# Syntaxdefinitionen

- Wie bereits erwähnt lässt sich die Gestalt einer Datendarstellung (Syntax) formal definieren unabhängig von deren Bedeutung (Semantik).
- Gleiches gilt für eine gesamte (Programmier-)sprache, d.h. z.B. deren Elemente.
- Programmiersprachen sind ein bisschen so wie Fremdsprachen:
  - Es gibt Wörter der Sprache
  - Es gibt Grammatik-Regeln, wie aus Wörtern “Sätze” gebildet werden können
- Die (Gestalt der) erlaubten Wörter und Regeln lassen sich formal definieren ohne deren Bedeutung zu kennen.

# Syntaxdefinition

- Eine formale, häufig gebrauchte Form für die Definition von Syntax (Syntaxdefinitionen) ist z.B. die *Backus-Naur-Form (BNF)*, die wir hier aber nicht genauer besprechen.
- Dennoch soll hier nochmal betont werden, dass eine formale Definition der Syntax von Daten und Programmiersprachen eine wichtige Grundlage fürs Programmieren darstellt.
- Wir werden im Laufe der Vorlesung auch immer wieder auf Syntaxdefinitionen zurückgreifen.
- Daher im folgenden ein kurzer Blick auf Syntaxdefinitionen, wir bleiben aber informell.

## Beispiel

Zunächst ein Beispiel aus dem alltäglichen Leben:

⟨***Essensbeschreibung***⟩ ***::= In der Mensa gab es diese Woche am*** ⟨***WOCHENTAG***⟩ ⟨***ESSEN***⟩ ***zum*** ⟨***MAHLZEIT***⟩***.***

Diese Zeile repräsentiert die Syntax eines Satzes “Essensbeschreibung” in deutscher Sprache. Die Worte ⟨*WOCHENTAG*⟩, ⟨*ESSEN*⟩ und ⟨*MAHLZEIT*⟩ stellen dabei Platzhalter dar, für die gewisse Regeln gelten:

- ⟨*WOCHENTAG*⟩ kann ein beliebiger Wochentag sein.
- ⟨*ESSEN*⟩ kann der Name eines beliebigen Essens sein.
- ⟨*MAHLZEIT*⟩ muss eines der Worte “Frühstück”, “Mittagessen”, oder “Abendessen” sein.

# Syntaxdefinition

- Mit diesem Regelwerk können wir nun gültige Sätze bilden, indem wir für die Platzhalter entsprechend gültige “Werte” einsetzen, z.B.
  - In der Mensa gab es diese Woche am Montag Pommes zum Mittagessen.
  - In der Mensa gab es diese Woche am Dienstag Himbeereis zum Frühstück.
  - In der Mensa gab es diese Woche am Samstag Schinkennudeln zum Abendessen.

# Syntaxdefinition

- Alle Sätze, die diesem Regelwerk nicht entsprechen, sind keine gültigen Sätze, z.B.
  - In der Mensa gab es diese Woche am heutigen Tag Pommes zum Mittagessen.
  - In der Mensa gab es diese Woche am Dienstag Himbeereis zum Breakfast.
  - In der Mensa gab es diese Woche am Samstag etwas ganz Ekelhaftes zum Abendessen.
  - In der Mensa hats am Samstag Schinkennudeln zum Abendessen gegeben.
  - In der Mensa gab es diese Woche am Samstag Schinkennudeln zum Abendessen
    Was ist hier das Problem?!?!?!

# Syntaxdefinition

- In unserem Kontext interessieren wir uns natürlich nicht für gültige Sätze der Umgangssprache (wie z.B. Essensbeschreibung), sondern für die Definition von gültigen Datendarstellungen sowie gültigen Programmtexten.
- Man kann nämlich Syntaxregeln für beides verwenden, die Definition von Datendarstellungen (das ist, was wir jetzt machen) aber eben auch für die Definition der Syntax von Programmiersprachen (das machen wir später).

# Syntaxdefinition

- Allgemein enthalten Syntaxdefintionen verschiedene Konstrukte, die wir informell schon kennen gelernt haben:
  - Ein Alphabet spezifiziert die Zeichen (manchmal auch ganze Worte), die verwendet werden dürfen. Die Zeichen des vorgegeben Alphabets heißen *Terminalzeichen*.
  - Eine Menge von *syntaktische Variablen* kennzeichnen den zu definierenden Begriff sowie (beliebig aber endlich viele) Hilfsbegriffe. Syntaktische Variablen kennzeichnen wir durch $\langle\ \rangle$-Klammern.
  - Eine Menge von (Syntax-)Regeln zur “Erzeugung” von syntaktisch validen Elementen.

# Syntaxdefinition

- In unserem Beispiel von vorhin

  ⟨***Essensbeschreibung***⟩ ***::= In der Mensa gab es diese Woche am*** ⟨***WOCHENTAG***⟩ ⟨***ESSEN***⟩ ***zum*** ⟨***MAHLZEIT***⟩***.***

  sind z.B.:

  - "In", "der", "Mensa", ... und "." *Terminalzeichen*.
  - ⟨*Essensbeschreibung*⟩, ⟨*WOCHENTAG*⟩, ⟨*ESSEN*⟩ und ⟨*MAHLZEIT*⟩ sind die *syntaktische Variablen*; ⟨*Essensbeschreibung*⟩ kennzeichnet den zu definierenden Begriff, ⟨*WOCHENTAG*⟩, ⟨*ESSEN*⟩ und ⟨*MAHLZEIT*⟩ sind Hilfsbegriffe.
  - Die einzige formale Regel definiert (mit dem Zeichen "::=") den Begriff ⟨*Essensbeschreibung*⟩. Tatsächlich werden aber drei weitere (informelle) Regeln verwendet, um die Hilfsbegriffe zu definieren.

# Syntaxdefinition

- Eine Regel hat also immer die Form $A ::= B$.
- $A ::= B$ bedeutet dabei informell, dass $A$ durch $B$ definiert ist.
- Wir sagen auch: “$A$ kann durch $B$ ersetzt werden”.
- $A$ muss dabei eine syntaktische Variable sein und $B$ muss eine “Kombination” (dazu gleich mehr) aus syntaktischen Variablen und/oder Terminalzeichen sein.
- Die Anwendung der Regeln (d.h. das sukzessive Ersetzen der linken durch die rechte Seite) zur Erzeugung eines Elements heißt *Ableitung*.

# Syntaxdefinition

- Genau genommen sollten wir auch die drei informellen Regeln für die Hilfsbegriffe formal aufschreiben (mit Terminalzeichen und ggfls. Variablen), in der Form $\langle WOCHENTAG \rangle$ := ...
- Dabei benötigen wir noch weitere Konstrukte um z.B. die "Alternative" auszudrücken ($\langle WOCHENTAG \rangle$ ist ja z.B. Montag *oder* Dienstag *oder* ...), bzw. ganz allgemein: die Kombination von Variablen und Terminalzeichen auf der rechten Seite einer Regel.

**Definition (Syntaxdefinition, Syntaxregel)**

*Eine Syntaxdefinition besteht aus einer endlichen Menge von Syntaxregeln der Form A ::= B. Dabei ist A eine syntaktische Variable und B eine Satzform, deren Gestalt (Syntax) wie folgt induktiv definiert:*

- *jedes Terminalzeichen oder jede syntaktische Variable ist eine Satzform*
- *sind $B_1$ und $B_2$ Satzformen, so ist auch $B_1 B_2$ eine Satzform*
- *sind $B_1$ und $B_2$ Satzformen, so ist auch $B_1 \mid B_2$ eine Satzform*
- *ist C eine Satzform, so ist auch $\{C\}^*$, $\{C\}^+$ und $\{C\}_0^1$ eine Satzform*

# Syntaxregeln

- Diese Definition spezifiziert den Aufbau, also die Syntax.
- Die Bedeutung (Semantik) dieser Notationen ist informell:
  Sei $A ::= B$ eine Syntaxregel mit
  - ist $B$ ein Terminalzeichen oder eine syntaktische Variable wird $A$ durch $B$ ersetzt.
  - hat $B$ die Form $B_1 B_2$, so wird $A$ durch $B_1 B_2$ ersetzt.
  - hat $B$ die Form $B_1 \mid B_2$, so wird $A$ entweder durch $B_1$ oder durch $B_2$ ersetzt.
  - hat $B$ hat die Form $\{C\}^*$, $\{C\}^+$ oder $\{C\}_0^1$, so wird $A$ durch beliebig viele $C$'s (Variante $\{\}^*$), beliebig viele aber mind. einem $C$ (Variante $\{\}^+$) oder durch genau einmal oder keinmal $C$ (Variante $\{\}_0^1$) ersetzt.

# Bsp.: Syntaxdefinition von natürlichen Zahlen

**Beispiel**

Die Menge der natürlichen Zahlen in Dezimaldarstellung kann durch folgende Regeln definiert werden:

1. 0 ist eine $\langle \textit{Dezimalzahl} \rangle$.
2. Jede Ziffer $x \in \mathcal{A}_{10} \setminus \{0\}$ ist eine $\langle \textit{Nichtnulldarstellung} \rangle$.
3. Ist $a$ eine $\langle \textit{Nichtnulldarstellung} \rangle$ und $y \in \mathcal{A}_{10}$ so ist $a \circ y$ eine $\langle \textit{Nichtnulldarstellung} \rangle$.
4. Jede $\langle \textit{Nichtnulldarstellung} \rangle$ ist eine $\langle \textit{Dezimalzahl} \rangle$.

# Bsp.: Syntaxdefinition von natürlichen Zahlen

Die entsprechenden formalen Regeln:

1. $\langle \textit{NichtNullZiffer} \rangle$ ::= 1|2|3|4|5|6|7|8|9
2. $\langle \textit{Ziffer} \rangle$ ::= 0 | $\langle \textit{NichtNullZiffer} \rangle$
3. $\langle \textit{NichtNullDarst} \rangle$::= $\langle \textit{NichtNullZiffer} \rangle$ | $\langle \textit{NichtNullDarst} \rangle \langle \textit{Ziffer} \rangle$
4. $\langle \textit{Dezimalzahl} \rangle$ ::= 0 | $\langle \textit{NichtNullDarst} \rangle$

oder alternativ etwas kompakter:

**Syntaxregel** (Dezimaldarstellung natürlicher Zahlen)

$\langle \textit{NichtNullZiffer} \rangle$ ::= 1|2|3|4|5|6|7|8|9

$\langle \textit{Ziffer} \rangle$ ::= 0 | $\langle \textit{NichtNullZiffer} \rangle$

$\langle \textit{Dezimalzahl} \rangle$ ::= 0 | $\langle \textit{NichtNullZiffer} \rangle$ $\{\langle \textit{Ziffer} \rangle\}^*$

# Beispiele

- Die Zeichenreihe 308 ist eine Dezimalzahl gemäß folgender Ableitung, d.h. Anwendungen der Regeln:

$$\begin{aligned}\langle \textit{Dezimalzahl} \rangle &\rightarrow \langle \textit{NichtNullZiffer} \rangle \ \{\langle \textit{Ziffer} \rangle\}^* \\ &\rightarrow \langle \textit{NichtNullZiffer} \rangle \ (0 \mid \langle \textit{NichtNullZiffer} \rangle)(0 \mid \langle \textit{NichtNullZiffer} \rangle) \\ &\rightarrow \langle \textit{NichtNullZiffer} \rangle \ 0 \ \langle \textit{NichtNullZiffer} \rangle \\ &\rightarrow 308\end{aligned}$$

- Grundsätzlich sind alle Zeichenreichen ableitbar, die durch endliche Anzahl von sukzessiven Ersetzungen entstehen.
- Alle diese Zeichenreihen, die aus einer syntaktischen Variable $A$ ableitbar sind werden als $\mathcal{L}(A)$ bezeichnet.
- Es gilt z.B. $308 \in \mathcal{L}(\langle \textit{Dezimalzahl} \rangle)$.

# Weitere Beispiele

Auf Basis der vorherigen Definitionen lassen sich auch ganze Zahlen und Gleitpunktzahlen definieren:

**Syntaxregel** (Dezimaldarstellung ganzer Zahlen)

$$\langle \textit{GanzeZahl} \rangle ::= \{-\}_0^1 \ \langle \textit{Dezimalzahl} \rangle$$

**Syntaxregel** (Dezimaldarstellung Gleitpunktzahlen)

$$\langle \textit{GleitpunktZahl} \rangle ::= \{-\}_0^1 \ \langle \textit{Mantisse} \rangle \ \{\mathrm{E} \langle \textit{GanzeZahl} \rangle\}_0^1$$

$$\langle \textit{Mantisse} \rangle ::= \langle \textit{Dezimalzahl} \rangle . \langle \textit{Dezimalstellen} \rangle$$

$$\langle \textit{Dezimalstellen} \rangle ::= 0 \mid \langle \textit{NichtNullStellen} \rangle$$

$$\langle \textit{NichtNullStellen} \rangle ::= 0 \mid \{\langle \textit{NichtNullZiffer} \rangle\}_0^1 \langle \textit{NichtNullZiffer} \rangle$$

# Weitere Beispiele

- Wir können aber wie erwähnt nicht nur die Syntax von Daten eindeutig definieren, sondern auch die Syntax von Programmiersprachen.
- Folgende Regeln definieren den Begriff *Identifikator*; Identifikatoren nutzen wir z.B. später als Variablennamen:

**Syntaxregel** (Identifikator)

⟨*Identifikator*⟩ ::= ⟨*Buchstabe*⟩ {⟨*Buchstabe*⟩ | ⟨*Ziffer*⟩}*

⟨*Buchstabe*⟩ ::= A|B|C| ... |Z|a|b|c|...|z

# Funktionaler Lösungsansatz

- Beispielanwendung $r = 81$:

$$DIV(100 - r, 5) = DIV(19, 5) = 3$$
$$MOD(100 - r, 5) = MOD(19, 5) = 4$$
$$DIV(MOD(100 - r, 5), 2) = DIV(4, 2) = 2$$
$$MOD(MOD(100 - r, 5), 2) = MOD(4, 2) = 0$$

d.h. man erhält $h(81) = (5_1, 5_2, 5_3, 2_1, 2_2, 1_1, 1_0)$ oder, einfacher geschrieben,

$$h(81) = (5, 5, 5, 2, 2).$$

# Funktionaler Lösungsansatz

- Eigenschaften:
  - Wenn man von der Bildung der Folgen aus den berechneten Anzahlen der Elemente 1, 2 und 5 absieht, sind nur die Auswertungen der Operationen „$-$“, *DIV* und *MOD* als Einzelschritte anzusehen. Setzt man diese als elementar voraus, ist die angegebene Definition von *h* eine Beschreibung des gesuchten Algorithmus.
  - Diese Darstellung nimmt keinen Bezug mehr auf eine „Abfolge von Aktionen“ (d.h. wie das Ergebnis „berechnet“ werden soll) sondern beschreibt den „funktionalen Zusammenhang“ zwischen *r* und *w* durch ineinander eingesetzte Anwendungen (Komposition) gewisser Basisoperationen (und damit wie das Ergebnis „aussehen“ soll).

# Variante des funktionalen Lösungsansatzes

- Leider lässt sich in vielen Fällen die gesuchte Abbildung nicht in derartig einfacher Weise explizit angeben.
- Ein wichtiges Prinzip für den Allgemeinfall von Abbildungen auf natürlichen Zahlen (und anderen v.a. induktiv definierten Mengen) ist das Prinzip der Rekursion, das wir im vorigen Kapitel kennengelernt haben.

# Variante des funktionalen Lösungsansatzes

## Algorithmus 5 (Wechselgeld 5)

$$h(r) = \begin{cases} \text{falls } r = 100, \text{ dann} \quad (), & (1) \\ \text{falls } 100 - r \geq 5, \text{ dann} \quad (5) \circ h(r+5), & (2) \\ \text{falls } 5 > 100 - r \geq 2, \text{ dann} \quad (2) \circ h(r+2), & (3) \\ \text{falls } 2 > 100 - r \geq 1, \text{ dann} \quad (1) \circ h(r+1), & (4) \end{cases}$$

Hier ist der Basisfall (1) für $h(100)$ definiert und die Rekursionsfälle (2),(3),(4) werden auf die Fälle $h(r+5)$, $h(r+2)$ und $h(r+1)$ zurückgeführt.

# Variante des funktionalen Lösungsansatzes

- Beispielanwendung $r = 81$:

$$
\begin{aligned}
h(81) &= (5) \circ h(86) && \text{(Fall 2)} \\
&= (5) \circ (5) \circ h(91) && \text{(Fall 2)} \\
&= (5) \circ (5) \circ (5) \circ h(96) && \text{(Fall 2)} \\
&= (5) \circ (5) \circ (5) \circ (2) \circ h(98) && \text{(Fall 3)} \\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ h(100) && \text{(Fall 3)} \\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ () && \text{(Fall 1)} \\
&= (5, 5, 5, 2, 2)
\end{aligned}
$$

# Variante des funktionalen Lösungsansatzes

- Trotz der rein funktionalen Darstellung der Zuordnung $r \mapsto w$ erinnert die Auswertung wieder an die Abfolge der Aktionen im entsprechenden Ablaufbeispiel (ähnlich wie in Algorithmus 3).
- Tatsächlich kann das Prinzip der Rekursion auch in der operativen Auffassung der Algorithmen 1-3 eingerichtet werden und dabei einen ähnlichen Effekt wie die Iteration erzielen.

# Variante des funktionalen Lösungsansatzes

- In einer rekursiven Abbildungsdefinition greift man auf Werte dieser Abbildung für kleinere (oder hier: größere) Argumente zurück.
- Analog kann in einem Algorithmus, der die Abfolge gewisser Aktionen in Abhängigkeit von Eingabewerten steuert, die Ausführung des Algorithmus selbst auch als eine derartige Aktion auftreten.

## Algorithmus 6 (Wechselgeld 6)

**Eingabe**: $r \in \mathbb{N}$

Setze $w = ()$

Führe den in Abhängigkeit von $r$ rekursiv definierten Algorithmus $A(r)$ aus:

$A(r)$:

Führe denjenigen der folgenden Schritte (1) - (3) aus, dessen Bedingung erfüllt ist (ist keine Bedingung erfüllt, ist die Ausführung zu Ende):

(1) Falls $100 - r \geq 5$, dann nimm 5 zu $w$ hinzu
und führe anschließend $A(r + 5)$ aus.

(2) Falls $5 > 100 - r \geq 2$, dann nimm 2 zu $w$ hinzu
und führe anschließend $A(r + 2)$ aus.

(3) Falls $5 > 100 - r \geq 1$, dann nimm 1 zu $w$ hinzu
und führe anschließend $A(r + 1)$ aus.

**Ausgabe**: $w$

# Algorithmische Konzepte

- In den vergangenenen Abschnitten haben wir einige grundlegende algorithmische Konzepte kennengelernt:
  - Eingabe- und Ergebnisdaten (insbesondere solche von komplexer Art) müssen geeignet dargestellt werden.
  - Die Ausführung gewisser Aktionen bzw. die Auswertung gewisser Operationen wird als in elementaren Schritten durchführbar vorausgesetzt.
  - Einzelne Schritte können zusammengesetzt werden (z.B. Nacheinanderausführung, Auswahl von Aktionen, Kompositionen von Abbildungen, etc.)
  - Fallunterscheidung, Iteration und Rekursion ermöglichen die Steuerung des durch den Algorithmus beschriebenen Verfahrens.

# Algorithmische Konzepte

- Diese Konzepte sind z.T. abhängig vom verwendeten Programmierparadigma.
- Zum Abschluss dieses Kapitels diskutieren wir daher nochmal die verschiedenen Paradigmen:
  - Imperatives (prozedurales) Programmieren
  - Funktionales Programmieren
  - Objektorientiertes Programmieren
  - Logik-Programmieren

# Historische Entwicklung

- Das “ursprüngliche” Paradigma
- Abgeleitet aus der Architektur der Computer (insbs. der zugrundeliegenden Befehlssteuerung: binär codierte Befehle, direkte Adressierung von Speicherzellen)
- Abstraktion von der Maschinenorientierung durch Variablenkonzept (statt binäre Speicheradressierung) und Rekursivität der Programmsteuerung (Programme können durch andere Programme verändert werden)
- Problemorientierte Programmierung

# Wesentliche Charakteristika

- Programm ist Folge elementarer Anweisungen
- Diese Anweisung haben typischerweise einen Effekt auf die zu verarbeitenden Daten (z.B. werden Daten verändert)
- Programmausführung: Abarbeitung der Anweisungen
- Wichtige Konzepte: Variablen, Verzweigung, Schleifen, Prozeduren
- Vertreter: FORTRAN, ALGOL, COBOL, Pascal, BASIC, C, Modula (und z.T., als deren Weiterentwicklungen: C++ und Java))

# Historische Entwicklungg

- Höhere Abstraktion: weg von der Maschinen-orientierten Problemauffassung hin zur Beschreibungsform der Problemstellung (daher auch applikative, d.h. anwendungsbezogene Programmierung)
- Eigenschaft des Problems, bzw. der Lösung, wird beschrieben (deklarativ), orientiert am Menschen statt an der Rechenanlage

# Wesentliche Charakteristika

- Programm ist Funktion von Eingabe- in Ausgabewerte
- Die Funktion hat keine Effekte auf die zu verarbeitenden Daten (z.B. werden Eingabedaten nicht verändert), sie gibt lediglich die Ausgabe zurück
- Programmausführung: Berechnung der Funktion
- Wichtige Konzepte: Ausdruck, Substitution/Auswertung, $\lambda$-Kalkül
- Vertreter: LISP, SML, Haskell

# Historische Entwicklung

- Bei großen SW-Systemen werden imperative/funktionale Programme schnell unübersichtlich
- Idee der abstrakten Datentypen wird zu benutzerdefinierten Datentypen erweitert
- Noch stärkere Abstraktion durch Datenkapselung

# Grundidee

- Daten sind Objekte, die miteinander interagieren (sich Nachrichten schicken)
- Klassen stellen benutzerdefinierte Datentypen zur Verfügung, d.h. definieren eine Menge von Objekten mit gewissen Eigenschaften (Attribute, die den Zustand der Objekte spezifizieren) und Verhalten (Methoden, die z.B. den Zustand verändern)
- Objekte verschiedener Klassen interagieren durch gegenseitiges Aufrufen von Methoden
- Einbettung von Klassen in eine Hierarchie ermöglicht Vererbung (Wiederverwendung) von Eigenschaften

# Beispiel: Wechselgeld

- In unserem Beispiel Wechselgeld (was eigentlich zu klein für eine klassische oo Programmierung ist) würde man ggfls. einen neuen Datentyp “Wechselgeld” definieren.
- Ein Objekt vom Typ “Automat”, der das Wechselgeld berechnet, könnte einem entsprechenden Wechselgel-Objekt Nachrichten schicken (eine entspr. Methode aufrufen), z.B. “erhöhe um 5 EUR”.
- Voraussetzung dafür: der Datentyp Wechselgeld bietet diese “Funktionalität” (Methode) an (andernfalls könnte er mit dieser Nachricht nichts anfangen).

# Beispiel: Wechselgeld

- D.h. man müsste sich überlegen, welche Nachrichten Objekte vom Typ “Wechselgeld” verarbeiten können sollten, d.h. welche Methoden bereitgestellt werden sollen.
- Neben dem Erhöhen des Betrags, könnte das z.B. die Abfrage der Anzahl der einzelnen Münzen/Scheine sein.
- Ein Automat könnte diese Informationen benötigen, um die entsprechende Anzahl an Münzen und Scheinen auszuzahlen.

# Beispiel: Wechselgeld

- Was ist der Vorteil?
  - Der Datentyp Wechselgeld kann unabhängig vom Wechselgeld-Algorithmus implementiert und verwendet werden.
  - Der Wechselgeld-Algorithmus ist damit weitgehend unabhängig von der Darstellung des Wechselgelds (Details des Datentyps “Wechselgeld”): solange die Schnittstelle klar ist (d.h. informell welche Methoden zur Verfügung stehen, wie diese aufgerufen werden können, und was das Resultat ist), muss der Automat keinerlei Details des Wechselgeld-Objekts kennen (weder wie die Methoden genau funktionieren, noch wie Wechselgeld-Objekte aufgebaut sind).

# Beispiel: Wechselgeld

- Wie gesagt, in diesem Beispiel ist das noch kein großer Ah-Effekt, bei komplexen Software-Systemen mit mehreren Entwicklern ist der Effekt allerdings umso größer.
- Nochmal: Einzelne Aspekte können komplett unabhängig von einander entwickelt und (wieder-)verwendet werden.

# Wesentliche Charakteristika

- Programm ist Menge von Klassen und Objekten
- Programmausführung: Interaktion zwischen Objekten
- Wichtige Konzepte: Klassen und Objekte, Datenkapselung, Vererbung, Polymorphie
- Vertreter: Java, C++, Python

# Grundidee

- Ausgehend von einer Menge von Fakten (Grundtatsachen und Axiome) wird mit Hilfe festgelegter Schlussregeln eine Aussage abgeleitet (oder falsifiziert).
- Programm ist Sammlung von Fakten und Regeln
- Programmausführung: Suche nach Antworten auf Anfragen
- Wichtige Konzepte: logisches Schließen, regelbasiert, prädikativ, constraintbasiert
- Vertreter: PROLOG

# Zusammenfassung

- Imperative Algorithmen:
  - spezifizieren die Abfolge von Aktionen, die typischerweise bestimmte Größen verändern.
  - spiegeln die technischen Möglichkeiten von Rechneranlagen wider, die Schritt für Schritt bestimmte grundlegende Aktionen ausführen können.
- Funktionale Algorithmen:
  - spezifizieren eine auswertbare Darstellung des funktionalen Zusammenhangs zwischen Ein- und Ergebniswerten.
  - spiegeln ein höheres Abstraktionsniveau wider: die eigentliche Spezifikation des Algorithmus als Abbildung ist praktisch losgelöst von den technischen Möglichkeiten der Rechenanlage (Aktionen Schritt für Schritt auszuführen).

# Zusammenfassung

- Objektorientierte Algorithmen:
  - spezifizieren eine höhere Abstraktionsebene zur Darstellung von komplexen Eingabe- und Ergebnisdaten sowie Zwischenzuständen während der Berechnung.
  - Idee der (Daten-)Objekte, die sich gegenseitig Nachrichten schicken
  - verwenden zur eigentlichen Lösung der gegebenen Aufgabe (wie die Nachrichten verarbeitet werden) funktionale und/oder imperative Algorithmen.