



## Glossar zur Vorlesung

# Einführung in die Informatik I

Wintersemester 2000/2001

Prof. Bernd Brügge, Ph.D.

Version: 1.06

Letzte Aktualisierung: 6. April 2001

[A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [Q](#) [R](#) [S](#) [T](#) [Ü](#) [U](#) [V](#) [W](#) [Z](#)

### A

[\[zum Glossar-Anfang\]](#)

**Ableitung** [Abschnitt 4, Seite 25]

Eine *Ableitung* entspricht der Anwendung von [Regeln](#) in einem [Ersetzungssystem](#).  
Man unterscheidet dabei zwischen [direkter Ableitung](#) und [indirekter Ableitung](#).

**Abstiegssfunktion** [Abschnitt 8, Seite 60]

Eine *Abstiegssfunktion* wird zum [Nachweis](#) der [Terminierung](#) von [rekursiven Funktionen](#) definiert:

Eine Abstiegssfunktion wird mit den selben [formalen Parametern](#) wie die rekursive Funktion [deklariert](#) und bildet die [Argumente](#) auf eine natürliche Zahl (oder ein Element einer beliebigen anderen Menge, auf der eine [fundierte Ordnung](#) definiert ist) ab.

**Abstrakte Algebra** [Abschnitt 3, Seite 43]

Eine *abstrakte Algebra* besteht aus

- einer Menge von [Elementaroperanden](#)
- einer [Signatur](#)
- einer Menge von [Gesetzen](#)

Die Menge der syntaktisch korrekten [Terme](#) in einer abstrakten Algebra ist durch die Elementaroperanden und die Signatur festgelegt.

#### Abstrakte Klasse [Abschnitt 13, Seite 52]

Eine *abstrakte Klasse* ist eine [Klasse](#), die nicht [instantiiert](#) werden kann.

Eine Klasse wird als abstrakt deklariert, wenn ihre Instantiierung bewusst verhindert werden soll, oder wenn sie mindestens eine [abstrakte Methode](#) enthält.

#### Abstrakte Methode [Abschnitt 13, Seite 52]

Eine *abstrakte Methode* ist eine [Methode](#), die nicht [implementiert](#) ist.

Enthält eine [Klasse](#) eine abstrakte Methode, so ist die Klasse selbst eine [abstrakte Klasse](#).

#### Abstraktion [Abschnitt 1, Seite 9]

Als *Abstraktion* bezeichnet man das Weglassen einzelner Details bei der [Analyse](#) der [Wirklichkeit](#) bzw. eines [Modells](#). Die Abstraktion liefert ein vereinfachtes Abbild der Wirklichkeit bzw. des Modells.

#### Adaptives System [Abschnitt 1, Seite 41]

Ein *adaptives System* ist ein [Informatik-System](#), das sich an Veränderungen in der [Wirklichkeit](#) anpassen kann, indem das dem System zugrundeliegende [Modell](#) entsprechend den geänderten Gegebenheiten adaptiert wird.

#### Aggregation [Abschnitt 2, Seite 7]

Die *Aggregation* stellt eine "Enthaltenseins"-Beziehung zwischen zwei [Objekten](#) dar. Die Beziehung "Objekt A aggregiert Objekt B" kann dabei gelesen werden als:

- A enthält B
- B ist ein Teil von A

#### Aktueller Parameter [Abschnitt 8, Seite 29]

Ein beim Aufruf einer [Operation](#) als [Operand](#) angegebener Wert wird als *aktueller Parameter* bzw. als *Argument* bezeichnet.

Der aktuelle Parameter wird für die Operationsausführung einem [formalen Parameter](#) eindeutig zugeordnet.

#### Algebra [Abschnitt 3, Seite 32]

Eine *Algebra* besteht aus

- einer bzw. mehreren Mengen
- einer Menge von Abbildungen in bzw. zwischen diesen Mengen

In der Informatik ist bzgl. der Formalisierung die Unterscheidung von [abstrakter](#) und [konkreter](#) Algebra von besonderer Bedeutung.

Richtet sich das Interesse mehr auf die Verwendung einer Algebra zur Durchführung von Berechnungen, spricht man auch allgemein von einer [Rechenstruktur](#), ohne in diesem

Zusammenhang zwischen abstrakt und konkret zu unterscheiden.

### Algorithmus [Abschnitt 4, Seite 6]

Ein *Algorithmus* ist ein Verfahren zur Verarbeitung von Daten. Die Beschreibung des Algorithmus ist *endlich* und *präzise*, und alle beschriebenen Arbeitsschritte sind [effektiv](#).

Endet die Ausführung eines Algorithmus immer nach einer endlichen Anzahl von Ausführungsschritten, spricht man von einem [terminierenden](#) Algorithmus.

Zusätzlich lassen sich Algorithmen noch danach klassifizieren, wie genau die Ausführung des Algorithmus festgelegt ist:

- Bei einem [determinierten](#) Algorithmus ist das Ergebnis für jede Eingabe reproduzierbar.
- Bei einem [deterministischen](#) Algorithmus ist die Reihenfolge der einzelnen Ausführungsschritte für jede Eingabe eindeutig festgelegt.

### Allgemeingültigkeit [Abschnitt 6, Seite 32]

Eine [Aussage](#) ist *allgemeingültig*, wenn sie für jede mögliche [Belegung erfüllt](#) ist.

### Alphabet [Abschnitt 4, Seite 29]

Ein *Alphabet* ist ein endlicher [Zeichenvorrat](#), über dem eine [lineare Ordnung](#) definiert ist.

### Analyse [Abschnitt 1, Seite 6]

Mit einer *Analyse* wird versucht, Probleme, die in einer gegebenen [Wirklichkeit](#) existieren, zu erkennen und zu verstehen. Darauf aufbauend lassen sich Anforderungen formulieren, die im Rahmen eines anschließenden [Entwurfsprozesses](#) zu berücksichtigen sind.

### Antisymmetrie

Eine auf einer Menge  $M$  definierte [Relation](#)  $R$  ist *antisymmetrisch*, wenn für alle Elemente  $a$  und  $b$  aus  $M$  gilt:

$$a R b \text{ und } b R a \implies a = b$$

### Anweisung [Abschnitt 9, Seite 28]

*Anweisungen* sind die Grundelemente zur Erstellung [imperativer Programme](#).

Bei der [imperativen Programmierung](#) und dabei speziell bei der [strukturierten Programmierung](#) unterscheidet man folgende Arten von Anweisungen:

- [Deklarationsanweisungen](#),
- [Anweisungssequenzen](#)
- [Zuweisungen](#)
- [bedingten Anweisungen](#)
- [Wiederholungsanweisungen](#) bzw. [Schleifen](#)
- [Prozeduraufrufen](#)
- [Rückgabe-Anweisungen](#)

### Anweisungssequenz [Abschnitt 9, Seite 29]

Eine *Anweisungssequenz* ist eine Folge von [Anweisungen](#), die der Reihe nach ausgeführt werden.

## Argument

siehe *aktueller Parameter*

## Array [Abschnitt 10, Seite 5]

siehe [Reihung](#)

## Attribut [Abschnitt 3, Seite 7]

Ein *Attribut* ist eine jederzeit messbare bzw. durch einen Wert erfassbare Eigenschaft eines [Objekts](#).

## Ausdruck [Abschnitt 8, Seite 16]

Ein *Ausdruck* ist ein [Term](#) innerhalb eines [Programms](#).

Die [Auswertung](#) des Ausdrucks während der Ausführung des Programms liefert einen Wert, dessen [Typ](#) dem Typ des Ausdrucks entspricht.

## Ausgangsmenge [Abschnitt 12, Seite 16]

Die *Ausgangsmenge*  $v^*$  eines [Knotens](#)  $v$  ist die Menge aller [Kanten](#)  $e$  aus der Kantenmenge  $E$  eines [Graphen](#), die von  $v$  ausgehen, d.h.  $v^* = \{e \text{ aus } E \mid e=(v, *)\}$

## Aussage [Abschnitt 5, Seite 11]

Eine *Aussage* ist eine [Nachricht](#), deren [Interpretation](#) innerhalb eines festgelegten [Bezugssystems](#) einen [Wahrheitswert](#) (entweder "wahr" oder "falsch") als [Information](#) liefert.

## Aussagenlogik [Abschnitt 6, Seite 54]

Die *Aussagenlogik* ist eine [Logik](#) auf Basis der [booleschen Algebra](#):

- die Menge der [Formeln](#) ist die Menge aller [syntaktisch korrekten Terme](#) der booleschen Algebra
- die Menge der [Axiome](#) ist eine Teilmenge der Formelmenge. Für die Axiome wird angenommen, dass es sich um [allgemeingültige Aussagen](#) handelt.
- Drei [Inferenzregeln](#):
  - [Modus Ponens](#)
  - [Tertium non datur](#)
  - Gleichheitsgesetz:  
Sind zwei Formeln  $f_1$  und  $f_2$  [semantisch äquivalent](#) bzgl. der [Gesetze](#) der booleschen Algebra, dann kann  $f_2$  aus  $f_1$  bzw.  $f_1$  aus  $f_2$  [abgeleitet](#) werden.

## Auswertung von Ausdrücken [Abschnitt 8, Seite 24]

Bei der *Auswertung* eines [Ausdrucks](#) werden folgende Schritte ausgeführt:

1. Auswertung der [Operanden](#), die ihrerseits Ausdrücke sind
2. Ausführung der [Operation](#) mit den Werten der zuvor ausgewerteten Ausdrücke

## Average-Case-Komplexität [Abschnitt 10, Seite 34]

Die *Average-Case-Komplexität* eines [Algorithmus](#) gibt die durchschnittlichen "Kosten"

an, die bei der Ausführung des Algorithmus entstehen, beschreibt also die [Komplexität](#) im Normalfall.

## Axiom

(1) [Abschnitt 6, Seite 45]

Ein *Axiom* ist eine speziell ausgezeichnete [Formel](#) innerhalb einer [Logik](#).

Axiome sind [Theoreme](#), die nicht [formal bewiesen](#) werden müssen. Sie stellen die Grundlage für formale Beweise anderer Formeln dar.

(2) [Abschnitt 4, Seite 45]

In einer [Chomsky-Grammatik](#) wird ein [Nonterminal](#) als *Axiom* ausgezeichnet.

Dieses Axiom ist der Ausgangspunkt für [Ableitungen](#), d.h. für die Anwendung von [Produktionen](#) der Grammatik.

## B

[\[zum Glossar-Anfang\]](#)

### Backus-Naur-Form [Abschnitt 4, Seite 51]

Die *Backus-Naur-Form* (BNF) ist eine Notation zur kompakten Formulierung von [Grammatiken](#):

- Notation für [Produktionen](#):  
*linke Seite* ::= *rechte Seite*
- [Nonterminale](#) werden in spitzen Klammern gesetzt:  $\langle \text{Nonterminal} \rangle$
- Produktionen mit derselben linken Seite werden zusammengefasst, indem die rechten Seiten als Alternativen (durch | voneinander getrennt) angegeben werden

In der *Erweiterten Backus-Naur-Form* können einzelne Teile der rechten Seite einer Produktion zusätzlich als optional oder auch mehrfach zulässig gekennzeichnet werden.

### Balancierter Binärbaum [Abschnitt 12, Seite 37]

Ein *balancierter Binärbaum* ist ein [Binärbaum](#), in dem für jeden [Knoten](#)  $v$  gilt:

Die beiden [Unterbäume](#) von  $v$  unterscheiden sich bzgl. ihrer [Höhe](#) und/oder der Anzahl der jeweils enthaltenen Knoten nur bis zu einem gewissen Grad, der durch Angabe eines sog. *Balancierungskriteriums* festgelegt wird.

Die Wahl des Balancierungskriteriums hat wesentlichen Einfluss auf die [Komplexität](#) der auf dem Baum ausführbaren [Operationen](#).

## Baum

(1) [Abschnitt 12, Seite 13]

Ein *Baum* ist eine hierarchisch aufgebaute [Datenstruktur](#), deren Darstellung als [Graph](#) einem Baum entspricht.

Man unterscheidet zwischen

- [ungerichteten Bäumen](#)
- [gerichteten Bäumen](#)

(2) [Abschnitt 12, Seite 17]

siehe [gerichteter Baum](#)

### Bedingte Anweisung [Abschnitt 9, Seite 55]

Eine *bedingte Anweisung* ist eine [Anweisung](#), bestehend aus

- einer [Bedingung](#)  $b$
- zwei Anweisungen  $a_1$  und  $a_2$

Zunächst wird die Bedingung  $b$  [ausgewertet](#). Ist diese Bedingung erfüllt, wird  $a_1$  ausgeführt; andernfalls wird  $a_2$  ausgeführt.

### Bedingung [Abschnitt 9]

Eine *Bedingung* ist ein boolescher [Ausdruck](#), d.h. die [Auswertung](#) der Bedingung liefert einen [Wahrheitswert](#), der aussagt, ob die Bedingung erfüllt ist oder nicht.

### Belegung [Abschnitt 6, Seite 50]

Gegeben sei eine [Algebra](#)  $A$  mit einer Menge von [Elementaroperanden](#)  $ID$  und der [Trägermenge](#)  $T$ .

Eine Abbildung  $\beta: ID \rightarrow T$ , die jedem Element aus  $ID$  einen Wert aus  $T$  zuordnet, heißt *Belegung*.

### Berechnung [Abschnitt 7, Seite 18]

Die *Berechnung* eines [Terms](#)  $t$  durch ein [Termersetzungssystem](#)  $R$  ist eine endliche Folge von Termen  $t_0, \dots, t_n$ , wobei gilt:

- für alle  $t_i$  gilt:  
 $t_i$  wird aus  $t_{i-1}$  durch einen [Termersetzungsschritt](#) mit einer [Termersetzungsregel](#) aus  $R$  erzeugt.
- $t = t_n$

### Best-Case-Komplexität [Abschnitt 10, Seite 34]

Die *Best-Case-Komplexität* eines [Algorithmus](#) gibt die minimalen "Kosten" an, die bei der Ausführung des Algorithmus entstehen, beschreibt also die [Komplexität](#)  $t$  im günstigsten Fall.

### Beweistheorie [Abschnitt 6, Seite 6]

Die *Beweistheorie* ist ein Teilbereich der [formalen Logik](#), der sich mit dem Aufbau [formaler Beweise](#) im Zusammenhang mit [formalen Sprachen](#) befasst.

Die Beweistheorie arbeitet auf rein [syntaktischer](#) Ebene.

### Bezugssystem [Abschnitt 5, Seite 11]

Ein [Informationsbezugssystem](#) für die [Interpretation](#) von [Aussagen](#) heißt *Bezugssystem*.

### Binärbaum [Abschnitt 12, Seite 21]

Ein *Binärbaum* ist ein [Baum](#) für den gilt:

Jeder [Knoten](#) des Baums hat maximal zwei [Kindknoten](#).

### Binärer Suchbaum [Abschnitt 12, Seite 26]

siehe [sortierter Binärbaum](#)

### Binäres Suchen [Abschnitt 11, Seite 9]

Für das *binäre Suchen* muss die [Reihung](#), die durchsucht werden soll, [sortiert](#) sein: Entspricht das mittlere Element  $e$  der Reihung nicht dem gesuchten Wert  $s$ , wird

- die Teilreihung links von  $e$  nach dem selben Prinzip durchsucht, falls  $s < e$
- die Teilreihung rechts von  $e$  nach dem selben Prinzip durchsucht, falls  $s > e$

### Bindung [Abschnitt 13, Seite 64]

Die Zuordnung eines Bezeichners zu einem beliebigen benennbaren Teil eines [Programms](#) wird als *Bindung* bezeichnet.

Über den gebundenen Bezeichner kann der Programmteil, der dem Bezeichner zugeordnet ist, innerhalb seines [Bindungsbereichs](#) eindeutig identifiziert werden.

### Bindungsbereich [Abschnitt 13, Seite 64]

Wird in einem [Programm](#) eine [Bindung](#) definiert, so gibt der *Bindungsbereich* den Abschnitt des Programms an, in dem diese Bindung existiert.

Während der Ausführung des Programms existiert eine Bindung, solange der Bindungsbereich nicht verlassen wird, d.h. solange die Beschreibungen der einzelnen auszuführenden Programmschritte Teil des Bindungsbereichs sind.

### Black box reuse [Abschnitt 13, Seite 38]

Als *Black box reuse* bezeichnet man die [Wiederverwendung](#) von [Systemen](#), von denen nur die [Schnittstellen](#) bekannt sind, aber nicht die interne Struktur.

### Boolesche Algebra [Abschnitt 5, Seite 14]

Die *boolesche Algebra* ist eine [Algebra](#) über der Menge der [Wahrheitswerte](#)  $B = \{0, 1\}$  mit

- den [Elementaroperanden](#)  $ID = \{x, y, z\}$
- folgender [Signatur](#):
  - $true: \rightarrow B$
  - $false: \rightarrow B$
  - $NOT: B \rightarrow B$
  - $AND: B \times B \rightarrow B$
  - $OR: B \times B \rightarrow B$

- folgenden [Gesetzen](#):

Gesetz für true:  $true = x \text{ OR } (NOT\ x)$   
 $true = NOT\ false$

Gesetz für false:  $false = (NOT\ x) \text{ AND } x$   
 $false = NOT\ true$

Involutionsgesetz:  $NOT\ (NOT\ x) = x$

Kommutativ-Gesetz:  $x \text{ AND } y = y \text{ AND } x$   
 $x \text{ OR } y = y \text{ OR } x$

Assoziativ-Gesetz:  $x \text{ AND } (y \text{ AND } z) = (x \text{ AND } y) \text{ AND } z$   
 $x \text{ OR } (y \text{ OR } z) = (x \text{ OR } y) \text{ OR } z$

Idempotenz-Gesetz:  $x \text{ AND } x = x$   
 $x \text{ OR } x = x$

|                     |   |
|---------------------|---|
| Neutralitätsgesetz: | $x \text{ OR } \text{false} = x$<br>$x \text{ AND } \text{true} = x$  |
| Absorptions-Gesetz: | $x \text{ AND } (x \text{ OR } y) = x$<br>$x \text{ OR } (x \text{ AND } y) = x$  |
| Distributiv-Gesetz: | $x \text{ AND } (y \text{ OR } z) = (x \text{ AND } y) \text{ OR } (x \text{ AND } z)$<br>$x \text{ OR } (y \text{ AND } z) = (x \text{ OR } y) \text{ AND } (x \text{ OR } z)$ |
| DeMorgan's Gesetz:  | $\text{NOT } (x \text{ AND } y) = (\text{NOT } x) \text{ OR } (\text{NOT } y)$<br>$\text{NOT } (x \text{ OR } y) = (\text{NOT } x) \text{ AND } (\text{NOT } y)$                |

**Boolesche Belegung [Abschnitt 5, Seite 20]**

Eine *boolesche Belegung*  $\beta: ID \rightarrow B$  ordnet jeder [booleschen Variable](#) aus ID einen [Wahrheitswert](#) aus B zu.

**Boolescher Term [Abschnitt 5, Seite 16]**

Ein *boolescher Term* ist ein [syntaktisch korrekter Term](#) einer [booleschen Algebra](#).

**Boolesche Variable [Abschnitt 5, Seite 15]**

Eine *boolesche Variable* ist ein [Elementaroperand](#) für eine [boolesche Algebra](#).

**Bubblesort [Abschnitt 10, Seite 14]**

Der *Bubblesort-Algorithmus* dient zum Sortieren von [Reihungen](#).

Bubblesort funktioniert nach dem Prinzip, das größte Element innerhalb einer Reihung ans Ende der Reihung zu "verschieben" und die verbliebenen, noch nicht sortierten Elemente nach dem selben Prinzip zu sortieren.

Die Verschiebung eines Elements wird durch Vertauschungen realisiert: zwei benachbarte Elemente der Reihung werden vertauscht, wenn sie die [Sortiertheit](#) verletzen.

**C**[\[zum Glossar-Anfang\]](#)**Chomsky-Grammatik [Abschnitt 4, Seite 46]**

Eine *Chomsky-Grammatik* (oder kurz *Grammatik*) ist ein 4-Tupel  $(T, N, P, Z)$  mit

- T: Menge von [Terminalen](#)
- N: Menge von [Nonterminalen](#)
- P: endliche Menge von [Produktionen](#)
- Z: [Axiom](#)

Eine Grammatik entspricht einem [Textersetzungssystem](#), dessen [Zeichenvorrat](#) aus Terminalen und Nonterminalen besteht.

**D**[\[zum Glossar-Anfang\]](#)**Datenstruktur**

Eine *Datenstruktur* ist eine Menge von Elementen, die Daten in strukturierter Form zusammenfasst. Die Strukturierung ist innerhalb einer Datenstruktur einheitlich. Von besonderer Bedeutung sind dabei [rekursive Datenstrukturen](#) bzw. rekursive [Datentypen](#), die [induktiv](#) definiert werden können.

Anders als bei [Rechenstrukturen](#) werden in einer Datenstruktur keine auf dieser Struktur

definierten [Operationen](#) festgelegt.

**Datentyp** [Abschnitt 8, Seite 74]

siehe [Rechenstruktur](#)

**Deklarationsanweisung** [Abschnitt 9, Seite 37]

*Deklarationsanweisungen* sind [Anweisungen](#), die zur Definition von [Prozeduren](#) und [Variablen](#) in [imperativen Programmen](#) verwendet werden.

**Design Pattern**

siehe [Entwurfsmuster](#)

**Detaillierter Entwurf** [Abschnitt 2, Seite 23]

Beim *detaillierten Entwurf* werden die [Klassen](#) und [Objekte](#) des zu entwickelnden [Systems](#) sowie deren [Programmierschnittstellen](#) identifiziert.

Dabei werden auch die [Algorithmen](#) und [Datenstrukturen](#) festgelegt, die für die [Implementation](#) der in der Programmierschnittstelle enthaltenen [Operationen](#) benötigt werden.

**Determiniertheit** [Abschnitt 4, Seite 33]

Ein [Algorithmus](#) ist *determiniert*, wenn er bei wiederholter Auswertung derselben Eingabedaten stets dasselbe Ergebnis liefert, d.h. die Berechnungsergebnisse sind reproduzierbar.

**Determinismus** [Abschnitt 4, Seite 32]

Ein [Algorithmus](#) ist *deterministisch*, wenn er bei wiederholter Auswertung derselben Eingabedaten stets die selbe Folge von Berechnungsschritten ausführt, d.h. die Berechnung ist schrittweise reproduzierbar.

**Dienst** [Abschnitt 3, Seite 33]

Ein *Dienst* ist eine [Operation](#), die ein [System](#) anderen Systemen zur Verfügung stellt. Die Dienste eines Systems werden während des [Systementwurfs](#) festgelegt.

**Direkte Ableitung** [Abschnitt 4, Seite 25]

Eine *direkte Ableitung* entspricht der einmaligen Anwendung einer [Regel](#) in einem [Ersetzungssystem](#).

Notation:

$l \Rightarrow r$  ( $r$  wird direkt abgeleitet aus  $l$ )

**Divide-and-Conquer** [Abschnitt 11, Seite 69]

Mit "*Divide-and-Conquer*" ("*Teile und herrsche*") bezeichnet man ein allgemeines Prinzip zur Lösung von Problemen:

1. *Aufteilung* des Problems in mehrere (möglichst gleich große) Teilprobleme
2. Lösung der Teilprobleme (evtl. [rekursiv](#))
3. *Zusammensetzen* der Teillösungen zur Gesamtlösung

**Dynamische Bindung** [Abschnitt 14, Seite 43]

Bei *dynamischer Bindung* wird erst während der Ausführung des [Aufrufs](#) einer [polymorphen Methode](#) festgelegt, welche der zur Wahl stehenden [Implementationen](#) der

Methode infolge des Aufrufs ausgeführt wird.

## E

[[zum Glossar-Anfang](#)]

### Ebene [Abschnitt 12, Seite 19]

Die *Ebene*  $e_h$  eines [Baums](#) ist die Menge aller [Knoten](#) des Baums mit der [Höhe](#)  $h$ .

### Effektivität

Ein Verfahren ist *effektiv*, wenn es durchführbar ist, d.h. alle Voraussetzungen zur Durchführung sind erfüllt.

### Effizienz

Die *Effizienz* eines [Algorithmus](#) beschreibt, wie gut ein Algorithmus zur Lösung eines Problems geeignet ist, indem es die [Komplexität](#)  $t$  des Problems mit der Komplexität  $t$  des Algorithmus vergleicht. Im Idealfall sind beide Komplexitäten gleich, d.h. der Algorithmus verursacht keine unnötigen "Kosten".

Die Effizienz ist ein Maß für die Qualität  $t$  von Algorithmen und ermöglicht den qualitativen Vergleich verschiedener Lösungen für ein Problem.

### Einbettung [Abschnitt 8, Seite 84]

Unter einer *Einbettung* versteht man die Lösung eines Problems durch die Angabe einer Lösung für ein allgemeineres Problem; die allgemeine Lösung enthält dann auch die Lösung des eigentlichen Problems als Spezialfall.

In der [funktionalen Programmierung](#) wird das Prinzip der Einbettung bei der [Deklaration](#) von [Funktionen](#) verwendet: Eine Funktion berechnet einen Wert, indem sie eine allgemeinere Funktion mit geeigneten [Argumenten](#) aufruft.

### Eingangsmenge [Abschnitt 12, Seite 16]

Die *Eingangsmenge*  ${}^*v$  eines [Knotens](#)  $v$  ist die Menge aller [Kanten](#)  $e$  aus der Kantenmenge  $E$  eines [Graphen](#), die in  $v$  enden, d.h.  ${}^*v = \{e \text{ aus } E \mid e=(*,v)\}$

### Eingebettetes System [Abschnitt 1, Seite 39]

*eingebettetes System* ist ein [Informatik-System](#) zur Lösung von Aufgaben im Verbund mit anderen, nicht zu den Informatik-Systemen gehörenden [Systemen](#).

### Elementaroperand [Abschnitt 3, Seite 40]

Ein *Elementaroperand* ist ein [Operand](#) innerhalb eines [Terms](#).

Bei der [Auswertung](#) eines Terms repräsentiert ein Elementaroperand einen Wert aus der [Trägermenge](#) der [Algebra](#), über der der Term definiert ist.

### Elternknoten [Abschnitt 12, Seite 19]

Ein [Knoten](#)  $v_i$  in einem [Baum](#) ist der *Elternknoten* eines Knotens  $v_j$ , wenn gilt:

Im [Pfad](#) von der [Wurzel](#) des Baums zum Knoten  $v_j$  ist  $v_i$  der direkte Vorgänger von  $v_j$ .

### Entscheidbarkeit [Abschnitt 4, Seite 43]

Ein [formales System](#)  $(U, \Rightarrow)$  ist *entscheidbar*, wenn für beliebige Elemente  $l$  und  $r$  aus  $U$  [effektiv](#) festgestellt werden kann, ob  $r$  aus  $l$  [abgeleitet](#) werden kann.

**Entwurf [Abschnitt 2, Seite 19]**

In der Phase des *Entwurfs* wird systematisch eine Lösung für die während der [Analyse](#) identifizierten Probleme entwickelt.

Die Entwurfsphase lässt sich in zwei Teilphasen unterteilen:

- [Systementwurf](#)
- [detaillierter Entwurf](#)

**Entwurfsmuster [Abschnitt 3, Seite 31]**

*Entwurfsmuster* (engl.: *Design Patterns*) sind allgemeine Beschreibungen von Lösungsansätzen für Probleme, die bei der Entwicklung von [Informatik-Systemen](#) vor allem während der [Analyse](#) und dem [Systementwurf](#) auftreten.

Entwurfsmuster erlauben die Wiederverwendung von bewährten Lösungsideen und können als Grundlage für den Gesamtentwurf von Informatik-Systemen verwendet werden.

**Erfüllbarkeit [Abschnitt 6, Seite 32]**

Eine [Aussage](#)  $x$  ist *erfüllbar*, wenn eine [Belegung](#)  $\beta$  existiert, für die die [Interpretation](#) der Aussage den [Wahrheitswert](#) "wahr" ergibt.

Die Aussage  $x$  ist dann für die Belegung  $\beta$  *erfüllt*.

**Ersetzungssystem**

Ein *Ersetzungssystem* besteht aus

- einer Menge von [Regeln](#)
- [Meta-Regeln](#) zur Festlegung, wo und in welcher Reihenfolge die Regeln angewendet werden.

**Erweiterte Backus-Naur-Form [Abschnitt 4, Seite 54]**

siehe [Backus-Naur-Form](#)

**F**

[\[zum Glossar-Anfang\]](#)

**Falsifizierung [Abschnitt 1, Seite 11]**

Die *Falsifizierung* eines [Modells](#) ist der Nachweis, dass das Modell fehlerhaft ist, d.h. es ist kein korrektes Abbild der modellierten [Wirklichkeit](#). Ein Modell kann z.B. durch die Formulierung eines Gegenbeispiels falsifiziert werden.

**Formale Logik [Abschnitt 6, Seite 13]**

Die *formale Logik* befasst sich mit der Formulierung [formaler Systeme](#) und deren [Interpretation](#).

Wesentliche Teilbereiche der formalen Logik sind die [Beweistheorie](#) und die [Modelltheorie](#).

**Formaler Beweis [Abschnitt 6, Seite 46]**

Ein *formaler Beweis* für eine [Formel](#)  $f$  ist eine endliche Folge  $F = \{f_1, \dots, f_n\}$  von wohldefinierten Formeln  $f_i$ , für die gilt:

- $f = f_n$

- $f_i$  ist entweder ein [Axiom](#) oder kann durch einmalige Anwendung einer [Inferenzregel](#) aus Formeln  $f_j$  ( $j < i$ ) [abgeleitet](#) werden.

### Formaler Parameter [Abschnitt 8, Seite 29]

Ein *formaler Parameter* repräsentiert innerhalb einer [Operation](#) einen beim Aufruf der Operation übergebenen [aktuellen Parameter](#).

### Formale Sprache [Abschnitt 4, Seite 29]

Gegeben seien ein [Zeichenvorrat](#)  $V$  und ein [Textersetzungssystem](#)  $T$ .

Die *formale Sprache*  $L(T, \perp)$  ist die Menge aller [Worte](#), die  $T$  mit dem Wort  $\perp$  aus der Menge  $V^*$  (Menge aller Worte über  $V$ ) als Eingabe erzeugen kann.

### Formales System [Abschnitt 4, Seite 42]

Gegeben seien eine endliche (abzählbar unendliche) Menge  $U$  und eine über  $U$  definierte [Relation](#)  $\Rightarrow$ .

Wenn ein [Algorithmus](#) existiert, der für jedes Tupel  $(\perp, \tau)$  aus  $\Rightarrow$  in endlich vielen Schritten  $\tau$  aus  $\perp$  berechnen kann, dann wird das Tupel  $(U, \Rightarrow)$  als *formales System* bezeichnet.

### Formel [Abschnitt 6, Seite 11]

In der [Logik](#) werden [Terme](#) als *Formeln* bezeichnet.

[Syntaktisch korrekte](#) Formeln werden auch als *wohldefinierte Formeln* (engl.: *well-formed formulas*) bezeichnet.

### Freier Identifikator [Abschnitt 7, Seite 7]

Ein *freier Identifikator* ist ein [Elementaroperand](#), der innerhalb von [Termen](#) stellvertretend für andere Terme verwendet wird.

Durch [Substitution](#) von freien Identifikatoren eines Terms wird eine [Instanz](#) dieses Terms erzeugt.

### Fundierte Ordnung [Abschnitt 8, Seite 59]

Eine [Ordnung](#) ist *fundiert* bzw. *noethersch*, wenn sie beschränkt ist, d.h. jede monotone Folge von Elementen innerhalb der Ordnung ist endlich.

### Funktion [Abschnitt 1, Seite 33]

Eine *Funktion* bildet eine Eingabe eindeutig auf ein Ergebnis ab. Die Anzahl der zulässigen [Argumente](#) wird durch die [Stelligkeit](#) der Funktion beschrieben.

### Funktionale Programmierung [Abschnitt 8]

Die *funktionale Programmierung* ist ein [Programmierstil](#), bei dem Berechnungen ausschließlich durch den [Aufruf von Funktionen](#) durchgeführt werden.

### Funktionale Semantik [Abschnitt 8, Seite 29]

Die *funktionale Semantik* eines [Programms](#) beschreibt das Verhalten des Programms als [Funktion](#), d.h. jeder möglichen Eingabe wird ein Ergebnis zugeordnet.

Für ein [funktionales Programm](#) wird dazu eine Vorschrift zur [Interpretation](#) von [Ausdrücken](#) vereinbart.

### Funktionales Programm [Abschnitt 8, Seite 6]

Ein *funktionales Programm* ist ein [Programm](#), das aus

- einer Menge von [Funktionsdeklarationen](#) und
- einem [Ausdruck](#), in dem die deklarierten Funktionen [aufgerufen](#) werden,

besteht.

Funktionale Programme sind das Ergebnis konsequenter [funktionaler Programmierung](#).

## Funktionalität

(1) [Abschnitt 3, Seite 5]

Die *Funktionalität* eines [Objekts](#) stellt die Summe aller Fähigkeiten des Objekts dar und entspricht der Menge aller [Operationen](#), die das Objekt ausführen kann.

(2) [Abschnitt 3, Seite 39]

Die *Funktionalität* einer [Operation](#) ist eine Beschreibung aller Merkmale, die die jeweilige Operation innerhalb einer [Algebra](#) eindeutig kennzeichnen. Die Funktionalität enthält Informationen über

- den Namen der Operation (Operationsbezeichner)
- die Anzahl der Operanden ([Stelligkeit](#))
- die [Sorten](#) der einzelnen Operanden
- die Sorte des Operationsergebnisses

Die Menge der Funktionalitäten aller Operationen innerhalb einer Algebra wird als [Signatur](#) bezeichnet.

## Funktionsaufruf [Abschnitt 8, Seite 21]

Ein *Funktionsaufruf* ist ein [Ausdruck](#), bestehend aus

- einem Funktionsnamen bzw. -bezeichner
- einer Liste von [Argumenten](#), die ihrerseits Ausdrücke sind.

Der [Typ](#) des Ausdrucks entspricht dem Typ, der im [Funktionskopf](#) der aufgerufenen [Funktion](#) für das Funktionsergebnis vereinbart ist.

## Funktionsberechnungssystem [Abschnitt 1, Seite 33]

Ein *System zur Berechnung von Funktionen* ist ein [Informatik-System](#) zur Berechnung von Funktionen  $f:A \rightarrow B$ .

Eine Eingabe (aus dem Definitionsbereich A) wird auf eine Ausgabe (aus dem Wertebereich B) abgebildet.

## Funktionsdeklaration [Abschnitt 8, Seite 6]

Unter einer *Funktionsdeklaration* versteht man die Vereinbarung einer [Funktion](#), also die Festlegung, welche Abbildung die Funktion tatsächlich durchführt, wenn sie [aufgerufen](#) wird.

Eine Funktionsdeklaration besteht aus

- dem [Funktionskopf](#) bzw. der [Funktionalität](#) der zu deklarierenden Funktion
- dem [Funktionsrumpf](#)

## Funktionskopf [Abschnitt 8, Seite 8]

Bei der [Deklaration](#) einer [Funktion](#) ist der *Funktionskopf* die programmiersprachliche Beschreibung der [Funktionalität](#) der zu deklarierenden Funktion.

### Funktionsrumpf [Abschnitt 8, Seite 8]

Bei der [Deklaration](#) einer [Funktion](#) enthält der *Funktionsrumpf* die programmiersprachliche Beschreibung der Abbildungsvorschrift für die zu deklarierende Funktion.

Ein Funktionsrumpf wird als [Ausdruck](#) formuliert.

## G

[\[zum Glossar-Anfang\]](#)

### Geheimnisprinzip [Abschnitt 3, Seite 15]

Unter dem *Geheimnisprinzip* (engl.: *Information Hiding*) versteht man das Verbergen von internen Details, auf die kein direkter Zugriff erwünscht ist.

Das Geheimnisprinzip wird in der [objekt-basierten](#) bzw. [objekt-orientierten](#) Programmierung über die [Kapselung](#) umgesetzt.

### Generalisierung [Abschnitt 3, Seite 18]

Bei einer *Generalisierung* werden [Merkmale](#) einer bestehenden [Klasse](#) B verwendet, um eine neue Klasse A als [Oberklasse](#) von B zu definieren.

Durch Generalisierung ist der Aufbau einer [Klassenhierarchie](#) möglich.

### Gerichtete Kante [Abschnitt 12]

Eine *gerichtete Kante*  $e=(v_1,v_2)$  beschreibt eine einseitige, nur in einer Richtung bestehende Verbindung von einem [Knoten](#)  $v_1$  zu einem Knoten  $v_2$  in einem [Graph](#).

Notation:  $v_1 \rightarrow v_2$

Existiert zu einer gerichteten Kante  $e=(v_1,v_2)$  auch die gerichtete Kante in entgegengesetzter Richtung  $e'=(v_2,v_1)$ , können beide Kanten zu einer [ungerichteten Kante](#) zusammengefasst werden.

### Gerichteter Baum [Abschnitt 12, Seite 16]

Ein *gerichteter Baum* ist ein [gerichteter Graph](#) ohne [Zyklen](#), in dem ein [Knoten](#) als [Wurzel](#) ausgezeichnet ist, mit dem alle anderen Knoten des Baums durch [Pfade](#) verbunden sind.

### Gerichteter Graph [Abschnitt 12, Seite 15]

Ein *gerichteter Graph* ist ein [Graph](#) mit [gerichteten Kanten](#).

### Geschlossenes System [Abschnitt 2, Seite 13]

Ein *geschlossenes System* ist ein [System](#), das keine Beziehungen zu seiner [Umgebung](#) besitzt, d.h. keine der [System-Komponenten](#) steht in Beziehung zu einer der Umgebungs-Komponenten. Die [Systemschnittstelle](#) ist somit leer.

### Gesetz [Abschnitt 3, Seite 34]

In einer [Algebra](#) beschreiben *Gesetze*, welche [Terme](#) der Algebra "gleichwertig" sind, also dieselben Elemente innerhalb der Algebra beschreiben.

### Grammatik [Abschnitt 4, Seite 44]

siehe [Chomsky-Grammatik](#)

## Graph

(1) [Abschnitt 12, Seite 15]

Ein *Graph*  $G=(V,E)$  besteht aus

- einer Menge von [Knoten](#)  $V$
- einer Menge von [Kanten](#)  $E$ , wobei  $E$  eine [Relation](#) über  $V$  ist

(2) [Abschnitt 12, Seite 15]

Ein *Graph* ist die (bildliche) Darstellung einer [Relation](#), bei der die [Knoten](#) des Graphen die Elemente darstellen, über denen die Relation definiert ist. Die Relation selbst, also die Beziehungen zwischen den Elementen, werden in Form von [Kanten](#) dargestellt, die die in Relation stehenden Knoten miteinander verbinden.

Jede Relation kann als die Menge der Kanten eines Graphen aufgefasst werden.

## Grundterm [Abschnitt 7, Seite 13]

Ein *Grundterm* einer [Algebra](#) ist ein [Term](#), der ausschließlich aus Aufrufen von [Operationen](#) besteht, die in der [Signatur](#) der Algebra enthalten sind.

Grundterme enthalten somit keine [freien Identifikatoren](#).

## Gültigkeitsbereich [Abschnitt 13, Seite 66]

Der *Gültigkeitsbereich* einer [Bindung](#) entspricht der [Lebensdauer](#) der Bindung abzüglich der Bereiche, in denen die Bindung [verschattet](#) ist.

Der Gültigkeitsbereich einer Bindung ist der Bereich, in dem die Bindung tatsächlich in Kraft ist, d.h. während der Ausführung des [Programms](#) gilt die in der Bindung für den Bezeichner festgelegte Zuordnung.

## H

[\[zum Glossar-Anfang\]](#)

### Haltende Regel

Eine *haltende Regel* ist eine [Regel](#) in einem [Markov-Algorithmus](#), deren Anwendung die Ausführung des Algorithmus [terminiert](#).

Notation: *linke Seite*  $\rightarrow$  *rechte Seite*

### Höhe

(1) [Abschnitt 12, Seite 17]

Die *Höhe* eines [Knotens](#) in einem [Baum](#) entspricht der Länge des [Pfad](#)s von der [Wurzel](#) des Baums zum Knoten.

(2) [Abschnitt 12, Seite 17]

Die *Höhe* eines [Baum](#) entspricht dem Maximum der Höhen aller seiner [Knoten](#).

## I

[\[zum Glossar-Anfang\]](#)

### Imperative Programmierung [Abschnitt 9]

Als *imperative Programmierung* bezeichnet man einen [Programmierstil](#), bei dem Berechnungen als eine Folge von [Programmzuständen](#) betrachtet werden.

In einem [imperativen Programm](#) erfolgen die Zustandsübergänge durch die Ausführung

von [Anweisungen](#).

### Imperatives Programm [Abschnitt 9, Seite 28]

Ein *imperatives Programm* ist ein [Programm](#), das aus folgenden Teilen besteht:

- eine [Anweisungssequenz](#) bzw. eine Menge von [Prozeduren](#)
- Daten, die durch [Anweisungen](#) manipuliert werden können

Der Zugriff auf Daten erfolgt üblicherweise mit Hilfe von [Variablen](#).

### Implementation [Abschnitt 2, Seite 19]

Unter einer *Implementation* versteht man die programmiersprachliche Umsetzung einer Lösung, die wiederum das Ergebnis eines [Entwurfs](#) ist.

Beim Erstellen und Testen einer Implementation unterscheidet man im wesentlichen 3 Phasen:

- *Codieren/Editieren*  
Eingabe des Programmtextes
- *Compilieren*  
Übersetzung des Programmtextes ==> ausführbares Programm
- *Exekutieren*  
Ausführen des Programms

### Implementationsvererbung [Abschnitt 13, Seite 39]

Bei der *Implementationsvererbung* wird neben der [Schnittstelle](#) auch die [Implementation](#) der [Oberklasse](#) an die [Unterklasse](#) vererbt.

Mit der Schnittstelle erbt die Unterklasse auch die [Funktionalität](#) der Oberklasse.

[Instanzen](#) der Unterklasse können daher wie Instanzen der Oberklasse verwendet werden. Zusätzlich übernimmt die Unterklasse auch das interne Verhalten der Oberklasse. Durch das [Überschreiben](#) von geerbten [Methoden](#) kann das interne Verhalten der Unterklasse individuell geändert werden.

### Index [Abschnitt 10, Seite 6]

Der *Index* einer [Reihung](#) bildet [Indexpositionen](#) auf die Positionen von Elementen ab.

Durch Angabe der Indexposition eines Elements kann damit direkt auf das Element zugegriffen werden.

### Indexposition [Abschnitt 10, Seite 5]

Die *Indexposition* eines Elements einer [Reihung](#) wird zum Zugriff auf das Element benötigt:

Der [Index](#) der Reihung bildet eine gegebene Indexposition auf die Position des zugehörigen Elements in der Reihung ab.

### Indirekte Ableitung [Abschnitt 4, Seite 25]

Eine *indirekte Ableitung* entspricht der  $n$ -maligen Anwendung von [Regeln](#) in einem [Ersetzungssystem](#).

Notation:

- $l \Rightarrow^+ r \quad (0 < n)$
- $l \Rightarrow^* r \quad (0 \leq n)$

## Induktion [Abschnitt 8, Seite 71]

Unter *Induktion* versteht man das Prinzip, ausgehend von einem einfachen Problem, dessen Lösung bekannt ist, beliebig komplexe Probleme zu formulieren, die dennoch lösbar sind.

Die Grundidee dabei ist, mit jedem Schritt das Problem so zu erweitern, dass die Lösung des erweiterten Problems mit Hilfe der bereits bekannten Lösung des ursprünglichen Problems gefunden werden kann. Die Induktion funktioniert damit genau entgegengesetzt zur [Rekursion](#).

Induktion bildet die Grundlage für Beweisverfahren wie die [vollständige Induktion](#) oder die [strukturelle Induktion](#).

## Inferenzregel [Abschnitt 6, Seite 45]

Die *Inferenzregeln* einer [Logik](#) sind [Metaregeln](#) zur [Ableitung](#) einer [Formel](#) aus den [Axiomen](#) bzw. anderen Formeln dieser Logik.

## Informatik-System [Abschnitt 1, Seite 31]

Ein *Informatiksystem* ist ein [System](#), das auf einem Rechner ausführbar ist.

## Information [Abschnitt 5, Seite 5]

Der abstrakte Gehalt bzw. die [Semantik](#) einer [Nachricht](#) wird als *Information* bezeichnet.

## Informationsbezugssystem [Abschnitt 5, Seite 8]

Ein *Informationsbezugssystem* ist ein Ausschnitt aus der [Wirklichkeit](#), der für die [Interpretation](#) einer [Nachricht](#) benötigt wird.

Die [Information](#), die sich aus der Interpretation ergibt, ist nur innerhalb des Informationsbezugssystems sinnvoll verwendbar.

## Informationssystem [Abschnitt 5, Seite 9]

Ein *Informationssystem*  $(A, R, I)$  besteht aus einem [semantischen Modell](#)  $A$ , einem [Repräsentationssystem](#)  $R$  und einer [Interpretation](#)  $I$ , die Repräsentationen auf [Informationen](#) aus  $A$  abbildet.

## Inorder [Abschnitt 12, Seite 43]

siehe [Inordnung](#)

## Inordnung [Abschnitt 12, Seite 43]

Gegeben sei ein [Binärbaum](#)  $b$ .

Eine [Traversierung](#) von  $b$  erfolgt in *Inordnung* (*Inorder*), wenn für jeden [Knoten](#)  $k$  aus  $b$  folgende Durchlaufreihenfolge eingehalten wird:

1. Linker [Unterbaum](#) von  $k$
2.  $k$
3. Rechter [Unterbaum](#) von  $k$

## Insertionsort [Abschnitt 10, Seite 44]

Der *Insertionsort-Algorithmus* dient zum Sortieren von [Reihungen](#).

Insertionsort funktioniert nach dem Prinzip, solange ein Element aus dem unsortierten Teil der Reihung zu entnehmen und an der richtigen Stelle in den sortierten Teil der

Reihung einzufügen, bis der unsortierte Teil leer ist.

### Instantiierung [Abschnitt 3, Seite 24]

Die *Instantiierung* einer [Klasse](#) entspricht der Erzeugung eines [Objekts](#), das die für diese Klasse von Objekten charakteristischen [Merkmale](#) aufweist.

### Instanz

(1) [Abschnitt 3, Seite 9]

Als *Instanz* einer [Klasse](#) wird jedes [Objekt](#) bezeichnet, das der entsprechenden Klasse angehört bzw. durch [Instantiierung](#) der Klasse erzeugt wird.

(2) [Abschnitt 7, Seite 10]

Eine *Instanz* eines [Terms](#), der [freie Identifikatoren](#) enthält, entsteht durch [Substitution](#) dieser freien Identifikatoren.

### Instanzmethode [Abschnitt 13]

Eine [Methode](#) wird auch als *Instanzmethode* bezeichnet, wenn besonders betont werden soll, dass die Methode, die für eine [Klasse](#) von [Objekten](#) definiert ist, von [Instanzen](#) dieser [Klasse](#) ausgeführt wird.

### Interaktives System [Abschnitt 1, Seite 35]

Ein *Interaktives System* ist ein [Informatik-System](#), das mit anderen [Systemen](#) interagiert. Eingabedaten werden von anderen Systemen empfangen, Ausgabedaten werden an andere Systeme gesendet.

### Interpretation

(1) [Abschnitt 5, Seite 8]

Durch *Interpretation* wird aus der [Repräsentation](#) einer [Nachricht](#) eine [Information](#) ermittelt.

Eine Interpretation findet stets innerhalb eines [Informationsbezugssystems](#) statt.

(2) [Abschnitt 5, Seite 9]

Gegeben seien ein [Repräsentationssystem](#)  $R$  und ein [semantisches Modell](#)  $A$ . Die [Repräsentation](#)  $r$  einer gegebenen [Nachricht](#) sei aus  $R$ .

Die *Interpretation*  $I: R \rightarrow A$  ordnet  $r$  eine [Information](#)  $I[r]$  aus  $A$  zu.

(3) [Abschnitt 6, Seite 50]

Gegeben sei eine [Algebra](#)  $A$  mit einer Menge von [Elementaroperanden](#)  $ID$ , einer [Signatur](#)  $\Sigma$  und der [Trägermenge](#)  $T$ .

Für jede [Belegung](#)  $\beta$  ist die *Interpretation* eines [Terms](#)  $t$  wie folgt definiert:

- für alle  $x$  aus  $ID$  gilt:  $I_{\beta}[x] = \beta(x)$
- für alle  $f$  aus  $\Sigma$  gilt:  $I_{\beta}[f(t_1, \dots, t_n)] = f(I_{\beta}[t_1], \dots, I_{\beta}[t_n])$

### Irreflexivität

Eine auf einer Menge  $M$  definierte [Relation](#)  $R$  ist *irreflexiv*, wenn für kein Element  $a$  aus  $M$  gilt:  
 $a R a$

**K**[\[zum Glossar-Anfang\]](#)**Kalkül** [Abschnitt 4, Seite 43]

Ein [formales System](#) ( $\cup, \Rightarrow$ ) wird als *Kalkül* bezeichnet, wenn die [Relation](#)  $\Rightarrow$  durch eine endliche Menge von [Regeln](#) und eine endliche Menge von [Metaregeln](#) definiert werden kann.

**Kante** [Abschnitt 12, Seite 15]

Als *Kante* bezeichnet man die Verbindung zwischen zwei [Knoten](#) in einem [Graph](#). Man unterscheidet dabei

- [gerichtete Kanten](#)
- [ungerichtete Kanten](#)

**Kapselung** [Abschnitt 3, Seite 15]

Die Zusammenfassung der [Implementationen](#) von [Attributen](#) und [Operationen](#) in einem [Objekt](#) bei gleichzeitiger Festlegung einer [Schnittstelle](#) für das Objekt wird als *Kapselung* (engl.: *Encapsulation*) bezeichnet.

Die Kapselung stellt ein wesentliches Merkmal der [objekt-basierten](#) bzw. [objekt-orientierten](#) Programmierung dar.

Der Zugriff auf die im Objekt gekapselten Implementationen ist von außen nur über dessen Schnittstelle möglich. Die Kapselung entspricht somit einer Umsetzung des [Geheimnisprinzips](#).

**Kardinalität** [Abschnitt 2, Seite 8]

Die *Kardinalität* oder auch [Vielfachheit](#) entspricht der Anzahl von Gegenständen auf einer Seite einer Beziehung, die durch diese Beziehung mit anderen Gegenständen in einen Zusammenhang gebracht werden.

**Kaskaden-Rekursion** [Abschnitt 8, Seite 48]

Eine [rekursive Funktion](#) ist *kaskadenartig rekursiv*, wenn sie nicht [linear rekursiv](#) ist, d.h. bei der [Auswertung](#) des [Funktionsrumpfs](#) kann mehr als ein rekursiver [Funktionsaufruf](#) stattfinden.

**Keller** [Abschnitt 3, Seite 44]

Ein *Keller* bzw. Stapel (engl.: *Stack*) ist eine [Rechenstruktur](#) mit [Operationen](#) zum Einfügen und Entnehmen einzelner Elemente.

Keller arbeiten nach dem LIFO-Prinzip ("last in, first out"), d.h. das zuletzt eingefügte Element wird als erstes wieder entnommen.

**Kindknoten** [Abschnitt 12, Seite 19]

Ein [Knoten](#)  $v_j$  in einem [Baum](#) ist ein *Kindknoten* eines Knotens  $v_i$ , wenn gilt:

Im [Pfad](#) von der [Wurzel](#) des Baums zum Knoten  $v_j$  ist  $v_j$  der direkte Nachfolger von  $v_i$ .

**Klasse** [Abschnitt 3, Seite 9]

Eine *Klasse* ist eine Menge von [Objekten](#) mit gemeinsamen [Merkmalen](#).

**Klassenhierarchie** [Abschnitt 3, Seite 15]

Eine *Klassenhierarchie* besteht aus [Klassen](#), die zueinander in einer

[Vererbungsbeziehung](#) stehen.

Eine Klassenhierarchie kann sowohl durch [Generalisierung](#) als auch durch [Spezialisierung](#) aufgebaut werden.

### Klassenschnittstelle [Abschnitt 13, Seite 60]

Besteht die Definition einer [Klasse](#) nur aus einer [Schnittstelle](#) ohne [Implementation](#), dann kann die Klasse als *Klassenschnittstelle* betrachtet werden.

Bei Verwendung einer Klassenschnittstelle als [Oberklasse](#) findet eine reine [Schnittstellenvererbung](#) statt, d.h. die Implementierung der Schnittstelle erfolgt ausschließlich in der [Unterklasse](#).

### Klassifikation [Abschnitt 3, Seite 15]

Unter *Klassifikation* versteht man die Einteilung von [Komponenten](#) bzw. [Objekten](#) in verschiedene [Klassen](#).

Komponenten und Objekte werden entsprechend ihrer [Schnittstelle](#) klassifiziert. Durch Klassifikation lassen sich existierende Komponenten und Objekte in [Klassenhierarchien](#) einordnen.

### Knoten [Abschnitt 12, Seite 15]

Ein *Knoten* ist ein Element eines [Graphen](#). Verbindungen mit anderen Knoten innerhalb des Graphen werden durch [Kanten](#) dargestellt.

### Komplexität [Abschnitt 10, Seite 32]

Unter der *Komplexität* eines [Algorithmus](#) versteht man die "Kosten", die bei der Ausführung des Algorithmus - abhängig von der Größe der Eingabedaten - verursacht werden.

Zwei häufig genutzte Kategorien von Komplexitäten sind

- [Platzkomplexität](#)
- [Zeitkomplexität](#)

Bei der [Analyse](#) der Komplexität eines Algorithmus wird außerdem noch die Situation bei der Ausführung des Algorithmus berücksichtigt; in der Regel werden die Komplexitäten für folgende Fälle betrachtet:

- [Best-Case-Komplexität](#)
- [Average-Case-Komplexität](#)
- [Worst-Case-Komplexität](#)

Üblicherweise werden Algorithmen anhand ihrer Komplexität in eine [Ordnung](#) eingeteilt, die einen Näherungswert für die Größenordnung der Komplexität beschreibt.

### Komponente [Abschnitt 2, Seite 3]

Eine *Komponente* ist ein Gegenstand in einem [System](#) bzw. in der [Umgebung](#) eines Systems.

### Konkrete Algebra [Abschnitt 3, Seite 49]

Eine *konkrete Algebra* stellt eine [Implementation](#) einer [abstrakten Algebra](#) dar. Sie besteht aus

- einer [Trägermenge](#), die eine Obermenge der [Elementaroperanden](#) der abstrakten Algebra ist
- einer Menge von [Funktionen](#), die alle in der [Signatur](#) der abstrakten Algebra enthaltenen [Operationen](#) gemäß  $\beta$  der für diese Algebra festgelegten [Gesetze implementieren](#).

Mit einer konkreten Algebra können [Terme](#) ausgewertet werden, d.h. das Element der Algebra, das durch einen Term beschrieben wird, wird berechnet.

### Konkrete Funktion [Abschnitt 3, Seite 49]

Eine *konkrete Funktion* ist die [Implementation](#) einer [Operation](#) einer [abstrakten Algebra](#). Eine konkrete Funktion ist somit Teil einer [konkreten Algebra](#).

### Konstruktoroperation

*Konstruktoroperationen* sind die [Operationen](#) einer [Algebra](#), durch deren (ein- oder mehrmalige) Anwendung sich alle Elemente der Algebra konstruieren lassen.

### Korrektheit

(1) [Abschnitt 8, Seite 72]

Eine ([rekursive](#)) [Funktion](#) ist *korrekt*, wenn gilt:

- Die Funktion ist [partiell korrekt](#).
- Die Funktion [terminiert](#)

Die partielle Korrektheit wird normalerweise mit Hilfe von [Induktion](#) nachgewiesen; für den [Terminierungsnachweis](#) kann eine [Abstiegsfunktion](#) definiert werden.

(2) [Abschnitt 6, Seite 53]

Eine [Logik](#) ist *korrekt* (engl.: *sound*), wenn jedes ihrer [Theoreme allgemeingültig](#) ist.

## L

[\[zum Glossar-Anfang\]](#)

### Lebensdauer [Abschnitt 13, Seite 66]

siehe [Bindungsbereich](#)

### Lineare Ordnung [Abschnitt 4, Seite 29]

Eine [partielle Ordnung](#)  $R$  ist eine *lineare Ordnung* (manchmal auch *totale Ordnung* genannt), wenn für alle Elemente  $a, b$  aus der Menge  $M$  ( $R$  Teilmenge von  $M \times M$ ) mit  $a \neq b$  gilt:

Entweder  $a R b$  oder  $b R a$

### Lineare Rekursion [Abschnitt 8, Seite 46]

Eine [rekursive Funktion](#) ist *linear rekursiv*, wenn bei der [Auswertung](#) des [Funktionsrumpfs](#) in jedem Fall höchstens ein rekursiver [Funktionsaufruf](#) stattfindet.

### Logik [Abschnitt 6, Seite 45]

Eine *Logik* ist in der [formalen Logik](#) definiert als ein [formales System](#), bestehend aus

- einer Menge von Symbolen
- einer Menge von [Formeln](#), die aus diesen Symbolen (durch Anwendung von [Regeln](#)) konstruiert werden können

- einer Menge von [Axiomen](#)
- einer Menge von [Inferenzregeln](#)

## M

[\[zum Glossar-Anfang\]](#)

### Markov-Algorithmus [Abschnitt 4, Seite 38]

Ein *Markov-Algorithmus* ist ein [Textersetzungssystem](#) in dem auch [haltende Regeln](#) als [Regeln](#) zulässig sind, und für das folgende Meta-Regeln gelten:

- Es wird immer die erste anwendbare Regel angewendet. Kann die Regel an mehreren Stellen angewendet werden, dann wird sie an der von links aus ersten dieser Stellen angewendet.
- Das System [terminiert](#), wenn keine Regel mehr anwendbar ist, oder unmittelbar nach der Anwendung einer haltenden Regel.

### Mehrfachvererbung [Abschnitt 13, Seite 56]

[Erbt](#) eine [Unterklasse](#) von mehreren direkten [Oberklassen](#), so spricht man von *Mehrfachvererbung*.

### Merkmal [Abschnitt 3, Seite 7]

Die *Merkmale* eines [Objekts](#) sind

- die [Attribute](#), die für das Objekt definiert sind
- die [Operationen](#), die auf dem Objekt ausgeführt werden können

### Meta-Regel [Abschnitt 4, Seite 23]

Eine *Meta-Regel* legt fest, wie die [Regeln](#) in einem [Ersetzungssystem](#) anzuwenden sind.

### Metasprache [Abschnitt 6, Seite 15]

Eine *Metasprache* ist eine Sprache, die zur Beschreibung einer [formalen Sprache](#) verwendet wird.

Metasprachen werden auch zur Formulierung von [Metaregeln](#) verwendet.

### Methode [Abschnitt 9, Seite 16]

Im Rahmen des [detaillierten Entwurfs](#) und der [objekt-basierten Implementierung](#) werden [Operationen](#) als *Methoden* bezeichnet.

Methoden werden mit Hilfe von [Methodendeklarationen](#) definiert und von [Objekten](#) ausgeführt.

Liefert eine Methode einen Ergebniswert, wird sie auch als [Funktion](#) bezeichnet, ansonsten spricht man von einer [Prozedur](#).

### Methodenaufruf [Abschnitt 9, Seite 50]

Beim *Aufruf* einer [Methode](#) werden die [Anweisungen](#) im [Rumpf](#) der Methode der Reihe nach ausgeführt.

Handelt es sich bei der Methode um eine [Funktion](#), wird nach Ausführung der Anweisungen das Funktionsergebnis zurückgegeben. Bei einer [Prozedur](#) wird die Ausführung des [Programms](#) fortgesetzt, indem die Anweisung unmittelbar nach dem Methodenaufruf ausgeführt wird.

**Methodendeklaration** [Abschnitt 9, Seite 16]

Die *Methodendeklaration* ist die programmiersprachliche Beschreibung einer [Methode](#). Eine Methodendeklaration besteht aus

- einem [Methodenkopf](#)
- einem [Methodenrumpf](#)

**Methodenkopf** [Abschnitt 9, Seite 45]

Der *Kopf* einer [Methodendeklaration](#) gibt die [Funktionalität](#) der zu deklarierenden [Methode](#) an.

Im Methodenkopf wird auch festgelegt, ob eine Methode einen Ergebniswert liefert oder nicht.

**Methodenrumpf** [Abschnitt 9, Seite 45]

Der *Rumpf* einer [Methodendeklaration](#) besteht aus einer Folge von [Anweisungen](#).

**Modell**

(1) [Abschnitt 1, Seite 8]

Als *Modell* bezeichnet man die Kombination aus

- einer Menge von Begriffen für physikalisch existierende oder gedachte (fiktive) "Gegenstände" (Dinge, Personen, Abläufe in der Zeit)
- Beziehungen zwischen diesen Begriffen

Ein Modell ist das Abbild einer physikalischen oder fiktiven [Wirklichkeit](#).

(2) [Abschnitt 6, Seite 7]

Liefert eine [Interpretation](#)  $\mathcal{I}$  für eine Menge von [Formeln](#)  $F$  den [Wahrheitswert](#) "wahr", dann ist die Interpretation  $\mathcal{I}$  ein *Modell* für die Menge  $F$ .

(3) [Abschnitt 6, Seite 53]

Liefert die [Interpretation](#)  $\mathcal{I}$  für alle [Theoreme](#) einer [Logik](#)  $L$  den [Wahrheitswert](#) "wahr", dann ist die Interpretation  $\mathcal{I}$  ein *Modell* für die Logik  $L$ .

**Modelltheorie** [Abschnitt 6, Seite 6]

Die *Modelltheorie* ist ein Teilbereich der [formalen Logik](#), der sich mit der [Interpretation formaler Sprachen](#) befasst.

Die Modelltheorie arbeitet auf [semantischer](#) Ebene.

**Modus Ponens** [Abschnitt 6, Seite 54]

*Modus Ponens* besagt:

Wenn eine [Aussage](#)  $x$  wahr ist und aus  $x$  die Aussage  $y$  folgt ( $x \implies y$ ), dann ist auch  $y$  eine wahre Aussage.

Modus Ponens ist eine der für die [Aussagenlogik](#) definierten [Inferenzregeln](#).

**N**

[\[zum Glossar-Anfang\]](#)

**Nachordnung** [Abschnitt 12, Seite 43]

Gegeben sei ein [Binä rbaum](#)  $b$ .

Eine [Traversierung](#) von  $b$  erfolgt in *Nachordnung* (*Postorder*), wenn für jeden [Knoten](#)  $k$

aus  $b$  folgende Durchlaufreihenfolge eingehalten wird:

1. Linker [Unterbaum](#) von  $k$
2. Rechter [Unterbaum](#) von  $k$
3.  $k$

**Nachricht** [Abschnitt 5, Seite 5]

Eine *Nachricht* ist eine wahrnehmbare Mitteilung.

**Noethersche Ordnung** [Abschnitt 8, Seite 59]

siehe [fundierte Ordnung](#)

**Nonterminal** [Abschnitt 4, Seite 45]

Ein *Nonterminal* (auch *syntaktische Variable* genannt) ist ein Zeichen aus dem [Zeichenvorrat](#) einer [Chomsky-Grammatik](#), dem eine syntaktische Bedeutung zugeordnet ist.

**Normalform** [Abschnitt 3, Seite 53]

Ein [Term](#) in einer [Algebra](#) entspricht der *Normalform*, wenn die Anzahl der im Term angewendeten [Operationen](#) minimal ist.

Ein Term in Normalform beschreibt den einfachstmöglichen, direkten Aufbau eines Elements der Algebra und kann daher auch durch Anwendung von [Gesetzen](#), die innerhalb der Algebra gelten, nicht weiter vereinfacht werden.

**Null-Indizierung** [Abschnitt 10, Seite 6]

Bildet der [Index](#) einer [Reihung](#) mit  $n$  Elementen jede [Indexposition](#)  $i$  zwischen Null und  $n-1$  auf die  $(i+1)$ -te Position ab, spricht man von einer *Null-Indizierung*.

## O

[\[zum Glossar-Anfang\]](#)

**Oberklasse** [Abschnitt 3, Seite 18]

Eine *Oberklasse* [vererbt](#) all ihre [Merkmale](#) an ihre [Unterklassen](#).

**Objekt** [Abschnitt 3, Seite 5]

Ein *Objekt* ist Teil eines [Systems](#) und repräsentiert einen beliebigen Gegenstand innerhalb dieses Systems. Ein Objekt kann selbst wiederum als Teilsystem betrachtet werden. Ein gegebenes Objekt definiert sich über seinen aktuellen [Zustand](#) und seine [Funktionalität](#). Soll die Zugehörigkeit des Objekts zu einer [Klasse](#) besonders betont werden, wird das Objekt auch als [Instanz](#) dieser Klasse bezeichnet.

**Objekt-basierte Programmierung** [Abschnitt 13, Seite 3]

*Objekt-basierte Programmierung* ist ein [Programmierstil](#), bei dem Daten bzw. [Attribute](#) und [Operationen](#) bzw. [Methoden](#) in [Objekten gekapselt](#) werden.

Analog zum Prinzip der [Klassifikation](#) können Objekte mit gleichen [Merkmalen](#) als [Klasse](#) definiert werden.

Der Ablauf eines objekt-basierten Programms besteht in der Ausführung von Methoden von Objekten, die durch den Aufruf von Methoden anderer Objekte auch untereinander kommunizieren können.

## Objekt-orientierte Programmierung [Abschnitt 13]

Die *objekt-orientierte Programmierung* ist ein auf der [objekt-basierten Programmierung](#) aufbauender [Programmierstil](#).

Zusätzlich zur [Kapselung](#) wird bei objekt-orientierter Programmierung durch die Möglichkeit der [Vererbung](#) von [Merkmalen](#) von [Objekten](#) die Bildung von [Klassenhierarchien](#) und damit die [Wiederverwendbarkeit](#) von [Klassen](#) unterstützt.

## Offenes System [Abschnitt 2, Seite 13]

Ein *offenes System* ist ein [System](#), das in Beziehung zu seiner [Umgebung](#) steht, d.h. eine oder mehrere [System-Komponenten](#) stehen in Beziehung zu Umgebungs-Komponenten. Diese Beziehungen stellen die [Schnittstelle des Systems](#) dar.

## Operand

Ein *Operand* repräsentiert einen Eingabewert für eine [Operation](#).

## Operation

(1) [Abschnitt 3, Seite 7]

Eine *Operation* ist eine Tätigkeit, die ein [Objekt](#) ausführen kann.

Im Rahmen des [detaillierten Entwurfs](#) und der [objekt-basierten Implementierung](#) werden Operationen auch als [Methoden](#) bezeichnet.

(2) [Abschnitt 4, Seite 35]

Eine *Operation* bildet eine Anzahl von Eingabewerten (die [Operanden](#)) auf einen Ergebniswert ab.

## Operationale Semantik [Abschnitt 8, Seite 89]

Die *operationale Semantik* eines [Programms](#) beschreibt die Abfolge der einzelnen Berechnungsschritte bei der Ausführung des Programms.

Ein [funktionales Programm](#) entspricht dabei im wesentlichen einem [Termersetzungssystem](#).

## Ordnung

(1) [Abschnitt 10, Seite 34]

Die *Ordnung*  $O(f(n))$  einer [Funktion](#)  $f: \mathbb{N} \rightarrow \mathbb{R}$  ist die Menge aller Funktionen, die nicht schneller wachsen als  $f$ , d.h.  $f$  ist bzgl. des Wachstums eine obere Schranke für alle in der Ordnung enthaltenen Funktionen:

$$O(f(n)) = \{t: \mathbb{N} \rightarrow \mathbb{R}^+ \mid \text{es gibt ein } c \text{ aus } \mathbb{R}^+ \text{ und ein } n_0 \text{ aus } \mathbb{N}, \text{ so dass für alle } n > n_0 \text{ gilt: } t(n) \leq c \cdot f(n)\}$$

Die [Komplexität](#) von [Algorithmen](#) kann durch Angabe einer Ordnung näherungsweise bestimmt werden. Als obere Schranke  $f$  werden dabei meist "einfache" Funktionen wie Polynome oder Logarithmen verwendet.

(2)

Eine *Ordnung* ist eine 2-stellige [Relation](#), die [antisymmetrisch](#) und [transitiv](#) ist.

Abhängig von ihren Eigenschaften lassen sich mehrere Arten von Ordnungen unterscheiden:

- [partielle Ordnung](#)

- [strenge Ordnung](#)
- [lineare Ordnung](#)

## P

[\[zum Glossar-Anfang\]](#)

### Partielle Korrektheit

(1) [Abschnitt 8, Seite 72]

Eine [Funktion](#) ist *partiell korrekt*, wenn sie der Spezifikation entspricht, d.h. alle Eingabewerte werden auf die richtigen Ergebniswerte abgebildet.

(2) [Abschnitt 7, Seite 23]

Eine [Termersetzungregel](#)  $l \rightarrow r$  eines [Termersetzungssystems](#) ist *partiell korrekt* bzgl. einer [Algebra](#)  $A$ , wenn gilt:

- $l$  und  $r$  sind [syntaktisch korrekte Terme](#) der Algebra  $A$
  - Die [Interpretation](#) beider Terme liefert für jede [Belegung](#)  $\beta$ :
- $$I_{\beta}[l] = I_{\beta}[r]$$

(3) [Abschnitt 7, Seite 24]

Ein [Termersetzungssystem](#) ist *partiell korrekt* bzgl. einer [Algebra](#)  $A$ , wenn alle [Termersetzungregeln](#) des Termersetzungssystems partiell korrekt bzgl.  $A$  sind.

### Partielle Ordnung

Eine [Ordnung](#) ist eine *partielle Ordnung*, wenn sie [reflexiv](#) ist.

### Pfad [Abschnitt 12, Seite 16]

Eine Folge von [Kanten](#)  $(e_1, \dots, e_n)$  mit  $e_i = (v_{i-1}, v_i)$  ist ein *Pfad* der Länge  $n$  in einem [Graph](#)  $G=(V,E)$  zwischen zwei [Knoten](#)  $v_0$  und  $v_n$ , wenn alle  $e_i$  in  $E$  und alle  $v_i$  in  $V$  enthalten sind.

Notation:  $v_0 \rightarrow^* v_n$

### Pivotwert [Abschnitt 11, Seite 59]

Der *Pivotwert* ist der Wert, der beim [Quicksort](#) verwendet wird, um die Elemente der zu sortierenden [Reihung](#) den jeweiligen Teilreihungen zuordnen zu können.

Um sicherzustellen, dass Quicksort [terminiert](#), darf der Pivotwert nicht kleiner als das kleinste bzw. nicht größer als das größte Element der zu sortierenden Reihung sein. Daher wird häufig ein Element der Reihung als Pivotwert ausgewählt.

### Platzkomplexität [Abschnitt 10, Seite 33]

Die *Platzkomplexität* eines [Algorithmus](#) gibt an, wieviel Speicherplatz zur Ausführung des Algorithmus bei vorgegebener Größe der Eingabedaten benötigt wird.

### Polymorphismus [Abschnitt 14, Seite 41]

In der [objekt-orientierten Programmierung](#) wird eine [Methode](#) als *polymorph* bezeichnet, wenn in verschiedenen [Klassen](#) mehrere verschiedene [Implementationen](#) derselben Methode existieren.

### Postorder [Abschnitt 12, Seite 43]

siehe [Nachordnung](#)

## Präorder [Abschnitt 12, Seite 43]

siehe [Vorordnung](#)

## Produktion [Abschnitt 4, Seite 46]

Eine *Produktion* ist eine *Regel* in einer [Chomsky-Grammatik](#).

## Programm

Ein *Programm* ist eine Beschreibung eines [Algorithmus](#), anhand derer der Algorithmus ausgeführt werden kann.

Programme werden zur [Implementierung](#) von [Informatik-Systemen](#) erstellt.

## Programmierparadigma

siehe [Programmierstil](#)

## Programmierschnittstelle [Abschnitt 2, Seite 23]

Die *Programmierschnittstelle* (engl.: *API: Application Programmer's Interface*) umfasst alle [Operationen](#), die im [detaillierten Entwurf](#) des zu entwickelnden [Systems](#) festgelegt werden.

## Programmierstil

*Programmierstile* bzw. *Programmierparadigmen* lassen sich danach unterscheiden, welche Konzepte für die Erstellung von [Programmen](#) im jeweiligen Stil wesentlich sind.

Wichtige Programmierstile sind u.a.:

- [Funktionale Programmierung](#)
- [Imperative Programmierung](#)
- [Objekt-orientierte Programmierung](#)

## Programmzustand [Abschnitt 9, Seite 31]

Der *Zustand* eines [imperativen Programms](#) zu einem bestimmten Zeitpunkt entspricht der Menge aller Daten, die das Programm zu diesem Zeitpunkt beinhaltet.

Übergänge zwischen einzelnen Programmzuständen ergeben sich während der Ausführung von [Prozeduren](#).

## Prozedur [Abschnitt 9, Seite 16]

Eine *Prozedur* ist Teil eines [imperativen Programms](#); Prozeduren werden mit Hilfe von [Prozedurdeklarationen](#) definiert.

## Prozeduraufruf [Abschnitt 9, Seite 50]

Beim *Aufruf* einer [Prozedur](#) werden die [Anweisungen](#) im [Rumpf](#) der Prozedur der Reihe nach ausgeführt.

Nach der Ausführung der aufgerufenen Prozedur wird die Ausführung des [imperativen Programms](#) fortgesetzt, indem die Anweisung unmittelbar nach dem Prozeduraufruf ausgeführt wird.

## Prozedurdeklaration [Abschnitt 9, Seite 16]

Die *Prozedurdeklaration* ist die programmiersprachliche Beschreibung einer [Prozedur](#).

Eine Prozedurdeklaration besteht aus

- einem [Prozedurkopf](#)

- einem [Prozedurrumpf](#)

### Prozedurkopf [Abschnitt 9, Seite 45]

Der *Kopf* einer [Prozedurdeklaration](#) gibt die [Funktionalität](#) der zu deklarierenden [Prozedur](#) an.

Im Gegensatz zu einer [Funktion](#) wird beim [Aufruf](#) einer Prozedur kein Ergebniswert zurückgeliefert.

### Prozedurrumpf [Abschnitt 9, Seite 45]

Der *Rumpf* einer [Prozedurdeklaration](#) besteht aus einer [Anweisungssequenz](#).

### Prozessüberwachungssystem [Abschnitt 1, Seite 37]

Ein *Prozessüberwachungssystem* ist ein [Informatik-System](#), das den Ablauf anderer [Systeme](#) überwacht.

Eingabedaten werden von anderen Systemen empfangen, Ausgabedaten werden an andere Systeme gesendet.

## Q

[\[zum Glossar-Anfang\]](#)

### Quicksort [Abschnitt 11, Seite 59]

Der *Quicksort-Algorithmus* dient zum Sortieren von [Reihungen](#) nach dem "[Divide-and-Conquer](#)"-Prinzip:

Zunächst werden die Elemente der Reihung anhand eines Vergleichs mit einem zuvor gewählten [Pivotwert](#) einer von drei Teilreihungen  $r_{<}$ ,  $r_{=}$  und  $r_{>}$  zugeordnet:

- $r_{<}$  enthält alle Elemente, die kleiner als der Pivotwert sind.
- $r_{=}$  enthält alle Elemente, die genauso groß wie der Pivotwert sind.
- $r_{>}$  enthält alle Elemente, die größer als der Pivotwert sind.

Nach der [rekursiven](#) Sortierung der Teilreihungen  $r_{<}$  und  $r_{>}$  entsprechen die verknüpften Teilreihungen  $(r_{<} \circ r_{=} \circ r_{>})$  der sortierten Reihung.

## R

[\[zum Glossar-Anfang\]](#)

### Realität [Abschnitt 1, Seite 8]

siehe [Wirklichkeit](#)

### Rechenstruktur [Abschnitt 3, Seite 32]

Eine *Rechenstruktur* (auch *Datentyp* genannt) entspricht im wesentlichen einer [Algebra](#), wobei die besondere Betonung auf dem strukturierten Aufbau der einzelnen Elemente liegt.

### Referenz [Abschnitt 10, Seite 21]

Eine *Referenz* ist ein Verweis auf ein [Objekt](#) bzw. ein beliebiges Datenelement. Die Referenz bezeichnet die Position des Objekts innerhalb des [Systems](#), zu dem das Objekt gehört, und ermöglicht den Zugriff auf das Objekt für andere [Komponenten](#) des Systems. Referenzen werden in [Variablen](#) gespeichert.

### Referenzvariable [Abschnitt 10, Seite 19]

Eine *Referenzvariable* ist eine [Variable](#), deren Wert eine [Referenz](#) ist.

## Reflexivität

Eine auf einer Menge  $M$  definierte [Relation](#)  $R$  ist *reflexiv*, wenn für alle Elemente  $a$  aus  $M$  gilt:  
 $a R a$

## Regel [Abschnitt 4, Seite 22]

Eine *Regel* in einem [Ersetzungssystem](#) besteht aus einer linken und einer rechten Seite. Bei Anwendung der Regel wird der Teil der Eingabe, dem die eine Seite der Regel entspricht, durch die andere Seite der Regel ersetzt. Bei [Textersetzungssystemen](#) ist die Ersetzungsrichtung von links nach rechts vorgegeben; bei einem [Termersetzungssystem](#) ist dagegen die Ersetzung in beide Richtungen möglich.

## Regel-Instanz [Abschnitt 7, Seite 16]

Eine *Regel-Instanz* wird aus einer [Termersetzungsregel](#) erzeugt, indem für beide Seiten der Termersetzungsregel die selbe [Substitution](#) durchgeführt wird.

## Reihung [Abschnitt 10, Seite 5]

Eine *Reihung* (engl.: *Array*) ist eine [Datenstruktur](#), bestehend aus

- einer Menge von linear angeordneten Elementen eines festen [Typs](#)
- einem [Index](#)

Der Zugriff auf ein einzelnes Element erfolgt über den [Index](#) der Reihung, indem die [Indexposition](#) des gewünschten Elements angegeben wird.

## Rekursion [Abschnitt 8, Seite 35]

Mit *Rekursion* bezeichnet man das Prinzip, ein Verfahren bzw. eine [Funktion](#) oder eine Struktur durch sich selbst zu definieren. Man spricht dann auch von [rekursiven Funktionen](#) bzw. [rekursiven Datenstrukturen](#).

Bei einer Rekursion wird das zu lösende Problem durch das Ausführen von Zerlegungsschritten so lange "vereinfacht" (reduziert), bis ein einfaches Problem übrigbleibt, das ohne weitere Reduzierung lösbar ist. Die Rekursion funktioniert damit genau entgegengesetzt zur [Induktion](#).

## Rekursive Datenstruktur

Eine *rekursive Datenstruktur* ist eine [Datenstruktur](#), deren Elemente andere Elemente der selben Datenstruktur enthalten können.

## Rekursive Funktion [Abschnitt 8, Seite 35]

Eine [Funktion](#) ist eine *rekursive Funktion*, wenn sie sich selbst [rekursiv aufruft](#), d.h. der [Funktionsrumpf](#) einer Funktion  $f$  enthält einen Aufruf der Funktion  $f$ .

Abhängig von der Anzahl bzw. dem Aufbau der rekursiven Aufrufe unterscheidet man verschiedene Arten von Rekursion:

- [Lineare Rekursion](#)
- [Kaskaden-Rekursion](#)
- [Verschränkte Rekursion](#)

**Relation [Abschnitt 4, Seite 42]**

Eine **2-stellige** (binäre) *Relation*  $R$  auf einer Menge  $M$  ist eine Teilmenge des cartesischen Produkts  $M \times M$ .

Zwei Elemente  $a$  und  $b$  aus  $M$  stehen in Relation  $R$  zueinander (Kurzschreibweise:  $a R b$ ), wenn gilt, dass das Tupel  $(a, b)$  ein Element von  $R$  ist.

Zur Ausführung dieser Zugehörigkeitsüberprüfung lässt sich eine entsprechende Operation  $R: M \times M \rightarrow \text{boolean}$  definieren.

**Repetitive Rekursion [Abschnitt 8, Seite 47]**

Eine **rekursive Funktion** ist *repetitiv rekursiv*, wenn sie **linear rekursiv** ist und der rekursive **Funktionsaufruf** bei der **Auswertung** des **Funktionsrumpfs** als letzter **Teilausdruck** ausgewertet wird.

**Repräsentation [Abschnitt 5, Seite 5]**

Unter einer *Repräsentation* versteht man die äußere Form einer **Nachricht**.

**Repräsentationssystem [Abschnitt 5, Seite 9]**

Ein *Repräsentationssystem* ist eine Menge von **Repräsentationen**.

**Rückgabe-Anweisung [Abschnitt 9, Seite 71]**

Die *Rückgabe-Anweisung* ist eine **Anweisung** zum Beenden der Ausführung einer **Prozedur**. Nachdem eine **aufgerufene** Prozedur mit einer Rückgabe-Anweisung beendet wurde, wird die Ausführung des **imperativen Programms** fortgesetzt, indem die Anweisung unmittelbar nach dem Prozeduraufruf ausgeführt wird.

**S**[\[zum Glossar-Anfang\]](#)**Schleife [Abschnitt 9, Seite 85]**

*Schleifen* (allgemein auch als *Wiederholungsanweisungen* bezeichnet) erlauben die wiederholte Ausführung von **Anweisungen**.

Eine Schleife besteht aus

- einer **Schleifeneintrittsbedingung**
- einem **Schleifenkörper**

Die Ausführung der Anweisungen im Schleifenkörper wird wiederholt, solange die Schleifeneintrittsbedingung erfüllt ist.

Ist die Anzahl der Wiederholungen bereits vor dem Ausführen der Schleife festgelegt, spricht man von einer **Zähler Schleife**.

**Schleifeneintrittsbedingung [Abschnitt 9, Seite 85]**

Eine *Schleifeneintrittsbedingung* ist eine **Bedingung**, die innerhalb einer **Schleife** erfüllt sein muss, damit die Anweisungen im **Schleifenkörper** ausgeführt werden.

**Schleifenkörper [Abschnitt 9, Seite 85]**

Ein *Schleifenkörper* ist eine Folge von **Anweisungen**, die innerhalb einer **Schleife** wiederholt ausgeführt werden.

**Schlüsselwert [Abschnitt 12, Seite 23]**

Ein *Schlüsselwert* ist ein [Wert](#), der zur Identifikation eines [Objekts](#) dient. Schlüsselwerte werden oft als Kriterium beim [Suchen](#) bestimmter Objekte verwendet, indem der Schlüsselwert des Objekts mit dem gesuchten Wert auf Gleichheit überprüft wird.

#### Selectionsort [Abschnitt 10, Seite 40]

Der *Selectionsort-Algorithmus* dient zum Sortieren von [Reihungen](#). Selectionsort funktioniert nach dem Prinzip, solange das kleinste Element aus dem unsortierten Teil der Reihung zu entnehmen und am Ende des sortierten Teils der Reihung einzufügen, bis der unsortierte Teil leer ist.

#### Semantik [Abschnitt 5, Seite 5]

Als *Semantik* einer [Nachricht](#) bezeichnet man die [Information](#), die sich durch die [Interpretation](#) der [Repräsentation](#) der Nachricht ergibt.

#### Semantische Äquivalenz [Abschnitt 5, Seite 10]

In einem [Informationssystem](#)  $(A, R, I)$  sind zwei [Repräsentationen](#)  $r_1$  und  $r_2$  aus  $R$  *semantisch äquivalent*, wenn die [Interpretation](#)  $I$  für  $r_1$  und  $r_2$  die gleiche [Information](#) aus  $A$  liefert.

Es muss also gelten:  $I[r_1] = I[r_2]$

#### Semantisches Modell [Abschnitt 5, Seite 9]

Ein *semantisches Modell* ist eine Menge von [Informationen](#).

#### Semi-Thue-System

siehe [Textersetzungssystem](#)

#### Sequentielles Suchen [Abschnitt 11, Seite 7]

Beim *sequentiellem Suchen* in einer [Reihung](#) werden die Elemente der Reihung der Reihe nach überprüft, bis das gesuchte Element gefunden oder das Ende der Reihung erreicht ist.

#### Sequenz

Eine *Sequenz* ist eine [induktiv](#) definierte [Datenstruktur](#), deren einzelne Elemente linear (sequentuell) angeordnet sind.

Die [Rechenstruktur](#) der Sequenzen definiert eine Reihe von [Operationen](#), die zum Arbeiten mit Sequenzen verwendet werden.

#### Signatur [Abschnitt 3, Seite 38]

Die *Signatur* einer [Algebra](#) umfasst die [Funktionalitäten](#) für alle [Operationen](#), die innerhalb der Algebra definiert sind.

#### Signaturgraph [Abschnitt 3, Seite 46]

Ein *Signaturgraph* ist die bildliche Darstellung einer [Signatur](#):

Die in der Signatur definierten [Sorten](#) und [Operationen](#) werden durch verschiedenartige Knoten symbolisiert. Gerichtete Kanten legen die Sorten für die Operanden (vom Sorten- zum Operationsknoten) bzw. für das Operationsergebnis (vom Operations- zum Sortenknoten) fest.

#### Sorte [Abschnitt 3, Seite 39]

Eine *Sorte* ist ein Bezeichner für eine [Trägermenge](#) innerhalb einer [Algebra](#).

Sorten werden in [abstrakten Algebren](#) zur Identifizierung der Definitionsmengen für Operanden verwendet. In [konkreten Algebren](#) können Sorten dann den entsprechenden Trägern zugeordnet werden.

### Sortierter Binärbaum [Abschnitt 12, Seite 11]

Ein [Binärbaum](#)  $b$  wird als *sortierter Binärbaum* bezeichnet, wenn für jeden [Knoten](#)  $v$  aus  $b$  gilt:

- Der [Schlüsselwert](#) des Knotens  $v$  ist größer als der Schlüsselwert jedes beliebigen Knotens aus dem linken [Unterbaum](#) von  $v$ .
- Der [Schlüsselwert](#) des Knotens  $v$  ist kleiner oder gleich dem Schlüsselwert jedes beliebigen Knotens aus dem rechten [Unterbaum](#) von  $v$ .

Wird ein sortierter Binärbaum als [Datenstruktur](#) für das [Suchen](#) von Knoteninhalten verwendet, spricht man auch von einem *binären Suchbaum*.

### Sortiertheit [Abschnitt 10]

Eine Menge von  $n$  linear angeordneten Elementen  $x_i$  ( $0 < i \leq n$ ), für die eine [partielle Ordnung](#) "kleiner-gleich" definiert ist, heißt *sortiert*, wenn für alle  $x_i, x_j$  ( $i < j$ ) gilt:

$x_i$  kleiner-gleich  $x_j$

### Spezialisierung [Abschnitt 3, Seite 18]

Bei einer *Spezialisierung* werden [Merkmale](#) einer bestehenden [Klasse](#)  $A$  an eine neue Klasse  $B$  [vererbt](#), für die zusätzlich noch weitere Merkmale festgelegt werden. Durch Spezialisierung ist der Aufbau einer [Klassenhierarchie](#) möglich.

### Spezifikation

Eine *Spezifikation* ist eine Beschreibung eines [Modells](#) der [Wirklichkeit](#).

### Spezifikationsvererbung [Abschnitt 13, Seite 39]

Bei der [Spezifikationsvererbung](#) wird nur die [Schnittstelle](#) der [Oberklasse](#) an die [Unterklasse](#) [vererbt](#).

Mit der Schnittstelle erbt die Unterklasse auch die [Funktionalität](#) der Oberklasse. [Instanzen](#) der Unterklasse können daher wie Instanzen der Oberklasse verwendet werden.

### Stack

siehe [Keller](#)

### Stapel

siehe [Keller](#)

### Statische Bindung [Abschnitt 14, Seite 42]

Bei *statischer Bindung* wird schon vor der Ausführung eines [Programms](#) für jeden [Aufruf](#) einer [polymorphen Methode](#) festgelegt, welche der zur Wahl stehenden [Implementationen](#) der Methode infolge des Aufrufs ausgeführt wird.

### Stelligkeit [Abschnitt 3, Seite 35]

Die *Stelligkeit* einer [Operation](#) gibt die Anzahl der [Operanden](#) an, die bei einem Aufruf

dieser Operation benötigt werden.

### Strenge Ordnung

Eine **Ordnung** ist eine *strenge Ordnung*, wenn sie **irreflexiv** ist.

### Striktheit [Abschnitt 8, Seite 25]

Eine **Operation** ist *strikt*, wenn gilt:

Liefert die **Auswertung** eines **Operanden** der Operation einen undefinierten Wert als Ergebnis, dann ist auch das Ergebnis der Operation selbst undefiniert.

### Strukturelle Induktion [Abschnitt 8, Seite 79]

Die *strukturelle Induktion* dient zum Beweisen von Aussagen über **induktiv** definierte **Datentypen**:

1. *Formulieren der Induktionsannahme*  
Zu zeigen: Induktionsannahme gilt für alle mit **Konstruktoroperationen** erzeugbaren Elemente des induktiv definierten Datentyps
2. *Induktionsanfang*:  
Induktionsannahme gilt für alle Elemente, die durch einmalige Anwendung einer Konstruktoroperation erzeugbar sind
3. *Induktionsschritt*:  
Induktionsannahme gilt für alle Elemente  $E(n)$ , die durch  $n$ -malige Anwendung von Konstruktoroperationen erzeugbar sind  $\implies$  Induktionsannahme gilt für alle Elemente  $E(n+1)$ , die durch einmalige Anwendung einer Konstruktoroperation auf ein Element aus  $E(n)$  erzeugbar sind

### Strukturierte Programmierung [Abschnitt 9, Seite 29]

Die *Strukturierte Programmierung* ist eine Variante der **imperativen Programmierung**, bei der keine **Anweisungen** verwendet werden dürfen, die die Strukturiertheit und damit die Lesbarkeit eines **imperativen Programms** beeinträchtigen könnten. Insbesondere sind keine Sprünge innerhalb eines **Programms** zulässig.

### Substitution [Abschnitt 7, Seite 9]

Als *Substitution* bezeichnet man das Ersetzen aller Vorkommen eines **freien Identifikators**  $x$  in einem **Term**  $t_1$  durch einen anderen Term  $t_2$ . Durch die Substitution erhält man einen neuen Term  $t_3$ .

Notation:  $t_3 = t_1 [t_2/x]$

### Subsystem [Abschnitt 2, Seite 9]

Ein *Subsystem* ist eine **Komponente** eines **Systems**, die selbst wieder ein System darstellt.

### Suchen

Das *Suchen* einzelner Elemente innerhalb einer Menge von Elementen ist neben dem **Sortieren** von Mengen eine der wichtigsten Gruppen von **Algorithmen**.

Bei einer Suche werden alle Elemente aus einer gegebenen Menge ausgewählt, die das vorgegebene Suchkriterium erfüllen. Oft wird ein **Schlüsselwert** als Suchkriterium angegeben.

### Suchen in Reihungen [Abschnitt 11]

Das *Suchen in Reihenungen* ist eine Variante des [Suchens](#), bei der die zu durchsuchende Menge als [Reihung](#) angeordnet ist.

Zum Suchen in Reihenungen bieten sich folgende [Algorithmen](#) an:

- [sequentielles Suchen](#)
- [binä res Suchen](#)

## Symmetrie

Eine auf einer Menge  $M$  definierte [Relation](#)  $R$  ist *symmetrisch*, wenn für alle Elemente  $a$  und  $b$  aus  $M$  gilt:

$$a R b ==> b R a$$

## Syntaktische Korrektheit [Abschnitt 3, Seite 42]

Die Menge aller *syntaktisch korrekten* [Terme](#) einer [Algebra](#) ist definiert als die Vereinigung

- aller in der [abstrakten Algebra](#) enthaltenen [Elementaroperanden](#)
- aller Aufrufe von [Operationen](#) der Algebra mit syntaktisch korrekten Termen als [Operanden](#). Anzahl, Reihenfolge und [Sorte](#) der einzelnen Operanden müssen dabei der [Funktionalität](#)  $t$  der Operation entsprechen.

## Syntaktische Variable [Abschnitt 4, Seite 45]

siehe [Nonterminal](#)

## Syntax [Abschnitt 6, Seite 9]

Die *Syntax* einer [Nachricht](#) beschreibt den Aufbau bzw. die Struktur der [Reprä sentation](#) der Nachricht. Die [Information](#), die die Nachricht transportiert, spielt bei der Betrachtung der Syntax keine Rolle.

## System

(1) [Abschnitt 1, Seite 31]

Ein *System* ist ein gedankliches Konstrukt zur Ausführung von Abbildungen zwischen gegebenem [Modell](#) und [Wirklichkeit](#).

(2) [Abschnitt 2, Seite 3]

Ein *System* besteht aus

- einer Menge von [Komponenten](#), die innerhalb eines gegebenen Bezugsrahmens in Zusammenhang zueinander stehen
- den Beziehungen zwischen diesen Komponenten

## Systementwurf [Abschnitt 2, Seite 23]

Beim *Systementwurf* werden die [Subsysteme](#) des zu entwickelnden [Systems](#) und deren [Schnittstellen](#) identifiziert.

## Systemgrenze [Abschnitt 2, Seite 3]

Die *Systemgrenze* stellt die Trennlinie dar zwischen

- [Komponenten](#), die zu einem [System](#) gehören
- Komponenten, die Teil der [Umgebung](#) des Systems sind

**Systemschnittstelle** [Abschnitt 2, Seite 3]

Die *Systemschnittstelle* ist die Menge aller Beziehungen zwischen den [Komponenten](#) eines [Systems](#) und den Komponenten der [System-Umgebung](#). Die Systemchnittstelle legt die Punkte fest, an denen andere Komponenten die [Systemgrenze](#) überschreiten können, um [Dienste](#) des Systems zu nutzen.

**T**[\[zum Glossar-Anfang\]](#)**Tautologie** [Abschnitt 6, Seite 32]

Eine *Tautologie* ist ein [allgemeingültiger](#) boolescher [Term](#), d.h. ein Term einer [booleschen Algebra](#).

**Teilbaum** [Abschnitt 12, Seite 19]

Wird ein beliebiger [Knoten](#) eines [Baums](#)  $b$  als [Wurzel](#) eines Baums  $t_b$  betrachtet, so nennt man  $t_b$  einen *Teilbaum* von  $b$ .

**Term** [Abschnitt 3, Seite 42]

Ein *Term* ist entweder

- ein Element aus einer Menge von [Elementaroperanden](#), oder
- die Anwendung einer [Operation](#), deren [Operanden](#) wiederum Terme sein müssen.

Die Elemente einer [Algebra](#) können durch [syntaktisch korrekte](#) Terme beschrieben werden.

**Termalgebra** [Abschnitt 3, Seite 41]

Eine *Termalgebra* ist eine [Algebra](#), die über einer Menge von [Termen](#) definiert ist und [Operationen](#) zum Aufbau von Termen bereitstellt.

Für jede [Signatur](#) lässt sich eine Termalgebra festlegen, die über der Menge aller Terme definiert ist, die sich anhand der Signatur formulieren lassen.

**Termauswertung**

Im Rahmen einer *Termauswertung* werden

- im [Term](#) vorkommende [Elementaroperanden](#) durch einen Wert (das jeweilige, ihnen zugeordnete Element der [Trägermenge](#)) ersetzt.
- im Term vorkommende Operationsanwendungen ausgeführt, d.h. die der jeweiligen [Operation](#) entsprechende [konkrete Funktion](#) aus der [konkreten Algebra](#) wird aufgerufen.

**Termersetzung** [Abschnitt 7, Seite 18]

*Termersetzung* entspricht der Umformung von [Termen](#) mit Hilfe eines [Ersetzungssystems](#).

Im Unterschied zu [Textersetzungssystemen](#) sind in Termen auch [freie Identifikatoren](#) zugelassen, die durch andere Terme [substituiert](#) werden können.

**Termersetzungsregel** [Abschnitt 7, Seite 16]

Eine *Termersetzungsregel* über einer [Signatur](#)  $\Sigma$  ist eine [Regel](#) mit [syntaktisch korrekten Termen](#) über  $\Sigma$  auf beiden Seiten.

### Termersetzungsschritt [Abschnitt 7, Seite 17]

Ein *Termersetzungsschritt* kann auf einem [Term](#)  $t$  folgendermaßen ausgeführt werden:

1. Erzeugen einer [Regel-Instanz](#) der anzuwendenden [Termersetzungsregel](#).
2. Ersetzen eines Teilterms von  $t$ , der einer Seite der Regel-Instanz entspricht, durch den Term auf der anderen Seite der Regel-Instanz.

### Termersetzungssystem [Abschnitt 7, Seite 18]

Ein *Termersetzungssystem* über einer [Signatur](#)  $\Sigma$  ist ein [Ersetzungssystem](#), dessen [Regeln](#) [Termersetzungsregeln](#) über  $\Sigma$  sind.

### Terminal [Abschnitt 4, Seite 45]

Ein *Terminal* ist ein Zeichen aus dem [Zeichenvorrat](#) einer [Chomsky-Grammatik](#). Im Gegensatz zu einem [Nonterminal](#) ist einem Terminal keine syntaktische Bedeutung zugeordnet.

### Terminaler Term [Abschnitt 7, Seite 18]

Ein *terminaler Term* ist ein [Term](#), der nicht weiter [abgeleitet](#) werden kann, d.h. es kann kein weiterer [Termersetzungsschritt](#) mehr ausgeführt werden (Annahme: Alle [Termersetzungsregeln](#) werden nur in einer Richtung angewendet).

### Terminierende Berechnung [Abschnitt 7, Seite 18]

Eine [Berechnung](#), die als Ergebnis einen [terminalen Term](#) liefert, ist eine *terminierende Berechnung*.

### Terminierung [Abschnitt 4, Seite 6]

Als *Terminierung* bezeichnet man das Beenden der Ausführung eines [Algorithmus](#) nach einer endlichen Zahl von Arbeitsschritten.

### Terminierungsnachweis [Abschnitt 8, Seite 59]

Der *Nachweis* der [Terminierung](#) von [rekursiven Funktionen](#) wird oft mit Hilfe einer [Abstiegsfunktion](#) geführt:

Parallel zu jedem [Aufruf](#) der rekursiven Funktion wird die Abstiegsfunktion aufgerufen. Die Terminierung einer rekursiven Funktion wird dann nachgewiesen, indem man zeigt, dass die Ergebnisse der Abstiegsfunktion mit jedem Rekursionsschritt kleiner werden; da gleichzeitig der Wertebereich der Abstiegsfunktion nach unten beschränkt ist, ist die Anzahl der ausführbaren Rekursionsschritte endlich.

### Tertium non datur [Abschnitt 6, Seite 54]

*Tertium non datur* besagt:

Für eine [Aussage](#)  $x$  gilt, dass entweder  $x$  oder NOT  $x$  wahr sein muss.

Tertium non datur ist eine der für die [Aussagenlogik](#) definierten [Inferenzregeln](#).

### Textersetzungssystem [Abschnitt 4, Seite 23]

Ein *Textersetzungssystem* (auch *Semi-Thue-System* genannt) ist ein [Ersetzungssystem](#), dessen [Regeln](#) über einem gegebenen [Zeichenvorrat](#) definiert sind.

Zusätzlich gelten folgende [Meta-Regeln](#):

- Aus der Menge der anwendbaren Regeln wird eine beliebige Regel ausgewählt

und angewendet. Ist die gewählte Regel an mehreren Stellen anwendbar, wird eine beliebige dieser Stellen als Anwendungsposition ausgewählt.

- Das System **terminiert**, wenn keine Regel mehr anwendbar ist.

Notation: *linke Seite* --> *rechte Seite*

### Theorem [Abschnitt 6, Seite 46]

Ein *Theorem* ist eine **Formel** innerhalb einer **Logik**, die entweder ein **Axiom** der Logik ist oder **formal bewiesen** werden kann.

### Totale Korrektheit [Abschnitt 7, Seite 25]

Ein **Termersetzungssystem** ist *total korrekt*, wenn es **partiell korrekt** ist und alle **Berechnungen terminierende Berechnungen** sind.

### Totale Ordnung

siehe **lineare Ordnung**

### Trägermenge [Abschnitt 3, Seite 38]

Eine *Trägermenge* ist eine Menge von Elementen einer **Algebra**.

### Transitivität

Eine auf einer Menge  $M$  definierte **Relation**  $R$  ist *transitiv*, wenn für alle Elemente  $a, b$  und  $c$  aus  $M$  gilt:

$$a R b \text{ und } b R c \implies a R c$$

### Traversierung [Abschnitt 12, Seite 43]

Als *Traversierung* eines **Baums** bezeichnet man das vollständige Durchlaufen aller **Knoten** des Baums.

Bzgl. der Reihenfolge des Durchlaufs werden bei **Binären Bäumen** folgende Varianten der Traversierung unterschieden:

- **Vorordnung**
- **Inordnung**
- **Nachordnung**

### Typ [Abschnitt 8, Seite 16]

siehe **Datentyp**

## Ü

[\[zum Glossar-Anfang\]](#)

### Überschreiben von Methoden [Abschnitt 13, Seite 49]

Eine in einer **Oberklasse**  $A$  definierte **Methode** kann in einer **Unterklasse** von  $A$  *überschrieben* werden, indem die von  $A$  **geerbte Implementation** der Methode durch eine andere Implementation ersetzt wird.

Durch das Überschreiben wird die Methode **polymorph**.

## U

[\[zum Glossar-Anfang\]](#)

### Umgebung [Abschnitt 2, Seite 3]

Die *Umgebung* eines [Systems](#) ist die Menge aller [Komponenten](#) außerhalb des Systems.

### Ungerichtete Kante [Abschnitt 12]

Eine *ungerichtete Kante*  $e=(v_1,v_2)$  beschreibt eine in beiden Richtungen bestehende Verbindung zwischen zwei [Knoten](#)  $v_1$  und  $v_2$  in einem [Graph](#).

Notation:  $v_1 \text{ --- } v_2$  bzw.  $v_1 \text{ <-> } v_2$

### Ungerichteter Baum [Abschnitt 12, Seite 16]

Ein *ungerichteter Baum* ist ein [ungerichteter Graph](#) ohne [Zyklen](#), für den gilt, dass alle [Knoten](#) paarweise durch einen [Pfad](#) miteinander verbunden sind.

### Ungerichteter Graph [Abschnitt 12, Seite 16]

Ein *ungerichteter Graph* ist ein [Graph](#), für den gilt, dass all seine [Kanten ungerichtete Kanten](#) sind.

Die Kantenmenge eines ungerichteten Graphen entspricht einer [symmetrischen Relation](#).

### Unterbaum [Abschnitt 12]

Gegeben sei ein [Knoten](#)  $v$  in einem [Baum](#)  $b$ .

Jeder [Teilbaum](#) von  $b$ , dessen [Wurzel](#) ein [Kindknoten](#) von  $v$  ist, ist ein *Unterbaum* von  $v$ .

### Unterklasse [Abschnitt 3, Seite 18]

Eine *Unterklasse* [erbt](#) alle [Merkmale](#) ihrer [Oberklassen](#). Darüber hinaus können beliebige weitere Merkmale für die Unterklasse vereinbart werden.

## V

[\[zum Glossar-Anfang\]](#)

### Validation [Abschnitt 2, Seite 29]

Bei einer *Validation* wird der Wahrheitsgehalt eines [Modells](#) gegenüber der [Wirklichkeit](#), die zu modellieren ist, überprüft.

### Variable [Abschnitt 9, Seite 32]

In der [imperativen Programmierung](#) ermöglichen *Variablen* den Zugriff auf Daten durch Angabe eines Namens.

Eine Variable besteht dabei aus

- dem Bezeichner bzw. Namen der Variablen, über den die Variable eindeutig identifizierbar ist
- dem Wert der Variablen

In [Ausdrücken](#) entsprechen Variablen [Elementaroperanden](#), d.h. bei der [Auswertung](#) eines Ausdrucks wird für den Bezeichner einer im Ausdruck verwendeten Variable der Wert [substituiert](#), den diese Variable zum Auswertungszeitpunkt hat.

Darf eine Variable nur Werte eines bestimmten [Typs](#) annehmen, so muss dieser bei der [Deklaration](#) der Variable festgelegt werden.

### Variablendeklaration [Abschnitt 9]

Bei der *Deklaration* einer [Variable](#) wird ein Bezeichner an eine Variable [gebunden](#). Ab dem Zeitpunkt der Deklaration kann die Variable über diesen Bezeichner eindeutig identifiziert werden.

Darf eine Variable nur Werte eines bestimmten **Typs** annehmen, muss bei der Deklaration auch der Variablentyp festgelegt werden.

## Vererbung

(1) [Abschnitt 3, Seite 18]

Die Übernahme aller **Merkmale** einer **Klasse** A bei der Definition einer anderen Klasse B wird als *Vererbung* bezeichnet. B ist dann eine **Unterklasse** der **Oberklasse** A.

Betrachtet man Klassen als Mengen von **Objekten**, so gehören alle **Instanzen** einer Unterklasse auch der jeweiligen Oberklasse an. B ist somit eine Teilmenge von A.

(2) [Abschnitt 13, Seite 46]

Die *Vererbung* im Rahmen der **objekt-orientierten Programmierung** erlaubt die **Implementierung** von **Vererbungsbeziehungen**, die während des **detaillierten Entwurfs** festgelegt werden.

Man unterscheidet

- **Implementationsvererbung**
- **Spezifikationsvererbung**

Vererbung ermöglicht die Verwendung bereits existierender **Klassen** als Basis für die Erstellung neuer Klassen und unterstützt damit die **Wiederverwendbarkeit** von Klassen während der Entwicklung von Programmen.

## Verifikation [Abschnitt 2, Seite 29]

Bei einer *Verifikation* wird der Wahrheitsgehalt eines **Modells** gegenüber der vorgegebenen **Spezifikation** überprüft.

## Verkettete Liste [Abschnitt 11, Seite 25]

Eine *verkettete Liste* ist eine **rekursive Datenstruktur**. Die Elemente einer Liste enthalten

- einen Wert (den Inhalt des Elements)
- eine **Referenz** auf das nächste Element der Liste

Verkettete Listen sind wie **Reihungen Datenstrukturen** mit linear angeordneten Elementen. **Algorithmen**, die auf Reihungen arbeiten, können daher auch für verkettete Listen verwendet werden. Verkettete Listen haben aber im Gegensatz zu Reihungen keinen **Index**, d.h. der Zugriff auf Elemente anhand einer Positionsangabe ist bei verketteten Listen weniger **effizient** als bei Reihungen.

## Verschattung [Abschnitt 13, Seite 66]

Wird ein bereits **gebundener** Bezeichner erneut gebunden, ohne die alte Bindung dadurch aufzulösen, so nennt man das die *Verschattung* der alten Bindung.

Während der Verschattung existiert die alte Bindung weiterhin, ist aber außer Kraft gesetzt, d.h. für den Bezeichner gilt die in der neuen Bindung festgelegte Zuordnung.

## Verschränkte Rekursion [Abschnitt 8, Seite 51]

Zwei oder mehr **rekursive Funktionen** sind *verschränkt rekursiv*, wenn sie sich gegenseitig rekursiv **aufrufen**.

## Vielfachheit [Abschnitt 2, Seite 8]

siehe [Kardinalität](#)

### Vollständige Induktion [Abschnitt 8, Seite 66]

*Vollständige Induktion* ist ein Beweisverfahren, das nach dem Prinzip der [Induktion](#) arbeitet:

1. *Formulieren der Induktionsannahme*  
Zu zeigen: Induktionsannahme gilt für alle  $n \geq n_0$
2. *Induktionsanfang:*  
Induktionsannahme gilt für  $n = n_0$
3. *Induktionsschritt:*  
Induktionsannahme gilt für  $n \implies$  Induktionsannahme gilt für  $n + 1$

### Vollständigkeit [Abschnitt 6, Seite 53]

Eine [Logik](#) ist *vollständig* (engl.: *complete*), wenn jede ihrer [allgemeingültigen Formeln](#) ein [Theorem](#) ist.

### Vorordnung [Abschnitt 12, Seite 43]

Gegeben sei ein [Binä rbaum](#)  $b$ .

Eine [Traversierung](#) von  $b$  erfolgt in *Vorordnung* (*Präorder*), wenn für jeden [Knoten](#)  $k$  aus  $b$  folgende Durchlaufreihenfolge eingehalten wird:

1.  $k$
2. Linker [Unterbaum](#) von  $k$
3. Rechter [Unterbaum](#) von  $k$

## W

[\[zum Glossar-Anfang\]](#)

### Wahrheitswert [Abschnitt 5, Seite 18]

Die Menge der *Wahrheitswerte* ist für die [boolesche Algebra](#) definiert als  $\{0, 1\}$ .

### White box reuse [Abschnitt 13, Seite 38]

Als *White box reuse* bezeichnet man die [Wiederverwendung](#) von [Systemen](#), von denen die [Schnittstellen](#) und die interne Struktur bekannt sind.

### Wiederholungsanweisung [Abschnitt 9, Seite 85]

siehe [Schleife](#)

### Wiederverwendbarkeit [Abschnitt 13, Seite 38]

Wenn eine existierende [Komponente](#) bzw. ein [Subsystem](#) bei der Erstellung mehrerer [Systeme](#) benutzt werden kann, so ist die Komponente bzw. das Subsystem *wiederverwendbar* (engl.: *reusable*).

Man unterscheidet

- [Black box reuse](#)
- [White box reuse](#)

Der Einsatz wiederverwendbarer Komponenten kann sowohl den [detaillierten Entwurf](#) als auch die [Implementierung](#) von Systemen deutlich vereinfachen.

## Wirklichkeit [Abschnitt 1, Seite 8]

Unter einer *Wirklichkeit* versteht man die Kombination aus

- einer Menge physikalisch existierender und/oder gedachter (fiktiver) "Gegenstände" (Dinge, Personen, Abläufe in der Zeit)
- Beziehungen zwischen diesen Gegenständen

Wirklichkeiten können somit sowohl physikalischer als auch fiktiver Natur sein.

## Worst-Case-Komplexität [Abschnitt 10, Seite 34]

Die *Worst-Case-Komplexität* eines [Algorithmus](#) gibt die maximalen "Kosten" an, die bei der Ausführung des Algorithmus entstehen, beschreibt also die [Komplexität](#) im ungünstigsten Fall.

## Wort [Abschnitt 4, Seite 20]

Ein *Wort* ist eine [Sequenz](#) von Zeichen.

Das leere Wort, also eine Sequenz mit 0 Zeichen, wird mit  $\epsilon$  (bzw. dem entsprechenden Symbol) bezeichnet.

## Wurzel [Abschnitt 12, Seite 17]

Die *Wurzel* eines [Baums](#) ist ein speziell ausgezeichnete [Knoten](#), der mit allen anderen Knoten des Baums durch [Pfade](#) verbunden ist.

## Z

[\[zum Glossar-Anfang\]](#)

## Zählschleife [Abschnitt 9, Seite 85]

Eine *Zählschleife* ist eine [Schleife](#), bei der die Anzahl der Wiederholungen der Ausführung des [Schleifenkörpers](#) bereits vor der Ausführung der Schleife festgelegt ist. Die [Schleifeneintrittsbedingung](#) ist in diesem Fall erfüllt, solange die geforderte Anzahl an Wiederholungen noch nicht erreicht wurde.

## Zeichenvorrat [Abschnitt 4, Seite 22]

Ein *Zeichenvorrat*  $\Sigma$  ist eine Menge von Zeichen.

## Zeitkomplexität [Abschnitt 10, Seite 33]

Die *Zeitkomplexität* eines [Algorithmus](#) gibt an, wieviele Schritte zur Ausführung des Algorithmus bei vorgegebener Größe der Eingabedaten benötigt werden.

## Zustand [Abschnitt 3, Seite 5]

Der *Zustand* eines [Objekts](#) setzt sich zusammen aus Teilzuständen d.h. den Werten der [Attribute](#) dieses Objekts. Die Änderung eines Attributwerts bewirkt somit eine Änderung des Objektzustands.

## Zuweisung [Abschnitt 9, Seite 34]

Eine *Zuweisung* ist eine [Anweisung](#), die einer [Variablen](#) einen Wert zuordnet. Dieser Wert ist das Ergebnis der [Auswertung](#) eines [Ausdrucks](#), die zu Beginn der Zuweisung stattfindet.

## Zyklus [Abschnitt 12, Seite 16]

Ein *Zyklus* ist ein [Pfad](#) mit einer Länge von mindestens 1, dessen Anfangs- und

**Endknoten** identisch sind. Es gilt also:

$$v_0 \xrightarrow{*} v_n \text{ mit } v_0 = v_n.$$

---

[Clemens Harlfinger](#) - 2001-04-06