Grundlagen der Informatik

Klaus Knopper

Stand: 16.12.2005

Organisatorisches

Vorlesungs-Inhalt, Zeitplan, Klausur.

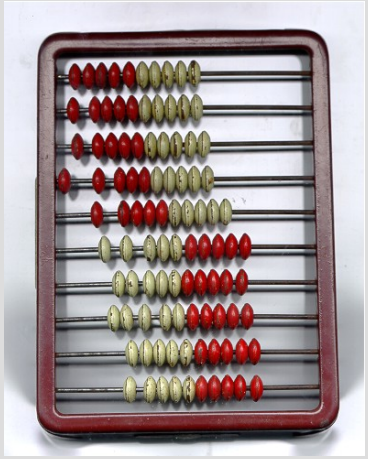
http://knopper.net/bw/gdi/

Definition

- Informatik ist ähnlich der Mathematik eine Strukturwissenschaft.
- Das Wort Informatik ist zusammengesetzt aus **Infor**mation und Auto**matik**.
- Die Informatik umfasst sowohl die Erforschung als auch die Lehre des mit Computern Machbaren.
- Als eigene Wissenschaft recht neu, die Wurzeln reichen lang zurück.

Geschichte der Informatik (1)

- Der Abakus ist ein einfaches mechanisches Rechenhilfsmittel.
- Verwendung von ca. 300 v.Chr bis heute.
- Neben den Grundrechenarten ist u.a. auch Wurzelziehen möglich.



Geschichte der Informatik (2)

- Um 1642 entwickelt Blaise Pascal einen Zweispeziesrechner.
- Addition und Subtraktion mittels Zahnrädern realisiert.
- Die Maschine fand weite Verbreitung und soll für Unruhe durch Arbeitsstellenverlust geführt haben.

Geschichte der Informatik (3)

- Gottfried Wilhelm Leibniz entwickelt um 1671 eine Rechenmaschine.
- Die Maschine kann mit allen vier Grundrechenarten rechnen.
- Erste Ideen zur Nutzung des Dualsystems für Rechenmaschinen.

Geschichte der Informatik (4)

- Im Jahr 1728 wird zum ersten Mal auf Holzkarten Informationen gespeichert.
- George Boole entwickelte in der ersten Hälfte des 19. Jahrhunderts die nach ihm benannte Boolesche Algebra. Sie bildet praktisch die mathematische Grundlage für jede digitale Rechen- und Steuerschaltung.

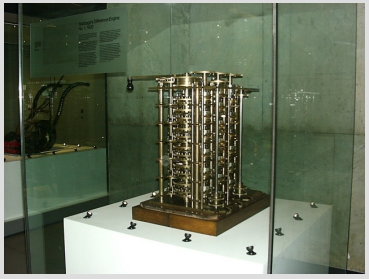
Geschichte der Informatik (5)

- Im Jahr 1822 entwirft Charles Babbage den ersten digitalen programmgesteuerten Rechenautomaten.
- Zu Lebzeiten stellte er seine Rechenautomaten nie fertig, denoch gilt er als Grundvater des modernen Computers.
- Ada Lovelace war der erste Mensch, der „Software“ schrieb. Sie war Mitarbeiterin von Charles Babbage.

Geschichte der Informatik (6)

Der Rechenautomat von Babbage hatte damals schon:

- Rechenwerk
- einen Zahlenspeicher
- ein separates Steuerwerk
- Ein- und Ausgabe-Einrichtungen (Lochkartenstrom, Drucker)



Geschichte der Informatik (7)

- Konrad Zuse entwickelte 1936 den ersten voll funktionstüchtigen Computer.
- Durch seine Spezifizierung der Programmiersprache *Plankalkül* entwarf er die erste universelle Programmiersprache der Welt.
- Zuse hat die Methode der computergerechten Fließkommazahlen auf Basis der Komponenten von Mantisse und Exponent theoretisch entwickelt und praktisch realisiert. (Vergl. IEEE 754)

Geschichte der Informatik (8)

Enigma, Turing und Bletchley Park.

- Militärische Anforderungen treiben die Entwicklung seit 1940 vorran.
- Turing war britischer Mathematiker und Kryptoanalytiker. Sein Modell der Turing-Maschine stellt die Grundlage der theoretischen Informatik dar.
- Mit Hilfe der Turing-Maschine wurde ein Modell zum Brechen des Enigma-Codes entwickelt.
- 1952 schrieb Turing ein Schachprogramm. Ohne einen Computer mit genügend Leistung es auszuführen, übernahm Turing dessen Funktion und berechnete jeden Zug selbst, was ihn um die 90 Minuten pro Zug kostete.

Zeittafel Programmiersprachen

1947 Plankalkül

1954 FORTRAN

1959 LISP

1960 BASIC

1971 Pascal

1972 C

1980 Smalltalk

1983 C++

1987 Perl

1995 Java

2001 C#

Gesellschaft und Informatik (1)

Informatik beeinflusst die Gesellschaft

- direkt durch Programme (Anwendungen),
- indirekt durch Programme (Videorekorder, Fernseher, etc),
- Vorgehensmodelle zur Entwicklung von Software,
- Abbilden von Geschäftsprozessen mit IT.

☞Produkte der Informatik beeinflussen die Gesellschaft in einem hohen Maß.

Gesellschaft und Informatik (2)

Das Fach Informatik untergliedert sich u.a. in:

- Die Theoretische Informatik beschäftigt sich mit Fragen wie Entscheidbarkeit und Komplexität.
- Die Praktische Informatik beschäftigt sich mit den Gebieten Softwaretechnik, Software- und Systemarchitektur und Programmiersprachen.
- Die Technische Informatik befasst sich mit Rechnerarchitektur, vernetzte Systeme und Schaltungen.
- Die Wirtschaftsinformatik befasst sich mit dem Einsatz der Informationstechnik in Unternehmen.

Gesellschaft und Informatik (3)

Einwurf: Proprietäre vs. Freie Software.

- Freie Software stellt Software als Resource zur Verfügung.
- Freie Software sichert dem Anwender (Benutzer und Programmierer) bestimmte Freiheiten.
- Freie Software stellt eine Basis (Lizenz) für eine Zusammenarbeit von Gruppen (oder Firmen) zur Verfügung.

Inhalt dieser Vorlesung

- Grundlagen vermitteln für aufbauende Vorlesungen.
- Einen Überblick über Themen des weiteren Studiums geben.
- Theoretische Grundlagen werden mit praktischen Beispielen verdeutlicht.

Wirtschaftsinformatik

Wirtschaftsinformatik ist eine „Schnittstellen-Disziplin“ zwischen der Informatik und den Wirtschaftswissenschaften. Von der Anwendungsseite deckt die Wirtschaftsinformatik daher konkret folgende Funktionen ab:

- Materialwirtschaft, Produktion, Logistik, Service- und Aftersales
- Kundenkontakt-Management (Customer Relationship Management)
- Vertragsmanagement und Abrechnungssysteme
- Rechnungswesen und Finanzbuchhaltung
- Projektmanagement
- Logistik-Management (Fuhrparkmanagement, Routenplanung, Frachtmanagement)

Einsatzszenario: Enterprise Information System

- Betriebliche Informationssysteme spiegeln die Geschäftsprozesse innerhalb von Unternehmen wider.
- Betriebliche Informationssysteme sind stark Datenbank-zentriert.
- Software für betriebliche Informationssysteme ist meist sehr komplex (100.000 bis mehrere Millionen Zeilen Programmcode).
- Ohne betriebliche Informationssysteme können Unternehmen heute ihr Geschäft nicht mehr betreiben.

Bezug zu der Vorlesung

Bei der Konzeption von solchen Systemen wird benötigt:

- Verständnis der abzubildenden realen Welt (Fachwissen).
- Kenntnisse der Methoden und Verfahren der Abbildung.
- Kenntnisse über die Grenzen der Abbildung.
- Kenntnisse über die Komplexität der Abbildung (Antwortzeit muss zu der Aufgabe passen).
- Kenntnisse über die Programmierung von Komponenten.
- Kenntnisse der Qualitätssicherung.
- Kenntnisse der Organisation und Dynamik von Gruppen.

Ziele der Vorlesung

- Handwerkzeug der Informatik in Grundzügen vermitteln (Informationen, Logik, Algorithmen, Grammatiken)
- Grundkenntnisse über die Komplexitätstheorie vermitteln.
- Vermitteln der verschiedenen Programmierparadigmen.
- Vermitteln der Grundlagen der Analyse und des Designs von Softwareprojekten.
- Vermitteln von Grundlagen in der Qualitätssicherung.

Grundlagen der EDV

Die elektronische Datenverarbeitung (EDV) arbeitet immer nach dem Grundprinzip

Eingabe ☞ Verarbeitung ☞ Ausgabe

Oder kurz „EVA“ Prinzip. „Eingabe“, „Verarbeitung“ und „Ausgabe“ beziehen sich hier nur auf die von Digitalrechnern handhabbaren DATEN.

Grundlagen der EDV: Hardware (1)

Ein Computer (dt. „Rechner“) kann verschiedene Hardware-Komponenten enthalten:

EINGABE-Geräte:

- Tastatur,
- Maus/Trackball/Touchpad, Joystick/Pad,
- Scanner, Mikrofon, Kamera,
- Messgeräte, Netzwerk ...

Grundlagen der EDV: Hardware (2)

Für die interne Technik:

VERARBEITUNGS-Komponenten:

- Zentrale Recheneinheit (CPU),
- Flüchtiger Speicher (RAM),
- Permanenter Speicher (BIOS, Festplatte, CD/DVD-Rom, FLASH, ...),
- Controller zur Schnittstellenverwaltung.

Vor allem die Kenntnis der Funktionsweise von flüchtigem und permanentem Datenspeicher ist wichtig für die Datensicherheit und Persistenz der Arbeitsergebnisse.

Grundlagen der EDV: Hardware (3)

Für den Zugriff auf die Ergebnisse:

AUSGABE-Komponenten:

- Grafikkarte ☞ Bildschirm/Display,
- Drucker,
- Soundkarte ☞ Lautsprecher/Kopfhörer,
- Braillezeile, Sprachausgabe, 3D-Displays/Lithographie,
- Netzwerk, Indikator-Geräte (LEDs, Anzeigen, DA-Wandler, ...)

Grundlagen der EDV: Software

Systemsoftware/Betriebssystem: Steuerung des Systems, macht die Hardwarekomponenten für Anwendungen erst verfügbar, verwaltet den Zugriff auf alle Resourcen des Systems, Sicherheit und Konsistenz von Benutzerdaten und Prozessen. ☞ Schnittstelle zwischen Hardware und Anwendersoftware.

Anwendersoftware: Interaktion des Anwenders mit dem Computer, erledigt Aufgaben wie Textverarbeitung, Tabellenkalkulation, Grafikerstellung, Nutzung von Internet-Diensten. ☞ Schnittstelle zwischen Anwender und Daten.

Informatik vs. Information

- Informatik besteht aus Information und Automatik.
- Der Begriff Information ist schwer zu definieren.
- Die Begriffe Information und Bedeutung sind unterschiedlich zu besetzen.
- Die Philosophie beschäftigt sich heutzutage mit der Abgrenzung von Information und Bedeutung.
- Nachrichtentechnik beschäftigt sich mit (technischen Aspekten von) Informationen (Claude Shannon).

Ebenen der Information

Es existieren drei Ebenen, unter denen der Begriff Information heute betrachtet wird:

- Syntax
- Semantik
- Pragmatik

Repräsentationen einer Information

Eine Information mit Syntax, Semantik und Pragmatik kann durch verschiedene Repräsentationen dargestellt werden.

- Jede Information besitzt mindestens eine Repräsentation.
- Der Satz „Heute ist ein schöner Tag“ kann aufgeschrieben oder gesprochen werden.
- Die römische Zahl **VI** und die arabische Zahl **6** sind zwei verschiedene Repräsentationen für den gleichen Wert.

Syntaktische Ebene der Information

- Auf der syntaktischen Ebene wird Information nur als Struktur gesehen.
- Inhalt hat keine Bedeutung.
- Der Informationsgehalt ist dabei ein Maß für die maximale Effizienz, mit der die Information verlustfrei übertragen werden kann.

Bei der Übertragung von Informationen von einem DVD-Player zu einem Fernseher interessiert sich das Kabel nicht dafür, welche Filme gerade übertragen werden. Auch die inhaltliche Qualität der Filme spielt keine Rolle. Die zu übertragenden Informationen müssen lediglich syntaktisch korrekt sein.

Unterscheidbarkeit und Informationsgehalt

- Grundprinzip der syntaktischen Information ist die Unterscheidbarkeit: Information enthält, was unterschieden werden kann.
- Unterscheidung setzt jedoch mindestens zwei unterschiedliche Möglichkeiten voraus.
- Der Informationsgehalt lässt sich direkt aus den unterschiedlichen Möglichkeiten bestimmen.

Bestimmung Informationsgehalt

1. Segmentierung der unterschiedliche Möglichkeiten mit Hilfe von Ja-Nein-Fragen.
2. Finden einer idealen Fragereihenfolge (unter Berücksichtigung von Wahrscheinlichkeiten).

Der Informationsgehalt einer Struktur bestimmt sich nun dadurch, wieviele Ja-Nein-Fragen man im Mittel bei einer idealen Fragestrategie braucht.

Beispiel Informationsgehalt (1)

Die Speisekarte eines Restaurants führt genau zwei unterschiedliche Speisen (Hamburger und Salat). Wie hoch ist der Informationsgehalt bei der Bestellung von Speisen bei einem Mitarbeiter?

Antwort: Der Informationsgehalt ist 1, da auf die Frage des Mitarbeiters „Wollen Sie einen Hamburger?“ mit einer Antwort des Bestellers entweder ein Hamburger oder ein Salat bestellt werden kann.

Beispiel Informationsgehalt (2)

Der Informationsgehalt der Zahlen von 0 bis 15 soll untersucht werden. Dieses Problem ist vergleichbar mit dem Spiel bei dem mit möglichst wenigen JA/NEIN-Fragen eine vorher festgelegte Zahl zwischen 0 und 15 erraten werden soll.

Naiver Ansatz: alle Zahlen durchraten (Ist es die Zahl ...)
☞ Ineffizient.

Geschickter Ansatz: Mit einer Frage immer (mindestens) die Hälfte der möglichen Zahlen ausschließen („größer als ...?“, „teilbar durch ...?“).
☞Der Informationsgehalt ist 4.

Darstellung Informationsgehalt

Reiht man die Fragen hintereinander auf, so erhält man eine Tabelle, in der die Antworten vermerkt werden können. Jede Antwortzeile kann auch kurz als Zustand bezeichnet werden.

F1	F2	F3	F4
ja	nein	ja	nein
nein	nein	ja	nein

Definition: Die kleinstmögliche Unterscheidung zwischen zwei Zuständen wird als **Bit** bezeichnet.

Semantische Ebene der Information

- Informationen ohne Bedeutung sind nutzlos.
- Der Syntax von Informationen wird interpretiert, um eine Bedeutung zu erhalten.
- Ein bestimmtes Bezugssystem muss angelegt werden, um die Bedeutung einer Repräsentation (Information) zu erhalten.
- Dieses Bezugssystem wird durch eine Interpretationsfunktion I festgelegt.

Interpretationsfunktionen

- Interpretationsfunktionen sind definiert durch:

$$
\begin{aligned}
W &= \text{Menge aller Repräsentationen} \\
O &= \text{Menge von Objekten} \\
I &: W \rightarrow O \\
I(r \in W) &= o, \text{bilde Repräsentation } r \text{ auf Objekt } o \in O \text{ ab}
\end{aligned}
$$

- Die Art und Weise wie die Funktion einzelnen Repräsentationen Objekte zuordnet, charakterisiert die Funktion ☞ es existieren viele Interpretationsfunktionen.

Die Menge O ist Teil des Bezugssystems.

Beispiele Interpretationsfunktionen

Sei W die Menge $\{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

- I_1 bildet die Teilmenge $\{0, 2, 4, 6, 8\}$ auf das Objekt „gerade Zahl“ und die Teilmenge $\{1, 3, 5, 7\}$ auf das Objekt „ungerade Zahl“ ab. Die Repräsentation 2 wird somit mit Hilfe von I_1 als „gerade Zahl“ interpretiert.
- I_2 bildet die Menge W auf die Wahrheitswerte „wahr“ und „falsch“ ab. Dabei wird 0 auf das Objekt „falsch“ abgebildet, alle anderen Repräsentation auf „wahr“. Die Repräsentation 2 wird somit mit Hilfe von I_2 als „wahr“ interpretiert.

Pragmatische Ebene der Information

- Diese Ebene kommt dem umgangssprachlichen Informationsbegriff am nächsten.
- Der pragmatische Informationsgehalt stellt den Informationsgewinn dar.
- Ohne Informationsgewinn ist eine Information wertlos.

In diesem pragmatischen Sinne ist ein wesentliches Kriterium von Information, dass sie das Subjekt, das die Information aufnimmt, verändert.

Beispiel pragmatische Ebene

Beispiel: Wie hoch ist der Informationsgewinn, wenn ich beim Warten auf dem Bus total durchnässt im Radio höre, dass es im gesamten Sendegebiet zurzeit regnet? Wie hoch ist der Informationsgewinn, wenn ich die selbe Information bereits vor Verlassen der Wohnung gehabt hätte?

- Wenn ich bereits im Regen stehe, stellt die Information „es regnet“ **keinen** Informationsgewinn dar. Diese Information bewirkt keine Veränderung des Systems.
- Wenn ich noch in meiner Wohnung bin, stellt die Information „es regnet“ **einen** Informationsgewinn dar. Das System kann sich verändern, indem ich z.B. einen Regenschirm einpacke.

Zusammenfassung

- Information ist ein Gewinn an Wissen.
- Information ist die Verringerung von Ungewissheit.
- Information ist eine Mitteilung, die den Zustand des Empfängers ändert.

Konsequenzen (1)

- Unterschiedliche Empfängers können Informationen unterschiedlich interpretieren.
- Empfänger müssen sich somit bezüglich der Interpretationsfunktionen einigen.

```
Metric conversion mistake made Mars orbiter crash

WASHINGTON (AP) -- For nine months, the Mars Climate
Orbiter was speeding through space and speaking to
NASA in metric. But the engineers on the ground were
replying in non-metric English.
```

Konsequenzen (2)

- Im Softwareerstellungsprozess sollten alle Beteiligten alle Informationen gleich interpretieren (Vertrieb, Programmierer, Kunden).
- Formale Spezifikation hilft, eine gemeinsame Interpretationsfunktion zu finden.
- Was nicht festgelegt wird, kann beliebig interpretiert werden!
  ☞ Quelle für mögliche Probleme.

Mathematische Grundlagen: Motivation

- Frühzeitig an die Strenge des mathematischen Sprachgebrauchs gewöhnen,
- die Terminologie der Mengenlehre wiederholen,
- eine „Interpretationsfunktion“ für die in dieser Vorlesung benötigten mathematischen Grundlagen (Repräsentationsmengen, Funktionen, später Grammatiken) festlegen.

Mathematische Grundlagen

- Mengen
- Relationen, Funktionen, Abbildungen

Mengen

- Beschreibung von Mengen
- Die Teilmenge
- Die leere Menge
- Durchschnitt und Vereinigung
- Differenz von Mengen
- Die Potenzmenge

Beschreibung von Mengen (1)

Unter einer **Menge** verstehen wir die Zusammenfassung gewisser Objekte unserer Anschauung oder unseres Denkens.

- Zwei Mengen sind gleich ($N = M$), wenn sie aus genau denselben Elementen bestehen.
- Es dürfen nur solche Mengen gebaut werden, bei denen entscheidbar ist, ob ein gewisses Objekt Element dieser Menge ist.

Beispiel 1: Die Menge aller natürlichen Zahlen von 3 bis einschließlich 9

Beispiel 2: Die Menge aller Studentinnen und Studenten in dieser Vorlesung.

Warum ist die „Menge aller sehr großen natürlichen Zahlen“ keine Menge?

Beschreibung von Mengen (2)

- Besteht eine Menge aus endlich vielen Elementen, so kann die Menge durch explizite Angabe der Elemente eindeutig beschrieben werden: $N = \{3, 4, 5, 6, 7, 8, 9\}$
- Bei Mengen ist die Reihenfolge ohne Bedeutung: $N = \{9, 3, 8, 5, 7, 6, 4\}$
- Man kann Elemente einer Menge beliebig oft auflisten, ohne dass sich die Menge ändert.
- (unendliche) Mengen können durch Eigenschaften beschrieben werden $M = \{x \mid x \text{ ist eine gerade Zahl}\}$

Teilmengen

Seien M und N Mengen, so gelten folgende Definitionen:

1. $x \in M$ bedeutet x ist Element von M
2. $x \notin M$ bedeutet x ist nicht Element von M
3. Wenn für alle $x \in N$ gilt $x \in M$, dann ist N eine **Teilmenge** von M. Oder kurz $N \subset M$.
4. Wenn $M = N$, so gilt $M \subset N$ und $N \subset M$.

Die leere Menge

Es hat sich als nützlich erwiesen, die Existenz einer Menge vorauszusetzen, die kein Element besitzt. Diese Menge bezeichnen wir als **leere Menge** $\emptyset$.

- Für jede Menge M gilt: $\emptyset \subset M$
- Sei M eine Menge, dann gilt: M ist nichtleer gdw.[a] (mindestens) ein $x \in M$ existiert.

[a] **g**enau **d**ann, **w**enn

Durchschnitt und Vereinigung

Durchschnitt und Vereinigung sind Operationen auf Mengen. Sie erzeugen aus zwei Mengen (M, N) eine neue Menge.

- Durchschnitt:[a] $M \cap N := \{x \mid x \in M \text{ und } x \in N\}$
- Vereinigung:[b] $M \cup N := \{x \mid x \in M \text{ oder } x \in N\}$

Übung: Skizzieren Sie den Durchschnitt und die Vereinigung von Mengen in der bekannten Form.

---

[a] Wir werden später sehen, dass dies der logischen „und“-Verknüpfung, bzw. der Multiplikation entspricht.

[b] Wir werden später sehen, dass dies der logischen „oder“-Verknüpfung, bzw. der Addition entspricht.

Definitionen

L, M und N seien Mengen. Dann gilt:

(1) $M \cap N \subset M$, $M \subset M \cup N$ (offensichtlich)

(2) $M \cap N = N \cap M$, $M \cup N = N \cup M$ (Kommutativgesetze)

(3) $M \cap (N \cap L) = (M \cap N) \cap L$, $M \cup (N \cup L) = (M \cup N) \cup L$ (Assoziativgesetze)

(4) $M \cap (N \cup L) = (M \cap N) \cup (M \cap L)$ (Distributivgesetz)

Übung: Lesen dieser formale Beschreibung in „natürlicher Sprache“ zum besseren Verständnis + Aufzeichnen.

Differenz von Mengen

Die Differenzbildung ist eine weitere Möglichkeit, um aus zwei Mengen eine „neue“ zu bilden.
M und N seien Mengen. Dann heißt

$$M \setminus N := \{x \mid x \in M \text{ und } x \notin N\}$$

Differenz von M und N.

Die Potenzmenge

Hat man die Menge M, so kann man alle Teilmengen von M zu einer neuen Menge, der sogenannten Potenzmenge (2^M oder $\mathcal{P}(M)$), zusammenfassen.
M sei eine Menge. Dann heißt

$$\mathcal{P}(M) := \{N \mid N \subset M\}$$

Potenzmenge von M.
Beispiel: Für $M = \{1, 2, 3\}$ ist

$$\mathcal{P}(M) := \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}.$$

Kann es leere Potenzmengen geben? Potenzmenge einer leeren Menge?

Relationen, Funktionen, Abbildungen

- Paarbildung und Kreuzprodukt
- Relationen
- Funktionen
- Abbildungen

Paarbildung

Normalerweise beschreibt man einen Punkt in der Ebene durch zwei Zahlen. Man führt ein Koordinatensystem ein, das aus zwei senkrecht zueinander stehenden Achsen, häufig als x- und y-Achse bezeichnet, besteht. Den Schnittpunkt der Achsen bezeichen wir als Bezugspunkt. Die Position eines Punktes in diesem Koordinatensystem lässt sich nun als Abstand zum Bezugspunkt ausdrücken.

$$P = (p_1, p_2)$$

p_1 ist dabei der senkrechte Abstand des Punktes zur y-Achse.
p_2 ist dabei der senkrechte Abstand des Punktes zur x-Achse.

Die Reihenfolge der Komponenten p_1 und p_2 ist zu beachten. Der Punkt $(1, 2)$ ist verschieden zum Punkt $(2, 1)$. (p_1, p_2) heißt daher auch **geordnetes Paar**

Kreuzprodukt

Sind M und N Mengen, so können wir die Menge $M \times N$ aller geordneten Paare (x, y) mit $x \in M$ und $y \in N$ bilden:

$$M \times N := \{(x, y) \mid x \in M \text{ und } y \in N\}$$

Die Menge $M \times N$ bezeichnen wir als **kartesisches Produkt** oder **Kreuzprodukt** der Mengen M und N.
Man kann nicht nur zwei Objekte x und y zu einem geordneten Paar (x, y) zusammenfügen, sondern n Objekte $x_1, x_2, \ldots, x_n$ zum **geordneten n-Tupel** $(x_1, x_2, \ldots, x_n)$.

Relationen

M und N seien Mengen. Eine Menge R heißt **Relation zwischen** M **und** N genau dann, wenn $R \subset M \times N$ gilt. Für $x \in M$ und $y \in N$ sagt man auch x und y erfüllen R (oder x und y stehen in der Relation R), wenn $(x, y) \in R$ gilt.

Definitions- und Wertebereich

R sei eine Relation. Dann heißt

$$\mathcal{D}(R) := \{x \mid \text{ es gibt ein } y \text{ mit } (x, y) \in R\}$$

Definitionsbereich von R und

$$\mathcal{W}(R) := \{y \mid \text{ es gibt ein } x \text{ mit } (x, y) \in R\}$$

Wertebereich von R.

Die nicht-mathematische Umschreibung:
Der **Definitionsbereich** legt fest, welche **Eingabewerte** erlaubt sind.
Der **Wertebereich** entspricht den **Ausgabewerten**, die berechenbar sind.

Funktionen

F heißt **Funktion** oder **eindeutige Relation** genau dann, wenn gilt:

- F ist eine Relation
- Für alle x, y und z gilt:
  $(x, y) \in F$ und $(x, z) \in F$ folgt $y = z$.

Eine Funktion F wird meist durch eine **Signatur** eindeutig beschrieben. Zu der Signatur gehören:

- Ein Bezeichner der Funktion.
- Angabe des Werte- und Definitionsbereichs.
- Eine Abbildungsvorschrift.

Das Tupel (x, y) einer Funktion f kann auch als $y = f(x)$ bezeichnet werden.

Injektivität, Surjektivität und Bijektivität

Sei f eine Funktion und Teilmenge von $N \times M$, so sagt man auch f bildet Objekte aus N auf M ab (f ist Abbildung von $N \rightarrow M$). Wir sagen diese Abbildung ist

surjektiv genau dann, wenn gilt:

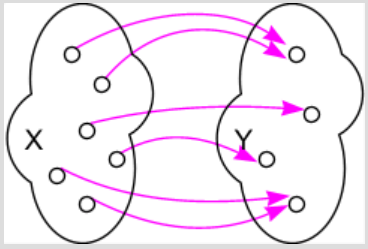
Zu jedem $y \in N$ gibt es (mindestens) ein $x \in M$ mit $f(x) = y$

injektiv genau dann, wenn gilt:

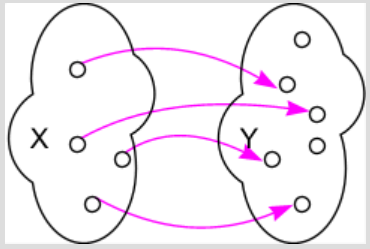
Für alle $x_1, x_2 \in M$ mit $f(x_1) = f(x_2)$ folgt $x_1 = x_2$

bijektiv genau dann, wenn gilt:

f ist surjektiv und injektiv.



surjektiv



injektiv

Mächtigkeit von Mengen

Sind M und N Mengen, so definiert man:
M und N sind **gleichmächtig** genau dann, wenn es eine bijektive Abbildung $f : M \to N$ gibt.

Äquivalenzrelationen

R sei eine Relation auf einer Menge M. Wir definieren nun[a]

- R ist **reflexiv auf** M genau dann, wenn für alle $x \in M$ gilt $(x, x) \in R$.
- R ist **symmetrisch auf** M genau dann, wenn für alle $x, y \in M$ gilt:
  Wenn $(x, y) \in R$ ist, so ist auch $(y, x) \in R$.
- R ist **transitiv auf** M genau dann, wenn für alle $x, y, z \in M$ gilt:
  Wenn $(x, y) \in R$ und $(y, z) \in R$ dann ist auch $(x, z) \in R$

[a] NB: Diese Folie ist ein Löschkandidat für die Vorlesung im nächsten Semester, da sie zum Verständnis von Algorithmen und Funktionen in der angewandten Informatik nicht allzuviel beiträgt.

Prädikate

P sei eine Relation auf einer Menge M. Statt $x \in P$ schreiben wir kurz Px, statt $x \notin P$ schreiben wir kurz $\neg Px$. Für jedes Objekt x kann eindeutig festgestellt werden, ob dieses Objekt in der Relation enthalten ist. Zu jeder Relation R existiert eine **charakteristische Funktion** χ_R.

$$\chi_R x = \begin{cases} 1 & \text{falls } Px \\ 0 & \text{falls } \neg Px \end{cases}$$

Der Wertebereich der charakteristische Funktion kann zu den Wahrheitswerten $\{$ wahr, falsch $\}$ interpretiert (ausgewertet) werden. Mathematisch sind **Prädikate** ein Synonym für Relationen. Umgangssprachlich wird eher die charakteristische Funktion einer Relation mit dem Begriff Prädikat identifiziert.

n-stellige Operationen

Jedes $f : M^n \to M$ heißt eine **n-stellige Operation** auf M. Für $f(a_1, \ldots, a_n)$ führen wir die abkürzende Schreibweise $f\vec{a}$ ein. Eine 0-stellige Operation hat mengenthorethisch die Gestalt $\{(\emptyset, c)\}$ mit $c \in M$ und wird auch als **Konstante** mit dem Wert c bezeichnet. Am häufigsten werden 2-stellige Operationen angetroffen. Bei diesen wird das entsprechende Operationssymbol in der Regel zwischen die Argumente gesetzt. Eine mit $\circ$ bezeichnete Operation $\circ : M^2 \to M$ heißt

kommutativ , wenn $a \circ b = b \circ a$ für alle $a, b \in M$

assoziativ , wenn $a \circ (b \circ c) = (a \circ b) \circ c$ für alle $a, b, c \in M$

idempotent , wenn $a \circ a = a$ für alle $a \in M$

invertierbar , wenn zu allen $a, b \in M$ Elemente $x, y \in M$ existieren mit $x \circ a = b$ und $a \circ y = b$

2-wertige Logik

In der 2-wertigen Logik existieren nur zwei verschiedene Wahrheitswerte, nämlich **wahr** und **falsch**. Wir verwenden für diese Wahrheitswerte folgende Repräsentation $\{1, 0\}$. 1 wird dabei als wahr interpretiert, 0 als falsch. Operationen auf Wahrheitswerte können mit Hilfe einer Wertetabelle beschrieben werden. Sei $M = \{0, 1\}$ und $\circ$ eine 2-stellige Operation auf M.

a	b	$a \circ b$
0	0	1
0	1	0
1	0	0
1	1	0

Nun können anstelle von $\circ$ konkrete Operationen eingeführt werden.

Konjunktion

Semantik: A und B; Sowohl A als auch B
Symbol: $\wedge$

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

Umgangssprachlich: „Und“-Verknüpfung. (Das Ergebnis ist NUR DANN wahr, wenn alle Operanden wahr sind).

Disjunktion

Semantik: A oder B
Symbol: $\vee$

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Umgangssprachlich: „Oder“-Verknüpfung. (Das Ergebnis ist IMMER DANN wahr, wenn MINDESTENS einer der Operanden wahr ist.)

Implikation

Semantik: Wenn A so B; Aus A folgt B
Symbol: $\rightarrow$

a	b	$a \rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

Äquivalenz

Semantik: A genau dann wenn B; A dann und nur dann wenn B
Symbol: $\equiv$

a	b	$a \equiv b$
0	0	1
0	1	0
1	0	0
1	1	1

Antivalenz

Semantik: Entweder nur A, oder nur B
Symbol: $+$

a	b	$a+b$
0	0	0
0	1	1
1	0	1
1	1	0

Nihilition

Semantik: Weder A noch B
Symbol: $\downarrow$

a	b	$a \downarrow b$
0	0	1
0	1	0
1	0	0
1	1	0

Unverträglichkeit

Semantik: Nicht zugleich A und B
Symbol: $\uparrow$

a	b	$a \uparrow b$
0	0	1
0	1	1
1	0	1
1	1	0

„WAHR, wenn NICHT zugleich A und B wahr sind. Sonst FALSCH.“

Negation

1-stellige Operation
Semantik: Wenn wahr, dann falsch. Wenn falsch, dann wahr.
Symbol: $\neg$

a	$\neg a$
0	1
1	0

Aussagenlogische Formeln

Wir wollen einen Formalismus schaffen, um logische Äquivalenzen zu erkennen und zu beschreiben. n-stellige Operationen sollen durch Formeln beschrieben werden. Dazu werden **Variablen** eingeführt. Variablen sind Platzhalter für die Werte wahr oder falsch und werden traditionell mit Hilfe der Symbole $p_0, p_1, \ldots$ dargestellt.

Induktive Formelbestimmung

F1 Die Variablen $p_0, p_1, \ldots$ sind Formeln, auch **Primformeln** genannt.

F2 Sind α, β Formeln, so sind auch die Zeichenfolgen $(\alpha \vee \beta), (\alpha \wedge \beta)$ und $\neg a$ Formeln.

Semantische Äquivalenz

Wann sind zwei Formeln α, β semantisch äquivalent?

Genau dann, wenn sie den gleichen Wertverlauf besitzen, also bei gleichen Eingaben, den gleichen Ausgabewert besitzen. Dies kann mit Hilfe von Wertetabellen überprüft werden.
Beispiel: Ist $\alpha \wedge \alpha$ semantisch äquivalent mit α?

a	a	$\alpha \wedge \alpha$
0	0	0
1	1	1

Repräsentation von Zahlen

- Zahlen können auf unterschiedliche Arten dargestellt werden
- Aufgabe: Zahlen aus der „realen Welt“ müssen im Computer abgebildet werden.
- Problem: Was ist die größte Zahl aus der Menge der natürlichen Zahlen?
- Problem: Wieviel (endlichen) Speicher braucht man für „unendlich“?

Abbilden von natürlichen Zahlen

- Abbilden von natürlichen Zahlen auf eine Folge von Bits.
- Abbilden von positiven und negativen Zahlen.
- Abbilden eines Intervalls $\rightarrow$ größte und kleinste darstellbare Zahl.

Dezimaldarstellung (1)

Die uns gebräuchlichte Darstellung von natürlichen Zahlen ist mit Hilfe der Symbole $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, +\}$. Beispiele für diese Repräsentation von Zahlen sind:

$$+23456, \qquad -18, \qquad 0$$

Zahlen beginnen mit einem Vorzeichensymbol aus der Menge $\{+, -\}$. Bei positiven Zahlen und der Null kann das Vorzeichensymbol weggelassen werden. Danach kommen Ziffernsymbole aus der Menge $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Mathematisch können die Zahlen als n-Tupel aufgefasst werden. Somit lassen sich Zahlen im Dezimalsystem folgendermaßen darstellen:

$$(c_{n-1}, \ldots, c_0) \in D^n$$

Dezimaldarstellung (2)

Es gibt eine bijektive Abbildung nach_N : $D^n \to \mathbb{N}$, die n-Tupel $\in D^n$ in die Menge der natürlichen Zahlen abbildet. Mit $c = (c_{n-1}, \ldots, c_0) \in D^n$ bildet

$$\text{nach\_N}(c) = \sum_{i=0}^{n-1} g(c_i) \cdot 10^i$$

Die Hilfsfunktion g bildet Elemente der Menge D auf die zugehörigen natürliche Zahlen ab. Die Funktion g stellt somit eine Interpretationsfunktion für die Repräsentation der natürlichen Zahlen in der Menge D dar, während nach_N eine Interpretationsfunktion für D^n ist. Durch die Existenz einer bijektiven Abbildung können wir in Zukunft natürliche Zahlen und Tupel aus D^n als Synonym betrachten.

Binärdarstellung

Positive natürliche Zahlen inklusive der Null können mit Hilfe der Symbole $B = \{1, 0\}$ repräsentiert werden. Somit werden dann alle positiven natürlichen Zahlen inklusive der Null durch Folgen von Symbolen aus der Menge B abgebildet.

$$1011011011$$

Bezeichne i die Position der Symbole in der Folge. Nun muss noch die Konvention der Stellenreihenfolge festgelegt werden, also ob wir links oder rechts anfangen, die Symbole der Folge durchzunumerieren. Zur Illustration zeigen wir die Folge mit der Position der Symbole im Index.

$$1_9 0_8 1_7 1_6 0_5 1_4 1_3 0_2 1_1 1_0$$

Die Menge aller n-stelligen Binärzahlen bezeichen wir als Menge B^n, wobei wir die Komponenten des n-Tupeln analog der Indizes durchnummerieren.

Binärdarstellung (2)

Aufgabe: Finden einer Abbildung

$$\text{binär\_nach\_dezimal}_{n,m} : B^n \to D^m$$

zur Umrechnung von Repräsentationen im Dualsystem in Repräsentationen im Dezimalsystem. Die Abbildung muß bijektiv sein und bildet n-stellige Dualzahlen auf m-stellige Dezimalzahlen ab. Für jedes $b \in B^n$ lautet die Abbildungsvorschrift:

$$\text{binär\_nach\_dezimal}_{n,m}(b) = \sum_{i=0}^{n-1} g(b,i) \cdot 2^i$$

Binärdarstellung (3)

Nun muss „nur“ noch die verwendete Hilfsfunktion $g : B^n \times IN \rightarrow IN$ bestimmt werden.

$$g(b,i) = \begin{cases} 0, \text{wenn die } i\text{-te Komponente des Tupels b } 0 \text{ entspricht.} \\ 1, \text{wenn die } i\text{-te Komponente des Tupels b } 1 \text{ entspricht.} \end{cases}$$

Wenn man die Symbole $\{0,1\}$ direkt als natürliche Zahlen interpretiert, kann die Hilfsfunktion g eingespart werden. Sei b eine Dualzahl als n-Tupel, dann bezeichne b_i die i-te Komponente des n-Tupels:

$$\text{binär\_nach\_dezimal}_{n,m}(b) = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Binärdarstellung (4)

Beispiel: Umrechnen der Binärzahl 100100100 in eine Dezimalzahl:

binär_nach_dezimal$_{9,3}((1,0,0,1,0,0,1,0,0))$

$$= \sum_{i=0}^{n-1} b_i \cdot 2^i$$

$$= 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$$

$$= 292$$

Binärdarstellung (5)

Aufgabe: Finden einer Abbildung

$$\text{dezimal\_nach\_binär}_{n,m} : D^n \rightarrow B^m$$

zur Umrechnung von Repräsentationen im Dezimalsystem in Repräsentationen im Dualsystem. Die Abbildung muß bijektiv sein und bildet n-stellige Dezimalzahlen a auf m-stellige Dualzahlen b ab. Da die Abbildung binär_nach_dezimal injektiv ist, existiert ein Tupel $b \in B^m$ mit $\text{binär\_nach\_dezimal}_{m,n} = a$. Dieses Tupel b soll nun aus der Dezimalzahl a konstruiert werden.

Binärdarstellung (6)

Zuerst konstruieren wir eine Hilfsfunktion $f : D^n \times \mathbb{N} \rightarrow B$, die uns für eine gegeben Dezimalzahl a den Wert der i-ten Komponente des gesuchten Tupels liefert.

$$f(a,i) = \begin{cases} 1 & , 2^{i+1} \geq a \geq 2^i \\ f(a-2^m, i) & , 2^{m+1} \geq a \geq 2^m, m \neq i \\ 0 & , \text{sonst} \end{cases}$$

Beispiel: Abbildung Dezimalzahl 292 $\rightarrow$ Dualzahl ???

$f(292,8) = 1,$
$f(292,7) = f(36,7) = 0,$
$f(292,6) = f(36,6) = 0,$
$f(292,5) = f(36,5) = 1,$
$f(292,4) = f(36,4) = f(4,4) = 0,$

$f(292,3) = f(36,3) = f(4,3) = 0,$
$f(292,2) = f(36,2) = f(4,2) = 1,$
$f(292,1) = f(36,1) = f(4,1) = f(0,1) = 0,$
$f(292,0) = f(36,0) = f(4,0) = f(0,0) = 0.$
$\Rightarrow$ 100100100

Binärdarstellung (7)

Mit Hilfe der Hilfsfunktion f kann nun die Funktion

$$\text{dezimal\_nach\_binär}_{n,m} : D^n \to B^m$$

folgendermaßen konstruiert werden. Sei $a \in D^n$, dann gilt:

$$\text{dezimal\_nach\_binär}_{n,m}(a) = (f(a,m), f(a,m-1), \ldots, f(a,0))$$

Rechnen mit Binärzahlen

Addition und Subtraktion ist einfacher als Multiplikation und Division. Im Nachfolgenden soll exemplarisch das Rechnen mit Binärzahlen anhand der Addition gezeigt werden. Im Grunde läßt sich die Addition auf die Fälle $0+0=0, 0+1=1, 1+0=1$ und $1+1=10$ reduzieren. Der Fall $1+1=10$ ist ein Sonderfall, da hier ein Überlauf auftritt. Berücksichtigt man den Überlauf, so ergeben sich folgende Fälle:

1. Summand (n-te Komponente)	0	0	0	0	1	1	1	1
2. Summand (n-te Komponente)	0	0	1	1	0	0	1	1
Überlauf (n-te Komponente)	0	1	0	1	0	1	0	1
Ergebnis (n-te Komponente)	0	1	1	0	1	0	0	1
Überlauf (n+1-te Komponente)	0	0	0	1	0	1	1	1

Das Ergebnis einer Addition zweier n-Tupel kann durch den Überlauf ein $n+1$-Tupel werden.

Hexadezimalzahlen

Analog zum Binärsystem lassen sich noch andere Repräsentationen konstruieren. In der Informatik hat das Hexadezimalsystem große Verbreitung gefunden. Im Hexadezimalsystem (griech. hexa „sechs“, lat. decem „zehn“, auch Sedezimalsystem von lat. sedecim „sechzehn“) werden Zahlen in einem Stellenwertsystem mit der Basis 16 (also einem 16er-System) dargestellt.

Dezimal	Hexadezimal	Binär	Dezimal	Hexadezimal	Binär
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Hexadezimalzahlen (2)

Das Hexadezimalsystem eignet sich sehr gut, um Folgen von Bits (verwendet in der Digitaltechnik) darzustellen. Vier Stellen einer Bitfolge (ein Nibble) werden wie eine Dualzahl interpretiert und entsprechen so einer Ziffer des Hexadezimalsystems da 16 die vierte Potenz von 2 ist. Die Hexadezimaldarstellung der Bitfolgen ist leichter zu lesen und schneller zu schreiben:

Binärzahl	1101	0110
Hexadezimalzahl	D	6

Hexadezimalzahlen (3)

Eine Möglichkeit, eine Zahl des Dezimalsystems in eine Zahl des Hexadezimalsystems umzurechnen, ist die Betrachtung der Divisionsreste, die entstehen, wenn die Zahl durch die Basis 16 geteilt wird.
Die Dezimalzahl 1278 lässt sich folgendermaßen in eine Hexadezimalzahl umwandeln:

$$\begin{array}{rcrcrcl} 1278 & : & 16 & = & 79 & \text{Rest} & 14 \quad (= E) \\ 79 & : & 16 & = & 4 & \text{Rest} & 15 \quad (= F) \\ 4 & : & 16 & = & 0 & \text{Rest} & 4 \quad (= 4) \end{array}$$

Von unten nach oben gelesen ergibt sich die Hexadezimalzahl **4FE**.

Hexadezimalzahlen (4)

Um eine Hexadezimalzahl in eine Dezimalzahl umzuwandeln, muss man die einzelnen Ziffern mit der jeweiligen Potenz der Basis multiplizieren. Der Exponent der Basis entspricht der Stelle der Ziffer, wobei die am weitesten links stehende Zahl eine Null zugeordnet wird. Dazu muss man allerdings die Ziffern A, B, C, D, E, F in die entsprechenden Dezimalzahlen $10, 11, 12, 13, 14, 15$ umwandeln.

$$4 \cdot 16^2 + 15 \cdot 16^1 + 14 \cdot 16^0 = 1278$$

Im Prinzip ist die mathematische Umrechnungsvorschrift analog zu der Vorschrift zum Umrechnen von Binärzahlen. Bei gegebener Hexadezimalzahl $(c_{n-1}, \ldots, c_0)$ ist die Vorschrift:

$$a = \sum_{i=0}^{n-1} c_i \cdot 16^i$$

Unterscheidung von Repräsentationen

Damit Repräsentationen in jedem Fall richtig interpretiert werden können, ist es unter Umständen notwendig, die Repräsentation eindeutig zu kennzeichnen.

Motivation: Ist die Repräsentation 10 nun eine Dezimal–, eine Hexadezimal– oder sogar eine Binärzahl? Der interpretierte Wert der Repräsentation wäre bei den drei Möglichkeiten unterschiedlich.

Lösung: Wenn es aus dem Kontext nicht ersichtlich ist, um welche Repräsentation es sich handelt, so muss die Repräsentation eindeutig gekennnzeichnet werden. Dies kann z.B. durch einen Zusatz (Markierung) wie $10_{\text{binär}}$ oder 10_2 für Binärzahlen geschehen. Somit sind $10_{16}, 10_{10}$ und 10_2 unterscheidbar.

Beispiele in C: `0x7b` `\173` `123`

Fließkommazahlen

- Frage: Wie lassen sich Zahlen mit Nachkommastellen (Fließkommazahlen) darstellen?
- Fließkommazahlen können aus zwei natürlichen Zahlen konstruiert werden. Dazu muss die Fließkommazahl mit Hilfe eines Bruchs ($\frac{a}{b}$) angenähert werden. Das Problem besteht nun in der Bestimmung von a und b. Die Bestimmung von a und b kann sich als recht kompliziert erweisen!
- Fließkommazahlen können mit Hilfe der mathematischen Formel $z = m \cdot b^e$ angenährt werden.

Fließkommazahlen (2)

Bei der Darstellung von Fließkommazahlen mit Hilfe der Formel $z = m \cdot b^e$ bezeichnet man

- z als Fließkommazahl,
- m als Mantisse,
- b als Basis und
- e als Exponent.

Die Darstellung von $1,23$ kann nun durch geschickte Wahl von m, b und e erfolgen. Setzen wir die Basis auf den Wert 10, so wäre eine mögliche Lösung:

$$1,23 = 123 \cdot 10^{-2}$$

Durch den Exponent -2 wird die Mantisse 123 durch 100 geteilt, also das Komma um zwei Positionen nach „Links“ verschoben.

Genauigkeit der Zahlen

Bei der Speicherung von Zahlen in „real existierenden“ Speichern sind der Genauigkeit der Zahlendarstellung Grenzen gesetzt. Diese Grenzen resultieren in der Unmöglichkeit zur Abbildung von unendlich vielen Zahlen mit endlich vielen Speicherplätzen. Es müssen somit Kompromisse bezgl. der Abbildung geschlossen werden.

Ganze Zahlen: Es ist nur ein beschränktes Intervall der natürlichen Zahlen abbildbar. Die Größe des Intervalls kann varieren, umfasst aber niemals den gesamten Zahlenraum der natürlichen Zahlen. Die Größe des Intervalls hängt mit der Anzahl der zur Speicherung der Zahl verwendeten Bits ab. Bei Verwendung von 32 Bit können 4294967296 unterschiedliche Zahlen gespeichert werden.

Genauigkeit der Zahlen

Fließkomma-Zahlen: Auch hier können nicht alle Zahlen abgebildet werden. Es entstehen Lücken in der Abbildung, es gibt also Zahlen, die nicht darstellbar sind. Bzw. es gibt unterschiedliche Zahlen, die durch dieselbe Abbildung angenährt werden.

Konsequenzen: Die Informatik muss berücksichtigen, dass bei Abbildungen von Zahlen Probleme und Rechenungenauigkeiten auftreten können. Diese Probleme dürfen nicht zu Fehlern im Programmablauf oder Systemausfällen führen.

Syntax von Repräsentationen

Repräsentationen besitzen zwei wichtige Komponenten,

- die Semantik und
- die Syntax.

Bislang haben wir die Syntax einer Repräsentation nicht näher betrachtet. Dies soll nun in diesem Abschnitt geschehen. Wir beschäftigen uns auf den nächsten Folien mit der Frage, wann eine Repräsentation syntaktisch korrekt ist und wann nicht. Dabei beschränken wir uns auf eine Teilmenge aller möglichen Repräsentationen, indem wir nur Repräsentationen betrachten, die mit Hilfe von Zeichen dargestellt werden können.

Beispiele: Stellt die Zeichenfolge `35. Dezember 1888` ein syntaktisch korrektes Datum dar? Ist der Satz `Invormaticker schreiben dank Rechtschreibprüfunk felerfrei` syntaktisch korrekt? Wie ist die Abgrenzung zu semantischer Korrektheit?

Definitionen

Alphabet und Worte Ein **Alphabet** T ist eine endliche, nichtleere Teilmenge von Symbolen. Jede Folge $t_1 \dots t_n$ von Symbolen mit der Eigenschaft $t_i \in T$ wird **Wort** genannt. Anders ausgedrückt ist ein Alphabet eine Menge von Zeichen (Symbolen), die zur Darstellung zur Verfügung stehen. Das Alphabet der lateinischen Buchstaben unterscheidet sich vom Alphabet der chinesischen Symbole. Worte sind Repräsentationen, die nur Symbole eines bestimmten Alphabets zur Darstellung benutzen.

Sternhülle eines Alphabets Unter der Sternhülle T^* eines Alphabets T versteht man die Menge aller Worte, die mit Hilfe des Alphabets gebildet werden können.

Wortlänge

Aufbauend auf den Definitionen lässt sich nun eine Funktion

$$l : T^* \rightarrow I\!N$$

zur Bestimmung der Wortlänge definieren. Sei $w = t_1 \dots t_n$ ein beliebiges Wort aus der Sternhülle des Alphabets T, so ist

$$l(w) = n \text{ oder kurz } |w| = n$$

Das leere Wort Das leere Wort ε ist als Wort mit der Länge 0 definiert. Es gilt somit $|\varepsilon| = 0$

Mengen von Repräsentationen

In der Praxis ist es wichtig, Repräsentationen zu Mengen zusammenzufügen. Wir bezeichnen eine Repräsentation r als **syntaktisch korrekt** bezüglich der Menge von Repräsentationen R, wenn $r \in R$. Ansonsten bezeichnen wir r als **syntaktisch fehlerhaft** bezüglich R.

Nun stellt sich die Frage wie festgestellt werden kann, ob eine Repräsentation zu einer Menge gehört oder nicht. Warum wird uns der naive Ansatz, der Speicherung aller Elemente einer (möglicherweise unendlichen) Menge und vergleichen auf Übereinstimmung in der Praxis nicht praktikabel?

Anderer Ansatz: Beschreiben der Syntax mit Hilfe von Regeln.

Beschreibung von Syntax

Die Syntax kann mit Hilfe von Regeln beschrieben werden. Diese Regeln können sowohl

- umgangssprachlich oder
- mit Hilfe einer formalen Methode

beschrieben werden. Die Schwierigkeit besteht dabei nicht in der Beschreibung des Syntax einer einzelnen Repräsentation, sondern in der Beschreibung der Syntax von zusammengehörenden Gruppen von Repräsentationen. Wie ist beispielsweise die Syntax aller natürlichen Zahlen definiert?

Umgangssprachliche Beschreibung

Wir wollen eine Syntax für die Repräsentation von Daten (hier: =Mehrzahl von Datum) festlegen. Der Einfachheit halber verzichten wir auf eine Berücksichtigung von Schaltjahren. Dies hat zur Folge, dass der Februar auch in Nicht-Schaltjahren 29 Tage haben darf. Zur Modellierung der Regeln dienen folgende umgangssprachliche Sätze:

1. Ein Datum besteht aus einer Zahl gefolgt von einem Punkt ␣.␣, einem Leerschritt ␣ und einer Zeichenfolge, die den Monatsnamen repräsentiert. Danach folgt ein weiterer Leerschritt ␣ und die vierstellige Jahreszahl.
2. Ist der Monatsname aus der Menge {Januar, März, Mai, Juli, August, Oktober, Dezember}, dann darf die Zahl vor dem Monatsnamen zwischen 1 und 31 liegen.
3. Ist der Monatsname aus der Menge {April, Juni, September, November}, dann darf die Zahl vor dem Monatsnamen zwischen 1 und 30 liegen.
4. Ist der Monatsname Februar, dann darf die Zahl vor dem Monatsnamen zwischen 1 und 29 liegen.

Diese Regeln müssen interpretiert werden. Eignet sich umgangssprachliche Beschreibung der Syntax, um Interpretationsfehler zu minimieren?

Formales Modell

Ein formales Modell zur Beschreibung der syntaktischen Struktur bestehen aus definierten Elementen, deren Semantik festgelegt ist. Wir schaffen somit ein zwei-stufiges Vorgehen:

Stufe 1: Wir legen eine Repräsentation zur Beschreibung der Syntax von Repräsentationen fest. Für diese Repräsentation definieren wir eine Interpretationsfunktion, die die Syntax und die Semantik festlegt. Stufe 1 stellt somit ein Werkzeug zur Beschreibung von syntaktischen Strukturen dar.

Stufe 2: Wir benutzen das Werkzeug aus Stufe 1, um syntaktische Strukturen zu beschreiben. Mit Hilfe der Interpretationsfunktion kann einfach die Beschreibung der syntaktischen Struktur „verstanden“ werden.

Syntaxdiagramme

Syntaxdiagramme stellen ein formales Modell zur Beschreibung der syntaktischen Struktur dar. Syntaxdiagramme besitzen mindestens

- einen Namen,
- einen Startpunkt und
- einen Endpunkt.

Weiterhin kann ein Syntaxdiagramm

- Schleifen,
- Alternativen und
- Ausgaben

enthalten.

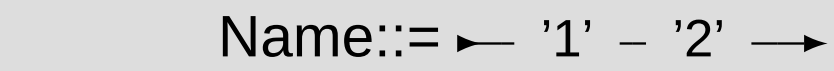
Syntaxdiagramme: Gerüst

Name::= 

Das minimale Syntaxdiagramm besteht aus einem Namen, einem Startpunkt und einem Endpunkt. Die „Bearbeitung“, also der Kontrollfluss, des Diagramms erfolgt indem ein Pfad vom Startpunkt zum Endpunkt durchlaufen wird. Die Menge der Ausgaben aller möglichen Pfade entspricht der Menge der von dem Diagramm erzeugbaren Repräsentationen. Dieses Diagramm besitzt lediglich einen Pfad, der keine Ausgaben erzeugt. Somit kann mit Hilfe dieses Diagramms lediglich die leere Repräsentation (auch leeres Wort ε genannt) erzeugt werden.

Syntaxdiagramme: Ausgaben

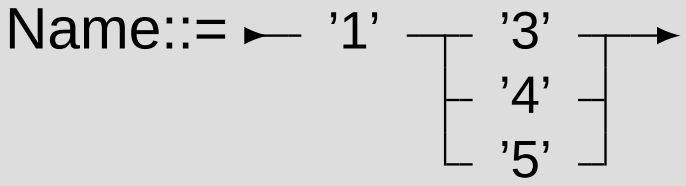
Innerhalb von Syntaxdiagrammen können Ausgaben erzeugt werden, Dies geschieht, indem Zeichenketten in das Syntaxdiagramm eingefügt werden. Erreicht der Kontrollfluss eine Zeichenkette, so wird diese ausgegeben. Die Ausgabe von Syntaxdiagramme beschränkt sich auf Zeichenketten, somit können nur eine Teilmenge alle möglichen Repräsentationen mit Hilfe von Syntaxdiagrammen beschrieben werden. Repräsentationen wie Töne oder ein gesprochenes Wort sind somit nicht modellierbar.



Dieses Syntaxdiagramm erzeugt die mit Hilfe von zwei Ausgaben (eine Ausgabe wäre auch möglich gewesen!) die Repräsentation 12.

Syntaxdiagramme: Alternativen

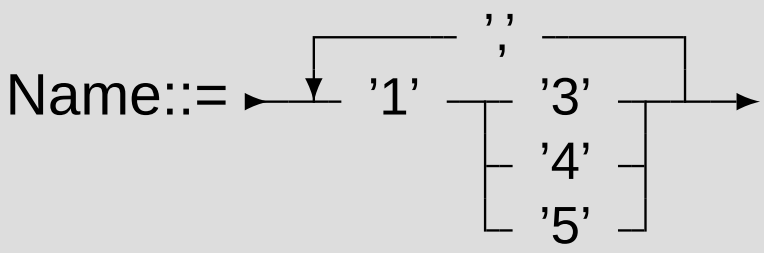
In Syntaxdiagramme können auch mehrere Pfade eingefügt werden. So können Alternativen realisiert werden. Alternative Pfade müssen keine Ausgaben erzeugen.



Dieses Syntaxdiagramm besitzt drei Pfade und erzeugt die Zeichenketten $13, 14, 15$.

Syntaxdiagramme: Schleifen

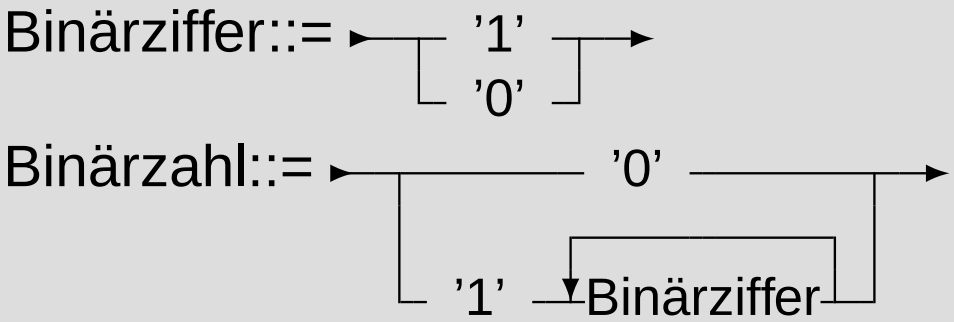
Bisher verlief der Kontrollfluss immer von links nach rechts. Durch die Einführung von Rückkopplungen können Schleifen realisiert werden. Diese Schleifen haben kein an die Schleife geknüpftes Prädikat, das bedeutet, durch eine Rückkopplung entstehen unendlich viele Pfade. In Rückkopplungen können auch Ausgaben erfolgen.



Es sind beliebige Folgen von $13, 14, 15$ erzeugbar. Besteht die Folge aus mehr als einem Element, so sind die Elemente durch ein Komma getrennt. Ein Pfad hat beispielsweise die Ausgabe 13, ein anderer die Ausgabe $15, 14$.

Syntaxdiagramme: Unterdiagramme

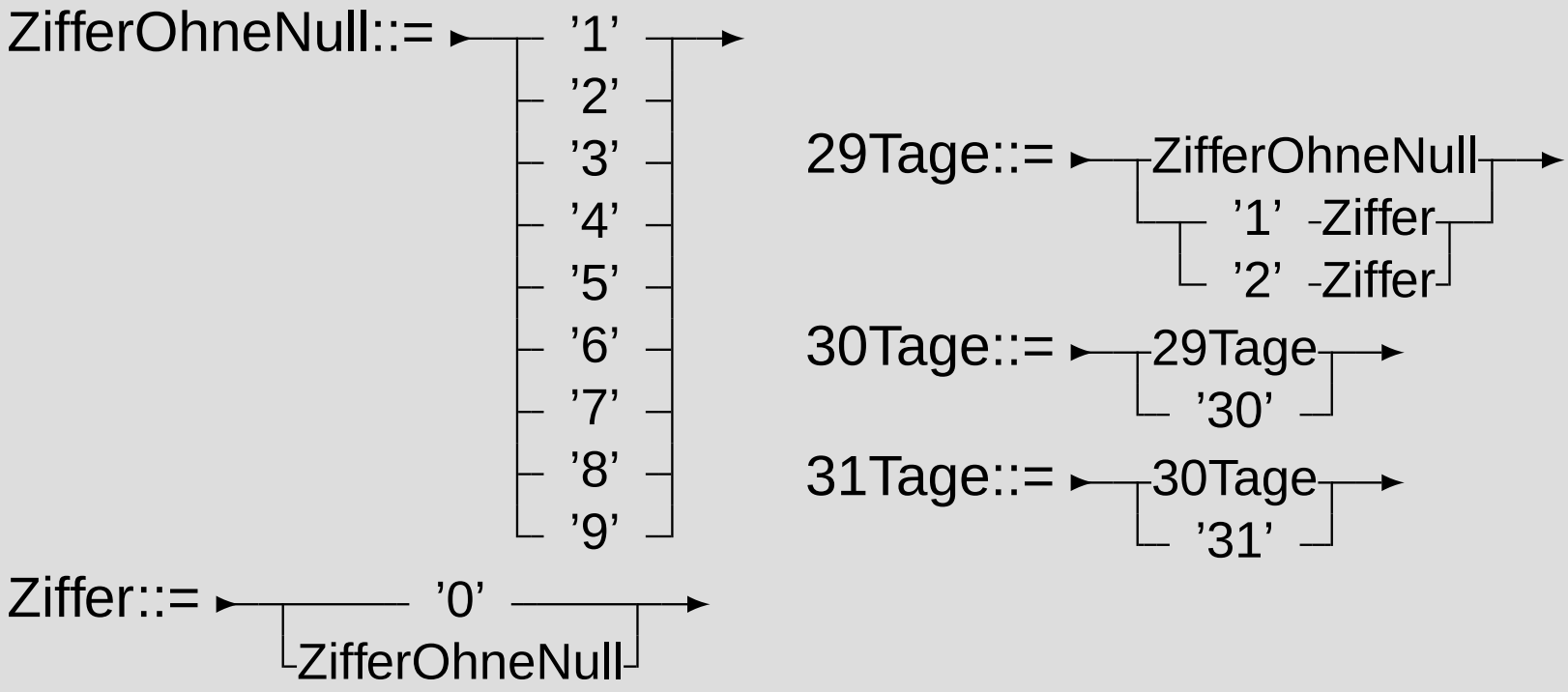
Zur Steigerung der Übersichtlichkeit können in Syntaxdiagrammen andere Syntaxdiagramme eingebunden werden. Dies geschieht mit Hilfe des Namens der Syntaxdiagramme. Zur Unterscheidung zwischen einem Namen und einer Ausgabe sollte die Ausgabe mit Hilfe von Anführungszeichen als Ausgabe gekennzeichnet sein.



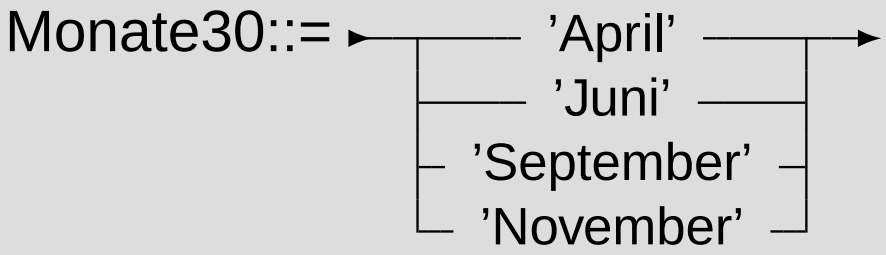
Das Syntaxdiagramm `Binärzahl` stellt Pfade zum Erzeugen aller möglichen Binärzahlen zur Verfügung. Dabei sollen keine fürenden Nullen ausgegeben werden. Mit Ausnahme der Binärzahl '0' fangen also alle Binärzahlen mit einer '1' an.

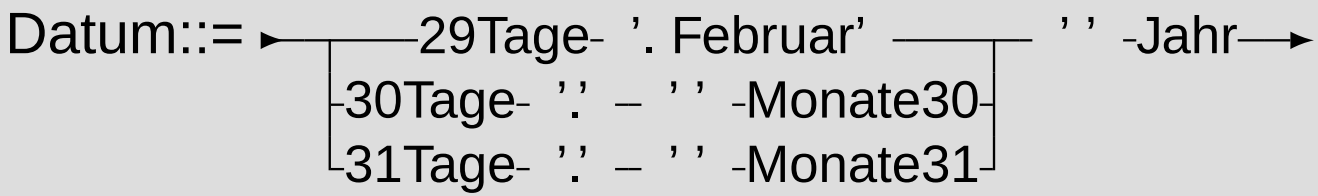
Syntaxdiagramme: Beispiel

Wir konstruieren nun ein Syntaxdiagramm für Daten:



Syntaxdiagramme: Beispiel





Gibt es einen Pfad für die Ausgabe „31. Januar 2001“?

Syntaxdiagramme: Zusammenfassung

+ Syntaxdiagramme stellen eine formale Methode zur Beschreibung von Mengen von Repräsentationen dar.
+ Syntaxdiagramme haben eine festgelegte Interpretationsfunktion.
+ Syntaxdiagramme visualisieren die Syntax, machen sie verständlich.
- Es ist schwer, Syntaxdiagramme mit Hilfe von Computern weiterzuverarbeiten (automatische Syntaxtests). → Notwendigkeit einer anderen formalen Beschreibung der Syntaxregeln.

Grammatiken (1)

Grammatiken sind eine festgelegte formale Beschreibung von Syntaxregeln. Die Idee von Grammatiken ist es, ein System zu erschaffen, mit dem alle Wörter einer Sprache erzeugt werden können. Hierbei sind Grammatiken ähnlich den Syntaxdiagrammen, bei denen durch Durchlaufen aller Pfade alle Wörter erzeugt werden. Eine Grammatik G ist ein 4-Tupel (N, T, P, S). Die Komponenten dieses Tupels sind:

N bezeichnet die Menge der Nichtterminalsymbole. Nichtterminale sind vergleichbar mit den Namen der Syntaxdiagramme. Nichtterminale können weiter ersetzt werden. Ein Wort darf keine Nichtterminale besitzen.

T bezeichnet die Menge der Terminalsymbole. Terminale sind Symbole unseres Alphabets.

Grammatiken (2)

P bezeichnet die Menge der Produktionsregeln. Produktionsregeln sind Regeln zum Ersetzen von Terminal– und Nichtterminalsymbolen durch andere Terminal– und Nichtterminalsymbole.

S bezeichnet das Nichtterminal, mit dem die Ersetzung immer anfängt.

Der Sprachwissenschafter Chomsky hat Grammatiken wissenschaftlich untersucht und sie in verschiedene Klassen eingeteilt. Wir betrachten hier nicht alle Klassen, sondern nur sogenannte Typ-2-Grammatiken. Typ-2-Grammatiken stellen besondere Anforderungen an den Aufbau der Produktionen, was dazu führt, dass es Sprachen gibt, die nicht durch Typ-2-Grammatiken erzeugt werden können. Für unsere Zwecke sind diese Grammatiken jedoch ausreichend.

Grammatiken: Beispiel

Sei $G = (N, T, P, S)$ eine Grammatik mit:

$$\begin{aligned} N &= \{A\} \\ T &= \{a, b\} \\ P &= \{A \to aAa, A \to bAb, A \to \epsilon\} \\ S &= \{A\} \end{aligned}$$

Da wir uns auf Typ-2-Grammatiken beschränkt haben, darf in der Produktionenmenge P nicht jede Ersetzung enthalten sein. Erlaubte Produktionen ersetzen genau ein Nichtterminal durch eine beliebige Kombination von Terminalen und Nichtterminalen. Es ist ebenfalls erlaubt Nichtterminale auf das leere Wort abzubilden.

Grammatiken: Alternativen

Mit Hilfe des Symbols | können Alternativen in der Menge der Produktionen beschrieben werden. Die Menge P lässt sich mit Alternativen folgendermaßen beschreiben:

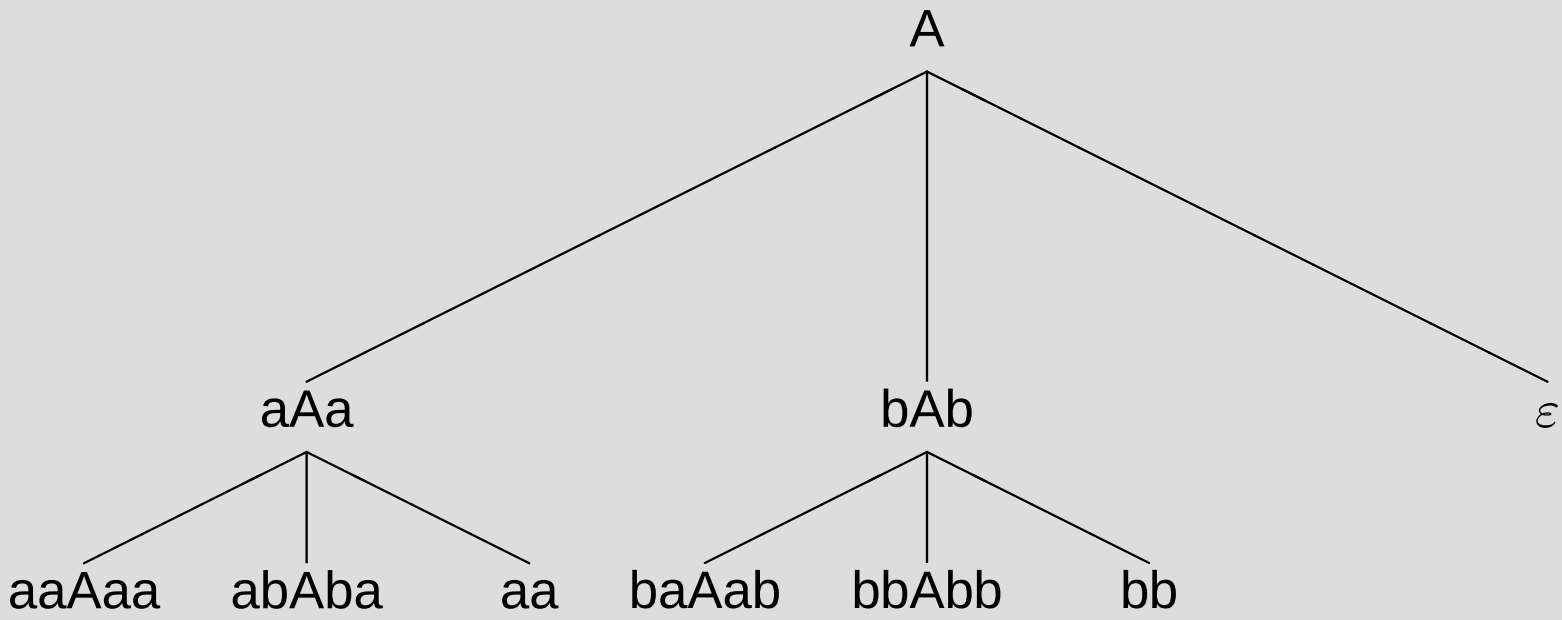
$$P \quad = \quad \{A \rightarrow aAa|bAb|\epsilon\}$$

Wichtig ist das Verständnis, das durch die Verwendung der Alternativen sich nicht die Menge der Produktionen veringert. Alternativen stellen lediglich eine Erleichterung beim Niederschreiben dar. Die Anzahl der Produktionen in P bleibt weiterhin 3.

Das Symbol $\rightarrow$ wird häufig durch die Zeichenfolge `::=` umschrieben. Dies kann sinnvoll sein, wenn das Symbol $\rightarrow$ nicht im Zeichenvorrat verfügbar ist.

Grammatiken: Syntaxbäume

Um festzustellen, welche Sprache L von einer Grammatik G erzeugt wird, erstellt man einen Syntaxbaum (auch Ableitungsbaum genannt). Die Wurzel des Baumes ist immer das Startnichtterminal der Grammatik. Bei unendlichen Sprachen sind die Syntaxbäume ebenfalls unendlich groß. Der folgende Syntaxbaum zeigt die ersten zwei Ableitungen für die Grammatik G:



Grammatik für Datumsangabe

$$
\begin{aligned}
N &= \{ \quad \text{ZifferOhneNull}, \text{Ziffer}, \text{Monate29}, \text{30Tage}, \text{Monate31}, \\
&\quad\ \ \text{Monate30}, \text{Monate31}, \text{Jahr}, \text{JahrRest}, \text{Datum} \ \} \\
T &= \{ \quad 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, \text{Januar}, \text{Februar}, \text{März}, \text{April}, \text{Mai} \\
&\quad\ \ \text{Juni}, \text{Juli}, \text{August}, \text{September}, \\
&\quad\ \ \text{Oktober}, \text{November}, \text{Dezember}, ., , \ \} \\
S &= \{ \quad \text{Datum}\}
\end{aligned}
$$

Grammatik für Datumsangabe

$P = \{$ ZifferOhneNull $\rightarrow' 1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'$,

Ziffer $\rightarrow$ ZifferOhneNull$|'0'$, 29Tage $\rightarrow$ ZifferOhneNull$|'1'$Ziffer$|'2'$Ziffer,

30Tage $\rightarrow$ 29Tage$|'30'$, 31Tage $\rightarrow$ 30Tage$|'31'$,

Monate31 $\rightarrow$ 'Januar'|'März'|'Mai'|'Juli'|'August',

Monate31 $\rightarrow$ 'Oktober'|'Dezember',

Monate30 $\rightarrow$ 'April'|'Juni'|'September'|'November',

Jahr $\rightarrow$ ZifferOhneNull|JahrRest,

JahrRest $\rightarrow$ Ziffer JahrRest$|\varepsilon$,

Datum $\rightarrow$ 29Tage '. ''Februar' ' 'Jahr,

Datum $\rightarrow$ 30Tage '. 'Monate30 ' 'Jahr,

Datum $\rightarrow$ 31Tage '. 'Monate31 ' 'Jahr $\}$

Ableiten von Wörtern

Ein Wort w heißt syntaktisch korrekt bezüglich der Grammatik G, wenn es eine Folge von Ableitungen gibt, die das Wort w erzeugen. Ein Wort w heißt syntaktisch falsch bezüglich der Grammatik G, wenn keine Folge von Ableitungen für w existiert.

Ist das Wort ’12. Februar 2001’ syntaktisch korrekt bezüglich der Grammatik G?

Ableiten von Wörtern

1.	29Tage. Februar Jahr	Ersetze 29Tage
2.	1Ziffer. Februar Jahr	Ersetze Ziffer
3.	12. Februar Jahr	Ersetze Jahr
4.	12. Februar ZifferOhneNullJahrRest	Ersetze ZifferOhneNull
5.	12. Februar 2JahrRest	Ersetze JahrRest
6.	12. Februar 2ZifferJahrRest	Ersetze Ziffer
7.	12. Februar 20JahrRest	Ersetze JahrRest
8.	12. Februar 20ZifferJahrRest	Ersetze Ziffer
9.	12. Februar 200JahrRest	Ersetze JahrRest
10.	12. Februar 200ZifferJahrRest	Ersetze Ziffer
11.	12. Februar 2001JahrRest	Ersetze JahrRest
12.	12. Februar 2001	Fertig

Grammatiken und Programmiersprachen

- Jede Programmiersprache besitzt eine Grammatik.
- Mit der Grammatik kann die Syntax von Programmen geprüft werden.
- Mit Hilfe der Grammatik kann eine Programmiersprache „syntaktisch“ verstanden werden.

Algorithmen

Ein **Algorithmus** ist eine genau definierte Berechnungsvorschrift zur Lösung eines Problems oder einer bestimmten Art von Problemen. Ein Algorithmus wird durch eine endliche Menge von Regeln definiert, die nacheinander angewendet und oft nach bestimmten Bedingungen wiederholt werden. Ein Algorithmus soll auch terminieren, d.h. nach endlich vielen Schritten wird mit einem Ergebnis gestoppt.

Algorithmen in der realen Welt

In der realen Welt gibt es viele bekannte Beispiele für Algorithmen:

- Kochrezept
- Reparatur- und Bedienungsanleitungen
- Waschmaschinenprogramme
- . . .

Eigenschaften von Algorithmen (1)

1. Das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein (statische Finitheit).
2. Jeder Schritt des Verfahrens muss auch tatsächlich ausführbar sein (Ausführbarkeit).
3. Das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen (dynamische Finitheit, siehe Platzkomplexität).
4. Das Verfahren darf nur endlich viele Schritte benötigen (Terminierung, siehe auch Zeitkomplexität).
5. Der Algorithmus muss bei denselben Voraussetzungen das gleiche Ergebnis liefern.
6. Die nächste anzuwendende Regel im Verfahren ist zu jedem Zeitpunkt eindeutig definiert.

Eigenschaften von Algorithmen (2)

Statische Finitheit: Die Beschreibung eines Algorithmus darf nicht unendlich groß sein. Als statische Finitheit wird die Endlichkeit des Quelltextes bezeichnet. Der Quelltext darf nur eine begrenzte Anzahl, wenn auch bei Bedarf sehr viele Regeln enthalten.

Dynamische Finitheit: Zu jedem Zeitpunkt der Ausführung darf der von einem Algorithmus benötigte Speicherbedarf nur endlich groß sein. Andernfalls wäre der Algorithmus nicht ausführbar. Dies wird als dynamische Finitheit bezeichnet.

Terminierung: Algorithmen sind terminierend, wenn sie für jede mögliche Eingabe nach einer endlichen Zahl von Schritten zu einem Ergebnis kommen. Die tatsächliche Zahl der Schritte kann dabei beliebig groß sein. Steuerungssysteme und Betriebssysteme und auch viele Programme, die auf Interaktion mit dem Benutzer aufbauen, erfüllen diese Eigenschaft nicht

Beschreibung von Algorithmen

Algorithmen können auf unterschiedliche Art und Weise beschrieben werden. Gebräuchliche Beschreibungen sind:

- natürlichsprachliche Sätze (evtl. nicht formal genug, unterschiedliche Interpretationsfunktionen)
- Struktogramme (ältere Methode)
- Flussdiagramme (ältere Methode)
- UML (Klassendiagramme, Aktivitätsdiagramme $\leftrightarrow$ s. Flussdiagramme)
- Pseudo-Quellcode

Natürlichsprachliche Beschreibung

Der Euklidische Algorithmus, der bereits um 300 v. Chr. beschrieben wurde, dient zur Ermittlung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen A und B:

1. Sei A die Größere der beiden Zahlen A und B (entsprechend vertauschen, falls dies nicht bereits so ist)
2. Setze A auf den Wert $A - B$
3. Wenn A und B ungleich sind, dann fahre fort mit Schritt 1, wenn sie gleich sind, dann beende den Algorithmus: Diese Zahl ist der größte gemeinsame Teiler.

Beispiel Euklidische Algorithmus

Berechne den größten gemeinsamen Teiler von 14 und 8!

	A	B	$A - B$
1.	14	8	6
2.	8	6	2
3.	6	2	4
4.	4	2	2
5.	2	2	0

Das Ergebnis ist 2

Pseudo-Quellcode

Eine formalere Beschreibung als die Natürlichsprachliche ist eine Beschreibung des Algorithmus mit Hilfe von Pseudo-Quellcode. Pseudo-Quellcode ist dabei von der Syntax an existierende Programmiersprachen angelehnt. Zur formalen Definition unseres Pseudo-Quellcode müssen wir nun folgende Schritte durchführen:

- Festlegen der Syntax.
- Festlegung der Interpretationsfunktion.

Grundgerüst (1)

Jede Algorithmusbeschreibung $\mathcal{B}$ in Pseudo-Quellcode ist ein 3-Tupel $\mathcal{B} = (E, V, A)$, wobei die einzelnen Komponenten folgendermaßen definiert sind:

- $\mathcal{B}$ beinhaltet die komplette Algorithmusbeschreibung. $\mathcal{B}$ ist ebenfalls der Name der Beschreibung.
- E beschreibt die Menge der Eingabevariablen
- A beschreibt die Menge der Ergebnisvariablen (Ausgabevariablen)
- V enthält ein n-Tupel mit n Verarbeitungsschritten

Wir werden diese einzelnen Bestandteile (Name, Eingabevariablen, Ergebnisvariablen und Verarbeitungsschritte) zur besseren Übersicht **nicht** in Tupelschreibweise sondern tabellarisch darstellen.

Grundgerüst (2)

Name: Hier steht der Name des Algorithmus.

Eingabe: Hier sind alle Eingabevariablen aufgeführt. Zusätzlich können diese natürlichsprachlich erklärt und beschrieben werden.

Ausgabe: Hier sind alle Ergebnisvariablen aufgeführt. Zusätzlich können diese natürlichsprachlich erklärt und beschrieben werden.

Vorgehen: Hier steht der eigentliche Pseudo-Quellcode.

Pseudo-Quellcode: Variablen

Eine **Variable** ist eine Größe, die verschiedene Werte annehmen kann. Sie ist also in ihrer Größe **veränderlich** oder **variabel**. Variablen werden auch **Platzhalter** genannt. Der Vorrat an verschiedenen Variablen ist unendlich groß, in einem Algorithmus dürfen allerdings nur endlich viele verschiedene Variablen verwendet werden. Variablen werden mit einem Bezeichner gekennzeichnet. Dieser Bezeichner identifiziert jede Variable eindeutig. Der Bezeichner muss mit einem Buchstaben anfangen. Beispiele für Bezeichner sind:

`a`, `b`, `variable1`, `t_i`, `h_5`

Variablenzuweisung

Eine **Variable** ist eine Größe, die verschiedene Werte annehmen kann. Wie werden Werte an Variablen gebunden? Die Bindung geschieht mit Hilfe der Zuweisung $:=$. Ein Beispiel für eine Zuweisung eines Wertes an die Variable A ist:

$$\underbrace{A}_{\text{lhs}} := \underbrace{\sum_{i=0}^{6} i + B}_{\text{rhs}}$$

Bei der Zuweisung unterscheidet man zwischen linker Seite (**l**eft **h**and **s**ide) und rechter Seite (**r**ight **h**and **s**ide). Auf der linken Seite dürfen nur Variablen, auf der rechten Seite beliebige auswertbare Ausdrücke vorkommen.

parallele Variablenzuweisung

Mit Hilfe des Operators **,** können auf der linken Seite einer Zuweisung mehrere Variablen vorkommen.

$$A, B := B, A$$

Der Operator **,** realisiert hier eine **parallele Zuweisung**, da die einzelnen Zuweisungen $A := B$ und $B := A$ aus unserem obigen Beispiel parallel ausgeführt werden.

Allerdings unterstützen nur sehr wenige Programmiersprachen (z.B. PERL) tatsächlich eine solche parallele Zuweisung.

Hintereinanderausführung

Mit Hilfe des Semikolons ; wird eine Anweisungen abgeschlossen. Es kann benutzt werden, um mehrere Anweisungenen hintereinander auszuführen. Das Symbol ; trennt somit die einzelnen Anweisungsschritte.

```
BEGIN
   A := 2;
   B := 5;
   A,B := B,A;
END;
```

`BEGIN` und `END` fassen in Pseudocode Anweisungsschritte zu einem logischen Anweisungsschritt zusammen (vergl. auch die Programmiersprache PASCAL).

Bedingte Ausführung

```
IF (Prädikat) THEN
  Anweisung1;
ELSE
  Anweisung2;
```

`Anweisung1` wird genau dann ausgeführt, wenn das Prädikat erfüllt ist. Ist das Prädikat nicht erfüllt, wird `Anweisung2` ausgeführt.

`BEGIN` und `END` werden verwendet, wenn mehrere Anweisungen unter der jeweiligen Bedingung ausgeführt werden sollen.

Wiederholte Ausführung

Für die wiederholte Ausführung von Anweisungen stehen in unserem Pseudo-Quellcode drei Konstrukte zur Verfügung:

- Führe die Anweisung n mal aus.
- *Solange* ein Prädikat erfüllt ist, führe die Anweisung aus.
- Führe die Anweisung solange aus, *bis* ein Prädikat erfüllt ist.

Wiederholte Ausführung (2)

Betrachten wir zuerst eine n-malige Ausführung einer Anweisung:

```
FOR index := start TO ende DO
  Anweisung;
```

Der Index wird von seinem Startwert solange verändert bis er den Endwert erreicht hat. Nach jeder Veränderung (und nur dann!) wird die Anweisung ausgeführt. Beispiele:

```
sum1 := 0;
FOR i := 1 TO 100 DO
  sum1 := sum1 + i;
sum2 := 0;
FOR i := 100 DOWNTO 1 DO
  sum2 := sum2 + i;
FOR i := 100 DOWNTO 200 DO
  sum2 := sum2 + i;
```

Wiederholte Ausführung (3)

Die sogenannte `FOR`-Schleife eignet sich nicht, wenn die Anzahl der Durchläufe von der Ausführung der Schleife abhängig ist, also noch nicht zu Beginn der Schleife feststeht. Um dennoch solche Schleifen realisieren zu können führen wir folgende Konstrukte ein:

```
WHILE Prädikat DO
  Anweisung;

REPEAT
  Anweisung 1;
  ...
  Anweisung n
UNTIL Prädikat;
```

Beispiel Pseudo-Quellcode

Name: Fibonacci

Eingabe: Es wird eine positive natürliche Zahl $n > 0$ als Eingabe verwendet.

Ausgabe: Die Ausgabe ist die positive natürliche Zahl f.

Beispiel Pseudo-Quellcode

Vorgehen:

```
fibminus1 := 1;
fibminus2 := 1;
f := 1;
IF n > 2 THEN
  FOR i := 3 TO n DO
    BEGIN
      f := fibminus1 + fibminus2;
      fibminus2 := fibminus1;
      fibminus1 := f;
    END;
```

Ausführungstabellen

Step	Aktuelle Anweisung	Folgende Anweisung	n	f	fibminus1	fibminus2	i
1.	fibminus1 := 1	fibminus2 := 1	3	?	1	?	?
2.	fibminus2 := 1	f := 1	3	?	1	1	?
3.	f := 1	IF $n > 2$ THEN	3	1	1	1	?
4.	IF $n > 2$ THEN	FOR i := 3 TO n DO	3	1	1	1	?
5.	FOR i := 3 TO n DO	f := fibminus1 + fibminus2	3	1	1	1	3
6.	f := fibminus1 + fibminus2	fibminus2 := fibminus1	3	2	1	1	3
7.	fibminus2 := fibminus1	fibminus1 := f	3	2	1	1	3
8.	fibminus1 := f	FOR i := 3 TO n DO	3	2	2	1	3
9.	FOR i := 3 TO n DO	Ende	3	2	2	1	3

Formale Eigenschaften von Algorithmen

- Korrektheit von Algorithmen
  - Man kann die Korrektheit von Algorithmen im allgemeinen nicht durch Testen an ausgewählten Beispielen nachweisen, denn mit **Testen kann lediglich die Anwesenheit, nicht jedoch die Abwesenheit von Fehlern nachgewiesen werden.**
  - Nachweis der Korrektheit geht nur über in der Regel sehr aufwendige und komplexe Korrektheitsbeweise. Diese Beweise sind **nicht** Inhalt dieser Vorlesung.
- Effizienz von Algorithmen
  - Speicherplatzverbrauch
  - Rechenzeit

Effizienz von Algorithmen

Wie lässt sich die Effizienz von Algorithmen messen?

- Nach der Implementierung des Algorithmus kann die Rechenzeit und der Speicherplatzverbrauch in Abhängigkeit der Eingabewerte gemessen werden (Benchmarking).
- Vor der Implementierung des Algorithmus kann aber bereits sein Laufzeitverhalten mathematisch bestimmt werden.

Motivation

Eingabe: Eine positive natürliche Zahl n.

Ausgabe: Die Summe s aller natürlichen Zahlen zwischen 0 und n.

Vorgehen:

```
S := 0;
FOR i := 0 TO n DO
  S := S + i;
```

Wie lange würde die Laufzeit des Algorithmus dauern, wenn jede Addition 5 ms benötigt, alle anderen Anweisungen 0 ms und $n = 100000$ ist?

Antwort: $100001 \cdot 5\text{ms} = 500005\text{ms} = 500,005\text{s}$

Laufzeitkomplexität (1)

Idee: Bestimmen der Anzahl der auszuführenden Anweisungen in Abhängigkeit der Eingabevariablen eines Algorithmus.

Problem: wie ermittelt man die Anzahl der auszuführenden Anweisungen?

Ziel: Aus jedem Algorithmus $\mathcal{A}$ wird eine Funktion f abgeleitet, die in Abhängigkeit der Eingabevariablen die Anzahl der Anweisungen liefert.

Laufzeitkomplexität (2)

Eingabe: Zwei positive natürliche Zahlen a, b.

Ausgabe: Eine positive natürliche Zahl s (die die Summe der Eingaben ist, siehe Vorgehen).

Vorgehen:

```
S := 0;
S := A + B;
```

Fazit: Bei diesem Algorithmus werden für alle möglichen Eingaben immer 2 Anweisungen ausgeführt.

Laufzeitkomplexität (3)

Eingabe: Eine positive natürliche Zahl n.

Ausgabe: Eine positive natürliche Zahl s.

Vorgehen:

```
s := 0;
FOR i := 1 TO n DO
  FOR j := i TO n DO
    s := s + 1;
```

Dies entspricht:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} 1 \quad = \quad ?$$

Einwurf: Rechnen mit Summen

Beim Rechnen mit Summen führen wir folgende Regeln ein:

$$\sum_{i=1}^{n} c = n \cdot c$$

$$\sum_{i=0}^{n} c = \sum_{i=1}^{n+1} c$$

$$\sum_{i=a}^{n} c \text{ mit } a < n = \left(\sum_{i=1}^{n} c\right) - \left(\sum_{i=1}^{a-1} c\right)$$

$$\sum_{i=a}^{n} c \text{ mit } a > n = 0$$

Einwurf: Rechnen mit Summen

$$\sum_{i=1}^{n} c \cdot a_i = c \cdot \sum_{i=1}^{n} a_i$$

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2 \cdot n + 1)}{6}$$

Laufzeitkomplexität (3)

$$
\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} 1 &= \sum_{i=1}^{n}\left(\sum_{j=1}^{n} 1 - \sum_{j=1}^{i-1} 1\right) \\
&= \sum_{i=1}^{n}(n-(i-1)) \\
&= \sum_{i=1}^{n}(n-i+1) \\
&= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1
\end{aligned}
$$

Laufzeitkomplexität (4)

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i}^{n} 1 &= n^2 - \frac{n\cdot(n+1)}{2} + n \\ &= (n^2+n) - \frac{n^2+n}{2} \\ &= \frac{n^2+n}{2}\end{aligned}$$

Für ein gegebenes n kann nun die Anzahl der benötigten Anweisungen berechnet werden:

$n = 1$: $\frac{1^2+1}{2} = 1$

$n = 10$: $\frac{10^2+10}{2} = 55$

Komplexitätsklassen

Um Algorithmen besser vergleichen zu können werden sie in Komplexitätsklassen zusammengefasst. Eine **Komplexitätsklasse** ist eine Kategorie von Algorithmen, zusammengefasst nach einem gemeinsamen Maß der Komplexität. Sie ist definiert durch das asymptotische Verhalten der Obergrenze (oder des Mittelwertes oder der Untergrenze) des Resourcenbedarfs (insbesondere an Laufzeit und Speicherplatz) in Abhängigkeit von einer Problemgröße n (Länge der Eingabe).
Eine Laufzeitfunktion g besitzt die Größenordnung f, wenn g Element der Komplexitätsklasse $O(f)$ ist. $O(f)$ ist folgendermaßen definiert:

$$O(f) = \{g | \exists c_1 > 0 : \exists c_2 > 0 : \forall n \in \mathcal{Z}^+ : g(n) \leq c_1 \cdot f(n) + c_2\}$$

Komplexitätsklassen (2)

Mit Hilfe der Definition von O lassen sich folgende Größenordnungen festlegen:

- logarithmisches Wachstum: $\log N$
- lineares Wachstum: N
- quadratisches, kubisches, ... Wachstum: $N^2, N^3, \ldots$
- n-log n-Wachstum: $N \cdot \log N$
- exponentielles Wachstum: $2^N, 3^N, \ldots$

Alle Algorithmen einer Komplexitätsklasse besitzen für große N das gleiche Laufzeitverhalten.

Komplexitätsklassen (3)

In welcher Komplexitätsklasse liegt unsere Laufzeitfunktion

$$g(n) = \frac{n^2 + n}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Die Funktion g ist in $O(N^2)$, denn mit $c_1 = 2$ und $c_2 = 1$ gilt für alle nichtnegativen, ganzzahligen N $g(N) \leq c_1 N^2 + c_2$. Somit ist $g(N)$ Element von $O(N^2)$.

Laufzeitkomplexität von `WHILE, IF`

Bislang haben wir die Laufzeitkomplexität nur von „einfachen“ Algorithmen betrachtet. Wir haben noch keine Überlegungen zum Einfluß von `WHILE`- und `IF`-Konstrukte auf das Laufzeitverhalten angestellt.
Die Bestimmung der Laufzeitkomplexität wird (sehr viel) komplizierter, wenn `WHILE`- und `IF`-Konstrukte in den Algorithmen verwendet werden. Oft kann man die Laufzeitkomplexität gar nicht in einer Funktion bestimmen sondern muss den

- best case,
- average case und den
- worst case

gesondert behandeln.

Beispiel

Eingabe: Ein Vektor $(a[1], \ldots a[n])$ aus natürlichen Zahlen $a[i]$.

Ausgabe: Die Summe s aller ungeraden Elemente des Vektors.

Vorgehen:

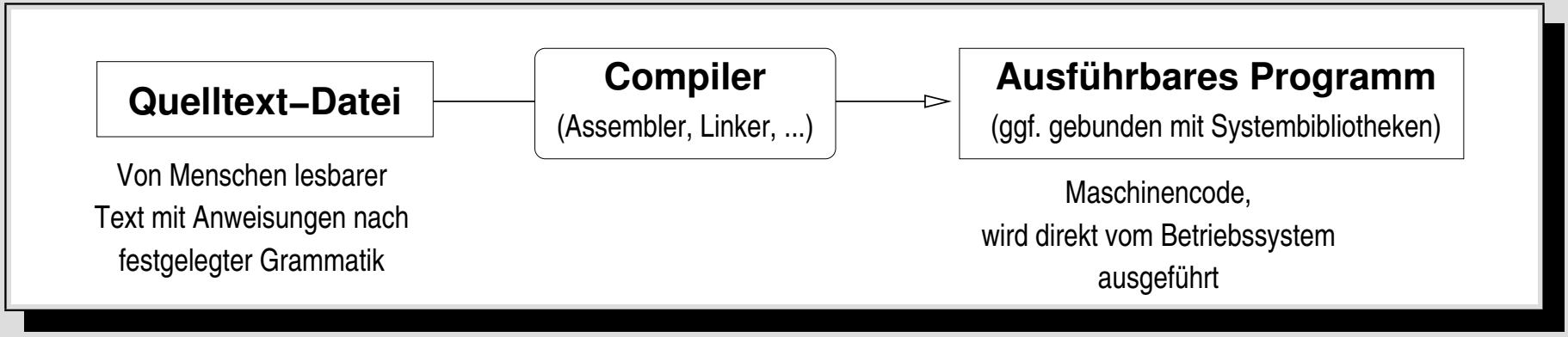
```
FOR i := 1 TO n DO
  IF (odd(a[i])) THEN
    sum := sum + a[i];
```

Komplexitätsverhalten:

- best case (Vektor enthält nur gerade Zahlen): $O(1)$
- avg case (Vektor enthält ungefähr gleichviel gerade wie ungerade Zahlen): $O(n)$
- worst case (Vektor enthält nur ungerade Zahlen): $O(n)$

Compiler

Wird ein Programmtext als Ganzes übersetzt, spricht man in Bezug auf den Übersetzungsmechanismus von einem **Compiler**. Der Compiler ist selbst ein Programm, welches als Dateneingabe den menschenlesbaren Programmtext bekommt und als Datenausgabe den Maschinencode liefert, der direkt vom Prozessor verstanden wird (zum Beispiel Objectcode, EXE-Datei) oder der in einer Laufzeitumgebung in einer „virtuellen Maschine“ (zum Beispiel JVM oder .NET) ausgeführt wird.



Interpreter

Wird ein Programmtext Schritt für Schritt übersetzt und der jeweils übersetzte Schritt sofort ausgeführt, spricht man von einem **Interpreter**. Interpretierte Programme laufen meist langsamer als kompilierte.

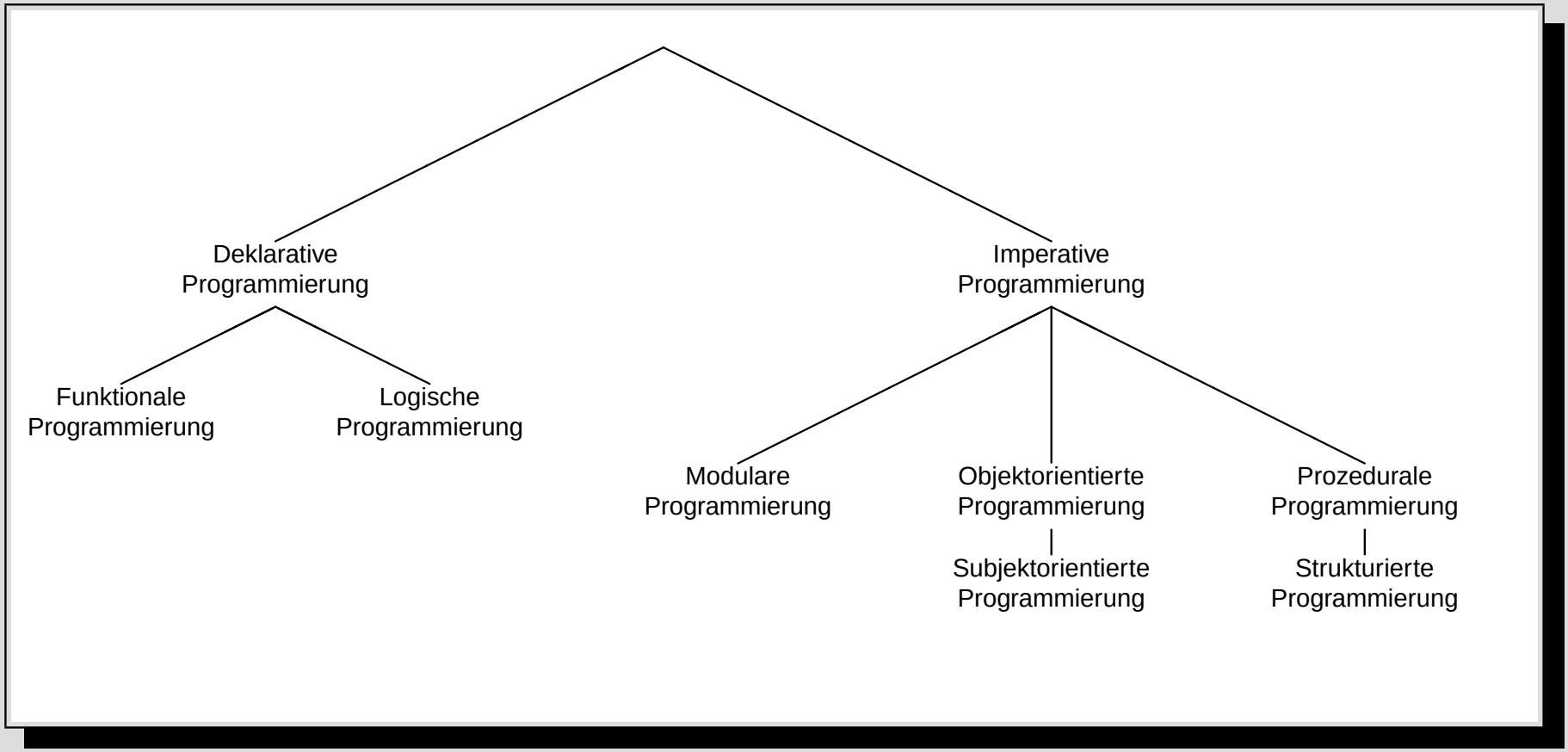
Programm, Programmcode, Quelltext

Eine logische Abfolge von Befehlen in einer Programmiersprache nennt man allgemein **Programm**, **Programmcode** oder **Quelltext**. (Quelltext betont besonders die Lesbarkeit). Dieser wird von Programmierern verfasst.

Programmierparadigma

Ein **Programmierparadigma** ist das einer Programmiersprache oder Programmiertechnik zugrundeliegende Prinzip. Das Wort **Paradigma** kommt aus dem Griechischen und bedeutet Beispiel, Vorbild, Muster oder auch Abgrenzung.

Programierparadigmen



Diese Darstellung ist bei weitem nicht vollständig!

Beispiel

Die Programmiersprache **Java** folgt den Paradigmen der **objektorientierten** und der **attributorientierten** Programmierung, die Programmiersprache **C++** ermöglicht **objektorientierte** Programmierung und **prozedurale** Programmierung, beide Sprachen entsprechen dem Paradigma der **imperativen** Programmierung und ermöglichen **generische** Programmierung.

Programmierparadigmen . . .

- . . . bestimmen den Aufbau des Quelltextes,
- . . . sind die Philosophie des Programmierens,
- . . . müssen vom Programmierer verstanden werden.
- . . . können in Programmiersprachen kombiniert werden.

Frage: Wie charakterisieren sich nun die verschiedenen Programmierparadigmen?

Deklarative Programmierung

Bei der **deklarativen Programmierung** wird nicht beschrieben, wie etwas zu geschehen hat bzw. was zu tun ist, sondern nur welches Ergebnis gewünscht ist. Es sollte also nicht mehr der Lösungsweg programmiert werden, sondern nur noch angegeben werden, wie ein Ergebnis auszusehen hat.

Als Verwirklichungen dieses Grundgedankens sind die **funktionale Programmierung** und die **logische Programmierung** hervorgegangen. Wir betrachten hier lediglich kurz die Prinzipien der funktionalen Programmierung.

Funktionale Programmierung

- Ein funktionales Programm ist eine Abbildung von Eingabedaten auf Ausgabedaten.
- In einem funktionalen Programm wird die *Reihenfolge der Berechnungsschritte* in der Regel *nicht* festgelegt.
- Funktionale Programmiersprachen erlauben es, Funktionen (wie Daten) als Argumente und Rückgabewerte anderer Funktionen zu behandeln. Dadurch ist es einfach, Operatoren auf Funktionen zu definieren. Dies macht funktionale Programme oft kürzer und abstrakter, erfordert jedoch oft eine Umgewöhnungszeit für Programmierer, die den funktionalen Programmierstil nicht gewohnt sind.

strict evaluation vs. lazy evaluation

Funktionale Sprachen kann man auch nach ihrer Auswertungsstrategie unterscheiden: Bei strenger Auswertung (engl. eager bzw. **strict evaluation**) werden Ausdrücke sofort ausgewertet. Dem gegenüber steht die Bedarfsauswertung (engl. **lazy evaluation**), bei der Ausdrücke erst ausgewertet werden, wenn deren Wert in einer Berechnung benötigt wird. Dadurch lassen sich z.B. unendlich große Datenstrukturen (die Liste aller natürlicher Zahlen, die Liste aller Primzahlen, etc.) definieren und bestimmte Algorithmen vereinfachen sich.

Typisierung

Weiterhin kann man funktionale Sprachen einteilen in **dynamisch** und **statisch typisierte Sprachen**, die sich dadurch ergeben, dass die Typprüfung während der Laufzeit bzw. während der Übersetzungszeit stattfinden kann.

Eine Typisierung ist vergleichbar mit der Festlegung von Definitions– und Wertebereich bei mathematischen Funktionen.

Der Verzicht auf zerstörerische Zuweisungen führt dazu, das etliche klassische Algorithmen und Datenstrukturen, die regen Gebrauch von dieser Möglichkeit machen, so nicht für funktionale Sprachen verwendet werden können und man nach neuen Lösungen suchen muss.

„Auch wenn die meisten dieser Bücher [über Datenstrukturen und Algorithmen] behaupten, das sie unabhängig von einer bestimmten Programmiersprache sind, so sind sie leider nur sprachunabhängig im Sinne Henry Fords: Programmierer können jede Programmiersprache benutzen, solange sie imperativ ist.“
– Chris Okasaki

Funktionale Programmiersprachen

- LISP
- Scheme
- OCaml
- Haskell
- Erlang
- XSLT

Haskell

- Haskell ist eine rein funktionale Programmiersprache (erlaubt somit ausschließlich das funktionale Programmierparadigma).
- Haskell unterstützt lazy evaluation.
- Haskell ist ein Interpreter.
- Haskell unterstützt statische Typisierung.

Die folgenden Beispiele wurden mit der Haskell-Implementierung `hugs` erstellt.

Auswerten von Ausdrücken

Das Berechnungsmodell von Haskell ist denkbar einfach: ein gegebener Ausdruck wird zu einem Wert (auch: Normalform) reduziert.

```
Prelude> 5+2+3
10
Prelude> sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3
1
```

Wir sagen: '`sin 0.3 * sin 0.3 + cos 0.3 * cos 0.3`' reduziert zu '1'.

Rekursive Funktionen

Rekursion bedeutet Selbstbezüglichkeit (von lateinisch recurrere = zurücklaufen). Sie tritt immer dann auf, wenn etwas auf sich selbst verweist. Ein rekursives Element muss nicht immer direkt auf sich selbst verweisen (direkte Rekursion), eine Rekursion kann auch über mehrere Zwischenschritte entstehen. Rekursion kann dazu führen, dass merkwürdige Schleifen entstehen. So ist z.B. der Satz „Dieser Satz ist unwahr“ rekursiv, da er von sich selber spricht. Eine etwas subtilere Form der Rekursion (indirekte Rekursion) kann auftreten, wenn zwei Dinge gegenseitig aufeinander verweisen. Ein Beispiel sind die beiden Sätze: „Der folgende Satz ist wahr“ „Der vorhergehende Satz ist nicht wahr“. Die Probleme beim Verständnis von Rekursion beschreibt der Satz: „Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen“.

Beispiel: Kaninchenpopulation (1)

Der Mathematiker Fibonacci stieß bei der einfachen mathematischen Modellierung des Wachstums einer Kaninchenpopulation nach folgender Vorschrift:

1. Zu Beginn gibt es ein Paar neugeborener Kaninchen.
2. Jedes neugeborene Kaninchenpaar wirft nach 2 Monaten ein weiteres Paar.
3. Anschließend wirft jedes Kaninchenpaar jeden Monat ein weiteres.
4. Kaninchen leben ewig und haben einen unbegrenzten Lebensraum.

. . .

Kaninchenpopulation (2)

. . . auf die nach ihm benannte rekursive Fibonacci-Folge:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n+2) &= f(n) + f(n+1) \end{aligned}$$

Kaninchenpopulation (3)

Zur Bestimmung der Elemente der Folge läßt sich nun folgende mathemethische Funktion fib : $IN^+ \rightarrow IN$ ableiten:

$$\text{fib}(n) = \begin{cases} 0 & \text{falls} \quad n = 0 \quad \text{(Rekursionsanfang)} \\ 1 & \text{falls} \quad n = 1 \quad \text{(Rekursionsanfang)} \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{falls} \quad n \geq 2 \quad \text{(Rekursionsschritt)} \end{cases}$$

In Haskell lässt sich die Fibonacci-Folge folgendermaßen berechnen:

```
fib :: Integer -> Integer
fib(0) = 0
fib(1) = 1
fib(n+2) = fib(n) + fib(n+1)
```

Zusammenfassung

Programmieren mit Funktionen bedeutet:

- Einfache Funktionen werden zu komplexen Funktionen zusammengesetzt, diese werden wiederum zu komplexeren Funktionen zusammengesetzt und so weiter.
- Am Schluss existiert eine Funktion, die das Ergebnis liefert.
- Wer mathematische Funktionen beschreiben kann, kann auch mit Funktionen programmieren.
- Quellcode ist sehr übersichtlich und in der Regel sehr kurz.

Fazit funktionale Programmiersprachen

- Funktionale Programmiersprachen haben in der Regel einen hohen Speicherverbrauch beim Interpretieren der Programme. Die Ausführungs-Geschwindigkeit ist in der Regel langsam.
- Programmieren mit Funktionen wird auch tatsächlich hin und wieder in der Praxis eingesetzt, z.B. bei Intel (Mikroprozessor-Verifikation)

Ausblick

Nach dem funktionalen Programmierparadigma wird noch das imperative Programmierparadigma vorgestellt. Imperative Programmiersprachen basieren auf der genauen Festlegung, welcher Befehl oder welche Anweisung wann ausgeführt wird. Wesentlich ist dabei die *Reihenfolge* von Anweisungen, und Kontrollstrukturen, die den Ablauf durch Bedingungen steuern. Hierzu wird die Programmiersprache C eingeführt.

Im weiteren Verlauf der Vorlesung wird auch noch das objektorientierte Programmierparadigma vorgestellt. Dieses Paradigma wird mit Hilfe der Programmiersprache Java praktisch begleitet.

Definition von Objektorientierung

Unter Objektorientierung versteht man ein Paradigma für

- die Analyse
- den Entwurf
- und die Implementierung

von objektorientierten Systemen.

Teilaspekte sind die die objektorientierte Analyse (OOA), Design (OOD) und Programmierung (OOP).

Was ist ein Objekt? (1)

Objekte sind physikalische oder konzeptuelle Dinge. Ein System besteht aus vielen Objekten.

Ein Objekt hat ein definiertes Verhalten.

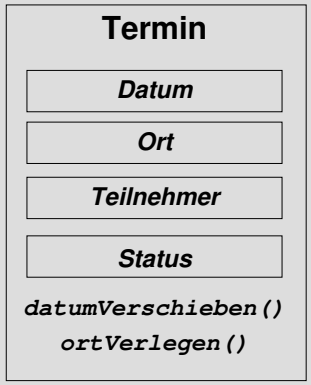
Das Verhalten setzt sich zusammen aus einer Menge genau definierter Operationen zur Erfüllung von Einzelaufgaben. Eine solche Operation wird i.d.R. beim Empfang einer „Nachricht“ ausgeführt.

Was ist ein Objekt? (2)

- Ein Objekt hat einen inneren Zustand.
  Der Zustand des Objekts ist seine „Privatsache“. Das Resultat einer Operation des Objekts (bei Empfang einer Nachricht) hängt jedoch vom aktuellen Zustand des Objektes ab.

- Ein Objekt hat eine eindeutige Identität.
  Die Identität eines Objekts ist unabhängig von seinen anderen Eigenschaften. Es können mehrere verschiedene Objekte mit identischem Verhalten und innerem Zustand im gleichen System existieren.

Beispiel: Termin-Objekt



Objekt „12. Projektbesprechung“ zur Repräsentation eines Termins

Verhalten des Termin-Objekts:
Reaktion auf Nachrichten wie „gibDatum“, „versendeEinladungen“

Zustand des Termin-Objekts (Eigenschaften):
Datum, Ort, Teilnehmer, geplant/bestätigt

Identität des Termin-Objekts:
Die „12. Projektbesprechung“ besteht unabhängig von ihren Eigenschaften wie: “Datum verschieben“, “Ort verlegen“, “Teilnehmer einladen“.

Objekte und Klassen

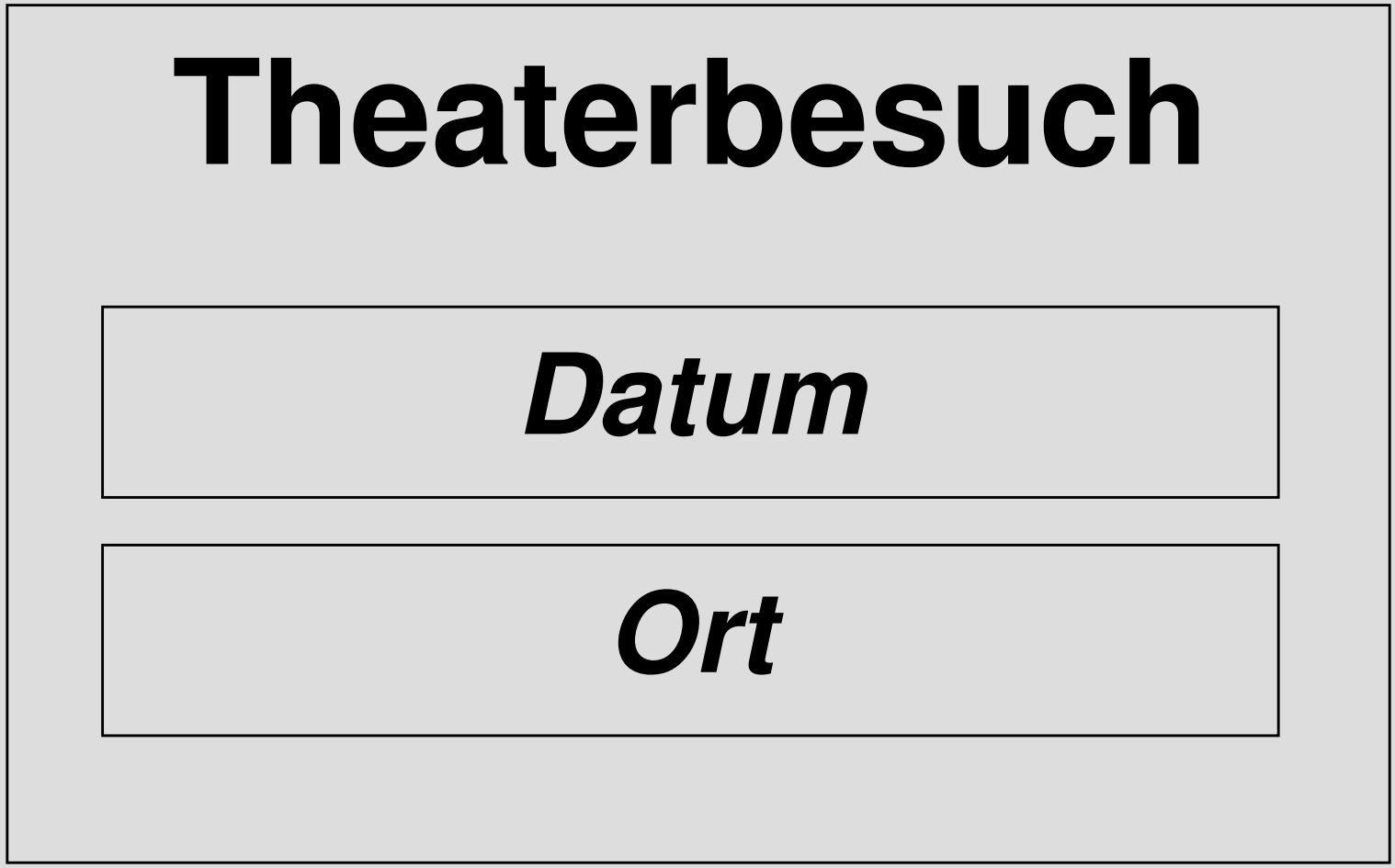
- Jedes *Objekt* gehört zu einer *Klasse*.

  Alle Objekte einer Klasse folgen dem gleichen Verhaltensschema und haben die gleiche innere Struktur. Man sagt: Eine Objekt-Variable ist eine *Instanz* einer Klasse.

- Klassen besitzen einen „Stammbaum“
  in dem Verhaltensschema und innere Struktur durch Vererbung weitergegeben werden. *Vererbung* bedeutet Spezialisierung einer Klasse zu einer Teilklasse.

Eine Nachricht kann verschiedenes Verhalten (verschiedene Operationen) auslösen, je nachdem zu welcher Teilklasse einer Oberklasse das empfangende Objekt gehört (Polymorphie).

Beispiele: Termin-Klassen



Klasse „Theaterbesuch“ mit *Ort* und *Datum* als Eigenschaften.

Objektverwaltung

Eine Klasse weiß ohne Objektverwaltung nicht, welche Objekte sie besitzt bzw. welche Objekte von ihr erzeugt (instanziert) wurden.

Objektverwaltung bedeutet, die Klasse „führt Buch“ über das Erzeugen und Löschen ihrer Objekte. Die Klasse erhält die Möglichkeit, Anfragen und Manipulationen auf der Menge der Objekte der Klasse durchzuführen (class extension, object warehouse). Die Objektverwaltung muss im Entwurf und in der Implementierung realisiert werden.

Geschichte der Objektorientierung (1)

Simula: Ole-Johan Dahl + Krysten Nygaard, Norwegen, 1967
Sprache zur Systemsimulation (d.h. Modellierung!)

Smalltalk: Allan Kay, Adele Goldeberg, H. H. Ingalls, Xerox Palo Alto Research Center (PARC), 1976-1980
Graphische Benutzungsoberflächen für Personal Computing

Seit 1986: ParcPlace Smalltalk und andere Hersteller

Geschichte der Objektorientierung (2)

C++: Stroustrup, Bell Labs (New Jersey), 1984
Erweiterung von C um objektorientierte Konzepte

Eiffel: Bertrand Meyer, 1988
Neudesign einer „sauberen“ Sprache für komplexe Softwaresysteme

Objektorientierte Analyse- und Entwurfsmethoden, ca. 1989-1992
Booch, Coad/Yourdon, Rumbaugh et al., Shlaer/Mellor, Coleman et al.

Java: Ken Arnold, James Gosling, Sun Microsystems,1995

Unified Modeling Language (UML): Rational Corp., 1996-97

Prinzipien der Objektorientierung

Dazu gehören:

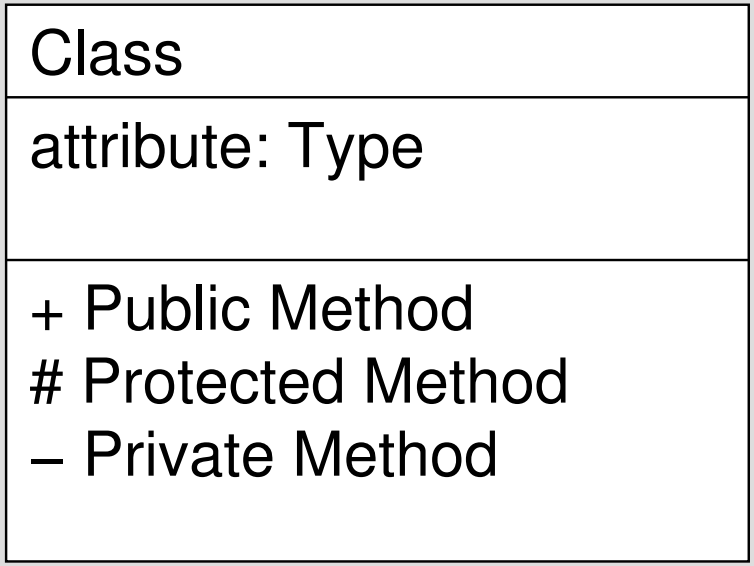
- Abstraktion
- Kapselung
- Wiederverwendung
- Beziehungen
- Polymorphismus

Abstraktion

Darunter versteht man die Trennung von Konzept und Umsetzung, indem man zwischen Objekten und Klassen unterscheidet. Von Klassen gebildete Objekte müssen instanziert werden, um ihre Attribute und das Verhalten zu nutzen, Klassen fassen Objekte des gleichen Typs zusammen.
Mit Hilfe von Klassen und Objekten ist es möglich, die Details einer Problemstellung zu ignorieren und sich auf die grundsätzlichen Gegebenheiten zu konzentrieren. Dadurch ist eine Reduzierung der Komplexität möglich.
Es findet also eine Abbildung der komplexen realen Welt in eine abstrakte logische Ebene (dargestellt durch Objekte und Klassen) statt.

Kapselung

Eine Klasse umfasst eine Menge von Daten (Attributen) und die darauf operierenden Funktionen (sog. *Methoden*). Diese Methoden enthalten Programmcode und dienen als Schnittstelle zur Kommunikation mit anderen Objekten. Die Zusammenfassung von Attributen und Methoden eines Objekts bezeichnet man als Kapselung.



Wiederverwendung

Die Prinzipien der Abstraktion und Kapselung können für die Wiederverwendung von Programmcode genutzt werden. Damit soll die Effizienz gesteigert und die Fehlerrate reduziert werden.

Beziehungen

Objekte werden meist nicht isoliert betrachtet, sie stehen meist in Verbindung mit anderen Objekten.
Es gibt versch. Arten von Beziehungen (sog. Assoziationen) zwischen Objektklassen. Eine Assoziationsart ist z.B die „Teile/Ganzes-Beziehung“, dabei setzt sich ein Objekt aus einer Anzahl anderer Objekte zusammen.
Eine andere Art der Beziehung ist die sog. Vererbungshierachie. Dazu zählen die Generalisierung und die Spezialisierung.

Polymorphismus

Unter Polymorphimus (Vielgestaltigkeit) versteht man die Fähigkeit von Objekten, Objekte anderer Klassen und daraus abgeleiteter Klassen aufzunehmen.
Damit kann die gleiche Methode auf verschiedene Objekte der Hierarchie angewendet werden.

Werkzeuge OOA/OOD (1)

Es gibt diverse Modellierungssoftware für die objektorientierte Analyse (OOA) und das objektorientierte Design (OOD). Im Zuge der ingenieurmäßigen (d.h. systematischen) und rechnergestützten Entwicklung sind viele dieser sog. CASE-Tools (Computer Aided Software Engineering) entstanden. Solch ein CASE-System sollte idealerweise den gesamten Softwarelebenszyklus unterstützten. Alle diese CASE-Werkzeuge basieren heute auf der grafischen Notationssprache UML (Unified Modeling Language), welche mittlerweile eine große Verbreitung hat.

Werkzeuge OOA/OOD (2)

Alle marktrelevanten Werkzeuge können meist aus den UML-Modellen direkt Programmfunktionen in unterschiedlichen Sprachen generieren. Einige sind zudem in der Lage, den Programmcode zu analysieren und durch Reverse Engineering entsprechende UML-Modelle anzupassen. Werden diese Verfahren systematisch genutzt, spricht man von Round-Trip-Engineering.

Beispiele für CASE-Tools sind Innovator (MID), Rational Rose (IBM) und ArgoUML (Open Source).

Unified Modeling Language (UML)

Die UML ist eine Modellierungsprache, also eine Sprache zur Beschreibung von Softwaresystemen. Es gibt eine einheitliche Notation für die Darstellung des gesamten Softwareprozesses. Dazu werden verschiedene Diagrammtypen und natürlichsprachige Beschreibungen genutzt.
Aktuell ist die Version 2.0, welche auch Programmierkonzepte aktueller Programmiersprachen wie C# und Java unterstützt.

Begriffe in der UML (1)

Artefakt: Ergebnisse oder Produkte eines Entwicklungsprozesses, z.B. eine Klasse oder eine Beschreibung eines Elements

Notiz: An jedes UML-Element kann eine Notiz angehängt werden. Diese beschreibt ein UML-Element näher. Grafisch wird dies durch eine gestrichelte Linie von der Notiz zum zu beschreibenden Element dargestellt.

Szenario: ein Ausschnitt aus dem Gesamtsystem, beschreibt bestimmte Aspekte des Systems

Begriffe in der UML (2)

Constraint: Eine Bedingung oder Einschränkung für die Implementierung eines Elements. Die Bedingung steht in geschweiften Klammern und wird mit einer gestrichelten Linie zum betreffenden Element verbunden.

Classifier: Ein Grundelement der UML, z.B. eine Klasse oder Assoziation. Ein Classifier entspricht einem definierten Element in UML.

Stereotyp: dient zur Zusammenfassung von Elementen in Kategorien. Zum Beispiel kann man Klassen in Entitätsklassen (entity), Oberflächenklassen (boundary) und Controllerklassen (control) einteilen.

Diagrammtypen der UML

Strukturdiagramme:
beschreiben die statische Struktur (zeitunabhängig) von Systemen. Es kann die interne Darstellung der Struktur von Klassen bis hin zur Definition von Architekturen komplexer Systeme genutzt werden.
Dazu zählen z.B. Klassendiagramme, Objektdiagramme, Kompositionsstrukturdiagramme, Verteilungsdiagramme

Verhaltensdiagramme:
beschäftigen sich mit dem dynamischen Verhalten von Systemelementen. Eine Unterkategorie sind die Interaktionsdiagramme, welche die Interaktion zwischen Kommunikationspartnern modellieren.
Beispiele sind Aktiviätsdiagramme, Anwendungsfalldiagramme, Zustandsdiagramme, Interaktionsdiagramme (Sequenzdiagramme, Kommunikationsdiagramme, etc.)

Objekte und Klassen in UML

Der Objektname:

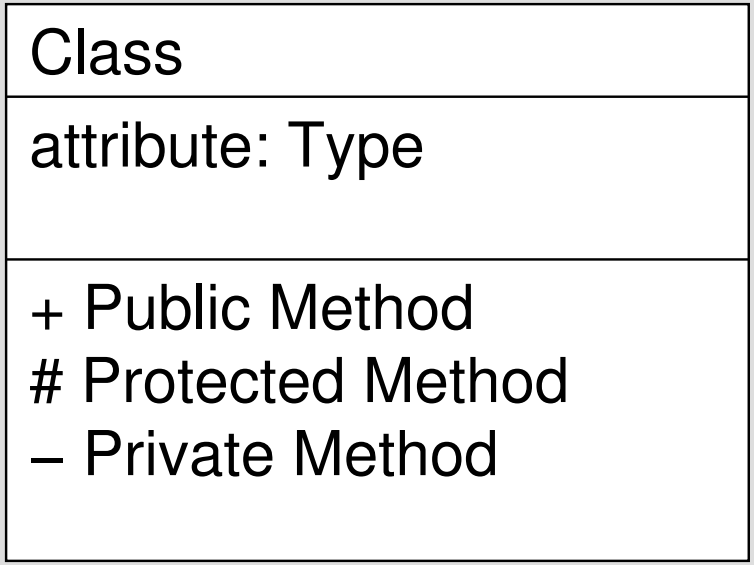
- identifiziert ein Objekt im Objektdiagramm
- muss innerhalb des betrachteten Kontexts (Diagramm) eindeutig sein

Der Klassenname:

- ist ein Substantiv im Singular, ergänzbar durch ein Adjektiv
- ist eindeutig innerhalb des Systems
- optional mit Paketnamen (`paketname::klassenname`)

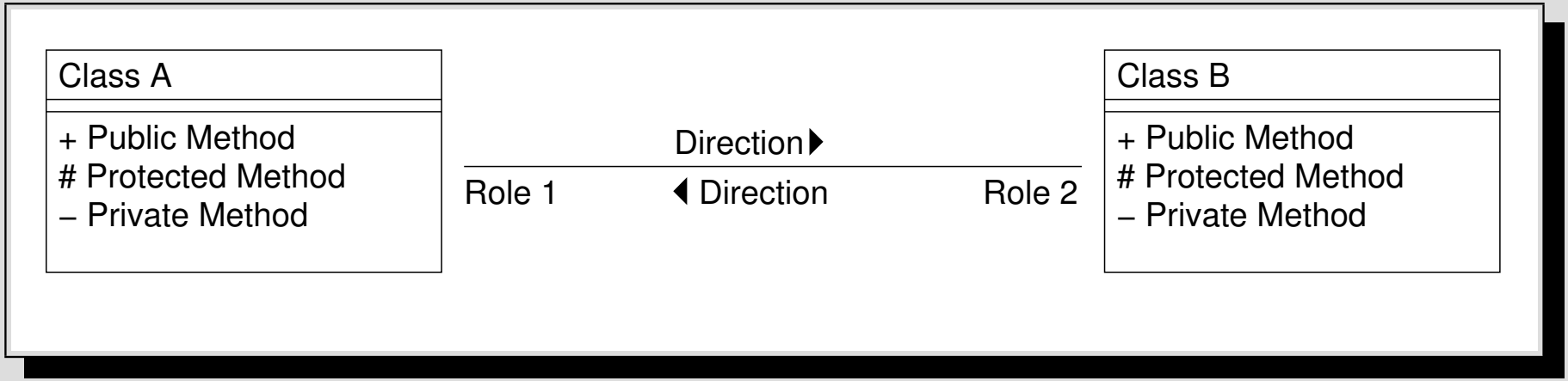
Klassendiagramm

Klassendiagramme stellen die statische Struktur von Systemen dar. Es werden die Klassen mit ihren Eigenschaften (Attributen) und ihrem Verhalten (Operationen) dargestellt. Zudem werden die Beziehungen zwischen den Klassen (Assoziationen) dargestellt.



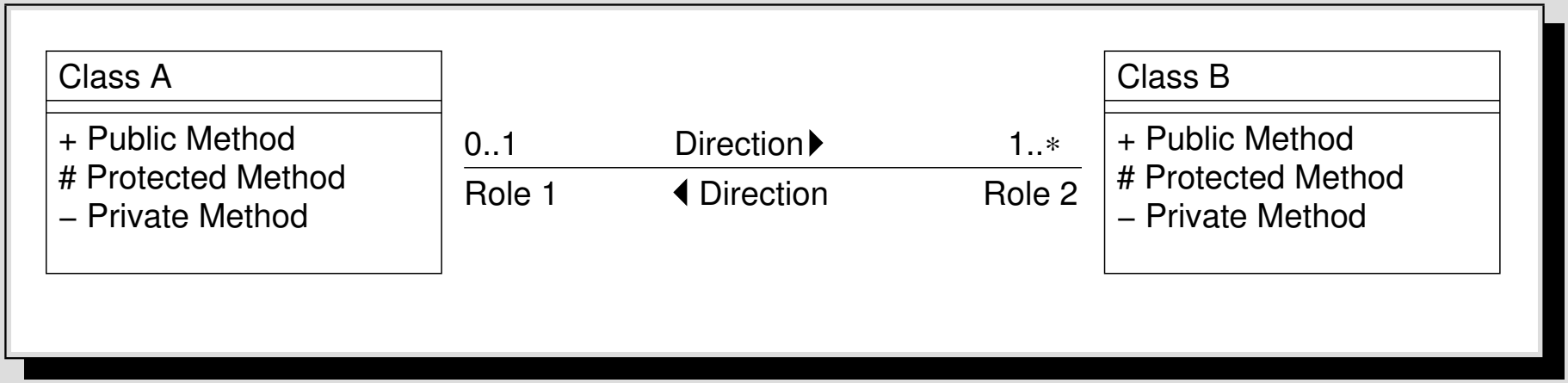
Assoziation

Assoziationen sind die Beziehungen zwischen den Klassen bzw. deren Objekten. Sie werden durch eine Linie zwischen den Klassen dargestellt. Objekte kommunizieren über die Assoziation miteinander. Assoziationen können gerichtet oder ungerichtet sein. Zudem kann die Assoziation einen Namen bekommen, welche die Beziehung zwischen den Klassen beschreibt.



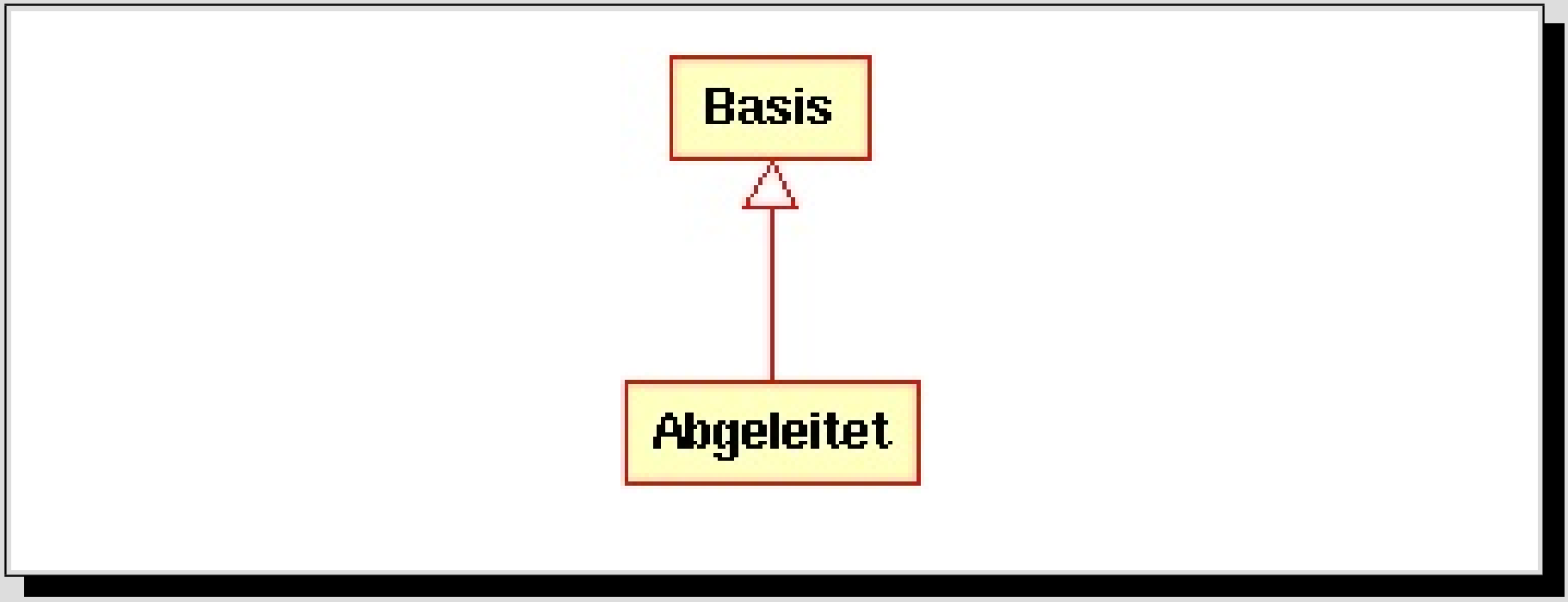
Multiplizitäten

Assoziationen zwischen Klassen können mit sog. Multiplizitäten versehen werden. Diese stellen dar, wieviele Objekte einer Klasse mit wievielen Objekten einer anderen Klassen in Verbindung stehen. Multiplizitäten werden durch eine Zahl auf der Assoziationslinie auf der Seite der betreffenden Klasse dargestellt.



Vererbung

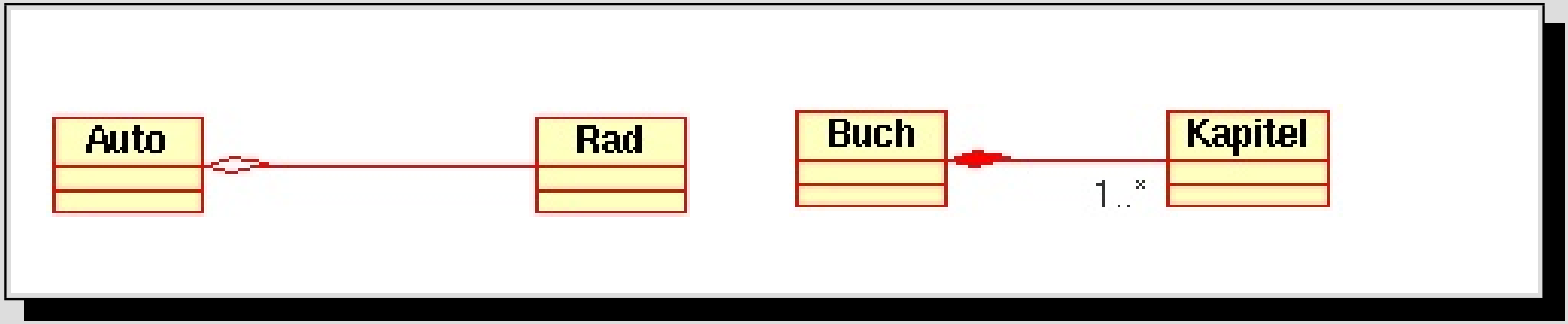
Klassen können über eine Vererbungshierachie miteinander verbunden sein. Dies wird durch einen Pfeil dargestellt, der auf die Oberklasse zeigt. Bei Spezialisierungen bzw. Generalisierungen gibt es keine Multiplizitäten.



Aggregation und Komposition

Mit einer Aggregation kann eine Teile/Ganzes-Beziehung dargestellt werden. Die Aggregation ist eine bestimmte Art der Assoziation. Dargestellt wird diese durch eine Raute an der Seite des Ganzen.

Eine Komposition entspricht einer Aggregation, die Beziehung ist aber existenziell, d.h. die Teile können nur exisieren, wenn das Ganze vorhanden ist. Mit einer ausgefüllten Raute kann eine Komposition dargestellt werden.

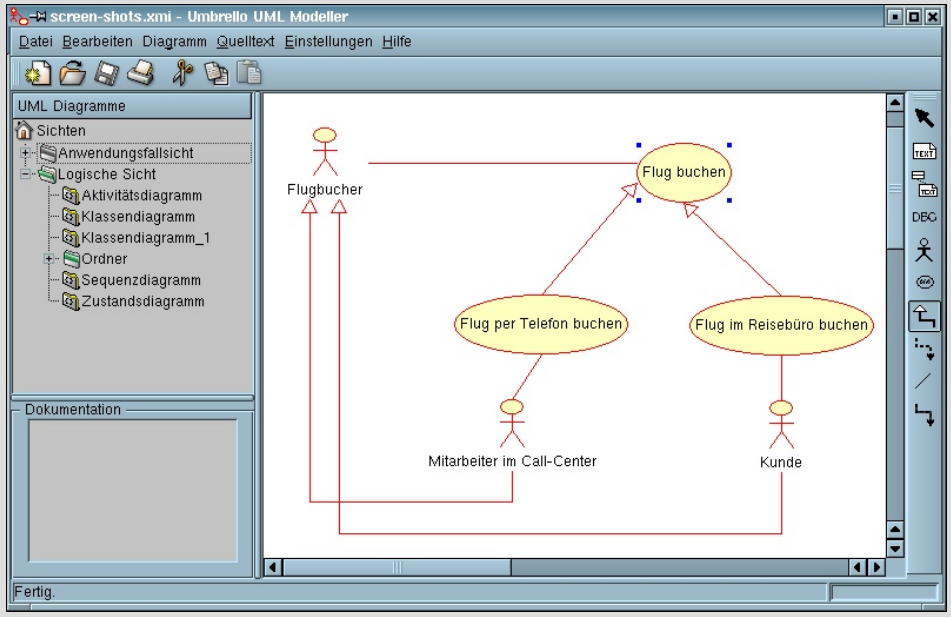


Objektdiagramm

Mit einem Objektdiagramm können die Beziehungen bestimmter Objekte von Klassen dargestellt werden. Es wird somit ein Schnappschuss des Programms zur Laufzeit erzeugt. Zu jedem Klassendiagramm kann mindestens ein Objektdiagramm erstellt werden.

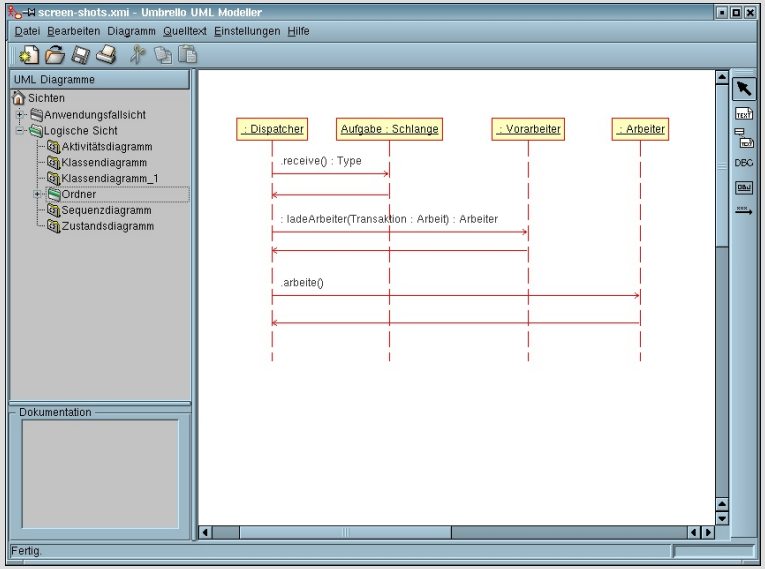
Anwendungsfalldiagramm

In Anwendungsfall- oder sog. use-case-Diagrammen kann das externe Systemverhalten aus Anwendersicht modelliert werden. Ein use case ist ein Geschäftsprozess aus Anwendersicht. use cases werden als Elipsen in einem Kasten, dem System, dargestellt und zu dem dazugehörigen Akteur (dargestellt als Strichmännchen), der dieses Anwendungsfall auslöst, verbunden.



Das Sequenzdiagramm

... stellt die Kommunikation der Objekte in einer bestimmten Szene dar. Die Kommunkation findet über Nachrichten zwischen Objekten statt, welche im Diagramm dargestellt werden. Der Nachrichtenaustausch findet in einer gewissen Reihenfolge statt. Dabei schreitet die Zeit anhand einer Zeitlinie von oben nach unten fort.



objektorientiertes Design (OOD)

Abbild des späteren Programms
- Beschreibung der Klassen, Methoden etc. in Pseudocode

Statisches Modell
- Erstellen der Architekturklassen
- physikalische Verteilung der Funktionalitäten auf Rechnerknoten

Dynamisches Modell
- übersichtliche Beschreibung der komplexen Objektkommunikation
- Erzeugen von Entwurfsmustern

objektorientierte Programmierung

Im Anschluss an das objektorientierte Design sind alle Einzelheiten des Aufbaus und der Funktionalität sowie Ablauf des Programm bekannt. Mit Hilfe einer Programmiersprache kann nun die Implementierung der Software vorgenommen werden.
Es gibt verschiedene objektorientierte Programmiersprachen (Smalltalk, C++, Java, C#, ...). Die verbreitetste und bekannteste ist Java der Firma Sun Mircosystems.

Beispiel für eine Java-Klasse:

```
static class Element {
     Object inhalt;
     Element nachfolger  = null;
     Element vorgänger = null;
     Element (Object obj) { inhalt = obj; }
   };
```

Software-Fehler

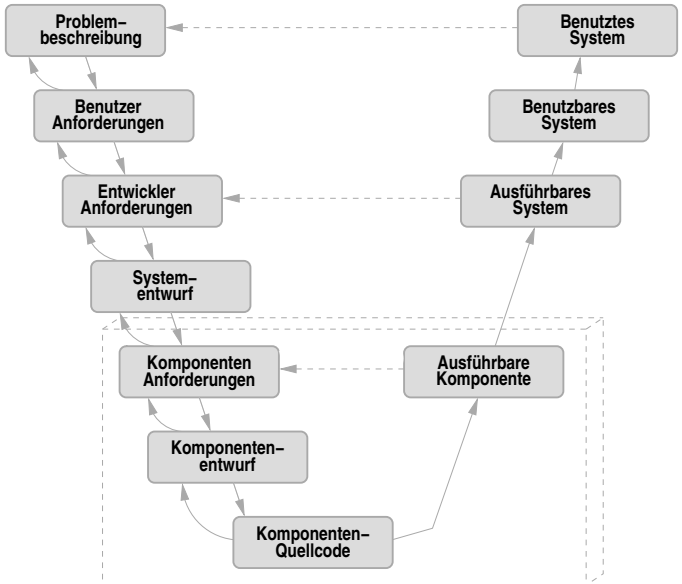
Verifizierung: Wurde die Software richtig gebaut?
Validierung: Wurde die richtige Software gebaut?

Die Validierung kann erst am (fast) fertigen Produkt anhand der Spezifikation durchgeführt werden.

- Die meisten Programme lassen sich (in endlicher Zeit) nicht mit mathematisch exakten Methoden auf Fehlerfreiheit prüfen.
- Die Kosten, um Fehler zu beheben, steigen von der Entwicklungsphase über die Implementation bis zur Anwendung hin stark an.

Validieren und Verifizieren von Software

Das klassische Verfahren des Software-Engineering ist das sogenannte V-Modell, bei dem in jedem Schritt, wenn möglich, kontrolliert wird, ob das Design den Spezifikationen entspricht, und die Implementation korrekt arbeitet.



Whitebox-Tests

Testen, ob alle in der Implementation vorhandenen Kontrollstrukturen und Anweisungen im Programm korrekt arbeiten.
Probleme:

- Nicht-erreichbare Anweisungen,
- Anzahl benötigter Testfälle, Eingabedaten.

Blackbox-Tests

Testen, ob das Programm seine Spezifikation erfüllt, ohne die konkrete Implementierung zu berücksichtigen.
Probleme:

- Anzahl benötigter Testfälle, Rechenzeit,
- Sonderfälle können schlecht erkannt werden, wenn die Implementierung nicht bekannt ist.

Arbeiten im Team

Problem: Eine umfangreiche Software soll/muss von mehreren Personen im Team entwickelt werden.

Lösungsansatz: Entwicklung in Teilprojekte zerlegen (Module, Objektorientierung)

Neues Problem: Wie koordiniert man die Programmierer, und sorgt dafür, dass es kein „Versions-Chaos“ bei den Quelltexten gibt?

Software-Design, Spezifikationen, APIs

Mit Hilfe der bereits angesprochenen Hilfsmittel des Software-Engineering (OOA, OOD, Design Patterns, UML) wird das Problem zunächst auf abstrakter Ebene analysiert, dann eine Spezifikation entworfen, und zur Implementierung eine API (Application Programming Interface) festgelegt, an die sich alle Programmierer halten müssen, wobei jeder an der API für seinen Teilbereich mitarbeitet.

Ohne eine genaue Spezifikation und den Programmierern bekannte API ist es kaum möglich, komplexe Software zu schreiben.

Quellcode-Verwaltung

Um die Arbeit während der Implementationsphase zu erleichtern, sind Systeme entwickelt worden, die verschiedene Versionen von Quelltextdateien verwalten, und Änderungen dokumentieren helfen. Dies sind beispielsweise RCS (Revision Control System für Einzelrechner) und CVS (Concurrent Versions System, ermöglichst es, dass Entwickler das komplette Projekt von einem Server auschecken, und ihre Änderungen wieder einpflegen können, ohne sich gegenseitig zu behindern).
Die meisten Open Source Projekte stellen einen CVS-Server für ihre aktuellen Quelltexte und Dokumentationen im Internet zur Verfügung, mit dem man auch die Fortschritte gut verfolgen kann.

Entwicklungsumgebungen

Entwicklungsumgebungen integrieren einen Editor mit *Syntax-Highlighting*, Dokumentations/API-*Browser*, *Debugger* und optimalerweise auch eine netzwerkfähige *Versionsverwaltung* für das Arbeiten im Team unter einer graphischen Oberfläche.

Obwohl man auch mit den einzelnen Komponenten sehr gut entwickeln kann, erleichtern solche integrierten Umgebungen mit entsprechenden Plugins die Entwicklungsarbeit im Team oft sehr. Ein Beispiel hierfür ist die in Java geschriebene Entwicklungsumgebung `eclipse`, die von mehreren großen Softwarefirmen gemeinsam entwickelt, und unter einer Open Source-Lizenz vertrieben wird.