Zuweisung eines Wertes an eine Variable beschreibt.

```
h = a
a = b
b = h
```

Abbildung 9.1: Dreieckstausch zweier Variablen a und b.

9.1 Programmiersprachen

Programmiersprachen sind künstliche Sprachen, die formal spezifiziert sind und durch ein syntaktisch korrektes Programm kann das Verhalten einer Rechenmaschine gesteuert werden. Der Zusammenhang zu Algorithmen liegt in der Tatsache, dass Algorithmen in Programmiersprachen notiert werden, damit sie automatisiert von unserem Computer ausgeführt werden können. Compiler ermöglichen es uns die Programme nicht als direkte Instruktionen einer CPU zu schreiben, sondern machen es möglich den Algorithmus in einer höheren Sprache zu schreiben und sie dann automatisiert für die CPU herunterbrechen zu lassen. Dadurch wird insbesondere die Lesbarkeit der Algorithmen verbessert, da ein Algorithmus nicht mit zu vielen Details vorliegt.

9.2 Pseudocode

Als Pseudocode wird Quelltext verstanden, der in keiner formal spezifizierten Sprache vorliegt, sondern nur den algorithmischen Ansatz skizziert. Allgemein übliche Konstrukte (if-then-else, while Schleifen) werden in einer verständlichen Form von Programmiersprachen übernommen.

9.3 Eigenschaften von Algorithmen

Algorithmen repräsentieren Verarbeitungssequenzen. Diese Sequenzen besitzen Eigenschaften, die analysiert und optimiert werden können. Wir wollen die

Frage beantworten, welche Eigenschaften das sind:

Platzkomplexität Wieviel Speicher braucht der Algorithmus?

Laufzeit / Zeitkomplexität Wie lange benötigt der Algorithmus zur Verarbeitung?

Determinismus Liefert der Algorithmus immer das selbe Ergebnis für die gleiche Eingabe?

Strategie Divide and Conquer, Dynamic Programming, Heuristische Verfahren, ...

Während die letzten beiden Eigenschaften neutral sind, sind die ersten beiden der Hauptgrund einen Algorithmus einem anderen vorzuziehen. Platz (also Speicher) ist nicht immer ausreichend vorhanden und die Laufzeit soll gering gehaltet werden, um Zeit und Energie zu sparen.

9.4 Rekursive und iterative Algorithmen

Eine weitere Eigenschaft von Algorithmen ist ein rekursives Verhalten. Man hat es geschafft, Maschinen so zu organisieren, dass sie ein rekursives Verhalten verarbeiten können. Damit sind auch rekursive Algorithmen interessant geworden, die oft einfach und klarer zu verstehen sind als ihr iteratives Äquivalent. Was bedeutet aber nun *Rekursion*?

> **Re · kur · si · on**; die, -/-, siehe Rekursion

Rekursion bedeutet die Definition einer Sache durch sich selbst. Im obigen Zitat wird etwa Rekursion durch sich selbst definiert und soll damit den Begriff illustrieren[1]. Äquivalent sind rekursive Algorithmen so aufgebaut, dass sie in ihrer Definition sich selbst nochmals aufrufen. Dabei besitzt der rekursive Aufruf jedoch typischerweise unterschiedliche Parameter als der Aufruf zuvor, sodass es irgendwann zum Eintreten der *Abbruchbedingung* kommt, die dieses rekursive Verhalten abbricht. Die Fibonacci-Zahlen lassen sich über eine Rekursion darstellen:

Listing 9.1: Haskell Quelltext zur Berechnung der Fibonacci-Zahlen.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

[1] Die Suchmaschine Google fragt (als Easteregg) bei der Eingabe von „recursion“ nach, ob der Benutzer denn nicht „recursion“ gemeint hat.

Im Listing 9.1 wird in der dritten Zeile die Funktion fib durch einen Aufruf von sich selbst definiert. Die Zeilen 1 und 2 stellen Abbruchbedingungen dar. Das iterative Äquivalent in der Programmiersprache C sieht etwa so aus wie Listing 9.2.

Listing 9.2: C Quelltext zur Berechnung der Fibonacci-Zahlen.

```
#include <stdio.h>

int fib(int n)
{
    int a = 0, b = 1, c, i;
    for (i=0; i<n-1; i++)
    {
        c = b;
        b = a + b;
        a = c;
    }
    return a;
}

int main()
{
    printf("%d\n", fib(20));
    return 0;
}
```

Kapitel 10

Komplexität und Nicht-Determinismus

In Kapitel 5 haben wir formalisiert, was es bedeutet eine Berechnung durchzuführen. Die Turingmaschine ist hierfür unser Werkzeug. Zugleich haben wir in Kapitel 8 ein relatives Maß zur Beschreibung der Effizienz von Algorithmen eingeführt. Diese Konzepte bilden die Grundlage für die Komplexitätstheorie, welche wir in diesem Kapitel vorstellen wollen.

Im Kapitel 8 haben wir definiert, dass die $\mathcal{O}$-Notation sowohl auf die Laufzeit als auch andere Ressourcen angewandt werden kann. Wir möchten in diesem Kapitel die $\mathcal{O}$-Notation auf Laufzeit und Speicherverbrauch anwenden.

Wir müssen jetzt unterscheiden beginnen, ob wir eine *deterministische Turingmaschine* (jene Turingmaschine, die wir bisher betrachtet haben) oder eine *nichtdeterministische Turingmaschine* (Turingmaschine, die Nichtdeterminismus unterstützt so wie er auf Seite 37 beschrieben ist).

10.1 Die Klasse $\mathcal{P}$

Wir möchten nun Klassen für Probleme definieren. Wir definieren die *Klasse* $\mathcal{P}$: Das ist jene Klasse, die Probleme umfasst, die auf einer deterministischen Turingmaschine (DTM) in polynomieller Laufzeit gelöst werden können.

Wir haben früher verschiedene Algorithmen betrachtet deren Laufzeit wir asymptotisch analysiert haben. All jene Algorithmen, deren Laufzeit konstant, logarithmisch, linear oder polynomiell mit n^k sind, liegen in der Klasse $\mathcal{P}$. Dazu zählt Binärsuche oder all jene Algorithmen, die eine fixe Anzahl von verschachtelten Schleifen über die Eingabemenge zur Lösung benötigen.

10.2 Die Klasse $\mathcal{NP}$

Nichtdeterminismus beschreibt die Situation, wenn mehrere Übergänge für eine gegebene Konfiguration existieren. Auf Basis dieses Konzepts definieren wir eine Turingmaschine, welche mehrere Übergänge pro Konfiguration besitzen kann. Dabei wird dieser Turingmaschine eine Orakelturingmaschine zur Verfügung gestellt, welche bei jedem mehrdeutigen Übergang entscheiden kann, welcher von den möglichen Wegen zur korrekten Lösung führt und teilt diese Auswahl unserer nichtdeterministischen Turingmaschine mit.

Für die Klasse $\mathcal{NP}$ gibt es 2 verschiedene Definitionen:

- Die Klasse $\mathcal{NP}$ umfasst all jene Probleme, die auf einer nichtdeterministischen Turingmaschine in polynomieller Zeit gelöst werden können.
- Gegeben sei eine Lösung für ein Problem. Die Klasse $\mathcal{NP}$ umfasst all jene Probleme für die eine Lösung (ein sogenanntes „Zertifikat“) in polynomieller Zeit überprüft („verifiziert“) werden kann.

10.3 Das Erfüllbarkeitsproblem SAT

Elementar für die Betrachtung der Klassen $\mathcal{P}$ und $\mathcal{NP}$ ist das Problem SAT (kurz für englisch „Satisfiability“). Dieses stellt die Frage, ob für eine gegebene boolsche Formel (siehe Kapitel 3) eine erfüllende Belegung gefunden werden kann.

Gegeben sei eine Lösung für SAT (also eine erfüllende Belegung für eine gegebene boolsche Formel). Es ist recht naheliegend, dass die Lösung in polynomieller Zeit verifiziert werden kann. Schließlich ist es dazu nur notwendig alle Operatoren zu evaluieren. Die Anzahl der Operatoren ist nur polynomiell abhängig von der Anzahl der Variablen (was unser n in der $\mathcal{O}$-Notation repräsentieren würde), die in der Formel vorkommen. Daher können wir eine Lösung in polynomieller Zeit verifizieren. Damit ist Definition 2 der Klasse $\mathcal{NP}$ erfüllt.

Für Definition 1 argumentieren wir wie folgt: Wir definieren einen Algorithmus, welcher alle verwendeten Variablen ermittelt (möglich in polynomieller Zeit). Nachfolgend definieren wir nichtdeterministisch eine Zuordnung der Wahrheitswerte zu den Variablen. Diese Zuordnung gibt uns der Algorithmus zurück. Da diese Zuordnung nichtdeterministisch abgelaufen ist, hat uns die Orakelturingmaschine mitgeteilt, welche Zuordnung die Korrekte ist. Auf diese Weise müssen wir nicht alle 2^n Variablenbelegungen (sondern nur eine) durchprobieren und erhalten damit einen polynomiellen, nichtdeterministischen Algorithmus.

10.4 Reduktionsbeweise

Angenommen wir hätten einen polynomiellen Algorithmus, welcher uns feststellt, ob eine Formel erfüllbar ist. Wir wollen weiters das Problem lösen festzustellen, ob eine Formel nicht erfüllbar ist. In diesem Fall werden wir darauf verzichten einen neuen Algorithmus zu formulieren (sofern er uns keine Vorteile bietet) und den bestehenden Algorithmus wiederverwenden.

Wiederverwendung kann uns helfen Denkarbeit zu sparen. Wir führen noch einen weiteren Gedanken: Wir definierten, dass dieser Algorithmus in polynomieller Zeit Ergebnisse liefert. Die Negation des Ergebnisses ist eine konstante Operation und kann auch durch einen Polynom beschrieben werden. Daraus folgt, dass die gesamte Laufzeit des Algorithmus polynomiell ist. Dem liegt zugrunde, dass die Addition zweier Polynome wieder ein Polynom ist.

Diese Wiederverwendung eines Algorithmus und verarbeiten in einer oberen Zeitschranke wird als „Reduktion“ bezeichnet. Wir können das Problem $\overline{\text{SAT}}$ (ob eine boolsche Formel nicht erfüllbar ist) mit einer polynomiellen Reduktion auf SAT (ob eine boolsche Formel erfüllbar its) zurückführen. Wir schreiben:

$$\overline{\text{SAT}} \leq_p \text{SAT}$$

10.5 $\mathcal{NP}$-Vollständigkeit

Unter $\mathcal{NP}$-Vollständigkeit versteht man eine Eigenschaft von Problemen. $\mathcal{NP}$-vollständig sind jene Probleme aus $\mathcal{NP}$, die polynomiell auf SAT reduziert werden können. Sie sind die „schwersten Probleme“ dieser Klasse. Um $\mathcal{NP}$-Vollständigkeit zu zeigen, müssen folgende Dinge bewiesen werden:

membership Das Problem liegt in $\mathcal{NP}$.

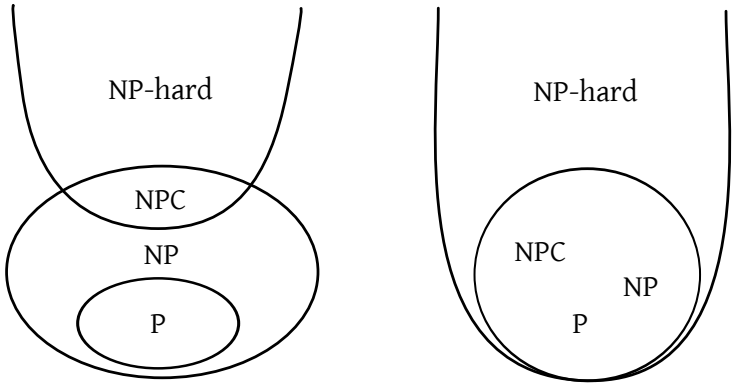
hardness Das Problem kann polynomiell auf SAT reduziert werden.

Interessant ist hierbei, dass es sich um eine transitive Beziehung handelt. Jedes $\mathcal{NP}$-vollständige Problem kann nicht nur auf SAT, sondern jedes beliebige andere $\mathcal{NP}$-vollständige Problem reduziert werden.

Die Klasse der $\mathcal{NP}$-vollständigen Probleme nennt sich $\mathcal{NPC}$.

10.6 $\mathcal{P}$ versus $\mathcal{NP}$

Die Frage $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ stellt die wichtigste offene Frage der theoretischen Informatik: Liegen alle Probleme in $\mathcal{NP}$ in $\mathcal{P}$? Können wir also für jeden Algorithmus auf einer nichtdeterministischen Turingmaschine (welcher polynomielle Zeit



(a) wenn $\mathcal{P} \neq \mathcal{NP}$. (b) wenn $\mathcal{P} = \mathcal{NP}$.

Abbildung 10.1: Venndiagramm der Klassen $\mathcal{NP}$, $\mathcal{P}$ und $\mathcal{NPC}$ unter beiden Annahmen

benötigt) einen äquivalenten Algorithmus auf einer deterministischen Turingmaschine finden, welcher auch im schlechtesten Fall nur polynomielle Zeit benötigt?

Wir haben festgestellt, dass Probleme aus $\mathcal{NPC}$ die schwersten Probleme von $\mathcal{NP}$ sind. Können wir für einen dieser Probleme einen deterministischen Algorithmus finden, welcher in polynomieller Zeit eine korrekte Lösung findet? Dann fallen die Klassen $\mathcal{P}$ und $\mathcal{NP}$ zusammen, da dann alle „einfacheren“ (= nicht-vollständigen) Probleme in $\mathcal{NP}$ auch in polynomieller Zeit gelöst werden können und zwischen allen Problemen in $\mathcal{NPC}$ eine transitive Beziehung besteht.

In Abbildung 10.1 ist dargestellt in welcher Beziehung $\mathcal{P}$, $\mathcal{NP}$ und $\mathcal{NPC}$ zueinander stehen unter der Annahme, dass $\mathcal{P} \subset \mathcal{NP}$ bzw. $\mathcal{P} = \mathcal{NP}$. $\mathcal{NP}$-hart sind all jene Probleme, die sich auf SAT reduzieren lassen.

Kapitel 11

Formale Sprachen

Die Informatik macht sich Gedanken um die systematische Verarbeitung von Information. Anhand verschiedener Modelle haben wir bereits gesehen, wie wir eine Eingabe nehmen, diese nach Vorschriftsregeln verarbeiten und dann eine Antwort auf das gestellte Anfangsproblem liefern. Dabei haben wir eine wichtige Frage wegfallen lassen: Woher weiß der Algorithmus, ob diese Eingabe valid ist? Woher weiß die Turingmaschine, dass das (was auf dem Band steht) eine korrekte Eingabe ist, die zu keinem undefinierten Verhalten führt?[1]

Das Wortproblem (WP) stellt folgende Frage:

> Gegeben sei eine Sprache S. Ermittle, ob ein Wort W Teil von S ist oder nicht.

Wir verwenden die neuen Begriffe *Wort* (oder String bzw. Zeichenkette) und *Sprache* (Vorschriften über die mögliche Aneinanderreihung von Wörtern). Eine Sprache beschreibt eine Menge von Wörtern[2]. Für diese muss eine Zugehörigkeitsbeziehung ermittelt werden können. Diesem Problem begegnen Informatiker in der Praxis ständig, da zB verifiziert werden muss, ob es sich bei einer Zeichenkette um eine Emailadresse handelt.

Wir eignen uns den theoretischen Hintergrund an und führen das Konzept der Sprachen und Grammatiken ein:

11.1 Eine einfache Sprache

Gegeben sei eine einfache Sprache S, die genau nur 1 Zeichenkette akzeptieren soll und zwar jene, die nur aus „`a`“ besteht. Stimmt das Eingabewort W

[1] Als Beispiel für undefiniertes Verhalten sei hier die Division durch Null gegeben.

[2] Beachte, dass Wort anders als im linguistischen Sinn betrachtet wird. So ist ein Satz wie „Der Gärtner ist der Mörder“ *ein* Wort einer Sprache.

mit der Zeichenkette „a“ überein, wird die Eingabe akzeptiert; andernfalls zurückgewiesen. So wird das Wortproblem entschieden. Da wir die Sprache als Menge repräsentieren können, verwenden wir Mengennotation: $S = \{\texttt{a}\}$ und das Wortproblem für einen String s lautet $s \stackrel{?}{\in} S$.

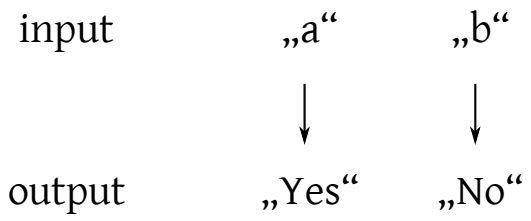


Abbildung 11.1: Zwei Beispiele für die Entscheidung des Wortproblems für die Sprache „a“.

Ein komplexeres Beispiel verwendet mehrere Zeichen: Die Sprache T als „abb“ akzeptiert genau jene Wörter, die eine Länge 3 und den Aufbau „abb“ haben.

Wir möchten in diesem Kapitel also Sprachen definieren. Wir haben eine textuelle Beschreibung verwendet. Wir haben auch Mengennotation verwendet. Wobei diese Ansätze durchaus legitim sind, besitzt der textuelle Ansatz das Problem fehleranfällig zu sein (Spezialfälle werden nicht spezifiziert) und Mengennotation kann komplexere Sprachen nicht übersichtlich abbilden. Wir befassen uns daher mit zwei weiteren Ansätzen in den folgenden Abschnitten:

- reguläre Ausdrücke (kompakte Schreibweise für eine bestimmte Untermenge von Sprachen)
- formale Grammatiken (ein mächtiges Werkzeug um allgemeine Grammatiken zu beschreiben)

Unsere einfachen Sprachen in Mengennotation

$$S = \{\texttt{a}\} \qquad T = \{\texttt{abb}\}$$

Unsere einfachen Sprachen als regulärer Ausdruck

Listing 11.1: S als regulärer Ausdruck.

```
a
```

Listing 11.2: T als regulärer Ausdruck.

```
abb
```

Unsere einfachen Sprachen als formale Grammatik

Listing 11.3: S als formale Grammatik $G = (V, \Sigma, P, S)$.

```
S → a
```

Listing 11.4: T als formale Grammatik $G = (V, \Sigma, P, S)$.

```
S → abb
```

11.2 Eine Sprache mit Wiederholungen

Wir möchten unsere Sprachendefinitionen erweitern, indem wir eine Anzahl abbilden können. So war es uns bislang nicht möglich eine Sprache (zB) „Akzeptiere alle Wörter die nur aus as bestehen“ zu formulieren. Hierfür müssen wir Wiederholungen repräsentieren.

11.2.1 In regulären Ausdrücken

Wir führen das Konzept der Quantoren ein. Ein regulärer Ausdruck ist eine Zeichenkette, die ein oder mehrere Zeichenketten spezifiziert. Grundsätzlich wird für reguläre Ausdrücke jenes Wort notiert, welches innerhalb der Sprache liegt. Wir lernen jetzt Konzepte kennen, um mehrere Wörter abzubilden.

`+`	Das vorige Zeichen kommt 1–∞ mal vor („1-mal oder öfters“)
`*`	Das vorige Zeichen kommt 0–∞ mal vor („beliebig oft“)
`?`	Das vorige Zeichen kommt 0–1 mal vor („optional“)

Tabelle 11.1: Quantoren in regulären Ausdrücken.

In Tabelle 11.1 werden Quantoren definiert. Kommen diese Zeichen in einem regulären Ausdruck vor, werden diese nicht wortwörtlich verstanden, sondern beziehen sich auf das letzte Zeichen oder die letzte Gruppe.

Wir betrachten ein Beispiel. Gesucht sei folgende Sprache R:

> Das Zeichen „`a`“ kann (aber muss nicht) vorkommen. Nachfolgend kommt ein „`b`“ und anschließend beliebig viele „`c`“.

Der folgende Ausdruck spezifiziert diese Sprache:

Listing 11.5: Die Sprache R als regulärer Ausdruck.

```
a?bc*
```

Das erste Fragezeichen bezieht sich auf das vorige Zeichen „a“ und macht es optional. Das folgende „b“ muss zwingend vorkommen. Dem „c“ ist ein Stern nachgestellt und ein Stern spezifiziert die Quantität „beliebig oft“.

11.2.2 In Mengennotation

In Mengennotation wird R als $\{a^n bc^m | 0 \leq n \leq 1, m \geq 0\}$ notiert. Hierbei wird die Anzahl der Vorkommnisse eines Zeichens oder einer Gruppe mit einer hochgestellten Zahl notiert. Wird die Domäne der Variable (n und m) nicht angegeben, spricht man von natürlichen Zahlen inklusive der 0.

11.2.3 Als formale Grammatik

In einer formalen Grammatik ist Wiederholung über Rekursion abzubilden.

Aber besprechen wir zuerst, was formale Grammatiken denn sind und wie sie Wörter erzeugen. Eine formale Grammatik $G = (V, \Sigma, P, S)$ besitzt ein endliches Vokabular V, ein Alphabet Σ, Produktionsregeln P und ein Startsymbol S. Das Erzeugen von Strings startet stets mit dem Startsymbol S und Produktionsregeln P erlauben es uns Zeichen sukzessive durch Zeichen aus V zu ersetzen. Wir unterscheiden dabei zwischen den Terminalsymbolen Σ und Nonterminalsymbolen $V \setminus \Sigma$. Terminalsymbole sind jene Symbole, die in der endgültigen Zeichenkette vorkommen. Für Nonterminalsymbole sind Regeln in P definiert, wie man sie ersetzen kann. Die Ersetzungen erfolgen so lange bis nur mehr Terminalsymbole enthalten sind.

Als ersten Schritt (um R zu definieren) beschreiben wir eine Sprache $R1$ mit ‚Das Zeichen „`a`“ kann vorkommen‘.

Listing 11.6: $R1$ als formale Grammatik $G = (V, \Sigma, P, S)$.

```
S → a
S →
```

Wir starten also mit einem Startsymbol „S“ und ersetzen es jetzt. Entweder wir entscheiden uns für die Zeile 1 und setzen „a“ ein oder einen leeren String „“ (Zeile 2).

Als zweiten Schritt beschreiben wir eine Grammatik $R2$ mit ‚Das Zeichen „`a`“ kann vorkommen. Nachfolgend kommt ein „`b`“‘.

Listing 11.7: $R2$ als formale Grammatik $G = (V, \Sigma, P, S)$.

```
S → ab
S → b
```

Nachdem ein optionales `a` angegeben wurde, folgt ein `b`. Für die Wiederholung von `c` verwenden wir ein Nonterminalsymbol `T` (welches per Konvention großgeschrieben wird).

Listing 11.8: R als formale Grammatik $G = (V, \Sigma, P, S)$.

```
S → abT
S → bT
T →
T → cT
```

Gehen wir die Produktionsregeln für den Eingabestring `abcc` durch. Wir starten wieder mit dem Startsymbol „S“. Wir ersetzen es mittels der 1. Zeile zu „abT“. Wir erkennen ein verbleibendes Nonterminal in „T“ in dem String und suchen uns hierfür entsprechende Regeln; die Regeln 3 und 4 kommen in Frage. Für den gegebenen Eingabestring müssen wir Regel 4 verwenden. Es entsteht der String „abcT“. Wieder ersetzen wir das „T“ durch die Regel 4. „abcT“ besitzt auch noch ein Nonterminal, welches jetzt jedoch mittels Regel 3 ersetzt wird. Es entsteht der String „abcc“ (bestehend nur aus Terminalsymbolen). Damit liegt der String „abcc“ in der durch die formale Grammatik spezifizierten Sprache.

11.3 Alternation in Sprachen

Gesucht sei folgende Sprache A:

> Die Zeichenkette beginnt mit „message of the “ und endet entweder mit „day“ oder „week“. Am Ende befindet sich weiters ein Rufzeichen „!“.

11.3.1 Als regulärer Ausdruck

In einem regulären Ausdruck lässt sich eine Alternation definieren, indem der vertikale Balken als Alternationsoperator verwendet wird. Dabei wird entweder die linke Seite vom Balken oder die rechte Seite herangezogen. Die Seiten können durch eine Gruppierung beschränkt werden, die durch runde Klammern definiert wird.

Listing 11.9: Die Sprache A.

```
message of the (day|week)!
```

11.3.2 Als formale Grammatik

Alternation haben wir schon implizit bei formalen Grammatiken kennen gelernt. Wir wählen aus mit welcher Regel wir ein Nonterminalsymbol ersetzen.

Listing 11.10: A als formale Grammatik $G = (V, \Sigma, P, S)$.

```
S → message of the T!
T → day
T → week
```

Wir haben jetzt die wichtigsten Konzepte betrachtet, um Sprachen definieren zu können. Wir möchten jetzt noch auf ein paar spezifische Details eingehen und die Mächtigkeit von Sprachen evaluieren.

11.4 Reguläre Ausdrücke

`.`	Ein beliebiges Zeichen (aus Σ)
`[ab]`	Eines der Zeichen aus {`a`, `b`} (Zeichenmenge)
`(ab)`	Fasst die Sequenz „ab“ zu einer Gruppe zusammen (Gruppierung)
`a|bc`	Entweder „a“ oder „bc“ (Alternation)
`+`	Das vorige Zeichen kommt 1–∞ mal vor („1-mal oder öfters“)
`*`	Das vorige Zeichen kommt 0–∞ mal vor („beliebig oft“)
`?`	Das vorige Zeichen kommt 0–1 mal vor („optional“)

Tabelle 11.2: RegEx Operatoren.

Reguläre Ausdrücke (engl. „regular expressions“ oder kurz RegEx) sind eine kompakte Schreibweise, um mehrere Zeichenketten darzustellen. In Tabelle 11.2 sind die Sonderzeichen nochmals zusammengefasst dargestellt. In der Tabelle wird eine Möglichkeit genannt eine Zeichenmenge anzugeben. Dieses Konzept kann genauso durch eine Alternation simuliert werden.

11.4.1 Non-greedy Verhalten

RegEx-Implementierungen gibt es unterschiedliche. Die zwei großen Implementierungen nennen sich „POSIX“ und „PCRE“. Diese unterscheiden sich

durch ihren Aufbau und ihre erweiterten Möglichkeiten. Ich möchte hier auf ein Detail eingehen.

Eine häufig vorkommende Struktur bei regulären Ausdrücken ist `(.*)`. Es handelt es sich um eine Gruppe von beliebigen Zeichen beliebiger Länge. Salopp gesprochen matcht dieser Ausdruck jeden String. Wie sieht es nun mit dem regulären Ausdruck `(.*)` und der Eingabe „abc“ aus? Greedy-Verhalten bezeichnet das Bestreben eines Ausdrucks, möglichst viel Inhalt zu matchen. Entsprechend würde dieser Ausdruck die gesamte Eingabe matchen, da sie versucht sich alles zuzuweisen. Dies verwirrt viele RegEx-Benutzer, da ein RegEx `a(.*)a` bei dem Eingabestring „abaaba“ nicht „aba“ zweimal matcht, sondern einmal „abaaba“. RegEx sind für gewöhnlich immer bestrebt möglichst viel zu matchen.

Non-greedy Verhalten kann man durch das Anfügen eines Fragezeichens nach dem Gruppenquantor erreichen; ist allerdings nur bei „PCRE“-Engines verfügbar. Für unser Beispiel wäre etwa `a(.*?)a` sinnvoll um den String in zwei Teilen zu matchen.

11.4.2 Deterministische Automaten

Abseits der Strings mit den jeweiligen Operatoren gibt es auch eine andere Darstellungsweise für reguläre Ausdrücke. Zu jedem regulären Ausdruck kann ein deterministischer Automat gefunden werden. Ein deterministischer Automat ist ein gerichteter, kantenbeschrifteter Graph. Wir erinnern uns, dass ein Graph aus Knoten und Kanten besteht. Es gibt einen Startknoten[3] und von diesem ausgehend werden die Zeichen vom Eingabestring gelesen. Je nach gelesenem Zeichen folgt man jenen Kanten, deren Beschriftung mit dem Zeichen übereinstimmt. Ein Eingabestring wird genau dann von der Sprache akzeptiert, wenn der String auf einem Endknoten zu Ende geht, der durch einen zusätzlichen inneren Kreis markiert ist.

Das Wesentliche ist, dass dieser Automat das Kriterium des Determinismus zu erfüllen hat. Dies bedeutet in jedem beliebigen Zustand des Automaten kann durch das gelesene Zeichen eindeutig bestimmt werden welcher Kante zu folgen ist. Oder in anderen Worten gibt es keine zwei Kanten mit der selben Beschriftung, die am selben Ausgangsknoten starten. In Abbildung 11.2 ist ein nicht-deterministischer Automat dargestellt, der diese Regel bricht.

Bei der Erstellung eines Automaten wird folgender Ansatz empfohlen:

1. Repräsentiere die Operatoren äquivalent, wie sie in Abbildung 11.3

[3] In diesem Dokument folgen wir der Konvention, dass der Startknoten der am weitesten links positionierte Knoten ist. Im Allgemeinen handelt es sich um den einzigen Quellknoten im Graphen.

dargestellt werden.

2. Reduziere den Automaten, indem unnötige Strukturen entfernt werden (zB wenn die linke und rechte Seite einer Alternation äquivalent ist).

3. Entferne Nichtdeterminismus.

Es gibt keinen universellen Algorithmus um Nichtdeterminismus entfernen zu können. Für jede reguläre Sprache existiert jedoch ein deterministischer Automat. Es sei folgender Hinweis gegeben: Während die Operatoren darauf optimiert sind anzugeben wie oft ein Zeichen oder eine Sequenz vorkommt, stellt der Automat stets die Frage „welches Zeichen kommt als nächstes?“. Es empfiehlt sich daher Beispielstrings aufzuschreiben und bei jedem Zeichen die Frage nach dem nächsten möglichen Zeichen zu stellen. Die Abzweigungen im Automaten verändern sich dadurch oft stark. In Abbildung 11.4 sind einige Beispielautomaten gegeben.

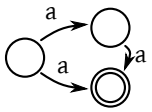


Abbildung 11.2: Beispiel eines nicht-deterministischen Automaten.

11.4.3 Konventionen und Beispiele

Ich möchte hier noch auf Konventionen im Bereich der formalen Grammatiken aufmerksam machen. So ist es üblich den leeren String „“ mit einem ε darzustellen, um Missverständnisse vorzubeugen und ihn als druckbares Zeichen darzustellen.

Wendet man reguläre Ausdrücke in Programmiersprachen an, entdeckt man, dass Ausdrücke auf Teile des Gesamtwortes angewendet werden und nicht den gesamten String matchen müssen. Hierfür existieren die Operatoren ^ und $, welche den Anfang und das Ende eines Strings repräsentieren. Jenes Zeichen welches einem $ vorangestellt ist, muss also das letzte Zeichen des Eingabestrings sein. Die Verwendung dieser Operatoren ist im sprachtheoretischen Kontext nicht notwendig, da stets davon ausgegangen wird, dass der gesamte String gematcht werden soll.

In Tabelle 11.3 sind Beispiele für reguläre Ausdrücke dargestellt.

matcht	*matcht nicht*
`(ab)?|cd`	
ε	abcd
ab	abab
cd	
`(ab|cd)*`	
ε	a
abcd	d
abab	abc
cd	
`a?a+`	
a	ε
aa	b
aaa	
`(a(b|c)+d|e?)+`	
ε	ad
acd	ea
abcd	d
acccde	
eabbde	
`a?b*c+`	
abc	a
ac	ab
c	aabc
bbbc	
`(a?b?b)*a?`	
ab	aa
abbb	baaba
ababbabbba	bbbaaa
`(ab+a*)+ab+`	
abab	aba
abbbab	abb
abbbaaabbbab	abbaaaba
abaabbb	abbbaaba

Tabelle 11.3: Beispiele für reguläre Ausdrücke und deren Matching-Verhalten.

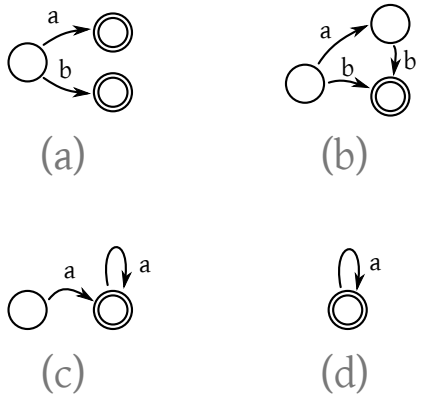


Abbildung 11.3: Die deterministischen Automaten der Sprachen (a) „`a|b`“ (b) „`a?b`“ (c) „`a+`“ (d) „`a*`“

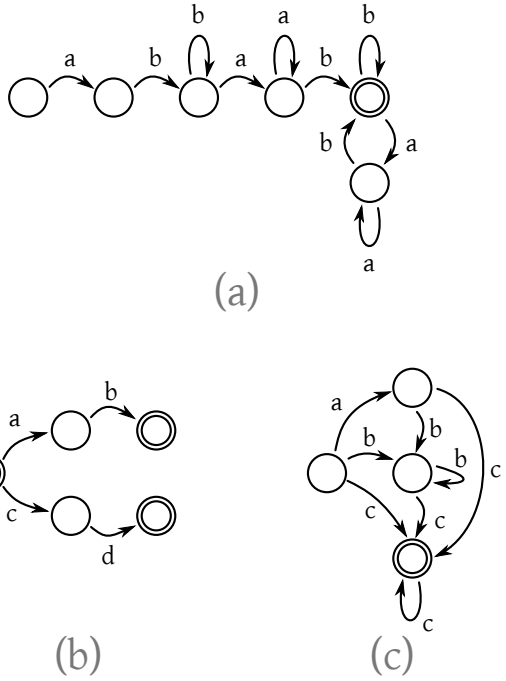


Abbildung 11.4: Die deterministischen Automaten der Sprachen (a) „`(ab+a*)+ab+`“ (b) „`(ab)?|cd`“ (c) „`a?b*c+`“

11.5 Kontextfreie Sprachen

Kontextfreie Sprachen (engl. „context-free languages“) sind mächtiger als reguläre Sprachen. Diese Behauptung besagt, dass die Menge der Sprachen die durch reguläre Ausdrücke repräsentiert werden können kleiner als die Menge der Sprachen ist, die mit kontextfreien Sprachen spezifiziert werden können. Wir betrachten ein Beispiel.

Gesucht sei folgende Sprache E:

> Die Anzahl von „a“ ist gleich der Anzahl von „b“.

Probieren wir einen regulären Ausdruck zu finden.

Listing 11.11: Regulärer Ausdruck für die Sprache E.

```
(ab)*
```

Die formale Grammatik für E sieht folgendermaßen aus:

Listing 11.12: Sprache E als formale Grammatik.

```
S → abS
S →
```

Bisher war jedoch die Reihenfolge der Zeichen undefiniert. Wir erweitern nun E zur Sprache F:

> Eine Sequenz von „a“ wird von einer Sequenz der gleichen Länge von „b“ gefolgt.

In Mengennotation wäre dies mit $\{a^n b^n \mid n \in \mathbb{N}, n \geq 0\}$ anzugeben.

Durch die Vorgabe der Position bekommen wir ein neues Problem. Wir versuchen es mit folgendem regulären Ausdruck:

Listing 11.13: Versuch eines regulären Ausdrucks für die Sprache F.

```
a*b*
```

Das geübte Auge erkennt, dass die Anzahl der erzeugten „a“ ungleich der Anzahl von „b“ sein kann. Daher repräsentiert dieser reguläre Ausdruck nicht die spezifizierte Sprache. So ist etwa „aaab“ in der Sprache des regulären Ausdrucks enthalten, aber nicht in der textuellen Beschreibung. Wir erkennen, dass die vorliegende Sprache sich nicht mehr regulär abbilden lässt, sondern „kontextfrei“ ist. Es handelt sich um unsere erste Sprache, die in der Menge der kontextfreien Sprachen aber nicht in der Menge der regulären Sprachen liegt.

Mithilfe einer formalen Grammatik können wir diese Sprache jedoch definieren:

Listing 11.14: Formale Grammatik für Sprache F.

```
S → aSb
S →
```

Basierend auf S können wir jetzt beliebige Wörter der Sprache ableiten. So nehmen wir etwa $n = 3$ an:

$$\text{S} \rightarrow \text{aSb} \rightarrow \text{aaSbb} \rightarrow \text{aaaSbbb} \rightarrow \text{aaaSbbb} \rightarrow \text{aaabbb}$$

Für kontextfreie Grammatiken dürfen wir Nonterminale definieren, die ersetzt werden. Die Idee für diese Grammatik ist es bei jeder Rekursion auf der linken und rechten Seite der Rekursion Zeichen zum endgültigen String hinzuzufügen.

Wir müssen dabei beachten, dass wir auf der linken Seite nur ein Nonterminal verwenden dürfen, um wirklich eine kontextfreie Sprache zu spezifizieren.

11.6 Kontext-sensitive Sprache

Eine weitere Menge stellen die kontext-sensitiven Sprachen dar. Dazu sei folgende Sprache zu formulieren:

$$H = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 0\}$$

Wir setzen wieder eine Grammatik an und versuchen durch Gegenbeispiele ihre Falschheit zu zeigen.

Listing 11.15: Lösungsversuch für Sprache H.

```
S → aSbB
S →
B → Bc
B →
```

Die Grammatik in Listing 11.15 ist inkorrekt. Mithilfe der Ableitung

$$\text{S} \rightarrow \text{aSbB} \rightarrow \text{abB} \rightarrow \text{abBc} \rightarrow \text{abBcc} \rightarrow \text{abcc}$$

können wir uns ein Wort bauen, welches aus der Grammatik aus Listing 11.15 stammt allerdings nicht Teil der Menge H ist. Daher ist die Grammatik fehlerhaft. Die Anzahl der „a“ und „b“ sind äquivalent, aber die „c“ sind unabhängig.

Wir müssen unsere Grammatik ausbauen und auf der linken Seite sowohl Terminale als auch Nonterminale verwenden:

Listing 11.16: Die kontext-sensitive Grammatik $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$.

```
S  →  aSBC
S  →
aB →  ab
bB →  bb
bC →  bc
CB →  BC
bC →  bc
cC →  cc
```

Zum ersten Mal verwenden wir ein Terminalsymbol und ein Nonterminalsymbol oder 2 Nonterminalsymbole zugleich auf der linken Seite. Dies ist erst bei kontext-sensitiven Sprachen erlaubt.

11.7 Chomsky-Hierarchie

Wir haben bei diesen Beispielen gesehen, dass nicht alle Grammatiken alle Sprachen abbilden können; daher manche mächtiger sind als andere. Noam Chomsky hat die Chomsky-Hierarchie formuliert, um Sprachen in Kategorien einzuteilen. Dabei sind reguläre Sprachen die schwächsten Sprachen. Kontextfreie Sprachen stehen über den regulären Sprachen und können auch alle regulären Sprachen abbilden. Am mächtigsten sind die rekursiv-aufzählbaren Sprachen, die es etwa erlauben die Anzahl der Rekursionen in den Ersetzungsregeln während der Verarbeitung zu verändern. Wir möchten hier auf weitere Unterteilungen und Details verzichten und auf die Lehrveranstaltung „Softwareparadigmen“ (716.060) verweisen, welche sich detailierter mit diesen Sprachen auseinander setzt.

Wir möchten noch kurz begründen und zusammenfassen, wie diese Unterscheidung der Klassen zustande kommt und wieso sie in der Praxis benötigt wird:

Reguläre Sprachen können effizient verarbeitet werden. Wir haben gesehen, dass wir jede reguläre Sprache durch einen endlichen, deterministischen Automaten verarbeiten können. Der endliche, deterministische Automat hat die Eigenschaft, dass in jedem Zustand sofort auf den Folgezustand geschlossen werden kann (sonst wäre das Kriterium des Determinismus nicht erfüllt). Somit kann das Wortproblem für eine gegebene Zeichenkette in linearer Zeit gelöst werden.

Kontextfreie Sprachen erfüllen dieses Kriterium (durch das Lesen eines Zeichens auf den Folgezustand schließen zu können) im Allgemeinen nicht. Dennoch können kontextfreie Sprachen in polynomieller Zeit und daher effizi-

ent verarbeitet werden. Sie haben als obere Schranke kubische Laufzeit um das Wortproblem zu entscheiden. In der Praxis wird für die formalen Sprachen Kontextfreiheit angestrebt, da sie eine gute Balance zwischen Mächtigkeit und Performance bieten.[4]

Kontext-sensitive Sprachen sind die ersten Sprachen, die nicht in polynomieller Zeit verarbeitet werden können. Sie erlauben es die Anzahl der Wiederholungen über mehrere Rekursionen zu bestimmen, sofern die Anzahl der Rekursionen vordefiniert ist. Man versucht kontext-sensitive Sprachen zu vermeiden.

Rekursiv-aufzählbare Sprachen können nur von Maschinen verarbeitet werden, die gleich mächtig wie eine Turingmaschine sind. Sie sind damit sehr mächtig, aber zu entscheiden ob ein Wort innerhalb einer solchen Sprache liegt, lässt sich möglicherweise gar nicht feststellen („Unentscheidbarkeit“).

Die Wahl eines ungeeigneten Werkzeugs um die Sprache zu verarbeiten oder die Einführung eines syntaktisch komplexen Konstrukts in einer Sprache kann zum Scheitern eines Softwareprojekts führen. Deshalb sei nochmals die Wichtigkeit dieses Themengebiets hervorgehoben.

Abbildung 11.5 visualisiert die Chomsky-Hierarchie.

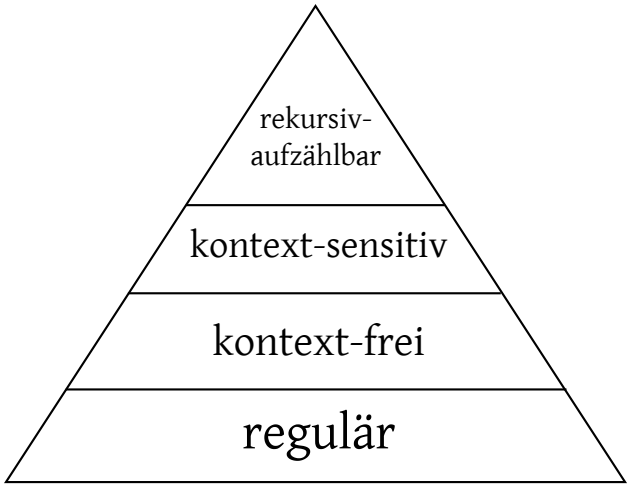


Abbildung 11.5: Die Chomsky-Hierarchie.

[4]Zum Beispiel sind die Markupsprache XML (Version 1.0) und Datenbankabfragesprache SQL (Version 2003) kontextfrei.

11.8 Anmerkung zu formalen Grammatiken

Wir haben in diesem Kapitel von 3 verschiedenen Notationen gesprochen:

Mengennotation Erlaubt es uns sehr schnell Sprachen aus allen vier Ebenen zu spezifizieren. Wird jedoch nicht in der Industrie angewandt.

Reguläre Ausdrücke Erlauben es reguläre Sprachen zu formulieren. In der Praxis kommen reguläre Ausdrücke in den Formen POSIX und PCRE vor. Diese Implementierungen erlauben es jedoch[5] sogar kontextsensitive Sprachen zu entscheiden. Man verwendet reguläre Ausdrücke primär, um einfache Sprachen (die typischerweise regulär sind) zu matchen.

Formale Grammatik Die Pfeilnotation erlaubt es uns Sprachen aus allen vier Ebenen anzugeben. Allerdings müssen Einschränkungen bezüglich des Aufbaus der Produktionsregeln beachtet werden, wenn es sich um keine rekursiv-aufzählbare Grammatik handelt. In Abbildung 11.6 sind die Einschränkungen aufgelistet.

Für die Notation von kontextfreien Sprachen würden sich also alle drei Notationen eignen. Wird eine Programmiersprache formal notiert, verwendet man Abwandlungen der formalen Grammatik: die Erweiterte Backus-Naur-Form (EBNF) oder Angereicherte Backus-Naur-Form (ABNF).

[5] PCRE implementiert Operatoren die Backreferences und Lookaheads genannt werden. Mit diesen können Sprachen entschieden werden, die weit über den regulären liegen.

Type	Language, Example	Machine	WP decidability	Grammar G = (V, Σ, P, S)
0	recursive enumerable $\{n^n\}$	Turing machine	undecidable	$\alpha A \beta \rightarrow \alpha \gamma \beta$ unrestricted
1	context-sensitive „natural languages“ $\{a^n b^n c^n\}$	linear bounded automaton		$\alpha A \beta \rightarrow \alpha \gamma \beta$ $A \in V$, $\alpha \in (V \cup \Sigma)^*$, $\beta \in (V \cup \Sigma)^*$, $\gamma \in (V \cup \Sigma)^+$
2	context-free $\{a^n b^n\}$	pushdown automaton	$O(n^3)$	$A \rightarrow \beta$ $A \in V$, $\beta \in (V \cup \Sigma)^*$
3	regular $\{a^n bc\}$	deterministic finite automaton	$O(n)$	$A \rightarrow a$ $A \rightarrow aB$ $A \rightarrow \varepsilon$ $a \in \Sigma$, $A \in V, B \in V$

Abbildung 11.6: Die 4 Typen von formalen Sprachen und deren Eigenschaften.

Kapitel 12

Bekannte Informatiker

Diese Liste erhebt keinerlei Anspruch auf Vollständigkeit und hebt nur Teile der Arbeiten jeweiliger Personen hervor.

John Backus * 1924 † 2007
: Leitung der Entwicklung der Programmiersprache Fortran. Bekannt für seine Backus-Naur-Form im Bereich der formalen Sprachen. Forschte anschließend im Bereich der funktionalen Programmierung.

Tim Berners-Lee * 1955
: Entwickelte am CERN den ersten Webserver und begründete damit das WWW. Bemüht sich um die Standardisierung von HTTP, HTML und anderen Technologien mithilfe seines W3C. Heute Fokus auf XML-Standards und seine Vision des „Semantic Web“.

George Boole * 1815 † 1864
: Untersuchte und entwickelte jenen Logikkalkül, der Grundlage der modernen Logik heute ist. Ihm zu Ehren sprechen wir von boolschen Variablen und boolschen Formeln.

Noam Chomsky * 1928
: Amerikanischer Linguist, welcher durch Untersuchung der formalen Sprachen eine Klassifikation („Chomsky-Hierarchie“) der Sprachen nach ihrer Mächtigkeit erreichte. Trug damit erheblich zur Weiterentwicklung des Compilerbaus bei. Politisch aktiv.

Alonzo Church * 1903 † 1995
: Der Logiker beschrieb das λ-Kalkül, zeigte dessen Äquivalenz zu Turingmaschinen (siehe „Church-Turing-These“) und schuf weitere Beiträge im Bereich der Logik und Philosophie.

Edgar F. Codd * 1923 † 2003
Er gilt als der Erfinder relationaler Datenbanken und prägte deren Theorie wesentlich mit. Da relationale Datenbanken noch heute die meist genutzten sind, findet sein Schaffen heute noch Anwendung.

Edsger Dijkstra * 1930 † 2002
Wesentliche Beiträge im Bereiche der Algorithmen (Graphentheorie, Ressourcenverwaltung). Prägte den Begriff „strukturierte Programmierung“ und half Programmierung zugänglich zu machen. Seine Notizen („EWDs“) sind heute noch wichtigstes Referenzmaterial.

Kurt Gödel * 1906 † 1978
Logiker des 20. Jahrhunderts welcher fundamentale Beiträge zur modernen Logik leistete (Modallogik, Prädikatenlogik). Er bewies seine Vollständigkeits- und Unvollständigkeitssätze.

Grace Hopper * 1906 † 1992
Als eine der Pioniere der Informatik entwickelte sie den ersten Compiler, verbreitete den Term „Debugging“ und half COBOL zu entwickeln.

Alan Kay * 1940
Mit seinen Arbeiten an graphischen Oberflächen und den daraus entstandenem objektorientierten Programmierparadigma trug er zur Entwicklung des Personal Computer bei.

Donald Knuth * 1938 Autor des Textsatzprogramms TEX. Bekannt für seine Beiträge im Bereich der Algorithmen (zB Knuth–Morris–Pratt Algorithmus). Arbeitet perfektionistisch seit 1962 an seiner Bücherserie „The Art of Computer Programming“.

Gottfried Leibniz * 1646 † 1716
Aufgrund seiner Betrachtung des binären Zahlensystems wird er als Mathematiker mit der Informatik in Verbindung gebracht.

J.C.R. Licklider * 1915 † 1990
Wesentliche Beiträge in den frühen Zeiten der künstlichen Intelligenz. Weitere Forschung auch in den Bereichen Mensch-Maschine-Interaktion.

Ada Lovelace * 1815 † 1852
Ihre Beiträge zur Analytical Engine und der ersten Formulierung eines Algorithmus' in einer Programmiersprache für eine automatisierbaren Maschine brachten ihr den Titel „erste ProgrammierIn der Welt“ ein.

John McCarthy * 1927 † 2011
Durch die Schaffung der Sprachfamilie LISP ermutigte er zahlreiche Entwickler für die Artificial Intelligence, die er wesentlich mitgeprägt hatte und einen Hype in den 60/70er Jahren auslöste.

Ted Nelson * 1937
Ted Nelson trägt als Technikphilosoph zur Informationstechnologie bei und begleitete etwa die Entwicklung des Hypertexts von Tim Berners-Lee mit und beobachtet Entwicklungen im Internet.

John von Neumann * 1903 † 1957
Mathematiker in den Bereichen Logik, Spieltheorie und Analysis. Er trug durch seine Von-Neumann-Architektur (Daten und Programme liegen beide binär codiert auf einem persistenten Speicher) zur Entwicklung der frühen technischen Informatik bei.

Dennis Ritchie * 1941 † 2011
Gemeinsam mit Ken Thompson entwickelte er das UNIX-System, welches Grundlage für zahlreiche Betriebssysteme bildet und einen gemeinsamen Standard schaffte.

Claude Shannon * 1916 † 2001
Claude Shannon gilt mit seiner Formulierung des Begriffs der Entropie als Begründer der Informationstheorie und trug mit Grundlagenforschung zur Übertragung von Signalen wesentlich zur Informationstechnologie bei.

Richard Stallman * 1953
Mit der Schaffung des GNU-Projekts gilt er als Hauptfigur der Freie-Software-Bewegung.

Ivan Sutherland * 1938
Vorreiter und aktiver Wissenschafter im Bereich der Computergraphik (zB Cohen-Sutherland-Algorithmus).

Andrew Tanenbaum * 1944
Andrew Tanenbaum ist bekannt geworden durch Beiträge im Bereich der Betriebssysteme und Netzwerke.

Ken Thompson * 1943
Gemeinsam mit Dennis Ritchie entwickelte er das UNIX-System, welches Grundlage für zahlreiche Betriebssysteme bildet und einen gemeinsamen Standard schaffte.

Alan Turing * 1912 † 1954

Als Begründer der theoretischen Informatik formalisierte er die Berechenbarkeit für die Informatik. Mit der Turingmaschine, dem Turingtest und seiner Vision einer Artificial Intelligence legte er den Grundstein für wichtige Automatisierungsprobleme.

Niklaus Wirth * 1934

Bekannt als Mitentwickler wichtiger prozeduraler Programmiersprachen. Neben Euler, Modula-2 und Algol W ist er berühmt für das Schaffen der Programmiersprache Pascal mit welchem Generationen von Programmierern ausgebildet wurden.

Konrad Zuse * 1910 † 1995

Konrad Zuse entwickelte die Z3, welche den ersten frei programmierbaren Computer darstellt und legt damit einen Grundstein für die Architektur von Rechenmaschinen.

Heinz Zemanek * 1920

Bekannt wurde Zemanek durch den Bau des ersten Computer, der einzig aus Transistoren gebaut war. Damit leitete er eine neue Ära in der Hardwarearchitektur ein. Der Name „Mailüfterl“ des Computers war eine Reaktion auf den amerikanischen Namen „Whirlwind“.

Literaturverzeichnis

[Bal04] Helmut Balzert. *Lehrbuch Grundlagen der Informatik*. Spektrum, Heidelberg, 2. edition, 2004.

[Gö31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

Index