Übersicht

Einführung

- Programmierparadigma =
  - *fundamentaler Programmierstil* und bedingt damit auch
  - „Denkmuster" für Entwurf und Formulierung von Algorithmen
- Wir trennen von *Algorithmenparadigmen* [Saake&Sattler]
- Einige Paradigmen haben wir schon kennengelernt
  - *Strukturierte* Programmierung
  - *Imperative* Programmierung
  - *Objektorientierte* Programmierung ✓
- Denn: Java folgt *all diesen* Paradigmen
- Verschiedene Paradigmen können miteinander vereinbar sein
- Viele Programmiersprachen folgen mehreren Paradigmen,
  Dabei oft Fokus auf ein Paradigma (in Java: OOP)

Prozedurale und strukturierte Programmierung

- *Prozedurale* P. = Unterteilung in Unterprogramme
  - Unterprogramm (Prozedur/Funktion) löst kleineres Teilproblem
  - Lesbarkeit/Wartbarkeit und Wiederverwendung von Code
- *Strukturierte* P. = Prozedurale P. + *Kontrollstrukturen*
  - Sequenz
  - Fallunterscheidung (Auswahl, bedingte Anweisung)
  - Schleife (Iteration, bedingte Wiederholung)
- Bemerkungen
  - Prozedurale P. $\subset$ Strukturierte P.
  - z.B. keine `goto`* Anweisung ($\Rightarrow$ „Spaghetti-Code“)
  - Ziel: *Kostenreduktion*(!) für Software
  - Pascal (1972), C (1972)*, Modula-2 (1978), Ada (1983),...
  - Spezialfall: Objektorientierte P. $\subset$ Strukturierte P.

- Als nächstes:
  drei für Programmiersprachen fundamentale Paradigmen

Übersicht

Funktionale Programmierung

- Algorithmus = Menge von Funktionen
- Ausführen/Berechnen = Auswerten der Funktionen
- *Funktions*begriff im Sinn von mathematischer Funktion

$$
\begin{aligned}
f : \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n &\rightarrow \mathcal{Y}_1 \times \mathcal{Y}_2 \times \cdots \times \mathcal{Y}_m \\
(y_1, y_2, \ldots, y_m) &= f(x_1, x_2, \ldots, x_n)
\end{aligned}
$$

- Es gibt *keine* veränderlichen Daten!
- Es gibt *keinen* Zustand, den wir direkt beobachten könnten!

- Bemerkung zu [Saake&Sattler] (Kapitel 3.2)
  - Begriff *applikative* Programmierung ...
  - Im Sinne von Anwenden/Auswerten von Funktionsdefinitionen
  - Wir betrachten die Begriffe hier als synonym.

Terme und Unbestimmte

- Definition von Funktionen durch *Terme*
  - z.B. $f(x) = 5x + 1$
- Die Argumente, z.B. x, heißen *Unbestimmte*
  - Unbestimmte sind *keine* Variablen!
  - Unbestimmte sind Symbole und stehen als „Platzhalter“
- Im folgenden verwendete Konvention für Unbestimmte
  - x, y, z sind vom Typ `int` (repräsentieren Werte $x \in \mathbb{Z}$)
  - p, q, r sind vom Typ `bool` ($p \in \{\texttt{true}, \texttt{false}\}$)
- Konvention für Terme
  - Operationen auf $\mathbb{Z}$ (z.B. $(x+2)x$) und Vergleiche (z.B. $x < y$)
  - Logische Operationen $\neg p$ (*nicht*), $p \vee q$ (*oder*) , $p \wedge q$ (*und*)
  - Fallunterscheidung `if-else`
- Wir beschränken uns auf Funktionen $f : \mathcal{X}_1 \times \cdots \times \mathcal{X}_n \to \mathcal{Y}$

Undefinierte Werte

- Wir erweitern hier für alle Typen deren Wertemenge um $\perp$
- Symbol $\perp$ steht für *undefiniert*
- z.B.

$$\begin{aligned} \texttt{int}\ x &\Rightarrow x \in \mathbb{Z} \cup \{\perp\} \\ \texttt{bool}\ x &\Rightarrow x \in \{\texttt{true}, \texttt{false}\} \cup \{\perp\} \end{aligned}$$

- Wir betrachten hier ein Modell einer Programmiersprache:

  Es gibt keine exakte Entsprechung von $\perp$ in einer „echten“ Programmiersprache!

Funktionsdefinition

Definition (Funktionsdefinition)

Sind $v_1, \ldots, v_n$ Unbestimmte vom Typ $\tau_1, \ldots, \tau_n$ (`bool` oder `int`) 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 τ. Dabei ist τ der Typ des Terms.

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

Beispiele für Funktionsdefinitionen

- $f(x, y) = x + y$
- $f(x, y, z) = \texttt{if } x \leqslant y \wedge y \leqslant z \texttt{ then true else false fi}$
- $f(x) = \texttt{if } x \geqslant 0 \texttt{ then } x \texttt{ else } -x \texttt{ fi}$
- $f(x, y) = \texttt{if } x \leqslant y \texttt{ then } x \texttt{ else } y \texttt{ fi}$
- $f(p) = \texttt{if } p \texttt{ then } 1 \texttt{ else } 0 \texttt{ fi}$
- $f(p, q) = \texttt{if } p \texttt{ then } q \texttt{ else false fi}$

$$
\begin{aligned}
&f : \texttt{bool} \times \texttt{bool} \;\to\; \texttt{bool} \\
&f(\texttt{false}, \texttt{false}) = f(\texttt{false}, \texttt{true}) = f(\texttt{true}, \texttt{false}) = \texttt{false} \\
&f(\texttt{true}, \texttt{true}) = \texttt{true}
\end{aligned}
$$

$$f(p, q) = p \wedge q$$

Erweiterung der Funktionsdefinition

- Erweiterung der Definition von Termen:

  Auswertungen definierter Funktionen sind Terme

  - Beispiel: $f(x,y) = 2 \cdot g(x) + y, \quad g(x) = x - 1$

  $$f(3,1) \;=\; 2 \cdot g(3) + 1 \;=\; 2(3-1) + 1 \;=\; 4 + 1 = 5$$

- Damit sind auch *rekursive* Funktionen möglich

Definition (Rekursive Funktion)

Eine Funktion heißt *rekursiv*, wenn sie – direkt oder indirekt – durch sich selbst definiert ist.

- Beispiel: Fakultätsfunktion $x! = x \cdot (x-1) \cdot (x-2) \cdots 2 \cdot 1$

$$\mathbf{fac}(x) = \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \cdot \mathbf{fac}(x-1)\ \texttt{fi}$$

Beispiel: Fakultätsfunktion $x!$

- $\mathbf{fac}(x) = \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \cdot \mathbf{fac}(x-1)\ \texttt{fi}$
- Berechne $\mathbf{fac}(3)$

$$
\begin{aligned}
\mathbf{fac}(3) \quad &= \quad \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 \cdot \mathbf{fac}(3-1)\ \texttt{fi} \\
&= \quad \texttt{if}\ \texttt{false}\ \texttt{then}\ 1\ \texttt{else}\ 3 \cdot \mathbf{fac}(3-1)\ \texttt{fi} \\
&= \quad 3 \cdot \mathbf{fac}(3-1) \ = \ 3 \cdot \mathbf{fac}(2) \\
&= \quad 3 \cdot \big(\texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 \cdot \mathbf{fac}(2-1)\ \texttt{fi}\big) \ = \ \ldots \\
&= \quad 3 \cdot \big(2 \cdot \mathbf{fac}(2-1)\big) = 3 \cdot 2 \cdot \mathbf{fac}(1) \\
&= \quad (3 \cdot 2) \cdot \big(\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 \cdot \mathbf{fac}(1-1)\ \texttt{fi}\big) \ = \ \ldots \\
&= \quad (3 \cdot 2) \cdot \big(1 \cdot \mathbf{fac}(1-1)\big) \ = \ (3 \cdot 2 \cdot 1) \cdot \mathbf{fac}(0) \\
&= \quad (3 \cdot 2 \cdot 1) \cdot \big(\texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 0 \cdot \mathbf{fac}(0-1)\ \texttt{fi}\big) \\
&= \quad (3 \cdot 2 \cdot 1) \cdot \big(\texttt{if}\ \texttt{true}\ \texttt{then}\ 1\ \texttt{else}\ 0 \cdot \mathbf{fac}(0-1)\ \texttt{fi}\big) \\
&= \quad (3 \cdot 2 \cdot 1) \cdot 1 \ = \ 6
\end{aligned}
$$

Undefinierte Ergebnisse

- Eine Funktion muss nicht für alle Eingaben definiert sein.
- Was ergibt $\mathbf{fac}(-1)$?

$$\begin{aligned}
\mathbf{fac}(-1) &= \texttt{if}\ -1=0\ \texttt{then}\ 1\ \texttt{else}\ -1\cdot\mathbf{fac}(-1-1)\ \texttt{fi} \\
&= -\mathbf{fac}(-2) \\
&= -\big(\texttt{if}\ -2=0\ \texttt{then}\ 1\ \texttt{else}\ -2\cdot\mathbf{fac}(-2-1)\ \texttt{fi}\big) \\
&= +2\cdot\mathbf{fac}(-3)\ =\ \ldots\ =\ \textbf{?}
\end{aligned}$$

- Die Berechnung *terminiert nicht*!
- Das Ergebnis ist *undefiniert* !
- Wir beschreiben das semantisch als

$$\mathbf{fac}(x) = \begin{cases} x<0: & \perp \\ x=0: & 1 \\ x>0: & x\cdot(x-1)! \end{cases}$$

Undefinierte Werte $\Rightarrow$ Nicht-Terminierung

Undefinierte Werte

Der Wert einer Funktionsauswertung $f(x)$, die *nicht terminiert*, ist *undefiniert*! — Wir schreiben $f(x) = \perp$.

- Vorstellung von $\perp$ als *unendlich lange* Berechnung
- Konsequenz
  - $f(\ldots, \perp, \ldots) = \perp$ für alle Funktionen f
  - Jeder Vergleich mit $\perp$ liefert $\perp$!
    z.B. $(\perp = \perp) \to \perp$ und $(x = \perp) \to \perp$ und auch $(x \neq \perp) \to \perp$
- Beispiel

$$
\begin{aligned}
f(x) &= \texttt{if}\ f(x) = 0\ \texttt{then}\ 1\ \texttt{else}\ 0\ \texttt{fi} \\
f(x) &= \perp \quad \text{für alle } x \in \texttt{int}
\end{aligned}
$$

Definition (Partielle Funktion)

Eine Funktion, die nicht für alle Elemente ihrer Definitionsmenge (*alle möglichen Eingaben*) einen wohldefinierten Wert liefert, heißt *partiell*.

- Beispiel: $\mathbf{fac}(x) = \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x \cdot \mathbf{fac}(x-1)\ \texttt{fi}$

$$\mathbf{fac}(x) \;=\; \begin{cases} x < 0: & \perp \\ x = 0: & 1 \\ x > 0: & x \cdot (x-1)! \end{cases}$$

Weitere Beispiele

- Indirekte Rekursion **even**, **odd**
- Addition und Multiplikation von natürlichen Zahlen
- Primzahlen **prim** : `int` $\rightarrow$ `bool`
- Fibonacci Zahlen
- McCarthys 91-Funktion

Beispiel: **even** und **odd**

- Regeln

$$\begin{aligned}\textbf{even}(0) &= \texttt{true}\\ \textbf{odd}(0) &= \texttt{false}\\ \textbf{even}(x+1) &= \textbf{odd}(x)\\ \textbf{odd}(x+1) &= \textbf{even}(x)\end{aligned}$$

- Rekursive Definition mit wechselseitiger Auswertung

$$\begin{aligned}\textbf{even}(x) &= \texttt{if}\ x=0\ \texttt{then true else}\ \textbf{odd}(x-1)\ \texttt{fi}\\ \textbf{odd}(x) &= \texttt{if}\ x=0\ \texttt{then false else}\ \textbf{even}(x-1)\ \texttt{fi}\end{aligned}$$

- Partielle Funktionen! – Erweiterung auf $\mathbb{Z}$

$$\begin{aligned}\textbf{even}(x) &= \texttt{if}\ x=0\ \texttt{then true else}\\ &\ \texttt{if}\ x>0\ \texttt{then}\ \textbf{odd}(x-1)\ \texttt{else}\ \textbf{odd}(x+1)\ \texttt{fi fi}\\ \textbf{odd}(x) &= \texttt{if}\ x=0\ \texttt{then false else}\\ &\ \texttt{if}\ x>0\ \texttt{then}\ \textbf{even}(x-1)\ \texttt{else}\ \textbf{even}(x+1)\ \texttt{fi fi}\end{aligned}$$

Beispiel: **even** und **odd**

- Berechne $\textbf{even}(3)$

$$
\begin{aligned}
\textbf{even}(3) &= \texttt{if } 3=0 \texttt{ then true else } \textbf{odd}(3-1) \texttt{ fi} \\
&= \textbf{odd}(2) \\
&= \texttt{if } 2=0 \texttt{ then false else } \textbf{even}(2-1) \texttt{ fi} \\
&= \textbf{even}(1) \\
&= \texttt{if } 1=0 \texttt{ then true else } \textbf{odd}(1-1) \texttt{ fi} \\
&= \textbf{odd}(0) \\
&= \texttt{if } 0=0 \texttt{ then false else } \textbf{even}(0-1) \texttt{ fi} \\
&= \texttt{false}
\end{aligned}
$$

Addition von natürlichen Zahlen

- Addition $x+y$ basierend auf
  - *Nachfolger*funktion $\textbf{succ}(x) = x+1$ und
  - *Vorgänger*funktion $\textbf{pred}(x) = x-1$ (*partiell*)
- Regeln

$$\begin{array}{rcl} x+0 & = & x \\ x+y & = & (x+1)+(y-1) \\ & = & \textbf{succ}(x)+\textbf{pred}(y) \quad \text{für } y>0 \end{array}$$

- Umsetzung

$$\textbf{add}(x,y) \;=\; \texttt{if}\ y=0\ \texttt{then}\ x$$
$$\texttt{else}\ \textbf{add}(\textbf{succ}(x),\textbf{pred}(y))\ \texttt{fi}$$

Addition nur mit **succ**

- Vermeide partielle Vorgängerfunktion **pred**
- Definiere (dreistellige) Hilfsfunktion

$$\mathbf{add3}(x,y,z) \;=\; \texttt{if}\ z=y\ \texttt{then}\ x\ \texttt{else}\ \mathbf{add3}(\mathbf{succ}(x),y,\mathbf{succ}(z))\ \texttt{fi}$$

  - Ergebnis x wie in **add**: es wird immer um 1 erhöht
  - y bleibt *unverändert*
  - z „zählt“ $0,1,\ldots,y$

- Damit

$$\mathbf{add}(x,y) = \mathbf{add3}(x,y,0)$$

- Erweiterung auf $y \in \mathbb{Z}$? z.B. analog **even**, **odd**

Multiplikation von natürlichen Zahlen

- Definiere Multiplikation $x \cdot y$ durch Addition
- Regeln

$$
\begin{aligned}
x \cdot 0 &= 0 \\
x \cdot y &= x \cdot (y-1) + x \\
&= \mathbf{add}(\mathbf{mult}(x, \mathbf{pred}(y)), x) \qquad \text{für } y > 0
\end{aligned}
$$

- Umsetzung

$$
\mathbf{mult}(x,y) = \texttt{if } y = 0 \texttt{ then } 0 \\
\texttt{else } \mathbf{add}(\mathbf{mult}(x, \mathbf{pred}(y)), x) \texttt{ fi}
$$

- Ohne **pred**: analog **add**, **add3**

Grundrechenarten auf $\mathbb{N}_0$ bzw. $\mathbb{Z}$

- Definition von der natürlichen Zahlen durch Nachfolger
- Grundrechenarten auf **succ** (und **pred**) zurückführbar
- Analog zu **add** und **mult**:
  - **pow**$(x, y) = x^y$
  - **sub**$(x, y) = x - y$ (partiell auf $\mathbb{N}_0$)
  - **div**$(x, y) = \lfloor \frac{x}{y} \rfloor$ (partiell)
  - **mod**$(x, y) = x \bmod y$ (partiell)
- Übungen: alternative, „schnellere" Konstruktion für **mult**, **pow**

Beispiel: Ist x Primzahl?

Definition (Primzahl)

Eine *Primzahl* ist eine natürliche Zahl, die größer als eins und nur durch sich selbst und durch eins teilbar ist.

- Definiere **prime** : $\texttt{int} \rightarrow \texttt{bool}$

$$
\begin{aligned}
\mathbf{prime}(x) &= \texttt{if}\ x < 0\ \texttt{then}\ \mathbf{prime}(x)\ \texttt{else} \\
&\ \texttt{if}\ x \leqslant 1\ \texttt{then false else}\ \mathbf{pr}(x,2)\ \texttt{fi fi} \\
\mathbf{pr}(x,y) &= \texttt{if}\ y \geqslant x\ \texttt{then true} \\
&\ \texttt{else}\ (\mathbf{mod}(x,y) \neq 0)\ \wedge\ \mathbf{pr}(x, \texttt{succ}(y))\ \texttt{fi}
\end{aligned}
$$

- **pr** testet Teilbarkeit für alle $2 \leqslant y < x$ $(\mathbf{mod}(x,y) = x \bmod y)$
- **prime** ist *partiell*: $\mathbf{prime}(x) = \perp$ für $x < 0$

Effizientere Variante von **prime**

1. Ersetze $y \geqslant x$ durch $y^2 > x$

   Denn für kleinsten Teiler k von x muss gelten $k \leqslant \sqrt{y}$

2. Teste Teilbarkeit durch 2 und keine weiteren *geraden* Zahlen

$$
\begin{aligned}
\mathbf{pr}(x,y) \quad = \quad & \texttt{if } y^2 > x \texttt{ then true} \\
& \texttt{else if } y = 2 \texttt{ then } (\mathbf{mod}(x,2) \neq 0) \ \wedge \ \mathbf{pr}(x,3) \\
& \texttt{else } (\mathbf{mod}(x,y) \neq 0) \ \wedge \ \mathbf{pr}(x, \texttt{succ}(\texttt{succ}(y))) \texttt{ fi}
\end{aligned}
$$

- Teilbarkeit testen $(\mathbf{mod}(x,y) \neq 0)$, falls nicht teilbar...
- Für $y = 2$: weiter mit 3
- Für $y \neq 2$: y ist ungerade, weiter mit $y + 2 = \mathbf{succ}(\mathbf{succ}(y))$

Fibonacci Folge

- Zahlenfolge $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- Regel

$$
\begin{aligned}
f_0 &= 0 \\
f_1 &= 1 \\
f_i &= f_{i-1} + f_{i-2} \qquad \text{für } i \geqslant 2
\end{aligned}
$$

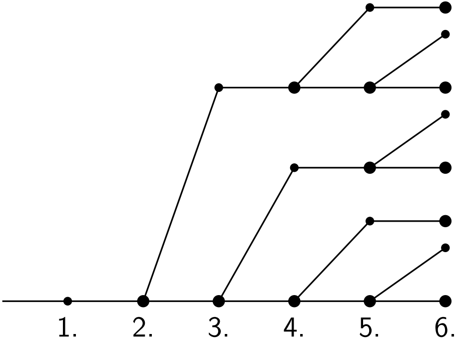
- Diese Folge kommt immer wieder bei Wachstumsvorgängen in der Natur (und auch für Algorithmen) vor.
- Viele interessante Eigenschaften u.a.
  Verwandtschaft mit dem *Goldenen Schnitt*

$$
\phi = \lim_{i \to \infty} \frac{f_{i+1}}{f_i} = \frac{1 + \sqrt{5}}{2}
$$

- Ein Beispiel: Stammbaum von Kaninchen

Kaninchen und Fibonacci Zahlen

- Zwei Lebensphasen $\bullet$ (klein) und $\bullet$ (groß=geschlechtsreif)
- Innerhalb eines Monats $\bullet \to \bullet$
- Jedes $\bullet$-Paar zeugt jeden Monat ein neues $\bullet$-Paar.
- Beginne im 1. Monat mit einem $\bullet$-Paar.
- Kaninchen sind monogam und unsterblich.



- Anzahl der Kaninchenpaare im i. Monat $= f_i$

Berechnung von Fibonacci Zahlen

- Rekursive Berechnung nach der Regel $f_0 = 0,\ f_1 = 1$ und $f_i = f_{i-2} + f_{i-1}$ für $i \geqslant 2$
- Umsetzung

$$\begin{aligned}\textbf{fib}(x) \quad = \quad & \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else} \\ & \texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ \textbf{fib}(x-2) + \textbf{fib}(x-1)\ \texttt{fi}\ \texttt{fi}\end{aligned}$$

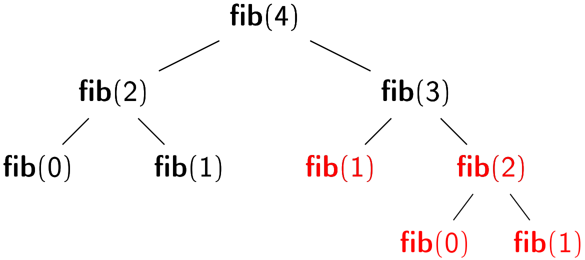
- Rekursion „anderer“ Art: **fib** *zweimal* im gleichen Zweig
- Beispiel: Berechne $\textbf{fib}(4)$

$$\begin{aligned}\textbf{fib}(4) \quad &= \quad \textbf{fib}(2) + \textbf{fib}(3) \\ &= \quad \big(\textbf{fib}(0) + \textbf{fib}(1)\big) + \big(\textbf{fib}(1) + \textbf{fib}(2)\big) \\ &= \quad \big(\textbf{fib}(0) + \textbf{fib}(1)\big) + \big[\textbf{fib}(1) + \big(\textbf{fib}(0) + \textbf{fib}(1)\big)\big] \\ &= \quad (0+1) + [1 + (0+1)] \ = \ 1 + [1+1] \ = \ 1+2 \ = \ 3\end{aligned}$$

- Offensichtlich sind einige Auswertungen *redundant*!

Auswertung von **fib**

- Wir stellen die rekursive Auswertungen als *Baum* dar.



- Die markierten Auswertungen sind redundant.
- Wir zählen wie folgt
  - Je 1 Auswertung für **fib**(0) und **fib**(1)
  - Für **fib**(x): Summe Auswertungen für **fib**($x-2$) und **fib**($x-1$)
  - *Gleiches Prinzip!* $\Rightarrow$ allgemein f_{x+1} Auswertungen nötig
- f_i wächst *exponentiell*, da $f_i = \frac{1}{\sqrt{5}}\left(\phi^i - (1-\phi)^i\right) \approx \frac{1}{\sqrt{5}}\phi^i$

Effizientere Auswertung von **fib**

- Wir würden die Fibonacci Zahlen wohl nicht so berechnen!
- Denn eigentlich sind für **fib**(4) nur 4 Auswertungen nötig.
- Lösung: Zwischenergebnisse speichern und einsetzen
- Mögliche „iterative" Umsetzung

$$\begin{aligned}
\textbf{ifib}(x) &= \texttt{if}\ x=0\ \texttt{then}\ 0\ \texttt{else} \\
&\ \texttt{if}\ x=1\ \texttt{then}\ 1\ \texttt{else}\ \textbf{ifib3}(x,0,1)\ \texttt{fi}\ \texttt{fi} \\
\textbf{ifib3}(x,y,z) &= \texttt{if}\ x=2\ \texttt{then}\ y+z \\
&\qquad\qquad \texttt{else}\ \textbf{ifib3}(x-1,z,y+z)\ \texttt{fi}
\end{aligned}$$

- x zählt Anzahl „Iterationen" i
- y und z speichern Zwischenergebnisse f_{i-2} und f_{i-1}
- Beispiel

$$\begin{aligned}
\textbf{ifib}(6) &= \textbf{ifib3}(6,0,1) = \textbf{ifib3}(5,1,1) = \textbf{ifib3}(4,1,2) \\
&= \textbf{ifib3}(3,2,3) = \textbf{ifib3}(2,3,5) = 3+5 = 8
\end{aligned}$$

Was berechnet die folgende Funktion?

- Zum Abschluss eine etwas kuriose Funktion

$$f(x) \;=\; \texttt{if}\ x > 100\ \texttt{then}\ x - 10\ \texttt{else}\ f(f(x+11))\ \texttt{fi}$$

- Nicht einfach zu sehen. Wir probieren ...

$$\begin{aligned}
f(100) &= f(f(111)) = f(101) = 91 \\
f(99) &= f(f(110)) = f(100) = \cdots = 91 \\
f(98) &= f(f(109)) = f(99) = \cdots = 91 \\
\cdots &= \cdots = 91 \;\; ?
\end{aligned}$$

- Vermutung f ist äquivalent zu g mit

$$g(x) \;=\; \texttt{if}\ x > 100\ \texttt{then}\ x - 10\ \texttt{else}\ 91\ \texttt{fi}$$

- Äquivalenz besteht tatsächlich: *McCarthys 91-Funktion*, *Beweis folgt später* (Korrektheit von Algorithmen)
- Frage: Kann man Äquivalenz von Algorithmen zeigen? Wie?

Ausblick: Funktionale Programmierung

- Bisher: Auswertung von Termen und *Rekursion*
  - Keine Schleifen
  - Keine Variablen
- Alle bisherigen Beispiele lassen sich genauso in Java umsetzen.
  - *Das ist eine gute Übung für die Klausur!*
- Was macht *Funktionale Programmierung* noch aus?
- Wesentlich: Funktionen als „Objekte" der Sprache
  - Funktions-Typen (z.B. `int` $\to$ `int`) sind Typen (wie z.B. `int`)
  - Funktionen können erzeugt werden (λ-Operator)

Definition (Funktional)

Eine Funktion, die eine Funktion als Argument erhält oder eine Funktion als Ergebnis liefert, heißt *Funktional* oder *Funktion höherer Ordnung*.

- Zentrales Element von funktionalen Programmiersprachen
- Alternative Schreibweise für $f(x, y)$ mit $f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ als

$$f\ x\ y \quad \text{mit} \quad f : \mathcal{X} \to (\mathcal{Y} \to \mathcal{Z})$$

damit definiert

$$g = f\ x$$

eine neue *Funktion* $g : \mathcal{Y} \to \mathcal{Z}$,
für die der Wert von x gebunden und y unbestimmt ist:

$$g(y) = f(x, y)$$

Ausblick: Funktionen erzeugen

- Funktionen können wie „Objekte“ erzeugt werden
- Wir schreiben hier $(\cdot) \to \cdot$

$$
\begin{aligned}
f(x,y) &= x+y & \qquad f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\
f(1,2) &= 1+2 = 3 & \\
g(x) &= (y) \to f(x,y) & \qquad g : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z}) \\
g(1) &= (y) \to f(1,y) = (y) \to 1+y & \\
(g(1))(2) &= 1+2 = 3 &
\end{aligned}
$$

- Wie eben: 2-stellige Funktion $f(x,y)$ wird zu 1-stelliger Funktion $g(x)$ durch Binden eines Arguments (*Currying*).

Typische Elemente Funktionaler Programmiersprachen

- Viele Sprachen haben ein Typsystem z.B. Haskell, ML
  - Typen können aus Definitionen abgeleitet werden
  - Typsystem nicht zwingend
- Oft Mustervergleiche (*pattern matching*), z.B. Haskell, ML
- Funktionen höherer Ordnung
- Umsetzung des *Lambda-Kalkül*
  - „Erzeugung“ von (anonymen) Funktionen
- Verschiedene Auswertungsstrategien insb. *lazy evaluation*
- Einfache aber mächtige Operationen auf Listen
  - Rekursion mit *head* (erstes Element) und *tail* (Rest)
  - *map*: Anwendung einer Funktion auf jedes Listenelement
  - *fold* (auch *inject, reduce, accumulate*): z.B. Summenbildung
- Meist keine *rein* funktionalen Sprachen
  - Variablen, . . . (Elemente imperativer P., auch OOP)

Abschließende Bemerkungen

- Wir hören auf, wenn es anfängt, spannend zu werden!
- Sonst müssten wir eine *funktionale* Programmiersprache lernen
- Warum ist es eine gute Idee, das (später) noch zu tun?
  - Auf den ersten Blick vielleicht gewöhnungsbedürftig, aber ...
  - Funktionale Programmierung macht Spaß!
  - Oft elegante, kurze, gut lesbare Programme
  - Programme ggf. einfach automatisch parallelisierbar
- Konzepte Funktionaler Programmierung im „Alltag"
  - Computeralgebrasysteme, z.B. Maxima, Maple, Mathematica
  - Populäre Scriptsprachen wie z.B. Python oder Ruby
  - Teile der C++ Standard- bzw. boost Bibliotheken
  - *Template metaprogramming* in C++
  - Neue Konzepte seit C++11
  - Programmierung des GNU Emacs Editors ;-)
- Beispiele von Elementen Funktionaler Programmierung ...

- Java definiert *lambda expressions*

```
// f : ℤ×ℤ→ℤ
java.util.function.BinaryOperator<Integer>
  f = (x,y) -> { return x+y; };

int z = f.apply(1,2);

System.out.println(f.apply(1,2)); // ⇒ 3
```

- Den Typen von `f` schreiben wir i.d.R. nicht explizit
- Seit Java 8 brauchbar und oft sehr praktisch(!), aber ...
- ... Möglichkeiten vergleichsweise eingeschränkt!
- Wir benötigen diese Schreibweise *nicht weiter*!

Ein JavaScript-Beispiel

```
let xhr = new XMLHttpRequest();
xhr.responseType = 'text';
xhr.open('GET', 'example.net/some/route');
xhr.onload = function() {
  if (xhr.status == 200) {               // OK
    // ... use result ...
    console.log(xhr.responseText);
  }
};
xhr.send(); // initiate request ...
            // ... and go on ...
```

- Anfrage an Server soll *nicht* auf die Antwort *warten*, sonst würde z.B. das Browser-Fenster solange „blockiert".
- Sobald eine gültige Antwort übermittelt wurde, wird die angegebene anonyme Funktion aufgerufen.
- Beachte: Dort ist Anfrage in `xhr` gebunden.

Kurze Zusammenfassung

- Funktionale Programmierung kennt keine Variablen!
    - Kein veränderbarer Zustand
    - (Außer Zustand des Auswertalgorithmus selbst)
- Termauswertung
    - Sequenz
    - Fallunterscheidung
- Funktionsauswertung
    - Rekursion ersetzt Schleifen
    - Rekursion als ein zentrales Element
- *Soweit alles noch in Java möglich!* — Ausprobieren!
- Funktionale Programmiersprachen
    - Erlauben Funktionen höherer Ordnung
    - „Rechnen mit Funktionen“ (λ-Kalkül)
- Als nächstes: *imperative Programmierung*

Übersicht

6 Programmierparadigmen

- Einführung
- Funktionale Programmierung
- Imperative Programmierung
- Logische Programmierung
- Zusammenfassung und Ausblick

Imperative Programmierung

- Algorithmus = Sequenz von Anweisungen
- Auswerten einer Anweisung = Zustandsänderung
- Zustand = Werte von Variablen
- Schrittweise Manipulation von veränderlichen Daten (Zustand)
- Orientierung an einem einfachen Prozessormodell
  - Abarbeiten von „Befehlen“
  - *später:* Registermaschine
- Erweiterung durch Strukturierte/Prozedurale/OO P.
  - Ohne diese Erweiterungen nicht praktikabel!
  - Java fällt damit in die Klasse imperativer Programmiersprachen
- Formale Beschreibung siehe z.B. [Saake&Sattler] (Kapitel 3.3)

Variablen

- Eine *Variable* besteht aus
  - einem eindeutigen *Bezeichner* (Namen, z.B. X) und
  - einem veränderlichen *Wert* (von einem bestimmten Typ)
- Die Anweisung $\mathrm{X} := \mathrm{t}$ heißt *Wertzuweisung*
  - X bezeichnet eine Variable
  - t ist ein Term (ohne Unbestimmte) mit Wert $w(\mathrm{t})$
  - t darf Variablen enthalten (auch X selbst)
- Semantik der Wertzuweisung $\mathrm{X} := \mathrm{t}$
  - Nach Ausführung von $\mathrm{X} := \mathrm{t}$ gilt $\mathrm{X} = w(\mathrm{t})$
- Vor Ausführung der ersten Wertzuweisung gilt $\mathrm{X} = \perp$

Zustand und Zustandstransformation

- *Zustand* = partielle Abbildung $Z : \mathcal{V} \to \mathcal{W}$
  - Menge von Variablen $\mathcal{V}$
  - Wertemenge $\mathcal{W}$ (hier alle Variablen vom gleichen Typ)
- Abbildung Z ordnet Variablen ihren momentanen Wert zu
  - *Vereinfacht*: Zustand als Menge von Variablen
- Zuweisung $X := t$ *transformiert* Z in neuen Zustand Z'
  - Dabei ändert sich der Wert der Variablen in $X \in \mathcal{V}$
  - Zustandstransformation als Funktion
- Komplexe Anweisungen durch
  - Sequenz
  - Auswahl (Fallunterscheidung `if`-`else`)
  - Iteration (`while` Schleife)
- Formale Definition in [Saake&Sattler] (Kapitel 3.3)
  - Definiere Semantik durch Konstruktion von Transformationen
  - Bedeutung intuitiv klar, wir benötigen Formalismus nicht weiter

Kurze Charakterisierung

- Algorithmus wird beschrieben durch Folge von Anweisungen
- Anweisung =
  - Wertzuweisung als elementare Anweisung
  - Sequenz = Folge von Anweisungen
  - Auswahl = bedingte Ausführung
  - Iteration = bedingte Wiederholung (Schleife)
  - Bedingung = Wahrheitswert abhängig von Zustand
- Jede elementare Anweisung $\Rightarrow$ Transformation des Zustands
- Zustand = Zuordnung von Werten zu Variablen(-namen)
- Bemerkungen
  - Definiere Iteration rekursiv $\Rightarrow$ Schleife muss nicht terminieren
  - Rekursion $\rightarrow$ Iteration: gleiche Mächtigkeit von imperativen und funktionalen Sprachen
  - Sprachelemente bereits ausreichend für *universelle* Programmiersprache

Syntax für Beispiele

- Wir verwenden eine einfache, fiktive imperative Sprache
- Beschränkung auf Typen `int` und `bool`
- *Terme* wie bisher aber
  - Variablen statt Unbestimmte
  - Keine Funktionsauswertung in Termen (keine Funktionen!)
  - Keine Fallunterscheidung in Termen
- Auswahl `if` P `then` α `else` β `fi`
- Iteration `while` P `do` α `od`
- Ein Programm besteht aus
  - Programmname
  - `var` $X, Y, \ldots$: `int` — Variablendeklaration
    $P, Q, \ldots$: `bool`
  - `input` $X_1, \ldots, X_n$ — Eingabe-Variablen
  - a — Anweisungen
  - `output` $Y_1, \ldots, Y_m$ — Ausgabe-Variablen

Beispiel: Fakultätsfunktion $x!$

- Berechne $x! = x \cdot (x-1) \cdot (x-2) \cdots 2 \cdot 1$

```
FAC
var      X,Y : int
input    X
Y:=1;
while X≠0 do
  Y:=Y*X;
  X:=X-1;
od
output Y
```

- Bezeichne mit $[\![\mathtt{FAC}]\!](x)$ die Auswertung mit Eingabe $X = x$
- Es gilt $[\![\mathtt{FAC}]\!](x) = \begin{cases} x \geqslant 0: & x! \\ x < 0: & \perp \end{cases}$

Beispiel: Auswertung von FAC(3)

- Bedeutung der einzelnen Schritte *intuitiv* klar

```
FAC:
var     X,Y : int;
input   X;
Y:=1;
while X≠0 do
  Y:=Y*X;
  X:=X-1;
od;
output Y;
```

#	Anweisung	X	Y
0	(input)	3	$\perp$
1	Y:=1	3	1
2	Y:=Y*X	3	3
3	X:=X-1	2	3
4	Y:=Y*X	2	6
5	X:=X-1	1	6
6	Y:=Y*X	1	6
7	X:=X-1	0	6
8	(output)	0	6

- *Formale* Auswertung von [FAC](3) siehe [Saake&Sattler]

Beispiel: Fibonacci Zahlen

- *Iterative* Berechnung nach der Regel $f_0 = 0,\ f_1 = 1$ und $f_i = f_{i-2} + f_{i-1}$ für $i \geqslant 2$

```
FIB:
var      X,Y,Z,W : int;
input    X;
if       X=0 then Y:=0;
else
  Y:=0; Z:=1;               # f_{i-2},f_{i-1}
  while X>1 do
    W:=Y; Y:=Z; Z:=Z+W; # temporary W=Y
    X:=X-1;
  od;
  Y:=Y+Z;
fi;
output Y;
```

- `FIB` entspricht der funktionalen Implementierung `ifib`

Weitere Beispiele

- *Einige Beispiele kennen wir schon in Java !*
- Euklids Algorithmus: berechne größten gemeinsamen Teiler
  - Rekursive und iterative Variante
  - Beispiele für iterative Auswertung in [Saake&Sattler]
- Berechnung von Primzahlen
  - Iterative Variante: Erste n Primzahlen
  - Rekursive Variante: Entscheide ob x prim
- Berechnung der Quadratwurzel nach Heron
- Als Übung z.B.
  - Addition und Multiplikation von natürlichen Zahlen
  - z.B. Umsetzung in Java (ggf. mit Ausgabe des Zustands)
- Abschließend:
  Frage nach der Semantik eines gegebenen Algorithmus

Was berechnet das folgende Programm?

- Das folgende Programm beschreibt einen Algorithmus.
- Es ist nicht einfach zu sehen, welchen . . . !?

```
XYZ:
var     W,X,Y,Z : int;
input   X;
Z:=0; W:=1; Y:=1;
while W⩽X do
  Z:=Z+1; W:=W+Y+2; Y:=Y+2;
od;
output Z;
```

- Eine Möglichkeit: verschiedene Eingabewerte probieren
- Immer noch schwer!
- *Später:* Wir zeigen $[\![\mathtt{XYZ}]\!](X) = \lfloor \sqrt{X} \rfloor$

Kurze Zusammenfassung

- *Imperative Programmierung* beschreibt Algorithmen durch
  - Folge von Anweisungen, die
  - den *Zustand* des Programms verändern
- Gegensatz zur Funktionalen Programmierung
  - Dort gibt es *keine* Zustandsänderungen
- Zustand = Menge von *Variablen* mit Werten
- Anweisungen =
  - Wertzuweisung
  - Sequenz von Anweisungen
  - Auswahl
  - Iteration
- In der „reinen Form“: Keine Funktionsaufrufe, keine Rekursion!
  - Erweiterung durch Strukturierte/Prozedurale Programmierung
- Objektorientierte Programmierung als Erweiterung
  - Zustand in Objekten gekapselt
  - Zustandsänderung nur im Kontext des Objekts (Methoden)

Zwischenstand: Funktionale vs Imperative Programmierung

- *Imperative* Programmierung
  - Intuitiv (Anweisung = Handlung, z.B. Kochrezept)
  - Strukturierte P./OOP $\Rightarrow$ komplexe Algorithmen beherrschbar
  - Grundlage für viele, weit verbreitete Sprachen, z.B. Fortran, Pascal, Modula-2, Ada, C/C++, Java, ...
- *Funktionale* Programmierung
  - Intuitiv (Code ähnelt oft mathematischer Vorschrift)
  - *Aber* ggf. gewöhnungsbedürftig
  - Weniger verbreitet, z.B. Lisp, Scheme, Clojure, ML, Haskell, Ocaml, Scala
- Probleme durch Zustandsänderungen
  - Lesbarkeit beeinträchtigt (*zeitliche* Abfolge wichtig!)
  - Korrektheit ggf. schwerer beweisbar
  - Schwerer optimierbar, parallelisierbar (durch Compiler)
  - Vorsicht: Seiteneffekte

Zur Risiken und Nebenwirkungen ...

Seiteneffekt

Als Seiteneffekt (*side effect*) bezeichnet man jede Art von *bleibender Veränderung*, die *nach* Abarbeitung einer – potentiell komplexen – Anweisung bestehen, d.h. beobachtbar, bleibt.

- Oft ist explizit eine „nebensächliche“ Änderung gemeint
- z.B. Zählvariable im Gegensatz zu `output` Variable
- Deshalb auch Abgrenzung als Nebenwirkung (synonym)
- Ein einfaches Beispiel

```
X:=Y+1;
```

```
Y:=Y+1;
X:=Y;
```

- Beide Sequenzen liefern das Ergebnis $X = Y + 1$.
- Rechts zusätzlich Änderung von $Y \Rightarrow$ *Seiteneffekt*
- Seiteneffekte wo möglich vermeiden! (Vorsicht bei ++ und -- !)

Aus der Praxis. . .

„Bei mir funktioniert es! – Euer Test muss falsch sein!!!“

```
public class Fibonacci {
  static int[] f = { 0, 1 };

  public static int fib(int n) {
    while (n>=1) {
      int z = f[0];
      f[0] = f[1];
      f[1] += z;

      --n;
    }

    return f[0];
  }
}
```

Logische Programmierung

- Sachverhalt wird beschrieben durch logische *Aussagen*
- Für eine *Anfrage* wird durch *Deduktion* eine *Antwort* ermittelt
  - Schlussfolgerung
  - Ableiten von neuen Aussagen aus bestehenden
  - auch *Deduktive Programmierung*
- Logische Aussagen alleine ergeben kein Berechnungsmodell!
- Benötige Interpretation: Deduktionsalgorithmus
- Beispiel
  - Menge von Aussagen = Kriminalfall
  - Deduktionsalgorithmus = Sherlock Holmes
  - "*A case of simple deduction, Watson.*"
- Diese Rolle übernimmt die Programmiersprache
  - z.B. Prolog-Interpreter
- *Im folgenden stark vereinfachte Darstellung.*

Aussagen und Aussageformen

- Aussage z.B. Susi *ist die Tochter von* Petra.
- Aussageform = Aussage mit Unbestimmten
  - X *ist Tochter von* Y.
- Belegung transformiert Aussageform in Aussage
  - $X \mapsto \text{Susi}$, $Y \mapsto \text{Petra}$
- Atomare Formeln ersetzen natürliche Sprache, z.B.
  - $\texttt{Tochter}(\text{Susi}, \text{Petra})$ *Aussage*
  - $\texttt{Tochter}(X, Y)$ *Aussageform*
- *Im Vergleich*:
  Atomare Formeln entsprechen (ungeschachtelten) booleschen Funktionstermen
  - z.B. $\texttt{tochter} : \mathcal{P} \times \mathcal{P} \to \texttt{bool}$ mit Menge aller Personen $\mathcal{P}$

Logik der Fakten und Regeln

- Alphabet
  - Unbestimmte $X, Y, Z, \ldots$
  - Konstanten $a, b, c, \ldots$
  - Prädikatensymbole $\mathtt{P}, \mathtt{Q}, \mathtt{R}, \ldots$ (mit Stelligkeit), z.B. `Tochter`
  - Logische Verknüpfungen (Konnektive, Junktoren)
    - Konjunktion $\wedge$
    - Implikation $\Rightarrow$
    - *Keine* Negation $\neg$ oder Disjunktion $\vee$!
- Atomare Formeln: $P(t_1, \ldots, t_n)$
- Fakten = alle t_i sind Konstanten (P sind ohne Unbestimmte)
  - z.B. $\mathtt{Tochter}(\text{Susi}, \text{Petra})$
- Regeln $= \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m \;\Rightarrow \alpha_0$
  - Atomare Formeln α_i
  - $\alpha_1 \wedge \cdots \wedge \alpha_m$ heißt *Prämisse* (dabei *leere* P. immer wahr)
  - α_0 heißt *Konklusion*

Beispiel: Ableitung neuer Fakten aus Fakten und Regeln

- Fakten

$$\texttt{Tochter}(\mathsf{Susi}, \mathsf{Petra})$$
$$\texttt{Tochter}(\mathsf{Petra}, \mathsf{Rita})$$

- Regel

$$\texttt{Tochter}(X, Y) \wedge \texttt{Tochter}(Y, Z) \quad \Rightarrow \quad \texttt{Enkelin}(X, Z)$$

- Ableitung neuer Fakten durch Implikation $\Rightarrow$
  - Finde *Belegung* der Unbestimmten in einer Regel, so dass
  - als *Prämisse* (linke Seite) bekannte Fakten stehen, dann
  - $\Rightarrow$ rechte Seite ergibt neuen Fakt
- Diese *Schlussregel* $P \wedge (P \Rightarrow Q) \;\Rightarrow\; Q$ heißt *modus ponens*
- Im Beispiel

$$X \mapsto \mathsf{Susi}, \quad Y \mapsto \mathsf{Petra}, \quad Z \mapsto \mathsf{Rita}$$
$$\Rightarrow \quad \texttt{Enkelin}(\mathsf{Susi}, \mathsf{Rita})$$

- „Algorithmus" D gegeben als Menge von *Fakten und Regeln*
- $F(D)$ sind alle aus D direkt oder indirekt *ableitbaren* Fakten
- Keine „Ausgabegabefunktion" — stattdessen Anfragen
- Anfrage γ ist eine Konjunktion von atomaren Formeln α_i

$$\gamma \; = \; \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_m$$

- Eine Antwort ist eine *Belegung* der Unbestimmten in γ, bei der aus allen α_i Fakten werden.
- Enthält γ keine Unbestimmten, dann Antwort $\in \{\texttt{true}, \texttt{false}\}$
- Beispiele
  - $\gamma = \texttt{Tochter}(\text{Susi}, \text{Rita}) \to$ Antwort `false`
  - $\gamma = \texttt{Enkelin}(X, \text{Rita}) \to$ Antwort $\{X = \text{Susi}\}$

Algorithmus zur Beantwortung von Anfragen?

- Beispiel: Anfrage $\gamma = \texttt{Enkelin}(X, \text{Rita})$
- Fakten: $\texttt{Tochter}(\text{Susi}, \text{Petra})$, $\texttt{Tochter}(\text{Petra}, \text{Rita})$
- Regel: $\texttt{Tochter}(X, Y) \wedge \texttt{Tochter}(Y, Z) \;\Rightarrow\; \texttt{Enkelin}(X, Z)$
- Deduktion

$$
\begin{array}{lcl}
\texttt{Enkelin}(X, \text{Rita}) & \underset{Z=\text{Rita}}{\mapsto} & \texttt{Tochter}(X, Y) \wedge \texttt{Tochter}(Y, \text{Rita}) \\
& \underset{Y=\text{Petra}}{\mapsto} & \texttt{Tochter}(X, \text{Petra}) \wedge \texttt{true} \\
& \underset{X=\text{Susi}}{\mapsto} & \texttt{true}
\end{array}
$$

- Antwort: $\{X = \text{Susi}\}$

Algorithmus zur Beantwortung von Anfragen

- Grundidee
  - Starte mit Anfrage γ
  1. Untersuche* Belegungen, die
     - einen Teil von γ mit Fakten gleichsetzen bzw.
     - einen Fakt aus γ mit einer rechten Seite der Regel gleichsetzen

     Setze diese Belegung ein.
  2. Wende passende Regeln „rückwärts“ an
     (Ersetze Konklusion durch Prämisse.)
  3. Entferne gefundene Fakten aus Anfragemenge
  - Wiederhole Schritte bis γ leer ist
- * Dieser Algorithmus ist *nicht deterministisch*
  - *Sherlock Holmes wählt immer die „richtigen“ Belegungen!*
- Reihenfolge spielt i.d.R. eine Rolle
  - Im Zweifelsfall systematisch alle Möglichkeiten probieren
  - z.B. durch Breitensuche *backtracking* (*später!*)

- **Fakt**: $\mathtt{succ}(n, n+1)$ für alle $n \in \mathbb{N}_0$
- **Regeln**

$$\stackrel{(1)}{\Rightarrow} \quad \mathtt{add}(X, 0, X)$$

$$\mathtt{add}(X, Y, Z) \wedge \mathtt{succ}(Y, V) \wedge \mathtt{succ}(Z, W) \stackrel{(2)}{\Rightarrow} \mathtt{add}(X, V, W)$$

- Beispiel: $\gamma = \mathtt{add}(3, 2, 5) \rightarrow \mathtt{true}$

$$\begin{aligned}
\gamma \quad & \overset{(2)}{\underset{X=3, V=2, W=5}{\longmapsto}} \quad \mathtt{add}(3, Y, Z) \wedge \mathtt{succ}(Y, 2) \wedge \mathtt{succ}(Z, 5) \\
& \overset{\text{Fakt}}{\underset{Y=1, Z=4}{\longmapsto}} \quad \mathtt{add}(3, 1, 4) \\
& \overset{(2)}{\underset{X=3, V=1, W=4}{\longmapsto}} \quad \mathtt{add}(3, Y, Z) \wedge \mathtt{succ}(Y, 1) \wedge \mathtt{succ}(Z, 4) \\
& \overset{\text{Fakt}}{\underset{Y=0, Z=3}{\longmapsto}} \quad \mathtt{add}(3, 0, 3) \quad \overset{(1)}{\underset{X=3}{\longmapsto}} \quad \mathtt{true}
\end{aligned}$$

Weitere Beispiele

- Siehe [Saake&Sattler] (Kapitel 3.4)
- $\texttt{add}(3, X, 5) \to \{X = 2\}$
- $\texttt{add}(X, Y, 5) \to (X, Y) \in \{(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)\}$
- $\texttt{add}(X, Y, Z) \to \perp$ (unendliches Ergebnis)
- $\texttt{add}(X, Y, 1) \to (X, Y) \in \{(0, 1), (1, 0)\}$

Beispiel: $\mathtt{add}(X, Y, 1)$

$$
\begin{tikzcd}
& \mathtt{add}(X,Y,1) \arrow[dl, "{X=1, Y=0}"', "{(1)}"] \arrow[dr, "{(2)}"] & \\
\texttt{true} & & \mathtt{add}(X,Y',Z') \wedge \mathtt{succ}(Y',Y) \wedge \mathtt{succ}(Z',1) \arrow[d, "Z'=0"] \\
& & \mathtt{add}(X,Y',0) \wedge \mathtt{succ}(Y',Y) \arrow[d, "{X=0, Y'=0}", "{(1)}"'] \\
& & \mathtt{add}(0,0,0) \wedge \mathtt{succ}(0,Y) \arrow[d, "Y=1"] \\
& & \mathtt{succ}(0,1) \;\mapsto\; \texttt{true}
\end{tikzcd}
$$

Kurze Zusammenfassung

- Logische *Aussagen* und *Aussageformen*
  - Als atomare Formeln
  - *Belegung*: Aussageform $\rightarrow$ Aussage
- Menge von *Fakten* und *Regeln*
  - Fakten: Aussagen (keine Unbestimmte)
  - Regeln: Ableiten von neuen Fakten durch Deduktion
  - Schlussregel: *modus ponens*
- *Anfrage* $\rightarrow$ *Antwort*
  - Dazu ist ein Deduktionsalgorithmus nötig!
  - Integraler Teil der Programmiersprache.

- Wenig Ähnlichkeit zu funktionaler oder imperativer P.
  - Spezielle Einsatzgebiete, z.B. Expertensysteme, Theorembeweiser, *model checking*, ...

Übersicht

Abschließend zu Programmierparadigmen

- Es gibt eine Vielzahl weiterer Programmierparadigmen
- Viele Programmiersprachen unterstützen *mehrere* Paradigmen
- Dabei steht i.d.R. ein Paradigma im Vordergrund
- Alle Paradigmen haben Stärken und Schwächen
  - z.B. Funktionale vs Imperative Programmierung
  - Oft Integration als Kompromiss
  - Selten Paradigma in Reinform (z.B. *rein* funktional)
- Beispiel Java
  - Objektorientiert
  - Strukturiert
  - Imperativ
  - wenige Elemente Funktionaler Programmierung (*closures*)
  - wenige Elemente *Generischer Programmierung*

Ausblick: Generische Programmierung

- Beschreibe Funktionen oder Objekte so, dass sie für verschiedene Datentypen verwendet werden können
- Beispiele
  - Absolutbetrag: gleiche Implementierung für `int`, `double`, ...
  - Such- oder Sortieralgorithmus für beliebige Objekte/Typen benötigt Prädikat für Vergleich $a < b$
  - Komplexe Zahlen als Klasse: gleiche Implementierung für `float` oder `double`
- „Polymorphie von Datentypen und Klassen“
  - z.B. Platzhalter T statt konkreter Datentyp `int`
  - Begriff *templates* in C++: Schablonen für Klassen,...
- Java *Generics*
  - Nachgerüstet in Version 1.5
  - Weniger mächtig als z.B. *templates* in C++
  - Mit Schwächen, aber sehr praktisch
  - Beispiel: `Comparable<T>`

Zusammenfassung

- Programmierparadigma =
  - *fundamentaler Programmierstil* und bedingt damit auch
  - „Denkmuster“ für Entwurf und Formulierung von Algorithmen
- Funktionale Programmierung
  - Auswertung von Funktionen
  - Keine Zustandsänderung (keine Variablen!)
  - Rekursion
- Imperative Programmierung
  - Anweisungen bedingen Zustandsänderungen
  - Zustand = Menge von Variablen
  - Iteration
  - *Spezialfall*: Objektorientierte Programmierung
- Logische Programmierung
  - Fakten und Regeln
  - Antwort auf Anfrage durch Deduktion
- In Java fehlten uns noch *generics*