Informatik I und II für Ingenieure

Skript zur Vorlesung

Organisation von Berechnungsabläufen und Rechnern
Jahrgang 2002/2003
Zuletzt aktualisiert: 13.04.2003

Gerald Sommer (gs@ks.informatik.uni-kiel.de)

Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik

Vorwort

Dieses Skript hat eine Geschichte, die eng gekoppelt ist an die Geschichte der Vorlesung "Informatik für Ingenieure". Im Wintersemester 1996/97 wurde diese spezielle Vorlesung für Ingenieure neu eingeführt. Folgende Dozenten haben seitdem diese Vorlesung gehalten:

Informatik I	WS 1996/97	G. Sommer
Informatik II	SS 1997	G. Sommer
Informatik I	WS 1997/98	G. Sommer
Informatik II	SS 1998	P. Kandzia
Informatik I	WS 1998/99	G. Sommer
Informatik II	SS 1999	G. Sommer
Informatik I	WS 1999/2000	G. Sommer
Informatik II	SS 2000	G. Sommer
Informatik I	WS 2000/01	R. Koch
Informatik II	SS 2001	R. Koch
Informatik I	WS 2001/02	R. Koch
Informatik II	SS 2002	R. Koch
Informatik I	WS 2002/03	R. v. Hanxleden
Informatik II	SS 2003	G. Sommer

Wie alle gedruckten Werke von größerem Umfang war das Skript nie frei von Schreib- und orthografischen Fehlern. Wir haben stets an deren Beseitigung gearbeitet. Viel wesentlicher ist aber, dass mit der Zeit auch inhaltliche Korrekturen und Verschiebungen der Vorlesung erfolgten, die sich im Skript niederschlugen. Ich danke allen an der Vorlesung beteiligten Dozenten, wissenschaftlichen Mitarbeitern und Studenten für diese Verbesserungen. Sie tragen dazu bei, dass auch heute das vorliegende Skript lebt und stets offen ist für Korrekturen und für Vorschläge zur weiteren Optimierung des Stoffes.

Gerald Sommer April 2003

Inhaltsverzeichnis

1 Einführung

1.1 Was ist Informatik

Die Informatik ist eine im Fächerkanon wissenschaftlicher Hochschulen sehr junge Wissenschaft.

An deutschen Universitäten begann die Ausbildung von Informatikern Ende der 60er Jahre. Der Einsatz von Informatikern in der Praxis und die Anwendungen der Informatik im täglichen Leben und im Beruf haben in dieser kurzen Zeit zu einer revolutionären Dynamik geführt. So stehen mit den heutigen PCs Computer mit Leistungen zur Verfügung, die weit diejenigen von Großrechnern in den 50er Jahren übertreffen.

Der Begriff Informatik

Es ist schwer zu definieren, was das Wesen der Informatik ausmacht, im Gegensatz zu klassischen Wissenschaften (Physik, Mathematik, etc.). Der Begriff *Informatik* wurde 1968 durch den damaligen Wirtschaftsminister Stoltenberg eingeführt und stellt ein Kunstwort dar, das sich aus den Begriffen *Information* und *Mathematik* zusammensetzt.

Definition: (Informatik)

Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung, Speicherung und Übermittlung von Informationen.

Informatik als Wissenschaft

Die Informatik beschäftigt sich mit

- Struktur, Wirkungsweise, Anwendung und Konstruktionsprinzipien von Informationsverarbeitungssystemen

- Strukturen, Eigenschaften und Beschreibungsmöglichkeiten von Information und Informationsverarbeitungsprogrammen
- Möglichkeiten der Strukturierung, Formalisierung und Mathematisierung von Anwendungsgebieten
- Modellbildung und Simulation.

Ist die Informatik eine Wissenschaft?

Wesentliches Moment der Definition einer Wissenschaft ist das Verfügen über eine theoretische Basis, aus der neue Erkenntnisse abgeleitet werden. Dies ist in der Informatik vielfältig unter Beweis gestellt. Als Beispiel sei die Entwicklung von Programmiersprachen und Rechnerarchitekturen aus theoretischen Erkenntnissen über Prozesse und Systeme der Informationsverarbeitung genannt.

Informatik ist eine Strukturwissenschaft

Carl Friedrich von Weizsäcker gibt in seinem Buch "Die Einheit der Natur" eine Unterteilung der Wissenschaften in

- Naturwissenschaften (Physik, Biologie)
- Ingenieurwissenschaften (Nachrichtentechnik)
- Geisteswissenschaften (Philosophie)
- Sozialwissenschaften (Soziologie, Politologie)
- Strukturwissenschaften (Mathematik, Informatik)

an und bezeichnet die Informatik als *Strukturwissenschaft*. Zitat:

> "... sie studiert Strukturen in abstracto, unabhängig davon, welche Dinge diese Strukturen haben, ja ob es überhaupt solche Dinge gibt."

Wurzeln der Informatik:

- Mathematik, Physik
- Nachrichtentechnik
- (Kybernetik: Wissenschaft von den Beziehungen zwischen dem Verhalten von Systemen, d.h. ihren Erscheinungsformen, und ihrer Struktur)

Es hat sich eine Entwicklung der Informatik zu einer selbständigen Wissenschaft mit vier wesentlichen Elementen vollzogen:

1. die formale Spezifikation von Systemen der Informationsverarbeitung
2. die Darstellung von Systemen durch Datenstrukturen und Algorithmen
3. die automatisierte Durchführung von Transformationen von Daten, um Information zu gewinnen
4. die Synthese und Analyse von Hardware- und Softwareinstrumenten als Basis der Implementierung von Algorithmen

Ambivalenz der Informatik:

1. Die Informatik erfordert

   das analytische Herangehen der Naturwissenschaften

   und

   die formalisierenden Methoden der Mathematik

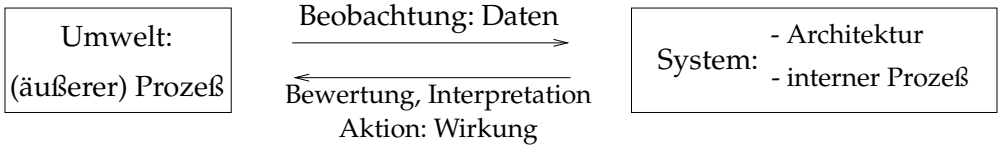
   zur Lösung eines Problems der Informationsverarbeitung.

   - Zuordnung eines konkreten Problems zu einer Problemklasse
   - Frage: Was ist Information, wie ist sie zu beschreiben und wie kann man sie aus Daten erhalten?
   - Frage: Wie sind informationsverarbeitende Systeme zu entwerfen?

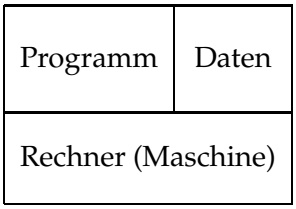
   Beispiel: Computer Vision $\sim$ Maschinensehen

   1. Wie ist die visuell wahrnehmbare Welt zu repräsentieren/strukturieren, um visuelle Wahrnehmung technisch zu realisieren?
   2. Wie sind Struktur und Dynamik von Systemen zu verstehen, um visuelle Wahrnehmung realisieren zu können?

   -



- Rechenprozeß erfordert: ⟶ Informatik beschäftigt sich mit
  - Rechnerstrukturen
  - Programmstrukturen
  - Datenstrukturen



2. Die Informatik lebt durch das konstruktive Element der Ingenieurwissenschaften: Ohne die Möglichkeit der Anwendung gäbe es die Informatik nicht!

   - Tätigkeiten eines Ingenieures [nach ENCYCLOPAEDIA BRITANNICA]:

     Die schöpferische Anwendung wissenschaftlicher Prinzipien auf den Entwurf und die Entwicklung von Strukturen, Maschinen, [...] im Hinblick auf eine gewünschte Funktion, Wirtschaftlichkeit und Sicherheit von Leben und Eigentum.

   - Wie lassen sich abstrakte Beschreibungen von Systemen informationsverarbeitender Prozesse auf konkrete Realisierungen abbilden?

     - Hardware $\sim$ "Rechner" $\rightarrow$ Rechnerarchitekturen, Computer Engineering

     - Software $\sim$ "Programme" $\rightarrow$ Software Engineering

     Auswahl aus unendlicher Zahl von Möglichkeiten unter Randbedingungen (Zwängen)

   - enge Benachbarung/Überschneidung von Informatik und Elektrotechnik
   - Gegenstand der Informatik ist vor allem anderen:
     - Analyse und (Re-)Organisation der Arbeit mit Hilfe informationstechnischer Mittel, ihre maschinelle Unterstützung oder ihre Ersetzung durch Maschinen und
     - die Entwicklung der Informationstechnik zu diesen Zwecken, insbesondere die Entwicklung des methodisch begründeten Entwurfes von Soft- und Hardware und der Integration informationstechnischer Komponenten zu Systemen.

Drei Paradigmen der Informatik:

1. Paradigma: *Theorie* (verwurzelt in Mathematik)
   Ziel: Schaffung einer kohärenten und validen Theorie nach 4 Schritten

a) Definition: charakterisiere die zu untersuchenden Objekte
b) Theorem: stelle Hypothese über mögliche Beziehungen unter diesen Objekten auf
c) Beweis: untersuche, ob diese Beziehungen wahr sind
d) Abstraktion: interpretiere die Ergebnisse

2. Paradigma: *Modellbildung* (verwurzelt in experimentellen Naturwissenschaften)
Ziel: Abstraktion abgeleitet aus der Beobachtung von Phänomenen nach 4 Schritten

a) bilde eine Hypothese
b) konstruiere ein Modell und leite eine Vorhersage ab
c) entwerfe ein Experiment und gewinne Daten
d) analysiere die Ergebnisse

3. Paradigma: *Entwurf* (verwurzelt in Ingenieurswissenschaften)
Ziel: Konstruktion eines Systems (Gerätes) nach 4 Schritten

a) benenne die Erfordernisse
b) gebe eine Spezifikation an
c) entwerfe und implementiere das System
d) teste das System

Folgerungen aus dem konstruktiven Element der Informatik:

- Die Informatik ist in bedeutendem Maße eine Ingenieurwissenschaft
- Die Informatik vermag durch ihre Ergebnisse die menschliche Gesellschaft enorm zu beeinflussen
  - Die technologische Entwicklung schreitet der gesellschaftlichen Entwicklung voran
  - Gefahr starker gesellschaftlicher Verwerfungen
    Muß das Machbare auch zu jeder Zeit realisiert werden?
- Der technologische Fortschritt treibt die interne Entwicklung der Informatik stark voran
  - Was gestern uninteressant war, weil technisch nicht realisierbar, muß heute entwickelt werden, weil morgen realisierbar

  - Softwarekrise: Die Entwicklung der Programmierkonzepte hinkt der Entwicklung der Hardware nach

- Die technische Realisierbarkeit bedeutet nicht die Durchsetzbarkeit auf dem Markt

---

Beispiel: Parallelrechner

Diktat des Faktischen: preisgünstige Universalrechner werden immer billiger, schneller, weiter verbreitet, einfacher zu bedienen/programmieren.
Die Programmierparadigmen der Parallelrechner sind noch nicht weit entwickelt: ihre Programmierung ist langwierig, teuer und kompliziert.

Gebietsgliederung der Informatik:

1. *Technische Informatik*
   - stellt die erforderlichen Gerätschaften (Hardware) bereit
   - beschäftigt sich mit der Konstruktion von Schaltwerken/Rechnern, Speicherchips, Prozessoren, Peripheriegeräten
   - sehr eng mit der Elektrotechnik verbunden
   - Hardware muß potentielle Anwendungen im Auge haben
   - gekennzeichnet durch Nutzung hardwarenaher Programmierung und Simulation der Hardware
2. *Praktische Informatik*
   - stellt im weitesten Sinne die Programme (Software) bereit
   - schlägt Brücke zwischen Hardware und Anwendungssoftware
   - beschäftigt sich mit der strukturierten Erstellung von Softwaresystemen: Informations-/Kommunikationssysteme, Betriebssysteme, Übersetzer
   - Künstliche Intelligenz: Wissensverarbeitung
   - gekennzeichnet durch hardwareferne Programmierung
3. *Theoretische Informatik*
   - stellt im weitesten Sinne die abstrakten Strukturen bereit
   - hat besonders enge Beziehung zur Algebra und Logik
   - beschäftigt sich mit: Automatentheorie, formalen Sprachen, Komplexität von Algorithmen, Berechenbarkeit von Problemen

4. *Angewandte Informatik*

   - beschäftigt sich mit dem Einsatz von Rechnern in unterschiedlichsten Anwendungsgebieten (z.B. Textverarbeitungssysteme, Sprachverarbeitung, Bildverarbeitung, Handschriftenerkennung)
   - "Bindestrich-Informatik"
     Medizinische Informatik, Wirtschaftsinformatik, Rechtsinformatik

1.2 Die Geschichte des maschinellen Rechnens

Die Geschichte des maschinellen Rechnens Die Informatik ist die Wissenschaft, die den Menschen bei der Ausführung *geistiger* und *körperlicher* Tätigkeiten unterstützt oder von ihnen befreit. Hierdurch ist ein inhärenter sozialer Sprengstoff im Wesen der Informatik begründet.

- Geistige Tätigkeiten:
  - Rechnen mit Zahlen und Symbolen
  - Bewerten von Daten und Aussagen
  - Verstehen von Text
  - Übersetzen von Text
  - Verstehen von natürlicher Sprache, Erzeugen natürlicher Sprache
  - Verstehen von Bildern, Maschinensehen
- Körperliche Tätigkeiten:
  - Regelung und Steuerung von Prozessen mittels Computer ("eingebettete Systeme")
  - Roboter
    - Industrierobotor: blind, "dumm", d.h. operieren nur unter vordefinierten Bedingungen
    - autonome Systeme: visuelle, sensorische Wahrnehmung, lernfähig, unter weniger eingeschränkten Bedingungen einsetzbar

Die Informatik hat ihre Wurzeln im Bedürfnis, geistige Tätigkeiten zu mechanisieren.

1.2.1 Das Rechnen mit Ziffern

Ziffern stellen Zahlzeichen eines Zahlwortes dar, z.B. "neun": IX oder 9 Die römischen Zahlen wurden nach einem *Additionssystem* gebildet,
z.B. 1996 $\sim$ MDCCCCLXXXVI oder MCMXCVI.
Die Zahlen wurden mit Zahlsteinchen, den "calculi" gelegt, hiervon stammt der Begriff "Kalkül" ab. Die Ziffern von Eins bis Neun sowie die Null als Zeichen für leere Stellen wurden durch die Inder (ca. 800 n.Chr.) eingeführt. Sie "erfanden" das Dezimalsystem, ein *Stellenwertsystem* zur Basis 10. Erst diese Einführung des Stellenwertsystemes der arabischen Ziffern erlaubte die Mechanisierung des Rechnens mit Ziffern.
Eine Zahl im Dezimalsystem besitzt die Darstellung

$$z_{10} = \sum_{i=0}^{n-1} \alpha_i 10^i,\ \alpha_i \in \{0, 1, \ldots, 9\}.$$

Z.B. $1996_{10} = 1 \cdot 10^3 + 9 \cdot 10^2 + 9 \cdot 10^1 + 6 \cdot 10^0$. Auch heute existieren im täglichen Leben noch weitere Zahlensysteme, z.B:
$z_{12} \sim$ Dutzend, $z_{60} \sim$ zur Zeiteinteilung (von den Babyloniern). Von Gottfried Wilhelm Leibniz (1679) stammte der Entwurf einer dual arbeitenden Rechenmaschine mit einem Stellenwertsystem zur Basis 2:

$$z_2 = \sum_{i=0}^{n-1} \alpha_i 2^i, \qquad \alpha_i \in \{0, 1\}.$$

Erst in diesem Jahrhundert erlangt das Dualsystem seine Bedeutung zum Rechnen: Konrad Zuse entwarf 1933 eine durch Relais gesteuerte Rechenmaschine. Technische Realisierung des Dualsystems: 1/0-Codierung über Schalter, Relais, Röhren, Transistoren:

1: Spannung/Strom
0: keine Spannung/Strom

Hierdurch ist der Aufbau komplexer Rechenwerke für arithmetische Grundoperationen ermöglicht, deren effiziente Ausführung durch logische Schaltkreise erfolgt.

1.2.2 Das Rechnen mit Symbolen

Das "Buchstabenrechnen" wurde von der indischen Mathematik im frühen Mittelalter eingeführt. Hieraus leitet sich die algorithmische Lösung arithmetischer Aufgaben ab, z.B. die Division (Adam Ries, 1492-1559). Damit kann eine erste, noch unscharfe Definition des Begriffes Algorithmus gegeben werden:

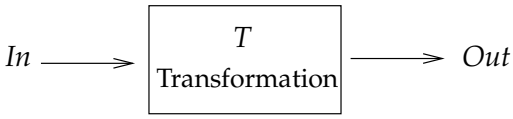
> Algorithmus: Umformung von gegebenen Größen auf Grund eines Systems von Regeln in andere Größen.

Z.B.: $(a + b)(a - b) = a^2 - b^2$ (Anwendung der Regeln der Algebra) Das Rechnen mit Ziffern und mit Symbolen ist auf einer allgemeineren Ebene als gleichwertig anzusehen:

1. Das klassische Rechnen kann als Manipulation von Zeichenketten aufgefaßt werden.

2. Jede Manipulation von Zeichenketten nennt man *Rechnen* (Begriffserweiterung gegenüber der Umgangssprache!)

   Damit versteht man unter Rechnen eine regelhafte (algorithmische) Abbildung einer Eingangszeichenkette auf eine Ausgangszeichenkette:

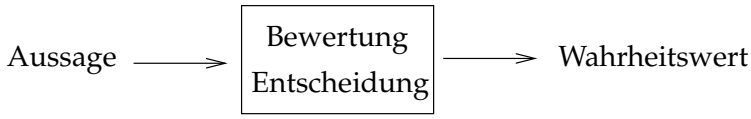


Unter Umständen kann für diese Abbildung (Transformation) ein geschlossener arithmetischer Ausdruck angegeben werden, generell ist sie aber durch eine abstraktere mathematische Funktion beschreibbar. Beispiele für das Rechnen mit Symbolen:

- 1963 Programmsystem zur Formelmanipulation
- 1950 erster Textmanipulator: Editieren von Text
  1960 automatische Silbentrennung
- mechanische Sprachübersetzung (erste Ansätze 1933 Smirnov-Troyanski), machte Entwicklungen auf dem Gebiet der Linguistik erforderlich. Die Berücksichtigung des Kontextes ist notwendig:
  "HUND" → "DOG" aber "HUNDERT" → "HUNDRED"
- Schachspiel (N.Wiener, 1948; Shannon, 1950: grundlegende Theorie)
- Mathematische Beweise.

1.2.3 Logisches Rechnen

Die beim Rechnen mit Symbolen angegebene Transformation T kann auch die Zuordnung eines Wahrheitswertes $Out \in \{\text{true, false}\}$ zu einer Zeichenkette *In* erzeugen, wenn *In* eine Aussage darstellt:



> Dabei sind *Aussagen* sprachliche Gebilde (Sätze), die zur Beschreibung und Mitteilung von Sachverhalten dienen, welche als wahr oder falsch bewertet werden können.

Es gilt das Prinzip der Zweiwertigkeit einer Aussage. Die *Aussagenlogik* beschäftigt sich mit der Bewertung komplexer Aussagen.

- Leibniz (1672-76): Ansätze für ein logisches Kalkül
- Boole (1847): Algebraisierung der Logik

- Shannon (1938): erkannte Parallelen in der Arbeitsweise von Relaisschaltungen und Aussagenkalkül → Entwicklung einer Schalttheorie, z.B. systematisches Vereinfachen von Schaltfunktionen
- Abbot (1951): Bau einer digitalen Rechenanlage zur Durchführung logischer Verknüpfungen
- Frege (1879): Begründung der modernen Logik

Die *Prädikatenlogik* stellt eine Verallgemeinerung der Aussagenlogik dar:
Sie ermöglicht die Einführung einer symbolischen Sprache für die Beschreibung von Algorithmen und ihren Objekten. Die Geschichte der Algorithmen ist die Geschichte des logischen Rechnens mit Symbolen! Heutiger Stand der Entwicklung:
Die Interpretation/Übersetzung von Anwenderprogrammen erfolgt nach logischen Regeln, die ihre Entsprechung in der Funktion der Schaltkreise des Rechners haben. Es existiert eine 1:1-Abbildung der Software auf die Hardware.

1.2.4 Das Rechnen mit Signalen

Die numerischen, symbolischen und logischen Berechnungsaufgaben sind äquivalent insofern, als daß sie sich auf diskrete Zeichen (Zahlen, Buchstaben, Operationssymbole) beziehen. *Symbole* sind Objekte des menschlichen Geistes, sie sind Modellierungen von Phänomenen der Natur (z.B. interpretierte Signale). *Signale* sind Phänomene der Natur. Sie haben unendlich viele Interpretationsmöglichkeiten. Welche hiervon gewählt wird, hängt von der Intention des Beobachters ab. Albert Einstein zur Crux der Modellbildung:

> "Insofern sich die Sätze der Mathematik auf die Wirklichkeit beziehen, sind sie nicht sicher, und insofern sie sicher sind, beziehen sie sich nicht auf die Wirklichkeit."

Für die Symbolverarbeitung kann man von folgenden Annahmen ausgehen:

1. Symbole sind Abstraktionen
   Sie sind "sauber", d.h. nicht gestört. Sie geben die Wirklichkeit zwar nur näherungsweise wieder, enthalten aber das für die Aufgabe Wesentliche.
2. Symbole sind endlich, sie erfordern einen endlichen (vielleicht sogar minimalen) Aufwand zu ihrer Repräsentation und Beschreibung.
3. Typische Berechnungsaufgaben sind von begrenzter Komplexität, sie führen in endlicher Zeit zum gewünschten Ergebnis. Das trifft bei weitem nicht auf alle Probleme zu!

4. Typische Berechnungsaufgaben sind deterministisch, in der Auswahl der Bearbeitungsschritte besteht keine Freiheit.

Prinzipiell sollte man für die Signalverarbeitung von folgenden Annahmen ausgehen:

1. Signale sind die Wirklichkeit, sie enthalten relevante und nicht relevante Komponenten. Sie enthalten Störungen, die mit dem relevanten Signalanteil untrennbar verbunden sind. Signale sind "schmutzig".

2. Signale sind kontinuierlich, sie erfordern im Prinzip einen unendlichen Aufwand zur Repräsentation oder Beschreibung. Die Repräsentierung von Signalen im Rechner erfolgt in der Form von digitalen, d.h. diskretisierten und quantisierten Signalen.

3. Es gibt wesentliche Aufgabenklassen, die von sehr hoher Komplexität sind. Sie können mit deterministischen Algorithmen nur mit sehr hohem Aufwand (in sehr großer Zeit) oder überhaupt nicht exakt gelöst werden.

4. Typische Berechnungsaufgaben sind stochastisch (nicht deterministisch), in der Auswahl der Bearbeitungsschritte werden Freiheiten zugelassen (z.B. Heuristiken, Zufallsprozesse zur Steuerung).

Mit dem Rechnen mit Signalen beschäftigen sich folgende relativ junge Disziplinen:

- Signaltheorie (ca. 1945)
- Nachrichtentheorie (ca. 1945)
- Mustererkennung (ca. 1960)
- Neuroinformatik (ca. 1980)

Zur Erfassung des Zusammenhangs zwischen einem gestörten Signal und einem Symbol werden die Theorie stochastischer Prozesse (entwickelt aus Wahrscheinlichkeitstheorie), die Informationstheorie und Entscheidungstheorie herangezogen. Das Problem, wie auf den symbolischen Gehalt von gestörten Signalen geschlossen werden kann, ist nur teilweise gelöst.

Beispiel:

- numerischer Code des Buchstaben H: 01001000_2 (ASCII-Code)
- gedruckter Buchstabe H: nur Störungen der Bilderzeugung

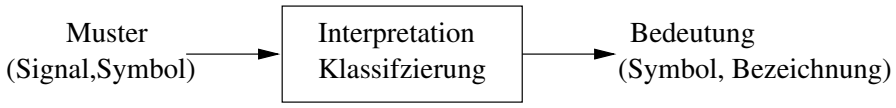
a	b	c
0 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 0 0 0	0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0	0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 0
korrekt abgetastet, keine Störungen	zu grob abgetastet, verrauscht	zusätzlich um 30 ° geneigt

- handgeschriebener Buchstabe H: zusätzlich auch Individualität des Schreibers

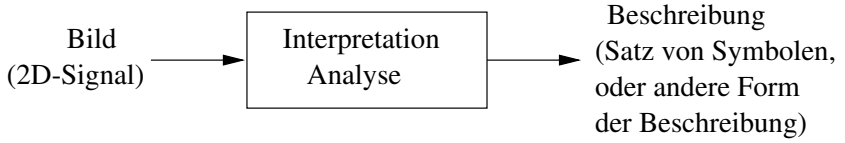
Frage: Wie kann aus dem Signalmuster das Symbol des Buchstaben H erkannt werden? Das Erkennen von maschinengeschriebener Schrift (OCR) wird beherrscht (Reduktion auf klassische Problemlösungen). Das Erkennen handgeschriebener Schrift jedoch nicht, wahrscheinlich ist eine Erweiterung klassischer Problemlösungen nötig.

Beispiele für das Rechnen mit Signalen sind

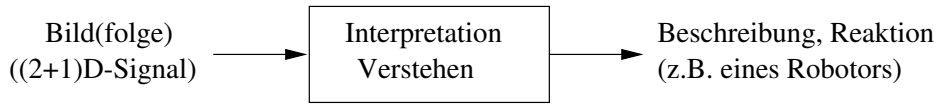
- *Mustererkennung* (Pattern Recognition, ca. 1960)
  Als *Muster* wird eine durch eine Menge von Messungen oder Beobachtungen erzeugte Struktur bezeichnet.



- *Bildverarbeitung* (Image Processing, ca. 1965)
  Unter einem *Bild* wird eine matrixförmige Anordnung von Helligkeitswerten verstanden.



- *Maschinensehen* (Computer Vision, ca. 1980)
  Eine *Szene* (durch Kamera beobachteter Weltausschnitt), gegeben durch ein Einzelbild oder eine Bildfolge, soll interpretiert werden, um Beschreibungen oder Reaktionen abzuleiten.



  Die Interpretation oder das Verstehen einer Bildfolge ist nicht als lineare Folge von Berechnungsprozessen realisierbar. Es stellt sich die Frage, ob sich die visuelle Wahrnehmung des Menschen technisch realisieren läßt. Das Symbolverarbeitungsparadigma der Kognitionswissenschaft führt nicht zur Lösung. Zu vielen Problemen der Wahrnehmung können keine Algorithmen explizit angegeben werden.
  In der *Neuroinformatik* sind in Form von neuronalen Netzen Ansätze zu Berechnungsverfahren begründet, die auf explizite Formulierungen von Algorithmen verzichten.

1.2.5 Die Entwicklungsgeschichte der Rechenmaschine

Die bis zu unseren heutigen Computern führende Entwicklung von Rechenmaschinen verlief in 3 Etappen:

1. 17. Jahrhundert: Mechanisierung des Rechnens mittels mechanischer Rechenwerke → einfache Ausführung der Grundrechenarten
2. 19. Jahrhundert: Automatisierung des Rechnens mittels Daten- oder Programmspeicher und Steuerwerk → komplexere Berechnungsabläufe
3. ca. 1940/1950: Einführung der frei programmierbaren elektronischen Universalrechenmaschine → Entwicklung bis zum heutigen Computer über 5 Generationen

Im folgenden werden einige Stationen der 3 Etappen skizziert:

- Leibniz(1673): 12-Dekaden-Rechenmaschine
  Schnelle Multiplikation mit Hilfe von Zehnerpotenzen, stufenweise verschiebbare Zahnradwalze (Staffelwalze), dieses Prinzip wurde von fast allen mechanischen Rechenmaschinen übernommen.
- Charles Babbage (1843): analytical engine
  Bahnbrechendes Konzept eines universellen Rechners mit

  - Rechenwerk
  - Steuereinheit für Iteration und bedingte Verzweigung
  - Zahlenspeicher
  - E/A über Lochkarten-Band nach Jacquard (1805)

  zur Berechnung ballistischer Tafeln für die britische Marine. Die analytical engine wurde jedoch nicht fertiggestellt, da nicht die nötige Präzision zur Fertigung der Zahnräder erreicht werden konnte.

- Hermann Hollerith (1886): Zähl- und Registriermaschine auf der Basis von Lochkarten zur Auswertung der amerikanischen Volkszählung 1890. Sie ermöglichte verschlüsselte Angaben von Personen auf Lochkarten und verfügte über eine elektrische Abtastapparatur, die über Fühler die Kodierung an magnetische Zählwerke weitergab.
  Hollerith war der Gründer der Firma International Business Machines (IBM).

- Konrad Zuse (Berliner Bauingenieur) war der Erfinder der ersten elektronischen Rechenmaschine.

  - Z1/Z2 (ca. 1938): Entwurf eines mechanischen Rechners. Die Realisierung der mechanischen Teile gelang nicht. Das Konzept war ähnlich der analytical engine von Babbage, es verwendete das Dualzahlsystem, realisierte Gleitkommazahlen und logische Operationen (Und, Oder, Negation)
  - Z3 (1941): erster frei programmierbarer, elektronischer, durch Relais gesteuerter Rechner
    Es wurden 2600 Fernmelderelais für das Rechenwerk und den Speicher verwendet. Die Z3 verfügte über 64 Speicherplätze für Zahlen, 22-stellige Dual- und 7-stellige Dezimalzahlen und beherrschte die 4 Grundrechenarten sowie das Radizieren. Es konnten 15-20 Additionsoperationen pro Sekunden durchgeführt werden, eine Multiplikation beanspruchte jedoch 4 Sekunden. Abbildung 1.1 skizziert den Rechenablauf.

- COLOSSUS (40er Jahre): erster Röhrenrechner
  Dechiffriermaschine unter Mitarbeit von Alan Turing, der in den 30er Jahren mit seiner Turing-Maschine ein Gedankenexperiment für einen maschinell deutbaren Algorithmenbegriff schuf.

- ENIAC (I.P.Eckert, J.W. Mauchly, 1946):

  - 18000 Röhren, 1500 Relais, 2000 mal schneller als elektromechanische Relais-Rechner
  - Additionen: 0.2 ms, Multiplikation zweier zehnstelliger Zahlen: 2.8 ms

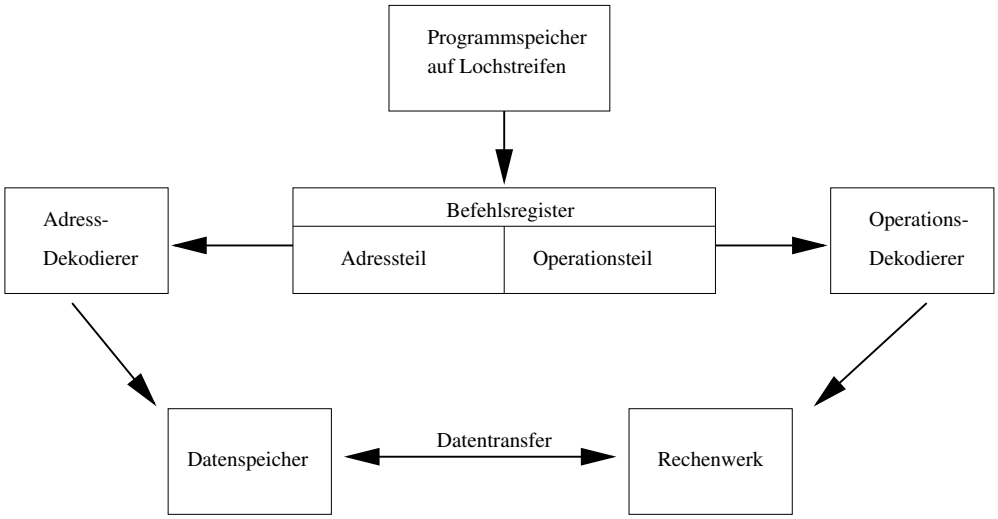


Abbildung 1.1: Rechenablauf der Z3.

  - Programmierung über Steckbretter
  - kein Programmspeicher

  ENIAC bildete den Ausgangspunkt des Rechnerbaus im Rahmen des Manhatten-Projekts.

- EDVAC (John von Neumann): Mit seinem ersten frei programmierbaren Allgemeinrechner setzt von Neumann den Beginn der modernen Rechnerarchitektur: "von-Neumann-Rechner"

  - binäre Kodierung
  - datenabhängige Programmverzweigung
  - sequentielle Verarbeitung
  - einheitlicher interner Daten- und Programmspeicher. Speicherung des Programms im internen Maschinenspeicher (zuvor wurden die Programme während der Ausführung eingegeben)
  - Realisierung datenabhängiger Programmschleifen ("While-Schleifen").

  Der Architektur liegt ein mathematisches Modell einer Berechnung (Turing-Maschine) zugrunde. Es wurde eine begriffliche Trennung zwischen Schaltlogikfunktionen und technischer Realisierung getroffen (McCulloch/Pitts-Zelle).

1.2.6 Die Generationen der elektronischen Rechenmaschine

Nach der Entwicklung von ENIAC und EDVAC setzte eine stürmische Entwicklung weiterer Unikate ein, die erst Anfang der 50er Jahre zu einer Standardisierung führte, durch die beginnende Kommerzialisierung durch IBM, UNIVAC u.a. Jede Generation ist durch typische Entwicklungsstufen von Software und Hardware charakterisiert.

1. Generation: 1953-1958
   - etwa 250 Anlagen
   - Vakuumröhren, handverdrahtet, Externspeicher: Magnetbänder, Magnettrommel
   - Zugriffszeiten 10^{-3} sec
   - Betriebssystem u. Compiler gab es nicht
   - Programmierung in Maschinensprache
2. Generation: 1958-1966
   - Transistoren und magnetische Kernspeicher
   - Erstellung von Systemsoftware, Compiler, E/A-Hilfen
     am Ende der Periode:
     - universelle Betriebssysteme
     - Compiler für Cobol, Algol, Fortran
     - Zugriffszeiten bis 10^{-6} sec
     - verdrahtet
   - erste wissenschaftliche Rechnungen, vermehrt Echtzeitinstallationen
   - Stapelbetrieb
   - Berufsbild des Informatikers entsteht
3. Generation: 1966-1974
   - integrierte Schaltungen: drastisch sinkende Hardwarepreise, Miniaturisierung, Zugriffszeiten bis 10^{-9} sec
   - Halbleiterspeicher, Bildschirm- und Dialogstationen.
     Am Ende der Periode: Mikroschaltkreistechnologie
   - sehr große und komplexe Rechenanlagen (Großrechner). "Softwarekrise" $\rightarrow$ Beginn verbesserter (systematischer) Softwareentwicklungsmethoden.
   - stürmische Entwicklung auf dem Gebiet der Echtzeitrechner, Prozeßrechner zur Steuerung werden immer kleiner (Kleinrechner). Echtzeit-Betriebssysteme

- Beginn einer Dezentralisierung der Datenverarbeitungsaufgaben

4. Generation: 1974-1982

   - Miniaturisierung der Schaltkreise in der Mikroelektronik (VLSI-Technologie, $> 10^5$ Gatter pro Chip). Mikrorechner: alle Einheiten eines Rechners auf einem Schaltkreis zusammengefaßt.
   - Schaltkreise für CPU: 130 000 Transistoren
   - 64KBit-Speicher auf einem Schaltkreis
   - Superrechner, Personalcomputer, Homecomputer
   - Konsolidierung: Hardware- und Softwaresysteme wachsen zu einer Einheit zusammen
   - teilweise Überwindung der Softwarekrise

5. Generation: seit 1982

   - rasante Miniaturisierung der VLSI-Technologie, Beispiel Speicher:
     1980: 64KBit RAM
     1983: 256KBit RAM
     1986: 1MBit RAM
     1992: 16MBit RAM
     2000: 128MBit RAM, 0.18 μ Technologie
   - rasante Zunahme der Rechengeschwindigkeit (Taktfrequenzen bis 1 GHz), Abnahme der Volumina, Zunahme der internen Speicher
   - rasante Entwicklung massiv paralleler Rechner
   - im Dreijahreszyklus (Moore's Law): Verdopplung der Taktfrequenz, Vervierfachung der Zahl der Transistoren pro Chip
     im Jahreszyklus: Verdopplung der Prozessorleistung
     seit 90er Jahren: Vervierfachung des Integrationsgrades im Zweijahreszyklus
   - generelle Informatik-Durchdringung aller Lebensbereiche

1.2.7 Ein Resumé: Wohin geht die Informatik?

1. "Quantensprünge" in der Hardware

   - Weitere Integration von Speicher- und Prozessor-Chips bis an die physikalischen Grenzen (bereits erreicht auf mikroelektronischem Weg: 100 Millionen Transistoren pro cm^2)
   - Weiteres Ausreifen des Parallelverarbeitungs-Paradigmas

- Photonik (Photonen als Informationsträger)
  - optische Prozessoren (z.Zt. 1000 Schaltelemente pro cm^2)
  - Vernetzung über Glasfaser
    - ▷ Bandbreite einer Glasfaser: 50 Terahertz (Billionen Schwingungen pro Sekunde)
    - ▷ Übermittlung durch eine Glasfaser: 300 Billionen Bit/s (theoretisch, bisher erreicht: 400 Gigabit/s)
    - ▷ Bandbreite in der Nachrichtentechnik z.Zt. 50 Gigahertz
  - erste optische Computer in ca. 10 Jahren

2. Nano-Technologie
   Integration informationsverarbeitender und mechatronischer Systeme auf dem Chip (Micro-Motoren, Micro-Spiegel Projektoren)

3. Globale und lokale Vernetzung
   Die gegenwärtige Internet-Euphorie ist nur ein Schatten künftiger Entwicklung.

4. Fortschreitende Theorienbildung/Mathematisierung
   - Automatisierung von Softwareentwurf, -entwicklung und -prüfung
   - Informationsverarbeitende Modelle für Wahrnehmungs- und kognitive Leistungen
   - Softwareentwicklung für Parallelverarbeitung

5. neue Paradigmen
   - Entwicklung von Künstlicher Intelligenz und Neuroinformatik
   - Quanten-Computer

6. Innovative Anwendungen und fortschreitende Computerisierung der "technologischen Zivilisation"
   - Informatik-Wissen gehört zu beliebigen anderen Disziplinen (wie Mathematik und Beherrschung einer Schriftsprache)
   - "Eingebettete" Computer, dediziert für bestimmte Anwedungen, mit Echtzeitanforderungen, gewinnen an Bedeutung gegenüber "klassischen" Computer für den universellen Einsatz
   - gesellschaftliche Konsequenzen müssen auch gesellschaftlich gesteuert werden

2 Von der Nachricht zur Information

Wir haben die Informatik als eine Wissenschaft eingeführt, die sich mit der maschinellen Verarbeitung, Speicherung und Übertragung von Informationen befaßt. In der Vorlesung interessieren die Prinzipien, wie Maschinen zur Informationsverarbeitung befähigt werden.

In diesem Abschnitt soll das Informationsverarbeitungsparadigma erläutert werden. Hierzu werden sowohl die erforderlichen Termini exakt definiert als auch die konzeptionellen Grundlagen der Informatik erläutert.

2.1 Systeme und Modelle

Der Begriff System stellt ein in den verschiedensten Wissenschaften bewährtes Strukturierungskonzept für die reale Welt dar. Er gestattet, einen Teil der realen Welt strukturell oder funktionell von seiner Umwelt zu unterscheiden. Insbesondere gestattet er, Computer (und ihre Komponenten), Programme und die zu lösenden Probleme zu beschreiben.

Definition: (System)

Ein System ist eine konzeptionell zusammengehörende, räumlich und zeitlich begrenzte Einheit, die aus einer Vielzahl von Komponenten (Elementen) besteht, welche miteinander in Beziehung stehen.

Ein System wird bestimmt

- durch die Eigenschaften der Komponenten
- deren Beziehungen untereinander und
- die Art und Weise der Einbettung in die Umgebung bzw. die Spezifik der Systemgrenzen.

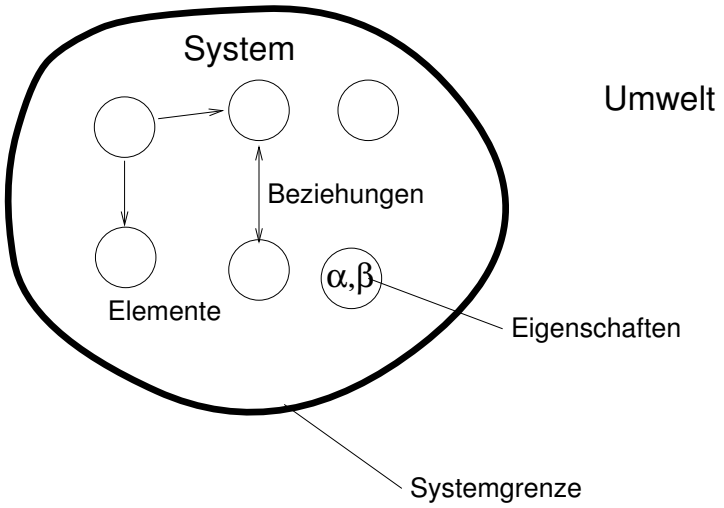


Abbildung 2.1: Ein System mit 6 Komponenten

Ein System ist eine erkenntnistheoretische Kategorie. Insbesondere kann es mehr oder weniger detailliert definiert sein. In der Konzentration auf die für eine Fragestellung wesentlichen Aspekte werden sowohl die betrachteten Komponenten (und ihre Wechselwirkungen) als auch die Systemgrenzen unterschiedlich gewählt.

Hierdurch kommen etwas unterschiedliche Definitionen von Systemen in den Wissenschaften zustande.

Natürliche Systeme (als Teil der realen Welt) sind im allgemeinen so komplexe Gebilde, daß deren vollständige Beschreibung unmöglich ist. Deshalb meinen wir streng genommen ein Modell eines Systems, wenn wir den Begriff verwenden. Wir nennen es auch künstliches System.

Definition: (Modell)

Ein Modell ist eine durch Abstraktion und Konzentration auf das für einen speziellen Sachverhalt Wesentliche gewonnenene Formulierung eines künstlichen Systems.

Die Gestaltung und der Entwurf von Modellen der Informationsverarbeitung ist die eigentlich schwierige Aufgabe der Informatik. Die Konzentration auf lediglich den technischen Umgang mit Rechnern und deren Programmierung stellt eine einschränkende Blickweise dar, die lediglich die Realisierung eines bereits gefundenen Verarbeitungsmodells bedeutet.

Wir werden im Kapitel 3, aber auch später immer wieder auf diesen Sachverhalt zurückkommen.

Beispiel: OCR-System

Die Erkennung von Buchstaben erfordert den Vergleich realer Buchstaben mit einem Modell der Buchstaben. Dabei sind die realen Buchstaben gestört, sie weichen vom "sauberen" Buchstaben-Modell ab. Das Modell ist entweder die Äquivalenzklasse der möglichen realen Buchstaben oder ein Prototyp.

In der Informatik stellt der Begriff des Zustands den Zugang zu Systemen dar.

Definition: (Zustand)

Der Zustand eines Systems wird durch die Gesamtheit seiner Komponenten, deren Eigenschaften (Merkmale, Parameter) und ihre Ausprägungen (Werte) sowie durch die Beziehungen der Komponenten bestimmt.

Beschreibbare oder beobachtbare Änderungen eines Systems beziehen sich sämtlich auf seinen Zustand. Die Folge der von einem System in einem Zeitintervall eingenommenen Zustände bezeichnet man als *Verhalten* des Systems.

Die Abarbeitung eines Programms durch einen Computer erfolgt durch regelhafte Zustandsänderungen seiner Komponenten.

Definition: (Zustandsvariable)

Die für eine Systembeschreibung bedeutend erachteten Freiheitsgrade werden als Zustandsvariable bezeichnet. In der Informatik interessieren vorwiegend diskrete Zustandsvariable.

Folglich wird der Zustand z eines Systems durch die Zustandsvariablen $z_i, i = 1, \ldots, N$, vollständig beschrieben. Man bezeichnet $z = (z_1, \ldots, z_N)$ als *Zustandsvektor*. Jede Zustandsvariable z_i nehme die Werte $\alpha_{i_j} \in \mathbb{N}$, $i_j = 1, \ldots, n_i$, an. Diese Werte sind die Zustände (mögliche Belegungen) der Zustandsvariablen z_i.

Als *Kardinalität* der Zustandsvariable z_i bezeichnet man $n_i = |z_i|$.

Definition: (Zustandsraum)

Der Zustandsraum ist die Menge aller möglichen Zustände eines Systems.

Der Zustandsraum ZR ergibt sich demnach als direktes Produkt der Zustandsvariablen, $ZR = z_1 \times \ldots \times z_N$.
Ein Zustand $z \in ZR$ wird also durch einen Punkt im Zustandsraum dargestellt. Mit N

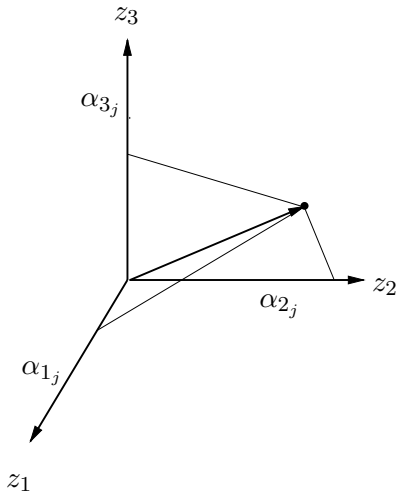


Abbildung 2.2: Ein Punkt im dreidimensionalen Zustandsraum

wird die *Dimension* und mit $V = |ZR| = n_1 \cdot \ldots \cdot n_N$ das *Volumen* des Zustandsraums bezeichnet. Insbesondere enthält der Zustandsraum den Anfangszustand und einen möglichen Endzustand des Systems.

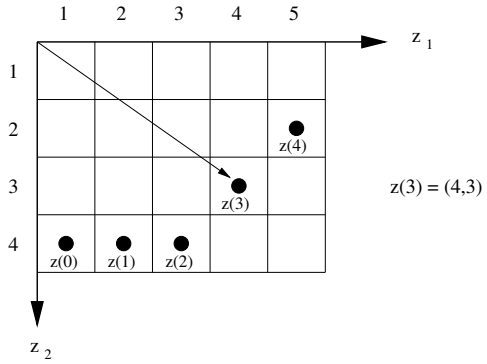
Definition: (Trajektorie)

Eine zusammenhängende Folge von Zuständen $\{z(t_0), z(t_1), \ldots, z(t_m)\}$ mit $t_0 < t_1 < \ldots < t_m$, $t_i \in \mathbb{N}$, heißt Trajektorie des Systems.

Die Trajektorie eines Systems beschreibt seine Zustandsänderungen (das Verhalten) als Weg im Zustandsraum. Der Anfangszustand wird mit $z(0)$ bezeichnet.

Beispiel:

Zustandsraum mit $N = 2, n_1 = 5, n_2 = 4$. Jede Zelle stellt einen möglichen Zustand dar. Es wird eine Trajektorie von $z(0)$ nach $z(4)$ beschrieben.



Ein wichtiges Konzept der Informatik entsteht dadurch, daß die Beachtung der Komponenten eines Systems und deren Beziehungen unterbleibt.

Definition: (Objekt)

Ein Objekt ist eine identifizierbare Einheit mit zeitlich veränderlichem Zustand ohne Berücksichtigung seiner Bausteine und deren Beziehungen.

Das Objektkonzept ist insbesondere dann von Bedeutung, wenn Systeme bezüglich ihrer Interaktionen betrachtet werden (Abb. 2.3).

Ein aktuelles Programmierparadigma, die *objektorientierte Programmierung*, trug dazu bei, das Schreiben und Lesen von Programmen wesentlich zu verbessern und die Modularisierung zu standardisieren.

- Das Systemkonzept ist rekursiv, d.h. die Komponenten eines Systems können
  1. selbst wieder als System betrachtet werden (wenn für das Verständnis des Gesamtsystems die Details der Teilsysteme von Interesse sind) oder
  2. selbst wieder als Objekt betrachtet werden (wenn nur die Zustände der Teilsysteme als Ganzes von Interesse sind).

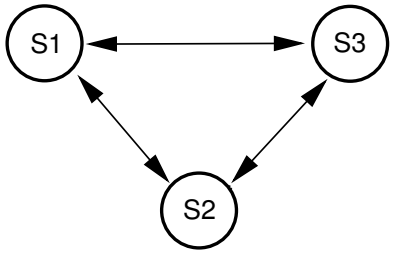


Abbildung 2.3: Drei Systeme interagieren miteinander

Das heißt, auch der Begriff *Umgebung eines Systems* ist relativ und hängt von der Hierarchieebene der Systemdefinition ab. Generell gilt aber, daß Systeme in ihre Umgebung eingebettet sind.

- Die Rekursivität besteht nur konzeptionell. Oft wird hieraus aber eine Methode der Systemmodellierung abgeleitet, die man Skalierbarkeit nennt.

Skalierbarkeit ist die Umkehrung des Prinzips der Teilbarkeit.

Definition: (Descartesches Prinzip "Teile und Herrsche")

1. Zerlege ein Problem in eine Menge von Teilproblemen.
2. Löse separat alle Teilprobleme.
3. Rekonstruiere hieraus die Lösung des Gesamtproblems.

Definition: (Skalierbarkeit)

Skalierbarkeit bedeutet, daß die Systemmodellierung im Kleinen auf größere Systeme übertragbar ist.

Dies gilt nur eingeschränkt, denn

1. es werden bei der Zerlegung in Teilsysteme deren Beziehungen vernachlässigt,
2. es wird angenommen, daß größere Systeme sich nur durch ein quantitatives Maß von kleineren Systemen unterscheiden (und somit eine Übertragung der Konzepte erlauben). Beim Übergang vom kleineren zum größeren System dürfen sich also keine qualitativen Unterschiede ergeben.

Beispiele:

1. Meinungsforschung: Extrapolation von einer kleinen Gruppe Befragter auf den wahlberechtigten Bevölkerungsanteil
2. Windkanalexperimente

   aber auch

3. Übergang von der makroskopischen zur mikroskopischen zur atomaren und zur Elementarteilchenphysik:
   In jedem Bereich wirken andere Kräfte, gilt eine andere Physik. Dies bedeutet, daß die Makrophysik nicht aus der Elementarteilchenphysik erklärbar ist (und umgekehrt).

Die Annahme der Skalierbarkeit erwies sich in der naturwissenschaftlich-technischen Gesellschaft oft als ein fruchtbarer Ansatz. Zunehmend werden wir aber auch mit katastrophalen Folgen dieses Herangehens konfrontiert (z.B. das Versagen eines elektronischen Stellwerks in Hamburg-Altona oder der Gepäckbeförderung am neuen Flughafen Denver).

Vielmehr gilt als Widerspruch zur Descarteschen Annahme:

"Das Ganze ist mehr als die Summe seiner Teile"

Die Modellierung unter dieser Grundannahme befindet sich in den Anfängen. Sie erfordert die Behandlung der Systeme als *nichtlineare dynamische Systeme*. In diesem Rahmen werden auch qualitative Zustandsänderungen von Systemen zugelassen. Hierfür steht der Begriff Emergenz.

Definition: (Emergenz)

Unter Emergenz versteht man das Auftreten neuer Qualitäten in der Zustandsänderung eines Systems.

Das heißt, aus der Wechselwirkung einer Vielzahl von Systemkomponenten ergibt sich eine qualitativ verschiedene Funktionalität des Gesamtsystems, die weder in der Struktur noch in der Funktionalität der Komponenten enthalten ist. Die spezifische Qualität

wird durch die Wechselwirkung des Systems mit seiner Umgebung hervorgerufen und kann sich somit ändern.

Beispiel: Tiefenwahrnehmung des menschlichen Systems.

Z.B. bei der Betrachtung von Stereogrammen ("magic eye") tritt der Tiefeneffekt, der beim ersten Hinschauen nicht zu erfassen ist, plötzlich auf. Die Tiefeninformationen werden als neue Qualität wahrgenommen.

Definition: (abgeschlossenes System)

Ein abgeschlossenes System hat keine Beziehungen zu seiner Umgebung. Insbesondere sind die Systemgrenzen fest und unveränderlich.

Die Konzeption des abgeschlossenen Systems ist eine Idealisierung realer Sachverhalte, die bei natürlichen Systemen nicht anzutreffen ist. Die bewußte Vernachlässigung der Beziehungen zur Umwelt ist aber dann gestattet und von großem Wert, wenn die Beziehungen zur Umwelt im betrachteten Zusammenhang ohne Bedeutung sind.

- Das physikalische Modell von abgeschlossenen Systemen war eine wesentliche Voraussetzung zur Formulierung der Thermodynamik (des thermodynamischen Gleichgewichts).
- Das Informationsverarbeitungsparadigma beruht ebenfalls auf dem Konsens der Modellierung abgeschlossener Systeme.
  Dies bedeutet für

  Computer: Konfiguration und Funktion können als bekannt und konstant vorausgesetzt werden.

  Programme: Die als Programm codierte Problemlösung ist adäquat der Problemstellung (z.B. bleibt letztere erhalten während der Abarbeitung des Programms).

Ein weiterer bedeutender Modellierungsaspekt für Informatik-Systeme betrifft deren Stationarität.

Definition: (stationäres System, statisches System)

Ein stationäres System stellt eine Idealisierung dar für den Umstand, daß sich die Komponenten und deren Beziehungen, aber auch die Grenzen des Systems und dessen Beziehungen zur Umwelt nicht ändern.

Stationarität und Abgeschlossenheit sind wesentliche Voraussetzungen dafür, ein Informatik-System als deterministisch zu betrachten.

Definition: (deterministisches System)

Bei einem deterministischen System ist das zukünftige Verhalten eindeutig aus dem Verhalten in der Vergangenheit und dem aktuellen Systemzustand ableitbar.

Derartige Systemeigenschaften sind nützlich zur Beherrschung ihrer Komplexität. Sie begrenzen aber auch deutlich deren Funktionalität, d.h. ihre Verwendbarkeit zur Lösung bestimmter Probleme.

Beispiel:

1. Diese Systeme können nicht aus Erfahrung lernen.
2. Diese Systeme besitzen eine akzeptable Funktionalität nur im Rahmen modellierter Problemstellungen. Die Welt ist aber viel komplexer und ändert sich ständig, so daß eine Adaption des Systems an geänderte Bedingungen durch den Programmierer zu erfolgen hat (im Prinzip hoffnungslos aufwendig).

Natürliche Systeme sind grundsätzlich offen, dynamisch und damit in der Regel auch nicht-deterministisch.

Definition: (offenes System)

Ein offenes System steht in Wechselwirkung mit seiner Umgebung.

Die Wechselwirkungen können sich darstellen als

- wechselseitiges Ausüben von Kräften
- Austausch von Stoffen
- Austausch von Signalen (z.B. akustisch, optisch, taktil, chemisch)

Ein offenes System kann dadurch als abgeschlossenes System modelliert werden, daß seine Grenzen erweitert werden (s. Abb. 2.4).

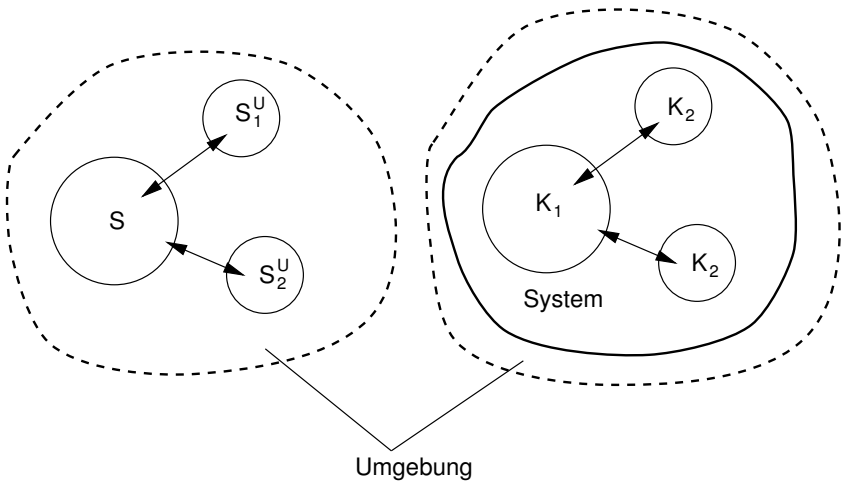


Abbildung 2.4: Übergang vom offenen zum abgeschlossenen System durch Erweiterung der Systemgrenzen

Definition: (dynamisches System)

Ein dynamisches System unterliegt Änderungen bezüglich seiner Komponenten, deren Wechselwirkung, seiner Grenzen und folglich seiner Funktionalität.

Diese Änderungen können ihre Ursache in den Beziehungen des Systems zur Umgebung haben.

Definition: (nicht-deterministisches System)

Bei einem nicht-deterministischen System ist das zukünftige Verhalten zufällig (stochastisch) oder wird durch außerhalb des Systems liegende Ursachen bestimmt.

Dynamische, offene Systeme sind in der Regel nicht-deterministisch.

Der durch die Umgebung induzierte Nicht-Determinismus ist Voraussetzung dafür, daß sich natürliche Systeme adaptieren können oder lernen.

Obwohl natürliche Systeme bestimmte Probleme sehr zuverlässig und schnell lösen können (z.B. Wahrnehmung), erweist sich die Modellierung künstlicher Systeme mit solchen Leistungen als sehr schwierig.

Ein nicht-deterministisches System kann durchaus determiniert sein. In der Regel ist es aber nicht-determiniert.

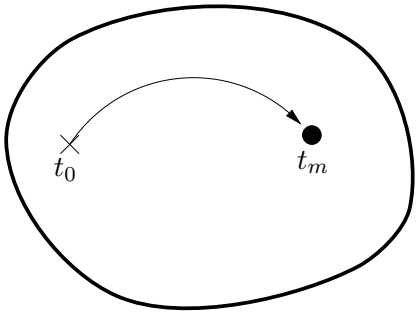
Definition: (determiniertes System)

Die Zustandstrajektorien eines determinierten Systems erreichen für gleiche Startzustände stets den gleichen Ort im Zustandsraum. Dieser Ort kann den Endzustand darstellen. Andernfalls heißt es nicht-determiniert.

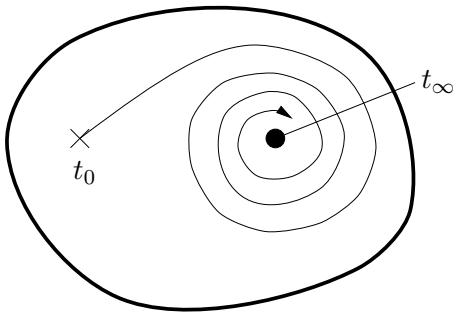
Systeme der Informationsverarbeitung sollen determiniert sein, d.h. ein eindeutiges Ergebnis liefern. Diese Forderung beruht auf der Vorstellung, daß
eine Berechnung als die Verfolgung einer Trajektorie eines Systems im Zustandsraum aufzufassen ist.
Eine weitere, oft nicht einzuhaltende Forderung an ein Informationsverarbeitungssystem betrifft die Endlichkeit der Trajektorie. Das System sollte nach einer vernünftigen Zeit zum Ergebnis kommen.

Definition: (terminierendes System)

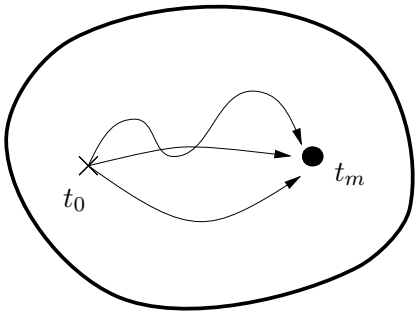
Ein System heißt terminierend, wenn es, ausgehend von einem Startzustand, nach endlich vielen Zustandsänderungen in einen Endzustand übergeht, andernfalls heißt es nicht terminierend.



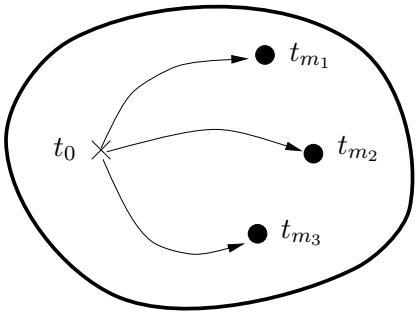
deterministisch, terminierend, determiniert



nicht terminierend, determiniert



nicht-deterministisch, terminierend, determiniert



terminierend, nicht determiniert

Abbildung 2.5: Illustration der Begriffe terminierend, determiniert, deterministisch

Weitere bedeutende Eigenschaftem informationsverarbeitender Systeme

Informationsverarbeitende Systeme verwenden in der Regel variable Daten, um ein Resultat zu berechnen.
Zwei Forderungen können gestellt werden:

a) das Resultat variiert in angemessener Weise mit der Variation der Daten

oder

b) unabhängig von der Variation der Daten soll das gleiche Resultat erzielt werden

---

Beispiele:

zu a) Lösung der Gleichung $y = x + a$, a Konstante
zu b) Unterscheidung: Äpfel / Birnen

Außerdem gestatten Computer nicht, Zahlen mit beliebiger Genauigkeit zu repräsentieren oder zu verarbeiten. Die Berechnung sollte hiervon nicht beeinflußt werden.

Fast alle Problemlösungen sind nicht exakte Berechnungen, sondern Approximationen!

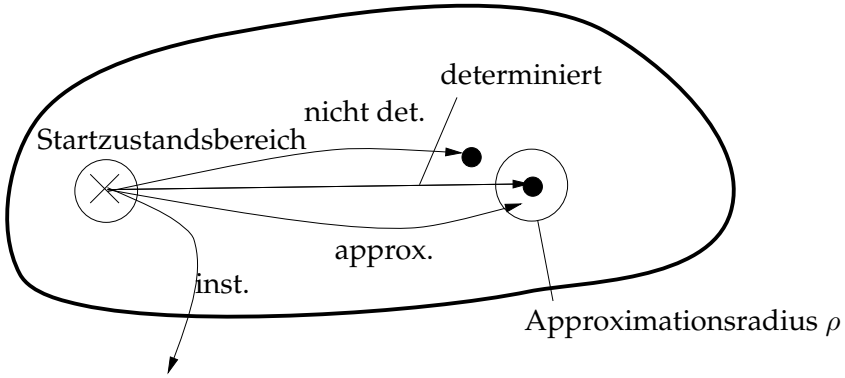
Beispiel:

Lösung der Gleichung $y = x + a$, a Konstante

Annahmen: $a \approx x \rightarrow y \approx 2a$
Störung $\epsilon \ll x$

1. eind. Lösung:	$y = a + b = c$	(determiniert)
2. stabiles System:	$(x + \epsilon) + a \approx x + a = y$	(approximierend)
	$(x \cdot \epsilon) + a \approx a$	(nicht determiniert)

3. instabiles System: $\frac{x}{\epsilon} + a \rightarrow \infty$



Definition: (stabiles System)

Ein System heißt stabil, wenn es auf kleine Störungen der Eingangsdaten mit kleinen Störungen der Ausgangsdaten reagiert. Andernfalls heißt es instabil.

In Abbhängigkeit vom zu wählenden Aproximationsradius wird das gestörte Ergebnis

- dem erwarteten zugeordnet oder
- als anderes Ergebnis akzeptiert

Ein *instabiles* System reagiert auf kleine Störungen der Eingangsdaten mit (ggf. unendlich) großen Störungen der Ergebnisdaten.

Definition: (robustes System)

Ein robustes System ist gegenüber Änderungen der Voraussetzungen im Vergleich zu den Modellannahmen stabil.

Das Ziel der Modellierung muß es sein, robuste Systeme zu erhalten.
Nichtlineare dynamische Systeme haben für bestimmte Parameterwerte ihrer Komponenten die Eigenschaft der Robustheit. Der Zielzustand dieser Systeme bildet einen *Attraktor*. Neuronale Netze sind künstliche Systeme mit dieser Eigenschaft.

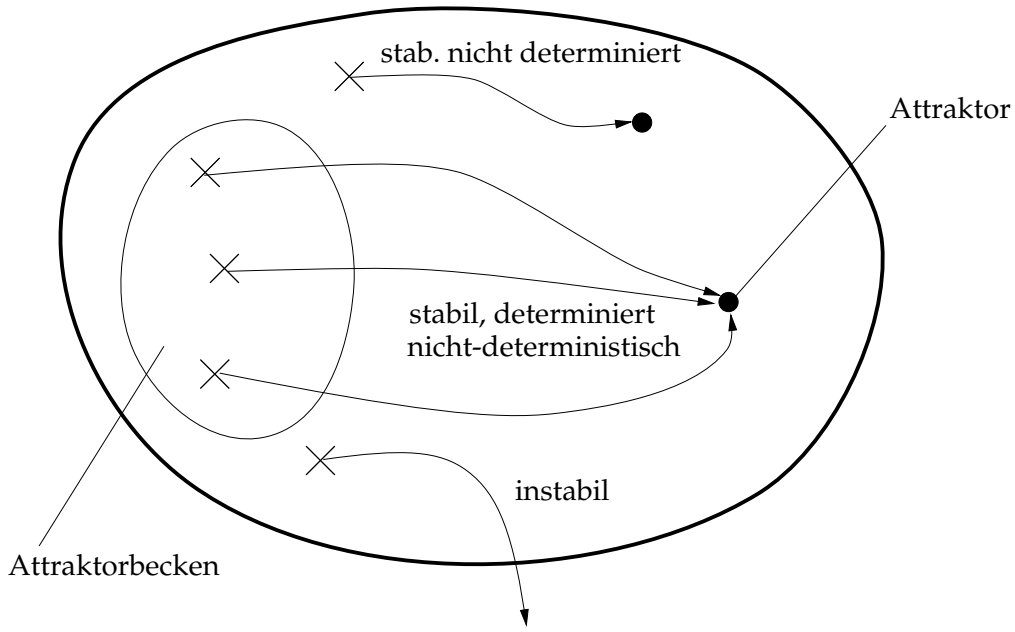


Abbildung 2.6: Für Startzustände im Attraktorbecken wird der Attraktor als Endzustand erreicht.

2.2 Nachricht, Datum und Information

Informationsverarbeitende Systeme stehen entweder untereinander oder/und mit ihrer Umgebung in Beziehungen.
Eigentlich trifft dies auf alle Systeme zu. Die meisten von ihnen können aber nur auf die einwirkenden Kräfte, Energien, Stoffe reagieren, um sich in Bezug auf die Situation bestenfalls in ein Gleichgewicht einzustellen.

Beispiel: Gravitationswechselwirkung der Planeten und der Sonne

> Die Gravitationswechselwirkung läßt sich zwar mittels Differentialgleichungen berechnen, aber niemand kommt auf die Idee, die Zustandsänderungen eines solchen Systems als Berechnung oder Informationsverarbeitung zu bezeichnen.

Informationsverarbeitende Systeme erzeugen aus den Mitteilungen anderer Systeme (oder der Umgebung) Informationen zum eigenen Vorteil oder geben Mitteilungen aus dem gleichen Grunde ab.
Solche Systeme müssen also einen Vorteil gewinnen. Dies setzt eine Zielstellung des Verhaltens voraus und die Fähigkeit, Mitteilungen zu Informationen zu wandeln:

biologische Systeme: Sicherung des Überlebens des Individuums und/oder der Art. Der Mechanismus der Selbstorganisation ist verantwortlich für die Informationsgewinnung und die hieraus folgenden Änderungen des Verhaltens.

künstliche Systeme (Computer): Der Mensch modelliert die Zielstellung und die Funktionalität der Zustandsänderung.

Außerdem verfügen informationsverarbeitende Systeme über spezielle Komponenten, um Mitteilungen aufnehmen bzw. abgeben zu können:

biologische Systeme: Rezeptoren und Effektoren

künstliche Systeme: Empfänger und Sender, Sensoren und Aktuatoren

Informationsverabeitende Systeme erkennen in Signalen Strukturen, denen eine Bedeutung zugeordnet wird. In Abhängigkeit vom Zustand des Systems oder/und von seiner konkreten Beziehung zur Umwelt kann die Bedeutung sehr unterschiedlich sein.
Indem das System Signale empfängt, erhält es Nachrichten, aus denen Information abgeleitet wird, die zu einer Systemreaktion (nach innen und/oder außen) führt.

Informationsverarbeitung führt in zweifacher Hinsicht zu Zustandsänderungen des Systems:

1. Informationsverarbeitung geschieht durch Zustandsänderungen und
2. Informationsverarbeitung bewirkt Zustandsänderungen.

2.2.1 Repräsentation von Information

Definition: (Signal)

Ein Signal ist die Darstellung einer Mitteilung durch die zeitliche (und räumliche) Veränderung einer physikalischen Größe.

Definition: (Nachricht)

Eine Nachricht ist die während des Transportes einer Mitteilung verwendete Strukturierung, abstrahiert von den physikalischen Besonderheiten des Signals. Eine Nachricht ist die *dynamische Repräsentationsform* der Information.

- Sender- und Empfängersysteme verfügen
  - entweder über gemeinsame Kenntnis der Strukturierungsvorschrift durch Konvention (Mensch - Mensch, Mensch - Computer, Instinkt der Tiere)
  - oder müssen diese Strukturierungsvorschrift lernen (Mensch - Mensch, . . .).

Definition: (Datum)

Ein Datum (Mehrzahl: Daten) ist die vom Empfänger einer Mitteilung verwendete *statische Repräsentationsform* einer Information.

- Nachrichten und Daten bezeichnen gleichartige physikalische Phänomene der realen Welt:
  - der Nachrichtentechniker sagt:
    - ▷ Signale werden moduliert oder

    - ▷ Informationen werden codiert.
  - diese physikalischen Phänomene sind
    - ▷ Zustände von Datenträgern (Datenspeicherung, Datenverarbeitung)
    - ▷ (zeitliche bzw. räumliche) Veränderung von Zuständen von Datenträgern (Datenverarbeitung) bzw. Nachrichtenträgern (Nachrichtenübertragung)

Definition: (Information)

Information ist die einer Nachricht (oder einem Datum) zugeordnete Bedeutung. Information existiert auf verschiedenen Ebenen der Abstraktion. Sie bedarf einer Form der Darstellung.

Definition: (Repräsentation)

Als Repräsentation bezeichnet man die äußere Form der Darstellung von Information.

- Nachrichten sind Repräsentationen der niedrigsten Abstraktionsstufe der Information.
- Daten stellen Repräsentationen auf unterschiedlichen Abstraktionsstufen dar.

Information ist eine immaterielle Wesenheit der realen Welt, die begrifflich schwer zu fassen oder gar zu quantifizieren ist.

- Zitate des Naturforschers Carl Friedrich von Weizsäcker:
  (Er befaßte sich mit der Frage, ob der Begriff Information einer naturwissenschaftlichen Betrachtung zugänglich ist.)

  1. ("Aufbau der Physik"): "Information ... ist ein Maß der Strukturmenge."
  2. ("Die Einheit der Natur"): "Information ist, was Information erzeugt."
  3. ("Die Einheit der Natur"): "Information ist nur, was verstanden wird."

  Diese Zitate betreffen drei unterschiedliche Aspekte der Information, mit deren Hilfe in vereinfachter Form durchaus Einsicht in das Wesen der Information erhalten werden kann:

  1. Zitat: *syntaktischer Aspekt*
  2. Zitat: *semantischer Aspekt*
  3. Zitat: *pragmatischer Aspekt*

2.2.2 Dimensionen des Informationsbegriffes

Die drei Dimensionen des Informationsbegriffes: (Küppers, 1990)

1. Die *syntaktische Dimension* umfaßt die Beziehung der Zeichen untereinander.
2. die *semantische Dimension* umfaßt die syntaktische Information und das, wofür sie steht.
3. Die *pragmatische Dimension* umfaßt die semantische Information und das, was dies für den beteiligten Sender und Empfänger als Handlungsherausforderung darstellt.

Definition: (Zeichen, Zeichenvorrat, Alphabet)

Ein Zeichen ist die kleinste bedeutungstragende Struktureinheit einer Nachricht. Es ist ein Element einer (endlichen) Menge von unterscheidbaren Dingen, dem Zeichenvorrat.
Ein Alphabet ist ein in vereinbarter Reihenfolge geordneter Zeichenvorrat.
(DIN 44300)

Beispiel: Zeichenvorrat

Die DNS wird aus Nukleinsäuren (Nukleotiden) gebildet:

A(denosinphosphat)

G(uanosinphosphat)

C(ytidinphosphat)

T(hymidinphosphat)

Beispiele: Alphabet

Dezimalziffern:	{0,1,2,...,9}
Binärziffern:	{0,1}
große lateinische Buchstaben:	{A,B,C,...,Z}

- Eine Nachricht ist eine Zeichenfolge.

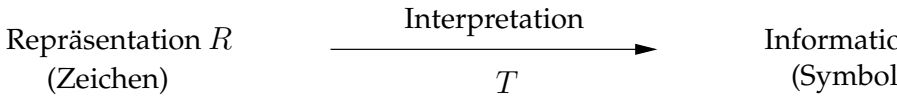
Beispiel:

Sätze der gesprochenen oder Schriftsprache

- Information entsteht aus Nachrichten durch Interpretation.
- Information wird durch Symbole repräsentiert.

Definition: (Syntax)

Die Syntax (Lehre vom Satzbau) ist der formale Aufbau von Sätzen und Wörtern einer Sprache, der durch Regeln (Grammatiken) beschrieben wird.



Definition: (Symbol)

Ein Symbol ist das Tupel (Zeichen, Bedeutung): $S = (R, I)$. Damit ist ein Symbol eine elementare, nicht weiter zerlegbare Einheit der Information.

Ludwig Wittgenstein: "Das Zeichen ist das sinnlich Wahrnehmbare am Symbol."

Definition: (Semantik)

In der Informatik wird die Semantik einer Repräsentation als das Tupel (Interpretation, Information/Bedeutung) $\mathcal{S} = (T, I)$ verstanden.

Voraussetzung der Interpretation ist ein für Sender und Empfängersystem verbindliches Bezugssystem.
Die Zuordnung zwischen Nachricht und Information ist demnach oft nicht eindeutig

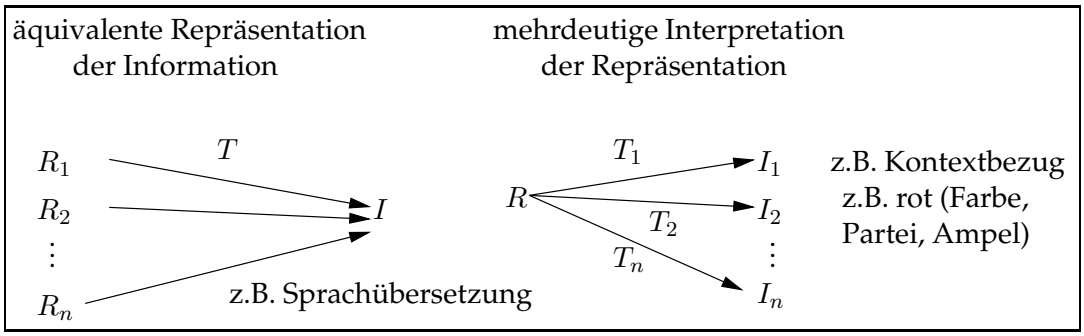


Abbildung 2.7: Nichteindeutige Zuordnungen zwischen Nachricht und Information

(s. Abb. 2.7)! Ein Beispiel hierfür ist z.B. die Formulierung "der Geist ist willig, aber das Fleisch ist schwach", die wortwörtlich ins Russische als "Der Wodka ist Gut, aber das Fleisch ist schlecht" übersetzt wurde.

Die gute Wahl einer Repräsentation ist von großer Bedeutung.
Kriterium: z.B. Minimierung der Verwechslungen bei konstanter Dauer der Interpretation.

Definition: (Informationsstruktur)

Eine Informationsstruktur ist das Tripel $(R, I; T)$. Die Informationsstruktur ist von grundlegender Bedeutung für alle in der Informatik modellierten Informationsverarbeitungsprozesse (Software und Hardware).

Wir werden in diesem Kapitel die in der Informatik verwendeten Zahlensysteme und im Kapitel 3 die Rechenstruktur der Booleschen Algebra kennenlernen.

Ein wesentlicher Aspekt der Modellbildung besteht darin, mit der prinzipiellen Unvollständigkeit von Informationsstrukturen adäquate Resultate zu erhalten. Dabei können alle drei Komponenten einer Informationsstruktur unzulänglich sein.

Beispiel:

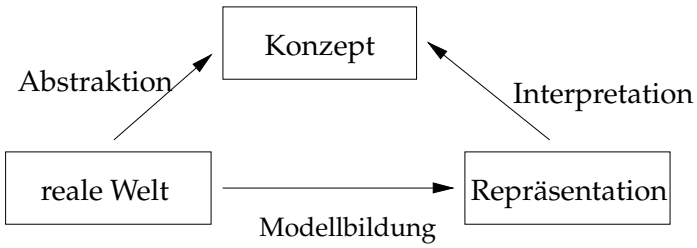
Repräsentation: Weder für $\mathbb{R}$ noch für $\mathbb{Z}$ existieren auf einer realen Maschine vollständige Repräsentationen.

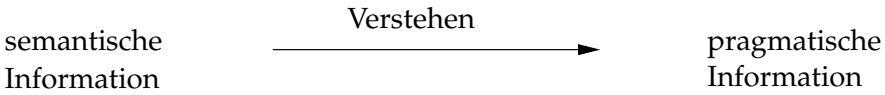
Information: Die Konzeptbildung unterliegt stets problemorientierten Beschränkungen. Die Information ist stets an ein Niveau der Konzeptbildung (und damit an die Pragmatik) gekoppelt.

Interpretation: Auch fehlerfrei formulierte Interpretationsvorschriften sind oft nur Approximationen:

Beispiel: Identifikation eines Stuhls (Individuum) als Stuhl (Äquivalenzklasse)
Da die explizite Formulierung der Äquivalenzklasse "Stuhl" nicht möglich ist, wird an deren Stelle häufig ein Prototyp verwendet.
"ein Stuhl" $\xrightarrow{T}$ "Stuhl" $\approx$ "Prototyp Stuhl"

Eine Zielstellung ist, stets zu erreichen, daß die mit einer Konzeptbildung (semantische Information) bezüglich der realen Welt verbundene Abstraktion adäquat realisiert wird durch die Folge von Modellbildungen der realen Welt und die entsprechende Interpretation der Repräsentationen.



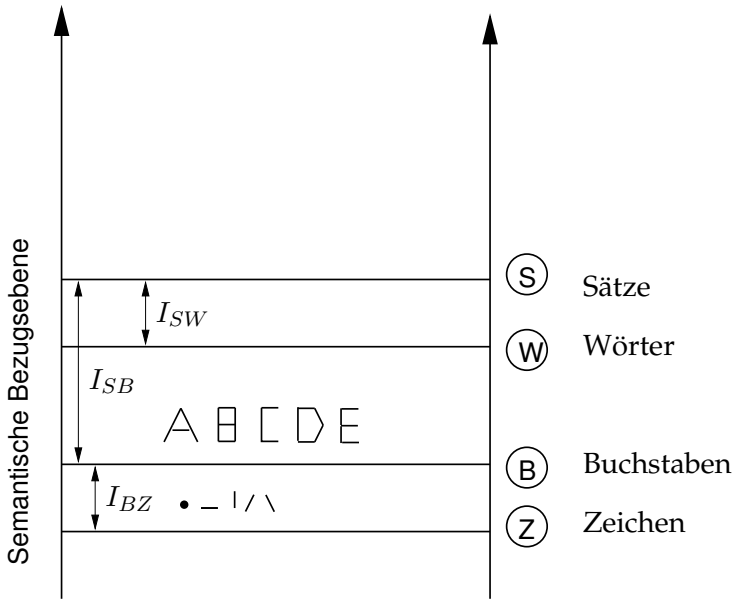


Die Bedeutung eines Konzepts (semantische Information) erschließt sich auf der pragmatischen Ebene. Den Übergang von der semantischen Ebene zur pragmatischen Ebene nennen wir Verstehen.

Definition: (Verstehen)

Als Verstehen bezeichnet man die Verkopplung der Bedeutungszuweisung (Interpretation) mit der Situation des Empfängers (Sachverhalte oder Gegenstände außerhalb der Nachricht). Dies bestimmt den pragmatischen Aspekt der Interpretation.

Beispiel: Satzbau unter Verwendung von Buchstaben, die aus Zeichen gebildet werden



Am obigen Beispiel erkennt man

- Symbole fixieren eine semantische Ebene als *Mikrozustand*.
- Kombinationen von Symbolen legen jeweils eine semantische Ebene als *Makrozustand* fest.

Daraus folgt

1. Information existiert nur in bezug auf zwei semantische Ebenen (Zitat 2 von C.F. v. Weizsäcker), die sich zueinander wie Mikro- und Makrozustand verhalten. Das zwischen Sender und Empfänger verbindliche Bezugssystem muß beide Ebenen umfassen.

2. Kombinationen von Symbolen sind wieder Symbole, also auch (nicht mehr elementare) Zeichenketten mit Bedeutung.

Ein und dieselbe Symbolfolge enthält je nach der semantischen Ebene, auf der sie abgefragt wird, verschiedene Mengen an syntaktischer Information.
(Wie diese zu berechnen ist, lernen wir im nächsten Abschnitt kennen.)
Je höher die semantische Ebene ist,

- umso komplexer ist die Repräsentation,
- umso komplexer ist das zur Interpretation zu vereinbarende Bezugssystem
- umso abstrakter muß die Funktion des Interpretationssystems sein
- umso effektiver kann der Informationsaustausch durch Nachrichten erfolgen.

Die verschiedenen Dimensionen des Informationsbegriffes sind nicht zu trennen.
(rekursiver Bezug: Pragmatik → Semantik → Syntax)
Der semantische Aspekt der Information wird erst wirksam, wenn die Information pragmatisch verbindlich wird.

2.3 Codierung, Informationsverarbeitung und Informationstheorie

Wir haben gelernt: Information existiert nur in unmittelbarer Ankopplung an Repräsentationen (in Gestalt der Interpretation von Zeichen als Symbole).
Es können unterschiedliche Repräsentationen zur Darstellung von Nachrichten oder Daten verwendet werden. Eine Darstellung hiervon muß allerdings als pragmatische Referenz dienen. Den Wechsel der Darstellungsform nennt man *Codierung*.
Nachrichten oder Daten sind als Zeichenketten codiert. Informationsverarbeitung stellt sich demzufolge als Transformation oder Manipulation von Zeichenketten dar (symbolbasiertes Informationsverarbeitungsparadigma).
Die *Informationstheorie* gestattet die quantitative Erfassung (Berechnung) von Strukturinformationen (syntaktischer Aspekt der Information) beim Wechsel der Repräsentationsform der Information.

Eine sehr gute Einführung in die hier behandelten Sachverhalte bietet
Hamming, R.W.: Information und Codierung, VCH Verlagsgesllschaft, 1987.

2.3.1 Codierung und Informationsverarbeitung

Definition: (Code)

DIN 44300:
Ein Code ist eine Vorschrift für die eindeutige

Zuordnung (Codierung) der Zeichen eines Zeichenvorrats (Urmenge) zu

denjenigen eines anderen Zeichenvorrats (Bildmenge).

Eine *Codierung* ist eine Abbildung $c : R \rightarrow R'$ auf der Menge der Repräsentationen.
Mit der Codierung werden in der Informatik (wie auch im täglichen Leben) unterschiedliche Ziele verfolgt:

1. Darstellung von Zeichen in einer für die Verarbeitung, Speicherung oder Übertragung besonders geeigneten Form.

2. Darstellung eines großen (unbeschränkten) Zeichenvorrats durch einen kleinen Zeichenvorrat.
3. Darstellung von Information durch einen Redundanz mindernden Code.
4. Darstellung von Information durch einen Redundanz vergrößernden Code.

Beispiele:

	tägl. Leben	Informatik
zu 1	*Verarbeitung:* nat. Sprache Anwendung von Fremdsprachen *Speicherung:* Schriftsprache lexikalisches Ordnungsprinzip (Telefonbuch) *Übertragung:* Signalzeichen/nat. Sprache Erhöhung der Störsicherheit der Erkennung durch Redundanz (Buchstabieralphabet)	*Verarbeitung/Speicherung:* Codierung beliebiger Probleme im Binärcode $\mathbb{B} = \{0, 1\}$ *Übertragung:* Verwendung komprimierender Codes (Kriterium: Zeit) Verwendung redundanter Codes (Kriterium: Sicherheit)
zu 2	Signalsprachen (Morsealphabet, Flaggensprachen)	$\mathbb{N} \rightarrow \mathbb{B}$ $\{0, 1, ..., 9\} \rightarrow \{0, 1\}$
zu 3	Stenographie	Bildtelephonie
zu 4	Buchstabieralphabet, z.B. N(ovember)	Bildverarbeitung: Bildpunktweise Bildrepräsentation (Quadrat)

Unabhängig vom verwendeten Zeichenvorrat besteht die Forderung an eine Informationsstruktur $(R, I; T)$, daß sämtliche gewünschte Information darstellbar ist.
Diese Forderung nach Vollständigkeit läßt sich mathematisch in folgender Weise ausdrücken:
Die Interpretation T stellt eine *surjektive Abbildung* $T : R \rightarrow I$ dar, d.h. es existiert zu jeder Information $i \in I$ eine Repräsentation $r \in R$ mit $T[r] = i$.
Die Abbildung $c : R \rightarrow R'$ induziert im allgemeinen eine Relation c' zwischen den R

und R' zugeordneten Informationen I und I', so daß gilt: $I\, c'\, I'$
Gewünscht wird häufig, daß c' nicht nur eine Relation, sondern eine Abbildung ist. Sind M und N nichtleere Mengen, und ist durch eine Vorschrift $x \mapsto f(x)$ jedem $x \in M$ genau ein Element $f(x) \in N$ zugeordnet, so sprechen wir von einer Funktion oder Abbildung. Durch die Relation "$y = f(x)$, falls $y^2 = x$" ist keine Funktion gegeben, da jedem $x \neq 0$ zwei Zahlen y mit $y^2 = x$ entsprechen. Dann erhält man eine informationstreue Codierung.

Definition: (informationstreue Codierung)

Eine Codierung $c : R \to R'$ heißt informationstreu, wenn sie eine Abbildung $c' : I \to I'$ induziert. In diesem Fall gilt die Operatorgleichung

$$c' \circ T = T' \circ c$$

mit $T' : R' \to I'$ und $T : R \to I$.

Die Gleichung $c' \circ T = T' \circ c$ ist als Operatorgleichung aufzufassen. Sie entspricht der Gleichung $c'\{T[R]\} = T'[c\{R\}]$.
Da die Abbildungen auf die gleiche Menge R wirken, kann auf ihre Darstellung in der Operatorschreibweise verzichtet werden.
$\circ$ ist hier das Zeichen der Hintereinanderausführung der Operatoren.
Hinweis: Die Hintereinanderausführung von Operatoren ist stets von rechts nach links zu lesen!
Für informationstreue Codierungen erhält man das folgende Diagramm:

$$\begin{array}{ccc} R & \xrightarrow{\;T\;} & I \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle c'} \\ R' & \xrightarrow[\;T'\;]{} & I' \end{array}$$

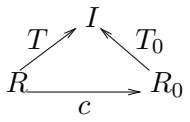
Die Abbildung $c' : I \to I'$ heißt *Vorschrift zur Informationsverarbeitung* oder *Abstraktion*.
Da diese Abbildung selbst nicht angebbar ist, weil auch I und I' nicht angebbar sind, wird die Abbildung $c : R \to R'$ als Ersatzoperation für die Informationsverarbeitung ausgeführt.
Man spricht von *Umcodierung*, wenn sowohl c als auch c' umkehrbar sind, wenn also die inversen Abbildungen $c^{-1} : R' \to R$ und $c'^{-1} : I' \to I$ existieren.
Es ist von besonderer Bedeutung, wenn c' die identische Abbildung darstellt, wenn also gilt: I'=I.

Die zugehörige Codierung heißt dann *Äquivalenztransformation*, weil sie semantisch äquivalente Transformationen verbindet.

Beispiele: Äquivalenztransformationen

1. Die formale Semantik-Definition einer Informationsstruktur $(R, I; T)$ erfolgt durch Rückführung auf eine andere (bekannte) Struktur $(R_0, I; T_0)$ durch eine informationstreue Umcodierung c, so daß

$$T := T_0 \circ c.$$



2. Übliches Rechnen in der Sprache der arithmetischen Ausdrücke.
   $5 + 3 = 8$ (hier wird links vom Gleicheitszeichen ein anderer Code verwendet, als rechts)
   $a + b = c + d \longrightarrow a = c + d - b$ (hier wird durch Umformen der Gleichung eine andere Codierung erzielt)
3. Repräsentationsformen von Zahlen in der Mathematik: Z.B. kann die "Eins" jeweils äquivalent dargestellt werden durch

$$1 \qquad 0! \qquad 2^0$$

Die semantische Äquivalenz von Repräsentationen gestattet solche auszuzeichnen, die von besonders einfacher Darstellung sind. Diese bezeichnet man als *Normalformen*.

Definition: (Normalformensystem, vollständig, eindeutig)

Ein Normalformensystem ist eine Menge von Normalformen $\mathcal{S} \subseteq \mathcal{R}$, wobei $\mathcal{R}$ eine Menge von Repräsentationen ist.
Ein Normalformensystem heißt

- vollständig,
  wenn für jedes $R \in \mathcal{R}$ eine semantisch äquivalente Normalform $S \in \mathcal{S}$ existiert
- eindeutig,
  wenn die auf S eingeschränkte Interpretation $T_S : S \to I$ eine injektive Abbildung ist. Eine Abbildung nennt man injektiv (ein-eindeutig, 1-1-Abbildung), wenn aus $f(x_1) = f(x_2)$ stets $x_1 = x_2$ folgt, oder andres ausgedrückt, wenn die Gleichung $f(x) = y$ für $y \in N$ höchstens eine Lösung x bestitzt.

Da auf der Menge der eindeutigen Normalformen die Interpretation eine eineindeutige Abbildung ist, können die entsprechenden Informationen jeweils mit ihren Normalformen gleichgesetzt werden!

Das heißt, Repräsentationen sind nicht Träger von Informationen, sondern stehen für Informationen. Da aber der Mensch als Nutzer des Computers die Interpretation der Normalformen realisiert, ist die Symbiose Mensch-Computer die Voraussetzung des Symbolverarbeitungsparadigmas der Informatik. Informationsverarbeitung durch Computer ist damit die Ausführung von Äquivalenztransformationen, die dem Menschen eine bessere Interpretation erlauben.

Beispiele: eindeutige und vollständige Normalformen

1. Sei R eine beliebige Repräsentation natürlicher Zahlen $\mathbb{N}$.

   $\rightarrow$ $S \sim$ Menge der nichtleeren Zeichensequenzen in Dezimaldarstellung ohne führende Nullen, z.B.
   ≪zwölf≫ = '12' $\quad \Sigma_{\mathbb{N}} = \{0, 1, \ldots, 9\}$
   oder

   $\rightarrow$ $S \sim$ Menge der Binärsequenzen ohne führende Nullen, z.B.
   ≪zwölf≫ = 'LL00', ≪drei≫='LL' $\quad \mathbb{B}$={0,L}

2. In der Menge der Boolschen Ausdrücke sind die Wahrheitswerte 'L' = ≪wahr≫ und '0' = ≪falsch≫ eindeutige Normalformen.

In der Informatik werden Repräsentationen in der Form von Zeichensequenzen behandelt.
Diese universelle Repräsentationsform gestattet die Manipulation sowohl von Texten, Zahlen als auch Operationen nach einem formalisierten Schema.

Beispiel:

$$\text{≪Katze≫: 'Katze'} \xrightarrow[\text{Datenverarb.}]{\text{Informationsverarb.}} \text{'CAT'}$$

$$\text{≪sieben≫: '5+2'} \xrightarrow[\text{Datenverarb.}]{\text{Informationsverarb.}} \text{'7'}$$

2.3.2 Zeichensequenzen

Definition: (Zeichensequenz, Wort)

Sei $\Sigma = \{x_1, x_2, \ldots, x_m\}$ ein endlicher Zeichenvorrat. Eine Folge von Zeichen

$$x := <x_{i_1} x_{i_2} \cdots x_{i_n}>, i_j \in \mathbb{N}_m, j \in \mathbb{N}_n$$

$x_{i_j} \in \Sigma$, heißt Zeichensequenz oder Wort der Länge $n = |x|$ über dem Zeichenvorrat Σ.

- Mathematisch interpretiert sind Zeichensequenzen der Länge n als n-Tupel zu bezeichnen. Die Menge der n-Tupel Σ^n erhält man durch Bildung des n-fachen *kartesischen* oder *direkten Produktes*

$$\Sigma^n := \underbrace{\Sigma \times \cdots \times \Sigma}_{n \text{ mal}}$$

der Menge Σ.
Also ist $x \in \Sigma^n$ ein Element aus der Menge aller Wörter der Länge n.

- Σ^2 ist die Menge der geordneten Paare $<ab>$, $a, b \in \Sigma$
Σ^3 ist die Menge der geordneten Tripel $<abc>$, $a, b, c \in \Sigma$

- Die Menge aller endlichen Sequenzen von Zeichen aus Σ sind definiert durch

$$\Sigma^+ := \bigcup_{n>0} \Sigma^n.$$

Die Elemente von Σ^+ heißen *Wörter* über Σ.

- Oft wird auch die *leere Sequenz* (das leere Wort) ϵ benötigt, $|\epsilon| = 0$. Es ist das einzige Element der Menge Σ^0:

$$\Sigma^0 := \{\epsilon\}.$$

- Damit erweitert sich die Menge alle endlichen Sequenzen von Zeichen aus Σ zu

$$\Sigma^* := \Sigma^+ \cup \{\epsilon\} = \bigcup_{n \in \mathbb{N}} \Sigma^n.$$

Beispiele:

1. Die natürlichen Zahlen $\mathbb{N}$ in Dezimaldarstellung sind Zeichenreihen aus $\Sigma = \{0, 1, \ldots, 9\}$.
   Alle Stellenwertsysteme zu irgend einer Basis repräsentieren die natürlichen Zahlen als Zeichenreihen (s. Abschnitt 2.5)
2. Insbesondere ist auch der binäre Zeichenvorrat $\mathbb{B} = \{0, \mathrm{L}\}$ geeignet, aus der Menge der Binärsequenzen

   $$\mathbb{B}^* = \{\epsilon, <0>, <\mathrm{L}>, <00>, <0\mathrm{L}>, <\mathrm{L}0>, \ldots\}$$

   ein Stellenwertsystem zu bilden.

Die einzige Beziehung zwischen den Zeichen einer Zeichenreihe ist die des Aufeinanderfolgens. Das Verketten oder die Konkatenation ist eine elementare Operation zwischen zwei Zeichenreihen.

Definition: (Konkatenation)

Unter Konkatenation versteht man die zweistellige Abbildung

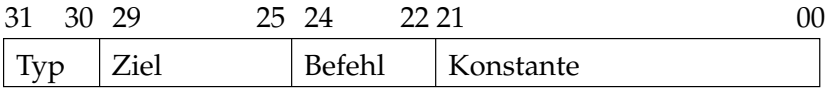
$$\mathrm{conc} : \Sigma^* \times \Sigma^* \to \Sigma^*.$$

Die Konkatenation zweier Zeichenreihen $x, y \in \Sigma^*$, $x = <x_1 \ldots x_s>, y = <y_1 \ldots y_t>$, ergibt eine Zeichenreihe $z \in \Sigma^*$, $z = <z_1 \ldots z_{s+t}> = <x_1 \ldots x_s\, y_1 \ldots y_t>$:

$$\mathrm{conc}(x, y) \equiv x \circ y = <xy> = z$$

Beispiele:

1. $\underset{\text{Formel}}{1 \circ 2} = \underset{\text{Zeichensequenz}}{<12>} = \underset{\text{Zahl (Zeichen)}}{'12'} = \underset{\text{Bedeutung}}{\ll \text{zwölf} \gg}$
2. Maschinenbefehle sind komplexe strukturierte binäre Zeichenreihen:



   Übersetzer erzeugen Konkatenationen,
   Prozessoren brechen Konkatenationen auf.

Die Reihenfolge der Ausführung der Verkettung mehrer Zeichenreihen spielt keine Rolle, es gilt das *Assoziativgesetz*. Für $u, v, w \in \Sigma^*$ gilt

$$(u \circ v) \circ w = u \circ (v \circ w).$$

Das Assoziativgesetz gilt auch für Σ^+. Die Menge Σ^+ bildet eine *Halbgruppe* (d.h. eine bezüglich einer assoziativen zweistelligen Operation abgeschlossene Menge).
Das leere Wort ϵ spielt bei der Konkatenation die Rolle des *neutralen Elements*:

$$w \circ \epsilon = w = \epsilon \circ w.$$

Die mit dieser Eigenschaft ausgestattete Menge Σ^* bezeichnet man als *Monoid*.
Mit den Begriffen Zeichen, Zeichenvorrat und Zeichensequenz läßt sich der Begriff Codierung interpretieren als Abbildung

$$c : \Sigma \rightarrow \Sigma'^*.$$

Ein Zeichenvorrat wird auf einen anderen abgebildet, dessen Elemente Wörter über einem anderen Zeichenvorrat sind.

Beispiel: Binärcodierung

$$\Sigma' = \mathbb{B} = \{0, \mathrm{L}\} \Rightarrow \Sigma'^* = \mathbb{B}^* \quad (\mathbb{B}^* \sim \text{Menge der Binärwörter})$$

$$2_{10} = \text{<L0>} = \mathrm{L0}_2$$

Werden nicht nur Zeichen $x_i \in \Sigma$ codiert, sondern Wörter $x \in \Sigma^*$, so sind zwei Möglichkeiten zu unterscheiden:

Definition: (Serienwortcodierung)

Die Codierung von Einzelzeichen

$$c : \Sigma \rightarrow \Sigma'^*$$

induziert die Serienwortcodierung

$$c_s^* : \Sigma^* \rightarrow \Sigma'^*,$$

definiert durch

$$\begin{aligned} c_s^*(\epsilon) &= \epsilon \\ c_s^*(x = <x_1 \cdots x_m>) &= <c(x_1) \cdots c(x_m)>. \end{aligned}$$

Definition: (Parallelwortcodierung)

Die Codierung von Einzelzeichen

$$c : \Sigma \rightarrow \Sigma'^n$$

induziert die Parallelwortkodierung

$$c_p^* : \Sigma^* \rightarrow (\Sigma'^n)^*,$$

definiert durch

$$\begin{aligned} c_p^*(\epsilon) &= \epsilon \\ c_p^*(x = <x_1 \cdots x_m>) &= <<c(x_1)> \cdots <c(x_m)>>. \end{aligned}$$

Beispiel: (Parallelcodierung) Binärcodierung der Ziffernfolge <134>

$$\begin{aligned} c_p^*(<134>) &= <<000L><00LL><0L00>> \\ c_s^*(<134>) &= <<L><LL><L00>> \\ &\quad \text{falls Binärcodierung in Normalform} \end{aligned}$$

Wenn in diesem Beispiel aber die Ziffernfolge <134> als Zahl im dekadischen Stellenwertsystem '134'=≪ einhundertvierunddreißig≫ interpretiert wird, kann diese Zahl als eigenes Zeichen $x_{134} = 134_{10}$ verstanden werden, dem eine Binärzahl $c(x_{134})$ durch Zeichencodierung zugeordnet wird (Seriencodierung):

$$c(<134_{10}>) = <10000110_2>.$$

Diese durch Zeichencodierung realisierte Wortcodierung beruht auf einer Codeerweiterung der verfügbaren Zeichen von Ziffern auf Zahlen (siehe auch Beispiel 3 weiter unten).

Eigenschaften

1. Bei der Parallelcodierung bleibt die Codewortstruktur erhalten, jedes einzelne Zeichen kann deshalb unabhängig codiert werden. Die Parallelcodierung ist auf Codes mit fester Wortlänge beschränkt (aus technischen Gründen bevorzugt, da dann keine Probleme der Rekonstruktion der Wörter $x \in \Sigma^*$ aus Code $x' \in (\Sigma'^n)^*$, d.h. Injektivität der Codierung ist gewährleistet).

2. Bei der Seriencodierung geht die Codewortstruktur verloren. Seriencodierung ist auch für Codes mit unterschiedlicher Wortlänge geeignet. Beispiel: Morse-Code (aus technischen Gründen nicht verbreitet).
   Im Fall unterschiedlicher Längen der Codewörter muß zur eindeutigen Rekonstruktion der Ursprungswörter die folgende Bedingung erfüllt werden.

Definition: (Präfix- oder Fano-Bedingung, Präfixcode)

Kein Wort $w \in \Sigma'^*$ ist der Anfang eines anderen Wortes $v \in \Sigma'^*$, d.h. es gibt kein $u \in \Sigma'^*$ mit $v = wu$.
Ein Code, der die Präfix- oder Fanobedingung erfüllt, heißt Präfixcode.

Ein Präfixcode stellt damit die Injektivität der Codierung sicher. Die Fano-Bedingung gestattet das eindeutige Erkennen der Wortfugen.

Beispiel: Morsecode

Erweiterung des Morsecodes auf den dreielementigen Zeichenvorrat $\Sigma' = \{\bullet, -, \sqcup\}$. Konvention: hinter jedem Wort folgt ein "$\sqcup$". Ohne das Zeichen $\sqcup$ kommt es zu Mehrdeutigkeiten, da die Morsecodes die Fano-Bedingung nicht erfüllen, z.B. $< \bullet\bullet\bullet >$ $\leftrightarrow$ "S"
$\leftrightarrow$ "EEE"
$\leftrightarrow$ "EI"
$\leftrightarrow$ "IE"

2.3.3 Binärcodierung und Entscheidungsinformation

Sowohl Nachrichtenübertragung als auch Informationsverarbeitung setzen voraus, daß die Zeichen und die durch sie repräsentierte Information im voraus bekannt sind.

Die sukzessive Interpretation (Identifikation) der Zeichen einer Zeichensequenz führt zur Interpretation der gesamten Nachricht.

Die *Informationstheorie* (C.Shannon, 1948) befaßt sich mit dem aus diesen Entscheidungen ableitbaren Informationsgehalt einer Nachricht (eines Datums). Sie berücksichtigt dabei den syntaktischen Aspekt der Information. Die Strukturinformation wird dabei als Entscheidungsinformation interpretiert. Sie erfaßt den minimalen Aufwand zur sequentiellen Klassifizierung der Zeichen einer Nachricht.

Der Bestimmung des Informationsgehaltes liegt folgendes Gedankenexperiment zu Grunde:
Gegeben seien Sender und Empfänger mit vereinbartem Bezugssystem. Der Empfänger soll mit gezielten Fragen an den Sender eine Nachricht erraten. Die Antwort des Senders darf sich nur auf {≪Ja≫, ≪Nein≫} oder {≪richtig≫, ≪falsch≫} beschränken. Das Informationsmaß ist ein Maß für die minimale Anzahl zu stellender Fragen, die zur Interpretation eines Zeichens oder einer Zeichensequenz nötig sind.

Offensichtlich hängt ein Maß der Entscheidungsinformation von der Häufigkeit des Auftretens eines Zeichens in einer Zeichensequenz ab (bzw. von der Häufigkeit eines Ereignisses bzw. von der Häufigkeit, mit der ein System einen gewissen Zustand annehmen kann). Dabei trägt ein häufig auftretendes Zeichen weniger Information als ein seltener auftretendes.
Zitat: *"Ein Zug, der entgleist, hat eben mehr Nachrichtenwert als einer, der ankommt"* (G. Lacour)

Nachrichten mit diese Eigenschaft heißen *Shannonsche Nachrichten*. Die Häufigkeit ist ein Maß für die Wahrscheinlichkeit, genauer: a priori Wahrscheinlichkeit.

Die minimal notwendige Anzahl von Entscheidungen sollte also die erwartete Häufigkeit des Auftretens von Zeichen in einer Nachricht berücksichtigen.
Dies ist möglich, wenn zum vereinbarten Bezugssystem auch die Kenntnis der a priori Wahrscheinlichkeiten gehört.

Definition: (Bit)

Gegeben sei die Binärcodierung eines Sachverhaltes als binäre Zeichensequenz (oder Binärwort) $w \in \mathbb{B}^*$ mit $w = <w_1 \cdots w_n>$. Jedes Binärzeichen $w_i \in \mathbb{B} = \{0, \mathrm{L}\}$ kann einen von zwei Werten annehmen. Es wird als Bit (binary digit) bezeichnet. Die Anzahl der Bits eines Binärwortes gibt dessen Länge an. Die Einheit für Bits ist b:

$$|w| = n[\mathrm{b}].$$

Ein Bit vermag also einen beliebigen zweiwertigen Zustand eines Systems oder Sachverhaltes zu repräsentieren.

Definition: (Byte)

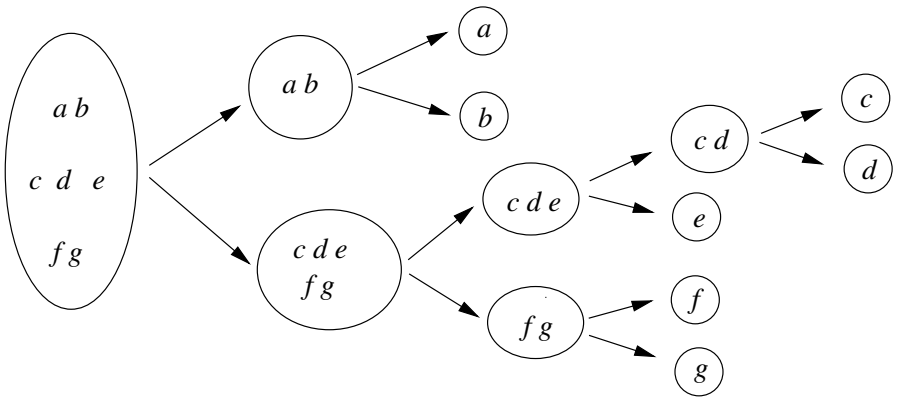
Binärsequenzen der Länge 8b heißen Byte. Die Maßeinheit ist B. Ein Byte vermag 2^8=256 Zustände zu repräsentieren.

Definition: (bit)

Ein bit ist die elementare Einheit der Entscheidungsinformation, das heißt, der zur Identifizierung eines Wertes eines Bits notwendige Aufwand.

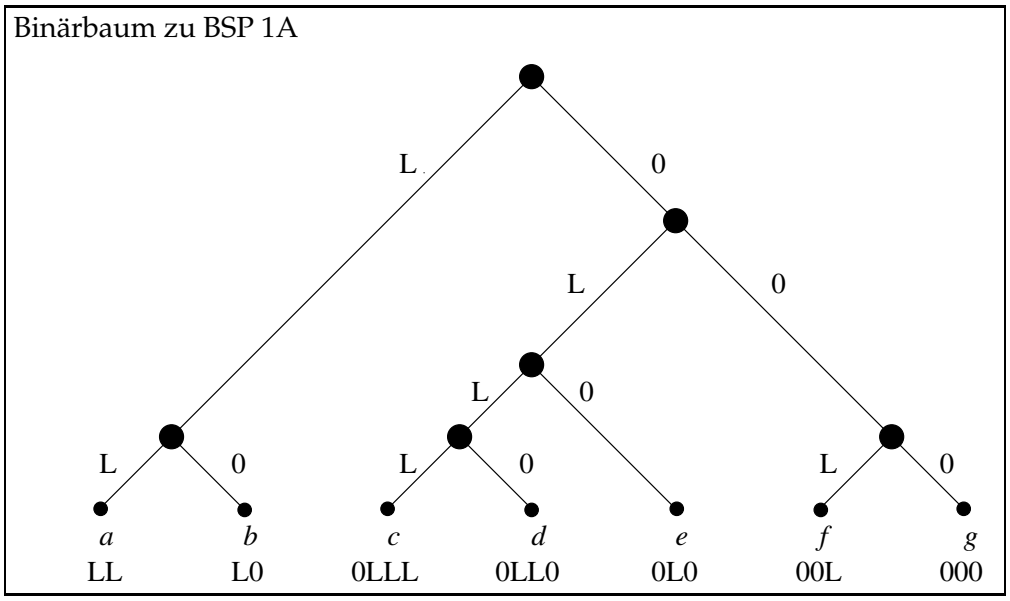
Jeder Sachverhalt, der N Zustände annehmen kann ($N = |\Sigma|$) läßt sich binär codieren $c : \Sigma \rightarrow \mathbb{B}^*$. Dies geschieht in der Art einer Entscheidungskaskade, d.h. der sukzessiven Zerlegung des Zeichenvorrates in zwei (nichtleere) Teilmengen, bis einelementige Teilmengen (die Zeichen bzw. Zustände) erreicht werden.

Beispiel: BSP 1A



$\Sigma = \{a, b, c, d, e, f, g\}$

Die Entscheidungskaskade für die binäre Codierung eines Sachverhaltes bildet die Identifikaton der möglichen Zeichen oder Zustände auf einen Binärbaum (*Codebaum*) ab.



Definition: (Binärbaum)

Ein Binärbaum ist eine hierarchisch geordnete Datenstruktur in Gestalt eines Graphen, dessen Knoten höchstens zwei Nachfolgerknoten haben. Als Wurzel bezeichnen wir den Knoten, für den es keinen Vorgängerknoten gibt. Ein Knoten ohne Nachfolger heißt Blatt.

Ein Baum heißt

- *ausgeglichen*, wenn alle Blätter die gleiche Tiefe haben und
- *vollständig*, wenn alle Knoten zwei oder keine Nachfolger haben.

Der mit einer Entscheidungskaskade korrespondierende Binärbaum repräsentiert mit seinen Knoten Teilmengen von Σ, wobei die Wurzel für ganz Σ steht und die Blätter einelementige Teilmengen (die Zeichen aus Σ) sind. Die Vereinigung aller Blätter ergibt den gesamten Zeichenvorrat Σ. Die von den inneren Knoten ausgehenden Kanten repräsentieren die Entscheidungswerte $w_j^i \in \{0, \mathrm{L}\}$, dabei gehen von jedem Knoten genau eine L- und eine 0-Kante aus (vollständiger Binärbaum).
Die Binärcodierung $w^i \in \mathbb{B}^*$ mit $w^i = <w_{1_i}^i \cdots w_{n_i}^i>$, $w_j^i \in \mathbb{B}$ eines Zeichens $z_i \in \Sigma$ erhält man durch Konkatenation der Entscheidungswerte w_j^i beim Durchlaufen des Binärbaums von der Wurzel (Σ) zu den Blättern (z_i).

Offensichtlich ist die Codierung des Beispiels

1. mehrdeutig möglich
2. ungünstig realisiert, falls die Wahrscheinlichkeitsgewichte gleich sind (die Binärworte haben unterschiedliche Länge)

Um den Aufwand für Übertragung, Speicherung oder Interpretation eines Codes zu reduzieren sind zwei Strategien realisierbar:

1. wenn alle zu codierenden Zeichen $z_i \in \Sigma$, $|\Sigma| = N$ mit gleicher Wahrscheinlichkeit $p_i = \frac{1}{N}$ auftreten:
   $\rightarrow$ Konstruktion eines ausgeglichenen Codebaumes
2. wenn jedes zu codierende Zeichen $z_i \in \Sigma$, $|\Sigma| = N$ mit der Wahrscheinlichkeit p_i auftritt, $\sum_{i=1}^{N} p_i = 1$:
   $\rightarrow$ Konstruktion eines bezüglich der Wahrscheinlichkeitsgewichte ausgeglichenen Codebaumes.

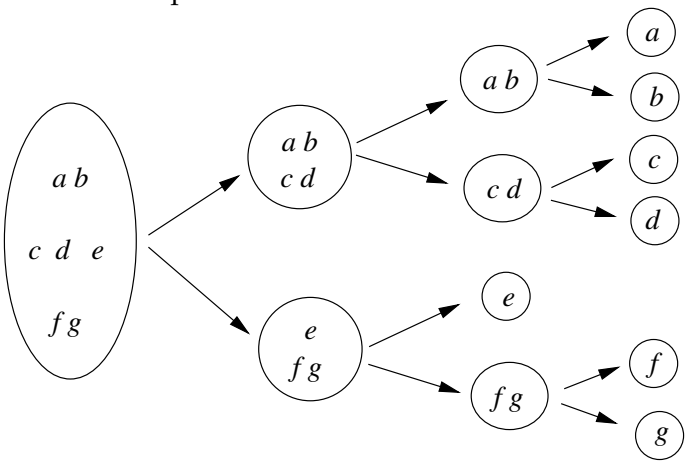
Ziel beider Strategien ist es, die Entscheidungskaskade so zu konstruieren, daß im Mittel eine minimale Anzahl von Entscheidungen benötigt wird.

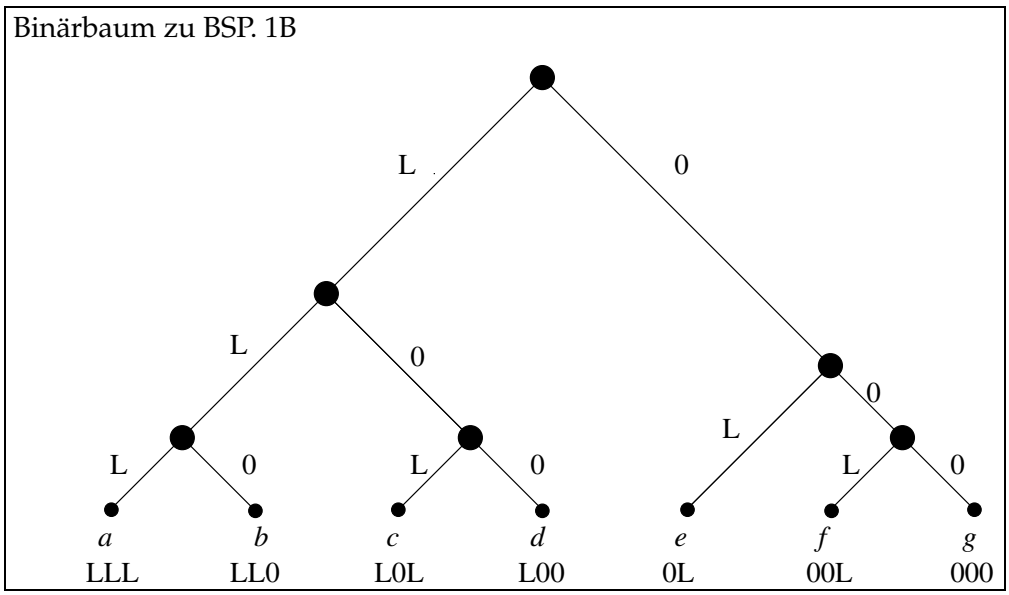
zu 1. Konstruktion eines ausgeglichenen Codebaumes:

Es sind zwei Fälle zu unterscheiden:
a) N beliebig: Zerlege Σ in jedem Entscheidungsschritt in etwa gleich große Teilmengen (BSP 1B). Es ist prinzipiell ein ausgeglichener Codebaum nur approximierbar.

Beispiel: BSP 1B

Σ wie in Beispiel 1A

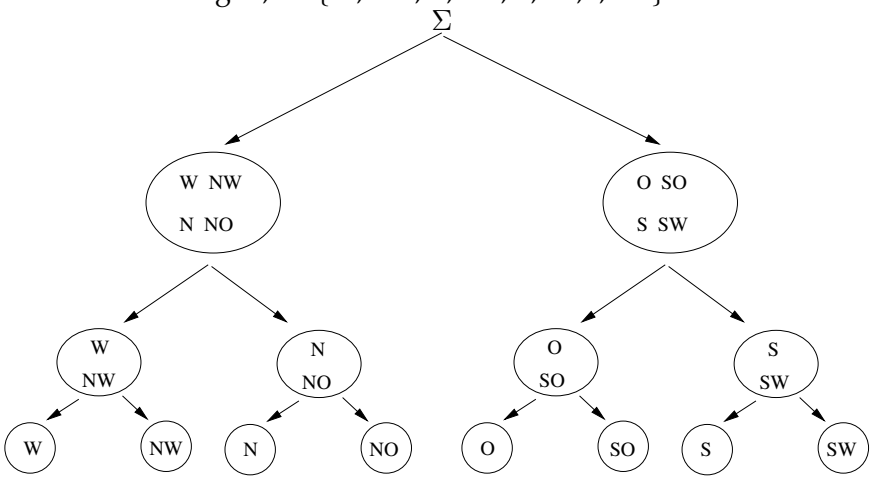




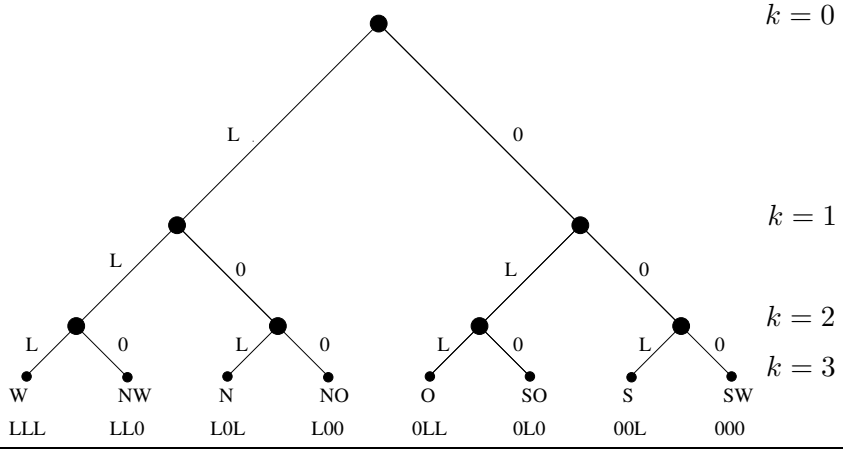
b) $N = 2^n$: Halbiere Σ in jedem Entscheidungsschritt (BSP 2A: $p_i = 2^{-n}$).
Es entsteht ein vollständiger, ausgeglichener Binärbaum der Tiefe $k_{\max} = n$, wobei alle inneren Knoten den Grad 2 haben und alle Blätter auf gleicher Tiefe liegen. Die Codeworte $w^i \in \mathbb{B}^*$ haben alle die gleiche Länge $|w^i| = n$.

Beispiel: BSP 2A

Himmelsrichtungen, Σ={W,NW,N,NO,O,SO,S,SW}



Binärbaum zu BSP. 2A

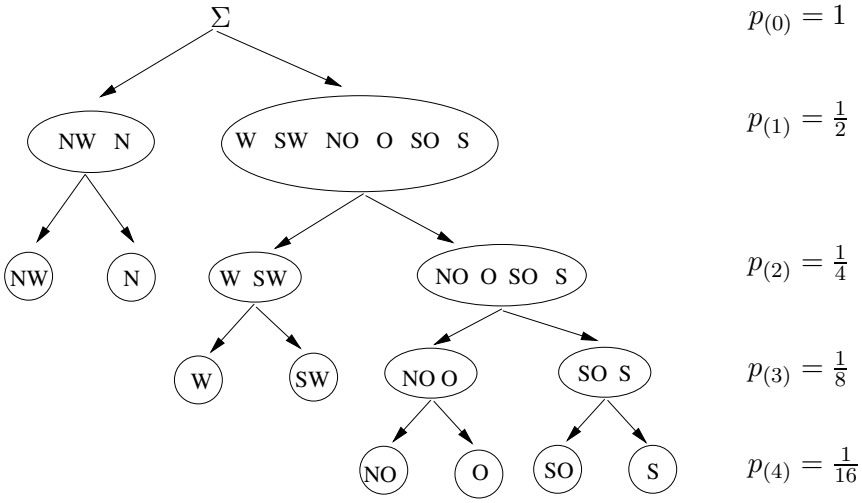


zu 2.: Konstruktion eines bezüglich der Wahrscheinlichkeitsgewichte ausgeglichenen Codebaumes:
Zerlege Σ so, daß bei jedem Schritt die Summe der Wahrscheinlichkeiten für die Zeichen der einen und der anderen Teilmenge möglichst gleich sind.

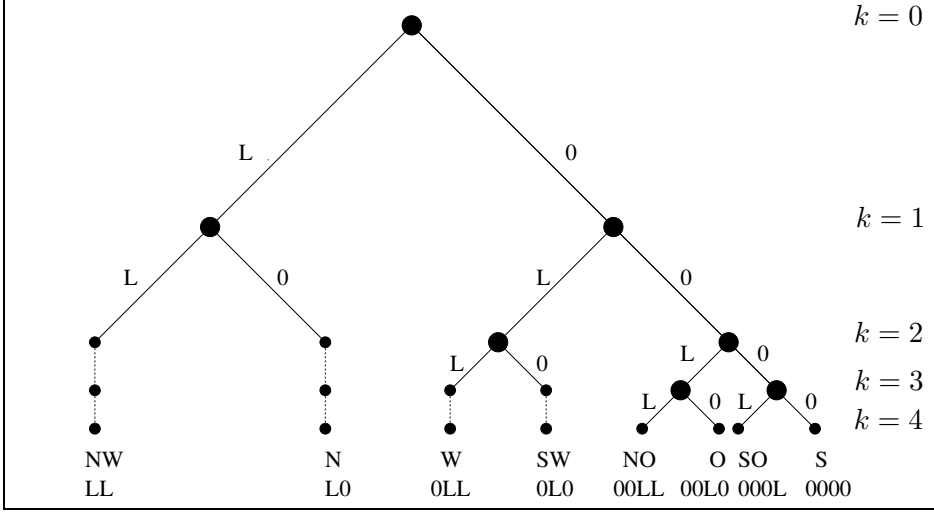
Beispiel: BSP 2B. Windrichtungen in Norddeutschland.

	W	NW	N	NO	O	SO	S	SW
p_i	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{8}$

$\sum_{i=1}^{8} p_i = 1$



Binärbaum zu BSP. 2B



Der so erzeugte Codebaum ist zwar nicht ausgeglichen, aber optimal bezüglich der

Relation $(p_i, |w^i|)$.
Das wahrscheinlichste Ereignis (die wahrscheinlichste Nachricht) hat den kürzesten Code!
Die in beiden Fällen angewendete Entscheidungsstrategie erzeugt einen vollständigen Binärbaum.
Beispiel 2B soll nun näher analysiert werden, um eine grundlegende Aussage über den Codierungsaufwand abzuleiten. Mit Σ_l^k sei der l-te Knoten auf der Ebene k (d.h. mit der Tiefe k) bezeichnet. Jeder dieser Knoten identifiziert Teilmengen von Σ, wobei die Wurzel ($k = 0$) den gesamten Zeichenvorrat Σ und die Blätter die Zeichen aus Σ repräsentieren. Mit $p(\Sigma_l^k)$ wird die Summe der Wahrscheinlichkeiten der Knoten l von Ebene k bezeichnet:

$$p(\Sigma_l^k) = \sum_{z_i \in \Sigma_l^k} p(z_i)$$

Die Knoten Σ_l^k werden solange sukzessive in jeweils zwei Mengen $\Sigma_{l,0}^k$ und $\Sigma_{l,\mathrm{L}}^k$ zerlegt, bis einelementige Teilmengen entstehen. Bei der Zerlegung der Knoten soll gelten:

$$p(\Sigma_{l,0}^k) \approx p(\Sigma_{l,\mathrm{L}}^k)$$

$\Sigma_{l,0}^k$ und $\Sigma_{l,\mathrm{L}}^k$ werden Σ_l^k als Nachfolger zugeordnet, d.h. es ex. ein l', so daß

$$\Sigma_{l,0}^k = \Sigma_{l'}^{k+1}\ ,\ \Sigma_{l,\mathrm{L}}^k = \Sigma_{l'+1}^{k+1} \text{ und } \Sigma_l^k = \Sigma_{l'}^{k+1} \cup \Sigma_{l'+1}^{k+1}$$

gilt.
Unter der Annahme aus BSP 2B, daß die Wahrscheinlichkeiten p_i (negative) Zweierpotenzen sind und so verteilt, daß in jedem Schritt eine Zerlegung eines Knotens Σ_l^k in $\Sigma_l^k = \Sigma_{l,0}^k \cup \Sigma_{l,\mathrm{L}}^k$ mit

$$p(\Sigma_{l,0}^k) = p(\Sigma_{l,\mathrm{L}}^k) = \frac{1}{2} p(\Sigma_l^k)$$

möglich ist, daß also ein bezüglich der Wahrscheinlichkeiten ausgeglichener Codebaum erzeugt wird, gilt:

$$p(\Sigma_l^k) = \frac{1}{2^k}.$$

Also ist jedes Zeichen nach

$$k_{\max}^i = \mathrm{ld}\left(\frac{1}{p_i}\right)$$

(mit $\mathrm{ld} = \log_2$) Schritten als Blatt $\Sigma_l^{k_{\max}^i}$ mit der Wahrscheinlichkeit

$$p_i = p(\Sigma_l^{k_{\max}^i}) = \left(\frac{1}{2}\right)^{k_{\max}^i}$$

repräsentiert.
Beziehungsweise sind für die Identifikation eines mit der Wahrscheinlichkeit p_i auftretenden Zeichens

$$k^i_{\max} = \mathrm{ld}\left(\frac{1}{p_i}\right)$$

Entscheidungsschritte nötig.
Folglich ist ein mit der Wahrscheinlichkeit p_i auftretendes Zeichen $z_i \in \Sigma$ in einem Binärwort $w^i \in \mathbb{B}^*$ der Länge (angegeben in Bit)

$$|w^i| = k^i_{\max} = \mathrm{ld}\left(\frac{1}{p_i}\right) [\mathrm{b}]$$

codierbar.
Im allgemeinen ist aber $k^i_{\max}$ und damit $|w^i|$ nicht ganzzahlig. Dann ist aber stets eine Codierung des Zeichens z_i in einem Binärwort w^i angebbar, so daß

$$|w^i| = l_i \qquad , \; l_i \in \mathbb{N} \; , \; l_i > 0$$

und

$$|l_i - k^i_{\max}| < 1.$$

Beispiel: Erraten einer Zahl zwischen 1 und 50

$N = 50, \;\; |\,\mathrm{ld}(\frac{1}{p_i})| = \mathrm{ld}\,50 = 5.6 \rightarrow l_i = 6[\mathrm{b}]$
Es müssen minimal 6 Entscheidungen getroffen werden.

Nun soll für einen zwischen Sender und Empfänger vereinbarten Zeichenvorrat Σ der mittlere Entscheidungsaufwand des Empfängers zur Entschlüsselung eines Zeichens $z_i \in \Sigma$ mit bekanntem $p(z_i)$ ermittelt werden. Das heißt, es wird der Mittelwert über alle $z_i \in \Sigma$ gebildet.
Demnach bezeichnet H auch die theoretisch zu erwartende mittlere Wortlänge oder den Erwartungswert der Codewortlänge für die $z_i \in \Sigma$. Dabei wird die Summe aller mit ihrer Auftrittswahrscheinlichkeit gewichteten Codewortlängen gebildet.

Geg.: Σ' , $|\Sigma'| = N'$, jedes Zeichen z_i sei mit der Häufigkeit N'_i in Σ' enthalten und es gebe N verschiedene Gruppen gleicher Zeichen z_i, also $\sum\limits_{j=1}^{N} N'_j = N'$. Für jedes Zeichen der Gruppe i sei ein Entscheidungsaufwand $k^i_{\max} = \mathrm{ld}(\frac{1}{p_i})$ erforderlich, $k^i_{\max} \in \mathbb{R}$, $p_i = \frac{N'_i}{N'}$.

Ges.: mittlerer Entscheidungsaufwand H pro Zeichen z_i, bezogen auf den Vorrat Σ' als arithmetisches Mittel.

Lösung:

$$H = \frac{1}{N'} \sum_{i=1}^{N} N'_i k^i_{\max} = \sum_{i=1}^{N} \frac{N'_i}{N'} k^i_{\max} = \sum_{i=1}^{N} p_i \operatorname{ld}\left(\frac{1}{p_i}\right)$$

Definition: (mittlerer Entscheidungsgehalt pro Zeichen)

Die Codierung aller Zeichen $z_i \in \Sigma$, $|\Sigma| = N$, $p(z_i) = p_i$, $\sum_{i=1}^{N} p_i = 1$ durch Binärworte $w^i \in \mathbb{B}^*$, $w^i = <w^i_1 \cdots w^i_{n_i}>$, $w^i_j \in \mathbb{B}$, erfordert einen mittleren Entscheidungsaufwand pro Zeichen

$$H = \sum_{i=1}^{N} \left[p_i \operatorname{ld}\left(\frac{1}{p_i}\right) \right] \text{[b]}.$$

H wird bezeichnet als

- mittlerer Entscheidungsgehalt pro Zeichen
- Information pro Zeichen (oder potentielle Information)
- *Entropie*

Beispiel:

$\Sigma' = \{a, a, a, a, b, b, c, d\}$, $N' = 8$, $N = 4$, $N'_a = 4$, $N'_b = 2$, $N'_c = N'_d = 1$. Dem entspricht ein Zeichenvorrat $\Sigma = \{a, b, c, d\}$, dem nun die Auftrittswahrscheinlichkeiten p_i der Zeichengruppen i zusätzlich als vereinbarte Information hinzugefügt werden: $p_a = \frac{1}{2}$, $p_b = \frac{1}{4}$, $p_c = p_d = \frac{1}{8}$. Mit $k^a_{\max} = \operatorname{ld}\left(\frac{1}{p_a}\right) = \operatorname{ld} 2 - \operatorname{ld} 1 = 1$, $k^b_{\max} = 2$, $k^c_{\max} = k^d_{\max} = 3$ erhält man sowohl für Σ' als auch für Σ eine Entropie $H = 1.75$. Wären hingegen alle 8 Zeichen verschieden, so ergäbe sich $H = 3$. Das mehrfache Auftreten der Zeichen reduziert den Informationsgehalt.

Es sind 3 Fälle bezüglich der a priori Wahrscheinlichkeiten zu unterscheiden:

1. $|\Sigma| = N,\ z_i \in \Sigma,\ p_i = \frac{1}{N}$

   $$\rightarrow H = \operatorname{ld} N \quad \sim \text{im allgemeinen nicht ganzzahlig}$$

   Aber wenn die Zerlegung so erfolgt, daß

   $$\left| |\Sigma^k_{l,\mathrm{L}}| - |\Sigma^k_{l,0}| \right| \leq 1,$$

   dann kann eine Auswahl von N Zeichen stets mit n Alternativentscheidungen getroffen werden, so daß es stets eine Codierung mit der Wortlänge n gibt mit

   $$n - 1 < \operatorname{ld} N \leq n.$$

2. $|\Sigma| = N,\ z_i \in \Sigma,\ N = 2^n,\ p_i = \frac{1}{2^n}$

   $$\rightarrow H = \operatorname{ld} N = n \quad \sim \text{ganze Zahl}$$

3. $|\Sigma| = N\ ,\ z_i \in \Sigma\ ,\ p_i$ beliebig, so daß $\sum_{i=1}^{N} p_i = 1$

   $$H = \sum_{i=1}^{N} p_i \operatorname{ld}\left(\frac{1}{p_i}\right) \leq \operatorname{ld} N$$

Beispiele:

Fall	Beispiel	N	p_i	Baum	H
1	BSP 1A	7	$\frac{1}{7}$	nicht ausg.	$\operatorname{ld} 7 = 2.81$
1	BSP 1B	7	$\frac{1}{7}$	nicht ausg.	$\operatorname{ld} 7 = 2.81$
2	BSP 2A	8	$\frac{1}{8}$	ausgeglichen	$\operatorname{ld} 8 = 3$
3	BSP 2B	8	versch.	nicht ausg., W. ausg.	$2 \cdot \frac{1}{4} \cdot 2 + 2 \cdot \frac{1}{8} \cdot 3 + 4 \cdot \frac{1}{16} \cdot 4 = 2.75$

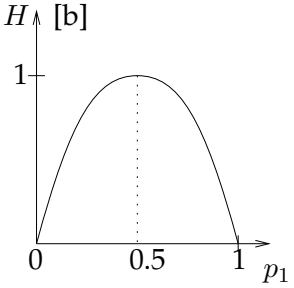
Einige interessante Eigenschaften der Entropiefunktion H:

1. $H = 0 \quad \longleftrightarrow \quad$ Es ex. $z_i \in \Sigma$ mit $p_i = 1$
   Das *sichere Ereignis* z_i bedingt, daß alle übrigen Ereignisse $z_j \in \Sigma,\ j \neq i$ die Wahrscheinlichkeit $p_j = 0$ besitzen.
   Das sichere Ereignis hat die potentielle Information Null und demzufolge auch die Codelänge Null.

2. $H = H_{\max} \quad \longleftrightarrow \quad p_i = \frac{1}{N}$ f.a. $i \in \{1, \ldots, N\}$
   Die Entropiefunktion erreicht ihr Maximum, wenn alle Wahrscheinlichkeiten gleich sind.
   Für nicht ausgeglichene Codebäume gilt $H < H_{max}$.

Beispiel:

$\Sigma = \{x_1, x_2\} \qquad p_2 = 1 - p_1$



Zwei gleichwahrscheinliche Zeichen haben die Information (Entropie) $H = 1$ bit.

---

3. In einem beliebigen Codebaum beträgt die tatsächliche mittlere Wortlänge eines Binärwortes $w_i \in \mathbb{B}^*$

$$L_Z = \sum_{i=1}^{N} p_i l_i \qquad , l_i = |w_i|$$

Beispiele:

	H	L_Z
BSP 1A	2.81	3.0
BSP 1B	2.81	2.86
BSP 2A	3	3
BSP 2B	2.75	3.25

Shannonsches Codierungstheorem (C. Shannon, 1948)

a) Es gilt stets
$$H \leq L_Z < H + 1$$

b) die Differenz $R = L_Z - H$ kann durch geeignete Codierung beliebig klein gemacht werden.

4. Die Differenz
$$R = L_Z - H \qquad \text{mit} \quad 0 \leq R < 1$$
bezeichnet man als *Code-Redundanz*.
Die Redundanz einer Nachricht enthält keine Information. Sie ist ein Maß für überflüssige Entscheidungen.
Sie ist
   - bei der Speicherung von Information oft lästig, weil sie den Code verlängert
   - bei der Übertragung von Information oft unerläßlich, weil sie die Störsicherheit verbessert.

Beispiel: Buchstabieralphabet

Alpha, Beta, Cäsar, ...

Beispiel: Deutsche Sprache

30 Zeichen (26 Buchstaben, 3 Umlaute, 1 Leerzeichen)

a) unabhängig, gleichwahrscheinlich
$$H_1 = \mathrm{ld}\, 30 = 4.9 \text{ bit}$$

b) unabhängig, verschieden wahrscheinlich ($p(''e'') \gg p(''q'')$)
$$H_2 = 4.11 \text{ bit}$$
rel. Redundanz $\delta R = \frac{H_1 - H_2}{H_1} = 0.16 \hat{=} 16\%$

c) Wortcodierung, Satzcodierung. Hierbei sind die Buchstaben der Worte und die Worte als Bestandteile der Sätze aber im Gegensatz zur allgemeinen Annahme der Shannonschen Informationstheorie nicht unabhängig. D.h., die Wahrscheinlichkeiten multiplizieren sich nicht und die Informationen der Bestandteile addieren sich nicht!

---

5. **Huffman-Codierung**
Ein optimaler Code zeichnet sich dadurch aus, daß der mittlere Entscheidungsaufwand pro Zeichen minimal ist.

   Die *Huffman-Codierung* ist ein Verfahren zur rekursiven Konstruktion eines optimalen Codes, der die tatsächliche erreichbare mittlere Wortlänge L_Z minimiert.

Lemma:

Sei $\Sigma = \{z_1, \ldots, z_N\}$ ein Alphabet geordneter Zeichen mit $p_1 \geq p_2 \geq \ldots \geq p_N > 0$. Sei außerdem $\Sigma' = \{z'_1, \ldots, z'_{N-1}\}$ ein Alphabet geordneter Zeichen mit $p'_i = p_i$, $i = 1, \ldots, N-2$, und $p'_{N-1} = p_{N-1} + p_N$. Ist c' mit der Codewortmenge $B' = \{c'(z'_1), \ldots, c'(z'_{N-1})\} \subset \mathbb{B}^*$ ein optimaler Präfixcode für Σ', so bildet c, definiert durch

$$\begin{aligned} c(z_i) &= c'(z'_i)\,,\ i = 1, \ldots, N-2, \\ c(z_{N-1}) &= c'(z'_{N-1}) \circ 0 \\ c(z_N) &= c'(z'_{N-1}) \circ \mathrm{L}, \end{aligned}$$

einen optimalen Präfixcode für Σ.

Das Huffmanverfahren basiert auf der rekursiven Anwendung des Lemma:

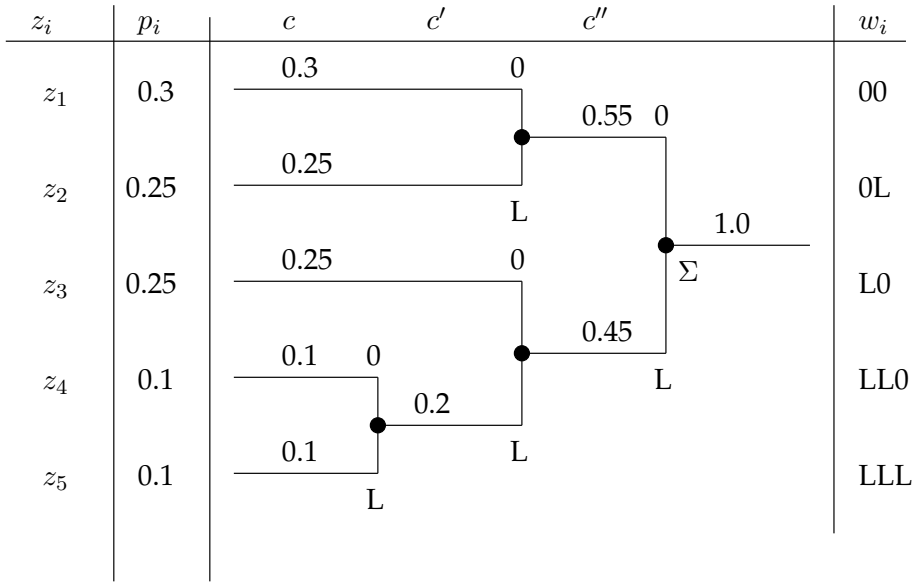
a) Konstruktion einer Folge von Alphabeten dadurch, daß jedes Alphabet als Verkürzung des Vorgängeralphabets gebildet wird, indem die Zeichen mit den beiden kleinsten Wahrscheinlichkeiten zu einem neuen Zeichen zusammengefaßt werden.

$$\Sigma \to \Sigma' \to \Sigma'' \to \ldots \Sigma^T$$

Terminierung: $|\Sigma^T| = 2$

b) Bestimmung des Binärcodes $\{c(z_i) | i = 1, \ldots, N\} \subset \mathbb{B}^*$, ausgehend von der Wurzel des Binärbaumes zu den Blättern z_i.

Beispiel:



$$H = 2.1855 \quad L_Z = 2.2$$

6. **Wortcodierung durch Codeerweiterung**
Hier soll ein Verfahren vorgestellt werden, das auch unter dem Namen *Codeerweiterung* bekannt ist. Dabei werden nicht die Einzelzeichen eines Zeichenvorrats, sondern die aus solchen Zeichen gebildeten Worte der Länge m binär codiert. Es ist interessant festzustellen, daß mit zunehmender Wortlänge der auf das Einzelzeichen bezogene Codierungsaufwand immer näher an den durch die Entropie gegebenen minimalen Codierungsaufwand rückt, also die Redundanz minimal wird.

Definition: (Wortcodierung durch Codeerweiterung)

Unter Wortcodierung verstehen wir die Codierung

$$c_W : \Sigma^m \rightarrow \mathbb{B}^*$$

von Wörtern $z^j = <z_1^j \cdots z_m^j>$ der Länge $|z^j| = m$, dh. $z^j \in \Sigma^m$, $z_i^j \in \Sigma$, $|\Sigma| = N$, $|\Sigma^m| = N^m$, $p(z_i^j) = p_i^j$, in Binärwörter $w^j \in \mathbb{B}^*$, $w^j = <w_1^j \cdots w_n^j>$, $w_i^j \in \mathbb{B}$, $|w^j| = n_j$.

Das Alphabet der Wörter ist also umfangreicher als das der Zeichen. Gleichzeitig reduziert sich die Wahrscheinlichkeit des Auftretens eines Wortes bzw. erhöht sich die durch dieses Wort repräsentierte potentielle Strukturinformation. Mit anderen Worten: Der Binärcode der Wörter ist länger als derjenige der Zeichen. Normiert man aber die tatsächliche Codewortlänge auf die Anzahl m der Zeichen pro Wort, zeigt sich, daß mit zunehmender Wortlänge die auf das Einzelzeichen bezogene Code-Redundanz gegen Null geht.

Ist $q_j = p_1 \cdot \ldots \cdot p_m$ die Verbundwahrscheinlichkeit des Wortes z^j (die Zeichen z_i^j werden als unabhängige Zeichen konkateniert) und ist n_j die Länge des Binärwortes w^j, so ist

$$L_Z = \frac{1}{m} L_W = \frac{1}{m} \sum_{j=1}^{N^m} q_j n_j$$

die mittlere Wortlänge des Binärcodes bei Wortcodierung pro Zeichen $z_i^j \in \Sigma$.
Je größer m gewählt wird, umso besser läßt sich eine Zerlegung von Σ^m in gleichwahrscheinliche Teilmengen erreichen und umso mehr wird $H_{\max}$ angenähert. In den Beispielen 3A und 3B werden die auf Codierung der Einzelzeichen bezogenen Größen H_Z, L_Z und R_Z angegeben. Bezogen auf Worte der Länge m existieren natürlich auch die entsprechenden Größen H_W, L_W und R_W. Wegen $H_W = H(\Sigma^m) = mH(\Sigma)$ und da für Zeichencodierung gilt

$$H(\Sigma) \leq L_Z < H(\Sigma) + 1$$

folgt nach Wortcodierung durch Codeerweiterung

$$H(\Sigma^m) \leq L_W < H(\Sigma^m) + 1$$

bzw.

$$H(\Sigma) \leq \frac{L_W}{m} < H(\Sigma) + \frac{1}{m}.$$

Also schränkt sich die Redundanz $R_Z = L_Z - H(\Sigma)$ auf den maximalen Betrag $\frac{1}{m}$ ein.

Shannonsches Theorem der fehlerfreien Codierung

Die m-te Erweiterung eines Codes erfüllt die Beziehung

$$H(\Sigma) \;\leq\; \frac{L_W}{m} \;<\; H(\Sigma) + \frac{1}{m}$$

.

Mittels Codeerweiterung kann für einen nicht ausgeglichenen Codebaum die Redundanz auf ein beliebiges Maß reduziert werden, also die Anzahl der aufzuwendenden Entscheidungen auf das theoretisch mögliche Minimum reduziert werden.
Da bei Codeerweiterung aber mit wachsendem m die Wahrscheinlichkeiten für die Wörter kleiner werden, werden die Binärcodes der Wörter tatsächlich länger. Also wächst auch der absolute Aufwand zur Entscheidung eines Wortes. Codeerweiterung ist also nicht geeignet, den absoluten Codierungsaufwand zu reduzieren. Lediglich der Anteil redundanter Information, also überflüssiger Entscheidungen, wird, bezogen auf den Wortvorrat, kleiner.
Im Fall eines ausgeglichenen Codebaums (gleich wahrscheinliche Ereignisse, Beispiel 3B) erübrigt sich die Codeerweiterung, die Redundanz ist immer 0.
Völlig anders verhält sich die Codierung natürlicher Sprache, weil in diesem Fall die Zeichenfolge in den Worten nicht unabhängig ist und folglich die Verbundwahrscheinlichkeit sich nicht rein multiplikativ ergibt.
Außerdem werden in der Informatik viele Codes verwendet, die mit den hier besprochenen informationstheoretisch begründeten nichts gemein haben, sondern aus irgendwelchen anderen praktischen Überlegungen entworfen wurden (z.B. Zeichencodes oder Codes der Maschinensprache). In solchen Codes kann man zwar ebenfalls die Größen H, L und R berechnen. Man kommt dann aber auch zu dem Ergebnis, daß $R > 1$ gelten kann.

Beispiel: BSP 3A (verschieden wahrscheinliche Zeichen)

$\Sigma = \{X, Y\}$ $\quad \Sigma^2 = \{XX, XY, YX, YY\}$
$\Sigma^3 = \{XXX, XXY, XYX, YXX, YXY, YYX, XYY, YYY\}$

1. Codierung von Σ:

Zeichen	Wahrsch.	Codierung	Länge Cod.	mittl.Wortl.Zeichen
X	0.8	L	1	0.8
Y	0.2	0	1	0.2

$$\begin{aligned} H_Z &= \sum_i p_i \operatorname{ld}\left(\tfrac{1}{p_i}\right) &&= 0.8 \operatorname{ld} \tfrac{5}{4} + 0.2 \operatorname{ld} 5 \\ &&&= 0.8 \cdot 0.3219 + 0.2 \cdot 2.3219 = 0.7219 \\ L_Z &= 0.8 + 0.2 = 1.0 \\ R_Z &= L_Z - H_Z = 0.2781 \end{aligned}$$

2. Codierung von Σ^2:

Zeichen	Wahrsch.	Codierung	Länge Cod.	mittl.Wortl.Zeichen
XX	0.64	L	1	0.64
XY	0.16	0L	2	0.32
YX	0.16	00L	3	0.48
YY	0.04	000	3	0.12

$$\begin{aligned} L_Z &= \frac{1}{2}(0.64 + 0.32 + 0.48 + 0.12) = 0.78 \\ R_Z &= L_Z - H_Z = 0.058 \end{aligned}$$

3. Codierung von Σ^3:

Zeichen	Wahrsch.	Codierung	Länge Cod.	mittl.Wortl.Zeichen
XXX	0.512	L	1	0.512
XXY	0.128	0LL	3	0.384
XYX	0.128	0L0	3	0.384
YXX	0.128	00L	3	0.384
XYY	0.032	000LL	5	0.16
YXY	0.032	000L0	5	0.16
YYX	0.032	0000L	5	0.16
YYY	0.008	00000	5	0.04

$$\begin{aligned} L_Z &= \frac{1}{3}(0.512 + 3 \cdot 0.384 + 3 \cdot 0.160 + 0.04) = 0.728 \\ R_Z &= L_Z - H_Z = 0.0061 \end{aligned}$$

Beispiel: BSP 3B (gleich wahrscheinliche Zeichen)

$\Sigma = \{X, Y\}$ $\quad \Sigma^2 = \{XX, XY, YX, YY\}$
$\Sigma^3 = \{XXX, XXY, XYX, YXX, YXY, YYX, XYY, YYY\}$

1. Codierung von Σ:

Zeichen	Wahrsch.	Codierung	Länge Cod.	mittl.Wortl.Zeichen
X	0.5	L	1	0.5
Y	0.5	0	1	0.5

$$\begin{aligned} H_Z &= \sum_i p_i \operatorname{ld}\left(\frac{1}{p_i}\right) = 2 \cdot \frac{1}{2} \operatorname{ld} 2 = 1.0 \\ L_Z &= 0.5 + 0.5 = 1.0 \\ R_Z &= L_Z - H_Z = 0 \end{aligned}$$

2. Codierung von Σ^2:

Zeichen	Wahrsch.	Codierung	Länge Cod.	mittl.Wortl.Zeichen
XX	0.25	LL	2	0.5
XY	0.25	L0	2	0.5
YX	0.25	0L	2	0.5
YY	0.25	00	2	0.5

$$\begin{aligned} L_Z &= \frac{1}{2}(4 \cdot 0.5) = 1.0 \\ R_Z &= L_Z - H_Z = 0 \end{aligned}$$

3. Codierung von Σ^3:

Zeichen	Wahrsch.	Codierung	Länge Cod.	mittl.Wortl.Zeichen
XXX	0.125	LLL	3	0.375
XXY	0.125	LL0	3	0.375
XYX	0.125	L0L	3	0.375
YXX	0.125	L00	3	0.375
XYY	0.125	0LL	3	0.375
YXY	0.125	0L0	3	0.375
YYX	0.125	00L	3	0.375
YYY	0.125	000	3	0.375

$$\begin{aligned} L_Z &= \frac{1}{3}(8 \cdot 0.375) = 1.0 \\ R_Z &= L_Z - H_Z = 0 \end{aligned}$$

2.4 Darstellung von Zeichen und Zahlen

Wir haben gelernt:

- Daten sind Repräsentationen für Informationen (Objekte).
- Informationsverarbeitung geschieht durch Datenverarbeitung.
- Datenverarbeitung geschieht durch regelhafte Manipulation von Daten.
- Daten können durch Konkatenationen zu komplexeren Daten (Datenstrukturen) verknüpft werden.

Daten unterschiedlichen Typs (unterschiedlicher Sorten) kommen zur Anwendung. Der Typ eines Datums bestimmt, welche Operationen mit diesem Datum ausführbar sind, denn nicht jedes Datum ist als Operand für jeden Operator geeignet.

Beispiele:

int	Menge der ganzen Zahlen
char	Menge der durch einzelne Zeichen bezeichneten Textsymbole
bool	Menge der Wahrheitswerte 'L'=≪wahr≫, '0'=≪falsch≫

Die formale Verkoppelung der Datentypen (Sorten) mit den auf diesen ausführbaren Grundoperationen in Form von Rechenstrukturen stellt die Basis einer jeden Programmiersprache dar (s. Kapitel 3).

Hier interessiert der Typ insofern, als er die unterschiedlich zu repräsentierenden Daten zu unterscheiden gestattet.

Die Anzahl der Objekte eines bestimmten Typs kann unendlich sein. Ein Objekt muß aber stets durch endlich viele Zeichen aus einem Zeichenvorrat wiederzugeben sein. Tatsächlich können in Rechnern nur endlich viele verschiedene Objekte eines bestimmten Typs repräsentiert werden.

Mit n Bits kann man bei Binärcodierung $N = 2^n$ verschiedene Zustände repräsentieren, die den Dezimalzahlen aus dem Bereich $0, \ldots, (2^N - 1)$ zugeordnet werden können (vgl. Tabelle 2.1).

Bits	Kombinationen	darstellbare Dezimalzahlen	möglicher Bereich
1	0,1	$2 = 2^1$	$0 \cdots 1$
2	00,01,10,11	$4 = 2^2$	$0 \ldots 3$
3	000,001,010,011, 100,101,110,111	$8 = 2^3$	$0 \ldots 7$
⋮			
8		$256 = 2^8$	$0 \ldots 255$
⋮			
16		65536=2^{16}	$0 \ldots 65535$
⋮			
32		4.294.967.296 $= 2^{32}$	$0 \ldots$4.294.967.295
⋮			
64		$\cdots = 2^{64}$	

Tabelle 2.1: Mit n Bits darstellbare Dezimalzahlen.

Definition: (Binärzahl)

Eine Dualzahl $z_2 \in \mathbb{B}^n$ der Länge n Bits heißt Binärzahl, wenn sie im Stellenwertsystem erklärt ist,

$$z_2 = \sum_{i=0}^{n-1} \alpha_i 2^i, \;\; \alpha_i \in \mathbb{B}.$$

Sie vermag $N = 2^n$ verschiedene Zustände eines Sachverhaltes zu repräsentieren.

Diese Zustände können Daten unterschiedlichen Typs oder auch Adressen der Daten im Arbeitsspeicher bzw. Operationen über diesen Daten sein.
Aus Gründen der Ökonomie des Rechnerbaues gilt:

- Es wird stets eine Codierung in fester Wortlänge n benutzt
- n ist stets eine Zweierpotenz

  PC: n=16 oder 32
  Workstation: n=32 oder 64

Jeder Rechner hat seine spezifische Datenwortlänge.

- Der Zugriff auf die Daten erfolgt meist parallel in Gruppen von mindestens 4 Bit Breite.
  Üblich sind folgende Bezeichnungen (für 32-Bit-Rechner):

4 Bit:	Nibble
8 Bit:	Byte
16 Bit:	Halfword
32 Bit:	Word
64 Bit:	Double Word

Die Verwendung von Codes der Basis 8 (*Oktalcode*) und 16 (*Hexadezimalcode*) führt zu einer komprimierten Darstellung:

Beispiel:

$$\underbrace{11\ldots1}_{16}{}_2 = 177777_8 = \mathrm{FFFF}_{16}.$$

Definition: (b-adische Zahlendarstellung)

Seien $b \in \mathbb{N}, b > 1$ eine Basis und $n_b \in \mathbb{N}$ die Stelligkeit einer Zahlendarstellung. Dann ist für $<\alpha_{n_b-1}\ldots\alpha_0>_b,\ \alpha_i \in \Sigma_b$, $i = 0, 1, \ldots, n_b - 1$, $n_b \in \mathbb{N}$, durch

$$z_b = \sum_{i=0}^{n_b-1} \alpha_i b^i$$

eine Zahlendarstellung zur *Basis b* gegeben.

Es sind zwei Fälle möglich:

1. $<\alpha_{n_b-1}\ldots\alpha_0>_b$ ist ein Wort aus Σ_b^*: Darstellung mit variabler Länge ($\alpha_{n_b-1} \neq 0$, d.h. keine führende Nullen, $n_b^{\min} = \lfloor \log_b z_{10} \rfloor + 1$)

   oder

2. $<\alpha_{n_b-1}\ldots\alpha_0>_b$ ist ein Wort aus $\Sigma_b^{n_b}$: Darstellung mit fester Länge (n_b fest vorgegeben, $n_b \geq n_b^{\min}$, führende Nullen zugelassen)

- b= 2 : Binärzahlsystem $\Sigma_2 = \{0, 1\}$
  b= 8 : Oktalzahlsystem $\Sigma_8 = \{0, 1, \ldots, 7\}$
  b=10 : Dezimalzahlsystem $\Sigma_{10} = \{0, \ldots, 9\}$
  b=16 : Hexadezimalzahlsystem $\Sigma_{16} = \{0, \ldots, 9, \mathrm{A}, \ldots, \mathrm{F}\}$

Dezimal $b = 10$	Binär $b = 2$	Oktal $b = 8$	Hexadez. $b = 16$	
0	0000	00	0	1 Bit
1	0001	01	1	
2	0010	02	2	2 Bits
3	0011	03	3	
4	0100	04	4	3 Bits
5	0101	05	5	
6	0110	06	6	
7	0111	07	7	
8	1000	10	8	4 Bits
9	1001	11	9	
10	1010	12	A	
11	1011	13	B	
12	1100	14	C	
13	1101	15	D	
14	1110	16	E	
15	1111	17	F	

Tabelle 2.2: Zahlen $0, \ldots, 15$ im Dezimal-, Binär-, Oktal- und Hexadezimalzahlsystem

- Mit der Wahl der Stellenzahl n_b einer b-adischen Zahl

$$z_b = \sum_{i=1}^{n_b - 1} \alpha_i b^i$$

  trifft man eine Entscheidung für den Umfang darstellbarer Zustände

$$N_b = b^{n_b}$$

- Sind diese Zustände abzählbar (normalerweise gegeben), entspricht dies der Möglichkeit der Codierung von N_b Zahlen z_b, $N_b \in \mathbb{N}$, $N_b = |\{z_{10}\}|$, $z_{10} \in \mathbb{N}$

$$C : z_{10} \longrightarrow z_b \qquad , z_{10}, z_b \in \mathbb{N}$$

  Z.B.: $b = 10$, $n_{10} = 2 \rightsquigarrow N_{10} = 10^2 \ : \ z_{10} \in \{00, 01, \ldots, 99\}$
  Z.B.: $z_{10} = 32 \ = \ 3 \cdot 10^1 + 2 \cdot 10^0$

- Für jede Zahl $x \in \mathbb{Z}$ und jede Zahl $y \in \mathbb{N},\ y{>}0$, gilt auf der Grundlage der Operation "*Division mit Rest*" die folgende Identität: $x = qy + r$ mit q als Quotient und r als Rest für alle b-adischen Zahlendarstellungen

$$x = (x \text{ div } y)\, y + x \bmod y$$

  - *ganzzahlige Division:* $q = x \text{ div } y = \left\lfloor \frac{x}{y} \right\rfloor$
    mit $\lfloor \alpha \rfloor = \max\{z \in \mathbb{Z} |\, z \leq \alpha, \alpha \in \mathbb{R}\}$
    (Untere Gauß-Klammer $\sim$ ganzzahliger Anteil der Division)
  - *Rest der ganzzahligen Division:* $r = x \bmod y = x - \left\lfloor \frac{x}{y} \right\rfloor y, \qquad 0 \leq x \bmod y < y$

  1. Es gilt $z_b = z_b \bmod N_b$
     $\Rightarrow$ Jede n_b-stellige b-adische Zahl ist gleich dem Rest ihrer ganzzahligen Division durch N_b.
     $\hookrightarrow$ Eine Zahl mit $n{>}n_b$ Stellen wird nur bezgl. der letzten n_b Stellen darstellbar sein. Das entspricht der Darstellung auf einem *Zahlenkreis* mit N_b Positionen.
     Z.B. $b = 10, \quad n_{10} = 2, \quad$ aber $\ n = 3$
     $432 \rightarrow 32 \quad \sim$ Die Zahl 432 wird auf die Zahl 32 abgebildet.
  2. Es gilt $\alpha_i = (z_b \text{ div } b^i) \bmod b$
     Die Koeffizienten α_i jeder b-adischen Zahl ergeben sich als Rest der ganzzahligen Division $\left\lfloor \frac{z_b}{b^i} \right\rfloor$ bzgl. der Basis b.

$$\alpha_i = \left\lfloor \frac{z_b}{b^i} \right\rfloor - \left\lfloor \frac{\left\lfloor \frac{z_b}{b^i} \right\rfloor}{b} \right\rfloor b$$

     Z.B.: $z_{10} = 32 = 3 \cdot 10^1 + 2 \cdot 10^0$
     $\alpha_0 = \left\lfloor \frac{32}{1} \right\rfloor \bmod 10 = 32 - \left\lfloor \frac{32}{10} \right\rfloor \cdot 10 = 32 - 3 \cdot 10 = 32 - 30 = 2$
     $\alpha_1 = \left\lfloor \frac{32}{10} \right\rfloor \bmod 10 = 3 - \left\lfloor \frac{3}{10} \right\rfloor \cdot 10 = 3 - 0 \cdot 10 = 3 - 0 = 3$

Hieraus leitet sich ein besonders einfaches Verfahren der Konvertierung von Codes zwischen Dual-, Oktal- und Hexadezimalsystem ab.

1. $z_8, z_{16} \longrightarrow z_2$: jede Oktal-/Hexziffer α_i^b wird für sich in die entsprechende Dualzahl umgewandelt
2. $z_2 \longrightarrow z_8, z_{16}$: Zusammenfassen der Dualziffern α_i^2 zu Gruppen von 3 bzw. 4 Bit Breite und Konvertierung dieser Gruppe in den Zielcode

(Das Verfahren gilt allgemein, wenn Quell- und Zielbasis zueinander durch eine Potenz im Verhältnis stehen: $q = p^{\pm n}$, z.B. 10 und 100).

Beispiele:

zu 1. $123_8 \longrightarrow 001\,010\,011_2$
$4A8_{16} \longrightarrow 0100\,1010\,1000_2$

zu 2. $10000101_2 \longrightarrow 205_8$ oder 85_{16}

2.4.1 Zeichencodes

Zeichencodes werden zur Repräsentation von Texten benötigt:

- Eingaben über die Tastatur
- Programme ("Quellprogramme")
- Datenbanken.

Zeichen werden ebenfalls binär repräsentiert.
Da der Umfang des Alphabets der Schriftsprache gering ist (Deutsch $\lesssim$ 30 Zeichen), enthalten Zeichencodes mehr Information:

1. Buchstaben (groß + klein)
2. Ziffern (1...9)
3. mathematische Operationssymbole (Klammern, Vergleiche, arithm. Operationen)
4. nicht darstellbare Steuerzeichen, z.B. dient CR (carriage return - Wagenrücklauf) der Formatierung (Zeilenabschluß) einer Eingabe über Tastatur

Es genügen 7 Bit zur Codierung des Zeichenvorrats einer Sprache. Es existieren verschiedene standardisierte Codes:

- "ISO 7-Bit Codes":

  ISO 8859-1: westeuropäische Sprachen
  ISO 8859-2...4: nord-, ostmittel, südosteur. Sprachen
  ISO 8859-5...9: kyrillisch, griechisch, hebräisch, arabisch

- ASCII-Code (American Standard Code for Information Interchange) $\hat{=}$ DIN 66003

			b8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
			b7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
			b6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	HEX
			b5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
b4	b3	b2	b1		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	▼
0	0	0	0	0	SONDERZEICHEN		SP	0	@	P	`	p				°	À	Ð	à	ð	0
0	0	0	1	1			!	1	A	Q	a	q		‘	¡	±	Á	Ñ	á	ñ	1
0	0	1	0	2			"	2	B	R	b	r	‚	’	¢	²	Â	Ò	â	ò	2
0	0	1	1	3			#	3	C	S	c	s	*ƒ*	“	£	³	Ã	Ó	ã	ó	3
0	1	0	0	4			$	4	D	T	d	t	„	”	¤	´	Ä	Ô	ä	ô	4
0	1	0	1	5			%	5	E	U	e	u	…	•	¥	µ	Å	Õ	å	õ	5
0	1	1	0	6			&	6	F	V	f	v	†	–	¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	7			'	7	G	W	g	w	‡	—	§	·	Ç	×	ç	÷	7
1	0	0	0	8			(	8	H	X	h	x	ˆ	˜	¨	¸	È	Ø	è	ø	8
1	0	0	1	9			)	9	I	Y	i	y	‰	™	©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z	Š	š	ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{	‹	›	«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|	Œ	œ	¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			-	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o			Ÿ	¯	¿	Ï	ß	ï	ÿ	F
HEX →					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Beispiele: ASCII-Code

'G' = 0100 $0111_2 = 47_{16} = 107_8$
'H␣A' = 0100 1000 0010 0000 0100 0001_2
= 48 20 41_{16}= 22020101_8
'32' =0011 0011 0011 0010_2=33 32_{16}=31462_8

Texte werden byteweise orientiert gespeichert: 1 Byte pro Zeichen

Zeichensequenzen werden als Zeichenketten in logisch zusammenhängenden Strukturen (String) gespeichert:

Beispiele:

Darstellung mit fester Länge (8 Bytes):

H	A	L	L	O	!	␣	␣

H	A	L	L	O	␣	P	E

Darstellung mit variabler Länge:

H	A	L	L	O	␣	P	E	T	E	R	!	NUL			

mit <NUL> als eindeutig festgelegtem Abschlußzeichen.

2.4.2 Darstellung von Zahlen

In der Mathematik kennt man unterschiedliche Zahlbereiche, die durch eine Enthaltenseinsrelation in Beziehung stehen:

$\mathbb{N}$	natürliche Zahlen	$\mathbb{N} = \{0, 1, 2, \ldots\}$
$\cap$		
$\mathbb{Z}$	ganze Zahlen	$\mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\}$
$\cap$		
$\mathbb{Q}$	rationale Zahlen	z.B. $\frac{3}{2} = 1.5$, $\frac{82}{7} = 11.\overline{714285}$
$\cap$		
$\mathbb{R}$	reelle Zahlen	z.B. $\sqrt{2},\ \pi,\ e$
$\cap$		
$\mathbb{C}$	komplexe Zahlen	$c = a + jb,\ a, b \in \mathbb{R},\ j := \sqrt{-1}$

Sie gestatten, die Klassen zu lösender arithmetischer Gleichungen zu erweitern, z.B.:

$\mathbb{N}$	$a + x = b,\ a \leq b$
$\mathbb{Z}$:	$a + x = b$
$\mathbb{Q}$:	$a \cdot x = b,\ a \neq 0$
$\mathbb{R}$:	$a + x^2 = b,\ a \leq b$
$\mathbb{C}$:	$a + x^2 = b$

Die beschränkte Stellenzahl der Computer hat grundlegende Konsequenzen für die in der Informatik zugelassenen Zahlbereiche:

- natürliche Zahlen sind bis zu einer Obergrenze exakt darstellbar
- ganze Zahlen halbieren diesen Darstellungsbereich
- rationale Zahlen sind mit endlicher Genauigkeit darstellbar
- reelle Zahlen sind prinzipiell nur durch Approximationen darstellbar
- komplexe Zahlen werden als Paare reeller Zahlen dargestellt

2.4.2.1 Darstellung natürlicher Zahlen

Die Vorgabe von n Bits für die Binärcodierung einer natürlichen Zahl schränkt den repräsentierbaren Bereich auf das Intervall $[0, \ldots, 2^n - 1] \subset \mathbb{N}$ ein. Zur Konvertierung der Darstellung natürlicher Zahlen ist im allgemeinen Fall, daß Quell- und Zielbasis q bzw.

p nicht durch eine Potenz im Verhältnis stehen ($q \neq p^{\pm m}$), die Darstellung natürlicher Zahlen nach dem *Horner-Schema* von Vorteil:

$$z_b = \sum_{j=0}^{n-1} \alpha_j b^j = (\ldots((\alpha_{n-1}b + \alpha_{n-2})b + \alpha_{n-3})b + \ldots + \alpha_1)b + \alpha_0$$

Ausgehend von der Identität

$$z_b = (z_b \text{ div } b)b + z_b \bmod b$$

beruht das Horner-Schema auf der rekursiven Anwendung der Division durch die Basis b und der Abspaltung der Koeffizienten α_i zu jedem Rekursionsschritt i. Auf diese Art und Weise wird, beginnend mit der niedrigsten Potenz (Null) in der Stellenwertdarstellung der Zahl z_b, deren Umwandlung in das Horner-Schema durchgeführt.

Initialisierung:

$$\begin{aligned} z_b &= z_{(0)} = (z_{(0)} \text{ div } b)b + z_{(0)} \bmod b = z_{(1)}b + \alpha_0 \\ &= \sum_{j=0}^{n-1} \alpha_j b^j = (\sum_{j=1}^{n-1} \alpha_j b^{j-1})b + \alpha_0 \\ \alpha_0 &= z_{(0)} \bmod b = z_{(0)} - z_{(1)}b \end{aligned}$$

Rekursion: für $0 < i \leq n-1$ ist das Ergebnis der ganzzahligen Division

$$\begin{aligned} z_{(i)} &= z_{(i-1)} \text{ div } b = (z_{(i)} \text{ div } b)b + z_{(i)} \bmod b = z_{(i+1)}b + \alpha_i \\ &= \sum_{j=i}^{n-1} \alpha_j b^{j-i} = (\sum_{j=i+1}^{n-1} \alpha_j b^{j-i-1})b + \alpha_i \\ z_{(n)} &= 0 \\ \alpha_i &= z_{(i)} \bmod b = z_{(i)} - z_{(i+1)}b = z_{(i-1)} \text{ div } b - z_{(i+1)}b \end{aligned}$$

also:

$$\begin{aligned} \alpha_1 &= z_{(1)} \bmod b = (z_{(0)} \text{ div } b) \bmod b \\ \alpha_2 &= z_{(2)} \bmod b = (z_{(1)} \text{ div } b) \bmod b = ((z_{(0)} \text{ div } b) \text{ div } b) \bmod b \\ &\vdots \\ \alpha_i &= z_{(i)} \bmod b = (\ldots((z_{(0)} \text{ div } b) \text{ div } b)\ldots \text{ div } b) \bmod b \end{aligned}$$

Beispiel:

$$
\begin{aligned}
z_{10} &= 2345 \\
&= 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 \\
z_{(0)} &= (2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0)10 + \quad \underline{5} \\
z_{(1)} &= 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 \\
&= (2 \cdot 10^1 + 3 \cdot 10^0)10 + \quad \underline{4} \\
z_{(2)} &= 2 \cdot 10^1 + 3 \cdot 10^0 \\
&= (2 \cdot 10^0)10 + \quad \underline{3} \\
z_{(3)} &= 2 \cdot 10^0 = \quad \underline{2}
\end{aligned}
$$

$$
z_{10} = ((2 \cdot 10 + 3)10 + 4)10 + 5
$$

Für die Konvertierung aus einer Quellsystem-Darstellung in eine Zielsystem-Darstellung gibt es verschiedene Verfahren: Rechnen im *Quellsystem* oder im *Zielsystem*, je nachdem welches Verfahren einfacher ist.

A) Divisionsmethode (Quellverfahren)

Notationen:

$z_q = \sum_{i=0}^{n_q-1} \alpha_i q^i$ ist Zahl im Quellsystem-Darstellung

$z_p = \sum_{i=0}^{n_p-1} \beta_i p^i$ ist Zahl im Zielsystem-Darstellung

$z_{(i)}$ Ergebnis des i-ten Schrittes der ganzzahligen Division von z_q durch die Basis des Zielsystems p_q, d.h. in Quellsystemdarstellung.

Aus dem Horner-Schema leitet sich folgendes Schema für die Berechnung der Koeffizienten β_i der Zielsystemdarstellung ab:

$$
\begin{array}{lcllcl}
z_{(0)} & = & z_q & \beta_0 & = & z_{(0)} \bmod p_q = z_q \bmod p_q \\
z_{(1)} & = & z_{(0)} \operatorname{div} p_q & \beta_1 & = & z_{(1)} \bmod p_q = (z_q \operatorname{div} p_q) \bmod p_q \\
z_{(2)} & = & z_{(1)} \operatorname{div} p_q & \beta_2 & = & z_{(2)} \bmod p_q = ((z_q \operatorname{div} p_q) \operatorname{div} p_q) \bmod p_q \\
\vdots & & & \vdots & & \\
z_{(i)} & = & z_{(i-1)} \operatorname{div} p_q & \beta_i & = & z_{(i)} \bmod p_q
\end{array}
$$

Das heißt, sukzessive Division von $z_{(i)}$ durch p liefert die Koeffizienten β_i als Divisionsreste. Die Division erfolgt solange, bis $z_{(i)} = 0$, d.h. bis die Zahl in Quellsystem-Darstellung durch fortgesetzte Division völlig aufgebraucht ist.

Beispiel: Konvertierung 4711_{10} in 1001001100111_2

$$
\begin{array}{lcl@{\qquad}lcl}
z_{(0)} & = & 4711_{10} & \beta_0 & = & z_{(0)} \bmod 2_{10} = 1_{10} = 1_2 \\
z_{(1)} & = & z_{(0)} \operatorname{div} 2_{10} = 2355_{10} & \beta_1 & = & z_{(1)} \bmod 2_{10} = 1_{10} = 1_2 \\
z_{(2)} & = & z_{(1)} \operatorname{div} 2_{10} = 1177_{10} & \beta_2 & = & z_{(2)} \bmod 2_{10} = 1_{10} = 1_2 \\
z_{(3)} & = & 588_{10} & \beta_3 & & = 0_2 \\
z_{(4)} & = & 294_{10} & \beta_4 & & = 0_2 \\
z_{(5)} & = & 147_{10} & \beta_5 & & = 1_2 \\
z_{(6)} & = & 73_{10} & \beta_6 & & = 1_2 \\
z_{(7)} & = & 36_{10} & \beta_7 & & = 0_2 \\
z_{(8)} & = & 18_{10} & \beta_8 & & = 0_2 \\
z_{(9)} & = & 9_{10} & \beta_9 & & = 1_2 \\
z_{(10)} & = & 4_{10} & \beta_{10} & & = 0_2 \\
z_{(11)} & = & 2_{10} & \beta_{11} & & = 0_2 \\
z_{(12)} & = & 1_{10} & \beta_{12} & & = 1_2 \\
z_{(13)} & = & 0_{10}
\end{array}
$$

Die Konvertierung vom Dezimal- in das Dualsystem erfolgt durch fortgesetzte Division der $z_{(i)}$ durch 2, bis das Divisionsergebnis Null erreicht ist. Die Koeffizienten im Dualsystem ergeben sich durch die Prüfung, ob das jeweilige Divionsergebnis gerade oder ungerade ist. Der Code im Zielsystem ergibt sich als Folge $<\beta_{n_b-1} \cdots \beta_1 \beta_0>$. Das Verfahren läßt sich kompakt darstellen:

Beispiel: $49_{10} = 110001_2$

$$
\begin{array}{rrcl}
49 \operatorname{div} 2 = & 24 & \mathrm{R} & 1 \\
24 \operatorname{div} 2 = & 12 & \mathrm{R} & 0 \\
12 \operatorname{div} 2 = & 6 & \mathrm{R} & 0 \\
6 \operatorname{div} 2 = & 3 & \mathrm{R} & 0 \\
3 \operatorname{div} 2 = & 1 & \mathrm{R} & 1 \\
1 \operatorname{div} 2 = & 0 & \mathrm{R} & 1
\end{array}
$$

B) Multiplikationsmethode (Zielverfahren)

Die Quellziffern werden in ihrer Zielsystem-Darstellung entsprechend dem Hornerschema sukzessive von links nach rechts mit der Quellbasis (in der Zielsystemdarstel-

lung) multipliziert und die jeweils nächste Quellziffer der folgenden Stelle hinzuaddiert.
D.h., in der Hornerschema-Darstellung des Quellsystems werden die Klammerausdrücke von innen nach außen gleichzeitig in das Zielsystem konvertiert und aufgelöst.

Beispiel: Konvertierung BCDEF$_{16}$ in 773615_{10}

$$\begin{aligned} \text{BCDEF}_{16} = z_q &= (((\text{B}_{16} \cdot 10_{16} + \text{C}_{16}) \cdot 10_{16} + \text{D}_{16}) \cdot 10_{16} + \text{E}_{16}) \cdot 10_{16} + \text{F}_{16} \\ &= (((11_{10} \cdot 16_{10} + 12_{10}) \cdot 16_{10} + 13_{10}) \cdot 16_{10} + 14_{10}) \cdot 16_{10} + 15_{10} \\ &= 773615_{10} = z_p \end{aligned}$$

Beispiel: Ein weiteres Beispiel mit anderer Zielbasis, z.B. $123_4 \rightarrow z_8$

$$\begin{aligned} z_8 &= (1_4 \cdot 10_4 + 2_4) \cdot 10_4 + 3_4 \\ &= (1_8 \cdot 4_8 + 2_8) \cdot 4_8 + 3_8 \\ &= 6_8 \cdot 4_8 + 3_8 \\ &= 30_8 + 3_8 = 33_8 \end{aligned}$$

2.4.2.2 Darstellung ganzer Zahlen

Zur Darstellung ganzer Zahlen $z \in \mathbb{Z}$ durch ein Binärwort der Breite n Bit wird der darstellbare Bereich von 2^n Zahlen symmetrisch um die Null verteilt. Damit halbiert sich etwa für gegebenes n die maximale darstellbare natürliche Zahl $z_{\max} \in \mathbb{N}$, so daß der frei werdende Darstellungsbereich für die negativen Zahlen verwendet wird.
Negative Zahlen werden durch die *Komplementbildung*

$$K(z) := -z$$

für n-stellige Binärzahlen $z_2 \in \mathbb{B}^n$, $0 \leq |z_{10}| < N = 2^n$, gebildet. Es kommen zwei Methoden der Komplementbildung zur Anwendung:

1. Stellen- oder *Einerkomplement*

$$K_1(z) = (2^n - 1) - z$$

2. echtes oder *Zweierkomplement*

$$K_2(z) = 2^n - z.$$

Beide Methoden orientieren sich an der Zielstellung, die Architektur des Rechenwerkes einfach zu halten.
Eigenschaften des Einerkomplements:

- Zahlbereich: $|z| \leq 2^{n-1} - 1$ (symmetrisch)
- wegen
$$-z = (2^n - 1) - z = \underbrace{11\ldots1}_{n}{}_2 - z$$
erhält man das Einerkomplement durch stellenweises Invertieren aller Bits von z.
- es exisitieren zwei Darstellungen der Null:
$$\begin{aligned} -0_{10} &= 11\ldots1_2 = <1_{n-1}1_{n-2}\ldots1_0>_2 \\ +0_{10} &= 00\ldots0_2 = <0_{n-1}0_{n-2}\ldots0_0>_2 \end{aligned}$$
- Ursprung des Namens:
$$z + (-z) = \langle\alpha_{n-1}\alpha_{n-2}\ldots\alpha_0\rangle_2 \; + \; \langle\overline{\alpha_{n-1}}\,\overline{a_{n-2}}\;\ldots\overline{\alpha_0}\rangle_2 = 11\ldots1_2$$
d.h. für die einzelnen Bits gilt
$$\alpha_i + \overline{\alpha_i} = 1$$

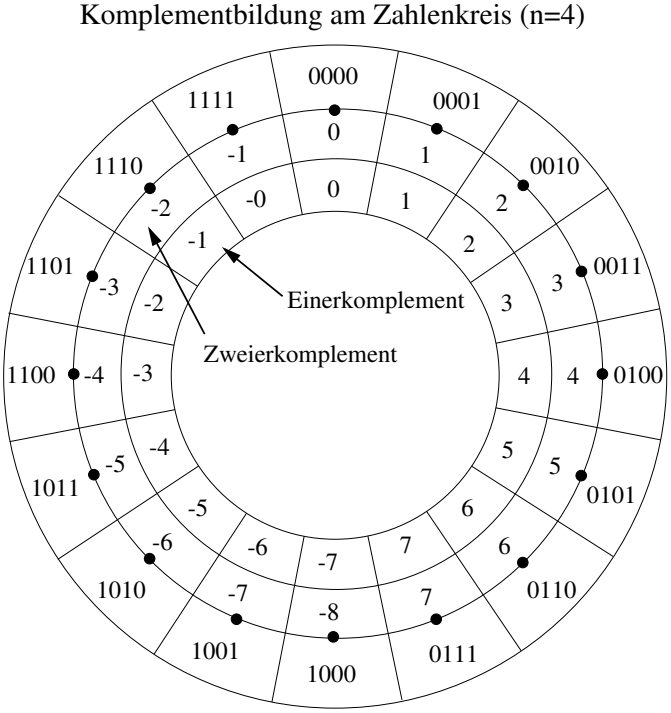
Eigenschaften des Zweierkomplements:

- Zahlbereich: $-2^{n-1} \leq z \leq 2^{n-1} - 1$ (asymmetrisch)
- das Zweierkomplement erhält man aus dem Einerkomplement durch Addieren einer Eins (modulo 2^n)
$$K_2(z) = K_1(z) + 1$$
- es existiert eine eindeutige Darstellung der Null:
$$0_{10} = 00\ldots0_2 = <0_{n-1}0_{n-2}\ldots0_0>_2$$
Deshalb wird das Zweierkomplement am häufigsten angewendet.

Es gilt folgender Zusammenhang zwischen den beiden Komplementbildungen:

$$\begin{aligned} K_2(z) = 2^n - z &= 2^n - \sum_{i=0}^{n-1}\alpha_i 2^i = \underbrace{1 + \sum_{i=0}^{n-1} 2^i}_{=2^n} - \sum_{i=0}^{n-1}\alpha_i 2^i \\ &= 1 + \sum_{i=0}^{n-1}(1-\alpha_i)2^i = 1 + K_1(z) \end{aligned}$$

Beispiel: Asymmetrische 2er-Komplement-Verteilung der Binärzahlen auf dem Zahlenkreis:



Die Komplementdarstellung einer Zahl ist mehrdeutig.

Beispiel:

$$\begin{aligned} n = 4 \quad z_{10} &\in \{0, \ldots, 7\},\ N = 2^4 = 16 \\ z_2 &\in \{0000, \ldots, 0111\} \end{aligned}$$

$5_{10} = 0101_2$
$K_2(5) = K_1(5) + 1 = 1010_2 + 1 = 1011_2 = 11_{10}$
die Dualdarstellung von -5 stimmt also mit der Darstellung von 16-5=11 überein!

Die Mehrdeutigkeit kann behoben werden durch folgende Feststellung:

$$\begin{aligned} \alpha_{n-1} = 0 &\quad \leftrightarrow \quad z \text{ ist positive Zahl} \\ \alpha_{n-1} = 1 &\quad \leftrightarrow \quad z \text{ ist negative Zahl} \end{aligned}$$

Deshalb kann die Zahl 11_{10} im vorigen Beispiel nicht mit 4 Bit dargestellt werden.

In jedem b-adischen Zahlensystem sind "$(b-1)$-Komplement" und "b-Komplement" berechenbar, vorausgesetzt, daß sie sich auf eine beliebige, aber feste Stellenzahl beziehen.

Die vier Grundrechenarten im b-adischen Zahlensystem werden prinzipiell wie im Dezimalsystem ausgeführt. Eine Besonderheit entsteht aus der Beschränkung auf eine feste Wortlänge.

Zu beachten ist, daß bei der stellenweisen Verrechnung (Addition, Multiplikation, Subtraktion wird auf die Addition mit dem Komplement zurückgeführt) eine Überschreitung des Wertebereiches der Koeffizienten (modulo b) oder der darstellbaren Zahlenbereiche (modulo N bei natürlichen Zahlen, modulo $2^{n-1}-1$ bei ganzen Zahlen) auftreten kann.

Addition zweier b-adischer Zahlen:
$z = x + y$ mit $x = \alpha_{n-1}\alpha_{n-2}\ldots\alpha_0,\ y = \beta_{n-1}\ldots\beta_0,\ z = \gamma_{n-1}\ldots\gamma_0,\ \alpha_i, \beta_i, \gamma_i \in \Sigma_b$ für $i = 0, \ldots, n-1$. Die Addition erfolgt durch sukzessives Addieren der Koeffizienten α_i und β_i, beginnend bei $i = 0$. Bei der Addition der Koeffizienten an der Stelle i ist der ggf. an der Stelle $i-1$ aufgetretene Übertrag u_i zu berücksichtigen. Die Addition der Komponenten erfolgt nach dem Schema

$$\gamma_i = (\alpha_i + \beta_i + u_i) \bmod b,$$

für den Übertrag gilt

$$u_i = (\alpha_{i-1} + \beta_{i-1} + u_{i-1}) \operatorname{div} b, \text{ mit } u_0 = 0.$$

Beispiele:

Dualsystem:

	1	0	0	1	1	α_i
	0	1	0	1	1	β_i
+	0	0	1	1	0	u_i
	1	1	1	1	0	γ_i

Hexadezimalsystem:

	3	7	F
	0	C	9
+	1	1	0
	4	4	8

Definition: (Überlauf)

Der Überlauf stellt ein Überschreiten des darstellbaren Zahlenbereiches dar.

Bei der Addition natürlicher Zahlen wird ein Überlauf durch den Übertrag $u_n = 1$ erzeugt. Er wird gewöhnlich als Berechnungsfehler signalisiert.

2.4.2.3 Darstellung rationaler Zahlen

Rationale Zahlen sind Bruchzahlen. Eine rationale Zahl steht für die Gesamtheit aller durch Erweiterung eines Bruches erzeugbaren Brüche. Insbesondere kann eine rationale Zahl auch bezüglich einer Basis b als b-adische Bruchzahl geschrieben werden.
Bei der Festlegung der Länge n dieser Repräsentation ist zwangsläufig damit zu rechnen, daß rationale Zahlen nur approximiert werden können.

Beispiele:

$\left.\begin{array}{l} \frac{1}{12} = 0.08\overline{3}_{10} \\ \frac{82}{7} = 11.\overline{714285}_{10} \end{array}\right\}$ Dezimalbrüche

$\frac{1}{16} = 2_{10}^{-4} = 0.0001_2$ Dualbruch

$\frac{5}{16} = 5 \cdot 16_{10}^{-1} = 0.5_{16}$ Hexadezimalbruch

$\frac{5}{16} = 0.3125_{10} = 0.0101_2 = 0.24_8 = 0.5_{16}$

Definition: (b-adische Bruchzahl)

Eine b-adische rationale Zahl $z \in \Sigma_b^s.\Sigma_b^{-r} \subset \mathbb{Q},\ |z| < b^s$,

$$z = \sum_{i=0}^{s-1} \alpha_i b^i . \sum_{i=-r}^{-1} \alpha_i b^i,$$

$\alpha_i \in \Sigma_b,$ entsteht aus der Konkatenation (angezeigt durch einen Punkt) zweier b-adischer Zahlen mit positivem bzw. negativem Exponenten. Hierbei steht der Punkt rechts neben dem Koeffizienten α_0.

Die Umkehrung der Divisionsmethode ergibt ein einfaches Verfahren zur Konvertierung b-adischer Bruchzahlen:

Der ganzzahlige Teil der Zahl wird unabhängig vom gebrochenen Teil konvertiert (z.B. Divisionsmethode).
Der gebrochene Teil der Zahl wird konvertiert, indem folgende Schritte iterativ ausgeführt werden:

1. Multiplikation des gebrochenen Zahlteils mit der Zielbasis

2. Auslesen des hierdurch entstehenden ganzzahligen Zahlteils und Verwendung als Codeziffer des gebrochenen Anteils im Zielcode (in direkter Reihenfolge)

Das Verfahren terminiert, wenn der gebrochene Zahlteil Null ergibt.

Beispiele:

a: $\frac{5}{16} \rightarrow (0.3125_{10} - 0) \cdot 2 \rightarrow (0.625_{10} - 0) \cdot 2 \rightarrow (1.25_{10} - 1) \cdot 2 \rightarrow (0.5_{10} - 0) \cdot 2 \rightarrow 1.0_{10}$
$\quad 0. \quad 0 \quad 1 \quad 0 \quad 1$
$\rightarrow 0.3125_{10} = 0.0101_2$

b: $0.3125_{10} \rightarrow (0.3125_{10} - 0) \cdot 8 \rightarrow (2.5_{10} - 2_{10}) \cdot 8 \rightarrow 4.0_{10}$
$\quad 0. \quad 2 \quad 4$
$\rightarrow 0.3125_{10} = 0.24_8$

c: $0.3125_{10} \rightarrow (0.3125_{10} - 0) \cdot 16 \rightarrow 5.0_{10}$
$\quad 0. \quad 5$
$\rightarrow 0.3125_{10} = 0.5_{16}$

d: $16.125_{10} = 10000.001_2$

Die Struktur der Darstellung einer rationalen Zahl muß vereinbart werden. Gebräuchlich sind zwei Darstellungsformen:

1. Festpunkt-Darstellung

2. Gleitpunkt-Darstellung

Definition: (Festpunkt-Darstellung)

Bei der Festpunkt-Darstellung einer b-adischen rationalen Zahl der Länge $n = s + r$ führt die Konvention

$$z = \sum_{i=0}^{s-1} \alpha_i b^i . \sum_{i=-r}^{-1} \alpha_i b^i = z' \cdot b^{-r}$$

mit

$$z' = \sum_{i=0}^{n-1} \alpha_i b^i$$

zu einer Standard-Darstellung durch das Paar (z', r).

Beispiele:

$16\frac{1}{8} = 16.125_{10}$

$$\begin{array}{llllll} b=10, & n=5 & , & r=3 & : & z' = 16125_{10} \quad \sim z_{10} = 16.125_{10} \\ b=2, & n=8 & , & r=3 & : & z' = 10000001_2 \quad \sim z_2 = 10000.001_2 \\ b=8, & n=3 & , & r=1 & : & z' = 201_8 \quad \sim z_8 = 20.1_8 \\ b=16, & n=3 & , & r=1 & : & z' = 102_{16} \quad \sim z_{16} = 10.2_{16} \end{array}$$

Die Festpunkt-Darstellung wird z.B. in der Prozeßsteuerung (hohe Genauigkeit, wenige komplexe arithmetische Operationen) und im Finanzwesen (Kontrolle der Rundungsfehler) angewandt.

Als Nachteil der Festpunkt-Darstellung ist der kleine darstellbare Zahlenbereich zu nennen. Insbesondere für wissenschaftliche Rechnungen wird ein wesentlich größerer darstellbarer Zahlenbereich bei einer relativ kleinen Zahl signifikanter Stellen benötigt.

Definition: (Gleitpunkt-Darstellung)

Die Gleitpunktdarstellung einer rationalen Zahl erfolgt in der Form

$$z = \pm M \cdot B^{\pm e},$$

wobei M die Mantisse, B die Basis der Darstellung und e deren Exponent sind. Es gilt $B \geq 2,\ 0 \leq M < 1$. Die Gleitpunktzahl heißt normalisiert, wenn gilt

$$\frac{1}{B} \leq M < 1 \text{ oder } z = 0,$$

andernfalls unnormalisiert.

Anmerkung: Man unterscheidet im allgemeinen 3 Formen der Normalisierten Gleitpunktdarstellung; mit $0 \leq d_i \leq B, i = 1...r$ mit Mantissenlänge r und $d_1 \neq 0$ unterscheidet man

- **1. Form:** $M = 0.d_1 d_2 d_3 ... d_r$
- **2. Form:** $M = d_1 . d_2 d_3 ... d_r$
- **3. Form:** $M = d_1 d_2 d_3 ... d_r .0$

Unsere hier gegebene Definition der Gleitpunkt-Darstellung bezieht sich auf die 1. Form, auf welche wir uns auch beziehen werden, sofern nicht anders vermerkt. Eine Ausnahme stellt z.B. der später eingeführte IEEE 754-Standard, welcher im Allgemeinen mit der 2., zum Teil auch in der 3. Form gegeben wird.

Die Gleitpunkt-Darstellung wurde von K. Zuse als halblogarithmische Darstellung eingeführt. Rechnerintern kann als Basis B 2, 8 oder 16 verwendet werden. Der Anwender programmiert meist in der Basis 10, die Wandlung bezüglich der vom Rechner verwendeten Basis und die Normalisierung erfolgen intern.

Die Normalisierung führt dazu, daß der Gleitpunkt links von der ersten (höchstwertigen) Stelle der Mantisse liegt und daß diese ungleich Null ist. Die Mantisse ist also eine nichtnegative reelle Zahl $M = \sum_{i=1}^{r} \alpha_i \mathrm{B}^{-i}$.

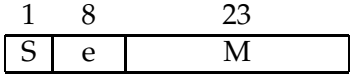
Beispiele:

$$B = 2 \overset{\text{Norm.}}{\longrightarrow} \left(\tfrac{1}{2}\right)_{10} \leq M < 1_{10} \longleftrightarrow 0.1_2 \leq M < 1_2$$
$$0.00001011_2 \overset{\text{Norm.}}{\longrightarrow} 0.1011_2 \cdot 2^{-4}$$
$$1000.0001_2 \cdot 2^{10} \overset{\text{Norm.}}{\longrightarrow} 0.10000001_2 \cdot 2^{14}$$

$$B = 8 \overset{\text{Norm.}}{\longrightarrow} \left(\tfrac{1}{8}\right)_{10} \leq M < 1_{10} \longleftrightarrow 0.1_8 \leq M < 1_8$$
$$0.02_8 \cdot 8^3 \overset{\text{Norm.}}{\longrightarrow} 0.2_8 \cdot 8^2$$

Eine 32-Bit Gleitpunktzahldarstellung (vereinfacht gegenüber IEEE-754 Standard)

Eine Möglichkeit zur rechnerinternen Darstellung von Gleitpunktzahlen mit einfacher Länge (32 Bits) ist folgende:



B = 2
$z = (-1)^{\mathrm{S}}(\alpha_1 \ldots \alpha_r)2^e = (-1)^{\mathrm{S}}(\sum_{i=1}^{r} \alpha_i \mathrm{B}^{-i})2^e$
Aufgrund der Normalisierung kann α_1=1 vorausgesetzt werden und muss nicht explizit codiert werden ("hidden bit") — also effektiv $r = 24$
S Vorzeichen der Mantisse (und damit von z)
e Exponent im Zweierkomplement.

Aus der Zweierkomplementdarstellung des Exponenten ergibt sich $e_{\min} \leq e \leq e_{\max}$ mit $e_{min} = -128$, $e_{max} = 127$

Beispiele:

$0\ \ 00000000\ \ 00000000000000000000000 = +1 * 2^0 * 0.1_2 = 0.5$
$0\ \ 00000010\ \ 00000000000000000000000 = +1 * 2^2 * 0.1_2 = 2$
$0\ \ 10000001\ \ 10100000000000000000000 = +1 * 2^{-127} * 0.1101_2 \approx 4.775 * 10^{-39}$
$1\ \ 00000011\ \ 10100000000000000000000 = -1 * 2^3 * 0.1101_2 = -6.5$

Allgemein ergeben sich für diese Art der Zahlendarstellung als betragsmäßig kleinste und größte darstellbare Zahlen $z_{min} = B^{e_{min}-1}$ und $z_{\max} = (1-B^{-r})B^{e_{\max}}$; es sind $N = 2(B-1)B^{r-1}(e_{\max}-e_{\min}+1)$ normalisierte Gleitpunktzahlen in diesem Standardformat darstellbar.
Der Darstellungsbereich ist

positiv: $z_{min} \leq z \leq z_{max}$
negativ: $-z_{max} \leq z \leq -z_{min}$

Das Intervall $(-z_{min} \ldots z_{min})$ und damit die Null werden mit dieser Zahlendarstellung nicht erfaßt.
Mit $B = 2$, $r = 24$, $e_{min} = -127$, $e_{max} = 128$ ergeben sich $z_{min} = 2^{-129} \approx 1.47 * 10^{-39}$, $z_{max} = (1-2^{-24})2^{127} \approx 1.70 * 10^{38}$. Es lassen sich $N = 2 * 2^{23} * 256 = 4294967296$ Zahlen darstellen, also $N = 2^{32}$ - wie nicht anders zu erwarten für eine 32-Bit Darstellung, in der alle Bitmuster gültige Zahlendarstellungen sind.
Der gegenüber der Festpunktdarstellung erheblich höhere Darstellungsbereich wird mit der reduzierten Genauigkeit erkauft: 23 Bit entsprechen 7 signifikanten Dezimalstellen.
Echte reelle Zahlen (z.B. π, $\sqrt{2}$) werden nur als bestmögliche Approximation durch Gleitpunktzahlen repräsentiert.

Die Verfügbarkeit so weniger Zahlen in einem derart großen Zahlenbereich legt nahe, daß Rundung eine wichtige Rolle spielt, damit Operationen über Gleitpunktzahlen selbst wieder eine Gleitpunktzahl ergeben. Wegen der enormen Konsequenzen für das Rechnen im Gleitpunktformat werden die Zusammenhänge kurz dargestellt. In der Numerischen Mathematik erfährt man hierzu mehr Einzelheiten.

Gleitpunktzahldarstellen nach IEEE-Standard 754-1985

Das IEEE (*Institute of Electrical and Electronics Engineers*) hat einen Standard für Gleitpunktzahldarstellungen und deren Operationen definiert. Die Darstellung in *single precision* (*einfache Genauigkeit*) ist eine 32-Bit Darstellung, aufgeteilt entsprechend der bereits eingeführten Darstellung: 1 Bit Vorzeichen (S) + 8 Bit Exponent (e) + 23 Bit Mantisse (M). Jedoch weicht die Interpretation der Zahl von der bereits eingeführten Codierung ab:

- Der Exponent wird nicht als Zweierkomplement kodiert, sondern in einer *biased representation* - mit einem um -127 verschobenen Zahlenbereich gegenüber der Interpretation als nicht-negative Zahl.
- Die Exponenten 0 und 255 werden für *Exceptions* reserviert.

Konkret werden S, e und M wie folgt interpretiert:

- Für $0 < e < 255$: $z = (-1)^S * 2^{e-127} * 1.M$, wobei "$1.M$" die Binärzahl ist, welche durch die Kombination einer 1 und M als Nachkommastellen entsteht. Also, für $M = \alpha_1 \dots \alpha_{23}$: $1.M = 1 + \sum_{i=1}^{23} \alpha_i 2^{-i}$. Die Interpretation der Mantisse ist also gegenüber der vereinfachten Darstellung um eine Stelle nach links verschoben, entspricht also von den drei eingeführten Formen der Normalisierung der zweiten Form (nicht der ersten). Dies sind die normalisierten Zahlen.
- Für $e = 0$ und $M \neq 0$: $z = (-1)^S * 2^{e-126} * 0.M$, mit $0.M = \sum_{i=1}^{23} \alpha_i 2^{-i}$. Dies sind die *nicht-normalisierten* Zahlen.
- Für $e = 0$, $M = 0$ und $S = 0$: $z = 0$. Es ist hier also, im Gegensatz zu unserer vereinfachten Gleitpunktzahldarstellung, also auch die Null darstellbar.
- Für $e = 0$, $M = 0$ und $S = 1$: $z = -0$ - es wird also auch eine "negative Null" zugelassen. Dies hat Vorteile z.B. bei der Arithmetik unter Einbeziehung von ∞; so behält die Aussage $1/(1/x) = x$ auch für $x = +/-\infty$ seine Gültigkeit. Es wird allerdings 0 = -0 definiert (und nicht -0 $<$ 0), so dass z.B. ein Vergleich $x = 0$ ein einfach vorhersagbares Verhalten hat.

- Für $e = 255$, $M = 0$ und $S = 0$: $z = \infty$.
- Für $e = 255$, $M = 0$ und $S = 1$: $z = -\infty$.
- Für $e = 255$ und $M \neq 0$: $z = NaN$ (Not a Number) - dies ist z.B. das Ergebnis von $0/0$ oder $\sqrt{-1}$.

Beispiele:

$0\ 00000000\ 00000000000000000000000 = 0$
$1\ 00000000\ 00000000000000000000000 = -0$

$0\ 11111111\ 00000000000000000000000 = \infty$
$1\ 11111111\ 00000000000000000000000 = -\infty$

$0\ 11111111\ 00000100000000000000000 = NaN$
$1\ 11111111\ 00100010001001010101010 = NaN$

$0\ 10000000\ 00000000000000000000000 = +1 * 2^{128-127} * 1.0_2 = 2$
$0\ 10000001\ 10100000000000000000000 = +1 * 2^{129-127} * 1.101_2 = 6.5$
$1\ 10000001\ 10100000000000000000000 = -1 * 2^{129-127} * 1.101_2 = -6.5$

$0\ 00000001\ 00000000000000000000000 = +1*2^{1-127}*1.0_2 = 2^{-126}$(Betragsmäßig kleinste normalisierte Zahl)
$0\ 00000000\ 10000000000000000000000 = +1 * 2^{-126} * 0.1_2 = 2^{-127}$
$0\ 00000000\ 00000000000000000000001 = +1*2^{-126}*0.00000000000000000000001_2 = 2^{-149}$ (Betragsmäßig kleinste nicht-normalisierte Zahl)

Die Darstellung der Gleitpunktzahlen im IEEE-Standard 754-1985 hat, gegenüber der vereinfachten Darstellung, folgende Vorteile:

- Für nicht-negative Zahlen entspricht die lexikalische Ordnung der numerischen Ordnung; dies ergibt sich aus der Anordnung des Exponenten vor der Mantisse, der Codierung des Exponenten (biased statt 2-Komplement), und der Kodierung der subnormalen Zahlen.
- $\mid e_{min} \mid \leq e_{max} \Rightarrow \frac{1}{2^{e_{min}}}$ gibt kein Überlauf. $\frac{1}{2^{e_{max}}}$ gibt Unterlauf, dies wird aber als weniger kristisch angesehen.

- Subnormale Zahlen: Die Null kann dargestellt werden, und erhöhte numerische Stabilität für das Rechnen mit betragsmäßig sehr kleinen Zahlen.

- Weitere Exceptions sind möglich: Not a Number (NaN), Infinities (Inf)

Neben der einfachen Genauigkeit definiert der IEEE-754 Standard auch andere Genauigkeiten, wie z.B. die *doppelte Genauigkeit* - mit 11 Bits für den Exponenten und 52 Bits für die Mantisse.

2.4.2.4 Arithmetische Operationen mit Gleitpunktzahlen

Seien $z_1 = (M_1, e_1)$ und $z_2 = (M_2, e_2)$ zwei normalisierte Gleitpunktzahlen. Die Ergebnisse der arithmetischen Grundoperationen in unnormalisierter Darstellung ergeben sich zu:

Addition: $z_3' = (M_1 + M_2 \cdot B^{e_2 - e_1}) \cdot B^{e_1}$

Subtraktion: $z_3' = (M_1 - M_2 \cdot B^{e_2 - e_1}) \cdot B^{e_1}$

Multiplikation: $z_3' = (M_1 \cdot M_2) \cdot B^{e_1 + e_2}$

Division: $z_3' = (M_1 / M_2) \cdot B^{e_1 - e_2}$

Hieraus wird klar, warum K. Zuse den Begriff "halblogarithmische Darstellung" für die Gleitpunktdarstellung wählte.
Für die Ausführung der Addition ergeben sich die drei Verarbeitungsschritte:

1. Exponentenausgleich mit Verschieben der Mantisse wegen der Wahl einer Referenz (hier: B^{e_1})

2. Addition der Mantissen

3. Normalisierung.

Beispiel: Gleitpunktzahl-Addition

$z_1 = (+100144, +2)$, $z_2 = (-833333, -1)$, $B = 10, r = 6$

1. Schritt: $z_1^\vee = z_1$, $z_2^\vee = (-000833, +2)$

2. Schritt: $z_3' = z_1^\vee + z_2^\vee = z_1(+099311, +2)$

3. Schritt: $z_3 = (+993110, +1)$

Ob als Referenz B^{e_1} oder B^{e_2} gewählt wird, wäre egal wenn es nicht aufgrund der endlichen Darstellung Rundungsfehler geben würde. Es sind aber tatsächlich unterschiedliche Ergebnisse zu erwarten:

a) Referenz $B^{e_1} = 10^2$
$z_3' = (0.100144 - 0.833333 \cdot 10^{-3})10^2 = 0.099311 \cdot 10^2$
$z_3 = 0.993110 \cdot 10^1$

b) Referenz $B^{e_2} = 10^{-1}$
$z_3' = (0.100144 \cdot 10^3 - 0.833333)10^{-1} = 99.310667 \cdot 10^{-1}$
$z_3 = 0.993106 \cdot 10^1$

Beispiel: Gleitpunktzahl-Subtraktion

$z_1 = (+203151, +2)$, $z_2 = (+345123, +3)$, $B = 10$

1. Schritt: $z_2^\vee = z_2$, $z_1^\vee = (-020315, +3)$

2. Schritt: $z_3' = (-324808, +3)$

3. Schritt: $z_3 = (-324808, +3)$

Mit der Referenz $B^e 2 = 10^3$ erhält man:
$z_3' = (0.203151 - 0.345123 \cdot 10^1)10^2 = -3.24808 \cdot 10^2$
$z_3 = -0.324808 \cdot 10^3$

2.4.2.5 Rundung von Gleitpunktzahlen

Das Ergebnis einer zweistelligen arithmetischen Operation über Gleitpunktzahlen ist im allgemeinen keine Gleitpunktzahl. Damit dieser Umstand überwunden werden kann, ist eine Änderung der arithmetischen Grundoperationen über Gleitpunktzahlen

erforderlich, die zur Folge hat, daß wichtige Eigenschaften der Algebra reeller Zahlen verloren gehen.
Sei $\circ$ eine zweistellige arithmetische Operation reller Zahlen,

$$\circ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}; z_1^* \circ z_2^* = z_3^* \qquad \text{mit } z_1^*, z_2^*, z_3^* \in \mathbb{R}.$$

Sei $\mathbb{F} \subset \mathbb{R}$ die Menge der darstellbaren Gleitpunktzahlen. Dann gilt für die oben behandelten arithmetischen Operationen

$$\circ' : \mathbb{F} \times \mathbb{F} \to \mathbb{R}; z_1 \circ' z_2 = z_3^{*\prime} \qquad \text{mit } z_1, z_2 \in \mathbb{F}, z_3^{*\prime} \in \mathbb{R}.$$

Gesucht ist eine arithmetische Operation, deren Ergebnis ebenfalls eine Gleitpunktzahl ist.

$$\circ_{\text{rnd}} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}; z_1 \circ_{\text{rnd}} z_2 = z_3 \qquad \text{mit } z_1, z_2, z_3 \in \mathbb{F}.$$

Für jedes $z^* \in \mathbb{R}$ gilt

$$|z_{\text{lower}}| \leq |z^*| \leq |z_{\text{higher}}| \qquad \text{mit } z_{\text{lower}}, z_{\text{higher}} \in \mathbb{F}.$$

Also ist die Forderung durch Einführen einer Rundungsfunktion $\text{rnd}(z^*)$ erfüllbar, so daß

$$\text{rnd}(z^*) = z, \quad z \in \mathbb{F}$$

und

$$z_1 \circ_{\text{rnd}} z_2 = \text{rnd}(z_1 \circ z_2) \qquad \text{Distributivität.}$$

Folgende Varianten sind gebräuchlich:

1. *Abschneiden:* Runden auf absolut kleinere der benachbarten Gleitpunktzahlen.

$$\text{rnd}(z^*) = z_{\text{rnd}}^* = z_{\text{lower}}$$

2. *optimale Rundung:* Runden auf nächstgelegenen Wert $z \in \mathbb{F}$.

$$\text{rnd}(z^*) = z_{\text{rnd}}^* = \begin{cases} z_{\text{higher}} & \text{, wenn } |z_{\text{higher}} - z^*| \leq |z_{\text{lower}} - z^*| \\ z_{\text{lower}} & \text{, sonst} \end{cases}$$

Der *absolute Rundungsfehler* ist

$$\varepsilon(z) = |\text{rnd}(z^*) - z^*|$$

und der *relative Rundungsfehler* ist

$$\rho(z) = \frac{\varepsilon(z)}{z} \text{für} z \neq 0.$$

Als Schranke des relativen Rundungsfehlers kann man die *relative Maschinengenauigkeit* $\Delta(B, r)$ angeben.

$$\Delta(B,r) = \begin{cases} B^{1-r} & \text{für Abschneiden} \\ 0.5 \cdot B^{1-r} & \text{für optimale Rundung.} \end{cases}$$

Deren Werte für einfache Genauigkeit sind

$$\Delta(2,24) = \begin{cases} 2^{-23} \approx 1.19 \cdot 10^{-7} & \text{für Abschneiden} \\ 2^{-24} \approx 5.96 \cdot 10^{-8} & \text{für optimale Rundung.} \end{cases}$$

Dies entspricht also 7 bzw. 8 Dezimalstellen. Bei doppelter Genauigkeit erhöht sich die relative Maschinengenauigkeit auf 16 bzw. 17 Dezimalstellen.
Also gilt

$$z_1 \circ_{\text{rnd}} z_2 = (z_1 \circ z_2)(1+\rho) \qquad \text{mit } |\rho| \leq \Delta.$$

Die Fehlerfortpflanzung kann zu erheblichem Gesamtfehler bei längeren Berechnungsfolgen führen. Aber bereits einfache zweistellige Operationen können zu absurden Ergebnissen (bzgl. der Algebra der reellen Zahlen) führen.
Bei Ausführung der Operation $\circ_{\text{rnd}}$ gehen wichtige Eigenschaften der Algebra der reellen Zahlen verloren:

1. Verlust der Assoziativität von Addition und Multiplikation

$$z_1 +_{\text{rnd}} (z_2 +_{\text{rnd}} z_3) \neq (z_1 +_{\text{rnd}} z_2) +_{\text{rnd}} z_3 \qquad \text{für } z_1, z_2, z_3 \in \mathbb{F}$$

$$z_1 \cdot_{\text{rnd}} (z_2 \cdot_{\text{rnd}} z_3) \neq (z_1 \cdot_{\text{rnd}} z_2) \cdot_{\text{rnd}} z_3$$

2. Verlust der Distributivität zwischen Addition und Multiplikation

$$z_1 \cdot_{\text{rnd}} (z_2 +_{\text{rnd}} z_3) \neq (z_1 \cdot_{\text{rnd}} z_2) +_{\text{rnd}} (z_1 \cdot_{\text{rnd}} z_3).$$

Beispiele: extreme Rundungsfehler

1. $z_1 = 10^{23}, z_2 = 2 \rightarrow z_3 = z_1 +_{\text{rnd}} z_2 = 10^{23}$
2. Auslöschung: Bei Subtraktion fast gleich großer Zahlen können für gebräuchliche Darstellungen relevante Stellen leicht verloren gehen.
   $z_1 = 0.746841 \cdot 10^2, z_2 = 0.746832 \cdot 10^2$
   Annahme: letzte beide Ziffern wegen Rundung unzuverlässig
   $z_3 = z_1 -_{\text{rnd}} z_2 = 0.000009 \cdot 10^2 = 0.9 \cdot 10^{-3}$
   Also sind alle Ziffern der Mantisse des Ergebnisses unzuverlässig!

---

3 Vom Problem zum Programm

Eine Aufgabe der Informationsverarbeitung umfaßt den Bogen von der Feststellung eines Problems bis zur Angabe der Lösung. Auf dem Weg vom Problem zur Lösung sind eine Reihe von Schritten nötig, die

- ein Problem als konkrete Realisierung eines allgemeineren Problems identifizieren,
- ein Problem lösbar machen, indem eine Abfolge vernünftiger Schritte definiert wird,
- Anpassungen an die verfügbaren Hilfsmittel darstellen.

Folgende 3 Etappen sollen in diesem Kapitel unterschieden werden:

1. *Spezifikation:* Problembeschreibung, die Hinweise auf die Formulierung eines Lösungsweges enthält (funktionale Spezifikation).
2. *Algorithmenentwurf:* Angabe eines Lösungsweges auf unterschiedlichen Ebenen der Abstraktion und Formalisierung, der durch eine Maschine ausführbar ist.
3. *Notation eines Programms:* Angabe eines auf einer konkreten Maschine ausführbaren Lösungsweges.

Die Grenzen der drei Etappen sind fließend. Von Etappe 1 nach 3 findet eine zunehmende Formalisierung der Ausdrucksmittel statt. Es ist vom Anfang an eine problemadäquate Korrektheit gefordert, jedoch erfordert die formale Notation möglicherweise das Akzeptieren von Approximationen.
Die zu Beginn des Kapitel 2 formulierten Zusammenhänge zwischen Systemen und Modellen kommen hier zur Wirkung.
Die Ausdrucksmittel zur Problemlösung durch Programmentwicklung bewegen sich in dem von sprachlicher Freiheit und Maschinennähe aufgespannten Raum (s. Abb. 3.1). Je mehr formale Mittel eingesetzt werden können, umso mehr läßt sich eine Automatisierung realisieren und umso mehr müssen aber auch Randbedingungen berücksichtigt werden. Spezifikation und Algorithmenentwurf sind nicht automatisierbar. Sie erfordern ein hohes Maß an Problembewußtsein und Heuristik (Engpaß Spezifikation).

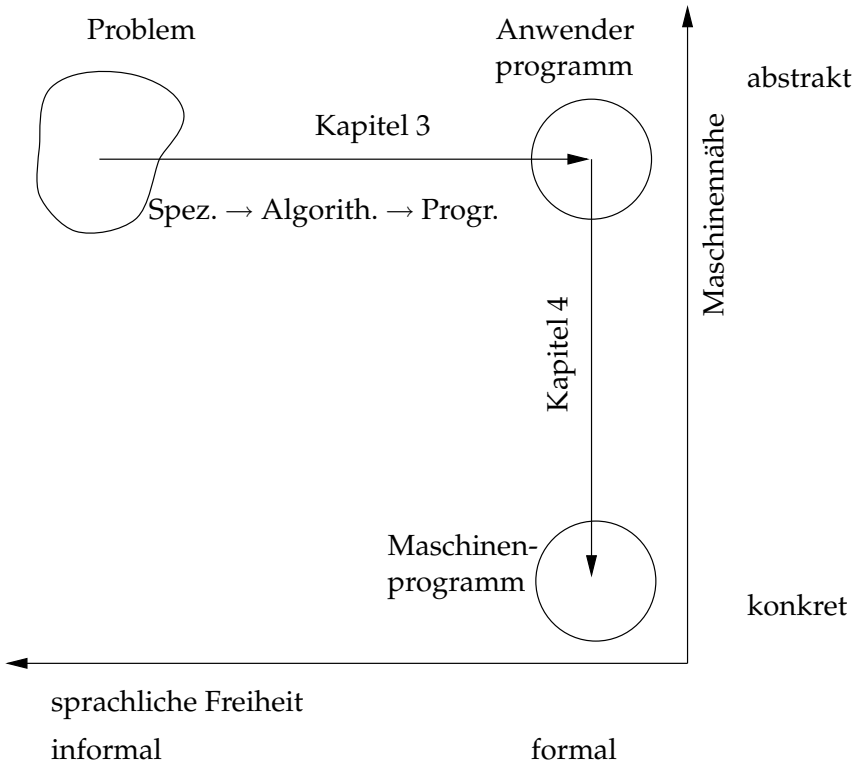


Abbildung 3.1: Durch sprachliche Freiheit und Maschinennähe aufgespannter Raum

Es gibt Probleme, für die kein Algorithmus angebbar ist (Berechenbarkeitsproblem). Diese sollen hier nicht interessieren. Andere Probleme sind von hoher Komplexität, d.h. sie können nur mit hohem Aufwand gelöst werden (Komplexitätstheorie).

Zur Herkunft des Begriffes Algorithmus:
Der persische Mathematiker Mohammed ... Al Khowarizmi schrieb ca. 800 n.Chr. ein Buch mit dem Titel “Kurzgefaßtes Lehrbuch für die Berechnung durch Vergleich und Reduktion”. Jeder Abschnitt des Buches beginnt mit “Dixit algorismi” (“So sprach Al Khowarizmi”).

Es werden folgende grundlegende Eigenschaften eines Algorithmus gefordert:

1. Beschreibung des Verfahrens durch einen endlichen Text.
2. Partitionierung des Verfahrens in eine Folge wohldefinierter Rechenschritte.
3. Die Rechnung kann Parameter besitzen ("Eingabegrößen") und führt nach endlich vielen Schritten zu einem eindeutig bestimmten Ergebnis.

Abstraktionsebenen und abstrakte/virtuelle Maschine:

Die Formalisierung von Problemlösungen als Algorithmen bzw. Programme ist die Voraussetzung für eine maschinelle Problemlösung. Die Beschreibung von Problemlösungen erfolgt dabei auf unterschiedlichen Abstraktionsebenen:

- Anwenderprogramm: sehr abstrakt (bzgl. der ausführbaren Maschinenbefehle)
- Maschinenprogramm: sehr konkret (aus Maschinensicht), aber sehr umständlich (bzgl. der Anwendersicht der Problemlösung).

Jeder sprachlichen Ebene zwischen diesen Extrema entspricht eine *abstrakte/virtuelle Maschine*. Eine Sprache definiert eine Maschine und umgekehrt definiert eine Maschine eine Sprache. Das heißt, ein Computer mit N Sprachebenen kann als System von N verkapselten virtuellen Maschinen betrachtet werden. Dabei besitzt jede Maschine ihre eigene Sprache.
Folgende Schichten virtueller Maschinen werden unterschieden:

L5/M5: problemorientiertes Sprachniveau
virtueller Computer definiert durch den Programmierer (implementiert als "C"-Programm)

L4/M4: Assembler-Sprachniveau

L3/M3: Betriebssystem-Sprachniveau

L2/M2: gewöhnliches Maschinen-Sprachnivaeu

L1/M1: Mikroprogramm-Sprachniveau
Mikroprogramme werden direkt durch die Hardware ausgeführt.

Dabei können das Assembler- und das Betriebssystem-Sprachniveau vertauscht sein.
Programme der Sprache Ln werden entweder interpretiert durch einen *Interpreter*, der auf M$(n-1)$ läuft oder übersetzt durch einen *Übersetzer/Compiler* in eine Sprache L$(n-1)$, die durch M$(n-1)$ ausführbar ist.

Ein Compiler erzeugt ein neues Programm auf einer niedrigeren Sprachebene. Ein Interpreter führt zu Instruktionen in der Sprache Ln äquivalente Instruktionen in der Sprache L$(n-1)$ aus. Dabei ist die Ausführunsgszeit in der Regel länger als bei einer übersetzten Sprache.

3.1 Spezifikation

Die Spezifikation bedeutet eine präzise Beschreibung des zu lösenden Problems. Die Erstellung einer präzisen Spezifikation kann einen erheblichen Teil der gedanklichen Arbeit zur Problemlösung darstellen.

Die Spezifikation stellt die Festlegung der Eingabe- und Ausgabebereiche sowie die Formulierung des funktionalen Zusammenhangs zwischen Eingabe- und Ausgabewerten für die Entwicklung eines Algorithmus dar.

In der Spezifikation wird der Leistungsumfang einer Problemlösung fixiert: Der Programmierer hat dafür Sorge zu tragen, daß ein erstelltes Programm der Spezifikation entspricht. Der Anwender hat dafür zu sorgen, daß das Programm im Rahmen der Spezifikation genutzt wird.

Anwender und Programmierer haben unterschiedliche Sichten auf ein Problem. Der Anwender hat ein *Nutzungsmodell* im Sinn, der Programmierer entwirft ein *Implementierungsmodell*. Indem das Nutzungsmodell in einem *Lastenheft* und das Implementierungsmodell in einem *Pflichtenheft* gemeinsam fixiert werden, wird Paßfähigkeit der Sichtweisen erreicht.

Als Grundprämisse gilt, daß ein Problem die Berechnung einer Funktion im mathematischen Sinne darstellt, d.h. eine Ableitung eines eindeutigen Wertes aus einer Menge von Parametern.

Die informalen Ebenen der Spezifikation bergen Probleme in sich:

- nicht explizites Benennen von Tatsachen, die zu den allgemeinen Erfahrungen (common sense) gehören
- Funktionen stellen oft nur partielle Abbildungen dar, d.h. nicht zu allen Parametern sind sinnvolle Werte berechenbar.

Beispiel: Parkplatzproblem

Auf einem Parkplatz stehen PKW's und Motorräder (ohne Beiwagen). Zusammen seien es n Fahrzeuge mit insgesamt m Rädern.

Die Berechnung des Funktionswertes $P(n, m)$ soll die Anzahl der PKW's liefern. Dabei sind n, m die Parameter der Funktion P.
Sei M die Anzahl der Motorräder.
Es gilt:

$$\begin{aligned} P + M &= n \\ 4P + 2M &= m \end{aligned}$$

Daraus folgt:

$$P = \frac{m - 2n}{2}$$

Aber: für n=3, m=9 folgt $P(3, 9) = 1.5$,
deshalb Forderung: m gerade
Aber: für n=5, m=2 folgt $P(5, 2) = -4$,
deshalb Forderung: $m \geq 2n$
Aber: für $n = 2$, $m = 10$ folgt $P(2, 10) = 3\ ,\ M(2, 10) = -1$,
deshalb Forderung: $m \leq 4n$,
also insgesamt: $2n \leq m \leq 4n$

Wir halten als *Spezifikationsregel* fest:

> Die funktionale Spezifikation eines Problems beschreibt die Menge der gültigen Eingabegrößen (Definitionsbereich) und die Menge der gültigen Ausgabegrößen (Wertebereich) mit allen für die Lösung wichtigen Eigenschaften, insbesondere den funktionalen Zusammenhang zwischen diesen.

In der Beschreibung des funktionalen Zusammenhangs zwischen Eingabe- und Ausgabegrößen erfolgt die implizite Angabe von Definitions- und Wertebereich:

Definition: (Vorbedingung)

> Die Vorbedingung beschreibt den Zustand vor der Ausführung des geplanten Algorithmus (seine Voraussetzungen).

Definition: (Nachbedingung)

> Die Nachbedingung beschreibt den Zustand nach der Ausführung des geplanten Algorithmus (seine Leistungen).

Definition: (Partielle Korrektheit)

Wenn die Vorbedingung für die Eingabegrößen gelten und wenn der Algorithmus terminiert, dann soll die Nachbedingung für die Ausgabegrößen gültig sein.

Beispiel: Spezifikation Parkplatzproblem

{Aufgabenbeschreibung}

Eingabe:	$m, n \in \mathbb{N}$
Vorbedingung:	m gerade, $2n \leq m \leq 4n$
Verfahren:	Berechne $P := \frac{m-2n}{2}$
Ausgabe:	$p(n, m) := P \in \mathbb{N}$
Nachbedingung:	Es gibt $P, M \in \mathbb{N}$ mit $\left\{ \begin{array}{rcl} P + M & = & n \\ 4P + 2M & = & m \end{array} \right\}$

Als guter Stil gilt, die Spezifikation in der Algorithmenbeschreibung zu wiederholen.
Die Spezifikation eines Problems hat bei zunehmend formalen sprachlichen Mitteln der Definition des Verfahrens (Algorithmus) auch die konkret auf dem Rechner bzw. in der angepeilten Programmiersprache verfügbaren Ressourcen zu berücksichtigen.
Hierzu gehören Datentypen.
Elementare Datentypen, die von einer Programmiersprache zur Verfügung gestellt werden, heißen auch *Rechenstruktur* .

Definition: (Datentyp)

Ein Datentyp ist eine Menge von Objekten S mit der für diese Objektmenge definierten Menge von Funktionen: $T = (S; f_1, \ldots, f_n)$. Dabei tritt S im Vor- und Nachbereich der Funktionen auf. Die Menge S heißt auch Sorte und die Funktionen heißen auch Operationen des Datentyps.

Der Dualismus von Daten/Objekten bzw. Funktionen führt bei der Betonung der Bedeutung der einen oder anderen Seite zu unterschiedlichen Paradigmen der Programmierung:

- *imperative* Programmierung

- *funktionale* Programmierung

Zunächst ist zu klären, wie Datentypen in der Spezifikation einer Problemlösung berücksichtigt werden müssen.

Beispiel: Datentyp eines Taschenrechners

$Q \sim$ Teilmengen der rationalen Zahlen mit max. 10 Dezimalstellen, $Q \subset \mathbb{Q}$
Datentyp eines einfachen Taschenrechners:

$$q_1 := (Q; +, -, \cdot, /)$$

für $f_i : Q \times Q \to Q,\ i = 1, \ldots 4$
$\to$ Ausführung von Prozentrechnung: Reduzierung auf verfügbaren Datentyp

$$\begin{aligned} ''\cdot 5\%'' &\to ''\cdot 0.05'' \\ ''+ 5\%'' &\to ''\cdot 1.05'' \end{aligned}$$

Ausweg:

$$q_2 := (Q; +, -, \cdot, /, \sqrt{x}, x^2, \%)$$

oder

$$q_3 := (Q; +, -, \cdot, /, \sqrt{x}, x^2, \%, \sin x, \cos x, \tan x, \exp x, \ln x)$$

$\to$ Problem, wenn Rechnung des q_2- bzw. q_3-Rechners auf q_1-Rechner ausgeführt werden soll.

Datentypen entsprechen in der Mathematik einer Algebra.

Eine (funktionale) Spezifikation beschreibt eine (partielle) Funktion von den Eingabebereichen zu den Ausgabebereichen. Dabei wird von den wirklichen Objekten, auf denen operiert werden soll, abstrahiert. An ihrer Stelle werden *Bezeichner/Identifikatoren* verwendet. Dabei können verschiedene Bezeichner den gleichen Wert annehmen. Die Bezeichnung soll aber eindeutig ein Objekt identifizieren.

Definition: (Spezifikation)

Eine (funktionale) Spezifikation besteht aus

1. *Deklarationen:* Vereinbarung problemspezifischer Objekte durch Angabe ihrer jeweiligen Bezeichner.
   - 1.1 *Konstanten:* Vereinbarung von Konstantenbezeichnern für problemspezifische Konstanten.
   - 1.2 *Datentypen:* Vereinbarung problemspezifischer Datentypen durch Angabe ihrer Typen-/Sortenbezeichner und Funktionsbezeichner (für problemrelevante Trägermengen und Operationen).
   - 1.3 *Funktionen:* Vereinbarung von Funktionsbezeichnern für problemspezifische Funktionen (auf alle Fälle Nennung der zu berechnenden Funktion).
   - 1.4 *Prädikate:* Vereinbarung von Prädikatsbezeichnern für problemspezifische Prädikate. Dies sind Funktionen, die einen Wahrheitswert zurückgeben.
2. *Eingabe:* Vereinbarung von Variablenbezeichnern für die Eingabeparameter der Spezifikation. Den Variablenbezeichnern sind Gegenstandsbereiche (Typen) zuzuordnen, über denen diese Bezeichner variieren.
3. *Vorbedingung:* Angabe einer Menge von Prädikaten, welche die Eingabe erfüllen soll.
4. *Ausgabe:* Vereinbarung von Variablenbezeichnern für die Ausgabeparameter der Spezifikation.
5. *Nachbedingung:* Angabe einer Menge von Prädikaten, welche die Leistung des zu entwickelnden Algorithmus beschreibt. Hier treten auch die Ausgabeparameter auf.

Beispiel: Suchproblem (Spezifikation)

Bestimme die Position $P(F, a)$ eines Elementes a in einer Folge F. Falls a in F nicht vorkommt, sei $P(F, a) = 0$.
Annahme: alle Folgeglieder sind verschieden

Deklarationen:

Typen:	S eine Menge, $\mathcal{F}(S)$ die Menge der endlichen Folgen über S
Funktionen:	$P: \mathcal{F}(S) \times S \to \mathbb{N}$

Eingabe:	$F = (A_1, \ldots, A_n) \in \mathcal{F}(S)$ und $a \in S$ Sei $n \geq 1$ (d.h. Folge nicht leer)
Vorbedingung:	$A_i \neq A_j$ für $i \neq j$
Ausgabe:	$P(F, a) = p$, falls $A_p = a$ für $p \in \{1, \ldots, n\}$, sonst $P(F, a) = 0$
Nachbedingung:	$(a = A_p \wedge 1 \leq p \leq n) \vee (\forall j \in \{1, \ldots, n\} : a \neq A_j \wedge p = 0)$

Dieses Beispiel werden wir im Abschnit 3.3 wieder aufgreifen, um verschiedene Zugänge des Algorithmenentwurfes zu demonstrieren. Zunächst wenden wir uns den Rechenstrukturen als Ressourcen elementarer Datentypen einer virtuellen Maschine zu.

3.2 Rechenstrukturen

Algorithmen werden in Bezug auf Datentypen formuliert, also Trägermengen von Datenelementen und den für diese effektiv verfügbaren Funktionen.

Elementare Datentypen, die nicht vom Programmierer auf der semantischen Ebene seines Problems zu definieren sind, sondern innerhalb einer Programmiersprache als Ressource des Rechners zur Verfügung gestellt werden, heißen Rechenstrukturen.

Rechenstrukturen beschreiben also die Beziehungen, die elementare Objekte (Datenelemente) mittels ihrer elementaren Operationen eingehen können.

Rechenstrukturen sind interpretierte elementare Ausdrucksmittel einer Programmiersprache. Sie enthalten neben den syntaktischen auch die semantischen Aspekte dieser Ressourcen:

- Zeichen für Zahlen (Konstanten, Variablen) wird deren Bedeutung zugewiesen ($\hat{=}$ Daten)
- Zeichen für Operationen (Operatoren) wird deren Anwendbarkeit in Form von algebraischen Grundgesetzen und anderen Gesetzen zugewiesen.

3.2.1 Signaturen, Grundterme und Terme

Wir lernten im Abschnitt 2.4 als Trägermengen elementare Datentypen kennen:

$$\mathbb{N}\ ,\ \mathbb{Z}\ ,\ \mathbb{Q}\ ,\ \mathbb{R}$$

Trägermengen sind Mengen primitiver Daten (primitive Datenbereiche). Auf der syntaktischen Ebene bezeichnet man diese Menge als *Sorten*, *Arten* oder *Typen*. Sie erhalten in den Programmiersprachen z.T. unterschiedliche Bezeichnungen (*Indikatoren*):

MATH	"C" R	"PASCAL" R	T	Bedeutung/Interpretation I (Trägermengen)
$\mathbb{N}$, $\mathbb{Z}$:	int	integer	$\longrightarrow$	Menge der ganzen Zahlen (Menge der nat. Zahlen wird als Untermenge behandelt)
$\mathbb{Q}$, $\mathbb{R}$:	float	real	$\longrightarrow$	Menge der numerisch-reellen Zahlen (Gleitpunktzahlen)
$\mathbb{B}$:	—	boolean	$\longrightarrow$	Menge der Wahrheitswerte
—:	char	char	$\longrightarrow$	Menge der durch Implementierung (Compiler) vereinbarten Schriftzeichen (1 Byte)

Hier stehen (R, T, I) für Repräsentation, Transformation und Interpretation von Information.

Den primitiven Daten werden *primitive* oder *Grundoperationen* zugeordnet. Dies geschieht zunächst auf der syntaktischen Ebene, indem die Funktionalität eines Operators als n-stellige Abbildung eingeführt wird. Die semantische Ebene dieser Abbildung enthält außerdem die aus den algebraischen Grundgesetzen folgende Bedeutung dieser Abbildung.

Definition: (Funktionalität)

Gegeben seien die Menge der Sorten S, d.h. der Namen der Trägermengen, und die Menge der Funktionssymbole F.

Die Funktionalität eines Funktionssymboles $f \in F$, ausgedrückt durch

$$\text{fct } f = (s_1, \ldots, s_n)s_{n+1},$$

weist darauf hin, daß f eine $n-$stellige Funktion, $n \in \mathbb{N}$,

$$f: \; s_1 \times \cdots \times s_n \rightarrow s_{n+1}$$

über den Sorten $s_i \in S$, $i = 1, \ldots, n+1$, darstellt.

Ist M eine Trägermenge und sind $m_i \in M$, $i = 1, \ldots, n$, die *Operanden* des Operators $f(m_1, \ldots, m_n)$, so heißen diese auch *Argumente* des Operators.
Oft ist es sinnvoll, zur Vermeidung partieller Abbildungen das Element $\perp$ (sprich: bottom) zur Repräsentation nicht definierter Funktionswerte einzuführen, so daß gilt

$$M^{\perp} = M \cup \{\perp\}.$$

Die Abbildung

$$f : M_1^{\perp} \times \ldots \times M_n^{\perp} \rightarrow M_{n+1}^{\perp}$$

heißt *strikte Abbildung* . Diese Funktionalität entspricht der Annahme, daß das Resultat der Anwendung einer Funktion auf eine Menge von Argumenten nur dann definiert ist, wenn alle Argumente definiert sind.
Operatoren, deren Operanden und Ergebnis von ein und derselben Sorte sind, heißen auch *innere* Operatoren dieser Sorte, andernfalls *äußere*. Hierfür existieren Entsprechungen auf der Seite der Operationen.

Beispiele:

innere Operatoren:	$f = +(a, b)$	fct f = (int,int)int
äußere Operatoren:	$f =< (a, b)$	fct f = (int,int)bool

Einstellige Operatoren $f(s_1)$ heißen auch *monadisch* (”C”: *unär*).

$$\text{fct } f : (s_1)s_2, \quad \text{z.B.}$$

a) $f.$ Präfixnotation
b) $.f$ Postfixnotation

Beispiele:

			”C”
a)			
	$-.$	-265	$-.$
	$\neg.$	$\neg$L	!1
	sin.	sin π	sin(.)
	abs.	abs(-12)	abs(.)
b)	$.\uparrow 2$	$5 \uparrow 2$	power(.,2)
	.=0	5=0	

Anm.: Die ”C”-Funktionen sin, abs, power sind keine elementaren Operatoren im eigentlichen Sinne, sie sind in den Standardbibliotheken definiert.

Zweistellige Operatoren $f(s_1, s_2)$ heißen auch *dyadisch* (”C”: *binär*)

$$\text{fct } f : (s_1, s_2)s_3$$

$.f.$ Infixnotation

Beispiele:

		”C”
.+.	14+7	.+.
.mod.	18 mod 4	.%.
$.\leq.$	12$\leq$13	.<=.

Nullstellige Operatoren $f()$ haben keine Argumente und liefern immer das gleiche Resultat $f = c$. Man nennt sie *Konstanten* .

Beispiele:

(FALSE, TRUE), (0,1), (0,L)
In ”C” gelten folgende Entsprechungen: TRUE $\hat{=}\, x \neq 0$, FALSE $\hat{=}\, 0$.

- Besondere einstellige Operatoren:

  1. *involutorische Operatoren:* : die nochmalige Anwendung hebt die vorherige auf, z.B.

     $$\neg\neg \mathrm{L} = \mathrm{L}$$

  2. *idempotente Operatoren*: nochmalige Anwendung ist wirkungslos, z.B.

     $$\text{abs abs } (-2) = \text{abs } (2) = 2$$

Prädikate sind Operatoren, die in die Menge der Wahrheitswerte $\mathbb{B} = \{0, \mathrm{L}\}$ abbilden, z.B.

$$. \leq . \qquad . = 0 \qquad \neg.$$

Definition: (Signatur)

Eine Signatur $\Sigma = (S, F)$ wird aus der Menge der Sorten S und der Menge der Funktionssymbole F gebildet. Für letztere ist die Funktionalität ihrer Elemente $f \in F$ gegeben aber die Bedeutung dieser Funktionssymbole bleibt ausgeklammert.

Definition: (Rechenstruktur)

Seien S und F Mengen von Bezeichnungen. Eine Rechenstruktur A besteht aus einer Familie $\{s^A : s \in S\}$ von Trägermengen s^A und einer Familie $\{f^A : f \in F\}$ von Abbildungen f^A zwischen diesen Trägermengen. Wir schreiben

$$A = (\{s^A : s \in S\}, \{f^A : f \in F\}).$$

Sowohl s^A als auch f^A sind auf der semantischen Ebene definiert!

Beispiel: Rechenstruktur Taschenrechner CALC

$S = \{\text{tasten, zustand}\}$ Sorten

Zuordnung der Trägermengen zu den Sorten:

$\text{taste}^{\text{CALC}} = T$ mit
$T = \{0, \ldots, 9, +, *, =\}$
$\text{zustand}^{\text{CALC}} = Z$ mit
$Z = \{(s, p, d) : s \in W \wedge p \in \{+, *, =\} \wedge d \in W\} \cup \{\text{error}\}$
$s \sim$ intern gespeicherter Wert
$p \sim$ momentan gespeichertes Operationssymbol
$d \sim$ sichtbare Anzeige
$W = \{0, \ldots, 10^{10} - 1\}$ Wertebereich der Arithmetik

$F = \{\text{ein}, \text{tip}, \underbrace{0, 1, \ldots, 9, +, *, =}\}$ Funktionssymbole
nullstellige Funktionssymbole: Resultat entspricht Taste
$\text{ein}^{\text{CALC}} : \rightarrow Z$ $\sim$ Einstellen des Grundzustands
$\text{tip}^{\text{CALC}} : T \times Z \rightarrow Z$ $\sim$ neuer Zustand aus Tastenbetätigung

Spezifizierung der Funktionen:
für $x \in \{0, \ldots, 9\}$, $t \in T$ gelte

$$
\begin{array}{lcl}
\text{ein}^{\text{CALC}} & = & (0, =, 0) \\
\text{tip}^{\text{CALC}}(x, (s, p, d)) & = & \begin{cases} \text{error} & \text{falls } \neg(d * 10 + x \in W) \\ (s, p, d * 10 + x) & \text{sonst} \end{cases} \\
\text{tip}^{\text{CALC}}(+, (s, p, d)) & = & (d, +, 0) \\
\text{tip}^{\text{CALC}}(*, (s, p, d)) & = & (d, *, 0) \\
\text{tip}^{\text{CALC}}(=, (s, p, d)) & = & \begin{cases} (s, =, s * d) & \text{falls } s * d \in W \wedge p = * \\ (s, =, s + d) & \text{falls } s + d \in W \wedge p = + \\ (d, =, 0) & \text{falls } p = \text{"="} \\ \text{error} & \text{sonst} \end{cases} \\
\text{tip}^{\text{CALC}}(t, \text{error}) & = & \text{error}
\end{array}
$$

Eingabe:
91+12=∗2=
→Term:
tip(=,tip(2,tip(∗,tip(=,tip(2,tip(1,tip(+,tip(1,tip(9,ein)))))))))
Zustand:
(103,=,206)

Zunächst sei der Begriff Grundterm erklärt. Dabei handelt es sich um operationale Verknüpfungen von Elementen einer gegebenen Sorte, also der Konstanten der entsprechenden Sorte. Die Sorte bool hat bekanntermaßen nur zwei Konstanten (TRUE, FALSE). Die Sorten nat und int haben aber unendlich viele Konstanten.

Definition: (Grundterme)

Sei $\Sigma = (S, F)$ eine Signatur. Die Menge der Grundterme der Sorte s, $s \in S$, ist definiert durch

- jedes nullstellige Funktionssymbol $f \in F$ mit fct $f = s$,
- jede Zeichenreihe $f(t_1, \ldots, t_n)$ mit $f \in F$ und fct $f = (s_1, \ldots, s_n)s$, falls t_i für alle $i \in \{1, \ldots, n\}$ ein Grundterm der Sorte s_i ist.

Die Menge aller Grundterme der Sorte s einer Signatur Σ sei mit W_{Σ^s} bezeichnet. W_{Σ^s} heißt auch *Grundtermalgebra* der Sorte s.

Grundterme der Sorte $s \in S$ werden also über den erlaubten Operationen $f \in F$ induktiv definiert.

Beispiele: Grundterme aus verschiedenen Sorten

a) Sorte nat der Rechenstruktur NAT (der natürlichen Zahlen)
nat$^{\mathrm{NAT}} = \mathbb{N}^{\perp}$ (Trägermenge)
Spezifikation einiger Funktionen aus F^{NAT}:
- zero$^{\mathrm{NAT}} = 0$
- für $x \in \mathbb{N}$:
  succ$^{\mathrm{NAT}}(x) = x + 1$ (successor)
  pred$^{\mathrm{NAT}}(x) = x - 1$, falls $x \geq 1$ (predecessor)
- für $x, y \in \mathbb{N}$
  add$^{\mathrm{NAT}}(x, y) = x + y$

Bsp.: succ(zero)
add(succ(succ(zero)),pred(succ(zero)))

b) Sorte bool der Rechenstruktur NAT (der natürlichen Zahlen)
bool$^{\mathrm{NAT}} = \mathbb{B}^{\perp}$
für $x, y \in \mathbb{N}$:

$$\leq^{\mathrm{NAT}}(x, y) = \begin{cases} 0 & \text{für } x > y \\ \mathrm{L} & \text{für } x \leq y \end{cases}$$

Bsp.: $3 \leq 5: \quad \leq (3,5) = \mathrm{L}$

Liegt eine Rechenstruktur A mit der Signatur Σ vor, so lassen sich die Grundterme in A interpretieren. Das heißt, man kann ihnen einen Wert a zuweisen.

Definition: (Interpretation eines Grundterms)

Den Übergang von einem Grundterm t (der Repräsentation) der Sorte s zum entsprechenden Element a der Menge in A nennt man Interpretation von t in A. Die Interpretation I^A stellt eine Abbildung

$$I^A : W_\Sigma \rightarrow \{a \in s^A : s \in S\}$$

dar. Man erhält sie, indem man im Grundterm die Funktionssymbole durch die entsprechenden Funktionen ersetzt.

$$I^A[f(t_1, \ldots, t_n)] = f^A(I^A[t_1], \ldots, I^A[t_n]).$$

Das Tripel $(W_\Sigma, \{a \in s^A : s \in S\}, I^A)$ bildet eine Informationsstruktur (=Tripel (Repräsentation, Information, Interpretation)).

Beispiele:

I^{NAT}[succ(zero)]=1

I^{NAT}[pred(zero)]= $\perp$

I^{NAT}[pred(add(succ(succ(zero)),zero))]=1

$I^{\mathrm{BOOL}}[0 \wedge (\neg \mathrm{L} \vee 0 \wedge \neg 0) \wedge \mathrm{L}] = 0$

Prioritäten: $\neg$ vor $\wedge$ vor $\vee$

Algorithmen sollen mit wechselnden Objekten (mit Objektparametern) durchgeführt werden. Die Namen dieser frei wählbaren Objektparameter nennt man *Identifikatoren* oder *Bezeichner*. Sie sind Platzhalter für Objekte, d.h. Terme oder Elemente, die erst später an deren Stelle eingesetzt werden.

Mit der Einführung der Identifikatoren wird das Konzept einer Variablen zu einer gegebenen Sorte in die operationale Verknüpfung von Elementen der Menge dieser Variablen eingeführt. Damit erweitert sich auch das Konzept der Grundterme zu dem der Terme. Ebenso wie Grundterme werden Terme induktiv definiert. Im Unterschied zu Grundtermen enthalten Terme aber auch Variablen. Sie bilden eine Obermenge zur

Menge der Grundterme. Während also das Konstrukt $x \wedge y$ als Term mit den Variablen x und y bezeichnet wird, ist das Konstrukt $0 \wedge \mathrm{L}$ ein Grundterm bezüglich der Sorte bool.

Definition: (Termalgebra mit Identifikatoren)

Seien $\Sigma = (S, F)$ eine Signatur und $X = \{X_s : s \in S\}$ eine Familie von Mengen von Identifikatoren. Die X_s seien paarweise disjunkt und disjunkt zu den Funktionssymbolen $f \in F$. Dann bezeichnet $W_\Sigma(X)$ die um X erweiterte Termalgebra, d.h. W_{Σ^*} mit

$$\Sigma^* = (S, F \cup \{x \in X_s : s \in S\})$$

und fct $x = s$ für $x \in X_s$.

Durch die *Substitution* eines Identifikators x der Sorte s in einem Term t mit Identifikatoren durch einen Term r der Sorte s wird ein Term $t[r/x]$ erzeugt. Dieser Vorgang heißt auch *Instantiierung*. Ein Term u heißt *Instanz* eines Terms t mit Identifikatoren, wenn u durch Substitution gewisser Identifikatoren aus t entsteht.

Beispiel:

$t :=$ mult(add(succ(x),y),z) Interpretation: $((x + 1) + y) * z$

Instantiierung:
$u = t$[zero/x,succ(zero)/y,succ(succ(zero))/z]

$\rightarrow$ mult(add(succ(zero),succ(zero)),succ(succ(zero)))
Interpretation: $((0 + 1) + (0 + 1)) * (0 + 1 + 1)$

Definition: (Belegung)

Sei A eine Rechenstruktur mit Signatur $\Sigma = (S, F)$ und sei X eine Familie von Mengen von Identifikatoren. Eine Abbildung

$$\beta : \{x \in X_s : s \in S\} \rightarrow \{a \in s^A : s \in S\},$$

die jedem Identifikator $x \in X_s$ ein Element $a \in s^A$ der Trägermenge s^A zur Sorte s zuordnet, heißt Belegung von X in A.

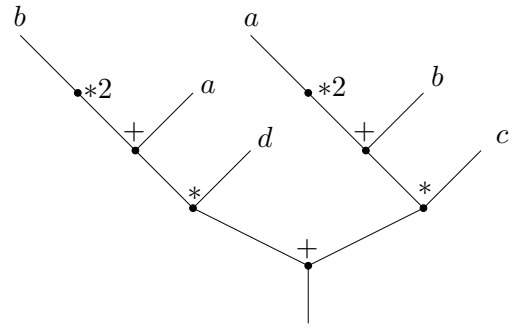
Für jede Belegung β ist die Interpretation I_β^A eines Terms t mit freien Identifikatoren aus X, definiert durch folgende Gleichungen:

$$\begin{aligned} I_\beta^A[x] &= \beta(x) \\ I_\beta^A[f(t_1,\ldots,t_n)] &= f^A(I_\beta^A[t_1],\ldots,I_\beta^A[t_n]) \end{aligned}$$

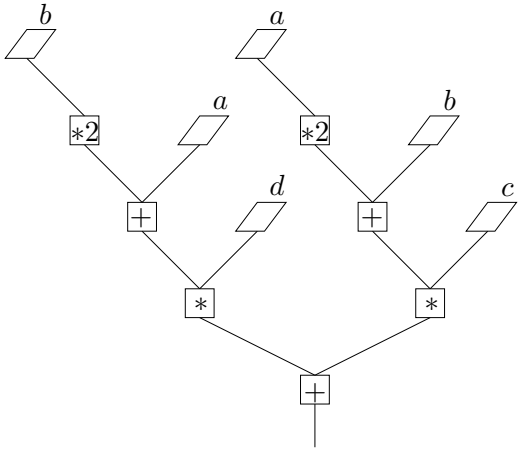
Beispiel: Abbildung eines Terms in Kantorovič-Baum bzw. Datenflußplan

Term: $(b * 2 + a) * d + (a * 2 + b) * c$
In diesem Beispiel sei $*2$ ein Grundterm.

Kantorovič-Baum:



Datenflußplan:



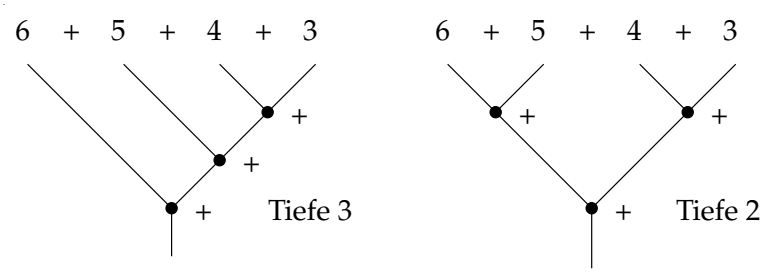
Speicherplatz für Daten

Zwischen sämtlichen in einem Term vorkommenden Operationen besteht eine zweistellige Relation “früher zu berechnen als”.

Dies führt zu einem topologischen Ordnungsprinzip (partielle Ordnung), dessen erzeugender Graph durch einen *Kantorovič-Baum* oder einen *Datenflußplan* ausgedrückt werden kann.

Sowohl die Tiefe als auch die Teilterme der Berechnungsstrukturen sind nicht eindeutig bestimmt.

Beispiel:



3.2.2 Die Rechenstruktur der Wahrheitswerte BOOL

Der Typ bool existiert nicht in "C". Dennoch wird auch in dieser Sprache mittels $\{0,1\} \subset \mathrm{int}$ und durch logische Verknüpfungen die Rechenstruktur realisiert.

Die Rechenstruktur der Wahrheitswerte ist die grundlegendste der Informatik. Sie gestattet, Ausdrücken bzw. Termen die Werte "wahr" oder "falsch" zuzuordnen.

Definition: (Boolesche Algebra)

Eine Algebra über dem Alphabet $\mathbb{B} = \{0,1\}$ mit den Basisoperationen $\cup$ (Vereinigung), $\cap$ (Durchschnitt) und $^-$ (Komplement) heißt Boolesche Algebra $(\mathbb{B}, \cup, \cap, ^-)$, wenn gilt für $x, y \in \mathbb{B}$:

$$\begin{aligned} x \cup y &:= \mathrm{Max}(x,y) \\ x \cap y &:= \mathrm{Min}(x,y) \\ \overline{x} &:= 1 - x \end{aligned}$$

und

$$0 = 0 \cap x\,, \quad 1 = x \cup 1$$

bezeichnen das kleinste bzw. größte Element von $\mathbb{B}$.

Die Bedeutung der Booleschen Algebra für die Informatik folgt daraus, daß es für die angegebene mengentheoretische Boolesche Algebra äquivalente Boolesche Algebren gibt, die gestatten, Programmstrukturen auf Schaltfunktionen einer Binärzahlarithmetik abzubilden:

a) Logik: $\mathbb{B} = \{0, 1\}$ $I^{\mathrm{BOOL}}[0]$:="FALSE"
$I^{\mathrm{BOOL}}[1]$:="TRUE"

$x, y \in \mathbb{B}$:	x, y sind Identifikatoren für Boolesche Werte	
$\cup \Leftrightarrow \vee$:	$I^{\mathrm{BOOL}}[x \vee y] := \mathrm{or}(I^{\mathrm{BOOL}}[x], I^{\mathrm{BOOL}}[y])$	(Disjunktion)
$\cap \Leftrightarrow \wedge$:	$I^{\mathrm{BOOL}}[x \wedge y] := \mathrm{and}(I^{\mathrm{BOOL}}[x], I^{\mathrm{BOOL}}[y])$	(Konjunktion)
$^{-}\Leftrightarrow \neg$:	$I^{\mathrm{BOOL}}[\neg x] := \mathrm{not}(I^{\mathrm{BOOL}}[x])$	(Negation)

b) Binärzahlarithmetik bzw. Schaltfunktionen:

$\mathbb{B} = \{0, 1\}$	$I^{\mathrm{BOOL}}[0] := 0$
	$I^{\mathrm{BOOL}}[1] := 1$ (Binärzahlarithmetik) bzw.
	$I^{\mathrm{BOOL}}[1] := \mathrm{L}$ (Schaltfunktion)
$x, y \in \mathbb{B}$	
$\cup \Leftrightarrow +$:	$I^{\mathrm{BOOL}}[x + y] := \mathrm{add}(I^{\mathrm{BOOL}}[x], I^{\mathrm{BOOL}}[y])$
$\cap \Leftrightarrow *$:	$I^{\mathrm{BOOL}}[x * y] := \mathrm{mult}(I^{\mathrm{BOOL}}[x], I^{\mathrm{BOOL}}[y])$
$^{-}\Leftrightarrow ^{-}$:	$I^{\mathrm{BOOL}}[\overline{x}] := \mathrm{compl}(I^{\mathrm{BOOL}}[x])$

Völlig gleichberechtigt wird auch die Darstellung $\mathbb{B} = \{0, \mathrm{L}\}$ verwendet.

zu a) Aussagen sind grundlegende Arten von Informationsträgern. Sie können wahr oder falsch sein. Für ein festgelegtes Bezugssystem gestattet eine Menge von wahren Aussagen, bestimmte Eigenschaften oder Zustände eines Objektes zu beschreiben. Aristoteles definierte Aussagen folgendermaßen:

> *Eine Aussage ist ein sprachliches Gebilde, von dem es sinnvoll ist zu sagen, es sei wahr oder falsch.*

So ist der Satz 'Der Schnee ist grün!' eine Aussage im obigen Sinne, denn

$$I^{\mathrm{BOOL}}[\text{'Der Schnee ist grün!'}]=\text{"FALSE"}$$

Die Äußerung 'Dieser Satz ist falsch!' führt hingegen zu einem Paradoxon, denn sie nimmt auf ihre eigene Bedeutung Bezug, ein Wahrheitswert ist nicht bestimmbar.

$$I^{\mathrm{BOOL}}[\text{'Dieser Satz ist falsch!'}]=\text{"}I^{\mathrm{BOOL}}[\text{'}\ldots\text{'}]=\text{"FALSE" "}$$

Eine Einschränkung umgangsprachlicher Ausdrucksmittel ist demzufolge notwendig und möglich durch die Einführung der formalisierten Sprache der *Aussagenlogik*. In der Aussagenlogik werden Aussagenformen behandelt, die gestatten, über die Zusammensetzung von elementaren Aussagen wiederum Aussagen zu bilden.

zu b) Die Berechnung binärer arithmetischer Ausdrücke erfolgt im Computer durch Schaltnetzwerke, die Schaltfunktionen realisieren.

Definition: (Schaltfunktion)

Seien $n, m \in \mathbb{N},\ n, m \geq 1$. Dann heißt eine Funktion

$$f:\ \mathbb{B}^n \rightarrow \mathbb{B}^m$$

Schaltfunktion.

In Kapitel 5 wird der Zusammenhang zwischen Schaltfunktionen und Boolescher Algebra ausführlich behandelt.

3.2.2.1 Boolesche Funktionen

Hier wollen wir uns zunächst für Boolesche Funktionen interessieren, die eine eindeutige Abbildung auf die Wahrheitswerte liefern.

Definition: (Boolesche Funktion)

Eine Funktion

$$f : \mathbb{B}^n \rightarrow \mathbb{B} \quad , n \in \mathbb{N},$$

heißt n-stellige Boolesche Funktion.

Zu vorgegebenem $n > 0$ gibt es 2^{2^n} verschiedene n-stellige Boolesche Funktionen. Diese werden durch jeweils 2^n Argumentvarianten aus $\mathbb{B}$ definiert. Da jede Argumentvariante einen Funktionswert aus $\mathbb{B}$ zur Folge hat, existieren also 2^{2^n} unterschiedliche Abbildungen für gegebenes n.

n=1 $f : \mathbb{B} \rightarrow \mathbb{B}$ $\rightarrow$ 4 Boolesche Funktionen

Mögliche Boolesche Funktionen:

$\beta(x)$	0	1		
$f_0(x)$	0	0	0	Nullfunktion
$f_1(x)$	0	1	x	Identität
$f_2(x)$	1	0	$\neg x$	Negation (not)
$f_3(x)$	1	1	1	Einsfunktion

Die Definition Boolescher Funktionen kann mittels Wertetafeln erfolgen, hier am Beispiel der obigen Negation illustriert:

$\beta(x)$	0	1
not(x)	1	0

n=2 $f : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ $\to$ 16 Boolesche Funktionen
Die Belegung beider Argumente kann auf $2^2 = 4$ verschiedene Arten mit 0 oder 1 erfolgen, für jede dieser vier verschiedenen Argumentvarianten existieren 2 mögliche Funktionswerte (0 oder 1), also $16 = 2^4$ verschiedene zweistellige Abbildungen.

Diese werden in der folgenden Tabelle aufgeführt, dabei werden vier Schreibweisen vorgestellt:

1 Schreibweise für Binärzahl-Arithmetik

2 Schreibweise der Arithmetik, erweitert um Vergleichsoperationen

3 Schreibweise der Logik, äquivalente Funktionsbezeichnungen (kürzere Schreibweise)

4 Schreibweise der Logik, nach Boolescher Algebra ($\mathbb{B}, \vee, \wedge, \neg$)

Die Tabelle 3.1 für zweistellige Boolesche Funktionen ist in folgender Weise zu lesen:

f_0**:** keine der Belegungen von x und y bildet nach dem Wert $f_0 = 1$ ab

f_1**:** nur die Belegungen ($x = 1,\ y = 1$) bilden nach $f_1 = 1$ ab. Alle anderen Belegungen bilden nach $f_1 = 0$ ab.

Das Muster der Abbildung von f_i nach den Werten 0 oder 1 ergibt einen vierstelligen Code für den Index i der Funktion f_i.
Beim Übergang von Schreibweise 3 nach 4 wird deutlich, daß sich jede n-stellige Boolesche Funktion aus wenigen ein- und zweistelligen Grundfunktionen (Disjunktion, Konjunktion, Negation) bilden läßt.
Im folgenden werden die Wertetafeln für die Konjunktion und Disjunktion angegeben:

and	$\beta(y)$	1	0
$\beta(x)$			
1		1	0
0		0	0

or	$\beta(y)$	1	0
$\beta(x)$			
1		1	1
0		1	0

Aus der Symmetrie der Wertetabellen folgt die Kommutativität der Konjunktion und Disjunktion.

3.2.2.2 Rechenstruktur der Booleschen Algebra der Wahrheitswerte

Die Rechenstruktur der Wahrheitswerte läßt sich mit den aussagenlogischen Operationen

$\beta(x)$ / $\beta(y)$	0 0 1 1 / 0 1 0 1	1	2	3	4	Bedeutung Interpretation
f_0	0 0 0 0	$x \cdot \overline{x}$	$\equiv 0$			Nullfunktion
f_1	0 0 0 1	$x \cdot y$	$\text{Min}(x, y)$	$x \wedge y$	$x \wedge y$	Konjunktion (AND)
f_2	0 0 1 0	$x \cdot \overline{y}$	$x > y$	$x \not\Rightarrow y$	$x \wedge \neg y$	Inhibition (Nicht-Wenn-Dann)
f_3	0 0 1 1	x	x	x	x	Identität (Tautologie)
f_4	0 1 0 0	$\overline{x} \cdot y$	$x < y$	$x \not\Leftarrow y$	$\neg x \wedge y$	Inhibition (Nicht-Wenn-Dann)
f_5	0 1 0 1	y	y	y	y	Identität (Tautologie)
f_6	0 1 1 0	$x \not\Leftrightarrow y$	$x \neq y$	$x \not\Leftrightarrow y$	$(x \wedge \neg y) \vee (\neg x \wedge y)$	Antivalenz (XOR)
f_7	0 1 1 1	$x + y$	$\text{Max}(x, y)$	$x \vee y$	$x \vee y$	Disjunktion (OR)
f_8	1 0 0 0	$\overline{x + y}$	$1 - \text{Max}(x, y)$	$x \Downarrow y$	$\neg x \wedge \neg y = \neg(x \vee y)$	Peirce-Fkt. (NOR, Weder-Noch)
f_9	1 0 0 1	$x = y$	$x = y$	$x \Leftrightarrow y$	$(x \wedge y) \vee (\neg x \wedge \neg y)$	Äquivalenz (Genau Dann-Wenn)
f_{10}	1 0 1 0	$\overline{y}$	$1 - y$	$\neg y$	$\neg y$	Negation (NOT)
f_{11}	1 0 1 1	$x + \overline{y}$	$x \geq y$	$x \Leftarrow y$	$\neg y \vee x$	Implikation (Wenn-Dann)
f_{12}	1 1 0 0	$\overline{x}$	$1 - x$	$\neg x$	$\neg x$	Negation (NOT)
f_{13}	1 1 0 1	$\overline{x} + y$	$x \leq y$	$x \Rightarrow y$	$\neg x \vee y$	Implikation (Wenn-Dann)
f_{14}	1 1 1 0	$\overline{x \cdot y}$	$1 - \text{Min}(x, y)$	$x \Uparrow y$	$\neg x \vee \neg y = \neg(x \wedge y)$	Sheffer-Fkt. (NAND)
f_{15}	1 1 1 1	$x + \overline{x}$	$\equiv 1$			Einsfunktion

Tabelle 3.1: Funktionswertetabelle

Konjunktion	(”and”)	Operator: $\wedge$
Disjunktion	(”or”)	Operator: $\vee$
Negation	(”not”)	Operator: $\neg$

bilden. Damit lassen sich alle möglichen Booleschen Funktionen ausdrücken.

Beispiele:

f_{13}: impl(x, y) = or(not(x),y)

impl	$\beta(y)$	1	0
$\beta(x)$			
1		1	0
0		1	1

f_9: equiv(x,y) = or(and(x,y),and(not(x),not(y)))
= and(impl(x,y),impl(y,x))

equiv	$\beta(y)$	1	0
$\beta(x)$			
1		1	0
0		0	1

Damit läßt sich die Rechenstruktur BOOL formal niederschreiben:

Structure BOOL

Signature: $\Sigma^{\mathrm{BOOL}} = (S^{\mathrm{BOOL}}, F^{\mathrm{BOOL}})$

Sorten: S^{BOOL}={bool} mit bool = $\mathbb{B}^{\perp}$

Funktionssymbole: $F^{\mathrm{BOOL}} = \{$Konstanten, Operationen$\}$

Konstanten: $1 : \mathbb{B}^{\perp}$ speziell: I^{BOOL}[1]=”TRUE”
$0 : \mathbb{B}^{\perp}$ speziell: I^{BOOL}[0]=”FALSE”

Operationen: $\neg$: $\mathbb{B}^{\perp} \to \mathbb{B}^{\perp}$
$\wedge$: $\mathbb{B}^{\perp} \times \mathbb{B}^{\perp} \to \mathbb{B}^{\perp}$
$\vee$: $\mathbb{B}^{\perp} \times \mathbb{B}^{\perp} \to \mathbb{B}^{\perp}$

speziell für $x, y \in \mathbb{B}$:
$I^{\mathrm{BOOL}}[\neg x] =$ not(x)
$I^{\mathrm{BOOL}}[x \vee y] =$ or(x,y)
$I^{\mathrm{BOOL}}[x \wedge y] =$ and(x, y)

Spezifikation: ”Eigenschaften: Gesetze der Booleschen Algebra, Gesetze semantischer Äquivalenzen”

End Structure

3.2.2.3 Gesetze der Booleschen Algebra

Seien $x, y, z \in \mathbb{B}$, d.h. Identifikatoren für beliebige Boolesche Werte. Seien außerdem $0, 1 \in \mathbb{B}$ Boolesche Konstanten, d.h. atomare Boolesche Terme, die unabhängig von einer Belegung eine eindeutige Interpretation haben. Sie sind äquivalent zu Booleschen Termen, deren Interpretation ebenfalls unabhängig von der Wahl der Belegung ist:

Komplementgesetz:	$1 \equiv (x \vee (\neg x))$	Gesetz für TRUE
	$0 \equiv (\neg 1)$	Gesetz für FALSE
bzw.	$0 \equiv (x \wedge (\neg x))$	
	$1 \equiv (\neg 0)$	
außerdem:	$(x \vee 0) \equiv x$	
	$(x \wedge 1) \equiv x$	
	$(x \vee 1) \equiv 1$	
	$(x \wedge 0) \equiv 0$	

Man erkennt:

1. Die Gleichungen drücken semantische Äquivalenzen aus. Wir schreiben $t_1 \equiv t_2$, wenn wir explizit auf die Äquivalenz von Termen hinweisen wollen, sonst schreiben wir vereinfachend $t_1 = t_2$.
2. Es gilt das Dualitätsprinzip der Booleschen Algebra.

Dualitätsprinzip der Booleschen Algebra

Alle Gesetze der Booleschen Algebra gehen wieder in gültige Äquivalenzen über, wenn konsistent jeder Boolesche Wert durch seine Negation und jede Disjunktion durch eine Konjunktion bzw. jede Konjunktion durch eine Disjunktion ersetzt werden.

Deshalb werden die Gesetze der Booleschen Algebra nur in einer Variante angegeben.

Gesetze der Booleschen Algebra

1. Involutionsgesetz	$\neg\neg x$	$\equiv$	x
2. Kommutativgesetz	$x \wedge y$	$\equiv$	$y \wedge x$
3. Assoziativgesetz	$(x \wedge y) \wedge z$	$\equiv$	$x \wedge (y \wedge z)$
4. Idempotenzgesetz	$x \wedge x$	$\equiv$	x
5. Absorptionsgesetz	$x \wedge (x \vee y)$	$\equiv$	x
6. Distributivgesetz	$x \wedge (y \vee z)$	$\equiv$	$(x \wedge y) \vee (x \wedge z)$
7. Gesetz von de Morgan	$\neg(x \wedge y)$	$\equiv$	$(\neg x) \vee (\neg y)$
8. Neutralitätsgesetz	$x \vee (y \wedge (\neg y))$	$\equiv$	x

3.2.3 Boolesche Terme

Die Gesetze der Booleschen Algebra gestatten, Boolesche Terme in semantisch äquivalente Boolesche Terme umzuwandeln.

Definition: (Boolesche Terme)

1. Die Konstanten 0 und 1 sind atomare Boolesche Terme.
2. Die durch die Identifikatoren $x \in X,\ I^{\mathrm{BOOL}}[x] = a,\ a \in \mathbb{B}$, gebildeten Konstanten sind atomare Boolesche Terme.
3. Die aus der Verknüpfung atomarer Boolescher Terme mit den Operationen "and" ($\wedge$), "or" ($\vee$) oder "not" ($\neg$) gebildeten Ausdrücke sind Boolesche Terme.
4. Ist t ein Boolescher Term, so ist auch $(\neg t)$ ein Boolescher Term.
5. Sind t_1 und t_2 Boolesche Terme, so sind auch

$t_1 \vee t_2$	("t_1 oder t_2")
$t_1 \wedge t_2$	("t_1 und t_2")
$t_1 \Rightarrow t_2$	("t_1 impliziert t_2")
$t_1 \Leftarrow t_2$	("t_2 impliziert t_1")
$t_1 \Leftrightarrow t_2$	("t_1 ist äquivalent t_2")
$t_1 \Uparrow t_2$	("nicht (t_1 und t_2)")
$t_1 \Downarrow t_2$	("nicht (t_1 oder t_2)")

Boolesche Terme.

Boolesche Terme mit freien Identifikatoren repräsentieren *Aussageformen* oder Aussageschemata. Für jede Belegung der Identifikatoren mit bestimmten Wahrheitswerten erhält man einen Wahrheitswert.

Beispiel: Beweis der semantischen Äquivalenz des Absorptionsgesetzes

Allgemein ist für die Äquivalenz zweier Terme t_1 und t_2 für alle Belegungen β zu zeigen: $I_\beta[t_1] = I_\beta[t_2]$.

Für den Beweis der Äquivalenz des Absorptionsgesetzes ist hier für alle Belegungen β zu zeigen

$$I_\beta^{\mathrm{BOOL}}[x \wedge (x \vee y)] = I_\beta^{\mathrm{BOOL}}[x]$$

Erinnerung: $I_\beta^A[x] = \beta(x)$, für $x \in X$

$$I^A_\beta[f(t_1,\ldots,t_n)] = f^A(I^A_\beta[t_1],\ldots,I^A_\beta[t_n])$$

$\beta(x)$	$\beta(y)$	$I^{\mathrm{BOOL}}_\beta[x \vee y]$	$I^{\mathrm{BOOL}}_\beta[x \wedge (x \vee y)]$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

Ein Boolescher Term repräsentiert eine Abbildung von Belegungen in Wahrheitswerte.

Das hier sichtbare gewordene Prinzip der induktiven Definition der Bedeutung Boolescher Terme über ihren strukturellen Aufbau ist von grundlegender Bedeutung für die Definition der Semantik von Programmen!

Boolesche Operationen unterliegen Vorrangregeln. Dies gestattet, voll geklammerte Terme in äquivalente unvollständig geklammerte Terme zu überführen.

Vorrangregeln für Boolesche Operatoren

1. Der einstellige Operator $\neg$ bindet stärker als die zweistelligen Operatoren ($\wedge, \vee, \wedge, \Rightarrow, \Leftrightarrow$).
2. Der Operator $\wedge$ bindet stärker als $\vee$ ("Punktrechnung geht vor Strichrechnung"). Am schwächsten binden $\Rightarrow$ und $\Leftrightarrow$.
3. Ungeklammerte Boolesche Terme, die durch Operatoren gleicher Bindungsstärke erzeugt werden, werden von links nach rechts geklammert.

Beispiel:

$x, y \in X,\ I^{\mathrm{BOOL}}[x] = a,\ I^{\mathrm{BOOL}}[y] = b,\ a, b \in \mathbb{B}$

$t = ((x \wedge ((\neg y) \vee (x \wedge (\neg x)))) \wedge y)$
$t' = x \wedge (\neg y \vee x \wedge \neg x) \wedge y$
Es gilt: $t \equiv t'$

Identifikatoren können auf der Repräsentationsebene auch als Stellvertreter für Terme aufgefaßt werden (Punkt 5 der Definition Boolescher Terme).

Beispiel:

$t = x \wedge y \vee \neg x$
Substitution: $t' = t[y \Leftrightarrow x/x] := \ t' = y \Leftrightarrow x \wedge y \vee \neg(y \Leftrightarrow x)$
Es gilt: $t \equiv t'$

Einige Substitutionsregeln für $t[r/x]$:

$x[r/x] := r$
$y[r/x] := y \qquad$ falls $y \neq x$
$0[r/x] := 0$
$(\neg t)[r/x] := \neg(t[r/x])$
$(t_1 \wedge t_2)[r/x] := (t_1[r/x] \wedge t_2[r/x])$

Sei t_1 ein Term, in dem u_1 als Teilterm vorkommt. Sei t_2 derjenige Term, der entsteht, wenn in t_1 der Teilterm u_1 durch einen Term u_2 ersetzt wird. Dann folgt aus $u_1 \equiv u_2$, daß $t_1 \equiv t_2$.
Die Äquivalenz Boolescher Terme (bzw. ihre Äquivalenzen zu einem Wahrheitswert) wird gezeigt, indem

1. alle Belegungen durchgerechnet werden (u.U. sehr aufwendig) oder
2. gezielt Äquivalenzen auf einem der Terme angewendet werden, bis er in den anderen überführt wird.

Jeder Boolescher Term kann in einen äquivalenten Booleschen Term umgerechnet werden, in dem nur noch die Operatoren $\{\neg, \wedge\}$ bzw. $\{\neg, \vee\}$ vorkommen.

Definition: (Basis Boolescher Terme)

Eine Menge von Operatoren, die genügt, um mit ihnen alle möglichen Booleschen Terme zu erzeugen, heißt Basis Boolescher Terme.

Basen Boolescher Terme

Die Operatormengen $\{\neg, \wedge\}$ bzw. $\{\neg, \vee\}$ bilden eine Basis Boolescher Terme. Eine minimale Basis bilden sowohl die Peirce- als auch die Sheffer-Funktion.

Beispiele:

$$\begin{array}{rcccl} \neg x & \equiv & \neg(x \vee x) & \equiv & x \Downarrow x \\ \neg x & \equiv & \neg(x \wedge x) & \equiv & x \Uparrow x \\ x \wedge y & \equiv & \neg\neg(x \wedge y) & \equiv & \neg(\neg x \vee \neg y) \quad \equiv \quad (x \Downarrow x) \Downarrow (y \Downarrow y) \end{array}$$

Minimale Basen für die Schaltalgebra sind auf unterster Maschinenebene aus Gründen der Schaltkreis-Ökonomie von großer Bedeutung.

Zusammenfassend halten wir fest, daß die Substitution äquivalenter Terme genutzt werden kann

a) um komplizierte Terme zu vereinfachen

b) um den Wert eines Terms zu berechnen

c) um unerwünschte Operatoren zu eliminieren

d) um eine Termdarstellung bezüglich einer Basis von Operatoren zu erreichen.

Beispiel:

zu c) Zur Elimination von Operatoren können z.B. folgende Äquivalenzen verwendet werden:

$$\begin{array}{lrcl} \text{Implikation:} & x \Rightarrow y & \equiv & \neg x \vee y \\ \text{Äquivalenz:} & x \Leftrightarrow y & \equiv & (x \Rightarrow y) \wedge (y \Rightarrow x) \\ \text{Sheffer-Fkt.:} & x \Uparrow y & \equiv & \neg(x \wedge y) \\ \text{Peirce-Fkt.:} & x \Downarrow y & \equiv & \neg(x \vee y) \end{array}$$

In Abschnitt 2.3 wurde der Vorzug von Normalformen zur Repräsentation von Information hervorgehoben. Auch Boolesche Terme lassen sich in Normalformen überführen.

Definition: (Normalformen Boolescher Terme)

Ist $x \in X$ ein atomarer Boolescher Term und ist l_{ij}, $i = 1, \ldots, n$ und $j = 1, \ldots, m_i$, ein Literal, d.h. $l_{ij} = x$ oder $l_{ij} = \neg x$, so bilden die disjunktive Normalform

$$t_\vee = \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} l_{ij} \right)$$

bzw. die konjunktive Normalform

$$t_\wedge = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} l_{ij} \right)$$

kanonische Repräsentationen Boolescher Terme. Sie sind ineinander überführbar.

Die ausführliche Behandlung der Normalformen erfolgt in Kapitel 5. Ihr Nutzen ist darin begründet, daß die Äquivalenz von Termen aus ihrer Struktur erkannt werden kann, ohne alle Belegungen durchrechnen zu müssen.
Für Gleichungen semantisch äquivalenter Boolescher Terme gelten die klassischen Regeln der Äquivalenzrelationen.

Definition: (Regeln der Äquivalenz Boolescher Terme)

Seien t, t_1, t_2, t_3 beliebige Boolesche Terme. Zwischen diesen Termen besteht eine Äquivalenzrelation, wenn folgende Regeln erfüllt werden:

1. Reflexivität: $t = t$
2. Symmetrie: gilt $t_1 = t_2$, so gilt auch $t_2 = t_1$
3. Transitivität: gilt $t_1 = t_2$ und $t_2 = t_3$, so gilt auch $t_1 = t_3$.

Definition: (Reduktion)

Wegen der Transitivität der Äquivalenz Boolescher Terme $t_1, \ldots, t_n$, die durch iterative Anwendung von Gesetzen aus einem Startterm t_0 entsprechend $t_i = t_{i+1}$ für $i = 0, \ldots, n-1$ hervorgehen, entsteht eine Folge von Termen

$$t_0 = t_1 = \ldots = t_{n-1} = t_n.$$

die man Reduktion (oder Umformung) nennt.

Beispiele:

1. $y \wedge \neg y$
   $= \neg y \wedge y$ Kommutativgesetz
   $= \neg y \wedge \neg\neg y$ Involutionsgesetz
   $= \neg(y \vee \neg y)$ de Morgans Gesetz
   $= \neg 1$ Gesetz für TRUE
   $= 0$ Gesetz für FALSE
2. $x \wedge (\neg x \vee y)$
   $= (x \wedge \neg x) \vee (x \wedge y)$ Distributivgesetz
   $= (x \wedge y) \vee (x \wedge \neg x)$ Kommutativgesetz
   $= x \wedge y$ Neutralitätsgesetz

3.2.4 Aussagenlogik

Die *mathematische Logik* befaßt sich mit der Formalisierung des Schlußfolgerns, also der Aufgabe, aus einer Menge von Aussagen neue Aussagen abzuleiten. Hierbei spielt die Boolesche Algebra als Algebra der Wahrheitswerte "TRUE", "FALSE" eine zentrale Rolle, denn Aussagen nehmen einen dieser Wahrheitswerte an (sonst sind es keine Aussagen).

In der Logik bezeichnet man (zweiwertige) Terme der Sorte bool als Formeln. Grundterme der Sorte bool bezeichnet man als elementare Aussagen. Die Operatoren der Logik werden auch als *Junktoren* bezeichnet.

In der Logik werden Regelsysteme (Ableitungs- oder Inferenzregeln) angegeben, die es erlauben, aus einer Menge von als wahr angenommenen Formeln (Axiome) weitere Formeln (Theoreme) abzuleiten. Die Ableitung einer Formel heißt auch (formaler) Beweis der Formel.

Die *Aussagenlogik* beschränkt sich dabei auf Beweise für Formeln, die nur Terme der Sorte bool darstellen. Die *Prädikatenlogik* nimmt auch auf andere Sorten Bezug.

Fast alle bisher kennengelernten Termstrukturen der Booleschen Algebra sind als aussagenlogische Formeln interpretierbar (nicht solche, in denen Identifikatoren für Terme stehen). Diese erhalten ihre Bedeutung erst dadurch, daß alle vorkommenden atomaren Aussagen durch die Wahrheitswerte ersetzt (belegt) werden und auf die Semantik der Operatoren $\vee$, $\wedge$ und $\neg$ zurückgegriffen wird.

Definition: (aussagenlogische Formel)

Ein Term t der Sorte bool heißt aussagenlogische Formel, wenn t nur aus den Termen "TRUE", "FALSE", atomaren Aussagen, den logischen Operatoren (Junktoren) $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ und Identifikatoren $x_1, \ldots, x_n$ der Sorte bool aufgebaut ist. Für jede Belegung von $x_1, \ldots, x_n$ mit Wahrheitswerten kann einer aussagenlogischen Formel ein Wahrheitswert zugeordnet werden.

Beispiel:

Es sei $t = p \wedge \neg q \vee r$ mit $\beta(p) = 0$, $\beta(q) = 0$, $\beta(r) = 1$.
Dann gilt

$$
\begin{aligned}
I_\beta[p \wedge \neg q \vee r] &= \begin{cases} 1 & , \text{ wenn } I_\beta[p \wedge \neg q] = 1 \vee I_\beta[r] = 1 \\ 0 & , \text{ sonst} \end{cases} \\
&= \begin{cases} 1, & \text{wenn } I_\beta[p \wedge \neg q] = 1 \\ 1, & \text{wenn } I_\beta[r] = 1 \\ 0, & \text{sonst} \end{cases} \\
&= 1, \text{ da } I_\beta[r] = 1 \text{ und } I_\beta[p \wedge \neg q] = I_\beta[p] \wedge I_\beta[\neg q] = 0 \wedge 1 = 0
\end{aligned}
$$

Definition: (Modell einer Formel)

Eine Interpretation I_β heißt Modell einer Formel t bzw. einer Formelmenge $\mathcal{T}$, wenn $I_\beta = 1$ für alle $t \in \mathcal{T}$.

Definition: (Tautologie)

Eine Formel t heißt erfüllbar, wenn es ein Modell für t gibt, andernfalls heißt sie unerfüllbar (Kontradiktion) . Ist eine Formel für alle Belegungen β erfüllbar, heißt diese allgemeingültig oder eine Tautologie.

Beispiel:

1. $x \wedge \neg x$ ist eine unerfüllbare Formel, da $I_\beta[x \wedge \neg x] = 0$ für alle β

2. $x \vee \neg x$ ist eine Tautologie, da $I_\beta[x \vee \neg x] = 1$ für alle β.

Daraus folgt: **Eine Formel t ist genau dann allgemeingültig (oder Tautologie), wenn $\neg$t unerfüllbar ist (oder Kontradiktion).**

Aussagen sind entweder einfach oder enthalten Teilaussagen, die durch Junktoren verknüpft werden.

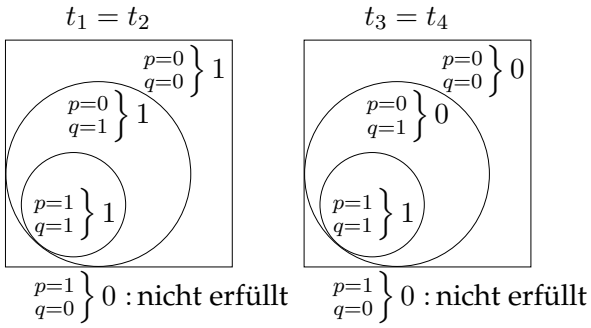
Die Reduktion von Termen stellt eine syntaktische Umformulierung von Aussagen dar, ohne an der Semantik etwas zu ändern.

Beispiel: Regen

p bezeichne die Aussage "es regnet"
q bezeichne die Aussage "die Straße ist naß"
$t_1 : \ p \Rightarrow q$, "aus p folgt q", "wenn p, dann q"
Also bezeichnet t_1 die Aussage "wenn es regnet, ist die Straße naß"
$t_2 : \ \neg p \vee q$, "es regnet nicht oder die Straße ist naß"
Es gilt: $t_1 \equiv t_2$ (durch Überprüfung aller Belegungen nachzurechnen)
$t_3 : \ p \wedge (p \Rightarrow q)$, "es regnet und wenn es regnet, ist die Straße naß"
$t_4 : \ p \wedge q$, "es regnet und die Straße ist naß"
Es gilt: $t_3 \equiv t_4$
t_1 und t_2 sind schwächere Aussagen als t_3 und t_4, da nicht ausgeschlossen ist, daß die Straße naß ist, obwohl es nicht regnet.

$\beta(p)$	$\beta(q)$	$I_\beta[t_1]$	$I_\beta[t_2]$	$I_\beta[t_3]$	$I_\beta[t_4]$
0	0	1	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	1	1	1	1	1

Also gilt $t_1 \Leftrightarrow t_2$ und $t_3 \Leftrightarrow t_4$.
Da die Terme t_1 und t_2 für mehr Belegungen erfüllt werden als die Terme t_3 und t_4, stehen sie für eine schwächere Aussage. Umgekehrt stehen t_3 und t_4 für eine stärkere bzw. spezifischere Aussage. Dies erkennt man auch im *Venn-Diagramm* einer mengentheoretischen Interpretation.



Beispiel: Party

„Meyer will eine Party geben und Anna, Berta, Carla oder Dora (kein ausschließendes Oder!) einladen. Jedoch gibt es einige Schwierigkeiten:
Anna (a) und Dora (d) mögen nicht am selben Tisch sitzen; wenn Carla (c) erfährt, daß Dora nicht eingeladen ist, sagt sie gleich ab. Und Berta (b) möchte, daß Anna mit ihr kommt. Welche Möglichkeiten, wenn überhaupt, hat Meyer?“

Zu erfüllenden Bedingungen, damit eine Aussageform t erfüllbar wird:

1. $\neg(a \wedge d) = \neg a \vee \neg d = a \Uparrow d$
2. $\neg(c \wedge \neg d) = \neg c \vee d = c \Rightarrow d$
3. $\neg(b \wedge \neg a) = \neg b \vee a = b \Rightarrow a$

Also muß die Aussage

$$\begin{aligned} t &= (a \Uparrow d) \wedge (c \Rightarrow d) \wedge (b \Rightarrow a) \\ &= \neg(a \wedge d) \wedge (\neg c \vee d) \wedge (\neg b \vee a) \end{aligned}$$

erfüllt werden. Gibt es hierfür Belegungen der Variablen (a, b, c, d), welche dies ermöglichen ?

	$\beta(a)\beta(b)\beta(c)\beta(d)$				$\boxed{1}$ $I_\beta[\neg(a \wedge d)]$	$\boxed{2}$ $I_\beta[\neg c \vee d]$	$\boxed{3}$ $I_\beta[\neg b \vee a]$	$\boxed{4}$ $\boxed{1} \wedge \boxed{2}$	$I_\beta[t]$ $\boxed{3} \wedge \boxed{4}$
0	0	0	0	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	1
2	0	0	1	0	1	0	1	0	0
3	0	0	1	1	1	1	1	1	1
4	0	1	0	0	1	1	0	1	0
5	0	1	0	1	1	1	0	1	0
6	0	1	1	0	1	0	0	0	0
7	0	1	1	1	1	1	0	1	0
8	1	0	0	0	1	1	1	1	1
9	1	0	0	1	0	1	1	0	0
10	1	0	1	0	1	0	1	0	0
11	1	0	1	1	0	1	1	0	0
12	1	1	0	0	1	1	1	1	1
13	1	1	0	1	0	1	1	0	0
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	0	1	1	0	0

Die folgenden Belegungen für (a,b,c,d) sind verträglich mit der *Erfüllbarkeit von t*: (0,0,0,0), (0,0,0,1), (0,0,1,1), (1,0,0,0), (1,1,0,0).

Meyer hat nur sehr eingeschränkte Möglichkeiten, seine Party zu organisieren:

- gar niemanden einladen
- Anna allein oder Dora allein einladen
- Anna und Berta einladen
- Carla und Dora einladen

Also liefert die obige Belegung ein *Modell für t* und es gilt t=1 für

$$t = t_1 \vee t_2 \vee t_3 \vee t_4 \vee t_5$$

mit

$$t_1 = \neg a \wedge \neg b \wedge \neg c \wedge \neg d$$
$$t_2 = \neg a \wedge \neg b \wedge \neg c \wedge d$$
$$t_3 = \neg a \wedge \neg b \wedge c \wedge d$$
$$t_4 = a \wedge \neg b \wedge \neg c \wedge \neg d$$
$$t_5 = a \wedge b \wedge \neg c \wedge \neg d$$

Beweis: Da $t_3 = \neg t_5$, also $t_3 \vee t_5 = 1$, gilt auch $t = t_1 \vee t_2 \vee t_4 \vee 1 = 1$.

Das Problem, ob für alle Terme t einer Formelmenge $\mathcal{T}$ gilt $I_\beta[t] = 1$ für *alle* Interpretationen I_β, die Modell für $\mathcal{T}$ sind, läßt sich effektiv durch logisches Schließen behandeln.

Definition: (logischer Schluß/Ableitung)

Gegeben seien eine Formelmenge bzw. eine Menge von Axiomen $\mathcal{T}$ (heißt auch Annahme oder Antezedent) und eine Formel t (heißt auch Folgerung oder Konsequenz). Ist $I[t] = 1$ für alle Interpretationen I_β, die Modell für $\mathcal{T}$ sind, schreibt man

$$\mathcal{T} \vdash t, \quad \text{"}t \text{ folgt semantisch aus } \mathcal{T}\text{"}$$

und nennt diese Operation einen logischen Schluß oder Ableitung.

Tautologien sind dadurch charakterisiert, daß sie aus der leeren Menge (ohne Annahmen) ableitbar sind: $\vdash t$.
Im folgenden werden einige wichtige Tautologien aufgeführt.

1. $\vdash ((\alpha \Rightarrow \beta) \Rightarrow \beta) \Leftrightarrow (a \vee \beta)$

2. $\vdash ((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha$ Peircesches Gesetz

3. $\vdash (\alpha \wedge (\alpha \Rightarrow \beta)) \Rightarrow \beta$ Tautologie vom Modus Ponens
   Bsp.: "es regnet und wenn es regnet ist die Straße naß" $\Rightarrow$ "die Straße ist naß"

Die Ableitung

$$t_1 \vdash t_2$$

impliziert, daß jede Belegung, die t_1 erfüllt, auch t_2 erfüllt. Dann heißt die Aussageform t_1 (semantisch) *stärker* oder *schärfer* als die Aussageform t_2 .

Das bedeutet, daß sich beim Übergang von t_1 nach t_2 nur Interpretationen der Aussageformen von FALSE (0) in TRUE (1) ändern dürfen.

Beispiel: Gesetz vom Dilemma

Der Term $a \vee b$ drückt ein Dilemma aus, das durch die schwächere Aussage c aufgelöst wird, wenn c sowohl zu a als auch zu b in einer Implikationsbeziehung steht. $(a \vee b) \wedge ((a \Rightarrow c) \wedge (b \Rightarrow c)) \vdash c$

			$\boxed{1}$	$\boxed{2}$	$\boxed{3}$	$\boxed{4}$		
$\beta(a)$	$\beta(b)$	$\beta(c)$	$I_\beta[a \vee b]$	$I_\beta[a \Rightarrow c]$	$I_\beta[b \Rightarrow c]$	$\boxed{2} \wedge \boxed{3}$	$\boxed{1} \wedge \boxed{4}$	$\beta(c)$
0	0	0	0	0	1	1	1	0
0	0	1	0	$\boxed{0}$	1	1	1	$\boxed{1}$
0	1	0	1	0	1	0	0	0
0	1	1	1	1	1	1	1	1
1	0	0	1	0	0	0	1	0
1	0	1	1	1	1	1	1	1
1	1	0	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1

Der durch die Ableitung $t_1 \vdash t_2$ beschriebene Zusammenhang für die Interpretation bei beliebigen Belegungen heißt auch *Wertverlaufsinklusion*.

Ein logischer Schluß $t_1 \vdash t_2$ stellt eine *semantische Folgerungsbeziehung* zwischen einer Menge von Annahme $\mathcal{T}$ und einer Folgerung t in einer Situation dar. Diese semantische Folgerungsbeziehung versucht man in der Logik durch eine *Folge syntaktischer Folgerungsbeziehungen* zu erfassen. Diese Folge heißt *Kalkül*.

Definition: (Kalkül)

Ein Kalkül K ist ein algorithmisches Verfahren, mit dem man logische Schlüsse syntaktisch in endlich vielen Schritten ableiten kann, ohne den Weg über die Interpretation der Formeln zu gehen.

Ausgehend von einer Menge $\mathcal{T}$ von Annahmen, die als richtig vorausgesetzt werden, schreibt man eine solche Ableitung als endliche Folge

$$S = [t_1, \ldots, t_m]$$

syntaktisch korrekter Formeln t_i, $i = 1, \ldots, m$, wobei für alle t_i gilt

$$\{t_1, \ldots, t_{i-1}\} \vdash t_i.$$

Schema einer Ableitung $S = [t_1, \ldots, t_{m+1}]$ über $\mathcal{T}$:

$$\begin{array}{ll} \mathcal{T} & \vdash t_1 \\ \mathcal{T} \cup \{t_1\} & \vdash t_2 \\ \quad \vdots & \\ \mathcal{T} \cup \{t_1, \ldots, t_m\} & \vdash t_{m+1} \end{array}$$

Ein Kalkül besteht aus Regeln (Schlußregeln, Inferenzregeln). Dabei spielen die Äquivalenzen der Booleschen Algebra eine zentrale Rolle.

- Ein Kalkül muß korrekt sein: Nur solche Formeln t dürfen sich in ihm herleiten lassen, die semantische Konsequenz der Annahmen $\mathcal{T}$ sind.
- Ein Kalkül ist vollständig, wenn sich aus der Formelmenge $\mathcal{T}$ alle Formeln herleiten lassen, die semantische Konsequenz von $\mathcal{T}$ sind.
- Ist ein Kalkül korrekt und vollständig, dann sind syntaktische Folgerungsbeziehungen und semantische Folgerungsbeziehungen gleichwertig.

Da Tautologien selbst keine Annahmen erfordern, eignen sie sich als Axiome $\mathcal{T}$ eines Ableitungsschemas (Kalkül).

Ableitungsregeln der Aussagenlogik

Es werden beispielhaft drei verschieden Arten von Ableitungsregeln angegeben:

1. Tautologien, z.B.
   $\vdash t \vee \neg t$ Tertium non datur (Gesetz vom ausgeschlossenen Dritten)
2. Wertverlaufsinklusionen, z.B.
   $\{t_1 \wedge (\neg t_1 \vee t_2)\} \vdash t_2$
   bzw. Modus Ponens
   $\{t_1 \wedge (t_1 \Rightarrow t_2)\} \vdash t_2$
3. Identitäten:
   Ist $t_1 \equiv t_2$ die Anwendung einer Regel der semantischen Äquivalenz Boolescher Terme, so gilt auch
   $\{t_1\} \vdash t_2$ Anwendung der Gleichheitsgesetze

Die Ableitungsregel des Modus Ponens

$$\alpha \wedge (\alpha \Rightarrow \beta) \vdash \beta$$

folgt aus der Tautologie des Modus Ponens

$$\vdash (\alpha \wedge (\alpha \Rightarrow \beta)) \Rightarrow \beta$$

als Folge der Wertverlaufsinklusion.
Es gilt die

Stärker-Regel der Implikation

$\vdash t_1 \Rightarrow t_2$ genau dann, wenn $t_1 \vdash t_2$.

Dann gilt auch folgender Zusammenhang zwischen Implikation und Ableitbarkeit: Gilt $\mathcal{T} \vdash t_1 \Rightarrow t_2$, dann gilt auch $\mathcal{T} \cup \{t_1\} \vdash t_2$.

3.3 Algorithmen

Auf dem Weg der Formulierung eines Problems durch ein Programm haben wir bisher gelernt:

- Das Problem muß durch fortschreitende Formalisierung in eine Gestalt transformiert werden, die sicherstellt, daß das Problem maschinell lösbar wird. Voraussetzung hierfür ist, daß das Problem berechenbar ist. Andernfalls muß es in ein berechenbares Ersatzproblem transformiert werden (z.B. als Approximation).
- Zielstellung der Spezifikation ist die Festlegung der Eingabe- und Ausgabebereiche und des funktionalen Zusammenhangs zwischen diesen. Hierin liegt ein erheblicher Teil der Arbeit zur Aufbereitung der Problemlösung.

  Die Spezifizierung des Verfahrens der Problemlösung erfolgt durch den Algorithmenentwurf.

  In diesem Abschnitt werden hierzu einführende Sachverhalte aufbereitet. Im 3. Semester erfolgt eine systematische Einführung in die Prinzipien des Algorithmenentwurfes und die Beziehung zu dabei verwendeten Datenstrukturen.
- Bei der Problemlösung hat der Programmierer die Freiheit, eigene Datenstrukturen zu entwerfen. Andererseits kann er/muß er die Ressourcen der Programmiersprache/Maschine nutzen. Diese elementaren Datenstrukturen heißen Rechenstrukturen. Diese werden durch die verfügbaren Sorten und Funktionen gebildet.
- Die Rechenstruktur der Wahrheitswerte ist deshalb von grundlegender Bedeutung, weil sie gestattet,
  - Programmstrukturen auf Schaltwerke abzubilden, um sie mit ihrer Hilfe abzuarbeiten und
  - Berechnung mit Aussagenlogik zu verschmelzen.

3.3.1 Strukturierungsmethoden - Notationen von Algorithmen

Definition: (Algorithmus)

Ein Algorithmus besteht aus einer Menge von Regeln für ein Verfahren, um aus gewissen Eingabegrößen bestimmte Ausgabegrößen herzuleiten. Dabei müssen folgende Bedingungen erfüllt sein:

1. *Endlichkeit:* Die Beschreibung erfordert eine endliche Anzahl von Schritten,
2. *Effektivität*: Jeder Schritt des Verfahrens muß ausführbar sein,
3. *Terminiertheit*: Das Verfahren kommt nach endlich vielen Schritten zu einem Ende,
4. *Determiniertheit*: Der Ablauf des Verfahrens ist zu jedem Zeitpunkt festgelegt.

Die Punkte 3 und 4 der obigen Definition werden manchmal nicht gefordert (z.B. autonomer Roboter).
Beispiele für "unendliche Texte" sind z.B.

1. Eine vollständige Beschreibung eines Bildes
2. "Berechne Kettenbruch φ":

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\cdots}}}}$$

Effektivität heißt prinzipielle Machbarkeit und bedeutet nicht *Effizienz* (wirtschaftlich vernünftige Machbarkeit).

Beispiel:

- nicht effektiv: "Berechne $\frac{1}{x} \quad \forall x \in \mathbb{N}$!"
- effektiv aber nicht effizient:
  "Suche den besten Folgezug eines gegebenen Schachspieles durch Berechnen aller möglicher Fortsetzungen bis zum Spielende!"

Wir betrachten hier den Algorithmenentwurf auf der Basis einer funktionellen Spezifikation unter dem Gesichtspunkt, daß der Algorithmus auf einer *virtuellen Maschine* ausführbar ist.

Zur Erinnerung: Eine virtuelle Maschine ist ein Mechanismus zur Ausführung bestimmter Operationen auf Datenobjekten, wobei diese einem bestimmten Datentyp entsprechen. Dabei hat eine virtuelle Maschine nur eine endliche Zahl socher Typen und kann nur eine endliche Zahl von Objekten speichern.

Ist die virtuelle Maschine des Anwendungsprogramms (*Quellmaschine*) nicht identisch mit der realen Maschine, auf welcher der Algorithmus laufen soll (*Zielmaschine*), so bedeutet ein Algorithmenentwurf die **Konstruktion einer Folge virtueller Maschinen solange, bis die Datenobjekte und Operationen auf der realen Maschine bekannt sind.**

Stellt die Zielmaschine die Datentypen der Ein- und Ausgabe zur Verfügung und ist ihr die zu berechnende Funktion bekannt, so ist keine Algorithmenentwicklung in mehreren Schritten notwendig, man spricht von einem *elementaren* Algorithmus (z.B. Taschenrechner).
Andernfalls lassen sich die Schritte des Algorithmenentwurfs folgendermaßen schematisch darstellen:

1. Realisierung (Implementierung) der Quellmaschine auf Zwischenmaschine 1

$\vdots$

n. Implementierung der Zwischenmaschine n auf der Zielmaschine.

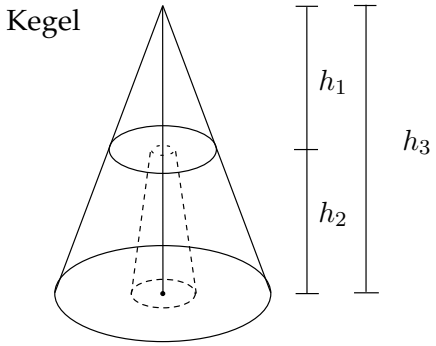
Es wird zwischen zwei grundsätzlichen Zugängen zum Algorithmenentwurf unterschieden:

1. *Top-Down-Methode*
   Entwicklung einer Folge virtueller Maschinen durch schrittweise Verfeinerung der Quellmaschine, d.h. ausgehend von der Problemspezifikation schrittweise Definition von Teilproblemen und Angabe der Unteralgorithmen.

2. *Bottom-Up-Methode*
   Entwicklung einer Folge virtueller Maschinen durch schrittweise Verallgemeinerung der Zielmaschine, d.h. ausgehend von einer konkreten Implementierung eines Problems wird eine schrittweise Einbettung in ein allgemeineres Problem vorgenommen, so daß die erreichte Quellmaschine auch die ursprüngliche Problematik als Teilproblem enthält.

Beispiel: A1/1

zu 1: Berechnung des Volumens eines Kegel- oder Pyramidenstumpfes



$$h_3 = h_1 + h_2$$

a) Zurückführung auf Volumina von Kegel und Pyramide:

$$V_{KS} = V_{K_3} - V_{K_1} \qquad V_{PS} = V_{P_3} - V_{P_1}$$

b) Zurückführung der Volumenberechnung auf Flächenberechnung:

$$V = F \cdot \frac{h}{3} \qquad F_{K_i} = \pi r_i^2$$

zu 2: Erweiterung des Programms zur Berechnung eines Hohl-Kegelstumpfes/Pyramidenstumpfes:

$$V_{HKS} = V_{KS}^a - V_{KS}^i$$

In der Praxis werden beide Methoden gemischt angewendet, die Top-Down-Methode zeichnet sich jedoch durch ein höheres Maß an innerer Logik aus.

Definition: (Top-Down-Entwicklungsmethode)

Die Entwicklung eines Algorithmus nach dem Top-Down-Verfahren (durch schrittweise *Verfeinerung*) findet in folgenden Schritten statt:

1. Schreibe eine funktionale Spezifikation für das Problem.
2. Zerlege das Problem in Unterprobleme und schreiben für diese ebenfalls Spezifikationen.
3. Beschreibe, wie sich die Lösungen der Unterprobleme zu einer Gesamtlösung kombinieren lassen.
   - Beweise die Korrektheit der Gesamtlösung unter der Voraussetzung, daß die Teillösungen korrekt sind.
   - Schätze den Aufwand des Algorithmus ab.
4. Entwerfe nach dem gleichen Schema Unteralgorithmen für die nicht-elementaren Unterprobleme.

- *Verifikation von Algorithmen*: Überprüfung der Korrektheit
  Ein korrekter Algorithmus wird bei Einhalten der Vorbedingungen und bei strikter Befolgung der Rechenvorschriften die Gültigkeit der Nachbedingungen implizieren. Durch Testen kann man nur die Anwesenheit von Fehlern nachweisen, nicht aber deren Abwesenheit!
  Deshalb sollte zusätzlich zu Tests ein formaler Beweis der Korrektheit geführt werden.

- *Komplexität von Algorithmen*: Überprüfung des Rechenaufwandes
  Aufwandsabschätzungen für Algorithmen erfolgen durch Zählen der Rechenschritte. Dabei wird zwischen minimalem (*best case*), maximalen (*worst case*) und mittlerem Aufwand unterschieden.
  Der mittlere Aufwand ist stark abhängig von den angebotenen Daten. Deshalb müssen hierüber Annahmen gemacht werden.
  Oft ist man nur an der Skalierung des Aufwandes bei Skalierung der Datenmenge interessiert. Hierüber gibt die *asymptotische Rechenzeit* Auskunft, die die Proportionalität der Anzahl der Rechenschritte zu Funktionenklassen angibt. So unterscheidet man z.B. logarithmische, lineare, polynomiale oder exponentielle Zeitkomplexität. Unter der Annahme, daß das Argument eines Algorithmus (einer Funktion) die Größe n hat, weisen die Angaben $O(n), O(\log n), O(n^p)$ oder $O(e^n)$ auf lineare, logarithmische, ... asymptotische Zeitkomplexität hin. Bei linearer Skalierung des Arguments skaliert der Berechnungsaufwand im angegebenen Sinn. Der Unterschied im Aufwand zur Berechnung wird in folgender Tabelle deutlich:

n	$\mathrm{ld}\, n$	n	$n \,\mathrm{ld}\, n$	n^2	e^n
10^0	0	1	0	10^0	3
10^1	3	10	$3 \cdot 10^1$	10^2	$2 \cdot 10^4$
10^2	6	100	$6 \cdot 10^2$	10^4	$3 \cdot 10^{43}$
10^3	9	1000	$9 \cdot 10^3$	10^6	$2 \cdot 10^{434}$
10^4	13	10000	$19 \cdot 10^4$	10^8	$9 \cdot 10^{4342}$
10^5	16	100000	$16 \cdot 10^5$	10^{10}	10^{43420}
10^6	19	1000000	$19 \cdot 10^6$	10^{12}	10^{434200}

Die Überprüfung des Rechenaufwandes und die Verifikation von Algorithmen werden im 3. Semester behandelt.

Beispiel: A2/1: Suchproblem

Bestimme die Position $P(F, a)$ eines Elements a in einer Folge F. Falls a in F nicht vorkommt, sei $P(F, a) = 0$ (vgl. auch Spezifikation auf Seite 110).

Deklarationen:
- Typen: S sei eine Menge
  $\mathcal{F}(S)$ sei die Menge der endlichen Folgen über S
- Funktionen: $P : \mathcal{F}(S) \times S \to \mathbb{N}$

Eingabe: $F = (A_1, \ldots, A_n) \in \mathcal{F}(S),\ a \in S,\ A_i \in S$ für $1 \leq i \leq n$
Sei $n \geq 1$ (d.h. Folge nicht leer)

Variablen: $p \in \mathbb{N}$

Vorbedingung: $A_i \neq A_j$ für $i \neq j$

Verfahren:
1. Wähle eine erste Suchposition p.
2. Falls $a = A_p$, brich die Rechnung ab.
   Andernfalls, wenn noch eine Suchposition existiert, wähle eine neue Suchposition p und wiederhole 2.
   Wenn keine neue Suchposition mehr existiert, setze $p = 0$ und brich die Rechnung ab.

Ausgabe: $P(F, a) := p$

Nachbedingung: $(\forall j \in \{1, \ldots, n\})(j \neq p \Rightarrow a \neq A_j) \wedge (a = A_p \vee p = 0)$

Die Umformulierung der Nebenbedingung gegenüber der Spezifikation auf Seite 110 zeigt, daß die Reihenfolge der Vergleiche mit dem Element a nicht von Bedeutung ist. Vielmehr ist wichtig,

- daß zu jedem Zeitpunkt gilt, daß die bereits betrachteten Elemente verschieden von a sind und
- daß die Suche eingestellt wird, wenn
  – alle Elemente betrachtet wurden oder
  – ein $A_p = a$ gefunden wurde.

Es wurde eine Variable p als Hilfsvariable eingeführt, deren Geltungsbereich lokal innerhalb des Algorithmus ist. Sie dient gleichzeitig als Ausgabevariable.
Es werden drei Unterprobleme identifiziert

1. Wähle eine erste Suchposition.
2. Existiert eine neue Position?
3. Wähle eine neue Position.

Dabei stellen sich folgende Fragen:

1. Welches Unterproblem muß zuerst bearbeitet werden?
2. In welcher Form wird die Folge der Eingabedaten angeboten, bzw. wie ist die neue Position zu wählen?
3. Mit welchen Strukturelementen sind die Unterprobleme verbindbar?

zu 1.:

Regel der minimalen Festlegung

Es wird dasjenige Unterproblem zuerst bearbeitet, dessen Lösung von den übrigen am wenigsten abhängt und deshalb deren Lösung am wenigsten festlegt.

In Beispiel 2A/1 ist Unterproblem 1 ("Wähle eine erste Suchposition.") das am wenigsten abhängige Unterproblem.

zu 2.: Als Annahme sei ein sequentieller Zugriff (d.h. kein freier wahlweiser Direktzugriff) möglich.

Dann bieten sich zwei Vorgehensweisen an:

Wähle erste Pos.	Existiert neue Pos.?	Wähle neue Pos.
Setze p auf 1	$p < n$?	Ersetze p durch $p + 1$
Setze p auf n	$p > 1$?	Ersetze p durch $p - 1$

Wir entscheiden uns für die zweite Möglichkeit und legen als erste Position $p = n$ fest. Die Überprüfung "Existiert neue Position" wird umformuliert zu "Existiert keine neue Position?". Dann folgt automatisch $p = 0$, wenn $a \notin F$. Das Verfahren läßt sich nun präziser angeben:

Beispiel: A2/2: sequentielle Suche

Verfahren:

1. Setze p auf n.
2. Falls $a = A_p$ gilt, brich die Rechnung ab.
   Andernfalls ersetze p durch $p - 1$.
   Falls $p = 0$ gilt, brich die Rechnung ab,
   sonst wiederhole Schritt 2.

zu 3: Die Formulierung von Algorithmen erfordert in erster Linie Ablaufstrukturen erzeugende Ausdrucksmittel. Beispiele für solche Ausdrucksmittel sind

a) Pseudocode
b) Ablaufdiagramme (graphisches Hilfsmittel)
c) Strukturdiagramme (graphisches Hilfsmittel)

a) Pseudocode

Im Pseudocode werden neben Anweisungen zur Beschreibung von Verzweigungen und Schleifen natürlichsprachliche Sätze zugelassen. Solche Befehlssätze beginnen mit einem Verb, haben keine Nebensätze und dürfen nicht "wenn" und "aber" enthalten. Zur Definition des Pseudocode siehe nächste Seite.

Beispiel: A2/3: Suchproblem in Pseudocode

Verfahren:

```
Wähle eine erste Suchposition p.
while a ≠ A_p do
    if Es gibt noch eine neue Suchposition then
        Wähle eine neue Suchposition.
    else
        Setze p auf 0.
        exit
    endif
endwhile
```

Beispiel: A2/4

Verfahren:

```
p := n     /* Wähle eine erste Suchposition */
while a ≠ A_p do
    p := p − 1     /* Wähle eine neue Suchposition */
    if p = 0 then     /* Es gibt keine neue Suchpos. */
        exit
    endif
endwhile
```

Definition: (Pseudocode)

Seien $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$ Anweisungen und π ein Aussageterm oder elementarer Aussagesatz der natürlichen Sprache.
Ein Algorithmus in Pseudocode ist eine endliche Folge von Anweisungen der folgenden Art:

1. einfacher Befehlssatz
2. Wertzuweisung $x := t$ "x wird t"
   x ist eine deklarierte Variable, t ist ein Term über den deklarierten Konstanten, Variablen und Funktionen
3. exit
4. while-Anweisung
   while π do
   α_1
   $\vdots$
   α_n
   endwhile
5. repeat-Anweisung
   repeat
   α_1
   $\vdots$
   α_n
   until π
6. if-Anweisung

if π then	oder	if π then
α_1		α_1
$\vdots$		$\vdots$
α_n		α_n
endif		else
		β_1
		$\vdots$
		β_m
		endif

7. Kommentare, Bemerkungen: /* Text */

Rechenvorgänge bei der Ausführung des Pseudocodes:

1. einfacher Befehlssatz: wird ausgeführt
2. Wertzuweisung: Ermittle den Wert des Terms t und notiere ihn anschließend als neuen Wert der Variablen x.
   Achtung: definiert der Term t eine Funktion, so kann er diese auch wieder als Argument enthalten (Rekursion). Dieser Fall erfordert eine spezielle Interpretation des Rechenvorgangs (Abschnitt 3.3.2).
3. exit : Kommt exit im Rumpf einer while - oder repeat -Anweisung vor, so wird diese beendet. Kommt exit außerhalb jeder while - oder repeat -Anweisung vor, so wird der gesamte Rechenvorgang beendet.
4. while -Anweisung: Werte den Term π aus. Im Falle FALSE ist die while -Anweisung beendet. Sonst Ausführung der Rumpf-Anweisungen und erneutes Ausführen der gesamten while -Anweisung.
   Achtung: Endlosschleifen können auftreten!
5. repeat -Anweisung: Führe die Rumpfanweisungen aus. Anschließend werte den Term π aus. Im Falle FALSE erneute Ausführung der repeat -Anweisung, im Falle TRUE ist die repeat -Anweisung beendet.
   Achtung: Endlosschleifen können auftreten!
6. if -Anweisung: Werte den Term π aus. Im Falle TRUE wird der Rumpf des then -Zweiges abgearbeitet. Danach ist die if -Anweisung beendet.
   Andernfalls
   a) wenn if -Anweisung ohne else -Zweig: Ausführung der if -Anweisung ist beendet.
   b) wenn if -Anweisung mit else -Zweig: Ausführung der Anweisungen im else -Zweig und Ende.

Bei Wertzuweisungen der Form $x := x - 1$ ist rechts immer der momentane Wert des Terms gemeint, welcher der links stehenden Variablen zugewiesen wird.

Zur Darstellung der Ablaufstrukturen eines Algorithmus mittels graphischer Hilfsmittel wird ein weiteres Beispiel (“Berechnung der Quadratwurzel”) verwendet. An diesem Beispiel wird auch eine kompaktere Schreibweise eines Algorithmus im Sinne der Spezifikation und Definition einer Funktion eingeführt.

Definition: (Funktionsdefinition)

Eine Funktionsdefinition genügt folgender Syntax:

def ≪Name≫(≪formale Parameter≫) ≡ ≪Rumpf≫.

Dabei stehen `Name` für den Namen der Funktion, `formale Parameter` für eine durch Kommata getrennte Liste von Identifikatoren und `Rumpf` ist Ausdruck (Term) der Funktion, in dem die Parameter als Identifikatoren auftreten.

Die Funktionsdefinition entspricht dem Verfahren des Algorithmus. Die Parameterliste ist nur innerhalb des Rumpfes von Belang und kann dort sowie in der Deklaration ≪`formale Parameter`≫ beliebig ausgetauscht werden. Das heißt, die Namensgebung kann frei erfolgen.

Beispiel: A1/2

def zylindervolumen (radius, höhe)≡ (radius · radius · pi · höhe)

oder

def zvol(r,h)≡(r · r · pi · h)

Nach dem Prinzip der schrittweisen Verfeinerung können im Rumpf einer Funktion wieder Funktionen auftreten (als Funktionsanwendungen). Dies erfordert natürlich, daß diese Hilfsfunktionen vorher ebenfalls definiert wurden.

Beispiel: A1/3

def Kreisfläche(radius) ≡ (radius · radius · pi)
def zvol(r,h) ≡ (Kreisfläche(r) · h)

Definition: (Funktionsanwendung)

Eine Funktionsanwendung stellt den Aufruf einer Funktion (im Rumpf einer anderen Funktion) dar und hat die Form

≪Name≫(≪Argumente≫),

wobei `Argumente` eine durch Kommata getrennte Liste von Termen (auch Konstanten) ist.

Wenn wir eine Funktion nutzen wollen, um eine andere Funktion darauf aufzubauen (wie im obigen Beispiel), interessiert nicht die Gestaltung des Rumpfes der Hilfsfunktion. Es muß nur die Schnittstelle klar definiert werden. Hierzu reicht die Angabe der Funktionalität und der Spezifikation (Vor- und Nachbedingung).

Beispiel: A1/4

fct zvol: real $\times$ real $\rightarrow$ real

In vielen Programmiersprachen wird die Angabe der Funktionalität mit der Definition der Funktion verschmolzen.

Beispiel: A1/5

function zvol(r: real, h: real): real;
(Kreisfläche(r) $\cdot$ h)

Wie bereits bekannt, legt die Spezifikation einer Funktion die Beziehung zwischen Argumentwerten und Resultatwerten fest.

Definition: (Spezifikation einer Funktion)

Die Spezifikation einer Funktion hat die Form

spc $f(x_1, \ldots, x_m) \equiv < z_1, \ldots, z_n >$:
pre $P[x_1, \ldots, x_m]$
post $R[x_1, \ldots, x_m, z_1, \ldots, z_n]$,

wobei die Vorbedingung P die Einschränkungen festlegt, denen die Argumente der Funktion f genügen müssen (ohne Einschränkungen ist P =TRUE). Die Nachbedingung R beschreibt die Beziehung zwischen den Argumenten und den Resultaten.

Die Spezifikation
spc $f(x) \equiv z$:
pre $P[x]$
post $R[x, z]$
entspricht der Aussage für eine korrekte Funktion
$(f(x) \equiv z) \Rightarrow (P[x] \Rightarrow R[x, z])$
oder
$(P[x] \wedge (f(x) \equiv z)) \Rightarrow R[x, z]$. (Modus Ponens)

Beispiel: A3/1: Berechnung der Quadratwurzel

Erinnerung: Die Lösung der quadratischen Gleicung $ax^2+bx+c=0$ erfolgt für reelle Lösungen nach dem Schema

$$x_{1/2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}\,;\ a \neq 0$$

Hilfsvariable: $d = b^2 - 4ac$ (Diskriminante)

spc wurz $(a, b, c) \equiv x$:
  pre $b \cdot b \geq 4 \cdot a \cdot c \ \wedge a \neq 0$
  post $a \cdot x \cdot x + b \cdot x + c = 0.$

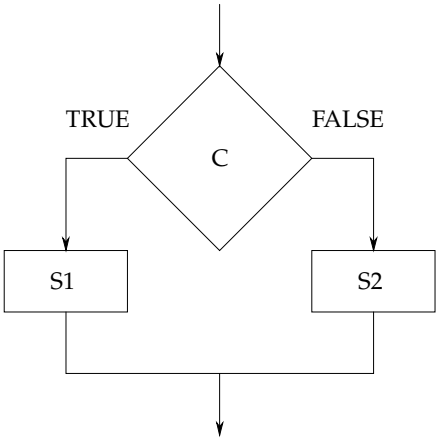
def wurz $(a, b, c) \equiv (x_1 \wedge x_2)$
  with $x_1 \equiv (-b + \sqrt{d})/(2 \cdot a)$,
  $x_2 \equiv (-b - \sqrt{d})/(2 \cdot a)$,
  where $d \equiv b \cdot b - 4 \cdot a \cdot c$

b) Ablaufpläne (Programmablaufpläne, Flußdiagramme)

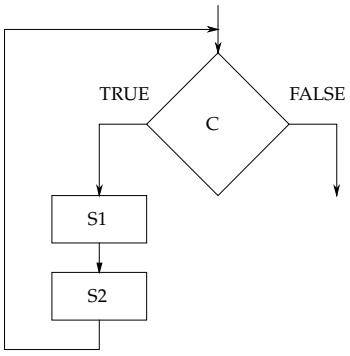
Ein Ablaufplan ist ein gerichteter Graph mit verschiedenen Arten von Knoten (Anweisungen, Verzweigungen, Ein-/Ausgabe), die mit bestimmten Ausdrücken bzw. Anweisungen markiert sind. Ablaufpläne sind gut zur Darstellung einfacher algorithmischer Abläufe geeignet. Nach DIN 66001 sind die in der Tabelle (3.2) dargestellten graphischen Symbole verwendbar.

Verzweigungen und Schleifen werden in folgender Weise dargestellt:

if C then S1 else S2 endif



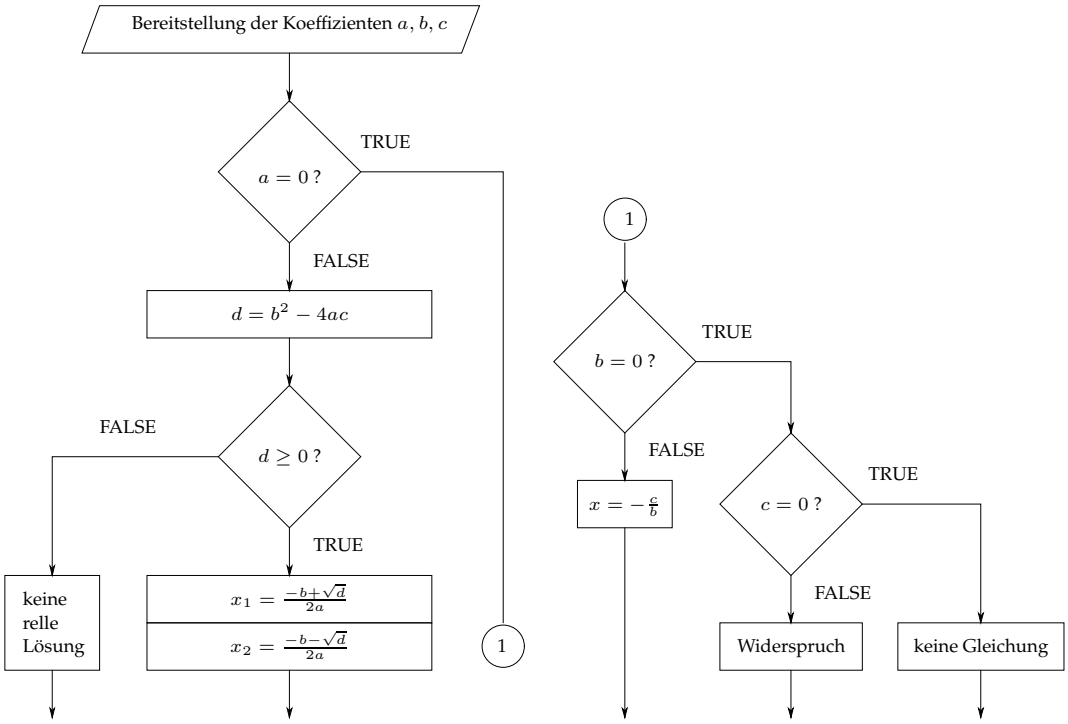
while C do S1; S2 endwhile



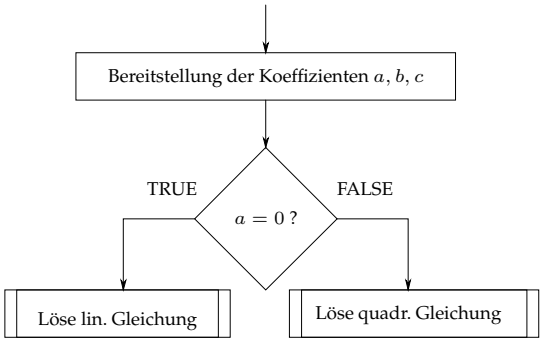
Elemente	Erläuterung
	**Operation;** allgemein, insbesondere soweit in folgenden 4 Sinnbildern nicht erfaßt
	**Verzweigung;** Stelle im Ablaufplan, an der, unter Beachtung einer Bedingung, eine unter mehreren möglichen Fortsetzungen eingeschlagen werden muß
	**Aufruf eines Unterprogramms;** dieses Unterprogramm wird normalerweise in einem weiteren Ablaufplan beschrieben sein
	**Programmodifikation;** z.B. Stellen von programmierten Schaltern, das Ändern von Indexregister oder das Modifizieren des Programms selbst (nicht empfehlenswert!)
	**Operation von Hand;** im allgemeinen mit einem Warten verbunden
	**Eingabe, Ausgabe**
	Ablauflinie; Vorzugsrichtungen von oben nach unten und von links nach rechts: Pfeilspitze zum nächstfolgenden Sinnbild möglich, bei Abweichung von den Vorzugsrichtungen erforderlich
	**Zusammenführung;** gleiche Fortsetzung von verschiedenen Stellen im Ablaufplan her
	**Übergangsstelle;** gleiche Bezeichnung zusammengehöriger Stellen, zur Verbindung getrennter Ablauflienien
	**Grenzstelle;** d.h. Anfang und Ende des Ablaufs, aber auch Programmunterbrechung
	Synchronisation bei Parallelbetrieb; insbesondere in den drei folgenden Fällen:
	**Aufspaltung bei Parallelbetrieb**
	**Sammlung bei Parallelbetrieb**
	**Synchronisationsschnitt**
	**Bemerkung;** kann an jedes andere Sinnbild angefügt werden

Tabelle 3.2: Sinnbilder für Programmablaufpläne nach DIN 66001

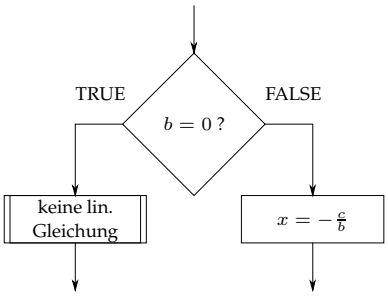
Beispiel: A3/2: Lösung einer quadratischen Gleichung



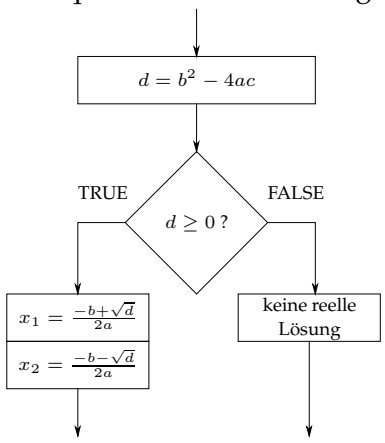
mit Schrittweiser Verfeinerung:



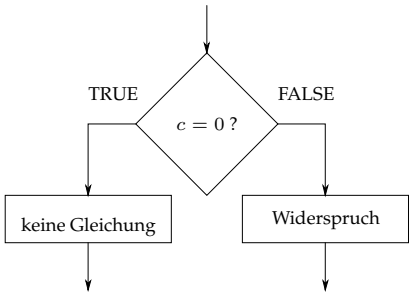
Löse lineare Gleichung:



Löse quadratische Gleichung:



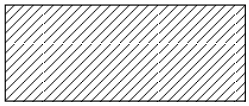
keine lineare Gleichung:



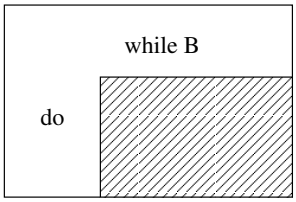
c) Struktogramme (Nassi-Shneiderman-Diagramme)

Struktogramme erlauben eine kompakte Darstellung, die gut der schrittweisen Verfeinerung folgt. Sie setzen sich aus folgenden Grundbildern zusammen, die beliebig geschachtelt werden können:

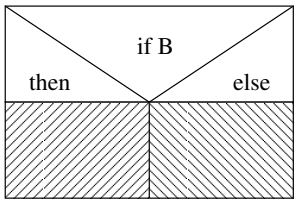
Anweisungen



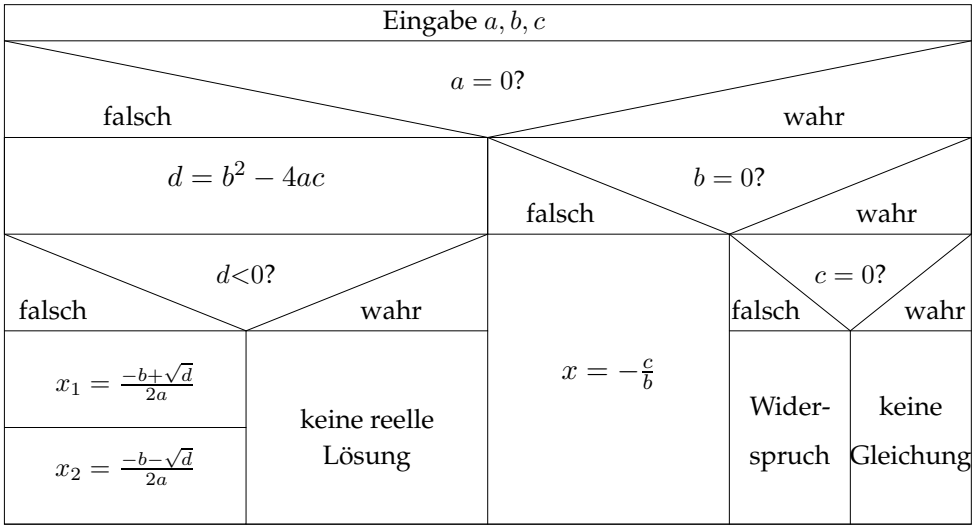
Wiederholungen



Alternativen



Beispiel: A3/3



3.3.2 Rekursion

Die bisher eingeführten Ausdrucksmittel für Algorithmen implizieren, daß endlich beschränkte Berechnungen ausgeführt werden, solange nur die Berechnungen der verwendeten Basisfunktionen terminieren. Damit ist die Länge der Berechnungssequenzen stets statisch beschränkt.

Die rekursive Definition von Funktionen ist ein mächtiges Werkzeug, das auf die Verknüpfung endlich langer Rechenvorschriften mit unbeschränkt langen Berechnungssequenzen zurückgreift.

Definition: (rekursive Funktionsdefinition)

Eine Funktionsdefinition

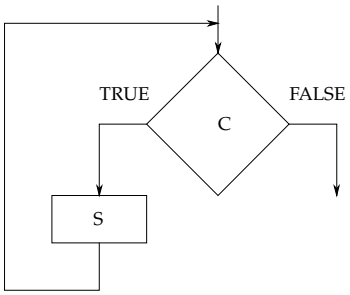
def ≪Name≫(≪formale Parameter≫) ≡ ≪Name ∪ Rumpf*≫
Rumpf = Rumpf* ∪ Name

heißt rekursiv, wenn im Rumpf der Funktion deren Name als vereinbarter Identifikator wieder vorkommt.

Alle nicht rekursiv formulierten Algorithmen lassen sich rekursiv umformulieren.

Beispiel: Rekursion

```
def proc ≡
        if C then
            S
            proc
        endif
```



Der Aufruf dieser Funktion proc bewirkt, daß bei jedem Erreichen der Rumpf-Anweisung proc eine neue *Inkarnation* der Funktion erzeugt wird.
Diese spezielle Arte der Rekursion heißt auch iterative Rekursion.

Definition: (iterative/repetitive Rekursion)

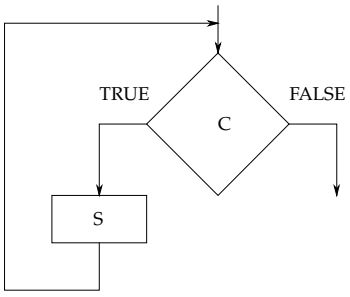
Eine iterative/repetitive Rekursion ist dadurch ausgezeichnet, daß der rekursive Aufruf der Funktion jeweils die letzte Aktion jeder Inkarnation ist.

Eine iterative Rekursion ist äquivalent einer while -Anweisung.

Beispiel:

Nicht rekursive Umformulierung des vorherigen Beispiels

```
def proc ≡
        while C do
            S
        endwhile
```



Im folgenden wollen wir eine Reformulierung des Suchproblems (Beispiel A2/4) als rekursiven Algorithmus vornehmen:

Zu Beginn des Verfahrens erfolgt die Wahl einer beliebigen Suchpositon p. Wenn $A_p \neq a$, dann wird weiteres Suchen in den Teilfolgen $(A_1, \ldots, A_{p-1})$ und $(A_{p+1}, \ldots, A_n)$ gestartet, also zwei Inkarnationen des Problems in kleinerem Maßstab erzeugt. Dadurch ist auch die Möglichkeit der Parallelisierung gegeben.

Beispiel: A2/5: rekursives Suchproblem (Zwischenstufe)

Bestimme die Position $s(F, a, l, r)$ eines Elementes a in einer Folge F zwischen den Positionen l und r (linker und rechter Rand des Suchraums)

Deklarationen:
- Typen: siehe Beispiel A2/1
- Funktionen: $s: \mathcal{F}(S) \times S \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (Suchfunktion)

Variablen: $p, p' \in \mathbb{N}$ (Positionen in den Teilfolgen)

Eingaben:
1. $F = (A_1, \ldots, A_n) \in \mathcal{F}(S),\ n \geq 1$
2. $a \in S$
3. $l \in \mathbb{N}$ (linker Suchrand)
4. $r \in \mathbb{N}$ (rechter Suchrand)

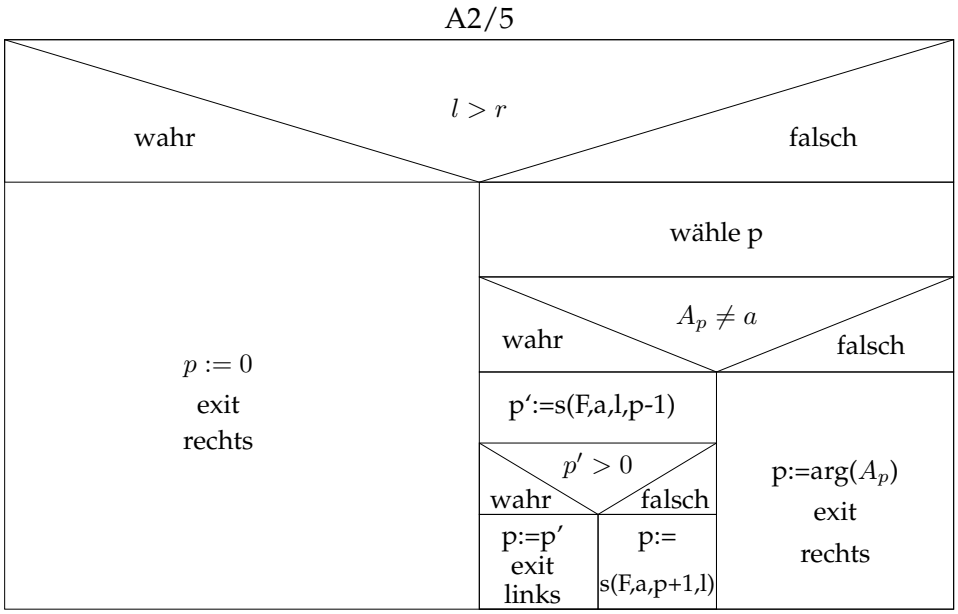
Vorbedingung: $(A_i \neq A_j$ für $i \neq j) \wedge (1 \leq l \ \wedge \ r \leq n)$

Verfahren:

```
def s(F, a, l, r) ≡
if   l > r then /* leere Folge */
     p := 0
else
     Wähle eine Position p zwischen l und r.
     if   A_p ≠ a then     /* noch nicht gefunden */
          p' := s(F, a, l, p − 1)     /* Suche in linker Teilfolge */
          if   p' > 0 then
               p := p'     /*in linker Teilfolge gefunden */
          else
               p := s(F, a, p + 1, r)     /*Suche in rechter Teilfolge */
          endif
     endif
endif
```

Ausgabe: p

Nachbedingung: $(a = A_p \wedge l \leq p \leq r) \vee (\forall j \in \{l, \ldots, r\})(a \neq A_j \wedge p = 0)$



Wie das Nassi-Shneiderman-Diagramm verdeutlicht, handelt es sich um eine nichtiterative Rekursion. Der rekursive Aufruf des linken Teilintervalls wird gefolgt von weiteren Anweisungen. Darunter befindet sich der rekursive Aufruf des rechten Teilintervalls. Diese Asymmetrie beider Rekursionen hat folgende Bedeutung.

Das Verfahren stellt sicher, daß nach dem Durchsuchen der linken Teilfolge auch die rechte Teilfolge noch durchsucht werden kann. Dabei wird die rechte Teilfolge nicht durchsucht, wenn die Suche in der linken Teilfolge erfolgreich war. Durch die Verwendung der Variablen p' ist sichergestellt, daß die ursprüngliche Suchposition p nicht verloren geht.

Drei spezielle Verfeinerungen des Befehlssatzes "Wähle eine Position p zwischen l und r" sind denkbar:

a) sequentielle rekursive Suche
"Wähle stets $p := r$." Dies hat zur Folge, daß die rechte Teilfolge immer leer ist. Dies führt zu einer Vereinfachung des Pseudocodes, ohne die Komplexität zu reduzieren, da stets eine Liste die Länge 1 und die andere die Länge $(r - l)$ hat.

b) binäre rekursive Suche
"Wähle $p :=$ div $((l + r), 2)$." Hierbei liegt die Suchposition in der Mitte des

Intervalls $[l, r]$. Dieses Verfahren hat eine asymptotisch bessere Komplexität als die sequentielle Suche. Ihr Vorteil kann aber erst im Fall c genutzt werden.

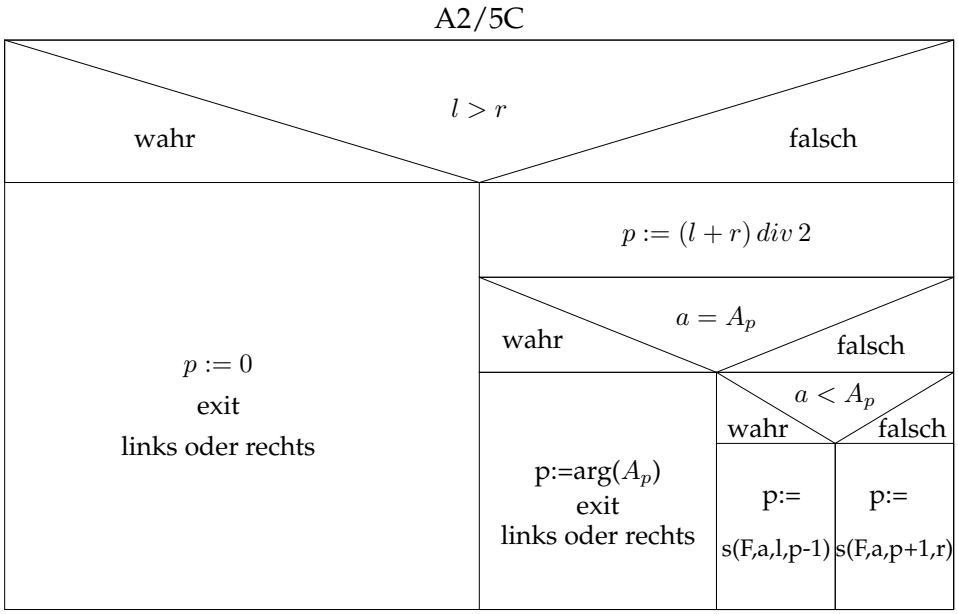
c) binäre rekursive Suche in aufsteigend geordneter Folge
Die Suche in einer geordneten Folge ist einfacher als die in einer ungeordneten Folge. Der Aufwand des Ordnens ist vor der Suche zu leisten, wenn die Liste nicht a priori sortiert ist. Der Aufwand zum Sortieren einer Liste der Länge n beträgt

- beim Sortieren durch Auswählen: $O(n^2)$
- beim Sortieren durch binäres Zerlegen in gleich große Teillisten: $O(n \log n)$.

Sortieren durch Auswählen bedeutet Auswählen des Minimums der aktuellen Restliste, Löschen dieses Elements aus der Liste, so daß diese weiter verkürzt wird und Anhängen des neuen Minimums an die sortierte Liste, welche anfangs leer ist. Tatsächlich kann das Sortieren mit der Suche dadurch verbunden werden, daß das Sortieren abbricht, wenn das gesuchte Element gefunden ist.

Beispiel: A2/5C: binäre rekursive Suche in geordneter Folge

```
def s(F, a, l, r) ≡
    if l > r then
        p := 0
    else
        p := (l + r) div 2 /* Suchposition in der Mitte */
        if a = A_p then
            exit
        else
            if a < A_p then
                p := s(F, a, l, p − 1) /* weitere Suche links */
            else
                p := s(F, a, p + 1, r) /* weitere Suche rechts */
            endif
        endif
    endif
```



Die symmetrische Struktur der rekursiven Aufrufe der Teilfolgen-Suche verändert die Logik für die Strategie des Austritts.

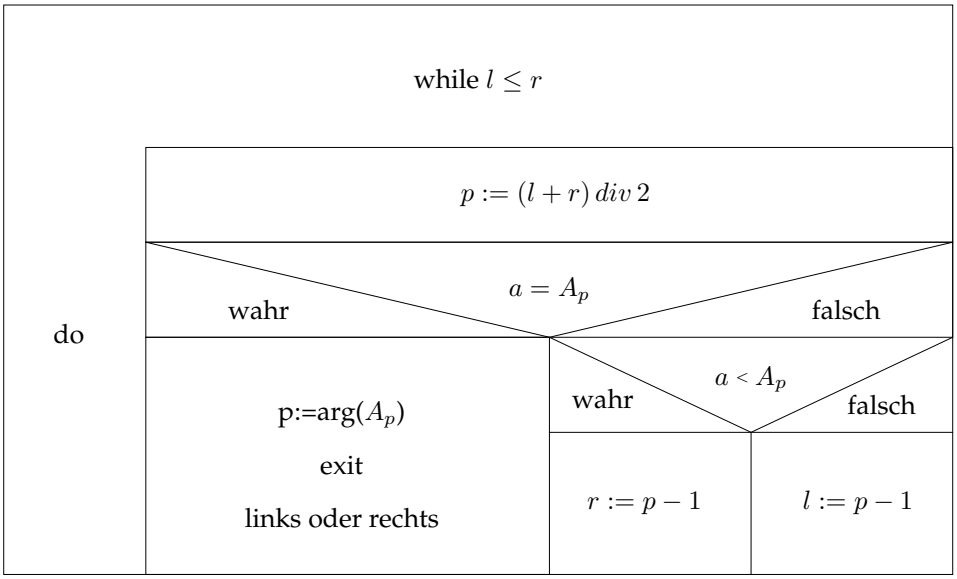
Das obige Beispiel ist typisch für eine iterative Rekursion, da die rekursiven Aufrufe von s am Ende des Algorithmus auftreten. Eine nichtrekursive Umformulierung mit einer while-Anweisung ist möglich:

Beispiel: A2/6C:

```
def s(F, a, l, r) ≡
    while l ≤ r do
        p := (l + r) div 2
        if a = A_p then
            exit
        else
            if a < A_p then
                r := p − 1
            else
                l := p + 1
            endif
        endif
    endwhile
```

Hierbei ist die Logik der Verarbeitung gegenüber A2/5C weiter modifiziert, siehe Nassi-Shneidermann-Diagramm.

A2/6C



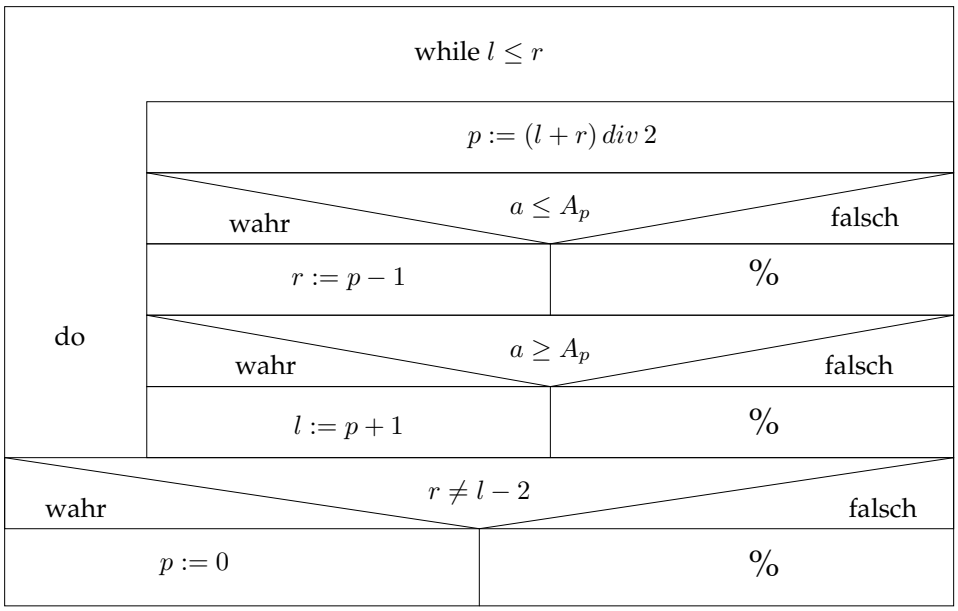
In Beispiel A2/6C ergibt sich das Problem, daß beim Austritt aus s die Austrittsursache nicht mehr erkennbar ist. Entweder wurde das Suchelement gefunden $(a = A_p)$ oder die Suchbedingung verletzt $(l \geq r)$.

Abhilfe schafft Beispiel A2/7C. Die Schleife wird auf genau eine Weise verlassen und es bleibt erkennbar, ob das gesuchte Element a gefunden wurde, denn dann ist $r = l-2$. Im Falle $a = A_p$ werden beide Zweige durchlaufen, was zu der festen Einstellung der Beziehung zwischen r und l führt.

Beispiel: A2/7C

```
def s(F, a, l, r) ≡
    while l ≤ r do
        p := (l + r) div 2
        if a ≤ A_p then
            r := p − 1
        endif
        if a ≥ A_p then
            l := p + 1
        endif
    endwhile
    if r ≠ l − 2 then
        p := 0
    endif
```

A2/7C



Die sequentielle Suche hat die asymptotische Komplexität $A^{\text{seq}}_{\text{max}} = O(n)$, d. h. die Anzahl der benötigten Rechenschritte ist proportional zur Folgenlänge n.
Die binäre Suche in einer geordneten Folge hat dagegen die asymptotische Komplexität

$A_{\max}^{\text{bin,geordn.}} = O(\log_2 n)$, d. h. die Anzahl der benötigten Rechenschritte ist proportional zum Logarithmus zur Basis 2 von n.

Der Vorteil der binären Suche gegenüber der sequentiellen Suche ist erst bei großen Argumenten offenbar. Bei einer Folgenlänge von 10^6 ist die Anzahl der benötigten Schritte bei der binärer Suche um den Faktor $50 \cdot 10^3$ geringer als bei der sequentiellen Suche.

Bei dieser Komplexitätsbetrachtung wurde der worst case betrachtet, also der Fall, daß $a \notin F$. Für die sequentielle Suche sind im statistischen Mittel nur $\frac{n}{2}$ Vergleiche nötig, was aber nichts daran ändert, daß die mittlere Komplexität proportional zu n bleibt.

Was passiert bei der Ausführung der Rekursion tatsächlich

Bei der Aktivierung einer weiteren Inkarnation der Rekursion (und auch bei der Ausführung einer anderen durch einen Algorithmus definierten Funktion innerhalb eines übergeordneten Algorithmus) wird die augenblickliche Rechnung zugunsten einer Zwischenrechnung unterbrochen. Dazu müssen die Parameter, Variablen und die Position des Wiedereintritts in der aktuellen Rechnung für jede aktuelle Inkarnation gemerkt werden. Am Ende müssen alle Inkarnationen vollständig beendet sein, d.h. sie müssen ihr jeweiliges Ende erreicht haben, die Parameter und Variablen ermittelt und evtl. an die jeweils übergeordnete Inkarnation weitergegeben werden.

Das Sichern und Restaurieren der Parameter, Variablen und Wiedereintrittsposition wird mit einer *Stapel/Stack*-Verwaltung realisiert.

Dazu kann man sich einen Stack als einen Stapel von Formularen vorstellen, auf dem neue Formulare aufgelegt und Formulare von oben wieder weggenommen werden können.

Ein Stack ist also eine Datenstruktur, auf der die Operationen

- push: oben auflegen
- pop: von oben entfernen

definiert sind.

Bei jeder Aktivierung einer Inkarnation werden der Instruktionszeiger und die Parameter auf den Stack "gepusht". Bei jeder Beendigung einer Inkarnation werden die gemerkten Informationen vom Stack "gepopt", d.h. die ursprünglichen Parameter werden wiederhergestellt und es wird an der Stelle der übergeordneten Inkarnation fortgefahren, an der die Unterbrechung zum Aufruf der neuen Inkarnation erfolgte.

Rekursion und Induktion

Die Rekursion hat ihre mathematische Wurzel in der *Induktion*. Eine rekursive Definition einer Funktion beinhaltet folgende Sachverhalte:

1. Es existieren eine oder mehrere *Basisregeln*, die einfache Objekte/Sachverhalte definieren.

2. Es existieren eine oder mehrere *Induktionsregeln*, mit deren Hilfe größere Objekte in Ausdrücken von kleineren Objekten definiert werden.

Beispiel: Fakultät

Der nicht-rekursiven Definition der Fakultätsfunktion

$$n! = \prod_{i=1}^{n} i$$

entspricht die rekursive Definition der Fakultätsfunktion $n! = n \cdot (n-1)!$

Basis: $1! = 1$

Induktion: $n! = n \cdot (n-1)!$

Beweis: Beweise, daß $p(n) = n! = n \cdot (n-1)!$ in der rekursiven Formulierung dem Produkt $p(n) = 1 \cdot 2 \cdot \ldots \cdot n$ entspricht.

Basis: $p(1)$ gilt. Jedes k-stellige Produkt von Einsen ergibt eins.

Induktion:

Annahme: $p(n)$ gilt, d.h. das Produkt $p(n) = 1 \cdot 2 \cdot \ldots \cdot n \equiv n!$ kann auch rekursiv bestimmt werden.

Dann gelte auch

$$(n+1)! = n! \cdot (n+1)$$

Substitution von $n!$

$$(n+1)! = 1 \cdot 2 \cdot \ldots \cdot n \cdot (n+1) \equiv p(n+1)$$

Damit ist die Induktionshypothese bewiesen.

Die Berechnung einer rekursiven Funktion erfolgt durch Zurückführen auf die Induktionsbasis. Die Induktionsbasis hat sicher eine "primitive" Lösung.

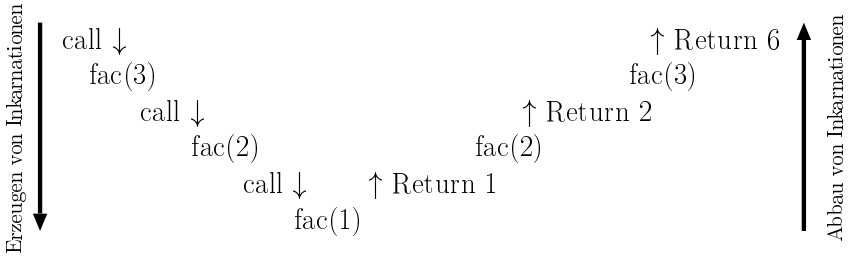
Eine rekursive Funktion eines Arguments (einer Menge von Argumenten) wird solange in einfachere Funktionen der gleichen Klasse abgebildet, bis diese primitiv erfüllbar sind (das ist die Induktionsbasis). Von dieser Basis aus erfolgt in den vorgezeichneten Schritten der Vereinfachung der Argumente die Rekonstruktion der ursprünglichen Funktion im Sinne der Berechnung ihrer Werte.

Beispiel: Fakultät

fct fac : int → int

```
def fac(n) ≡
    if n ≤ 1 then      /* Basis */
        p := 1
    else
        p := n · fac(n − 1)      /* Induktion */
    endif
```

Beispiel: fac(3)



Rekursion bedarf eines organisatorischen Aufwandes. Deshalb darf nicht geschlossen werden, daß die konzeptionelle Eleganz der Rekursion zwingend mit einem Gewinn an Effizienz einhergeht. Dies hat folgende Ursachen:

1. nicht jede rekursive Formulierung einer Funktion läßt gegenüber der nicht-rekursiven Formulierung eine geringere asymptotische Komplexität erwarten.

   Die asymptotische Komplexität beschreibt das Skalierungsverhalten der Komplexität der Funktion bei Skalierung ihrer Argumente. Die asymptotische Komplexität ist also unmittelbar mit der induktiven Interpretation der Rekursion verbunden.

2. Das *Laufzeitverhalten* von Funktionsaufrufen (Calls) bei solchen Implementierungen hängt von vielen konkreten technischen Gegebenheiten ab:

   - Qualität des Computers
   - Eigenschaften des Betriebssystems
   - Wirkungsweise des Prozessors

   Das Laufzeitverhalten von Funktionen wird in Kapitel 4 besprochen.

Beispiel: asymptotische Komplexität

1. Fakultät: lineare Komplexität $O(n)$ für rekursive und nicht-rekursive Formulierung

2. Potenzfunktion: ganzzahlige, nicht-negative Potenz einer reellen Zahl

   nicht-rekursiv:

$$x^n = \prod_{i=1}^{n} x$$

   erfordert $n - 1$ Multiplikationen, also gilt $O(n)$

   rekursiv:

$$x^n = \begin{cases} 1 & n = 0 \\ (x^{n/2})^2 & n > 0, \quad n \text{ gerade} \\ x \cdot (x^{(n-1)/2})^2 & n > 0, \quad n \text{ ungerade} \end{cases}$$

   erfordert $\log_2 n$ Multiplikationen, also gilt $O(\log_2 n)$

Rekursion und "Teile und Herrsche"

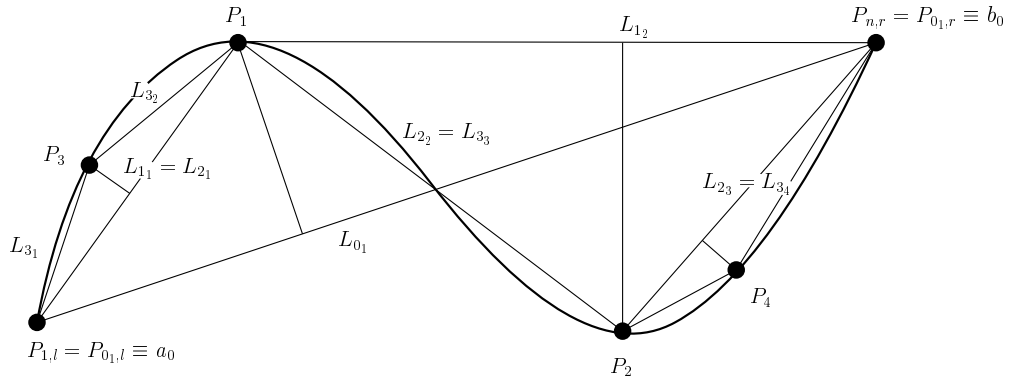
Die Rekursion folgt dem erkenntnistheoretisch bedeutsamen Prinzip *"Teile und Herrsche"* (siehe Abschnitt 2.1). Das Problem wird solange in Unterprobleme zerlegt, bis deren Lösung angebbar ist. Hieraus wird durch Kombination der Teillösungen die Gesamtlösung des Problems rekonstruiert. Dies erfordert:

1. Unterprobleme müssen einfacher gestaltet sein als das ursprüngliche Problem.

2. Nach endlich vielen Schritten muß ein Unterproblem erreichbar sein, dessen Lösung einfach angebbar ist. Das heißt, das Verfahren muß terminieren.

Die *Methode der schrittweisen Verfeinerung* hat ihre Wurzel ebenfalls im Teile-und Herrsche-Prinzip.

Beispiel: stückweise Geradenapproximation einer Kurve



Informale Beschreibung als Iteration (nicht rekursiv):

Gegeben:

- Kurve $C(a_0, b_0)$ mit den Endpunkten a_0, b_0.
- Approximationsschwelle δ (maximal zulässige Distanz zwischen der Kurve und zu findenden Sehnen L.

Gesucht: Sehnenfolge $\mathcal{L} = L_1 \circ L_2 \circ \ldots \circ L_n$ mit $d_i \leq \delta$, $i = 1, \ldots, n$, d_i maximale Distanz der Sehne L_i, die in den Punkten $P_{i,l}$ und $P_{i,r}$ die Kurve C schneidet.

Verfahren: $j := 0$ /* Start Iteration */

while

{ Es gibt in der Liste der Sehnen $\mathcal{L}_j = \{L_{j_i} | i = 1, \ldots, j+1\}$ eine Sehne L_{j_k}, $k_j = \arg\max_i\{d_{j_i}\}$ mit $d_{j_k} > \delta$ }

do

1. Ermittle den korrespondierenden Punkt $P_{(j+1)_k}$ auf der Kurve C, der zwischen $P_{j_k,l}$ und $P_{j_k,r}$ liegt.
2. Konstruiere aus der Sehne L_{j_k} die Sehnen $L_{(j+1)_k}$ und $L_{(j+1)_{k+1}}$, wobei gilt
$$P_{(j+1)_k,l} := P_{j_k,l}$$
$$P_{(j+1)_k,r} := P_{j_k}$$
$$P_{(j+1)_{k+1},l} := P_{j_k}$$
$$P_{(j+1)_{k+1},r} := P_{j_k,r}$$
3. Erzeuge aus Liste $\mathcal{L}_j$ die Liste $\mathcal{L}_{j+1}$, indem die Sehne L_{j_k} durch die Sehnen $L_{(j+1)_k}$ und $L_{(j+1)_{k+1}}$ ersetzt wird.

endwhile

erforderlich: Funktion Dist zur Berechnung von d (evtl. noch weitere Funktionen); Datenstruktur Liste

Bemerkungen: Das Verfahren terminiert, weil die Folge

$$d_{(j-1)_k} > d_{j_{k'}} > d_{(j+1)_{k''}}$$

für irgend welche k, k' und k'' streng monoton ist.

Der Index k_j tritt an irgend einer Stelle in der Folge der Indizes $\{1, \ldots, j+1\}$ auf. Das Beispiel hat also Ähnlichkeit mit Beispiel A2/5 (Suche in nicht geordneter Folge). Es hat auch Ähnlichkeit mit Beispiel 1 in Kapitel 2, wenn man sich vorstellt, daß das Auffinden von n geeigneten Sehnen dem Erraten von n Zeichen (n unbekannt) eines Alphabets entspricht.

Rekursionsformen:

Es werden mehrere verschiedene Arten der Rekursion unterschieden:

1. Lineare Rekursion

Definition: (lineare Rekursion)

Tritt im Rumpf einer rekursiven Funktionsdefinition ein Aufruf der Funktion in jedem Zweig einer Fallunterscheidung höchstens einmal auf, so heißt die Rechenvorschrift linear rekursiv.

Die Interpretation einer rekursiven Funktionsdefinition bedeutet, rekursiv vereinbarten Funktionsidentifikatoren in eindeutiger Weise eine Abbildung zuzuweisen. Hierfür gibt es zwei Zugänge:

- induktive Deutung
- Deutung durch Fixpunktgleichungen

Bei beiden Verfahren wird eindeutig festgelegt, mit welcher Abbildung ein Identifikator durch eine rekursive Funktionsdefinition belegt wird.

Einfacher ist die operationale Deutung rekursiver Funktionsdefinitionen. Hierzu nutzt man z.B. die Methode der Entfaltung.

Beispiel: A4/1: ganzzahlige Division

Erinnerung: Für jedes $n \in \mathbb{N}$ existiert eine Zerlegung

$$n = m \cdot y + r,$$

$m, r \in \mathbb{N},\ m > 0,\ m > r \geq 0$, wobei

$$\begin{aligned} y &= n \text{ div } m = \left\lfloor \frac{n}{m} \right\rfloor := \max\left\{ z \in \mathbb{N} \middle| z \leq \frac{n}{m} \right\} = \text{ div } (n, m) \\ r &= n \bmod m = n - \left\lfloor \frac{n}{m} \right\rfloor m = \bmod(n, m) \end{aligned}$$

Folglich gilt die Identität

$$n = \text{ div } (n, m) \cdot m + \bmod(n, m)$$

Fallunterscheidung:

1. $n < m \Rightarrow \text{ div } (n, m) = 0,\ r = \bmod(n, m) = n$
2. $n \geq m \Rightarrow \text{ div } (n, m) = \text{ div } (n - m, m) + 1$
   (rekursiver Ansatz!)
   Es gilt: $\bmod(n, m) = \bmod(n - m, m)$

spc div$(n, m) \equiv y$:
  pre $m > 0$
  post $n = m \cdot y + r \quad \wedge \quad m > r \geq 0$

def div$(n, m) \equiv$
  if $n < m$ then 0 else div $(n - m, m) + 1$ endif

Definition: (Methode der Entfaltung)

Die Methode der Entfaltung (*Expansion*) des Aufrufes einer rekursiven Funktion besteht im Einsetzen des Rumpfes einer Funktionsdefinition an die Stelle des Identifikators im Aufruf.

Durch wiederholte Anwendung der Regel der Entfaltung kann unter Umständen ein rekursiver Aufruf auf einen Wert reduziert werden.

Beispiel: A4/2: Modulo-Funktion

$\underline{\text{def}}\ \text{mod}(n, m) \equiv$
$\quad \underline{\text{if}}\ n < m\ \underline{\text{then}}\ n\ \text{else}\ \text{mod}(n - m, m)\ \underline{\text{endif}}$

Gegeben: Funktionsaufruf $\text{mod}(11, 4)$, gesucht: Wert der Funktion

$\text{mod}(11, 4) \Leftrightarrow$ (<u>if</u> $n < m$ <u>then</u> n <u>else</u> $\text{mod}(n - m, m)$ <u>endif</u>)(11,4)
$\Leftrightarrow$ <u>if</u> $11 < 4$ <u>then</u> 11 <u>else</u> $mod(11 - 4, 4)$ <u>endif</u>
$\Leftrightarrow$ mod(7,4)
$\Leftrightarrow$ (<u>if</u> $n < m$ <u>then</u> n <u>else</u> $\text{mod}(n - m, m)$ <u>endif</u>)(7,4)
$\Leftrightarrow$ <u>if</u> $7 < 4$ <u>then</u> 7 <u>else</u> $\text{mod}(7 - 4, 4)$ <u>endif</u>
$\Leftrightarrow \text{mod}(3, 4)$
$\Leftrightarrow$ (<u>if</u> $n < m$ <u>then</u> n <u>else</u> $\text{mod}(n - m, m)$ <u>endif</u>)(3,4)
$\Leftrightarrow$ <u>if</u> $3 < 4$ <u>then</u> 3 <u>else</u> $\text{mod}(3 - 4, 4)$ <u>endif</u>
$\Leftrightarrow 3$

2. Repetitive/Iterative Rekursion

Die Definition wurde bereits auf Seite 160 gegeben. Die iterative Rekursion ist ein Spezialfall der linearen Rekursion. Beispiele sind die sequentielle Suche und die Funktionen div und mod.

Die Eigenschaft, daß ein rekursiver Aufruf als letzte Aktion in der Funktionsdefinition erfolgt, führt dazu, daß die Auswertung der aktivierten Inkarnation vollständig abgeschlossen wird, bevor die Auswertung einer neuen Inkarnation beginnt.

Das heißt, es müssen die Identifikatoren aus lokalen Deklarationen (Unterfunktionen) und die aktuellen Parameter nicht aufbewahrt werden, wenn wieder zur unterbrochenen Inkarnation zurückgekehrt wird.

3. Vernestete Rekursion

Definition: (vernestete Rekursion)

Treten im Rumpf einer rekursiven Funktionsdefinition in den aktuellen Parameterausdrücken eines rekursiven Aufrufs weitere rekursive Aufrufe auf, so heißt die Rekursion vernestet oder verschachtelt.

Vernestete Rekursionen führen zu baumartigen, nichtlinearen Aufrufstrukturen, weshalb sich Beweise der Eigenschaften vernestet rekursiver Algorithmen als schwierig erweisen.

Beispiel: A4/3: Modulofunktion

Ziel ist die Umwandlung der Identität

$$n = \text{ div } (n, m) \cdot m + \text{mod}(n, m)$$

in eine effizienter zu nutzende Identität.

Grundlagen:

1. $\text{ div } (n, m) = \left\lfloor \frac{n}{m} \right\rfloor \sim 2 \cdot \text{ div } (n, 2m) = 2 \left\lfloor \frac{n}{2m} \right\rfloor =: d$
2. $\text{mod}(n, m) = \text{mod}(\text{mod}(n, 2m), m)$

Daraus folgt die neue Identität

$$n = (2 \cdot \text{ div } (n, 2m) + \alpha) \cdot m + \text{mod}(\text{mod}(n, 2m), m)$$

mit $\alpha = \begin{cases} 1 & \text{für } n - m \geq d \cdot m \\ 0 & \text{sonst} \end{cases}$

Hieraus leitet sich die folgende Prozedur zur Realisierung der Identität ab:

a) $n < m : d = 0$
$n = (0 + 0) \cdot m + n$

b) $m \leq n < 2m : d = 0$
$n = (0 + 1) \cdot m + (n - m)$,
denn $\text{mod}(\underbrace{\text{mod}(n, 2m)}_{=n, \text{ da } n<2m}, m) = \text{mod}(n, m) = \text{mod}(n - m, m) = n - m$

c) $n \geq 2m : \ d \geq 2$
   i) $n - m < d \cdot m$:
   $n = d \cdot m + \text{mod}(\text{mod}(n, 2m), m)$
   ii) $n - m \geq d \cdot m$:
   $n = (d + 1) \cdot m + \text{mod}(\text{mod}(n, 2m), m)$

Damit läßt sich die Funktion mod formulieren:

```
def mod(n, m) ≡
  if n < m then n                                  /* (a) */
    else if (m ≤ n  ∧  n < 2m) then n − m          /* (b) */
    else mod(mod(n, 2m), m)                        /* (c) */
  endif
```

Beispiel eines Aufrufs:

$$
\begin{aligned}
\mathrm{mod}(137,19) &\Leftrightarrow \mathrm{mod}(\mathrm{mod}(137,38),19) && \text{wegen (c)}\\
&\Leftrightarrow \mathrm{mod}(\mathrm{mod}(\mathrm{mod}(137,76),38),19) && \text{wegen (c)}\\
&\Leftrightarrow \mathrm{mod}(\mathrm{mod}(\mathrm{mod}(61,76),38),19) && \text{wegen (b)}\\
&\Leftrightarrow \mathrm{mod}(\mathrm{mod}(61,38),19) && \text{wegen (a)}\\
&\Leftrightarrow \mathrm{mod}(23,19) && \text{wegen (b)}\\
&\Leftrightarrow \mathrm{mod}(4,19) && \text{wegen (b)}\\
&\Leftrightarrow 4 && \text{wegen (a)}
\end{aligned}
$$

Ein Argument eines rekursiven Aufrufs der Modulo-Funktion ist selbst repräsentiert durch den Wert einer Modulo-Funktion.
Während die repetitive rekursive Modulofunktion von linearer Zeitkomplexität ist, ist die vernestet rekursive Modulofunktion von logarithmischer Komplexität:

$$A^{\mathrm{lin}}_{\mathrm{mod}} = O(n \text{ div } m) \qquad A^{\mathrm{nest}}_{\mathrm{mod}} = O(\log_2(n \text{ div } m))$$

Beispiel:

$n = 1000, m = 2$

linear rekursive Modulofunktion: 500 Inkarnationen
vernestet rekursive Modulofunktion: 10 Inkarnationen

4. Kaskadenartige Rekursion

Definition: (kaskadenartige Rekursion)

Treten im Rumpf einer rekursiven Funktionsdefinition in mindestens einem Zweig einer Fallunterscheidung zwei oder mehr rekursive Aufrufe auf, so spricht man von kaskadenartiger (baumartiger, nichtlinearer) Rekursion

3.4 Grundzüge der zustandsorientierten Programmierung

Im wesentlichen werden drei Klassen von Programmiersprachen bzw. Paradigmen der Programmierung unterschieden:

1. Funktionale/deklarative Programmiersprachen: z.B. ML, Lisp

   Hierbei entspricht das Programmieren dem Formulieren von Funktionsdefinitionen. Die Programmausführung stellt die Berechnung eines Ausdrucks mit Hilfe dieser Funktionen und die Berechnung eines Ergebniswertes dar. Die deklarativen bzw. funktionalen Programmiersprachen werden auch als *applikative* Programmiersprachen bezeichnet.

2. Logische Programmiersprachen: z.B. PROLOG

   Bei den logischen Programmiersprachen besteht das Programmieren aus dem Formulieren logischer Prädikate. Die Programmausführung erfolgt durch das Stellen von Anfragen, ob die Anwendung der Prädikate auf (Eingabe-)Terme den Wahrheitswert wahr ergibt.

3. Imperative/zustandsorientierte Programmiersprachen: z.B. Pascal, Modula-2, C

   Bei den imperativen Programmiersprachen erfolgt das Programmieren durch das Definieren von Datenobjekten und von Aktionen auf diesen. Die Programm-Ausführung besteht in der Veränderung der den Objekten zugewiesenen Werte durch die Wirkung der Aktionen. Imperative Programmiersprachen werden auch als *prozedural* bezeichnet.

3.4.1 Das zustandsorientierte Programmier-Paradigma

Zustandsorientierte und funktionale Programmierung sind nichts grundsätzlich Verschiedenes sondern betonen nur unterschiedliche Aspekte der Informationsverarbeitung:

1. funktionale Programmierung: Die Objekte sind Argumente und Ergebnisse von Funktionen. D.h., die Funktionen bleiben über die Zeit erhalten, aber die Objekte (Argumente) ändern sich beliebig. Ein Programm ist also eine Abbildung zwischen Daten.

2. zustandsorientierte Programmierung: Die Objekte erhalten eine eigene Identität und Existenzberechtigung. Dabei modifizieren Operationen/Aktionen Objekte, ohne ihre Identität zu ändern. Damit beschreibt ein Programm die Zustandsübergänge eines Objektes.

Der Übergang zwischen diesen beiden Betrachtungsweisen ist fließend und hängt vom Problem ab!

Ein Beispiel für die funktionale Betrachtungsweise ist die Volumenberechnung von Zylindern/Pyramiden (Beispiel A1).

Datenbanken hingegen fallen unter die zustandsorientierte Betrachtungsweise. Denn es macht keinen Sinn, eine Datenbank nach der Modifikation ihrer Zustände als etwas anderes anzusehen. Die Identität der Datenbank bleibt erhalten.

Zustände sind Abstraktionen von Historie: Der Zustand eines Systems zum Zeitpunkt t hängt davon ab, welche Aktionen bis zu diesem Zeitpunkt stattfanden.

Gegeben seien zwei Zustände σ_1 und σ_2 eines Objekts, die durch die Aktion α ineinander überführt werden:

$$\sigma_1 \xrightarrow{\alpha} \sigma_2$$

Die Zustände σ_1 und σ_2 sind Elemente eines Zustandsraumes (Zustandsmenge) Σ.

Die Ausführung einer Anweisung einer imperativen Programmiersprache bewirkt funktional eine Abbildung des Zustandsraumes auf sich:

$$I :< \text{statement} > \Rightarrow (\Sigma \rightarrow \Sigma)$$

Damit wird die Semantik einer Anweisung als Zustandsabbildung interpretiert.

Das imperative Programmier-Paradigma hat seine Entsprechung in dem *von-Neumann-Modell* einer Rechnerarchitektur:

- Maschine mit linear angeordnetem Speicher
- Programme sind wie Daten abgelegt
- Zustand der Maschine ist durch Inhalt des Speichers (+ Register) gegeben
- mit jedem Bearbeitungsschritt erfolgt die Änderung des Zustandes des Speichers

Damit bestimmt der Speicherzustand die Folge der Berechnungsschritte und die Berechnungsschritte beeinflussen wiederum den Speicherzustand.

Durch die Unterscheidung zwischen Daten und Programmen erfolgt eine Aufteilung des Zustandsraumes:

$$\begin{aligned} \Sigma &= \Sigma_{\text{Daten}} \cup \Sigma_{\text{Kontrolle(Programm)}} \\ \Sigma_{\text{Kontrolle}} &= \Sigma_{\text{Operanden}} \cup \Sigma_{\text{Operationen}} \end{aligned}$$

Durch die Einführung geeigneter Daten- und Kontrollstrukturen wird der Zustandsraum imperativer Programme strukturiert (s. Abschnit 3.4.2).

Hierfür gibt es in imperativen Programmiersprachen Standardstrukturen, von denen im Falle der Programmiersprache “C” allerdings zum Teil erheblich abgewichen wird.

Alles, was über die Spezifikation und Definition von Funktionen in den Abschnitten 3.2 und 3.3 eingeführt wurde, gilt auch im imperativen Programmier-Paradigma. Insbesondere gelten die Gesetze der Aussagenlogik zur Beschreibung der Abbildung von Zuständen auf Zustände. Diese müssen aber auf die Gesetze der Prädikatenlogik erweitert werden.

Definition: (Zusicherung)

Ein Prädikat P, das die Eigenschaften eines Zustandes σ beschreibt, heißt Zusicherung (assertion) über dem Zustand σ. Die Zusicherung P beschreibt die Menge

$$\Sigma(P) = \{\sigma | P(\sigma)\}$$

aller Zustände σ, für die P gilt.

Zusicherungen sind *prädikatenlogische Formeln*. Die Ableitung

$$P \vdash Q$$

beschreibt, daß ein Zustand σ, für den die Zusicherung P gilt, auch die Zusicherung Q erfüllt.

Zusicherungskalkül nach Hoare (1969)

Seien S eine Anweisung und P, Q beliebige Boolesche Ausdrücke, in denen unter anderem die Programmvariablen aus der Anweisung S frei auftreten dürfen, dann steht die Schreibweise

$$\{P\}\ S\ \{Q\}$$

für die Aussage:
Für jeden Eingangszustand, für den vor Ausführung von S die Zusicherung P gilt, gilt bei Terminierung der Ausführung von S für den Nachfolgezustand die Zusicherung Q oder $\forall \sigma_1, \sigma_2 \in \Sigma$:
$I_{\sigma_1}[P] = \mathrm{TRUE} \wedge I[S](\sigma_1) = \sigma_2 \ \wedge \ \sigma_2 \neq \perp \Rightarrow I_{\sigma_2}[Q] = \mathrm{TRUE}$

Durch den Hoare-Kalkül ist nur die *partielle Korrektheit* beschrieben, terminiert die Ausführung von S nicht, dann ist die Aussage trivialerweise richtig.
P und Q stehen für die Vor- und Nachbedingungen der Anweisung S, durch (P, Q) ist die Spezifikation der Anweisung S gegeben. Das Konzept der Zusicherung ist auf jede Aktion eines Programmes oder auf das gesamte Programm übertragbar, um so die

partielle Korrektheit einer Aktion bezüglich der Spezifikation zu überprüfen. Dieser Beweis der Korrektheit wird *Verifikation* genannt.

Die obige Interpretation des Zusicherungskalküls entspricht einer *Vorwärtsanalyse*. Dabei besteht das Problem, daß eine konkrete Vorbedingung in Verbindung mit der Aktion zu vielen gültigen Nachbedingungen führen kann.
Praxisorientierter ist die *Rückwärtsanalyse*: Welche Vorbedingungen erfüllen für eine gegebene Nachbedingung und eine gegebene Aktion die gewünschte Nachbedingung?

Die Menge aller möglichen Vorbedingungen ist halbgeordnet:

- P ist schwächer als P', wenn gilt $P' \Rightarrow P$
- $\bigvee_{j=1}^{n} P_j$ ist schwächer als jedes $P_j,\ j = 1, \ldots, n$, also $P_j \Rightarrow \bigvee_{j=1}^{n} P_j$.

Beispiel:

P_1="Auto hat Radio mit defektem Sendersuchlauf"
P_2="Auto hat Radio mit defekter Lautstärkeregelung"
P_3="Auto hat defektes Radio"
P_4="Auto hat Radio"
Es folgt: $P_3 \Rightarrow P_4$ $\quad P_1 \Rightarrow P_1 \vee P_2$
$P_1 \Rightarrow P_3$ $\quad P_2 \Rightarrow P_1 \vee P_2$
$P_2 \Rightarrow P_3$
Q="Auto hat gutes Radio"
Gesucht ist die Aktion α (z.B. Reparatur des Radios), die mit der Vorbedingung P_i die Nachbedingung Q erfüllt:

$$\{P_i\}\alpha\{Q\}$$

P_4 ist die schwächste Vorbedingung für die Nachbedingung (es muß überhaupt ein Radio vorhanden sein, damit dies ein gutes Radio sein kann).

Regel für die Abschwächung von Zusicherungen

Gegeben seien die Prädikate P, P', Q, Q' mit

$$P' \Rightarrow P,\ Q \Rightarrow Q',$$

dann gilt

$$\{P\}\alpha\{Q\} \vdash \{P'\}\alpha\{Q'\}.$$

Wird die Vorbedingung P durch eine stärkere Vorbedingung P' ersetzt, ist mit P' auch P erfüllt. Wird die Nachbedingung Q durch eine schwächere Nachbedingung Q' ersetzt, so ist dies ebenfalls im Sinne der Schlußfolgerung zulässig.

Beispiel:

Sei α Zuweisung $i := 1$. Dann gilt:

$$\{P : i \text{ beliebig}\}\ i := 1\ \{Q : i = 1\}\ \Rightarrow\ \{P' : i < 0\}\ i := 1\ \{Q' : i > 0\}$$

Die schwächste Vorbedingung beschreibt die Menge all derjeniger Zustände, von denen aus durch α die Nachbedingung Q erreicht werden kann.

Definition: (schwächste Vorbedingung)

Seien α eine Aktion und Q eine Nachbedingung. Dann heißt P schwächste Vorbedingung (für α und Q), wenn gilt

1. $\{P\}\alpha\{Q\}$
2. $\forall P'$ mit $\{P'\}\alpha\{Q\}$ gilt: $P' \Rightarrow P$.

Kalkül der schwächsten Vorbedingung (E.W. Dijkstra, 1976)

Die Korrektheit eines Programms kann durch Rückwärtsanalyse dadurch nachgewiesen werden, daß die ursprüngliche Spezifikation der Eingabedaten die schwächste Vorbedingung des Gesamtprogramms erfüllt.

Dabei ist die Vorbedingung P_i der Anweisung mit der Nummer i zugleich Nachbedingung Q_{i-1} der Anweisung mit der Nummer $i - 1$.

Eine häufige Ursache inkorrekter Programme besteht darin, daß die Zusicherung der Vorbedingung nicht die schwächste Zusicherung darstellt (z.b. die Vereinigung aller möglicher Varianten der Eingabedaten).

Beispiel: Anwendung der Regel der schwächsten Vorbedingung

Seien α die Zuweisung $i := i + 1$ und $\{Q : i > 0\}$ Nachbedingung.
Offensichtlich ist $\{P : i \geq 0\}$ die schwächste Vorbedingung, die diese Nachbedingung erfüllt. Denn:

Seien $i_{\text{vor}} = I$ und $i_{\text{nach}} = i_{\text{vor}} + 1$, dann gilt $i_{\text{nach}} = I + 1$. Hierfür muß die Nachbedingung erfüllt sein, also $\{Q : I + 1 > 0\}$
Da aber $I = i_{\text{vor}}$, folgt

$$\{P : \ i + 1 > 0\}\ i := i + 1\ \{Q : \ i > 0\}.$$

Wegen

$$\{i + 1 > 0\} \Leftrightarrow \{i > -1\} \Leftrightarrow \{i \geq 0\}$$

folgt

$$\{P : \ i \geq 0\}\ i := i + 1\ \{Q : \ i > 0\}$$

Hieraus folgt das Zuweisungsaxiom:

Zuweisungsaxiom

Seien x eine Variable und t ein beliebiger, aber wohl definierter Ausdruck einer Zuweisung $x := t$ und Q eine Zusicherung, dann gilt

$$\{Q[t/x]\}\ x := t\ \{Q\}$$

Das Zuweisungsaxiom führt die Zuweisung (assignment) auf die Substitution zurück. Entsprechende Axiome lassen sich für alle Anweisungen einer imperativen Programmiersprache formulieren.

3.4.2 Einige Strukturierungskonzepte der Programmiersprache "C"

Der Zustand eines Programmes ist keine unzerlegbare Einheit, sondern setzt sich aus vielen Einzelkomponenten zusammen.
Zur Strukturierung des Zustandsraums tragen bei:

- die Operanden : Konstanten, Variable und aggregierte Datenstrukturen
  Sie ändern entweder selbst ihren Zustand (Wert) oder dienen dazu, eine solche Zustandsänderung zu berechnen.

- die Operationen: Anweisungen, Funktionen, Prozeduren
  Sie beschreiben die Art und Weise, wie Zustände eines Zustandsraums zu erreichen sind.

- *Lebensdauer* und *Gültigkeitsbereich* der *Bindungen* von Identifikatoren an Operanden und Operationen: Jeder in einer Anweisung oder in einem Ausdruck auftretende Identifikator ist entweder frei (ungebunden) oder es existiert genau eine gültige Deklaration oder eine gültige Angabe in einer Parameterliste, durch die der Identifikator gebunden ist.

  Achtung: Ein häufiger Programmierfehler besteht in der Nichtbeachtung der Lebensdauer von Bindungen (Mehrfachdefinition von Variablen, Nichtbeachten des begrenzten Gültigkeitsbereiches).

  Die *strukturierte Programmierung* ist ein Entwurfsprinzip, das dazu dient, die Bindungsstruktur so einfach und durchschaubar wie möglich zu gestalten.

  Seiteneffekte oder *Nebeneffekte* sind Eigenschaften von Operationen (Ausdrücken, Anweisungen, Prozeduren, Funktionen), die nicht beabsichtigte Zustandsänderungen bewirken können.

3.4.2.1 Strukturierungskonzept für die Operanden des Zustandsraums

a) Variable

Eine *Variable* ist die elementare Einheit zur Beschreibung eines Zustandes eines Programms. Alle in einem Programmstück verfügbaren Variablen bilden die Komponenten des globalen Zustandes.

Eine Variable steht für das Tripel (Referenz, Speicher, Wert)

Referenz: (unveränderliches) eindeutiges Kennzeichen der Variablen, sie steht für einen Verweis auf die Variable und die Identität einer Variablen.

Eine Referenz ist eine begriffliche Abstraktion, die über den Begriff einer Adresse hinausgeht. Dies ist erforderlich, weil Variable sowohl als Kopien mehrfach im Programmablauf existieren können oder weil Variable an veränderlichen Orten gespeichert sein können.

Speicher: Platz im Speicher des Rechners.

Wert: Inhalt des Speichers der Variablen.
Wenn dieser Wert unveränderlich ist, dann ist die Variable eine Konstante.

Zugriffsfunktion: Der Zugriff auf eine Variable erfolgt über eine Zugriffsfunktion. Deren Aufruf berechnet die Referenz einer Variablen und wird im Programmtext als Name (der Variablen/Konstanten) angegeben.

Identifikator: einfachste Form für Namen, auch: Bezeichner

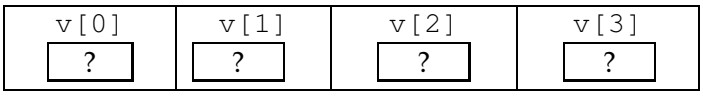
Für Variable ist die Angabe ihrer Typen/Sorten erforderlich, um die Festlegung der Größe des Speicherplatzes und Adressierungsrechnungen (z.B. im Feldern, Strukturen) zu ermöglichen.

b) Felder

Ein *Feld* ist die systematische Aneinanderreihung (z.B. als Vektor, Matrix) von gleichartigen Objekten, so daß diese für das Programm eine Einheit bilden. Dabei besitzen alle Komponenten eines Feldes den gleichen Typ. Ein bequemer Zugriff auf alle Komponenten ist möglich.

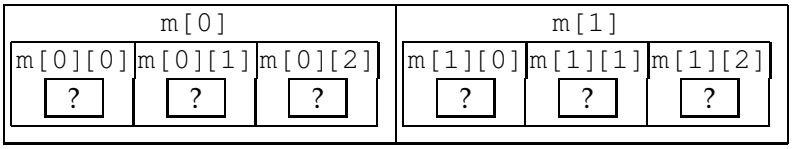
Beispiel:

```
float v[4];
```



Speicheranordnung Vektor mit 4 Komponenten

```
int m[2][3];
```



Speicheranordnung (2×3)-Matrix

Der Zugriff auf eine bestimmte Komponente eines Feldes erfordert eine Adreßrechnung der folgenden Art:

$$A_{\text{Komp}} = A_{\text{Feld}} + \text{Offset}.$$

Dabei weist der Name des Feldes auf das Element mit dem Index Null und Offset bezeichnet den relativen Abstand der Komponente zum Anfang des Feldes. Solange A_{Feld} nur symbolisch durch den Feldnamen gegeben ist, kann auf eine Komponente zugegriffen werden, ohne die absolute Basisadresse des Feldes zu kennen.
In "C" sind bei Feldern zwei Arten der Adressierung möglich:

1. Adressierung über Indizes
2. Adressierung über Zeiger (Pointer)

Ein *Zeiger* (*Pointer*) ist selbst eine Variable eines bestimmten Typs, die eine Adresse aufnehmen kann. Damit repräsentiert ein Zeiger eindeutig einen Speicherplatz (jedoch in einer höheren Programmiersprache nur symbolisch).
Für Zeiger sind zwei Operatoren von Bedeutung:

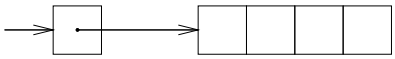
Adreßoperator: &
: Der Adreßoperator dient der *Referenzierung* eines Objekts, d.h. er liefert als Wert eines Ausdruckes, angewendet auf ein Objekt im Speicher (Variable, Felder) dessen Adresse als Wert eines Zeigers (Pointers).

Inhaltsoperator: *
: Der Inhaltsoperator dient der *Dereferenzierung* eines Objekts, d.h. er liefert bei Anwendung auf einen Zeiger den Inhalt eines Objektes, auf den der Zeiger gerichtet ist.

Beispiel: P1

Notationen: mit $< \cdot >$ wird der Wert einer Variablen, mit $\{\cdot\}$ die Adresse einer Variablen angegeben.

`float v[4];`	
`float *p, n;`	p ist Zeigervariable vom Typ float
`p=&v[0];`	p erhält als Wert die Adresse von v[0]
`p=v;`	äquivalent zu oben, da der Name des Feldes v auf erstes Element (mit Index Null) verweist



Ausgehend von dieser Situation betrachten wir die Wirkung einiger Zeigeroperationen. Dabei soll jede der folgenden Anweisung für sich stehen und sich auf die oben skizzierte Situation beziehen. Die aufgeführten Anweisungen sind also nicht im Sinne einer Folge von Anweisungen zu interpretieren.

`n=*p;`	`n` erhält als Wert den Inhalt von `v[0]`
	`<n>=<v[0]>`
`n=*(p+1);`	`<n>=<v[1]>`

Die Inkrementierung erfolgt um eine Float-Adresse!

Beispiele für zeigerbasierte Zugriffe mittels Inkrementoperator ++:

`n=*(++p);`	<n>=<v[1]>. Aber Nebeneffekt: <p>={v[1]}
`n=*++p;`	äquivalent zu vorheriger Anweisung.
`n=*p++;`	<n>=<v[0]>. Nebeneffekt: <p>={v[1]}
`n=(*p)++;`	<n>=<v[0]>. Nebeneffekt: <v[0]> wird inkrementiert
`n=++*p;`	p bleibt unverändert, aber <v[0]>wird inkrementiert.
`n=++(*p);`	äquivalent zur vorherigen Anweisung

In "C" besteht folgende Äquivalenz von Ausdrücken:

```
v[i]  ⇔  *(v+i)
&v[i] ⇔  v+i
```

Der C-Compiler behandelt Stringkonstanten "..." wie Felder ohne Namen, deren letztes Element den Zeichencode für binär Null darstellt. Pointer müssen demzufolge den Typ `char *` besitzen.

Beispiel: P2

`char z[]="HALLO!";`	z ist String-Variable, d.h. die String-Konstante "HALLO" wird durch den Compiler in diese kopiert (als Anfangswert), darf aber im Programmablauf geändert werden.
`char *p;`	
`p=z;`	<p>={z[0]}
`p++;`	<p>={z[1]}
`*p='B';`	Ersetzung: <z[1]>= 'B';

Nicht erlaubt ist aber folgendes:

`char *p;`	
`p="HALLO";`	p ist Zeiger auf Stringkonstante (string literal). Diese wird vom Compiler als Konstante compiliert!
`*p='B';`	Deshalb ist diese Ersetzung nicht erlaubt!

---

c) Strukturen

Eine Struktur ist eine pragmatische Aneinanderreihung von ungleichartigen Objekten (unterschiedlichen Typs).

Beispiel:

Deklaration zweier Strukturen `p1, p2` vom Strukturtyp `KK` (mit zwei Komponenten vom Typ Integer)

```
struct KK
{
  int m,n;
} p1,p2;
```

Der Zugriff auf Komponenten von Strukturen erfolgt mit den folgenden Operatoren:
Der Punktoperator `.` dient der Dereferenzierung der Komponenten einer Struktur.

Beispiel:

Der Struktur `p1` wird der Wert (15,10) für das Tupel (m, n) zugewiesen:

```
struct KK p1;
p1.m=15; p1.n=10;
```

In der Praxis werden große Strukturen als Parameter von Funktionen via Zeiger übergeben (um das Kopieren der Struktur bei der Parameterübergabe zu vermeiden). Beim Zugriff auf die Komponenten einer als Zeiger übergebenen Struktur sind dann zwei Dereferenzierungen nötig: zum einen die Dereferenzierung des Zeigers, zum anderen die Dereferenzierung der Komponente.

Beispiel:

```
struct KK *p3;
```

`p3` ist ein Zeiger auf eine Struktur vom Typ `struct KK`. Dann ist `*p3` die Struktur selbst und `(*p3).m` bzw. `(*p3).n` sind die Komponenten der Struktur.

Diese doppelte Dereferenzierung wird mit dem Pfeiloperator `->` vermieden, er stellt einen äquivalenten Zugriff auf die Komponenten der Struktur durch `p3->m` zur Verfügung.

3.4.2.2 Strukturierungskonzepte für Operationen des Zustandsraumes

Die durch ein Programm induzierten Zustandsübergänge entsprechen Wertänderungen von Variablen. Die Anzahl der Variablen bestimmt die Dimension des Zustands-

raumes. Sein Volumen hängt vom Typ der Variablen ab, also davon, welche Zustände jede Variable annehmen kann.

Typisch für die Architektur der von-Neumann-Maschine ist die *Sequentialisierung* der möglichen bzw. erforderlichen Zustandsübergänge. Demzufolge muß die Änderung der Werte unabhängiger Variabler auch sequentiell erfolgen. Dies drückt sich darin aus, daß ein Programm aus einer Folge von Anweisungen besteht.

Anweisungen (*statements*) erzeugen Zustandsübergänge. Sie beschreiben die Operationen, die ein Programm bzw. Unterprogramm (oder Funktion) ausführen kann. Daraus folgt die Bezeichnung "imperative" oder "prozedurale" Programmiersprache.

Obwohl der Sprachumfang einer imperativen Programmiersprache weitgehend standardisiert ist, gibt es sowohl in der Syntax als auch in der Anwendung (Semantik) der Sprachelemente in den verschiedenen Sprachen deutliche Unterschiede.

Die Programmiersprache "C" kennt folgende Anweisungstypen, die hier in Backus-Nauer-Form (BNF) angegeben werden:

```
<statement>::= <expression-statement>|     Ausdrucksanweisungen
               <compound-statement>|       zusammenges. Anweisungen
               <selection-statement>|      Auswahlanweisungen
               <iteration-statement>|      Schleifenanweisungen
               <labeled-statement>|        markierte Anweisungen
               <jump-statement>.           Sprunganweisungen
```

Auf der Rechten Seite des "::="-Zeichens sind die verschiedenen Arten der Ausdrücke (durch | als Alternativen gekennzeichnet) angegeben. Ein `statement` ist also entweder ein `expression-statement` oder ein `compound-statement` usw.

a) Zuweisung

Als *Zuweisung* (assignment) wird die elementare Anweisung zur Änderung des Zustandes einer Variablen bezeichnet. Sie besetzt eine Variable `x` mit dem Wert eines Ausdruckes (expression) `t`.

Allgemein gilt:
`<assignment-statement>::=<identifier > = <expression>`

Beispiel:

`x=x+1` sei eine Zuweisung
$\sigma_1.x$ sei der Zustand der Komponente x vor der Ausführung der Zuweisung
$\sigma_2.x$ sei der Zustand der Komponente x nach der Ausführung der Zuweisung
$n = \sigma_1.x$ sei die Belegung des Zustandes vor der Ausführung der Zuweisung

Dann bewirkt die obige Zuweisung daß
$\sigma_2.x = \sigma_1.x + 1$, d.h. $\sigma_2.x = n + 1$
und
$\sigma_2.y = \sigma_1.y$ für Identifikatoren $y \neq x$.

In "C" ist die *Ausdrucksanweisung* die elementare Form
`<expression-statement>::=<expression>`$_{\text{opt}}$`;`
Jeder Ausdruck wird durch das Semikolon zur Anweisung. Damit ist das Semikolon ein Abschlußsymbol und kein Trennsymbol. Eine Form der Anweisung
`<expression>;` macht nur bei Ausdrücken Sinn, die Nebeneffekte erzeugen.

Beispiel: Äquivalente Ausdrücke der Inkrementierung:

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

Sind gleichwertige Ausdrücke (aber nicht als Operanden von Ausdrücken oder als Argumente von Funktionsaufrufen).

Der Zuweisungsoperator "=" ist ein normaler binärer Operator. (Als Vergleichsoperator wird "==" verwendet.)

Das bewirkt, daß der Ausdruck `x=y;` selbst einen Wert hat, der als Operand eines weiteren Ausdrucks verwendet werden kann, z.B.
`z=x=y`.
Zuweisungsoperatoren werden von rechts nach links ausgewertet, sie besitzen eine sehr niedrige Priorität.

Es existieren eine Reihe kombinierter Zuweisungsoperatoren:

Beispiel:

```
+=  :   x+=10;    ⇔ x=x+10;
*=  :   x*=a+10;  ⇔ x=x*(a+10);
```

b) Zusammengesetzte Anweisungen

Viele imperative Programmiersprachen sind blockstrukturiert. Blöcke bilden semantische Einheiten. Oft wird die Gültigkeit von Variablen auf einen Block beschränkt. Dann müssen die Variablen auch dort deklariert werden. In "C" werden Blöcke durch zusammengesetze Anweisungen realisiert:

<compound-statement>$_{\text{opt}}$::=
{
<declaration-list>$_{\text{opt}}$
<statement-list>$_{\text{opt}}$
}

Eine zusammengesetzte Anweisung gilt als einzelne Anweisung und darf an deren Stelle überall stehen.

c) Funktionen

Imperative Programmiersprachen kennen

- *Prozeduren* (Unterprogramme) zur Zusammenfassung logisch geschlossener Programmabschnitte, die häufig genutzt werden und
- *Funktionen* zur häufig wiederholten Berechnung der Werte von Variablen bei unterschiedlichen Parameterbelegungen (Argumenten).

Prozeduren und Funktionen werden durch Aufrufe aktiviert. Dies setzt voraus, daß sie deklariert und definiert sind.
Prozeduren und Funktionen werden auch als Black Box eingesetzt, von denen man beim Aufruf nur wissen muß

- welche Parameter sie erfordern und
- welche Werte sie liefern.

Im Gegensatz zu Funktionen geben Prozeduren keine Werte zurück.

In "C" kennt man nur Funktionen. Ob diese Werte zurückliefern oder nicht, ist Angelegenheit des Programmierers. Funktionen des Typs `void` entsprechen Prozeduren anderer imperativer Programmiersprachen.

Man unterscheidet zwischen *Funktionsdeklarationen* und *Funktionsdefinitionen*:

<function-declaration>::=
  <result-type>$_{\text{opt}}$<function-name>(<parameter-list>$_{\text{opt}}$);
<function-definition>::=
  <result-type>$_{\text{opt}}$<function-name>(<parameter-list>$_{\text{opt}}$)<block>

Bei einer Funktionsdeklaration werden nur Typ und Parameter einer Funktion festgelegt, also dem Compiler mitgeteilt. Die Festlegung des Funktionsrumpfes erfolgt durch die Funktionsdefinition. Die Funktionsdeklaration ist nicht zwingend. Sie ist aber z.B. dann nötig, wenn Deklaration und Definition in unterschiedlichen Quelltextdateien (*Module*) erfolgen. So können Funktionen, die in einem Modul definiert sind, durch erneute Deklaration auch anderen Modulen zugänglich gemacht werden.

Ein *Funktionsaufruf* hat folgende Gestalt:

<function-call>::=<function-name>(<argument-list>$_{\text{opt}}$)(;)

Ein Funktionsaufruf ist ein Ausdruck, d.h. er kann sowohl als Teil einer Anweisung (ohne Semikolon) als auch als Ausdrucksanweisung (mit Semikolon) für sich stehen.

Bei imperativen Programmiersprachen werden folgende Arten der *Parameterübergabe* an Funktionen unterschieden:

Call-by-Value: Es werden Parameterwerte an das Unterprogramm/die Funktion übergeben. Die Funktion hat keinen direkten Zugriff auf die Variablenwerte des aufrufenden Programmes. Lediglich Kopien werden übergeben, so daß sich Änderungen an diesen Kopien nicht außerhalb der Funktion auswirken. Call-by-Value ist der Standard in "C".

Call-by-Reference: Es werden Zeiger auf Variablen (Referenzen) des aufrufenden Programms als Parameter übergeben. Damit ist ein ändernder Zugriff auf die von außen als Argumente übergebenen Variablen aus einer Funktion heraus möglich. In "C" ist diese Übergabeart durch explizite Deklaration der formalen Parameter als Zeiger realisiert.

Call-by-Name: Es werden Adressen der Parameter (Namen) an das Unterprogramm/die Funktion übergeben. Die Werte der Parameter sind zwar veränderbar, aber nicht ihre Adresssen. Die Parameterausdrücke werden dann erst ausgewertet, wenn sie in der Funktion/Prozedur benötigt werden. Dies kann bei wiederholtem Aufruf sehr unwirtschaftlich sein. Einmaliges Berechnen der Adreßbezüge des

Argumentausdruckes beim erstmaligen Aufruf und Zwischenspeichern für spätere Aufrufe ist effektiver (Call-by-Need). Die Übergabeart Call-by-Name ist in "C" nicht möglich.

Zusätzlich zu den oben beschriebenen Möglichkeiten der Funktionsdeklaration sind in "C" auch Funktionsdeklarationen als *Prototyp* möglich. Dabei ist die Deklaration in der Parameterliste auf die Angabe der Typen der Parameter beschränkt. Auf die Angabe der Namen wird verzichtet. Die Deklaration von Prototypen ermöglicht dem Compiler, die Korrektheit von Funktionsaufrufen zu überprüfen und ggf. eine implizite Umwandlung des Typs eines Argumentes in den Typ des formalen Parameters der Definition vorzunehmen.

Im Gegensatz zu den meisten imperativen Programmiersprachen erlaubt "C" keine Verschachtelung von Funktionen, also die Deklaration oder Definition einer Funktion innerhalb einer übergeordneten Funktion. Die Ursache ist die im Vergleich zu anderen imperativen Programmiersprachen schwach ausgeprägte Blockstruktur von "C". Jedoch vereinfacht sich die Laufzeitumgebung für den Aufruf von Funktionen. Die Begriffe der Laufzeitumgebung und des Aktivierungsrekords werden im Kapitel 4 eingeführt.

In "C" sind aber sowohl *direkte* (Funktion ruft sich selbst auf) als auch *indirekte* Rekursion (Funktion 1 ruft eine andere Funktion 2 auf, die wiederum Funktion 1 aufruft) möglich.

3.4.2.3 Nebeneffekte in der Programmiersprache "C"

In der Eigenschaft *Nebeneffekte* oder *Seiteneffekte* zuzulassen unterscheiden sich die Programmier-Paradigmen (funktional, logisch, imperativ) am deutlichsten:

- funktionale und logische Programmiersprachen sind prinzipiell frei von Nebeneffekten (jedoch müssen diese für die Ein/Ausgabe zugelassen werden).
- in imperativen Programmiersprachen gehören Nebeneffekte zu den inhärenten Eigenschaften.

Definition: (Nebeneffekt)

Verändert die Bestimmung eines Operanden `a` eines Ausdrucks den Wert eines anderen Operanden `b`, dann hat die Berechnung von `a` einen Nebeneffekt auf die Berechnung von `b`.

Imperative Programmiersprachen leben von Nebeneffekten, der Programmierer (unterstützt vom Compiler) hat dafür zu sorgen, daß Nebeneffekte nur unter kontrollierten Bedingungen eintreten und dadurch unschädlich sind.

Beispiel:

```
x=y;
```

Der Wert der Variable `y` wird als Nebeneffekt der Variablen `x` zugewiesen.

Jeder Ausdruck wird formal zu einer Anweisung (Ausdrucksanweisung), indem ihm ein Semikolon nachgestellt wird. Dies setzt aber voraus, daß die Ausdrücke Nebeneffekte erzeugen. Dies leisten Ausdrücke mit Zuweisungs-, Inkrement-, Dekrementoperatoren oder Funktionsaufrufen mit Nebeneffekten.

Beispiel: äquivalente Ausdrücke für die Inkrementierung

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

sind im Fall von Ausdrucksanweisungen äquivalent.

Bei Verwendung in anderen Anweisungen gilt dies nicht:

a) `x++` z.B. `a[x]=x++;`
Erst nach der Berechnung des gesamten Ausdrucks hat die Variable `x` einen inkrementierten Wert (Postfix-Operator).

b) `++x` z.B. `a[x]=++x;`
Der bereits inkrementierte Wert wird als Operand verwendet (Präfix-Operator). Die Variable `x` hat aber auch hier erst nach der Berechnung des gesamten Ausdruckes den neuen Wert zugewiesen.

c) `x+=1;` und `x=x+1;` entsprechen der Präfixschreibweise

Der Standard der Sprache "C" definiert *Synchronisationspunkte* für das Eintreten von Nebeneffekten:

- beim Erreichen dieser Punkte müssen alle bisherigen Nebeneffekte eingetreten sein und
- es dürfen noch keine Nebeneffekte des nachfolgenden Codes eingetreten sein.

Beispiel:

Beim Aufruf einer Funktion ist der Synchronisationspunkt nach der Berechnung der Argumentwerte des Aufrufs gegeben.

```
a= f( )+ g( );
```

Beide Funktionen müssen ausgewertet werden, bevor die Addition ausgeführt wird. Die Reihenfolge des Aufrufs von `f` und `g` ist nicht eindeutig (wegen der Kommutativität der Addition).
Probleme treten auf, wenn beide Funktionen Nebeneffekte bzgl. gleicher globaler Variablen haben. Dann wäre ein möglicher Ausweg:

```
a=f( );
a+=g( );
```

Hat `g` Nebeneffekte für `a`, so liefert auch diese Lösung nicht das gewünschten Ergebnis.

Zur Vermeidung von Problemen mit Nebeneffekten sollten die Ratschläge der Programmier-Handbücher sowie folgende allgemeine Ratschläge beachtet werden:

1. Das Hauptprogramm (main) darf allen globalen Variablen sowie seinen lokalen Variablen Werte zuweisen.

2. Eine Funktion darf nur ihren lokalen Variablen Werte zuweisen und Funktionswerte zurückliefern.

3. Eine zu inkrementierende/dekrementierende Variable sollte innerhalb eines Ausdrucks nur einmal vorkommen. (nicht: `b=a++ +a;`)

4. Kommt eine Variable, der ein Wert zugewiesen wird, auch als Operand des Ausdrucks vor, muß notwendigerweise die Auswertung des Ausdrucks (einschließlich der Variablen) vor der Wertzuweisung abgeschlossen sein.

3.4.2.4 Gültigkeitsbereiche und Lebensdauer von Bindungen

Identifikatoren werden als Platzhalter in Programm verwendet für folgende Entitäten:

- Datenelemente
- Programmvariable
- Funktionen
- Prozeduren.

Da ein Programm leicht eine gut überschaubare Größe überschreitet und da Programme im allgemeinen ein stark strukturiertes und partitioniertes Gebilde darstellen, ist die Zuordnung Name–Entität nicht trivial sicherzustellen.

Der Programmierer muß stets die semantische Identität zwischen der zu repräsentierenden Entität und dem repräsentierenden Namen überblicken.

Der "C"-Compiler muß stets die syntaktische Identität bzgl. Typgleichheit von Deklaration und Definition zwischen der zu repräsentierenden Entität und dem repräsentierenden Namen herstellen können.

Man unterscheidet die Begriffe Lebensdauer oder Gültigkeitsbereich einer Bindung, jenachdem ob die deklarierte Entität (Lebensdauer) oder der zur Deklaration verwendete Name (Gültigkeitsbereich) primär betrachtet werden.

Definition: (Bindung)

Jeder in einer Anweisung oder in einem Ausdruck auftretende Identifikator ist entweder frei (ungebunden) oder es existiert genau eine gültige Deklaration oder eine gültige Angabe in einer Parameterliste, durch die der Identifikator gebunden ist.

Definition: (Lebensdauer einer Bindung)

Die Lebensdauer einer vereinbarten Bindung eines Identifikators an eine Entität ist derjenige Ausschnitt eines Programmablaufes, in dem die vereinbarte Angabe der Entität existiert. Die Lebensdauer ist ein Begriff, der sich auf die Phase der Programmausführung bezieht.

Innerhalb der Lebensdauer einer Bindung ist der Zustandsraum durch die Existenz der benannten Entität bestimmt. Mit Beendigung der Lebensdauer existiert diese Entität nicht mehr, d.h. der Speicherplatz wird nicht mehr repräsentiert (vom Compiler) und die Dimension des Zustandsraumes ändert sich.

Der Compiler kennt nur Namen, die in dem gerade zu übersetzenden Modul deklariert sind, beginnend von der Stelle ihrer Deklaration an.
Der "C"-Compiler startet, nachdem der *Präprozessor* alle Textersetzungen von Makronamen abgeschlossen hat. Das heißt, was über die Grenzen eines Moduls deklariert sein soll, muß dem Präprozessor angezeigt sein.

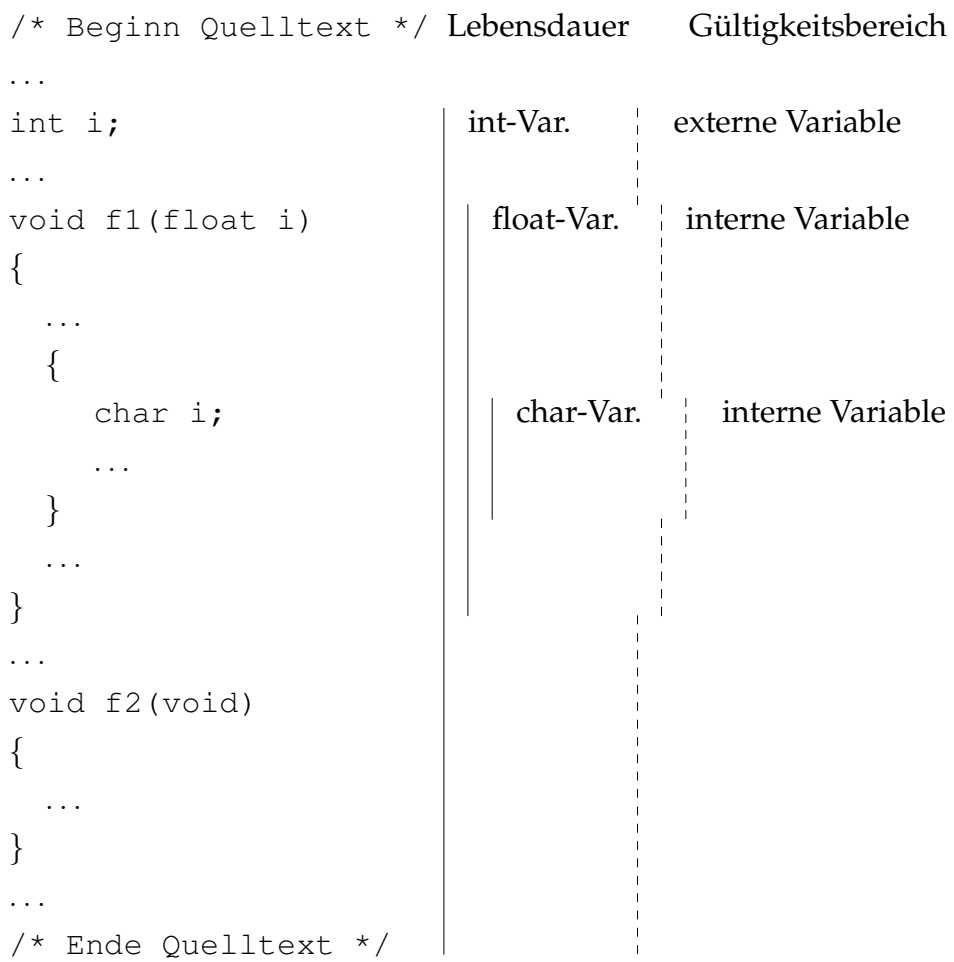
Außerhalb von Funktionen deklarierte Namen existieren von der Stelle ihrer Deklaration bis zum Ende des Moduls.
Deklarationen innerhalb von zusammengesetzen Anweisungen (Blöcke) sind auch nur dort gültig.
Namen formaler Parameter von Funktionen sind nur innerhalb von Funktionen gültig.
Namen formaler Parameter von Prototypen sind nur innerhalb der Deklaration gültig.

Definition: (Gültigkeitsbereich einer Bindung)

Der Gültigkeitsbereich einer Bindung entspricht der Lebensdauer, außer in Abschnitten eines Programm-Textes, in dem dieselben Identifikatoren neue Bindungen eingehen.

Das entspricht dem in "C" bekannten Prinzip der *Verschattung*:
Ist der formale Parameter einer Funktion mit dem gleichen Namen belegt wie eine außerhalb der Funktion deklarierte Entität, so kann innerhalb der Funktion nur auf den gleichnamigen formalen Parameter zugegriffen werden. Die Gültigkeit des Namens ist auf die Funktion beschränkt.
In gleicher Weise verdecken Namen, die in einer zusammengesetzten Anweisung (also auch innerhalb von Funktionen) vereinbart werden, eventuell gleichnamige Entitäten außerhalb dieses Blockes.

Beispiel:



Die hier zur Verdeutlichung unterschiedlich gewählten Typen spielen keine Rolle. Das Verschattungs-Prinzip ist auch dann gültig, wenn sämtliche Definitionen der Variablen `i` den selben Typ hätten.

In "C" wird zwischen verschiedenen Arten von Größen unterschieden:

- interne Größen: sind nur innerhalb von Blöcken vereinbart
- externe Größen: sind außerhalb von Blöcken vereinbart
- lokale Größen: stehen innerhalb des Moduls zur Verfügung, in dem sie vereinbart werden

- globale Größen: stehen potentiell in allen Modulen zur Verfügung (nur Variable und Funktionen)

Diese Eigenschaften können durch Attribute der Deklaration (Speicherklassen) beeinflußt werden:

`static`: Externe Größen (Variable und Funktionen) werden zu lokalen Größen

`extern`: Auftrennung von Deklaration und Definition einer Variablen. Die Definition einer Variablen (Zuweisung von Speicherplatz) kann an anderer Stelle (in einem anderen Modul) als deren Deklaration (Vereinbarung des Typs) erfolgen.

In "C" ist es üblich, eine Zusammenfassung der Deklarationen aller globalen Größen eines Moduls in einer dem Modul zugeordneten *Header-Datei* vorzunehmen. Diese Deklarationen werden durch eine Präprozessoranweisung durch den Compiler eingelesen.

Beispiel:

```
#include <stdio.h>
```
Anweisung für den Präprozessor, diese Anweisung durch die Datei `stdio.h` zu ersetzen.

Externe Variable stehen während der gesamten Dauer der Ausführung eines Programmes zur Verfügung. Deshalb wird ihnen bereits durch den Compiler Speicherplatz zugewiesen.
Interne Variable stehen nur innerhalb der Blöcke zur Verfügung, in denen sie deklariert wurden. Demzufolge erfolgt die Speicherzuordnung dynamisch während der Laufzeit des Programmes, die Speicherplatzreservierung und Freigabe erfolgt gekoppelt mit dem Ein- und Austreten aus Blöcken.

Literaturverzeichnis

[1] F.L. Bauer, G. Goos. *Informatik. Band1: Eine einführende Übersicht*. Springer, 1991.

[2] F.L. Bauer, G. Goos. *Informatik. Band2: Eine einführende Übersicht*. Springer, 1992.

[3] W. Bauer et al. *Studien- u. Forschungsführer Informatik*. Springer, 1989.

[4] R. Berghammer. *Informatik I und II. Vorlesungsskript 1995/96*. Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1996.

[5] M. Broy. *Informatik I: Problemnahe Programmierung*. Springer, 1992.

[6] M. Broy. *Informatik II: Rechnerstrukturen und maschinennahe Programmierung*. Springer, 1993.

[7] M. Broy. *Informatik III: Systemstrukturen und systemnahe Programmierung*. Springer, 1994.

[8] W. Coy. *Informatik-Spektrum 12*, 1989.

[9] W. Coy. *Aufbau und Arbeitsweise von Rechenanlagen*. Vieweg, 1992.

[10] P.J. Denning et al. *IEEE Computer Magazine*, Febr. 1989.

[11] G. Goos. *Vorlesungen über Informatik. Band I: Grundlagen und funktionales Programmieren*. Springer, 1995.

[12] G. Goos. *Vorlesungen über Informatik. Band II: Objektorientiertes Programmieren und Algorithmen*. Springer, 1996.

[13] B.W. Kernighan, D.M. Ritchie. *Programmieren in C*. C. Hanser, 1990.

[14] H. Klaeren. *Vom Problem zum Programm*. Teubner, Stuttgart, 1991.

[15] W. Kluge. *Informatik II. Vorlesungsskript 1987*. Inst. f. Informatik. u. Prakt. Math., CAU Kiel, 1987.

[16] W. Kluge, C. Aßmann. *Informatik für Ingenieure II. Vorlesungsskript 1996*. Inst. f. Informatik u. Prakt. Math., CAU Kiel, 1996.

[17] B.O. Küppers. *Der Ursprung biologischer Information*. Piper, 1990.

[18] H. Liebig, T. Fink. *Rechnerorganisation*. Springer, 1993.

[19] H. Neumann, H.S. Stiehl. *Einführung in die Informatik für Mathematiker und Naturwissenschaftler. Vorlesungsskript 1992*. Fachbereich Informatik, Universität Hamburg, 1992.

[20] W. Oberschelp, G. Vossen. *Rechneraufbau und Rechnerstrukturen*. Oldenburg, 1994.

[21] R. P. Paul. *SPARC Architecture, Assembly Language Programming, & C*. Prentice Hall, 1994.

[22] P. Pepper. *Grundlagen der Informatik*. Oldenburg, 1995.

[23] *Spektrum der Wissenschaft*, Juli 1996.

[24] C.L. Tomdo, S.E. Gimpel. *Das C-Lösungsbuch*. C. Hanser, 1990. (zu Kernigham, Ritchie: Programmieren in C).

[25] C.F. v. Weizsäcker. *Die Einheit der Natur*. 1971.

[26] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, 1992.

Für die Vorlesung eignen sich besonders die Referenzen [1], [5], [6], [6], [7], [8], [13], [20],[22], [24].

Index

Unterstrichene Seitenzahlen verweisen auf die Definitionen der zugehörigen Begriffe.