Algorithmen und Datenstrukturen

# 2. PROGRAMMIERPARADIGMEN

# Überblick 1/2

- Algorithmenentwurf ist eine kreative Disziplin
- Es gibt keine allgemeingültige Anleitung zum Entwerfen und Analysieren von Algorithmen
- Wir werden uns in dieser Vorlesung mit vielen Beispielen beschäftigen, die als Inspiration und Werkzeug dienen, weitere Algorithmen zu entwerfen
- Einige theoretische Grundlagen sind allerdings notwendig

# Überblick 2/2

In den nächsten drei Kapiteln beschäftigen wir uns näher mit

**Programmierparadigmen**
Was für Möglichkeiten gibt es Algorithmen zu entwickeln und zu implementieren?

**Laufzeitanalysen**
Wie kann man die Laufzeit eines Algorithmus analytisch ableiten und einordnen?

**Entwurfsmuster**
Was sind generelle Prinzipien für das Design eines Algorithmus?

# Programmierparadigmen

„Unter einem Paradigma versteht man unter anderem in der Wissenschaftstheorie ein `**Denkmuster, welches das wissenschaftliche Weltbild eine Zeit prägt**´ - ein Algorithmenparadigma sollte daher ein Denkmuster darstellen, das die Formulierung und den Entwurf von Algorithmen und damit letztendlich von Programmiersprachen grundlegend prägt.“

Oder etwas kürzer:

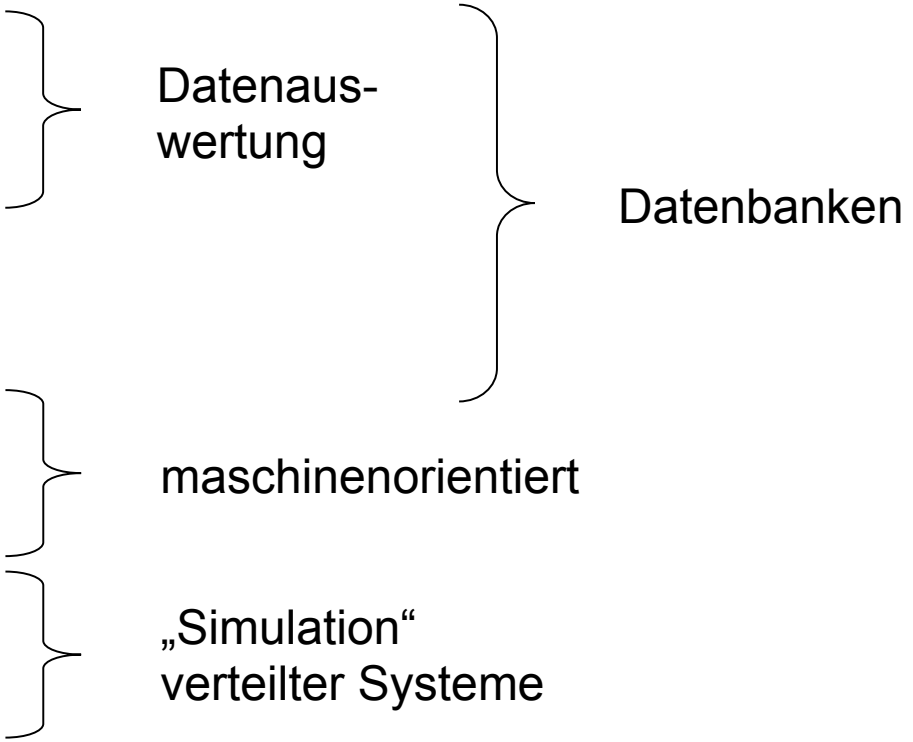
- Ein Muster für den Entwurf und die Formulierung von Algorithmen

# Paradigmen zur Algorithmenkonstruktion

- **Funktional**: Verallgemeinerung der Funktionsauswertung. Rekursion spielt eine wesentliche Rolle.
  - f(x) := 2 g(x) + h(x)
  - h(x) := 1 + h(x-1)
- **Logisch**: basierend auf logischen Aussagen und Schlußfolgerungen
  - „wenn a verwandt mit b und b verwandt mit c, dann ist a verwandt mit c“
- **Imperativ**: basierend auf einem einfachen Maschinenmodell mit gespeicherten und änderbaren Werten. Primär werden Schleifen und Alternativen als Kontrollbausteine eingesetzt.
  - „erst: erhöhe a, dann multipliziere b mit c, dann subtrahiere a mit c, ….“
- **Objektorientiert**: basierend auf Nachrichtenaustausch zwischen Objekten und Vererbung von Klassen
- Bsp. Java: objektorientiert, imperativ, Elemente von funktional

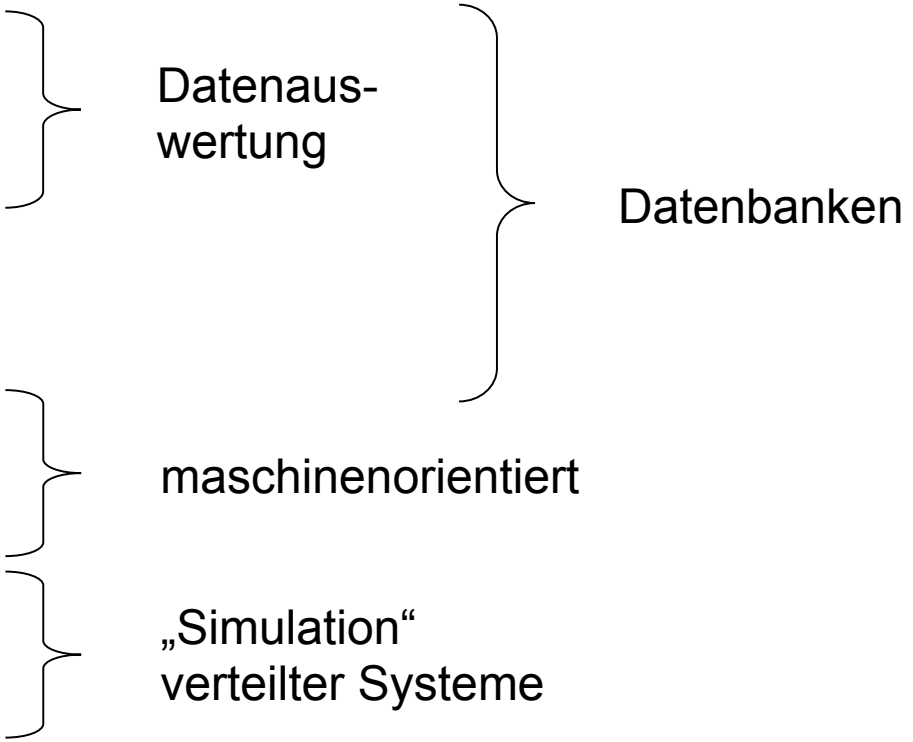
# Paradigmen und Programmiersprachen

- Funktional
  - Haskell, ML, Lisp
- Logisch
  - Prolog
- Imperativ
  - C, Pascal
- Objektorientiert
  - Smalltalk, Eiffel
- Mischungen:
  - C++, C#, Java



# Paradigmen und Programmiersprachen

- **Funktional**
  - Haskell, ML, Lisp
- **Logisch**
  - Prolog
- **Imperativ**
  - C, Pascal
- Objektorientiert
  - Smalltalk, Eiffel
- Mischungen:
  - C++, C#, Java



Algorithmen und Datenstrukturen

# 2.1 FUNKTIONALE ALGORITHMEN

# Funktionale Algorithmen

Grundidee:
Algorithmus = Funktion (im mathematischen Sinn)

**Definition** (Funktion)

Eine Funktion f ist eine Relation zwischen einer Eingabemenge X und einer Ausgabemenge Y ($f \subseteq X \times Y$) mit der Eigenschaft:

Für alle $x \in X, y, y' \in Y$ mit $(x,y),(x,y') \in f$ gilt $y=y'$

Wir schreiben dann üblicherweise $f(x)=y$ anstatt $(x,y) \in f$ und deklarieren eine Funktion durch $f: X \rightarrow Y$

# Funktionen und Terme

- Ist f: X→Y eine Funktion so heißt X Eingabemenge und Y Ausgabemenge
- In der funktionalen Programmierung sind Ein- und Ausgabemengen üblicherweise **Terme** eines bestimmten Typs

# Funktionen und Terme

**Definition** (Term)

Sei T ein Typ, $V_T$ eine Menge von Variablen vom Typ T und $C_T$ eine Menge von Konstanten vom Typ T. Dann

- Jedes $X \in V_T$ ist ein Term vom Typ T
- Jedes $a \in C_T$ ist ein Term vom Typ T
- Ist $f:T^k \rightarrow T$ eine Funktion und $t_1,...,t_k$ sind Terme vom Typ T, so ist $f(t_1,...,t_k)$ ein Term vom Typ T

# Funktionen und Terme

**Beispiel** (Terme natürlicher Zahlen)

Sei *int* der Typ der natürlichen Zahlen, $V_{int}$ eine Menge von Variablen vom Typ $T_{int}$ und $C_{int} = \mathbb{N} = \{1,2,3,...\}$

Mögliche Funktionen auf natürlichen Zahlen sind

- $+: int \times int \rightarrow int$
- $*: int \times int \rightarrow int$

3+4, (8+9)*10, X*4+1 sind dann Terme natürlicher Zahlen

# Funktionen und Terme

**Beispiel** (Bool'sche Terme)

Sei *bool* der Typ der Bool'schen Terme, $V_{bool}$ eine Menge von Variablen vom Typ $T_{bool}$ und $C_{bool}$ = {true,false}

Mögliche Funktionen auf Bool'schen Termen sind

- $\wedge : bool \times bool \rightarrow bool$
- $\neg : bool \rightarrow bool$

true $\wedge$ false und $\neg Y \wedge X$ sind dann Bool'sche Terme

# Funktionen und Terme

Sind $v_1, \ldots, v_n$ Variablen vom Typ $T_1, \ldots, T_n$ und ist $t(v_1, \ldots, v_n)$ ein Term, so heißt

$$f(v_1, \ldots, v_n) := t(v_1, \ldots, v_n)$$

eine Funktionsdefinition vom Typ T.

T ist dabei der Typ des Terms $t(v_1, \ldots, v_n)$.

mit

- f: Funktionsname
- $v_1, \ldots, v_n$ formale Parameter
- $t(v_1, \ldots, v_n)$: Funktionsausdruck

# Signatur einer Funktion

- Funktion *f* hat folgende Funktionsdefinition:

$$f(v_1, \ldots v_n) \;:=\; t(v_1, \ldots, v_n)$$

mit

$$v_1, \ldots, v_n \text{ sind vom Typ } T_1, \ldots, T_n$$

$$t(v_1, \ldots, v_n) \text{ ist vom Typ } T$$

- Die Signatur von *f* ist:

$$f : T_1 \times \ldots \times T_n \rightarrow T$$

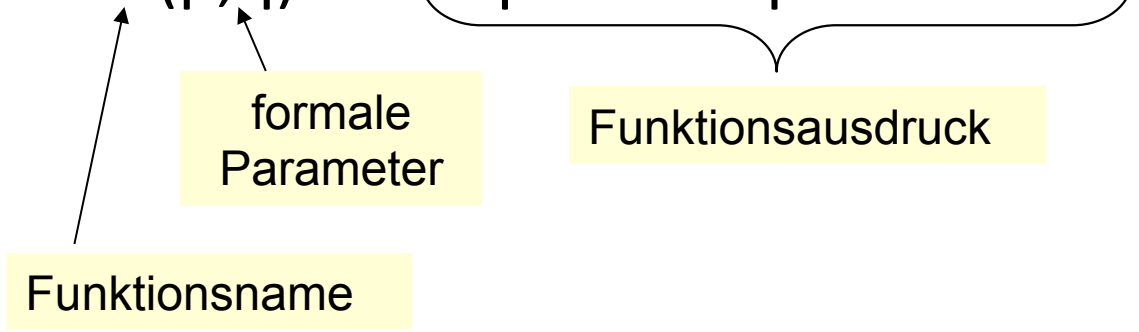
Name mit Stelligkeit

Parameter mit Typen

Typ des Rückgabewerts

# Beispiele für Funktionsdefinitionen

- f(p,q,x,y) := if (p ∨ q) then 2x + 1 else 3y -1
- g(x) := if even(x) then x / 2 else 3x -1
- h(p,q) := if p then q else false



# Auswertung von Funktionen

- Definierte Funktionen können mit konkreten Werten ***aufgerufen*** werden.

Sind nun $a_1, \ldots, a_n$ konkrete Werte vom Typ $T_1, \ldots, T_n$, so ersetzt man in $f(a_1, \ldots, a_n)$ jedes Vorkommen der Unbestimmten $v_i$ mit $a_i$ (i = 1, … , n). Somit kann der entstehende Term **ausgewertet** werden.

- Dabei heißen die konkreten Werte $a_1, \ldots, a_n$ aktuelle Parameter.
- Ausdruck $f(a_1, \ldots, a_n)$ heißt Funktionsaufruf

# Beispiel für Auswertungen

- f(p,q,x,y) := if (p ∨ q) then 2x + 1 else 3y -1

  Signatur: *f: bool x bool x int x int → int*

  Aufruf: *f(true,true,3,4)* wird zu 7 ausgewertet

- g(x) := if even(x) then x / 2 else 3x -1

  Signatur: g: *int → int*

  Aufruf: *g(2)* wird zu 1, *g(9)* wird zu 26 ausgewertet

- h(p,q) := if p then q else false

  Signatur: *h: bool x bool → bool*

  Aufruf: *h(false,false)* wird ausgewertet zu *false*

Erkennen Sie welche logische Funktion h beschreibt?

# Schachtelung von Funktionen

- Aufrufe definierter Funktionen können in Termen vorhanden sein
- Beispiel:
  - f(x,y) := if g(x,y) then h(x+y) else h(x-y)
  - g(x,y) := (x==y) $\vee$ odd(y)
  - h(x) := j(x+1) $\cdot$ j(x-1)
  - j(x) := 2x - 3
- Eine Funktionsdefinition f heißt **rekursiv**, wenn direkt oder indirekt (über andere Funktionen) ein Funktionsaufruf f(...) in ihrer Definition auftritt.

# Beispiel für Auswertung

f(1,2) ↦ if g(1,2) then h(1+2) else h(1-2)

↦ if 1 == 2 ∨ odd(2) then h(1+2) else h(1-2)

↦ if 1 == 2 ∨ false then h(1+2) else h(1-2)

↦ if false ∨ false then h(1+2) else h(1-2)

↦ if false then h(1+2) else h(1-2)

↦ h(1-2)

↦ h(-1)

↦ j(-1+1) · j(-1-1)

↦ j(0) · j(-1-1)

↦ j(0) · j(-2)

↦ (2 · 0 -3) · j(-2)

↦ (-3) · (-7)

↦ 21

# Rekursive Funktionsdefinition

$$
\begin{aligned}
f(x,y) \quad := \quad & if\ x = 0\ then\ y\ else( \\
& if\ x > 0\ then\ f(x-1,y)+1 \\
& else \quad -f(-x,-y)\ )
\end{aligned}
$$

Auswertungen:

$$f(0,y) \mapsto y \text{ für alle } y$$

$$f(1,y) \mapsto f(0,y)+1 \mapsto y+1$$

$$f(2,y) \mapsto f(1,y)+1 \mapsto (y+1)+1 \mapsto y+2$$

...

$$f(n,y) \mapsto y+n \text{ für alle } n \in int, n > 0$$

$$f(-1,y) \mapsto -f(1,-y) \mapsto -(-y+1) \mapsto y-1$$

...

$$f(x,y) \mapsto x+y \text{ für alle } x,y \in int$$

# Zusammenfassung: Funktionale Algorithmen

Ein funktionaler Algorithmus ist eine Menge von Funktionsdefinitionen $f_1$ bis $f_m$ mit:

$$\begin{aligned} f_1(v_{1,1},\ldots,v_{1,n_1}) &:= t_1(v_{1,1},\ldots,v_{1,n_1}), \\ &\vdots \\ f_m(v_{m,1},\ldots,v_{m,n_m}) &:= t_m(v_{m,1},\ldots,v_{m,n_m}). \end{aligned}$$

- Die erste Funktion $f_1$ wird wie beschrieben ausgewertet und ist die Bedeutung (=Semantik) des Algorithmus
- $f_1$ ist die Zustands-bestimmende Eingabe aus der die Werte der Ausgabe abgelesen werden
- Funktionale Algorithmen sind die Grundlage einer Reihe von universellen Programmiersprachen, z.B. APL und Lisp. Diese Programmiersprachen werden als ***funktionale Programmiersprachen*** bezeichnet.

# Definiertheit

- Auf welchen Eingaben ist der Algorithmus definiert?

$$f(x) := if\ x == 0\ then\ 0\ else\ f(x-1)$$

Auswertungen

$$f(0) \mapsto 0$$

$$f(1) \mapsto f(0) \mapsto 0$$

$$f(2) \mapsto f(1) \mapsto f(0) \mapsto 0$$

$$f(x) \mapsto 0\ \forall\ x \in int,\ x > 0$$

$$f(-1) \mapsto f(-2) \mapsto \ldots$$

Auswertung terminiert nicht!

Also gilt:

$$f(x) := \begin{cases} 0 & \text{falls } x \geq 0 \\ \bot & \text{sonst} \end{cases}$$

# Weiteres Beispiel: Fakultät

$$x! \;:=\; x \cdot (x-1) \cdot (x-2) \ldots 2 \cdot 1 \text{ für } x > 0$$

- Bekannte Definition 0! := 1, x! := x (x-1)!
- Problem: negative Eingabewerte
- 1. Lösung
  - fac(x) := if (x==0) then 1 else x · fac(x-1)
- Bedeutung:

$$fac(x) \;:=\; \begin{cases} x! & \text{falls } x \geq 0 \\ \bot & \text{sonst} \end{cases}$$

# Beispiel Fakultät - Version 2

$$x! \;:=\; x \cdot (x-1) \cdot (x-2) \ldots 2 \cdot 1 \text{ für } x > 0$$

- Bekannte Definition 0! := 1, x! := x (x-1)!
- Problem: negative Eingabewerte
- 2. Lösung
  - fac(x) := if (x ≤ 0) then 1 else x · fac(x-1)
- Bedeutung:

$$fac(x) \;:=\; \begin{cases} x! & \text{falls } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

Vergleich: Version 1

$$fac(x) \;:=\; \begin{cases} x! & \text{falls } x \geq 0 \\ \bot & \text{sonst} \end{cases}$$

# Algorithmus-Entwicklung: ggT funktional

- Berechnung des ggT als funktionaler Algorithmus
- Hintergrundwissen
  - Für x, y > 0 gilt:

$$\begin{aligned} ggT(x,x) &:= x \\ ggT(x,y) &:= ggT(y,x) \\ ggT(x,y) &:= ggT(x,y-x) \text{ für } x < y \end{aligned}$$

- Funktionale Spezifikation

$$ggT(x,y) := \begin{cases} \text{if } (x \leq 0) \vee (y \leq 0) & \text{then} & ggT(x,y) & \text{else} \\ \text{if } x == y & \text{then} & x & \text{else} \\ \text{if } x > y & \text{then} & ggT(y,x) & \text{else} \\ & & ggT(x,y-x) & \end{cases}$$

# Größter gemeinsamer Teiler

$$ggT(x,y) := \begin{cases} \text{if } (x \leq 0) \vee (y \leq 0) & \text{then} & ggT(x,y) & \text{else} \\ \text{if } x == y & \text{then} & x & \text{else} \\ \text{if } x > y & \text{then} & ggT(y,x) & \text{else} \\ & & ggT(x, y-x) & \end{cases}$$

- Beispiel Auswertung:

$$\begin{aligned} ggT(39,15) &\mapsto ggT(15,39) \mapsto ggT(15,24) \\ &\mapsto ggT(15,9) \mapsto ggT(9,15) \\ &\mapsto ggT(9,6) \mapsto ggT(6,9) \\ &\mapsto ggT(6,3) \mapsto ggT(3,3) \\ &\mapsto 3 \end{aligned}$$

- ggT korrekt für positive Eingaben, bei negativen Eingaben undefiniert (d.h. terminiert nicht)

# Algorithmuskonstruktion: ggT

- Definiere Zielfunktion
  - ggT

- Liste Bedingungen für Abbruch
  - $x \leq 0$
  - $y \leq 0$
  - $x == y$

  gebe Evaluierung oder Ausnahmen in den Fällen des Abbruchs an

- Liste Bedingungen für rekursive Verwendung der Funktion, „einfachste“ Rekursion zuerst
  1. $x,y > 0,\ x \geq y$
  2. $x,y > 0,\ x < y$

  gebe Evaluierungen in den rekursiven Fällen an

# Von der Spezifikation zum Programm

$$ggT(x,y) := \begin{cases} \text{if } (x \leq 0) \vee (y \leq 0) & \text{then} & ggT(x,y) & \text{else} \\ \text{if } x == y & \text{then} & x & \text{else} \\ \text{if } x > y & \text{then} & ggT(y,x) & \text{else} \\ & & ggT(x, y-x) & \end{cases}$$

**Programm:**

```
public static int ggT(int x, int y){
     if  ((x <= 0) || (y <= 0))
         throw new ArithmeticError("negative Daten bei ggt"));
     else   if  (x==y)  then return x;
                else
                          if x > y  then return ggT(y,x);
                          else return ggT(x,y-x);
}
```

# Beispiel: Fibonacci-Zahlen

- Hintergrundwissen
  - Zahlenreihe
  - Progression bei Vermehrung von Kaninchen

Intuition:

1. Am Anfang gibt es ein Kaninchenpaar
2. Jedes Paar wird im zweiten Monat zeugungsfähig
3. Danach zeugt es jeden Monat ein weiteres Paar
4. Kein Kaninchen stirbt

# Lösung: Fibonacci-Zahlen

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_2 &= 1 = f_0 + f_1 \\
f_3 &= 2 = f_1 + f_2 \\
f_4 &= 3 = f_2 + f_3 \\
&\ldots
\end{aligned}
$$

- Allgemein:

```
fib(x)  :=  if (x==0) then 0
                else if (x==1)  then  1
                    else  fib(x-1) + fib(x-2)
```

Bedeutung:

$$
fib(x) = \begin{cases} \text{x-te Fibonacci-Zahl} & \text{falls } x \geq 0 \\ \perp & \text{sonst} \end{cases}
$$

# Weiteres Beispiel

- Funktionaler Algorithmus mit mehreren Funktionen
- Algorithmus: Testet ob eine Zahl gerade ist (even(x)).
  - Mathematische Regeln:

even(0) = true

odd(0) = false

even(x+1) = odd(x)

odd(x+1) = even(x)

… zum Algorithmus

# … Algorithmus

```
even (x)   =   if  x = 0  then
                        true
                 else
                        if x > 0 then
                              odd(x-1)
                        else
                              odd(x+1)

odd(x)   =   if x = 0 then
                        false
                 else
                        if x > 0 then
                              even(x-1)
                        else
                              even(x+1)
```

Vorgehen:
Mathematische Regeln direkt in Algorithmus umsetzen

Algorithmen und Datenstrukturen

# 2.2 LOGISCHE ALGORITHMEN

# Logische Programmierung

- Deklaratives Programmierparadigma
- Ähnlich wie funktionale Programmierung
  - Keine schrittweise (=imperative) Abhandlung von Schritten
- Logische Zusammenhänge werden in Form von Klauseln aufgestellt
- Basiert auf Prädikatenlogik erster Stufe, im speziellen auf Hornlogik
- Konkrete Programmiersprachen: Prolog, Answer Set Programming
- Hier: Prolog

# Grundlagen zur Prädikatenlogik/Hornlogik 1/2

Sei U eine Menge von Konstanten, V eine Menge von Variablen, und P eine Menge von Prädikatsymbolen.

- Ein *Term* ist entweder eine Konstante oder eine Variable (prinzipiell sind auch Funktionsterme möglich, werden hier aber ignoriert)
  - X,Y,... sind Variablen (und Terme)
  - anna, bob, dave,... sind Konstanten (und Terme)
- Ein *Atom* ist ein n-stelliges Prädikat, gefolgt von n Termen
  - parent(bob,anna) ist ein Atom
  - sibling(anna, X) ist ein Atom

# Grundlagen zur Prädikatenlogik/Hornlogik 2/2

- Eine *atomare Konjunktion* ist eine Menge von Atomen
  - parent(X,anna) $\wedge$ sibling(anna,Y) $\wedge$ parent (anna,tina)
  - Bei logischer Programmierung wird oft das Komma für die Konjunktion verwendet:
    parent (X,anna), sibling(anna,Y), parent(anna,tina)
- Eine *Hornklausel* ist eine Implikation einer atomaren Konjunktion zu einem Atom
  - grandparent(X,Y) $\Leftarrow$ parent(X,Z) $\wedge$ parent(Z,Y)
  - In Prolog: grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
- Anmerkung: Ein Hornklausel ist eigentlich definiert als eine Disjunktion mit maximal einem positiven Atom

# Prolog-Programm

- Ein Prolog-Programm ist eine Menge von Hornklauseln und Fakten (=Atome ohne Variablen)
- Beispiel:
  grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
  brother(X,Y) :- X!=Y, male(Y), parent(Z,X), parent(Z,Y).
  hasUncle(X) :- parent(Y,X), brother(Y,_).
  parent(bob, anna).
  parent(carl, bob).
  male(bob).
- Anmerkung: „_“ ist eine beliebige (unbenannte) Variable

# Logische vs. Funktionale Programmierung

- Hornklauseln ≈ Funktionen (im Sinne von atomaren Operationen)
- Gemeinsamkeiten
  - Rekursion als zentrales Paradigma
  - Mathematische Basis
- Unterschiede
  - Atome sind entweder wahr oder falsch
  - Funktionwerte können beliebigen Typ haben

# Anfragen an Prolog-Programme 1/2

- Prolog ist eine Anfrage-basierte Programmiersprache
- Jede Ausführung eines Prolog-Programms muss mit einer Anfrage parametrisiert werden
- Beispiel:
  grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
  brother(X,Y) :- X!=Y, male(Y), parent(Z,X), parent(Z,Y).
  hasUncle(X) :- parent(Y,X), brother(Y,_).
  parent(bob, anna).
  parent(carl, bob).
  male(bob).
- Anfragen:
  - ?grandparent(carl,anna) → Antwort *YES*
  - ?male(anna) → Antwort *NO* (Closed World Assumption)

# Anfragen an Prolog-Programme 2/2

- Anfragen können auch Variablen enthalten:
  - s(X,Y) :- r(X,Y), t(Y).
    r(a,b). r(a,e). r(c,d).
    t(b). t(d).
  - Anfragen:
    - ?s(a,X) → Antwort X=b
    - ?r(a,X) → Antwort X=b, X=e
- Die Semantik logischer Programme leitet sich direkt von der klassisch logischen Semantik der Prädikatenlogik ab (siehe Logik-Vorlesung). Techniken:
  - Grundierung des Programms (ersetze Variablen durch alle Kombinationen von Konstanten) und aussagenlogische Verarbeitung
  - Unifikation des Anfrageterms und Backtracking

# Beispiel

Problem: Wegfindung in gerichteten Graphen

- Gegeben ein Graph mit Knoten $a_1,...,a_n$
- Gibt es einen Weg zwischen Knoten $a_i$ und $a_j$ (für beliebige i,j)?

Lösung als Prolog-Programm:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge (X,Z), path(Z,Y).
edge(a1,a2). edge(a2,a3). edge(a2,a4). edge(a5,a1).
```

Anfragen:

- ?path(a1,a3) → Antwort YES
- ?path(a5,X) → Antwort X=a1, X=a2, X=a3, X=a4 (alle von a5 erreichbare Knoten)

# Erweiterung: Listen

- Eine *Liste* ist
  - entweder die leere Liste oder
  - ein Term gefolgt von einer Liste.
- Definition in Prolog:
  list([]).
  list([X|Y]) :- list(Y).
- Der |-Operator trennt den Kopf (Head=erstes Element) einer Liste vom Rumpf (Tail=Restliste) ab
- Beispiele:
  - Liste von Zahlen: [1|[2|[3]]] = [1,2,3]
  - Liste von beliebigen Termen:
    [male(bob), female(anna),male(carl)]

# Listenmanipulation

- Aneinanderreihung:
  append(X,Y,Z): X ist die Liste, die entsteht, wenn Z an Y angehängt wird
  - append(X,[],X).
    append([Y|X], [Y|Z],L) :- append(X,Z,L).
- Invertierung:
  invert(X,Y): X ist die Invertierung von Y
  - invert([],[]).
    invert([X|Y],L) :- invert(Y,Z), append(L,Z,[X]).

Algorithmen und Datenstrukturen

# 2.3 IMPERATIVE ALGORITHMEN

# Imperative Algorithmen

- verbreitete Art um Algorithmen für Computer zu formulieren
- Basiert auf den Konzepten ***Anweisung*** und ***Variablen***
- Wird durch Programmiersprachen Java, C, PASCAL, FORTRAN, COBOL, Maschinencode, … realisiert
- Prinzip: Abstraktes Rechnermodell
  - ➔ Werte speichern und
  - ➔ Schrittweise bearbeiten
- Nicht so elegant, verständlich und wartbar wie Funktional, Objektorientiert oder Logisch

# Variablen

- Eine **Variable** besteht aus einem Namen (z.B. X), einem veränderlichen Wert und einem Typ
- Variablen sind Speicherplätze für Werte
- Ist t ein Term ohne Variablen und w(t) sein Wert, dann heißt das Paar X:=t eine **Wertzuweisung**. Ihre Bedeutung ist festgelegt durch
  - Nach Ausführung von X:=t gilt X=w(t)
  - Vor Ausführung der ersten Wertzuweisung gilt X = ? (undefiniert)
- Beispiele:
  - X := 7
  - X := (3-7) · 9
  - F := true
  - Q := ¬ (true v false) v ¬¬ true

# Zustände

- Ist $\underline{X} = \{X_1, X_2, \ldots\}$ eine Menge von Variablen(-namen) von denen alle nur Werte aus der Wertemenge W haben können (alle Variablen vom gleichen Typ), dann ist der Zustand Z eine partielle Abbildung
  - $Z: \underline{X} \rightarrow W$ (Zuordnung des momentanen Wertes)

- Zum Beispiel in einem gewissen Zustand:
  - $Z(X_1) = 42$
  - $Z(X_2) = 17$
  - $Z(X_3) = 23$
- Nach $X_1 := 29$ folgt:
  - $Z(X_1) = 29$
  - $Z(X_2) = 17$
  - $Z(X_3) = 23$

# Zustände

- Ist $\underline{X} = \{X_1, X_2, \ldots\}$ eine Menge von Variablen(-namen) von denen alle nur Werte aus der Wertemenge W haben können (alle Variablen vom gleichen Typ), dann ist der Zustand Z eine partielle Abbildung
  - $Z: \underline{X} \rightarrow W$ (Zuordnung des momentanen Wertes)

- Ist $Z: \underline{X} \rightarrow W$ ein Zustand und wählt man eine Variable X und einen Wert w aus dem Wertebereich W, so ist der ***transformierte*** Zustand wie folgt definiert:

$$Z\langle X \leftarrow w\rangle : \underline{X} \rightarrow W \text{ mit}$$

Neuer Zustand

$$Z_{\langle X \leftarrow w\rangle}(Y) \mapsto \begin{cases} w, & \text{falls } X = Y \\ Z(Y), & \text{sonst} \end{cases}$$

Alter Zustand

# Ausdrücke

- Ausdrücke entsprechen im Wesentlichen den Termen einer funktionalen/logischen Sprache, jedoch stehen an der Stelle von Unbestimmten nun Variablen
- Die Auswertung von Termen ist zustandsabhängig:
  - Gegebener Term: $2 \cdot X + 1$
  - Wert im Zustand Z ist durch $2 \cdot Z(X) + 1$ bestimmt

# Wert eines Ausdrucks

- Wert eines Ausdrucks $t(X_1, \ldots, X_n)$ wird mit $Z(t(X_1, \ldots, X_n))$ bezeichnet:
  - $Z(2 \cdot X + 1) = 2 \cdot Z(X) + 1$

- Damit auch möglich:
  - $X := t(X_1, \ldots, X_n)$ (Wertzuweisung mit Variablen)

- Transformierter Zustand ist definiert:

Lese: „X wird der Wert des Ausdrucks $t(X_1, \ldots, X_n)$ zugeordnet“

$$[\![X := t(X_1, \ldots, X_n)]\!](Z) = Z\langle X \leftarrow Z(t(X_1, \ldots, X_n))\rangle$$

- „Semantikklammern“: Bedeutung einer Anweisung hier definiert als eine Funktion auf Zuständen

# Anweisungen

- Arten von Anweisungen
  - Elementare Anweisungen

    (→ Wertzuweisungen)
  - Komplexe Anweisungen
- Semantik einer Anweisung:

Funktion, die einen Zustand in einen neuen Zustand überführt. $[\![\alpha]\!](Z)$

- Allgemein:

  Wirkungsweise von α auf Zustand Z

# Zuweisungen als Anweisungen

- Beispiel: Wertzuweisung

  $\alpha_1$ = (X := 2 · Y + 1)

$\alpha_1$ ist eine **elementare Anweisung**

- Transformation (Funktion auf Zustände):

$$[\![\alpha_1]\!](Z) \;=\; Z\langle X \leftarrow 2 \cdot Z(Y) + 1\rangle$$

- Zuweisung berechnet neuen Zustand
  - Alter Zustand: $Z$
  - Neuer Zustand: $[\![\alpha_1]\!](Z)$

# Zuweisungen als Anweisungen (2)

- Beispielzuweisung mit gleicher Variable auf beiden Seiten:

$$\alpha_1 \;=\; \langle X \;:=\; 2 \cdot X + 1 \rangle$$

- Transformation in

$$[\![\alpha_1]\!](Z) \;=\; Z\langle X \;\leftarrow\; 2 \cdot Z(X) + 1 \rangle$$

- Bei der letzten Anweisung handelt es sich nicht um eine rekursive Gleichung!

- Hinweis: Wertzuweisungen sind die einzigen elementaren Anweisungen

# Komplexe Anweisungen

- Bisher: elementare Anweisungen (Wertzuweisungen) können als Funktionen auf Zustände verstanden werden.

- Komplexe Anweisungen: Nehmen Konstrukte/Bausteine von imperativen Algorithmen:

  1. Sequenz
  2. Auswahl/Selektion
  3. Iteration

Bausteine

- Semantik wird wiederum durch Konstruktion von Funktionen definiert
- Hinweis: Iteration ist das Gegenstück zu rekursiven Funktionsaufrufen bei funktionalen Algorithmen

# Sequenz

- Sequenz (Folge): Sind $\alpha_1$ und $\alpha_2$ Anweisungen, so ist

  $\alpha_1$ ; $\alpha_2$

  auch eine Anweisung

- Zustandstransformation:

beschreibt Semantik der Sequenz!

$$[\![\alpha_1 \ ; \ \alpha_2]\!](Z) \ = \ [\![\alpha_2]\!]([\![\alpha_1]\!](Z))$$

- Semantik: Schachteln der Funktionsaufrufe, Hintereinander-Ausführung der beiden Funktionen

# Selektion

- Auswahl (Selektion): Sind $\alpha_1$ und $\alpha_2$ Anweisungen und *B* ein boolscher Ausdruck, so ist auch

  *if B then* $\alpha_1$ *else* $\alpha_2$

  eine Anweisung.
- Zustandstransformation:

$$[\![if\ B\ then\ \alpha_1\ else\ \alpha_2]\!](Z) = \begin{cases} [\![\alpha_1]\!](Z), & \text{falls } Z(B) = true \\ [\![\alpha_2]\!](Z), & \text{falls } Z(B) = false \end{cases}$$

- Voraussetzung: Z(B) ist definiert, sonst Bedeutung der Auswahlanweisung undefiniert

# Iteration

- Wiederholung (Iteration, oder auch Schleife): Ist α eine Anweisung und B ein boolscher Ausdruck, so ist

  *while B do α*

  auch eine Anweisung.

- Zustandstransformation:

$$[\![while\ B\ do\ \alpha]\!](Z) = \begin{cases} Z, & \text{falls } Z(B) = false \\ [\![while\ B\ do\ \alpha]\!]([\![\alpha]\!](Z)), & \text{sonst} \end{cases}$$

- Ist Z(B) undefiniert, so ist die Bedeutung dieser Anweisung ebenfalls undefiniert.

# Umsetzung in Programmiersprachen

- In realen imperativen Programmiersprachen gibt es fast immer diese Anweisungen
- *Imperative Algorithmen* sind die Grundbausteine imperativer Programmiersprachen
- **while**-Schleifen sind rekursiv definiert, ihre rekursive Auswertung braucht nicht zu terminieren
- Bereits Programmiersprachen mit diesen Sprachelementen sind *universell*.
- Wir beschränken uns zunächst auf die Datentypen **bool** und **int**.

➔ Wir können nun die *Syntax imperativer Algorithmen* festlegen

# Syntax imperativer Algorithmen

<Programmname>:

**var** X,Y, …: **int**; P,Q,…: **bool**; (Variablen-Deklaration)

**input** $X_1, \ldots, X_n$; (Eingabe-Variablen)

α (Anweisungen)

**output** $Y_1, \ldots, Y_m$ (Ausgabe-Variablen)

---

… nun zur Semantik:

- Festlegung der formalen Bedeutung etwas komplexer als bei funktionalen Algorithmen

- gleiches Ziel: **Funktion** zur Semantikfestlegung

# Semantik imperativer Algorithmen

- Die Bedeutung (Semantik) eines imperativen Algorithmus ist eine ***partielle Funktion***:

$$[\![PROG]\!] : \quad W_1 \times \ldots \times W_n \mapsto V_1 \times \ldots V_{\boldsymbol{m}}$$

$$[\![PROG]\!](w_1, \ldots, w_n) = (Z(Y_1), \ldots, Z(Y_m))$$

$$\text{wobei } Z = [\![\alpha]\!](Z_0),$$

$$Z_0(X_i) = w_i, \; i = 1, \ldots, n$$

$$\text{und } Z_0(Y) = \perp, \text{für Variablen } Y \neq X_i \; (i = 1, \ldots, n)$$

Es gilt:

| | |
|---|---|
| PROG | Programmname |
| $W_1, \ldots, W_n$ | Wertebereiche der Typen von $X_1, \ldots, X_n$ |
| $V_1, \ldots, V_m$ | Wertebereiche der Typen von $Y_1, \ldots, Y_m$ |

# Semantik imperativer Algorithmen (2)

- Dies bedeutet: Der Algorithmus definiert eine Transformation auf den gesamten initialen Zustand (geg. durch die Eingabe). Die Bedeutung gibt die Werte der Ausgabevariablen an.

$$
\begin{aligned}
[\![PROG]\!](w_1, \ldots, w_n) &= (Z(Y_1), \ldots, Z(Y_m)) \\
\text{wobei } Z &= [\![\alpha]\!](Z_0), \\
Z_0(X_i) &= w_i,\ i = 1, \ldots, n \\
\text{und } Z_0(Y) &= \bot, \text{für Variablen } Y \neq X_i\ (i = 1, \ldots, n)
\end{aligned}
$$

Hinweis: Funktion Z ist nicht definiert, falls die Auswertung von α nicht terminiert.

# Charakterisierung

- Die Algorithmenausführung imperativer Algorithmen besteht aus einer Folge von Basisschritten, genauer Wertzuweisungen.
- Diese Folge wir mittels Selektion und Iteration basierend auf booleschen Tests über dem Zustand konstruiert. Jeder Basisschritt definiert eine Transformation des Zustands.
- Die Semantik des Algorithmus ist durch die Kombination all dieser Zustandstransformationen zu einer Gesamttransformation festgelegt.

# Beispiele für imperative Algorithmen

- Fakultätsfunktion: 0! = 1, x! = x · (x-1)!  für x>0

```
FAC   var X,Y: int;
      input X;
      Y := 1;
      while X > 1 do   Y := Y · X;  X := X -1
      output Y
```

α

Es ist:

$$[\![FAC]\!](x) = \begin{cases} x! & \text{für } x \geq 0 \\ 1 & \text{sonst} \end{cases}$$

- Falls die Bedingung der **while**-Schleife X ≠ 0 lautet, dann ist:

$$[\![FAC]\!](x) = \begin{cases} x! & \text{für } x \geq 0 \\ \bot & \text{sonst} \end{cases}$$

# Beispiel für die Semantik imperativer Algorithmen

- Gesucht ist das Ergebnis des Aufrufs FAC(3)
  - Abkürzung *while β* für die Zeile:

    while X > 1 do Y := Y · X; X := X -1
  - Signatur der Semantikfunktion:

$$[\![FAC]\!] : int \rightarrow int$$

  - Funktion ist durch Lesen von Y im Endzustand Z definiert

$$[\![FAC]\!](w) = Z(Y)$$

  - Endzustand Z ist definiert durch folgende Gleichung:

$$Z = [\![\alpha]\!](Z_0)$$

# Beispiel für die Semantik imperativer Algorithmen

- Gesucht ist das Ergebnis des Aufrufs FAC(3)
  - Funktion ist durch Lesen von Y im Endzustand Z definiert

$$[\![FAC]\!](w) = Z(Y)$$

  - Endzustand Z ist definiert durch folgende Gleichung:

$$Z = [\![\alpha]\!](Z_0)$$

  - Wobei α die Folge aller Anweisungen des Algorithmus ist.

- Initialer Zustand $Z_0$ ist definiert als $Z_0 = (X=w, Y=\perp)$
- Zustände abkürzend ohne Variablennamen: $Z_0 = (w, \perp)$

# Beispiel für Semantik imperativer Algorithmen

$$
\begin{aligned}
Z &= [\![\alpha]\!](Z_0) \\
&= [\![\alpha]\!](3, \bot) \\
&= [\![Y := 1;\ while\ \beta]\!](3, \bot) \\
&= [\![while\ \beta]\!]([\![Y := 1]\!](3, \bot)) \\
&= [\![while\ \beta]\!](3, \bot)\langle Y \leftarrow 1\rangle \\
&= [\![while\ \beta]\!](3, 1) \\
&= \begin{cases} Z, & \text{falls } Z(B) = false \\ [\![while B\ do\ \alpha']\!]([\![\alpha']\!](Z)), & \text{sonst} \end{cases} \\
&= \begin{cases} (3,1), & \text{falls } Z(X > 1) = (3 > 1) = false \\ [\![while\ \beta]\!]([\![Y := Y \cdot X;\ X := X - 1]\!](Z)), & \text{sonst} \end{cases} \\
&= [\![while\ \beta]\!]([\![Y := Y \cdot X; X := X - 1]\!](3, 1)) \\
&= [\![while\ \beta]\!]([\![X := X - 1]\!]([\![Y := Y \cdot X]\!](3, 1))) \\
&= [\![while\ \beta]\!]([\![X := X - 1]\!](3, 3)) \\
&= [\![while\ \beta]\!](2, 3)
\end{aligned}
$$

# Beispiel für Semantik imperativer Algorithmen (2)

$$= \begin{cases} (2,3), \text{ falls } Z(X > 1) = (2 > 1) = false \\ [\![while\ \beta]\!]([\![Y := Y \cdot X;\ X := X - 1]\!](Z)), \text{ sonst} \end{cases}$$

$$= [\![while\ \beta]\!]([\![Y := Y \cdot X; X := X - 1]\!](2,3))$$

$$= [\![while\ \beta]\!]([\![X := X - 1]\!]([\![Y := Y \cdot X]\!](2,3)))$$

$$= [\![while\ \beta]\!]([\![X := X - 1]\!](2,6))$$

$$= [\![while\ \beta]\!](1,6)$$

$$= \begin{cases} (1,6), \text{ falls } Z(X > 1) = (1 > 1) = false \\ [\![while\ \beta]\!]([\![Y := Y \cdot X;\ X := X - 1]\!](Z)), \text{ sonst} \end{cases}$$

$$= (1,6)$$

# Beispiel für Semantik imperativer Algorithmen (3)

- Dies bedeutet:

$$\begin{aligned} Z &= [\![\alpha]\!](Z_0) \\ &= [\![\alpha]\!](3, \bot) \\ &\dots \\ &= (1, 6) \end{aligned}$$

Damit gilt:

$$[\![FAC]\!](3) = Z(Y) = 6$$

# Was haben wir beobachtet?

- Der Übergang von der 3. auf die 4. Zeile folgt der Definition der Sequenz, indem der Sequenzoperator in einen geschachtelten Funktionsaufruf umgesetzt wird.
- Nur in der 5. Zeile wurde eine Wertzuweisung formal umgesetzt, später sind sie einfach verkürzt direkt ausgerechnet.
- In der 7. Zeile haben wir die Originaldefinition der Iteration eingesetzt (nur mit Kürzel α' statt α, da α bereits verwendet wurde)
  - Im Bsp.: $\alpha' = \{Y := Y \cdot X; X := X - 1\}$
- Das Z in der 7. und 8. Zeile steht für den Zustand (3,1). (In späteren Zeilen analog für den jeweils aktuellen Zustand.)
- Bei diesem Beispiel sieht man folgendes sehr deutlich: Die Ausführung einer **while**-Schleife erfolgt analog zur *rekursiven* Funktionsdefinition!

# Funktional vs. Imperativ – Beispiel

- Bekannt: Fibonacci-Zahlen funktional

```
fib(x)  :=  if (x==0) then 0
      else if (x==1)  then  1
           else  fib(x-1) + fib(x-2)
```

- Imperative Umsetzung:

```
FIB var X,A,B,C: int;
   input X;
     A := 0; B:=1; C:=1;
     while  X > 0 {
       C := A+B;
       A := B;
       B := C;
       X := X-1;
     }
   output A;
```

Bedeutung:
Für beliebige X gibt die Auswertung das Ergebnis von FIB(X).
→ Wir erkennen, der imperative Algorithmus FIB berechnet folgende Funktion:

$$[\![FIB]\!](x) = \begin{cases} \text{x-te Fib. Zahl} & \text{falls x} \geq 0 \\ 0 & \text{sonst} \end{cases}$$

# Größter gemeinsamer Teiler – Version 1

```
GGT1 var X,Y: int;
     input X,Y;
     while  X ≠ Y {
          while  X > Y   { X :=  X-Y; }
          while  X < Y   { Y :=  Y-X; }
     }
output X;
```

Auswertung:
(X=19,Y=5)

| X | Y |
|---|---|
| 19 | 5 |
| 14 | 5 |
| 9 | 5 |
| 4 | 5 |
| 4 | 1 |
| 3 | 1 |
| 2 | 1 |
| 1 | 1 |

Berechnung: Subtraktion jeweils kleinerer Zahl

Beobachtung: GGT mittels Subtraktion ist nicht unbedingt effizient!

# Größter gemeinsamer Teiler – Version 2

```
GGT2 var X,Y,R: int;
       input X,Y;
       R := 1
       while  R ≠ 0 {
            R := X % Y;  X := Y;  Y := R;
       }
output X;
```

Annahme: X > Y

Berechnung:
Modulo-Funktion

Auswertungen: (X=19,Y=5) und (X=1000,Y=2)

| X | Y | R |
|---|---|---|
| 19 | 5 | 1 |
| 5 | 4 | 4 |
| 4 | 1 | 1 |
| 1 | 0 | 0 |

| X | Y | R |
|---|---|---|
| 1000 | 2 | 1 |
| 2 | 0 | 0 |

Frage: Wie viele Schritte benötigt GGT1 für das zweite Beispiel? Ist GGT2 schneller?

# Größter gemeinsamer Teiler

```
GGT2 var X,Y,R: int;
     input X,Y;
     R := 1
     while  R ≠ 0 {
          R := X % Y;  X := Y;  Y := R;
     }
     output X;
```

- Wie ist das Verhalten für negative X oder Y?

$$[\![GGT2]\!](x,y) = \begin{cases} ggT(x,y), & \text{falls } x,y > 0 \\ y, & \text{falls } x = y \neq 0 \text{ oder } x = 0, y \neq 0 \\ \bot, & \text{falls } y = 0 \\ ggT(|x|,|y|), & \text{falls } x < 0 \; und \; y > 0 \\ -ggT(|x|,|y|), & \text{falls } y < 0 \end{cases}$$

# Vergleich GGT1 und GGT2

- Intuitiv ist GGT2 schneller als GGT1
  - Doch wie kann man das zeigen?

➔ Komplexität von Algorithmen (siehe nächstes Kapitel)

Algorithmen und Datenstrukturen

# 2. PROGRAMMIERPARADIGMEN
## ZUSAMMENFASSUNG

# Zusammenfassung

- Funktionale Programmierung
  - Funktionsauswertung
  - Rekursion
- Logische Programmierung
  - Hornlogik
  - Anfragen
  - Listen
- Imperative Algorithmen
  - Zustände
  - Wertzuweisung
  - Semantische Auswertung
  - Syntax