Überblick

Funktionen

- Das Konzept der Ausdrücke erlaubt uns nun, einfache funktionale Algorithmen zu entwerfen
- Zur Veranschaulichung dient folgendes Beispiel aus der elementaren Physik:

  Ein frei beweglicher Körper mit der Masse m (> 0) werde aus der Ruhelage eine Zeit t lang mit einer auf ihn einwirkenden konstanten Kraft k bewegt. Gesucht ist ein Algorithmus, der (in Abhängigkeit von m, t und k) die Strecke s bestimmt, um die der Körper aus seiner ursprünglichen Lage fortbewegt wird.

Funktionen

- Die Datendarstellung ist in diesem Beispiel sehr einfach: Die Größen m, t und k sind offensichtlich aus $\mathbb{R}$ (mit $m > 0$ und $t \geq 0$)
- Der gesuchte Algorithmus kann damit als Abbildung (Funktion)

$$\mathit{Strecke} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

formuliert werden, wobei die Berechnungsvorschrift explizit (als Ausdruck) unter Verwendung von Grundoperationen angegeben werden soll

Funktionen

- Die Idee für diesen Algorithmus ergibt sich aus folgenden einfachen Gesetzen:
  - Die auf einen Körper mit Masse m einwirkende Kraft k erzeugt eine konstante Beschleunigung
$$b = \frac{k}{m}$$
  - Der in der Zeit t zurückgelegte Weg bei einer Bewegung mit konstanter Beschleunigung b ist
$$s = \frac{1}{2} \cdot b \cdot t^2$$
  - Einsetzen liefert
$$strecke(m, t, k) = (k \cdot t^2)/(2 \cdot m)$$
  - Dies beschreibt, wie man s durch Anwendung verschiedener Grundoperationen auf m, t und k bestimmen kann, ist also offensichtlich bereits der gesuchte (funktionale Algorithmus)

Erweiterung des Modulkonzepts

- Wir können nun das Modulkonzept wie angekündigt erweitern
- Zur Erinnerung: ein Modul ist eine Menge von Sorten mit einer Menge von Operationen über diesen Sorten
- Wir können nun die Menge der Sorten $\{\mathbb{R}\}$ und die Menge der Operationen bestehend aus $strecke$ als Modul (das wir vielleicht *Bewegung* nennen könnten) auffassen
- Anders als bisher, sind die Operationen (in diesem Fall die Funktion $Strecke$) nun nicht mehr abstrakt, sondern durch einen (funktionalen) Algorithmus explizit angegeben

Erweiterung des Modulkonzepts

```
1  module Bewegung
2      operations
3          function strecke(m : ℝ, t : ℝ, k : ℝ) → ℝ
4              output  Bewegte Strecke eines Körpers mit Masse m bei Krafteinwirkung k nach Zeit t
5              pre     m > 0, t ≥ 0
6              body    (k · t²)/(2 · m)
```

- Unter **operations** (Zeile 2) stehen alle Funktionsvereinbarungen des Moduls (hier nur die von *strecke*, Zeile 3))
- Für die einzelnen Operationen sind angegeben (am Bsp. *strecke*):
  - Die Signatur der Operation (Zeile 3)
  - Optionale Beschreibung des Ergebnis (Zeile 5)
  - Evtl. *Vorbedingungen*, zur Einschränkung des Def.bereichs (Zeile 5)
  - Der eigentliche *Funktionsrumpf* (Berechnungsvorschrift, Algorithmus) hinter dem Schlüsselwort **body** (Zeile 6)

Erweiterung des Modulkonzepts

- m, t und k sind Variablen der Sorte $\mathbb{R}$, die sog. (*formalen*) *(Eingabe-) Parameter* von $strecke$
- Im Funktionsrumpf darf ein beliebiger Ausdruck stehen; als Variablen für diesen Ausdruck sind nur die formalen Parameter des Algorithmus erlaubt
- Der Vollständigkeit halber: Im Rumpf der Funktion $strecke$ haben wir eine implizite Sortenanpassung verwendet!

Benutzerdefinierte Funktionen in Ausdrücken

- Allgemein können Module mehrere Funktionen zusammenfassen, d.h. auch unser Modul *Bewegung* könnte weitere Funktionen (z.B. die Endgeschwindigkeit ($k/m \cdot t$) oder die verrichtete Arbeit ($k \cdot strecke(m, t, k)$)) bereitstellen.
- Das können auch 0-stellige Funktionen sein, z.B.

function $PI \rightarrow \mathbb{R}$
output Annährung der Zahl π
body 3.14159

die als „Synonym“ für Literale der Sorte des Bildbereichs (hier $3.14159 \in \mathbb{R}$) aufgefasst werden können.

Benutzerdefinierte Funktionen in Ausdrücken

- Die Funktion *strecke* kann dabei auch innerhalb der anderer Algorithmen *verwendet* (*aufgerufen*) werden genauso wie die bisherigen Basisoperationen.
- Dazu kann die Syntax von Ausdrücken entsprechend erweitert werden:

  Sind $a_1, \ldots, a_n$ Ausdrücke der Sorten $S_1, \ldots, S_n$ und f eine benutzerdefinierte Funktion (Algorithmus) eines konkreten Moduls mit der Signatur $f : S_1 \times \ldots \times S_n \rightarrow S_{n+1}$ (wobei $S_1, \ldots, S_n \in S$), dann ist $f(a_1, \ldots, a_n)$ ein Ausdruck der Sorte S_{n+1}.

Erweiterung des Modulkonzepts

Hier nun noch das „vollständige“ Modul *Bewegung*:

module *Bewegung*

operations

function $strecke(m : \mathbb{R}, t : \mathbb{R}, k : \mathbb{R}) \to \mathbb{R}$

output Strecke

pre $m > 0, t \geq 0$

body $(k \cdot t^2)/(2 \cdot m)$

function $endgeschwindigkeit(m : \mathbb{R}, t : \mathbb{R}, k : \mathbb{R}) \to \mathbb{R}$

output Endgeschwindigkeit

pre $m > 0, t \geq 0$

body $(k/m) \cdot t$

function $arbeit(m : \mathbb{R}, k : \mathbb{R}, t : \mathbb{R}) \to \mathbb{R}$

output Arbeit

pre $m > 0, t \geq 0$

body $k \cdot strecke(m, t, k)$

Funktionsausführung/-aufruf

- Nun, da wir verschiedene Algorithmen „programmiert“ haben, wollen wir sie natürlich auch benutzen können, z.B. wollen wir die Strecke, die ein Körper mit Masse 1.0 zurücklegt, wenn er mit einer Kraft von 3.0 über 2.0 Zeiteinheiten bewegt wird[9], mit Hilfe der Funktion $strecke$ berechnen.
- Die Ausführung eines Algorithmus ist ein *Funktionsaufruf* mit konkreten Eingabewerten.
- Was passiert bei so einem Funktionsaufruf?

[9] Über die physikalischen Maß-Einheiten haben wir uns bisher nicht gekümmert und tun dies hier auch nicht

Funktionsausführung/-aufruf

- Der Aufruf des Algorithmus $strecke$ mit konkreten Werten ist nichts anderes als eine Substitution
  - Der Algorithmus wird durch einen Ausdruck definiert.
  - Der *Wert* dieses Audrucks ergibt sich aus der Substitution der Eingabevariablen durch die entsprechenden Werte, mit denen der Algorithmus aufgerufen wird
  - Diese Substitution σ wird auch *Variablenbelegung* genannt; die Eingabevariablen werden jeweils mit einem Literal der entsprechenden Sorte belegt
  - $V(\sigma)$ bezeichnet die Menge der (Eingabe-)Variablen in dieser Substitution
  - z.B. beim Aufruf von $strecke(1.0, 2.0, 3.0)$ werden die Variablen m,t,k mit den Literalen 1.0,2.0,$3.0 \in \mathbb{R}$ belegt, d.h. $V(\sigma) = \{m, t, k\}$ und

$$\sigma = [m/1.0, t/2.0, k/3.0]$$

Wert eines Ausdrucks

Formal

- Gegeben eine Substitution/Variablenbelegung σ ($V(\sigma)$ bezeichnet die Variablen in σ)
- Rekursive Definition des *Wertes* $W_\sigma(u)$ *eines Ausdrucks* u bzgl. σ:
  - Wenn u eine Variable $v \in V(\sigma)$ ist, so ist $W_\sigma(u) = u\,\sigma$
  - Wenn u ein Literal op ist, so ist $W_\sigma(u) = op$
  - Wenn u eine Anwendung eines n-stelligen Operators $op(a_1, \ldots, a_n)$ ist, so ist $W_\sigma(u) = op(W_\sigma(a_1), \ldots, W_\sigma(a_n))$
  - Wenn u ein Funktionsaufruf $f(a_1, \ldots, a_n)$ mit Funktionsrumpf r ist, so ist $W_\sigma(u) = W_\sigma(r)$

Wert eines Ausdrucks

- Bemerkungen
  - $W_\sigma(u)$ ergibt als Wert ein Objekt der Sorte von u, bezeichnet durch das entsprechende Literal der Sorte
  - Wir schreiben $W(u)$ statt $W_\sigma(u)$, wenn σ aus dem Kontext klar ersichtlich ist
  - Wenn u eine Variable $v \notin V(\sigma)$ ist, so ist $W_\sigma(u)$ *undefiniert*
  - Um nur definierte Werte für Ausdrücke zu erhalten, fordern wir:
    - Der Ausdruck im Rumpf einer Funktion f darf nur die formalen (Eingabe-) Parameter der Funktion enthalten (z.B. im Rumpf der Funktion $strecke$ ist die Menge der Variablen $V = \{m, k, t\}$)
    - Beim Aufruf einer Funktion müssen alle formalen Parameter mit konkreten Werten belegt sein, d.h. für alle Variablen in V ist eine entsprechende Substitution mit einem konkreten Wert in σ definiert

Wert eines Ausdrucks

Beispiel:

Aufruf von $strecke(1.0, 2.0, 3.0)$ entspricht einer Substitution
$\sigma = [m/1.0, t/2.0, k/3.0]$ mit $V(\sigma) = \{m, t, k\}$

$$
\begin{aligned}
W(strecke(1.0, 2.0, 3.0)) &= W((k \cdot t \cdot t)/(2 \cdot m)) \\
&= W(k \cdot t \cdot t)/W(2 \cdot m) \\
&= (W(k) \cdot W(t) \cdot W(t))/(W(2) \cdot W(m)) \\
&= (k[k/3.0] \cdot t[t/2.0] \cdot t[t/2.0])/(2 \cdot m[m/1.0]) \\
&= (3.0 \cdot 2.0 \cdot 2.0)/(2 \cdot 1.0) \\
&= 12.0/2.0 \\
&= 6.0
\end{aligned}
$$

Beachten Sie die implizite Sortenanpassung.

Wert eines Ausdrucks

Weiteres Beispiel:

Aufruf von $endgeschwindigkeit(1.0, 2.0, 3.0)$ entspricht ebenfalls einer Substitution $\sigma = [m/1.0, t/2.0, k/3.0]$ mit $V(\sigma) = \{m, t, k\}$

$$
\begin{aligned}
W(endgeschwindikeit(1.0, 2.0, 3.0)) &= W((k/m) \cdot t) \\
&= W(k/m) \cdot W(t) \\
&= (W(k)/W(m)) \cdot W(t) \\
&= (k[k/3.0]/m[m/1.0]) \cdot t[t/2.0] \\
&= (3.0/1.0) \cdot 2.0 \\
&= 6.0
\end{aligned}
$$

Wert eines Ausdrucks

Und noch ein Beispiel:

Aufruf von $arbeit(1.0, 2.0, 3.0)$ entspricht einer Substitution
$\sigma = [m/1.0, t/2.0, k/3.0]$ mit $V(\sigma) = \{m, t, k\}$

$$
\begin{aligned}
W(arbeit(1.0, 2.0, 3.0)) &= W(k \cdot strecke(m, t, k)) \\
&= W(k) \cdot W(strecke(m, t, k)) \\
&= k[k/3.0] \cdot W((k \cdot t \cdot t)/(2 \cdot m)) \\
&= 3.0 \cdot \ldots \\
&= 3.0 \cdot 6.0 \\
&= 18.0
\end{aligned}
$$

Bedingte Ausdrücke

- Um Fallunterscheidung zu modellieren, benötigen wir nun noch das Konzept der *bedingten Ausdrücke* (*Terme*)
- Dazu erweitern wir die induktive Definition der Ausdrücke um folgenden Fall
  - Voraussetzung: die Menge der Sorten S enthält die Sorte $\mathbb{B}$
  - Ist b ein Ausdruck der Sorte $\mathbb{B}$ und sind a_1 und a_2 Ausdrücke der selben Sorte $S_0 \in S$, dann ist

    if b **then** a_1 **else** a_2 **endif**

    ein Ausdruck der Sorte S_0.
  - b heißt *Wächter*, a_1 und a_2 heißen *Zweige*.
- Bemerkung: a_1 und/oder a_2 können offenbar wiederum bedingte Ausdrücke (der Sorte S_0) sein, d.h. man kann bedingte Ausdrücke (beliebig) ineinander schachteln

Bedingte Ausdrücke

Beispiele

- Für $x \in \mathbb{Z}$ ist der Absolutbetrag von x bestimmt durch folgenden Algorithmus (in unserer Moulschreibweise)

function $abs(x : \mathbb{Z}) \to \mathbb{R}$
  output Absolutbetrag
  body **if** $x \geq 0$ **then** x **else** $-x$ **endif**

Bedingte Ausdrücke

Beispiele (cont.)

- Im nächsten Algorithmus ist die Schachtelung von bedingten Ausdrücken zu sehen.
- Wir wollen dabei die der Größe nach mittlere von drei natürlichen Zahlen $x, y, z \in \mathbb{N}_0$ bestimmen:

function $mitte(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z}) \rightarrow \mathbb{R}$
  output die der Größe nach mittlere Zahl
  body
    if $(x < y) \wedge (y < z)$ **then** y **else**
      if $(x < z) \wedge (z < y)$ **then** z **else**
        if $(y < x) \wedge (x < z)$ **then** x **else**
          if $(y < z) \wedge (z < x)$ **then** z **else**
            if $(z < x) \wedge (x < y)$ **then** x **else** y **endif**
    endif endif endif endif

Bedingte Ausdrücke

- Der Wert $W(u)$ eines bedingten Ausdrucks u

  if b **then** a_1 **else** a_2 **endif**

  ist abhängig von $W(b)$:
  - Ist $W(b) = TRUE$, so ist $W(u) = W(a_1)$
  - Ist $W(b) = FALSE$, so ist $W(u) = W(a_2)$

Bedingte Ausdrücke

- Beispiel: Aufruf von $abs(-3)$ entspricht einer Substitution $\sigma = [x/-3]$

$$
\begin{aligned}
W(abs(-3)) &= W(\textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x \textbf{ endif}) \\
\text{Dazu: } W(x \geq 0) &= (x \geq 0)[x/-3] \\
&= x[x/-3] \geq 0[x/-3] \\
&= -3 \geq 0 = \mathit{FALSE}
\end{aligned}
$$

Also:

$$
\begin{aligned}
W(\textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x \textbf{ endif}) &= W(-x) = -x[x/-3] \\
&= --3 = 3
\end{aligned}
$$

Rekursion

- Mit bedingten Ausdrücken lassen sich nun auch rekursive Funktionen (Algorithmen) formulieren.
- Beispiel Fakultätsfunktion:

function $fak(x : \mathbb{N}_0) \to \mathbb{N}_0$
output Fakultät
body **if** $x = 0$ **then** 1 **else** $x \cdot fak(x - 1)$ **endif**

- Der Wert der Funktion für einen konkreten Aufruf ist mit den bisherigen Konzepten berechenbar.
- Damit ist obige Funktion auch wirklich ein Algorithmus zur Berechnung der Fakultätsfunktion

Rekursion

- Das Ergebnis der Anwendung (des Aufrufs) von fak für die Eingabe 3, $W(fak(3))$, errechnet sich z.B.:

1) Aufruf $fak(3)$, d.h. $\sigma = [x/3]$

$$
\begin{aligned}
& W_{[x/3]}(fak(x)) \\
&\quad = W_{[x/3]}(\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \cdot fak(x-1) \textbf{ endif}) \\
&\quad = W_{[x/3]}(x \cdot fak(x-1)) \qquad (\text{da } W_{[x/3]}(x = 0) = FALSE) \\
&\quad = W_{[x/3]}(x) \cdot W_{[x/3]}(fak(x-1)) \\
&\quad = 3 \cdot W(fak((x-1)[x/3])) \\
&\quad = 3 \cdot W(fak(2))
\end{aligned}
$$

Rekursion

- Fortsetzung:

2) Aufruf $fak(2)$, d.h. $\sigma = [x/2]$

$$
\begin{aligned}
& W_{[x/2]}(fak(x)) \\
&= W_{[x/2]}(\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \cdot fak(x-1) \textbf{ endif}) \\
&= W_{[x/2]}(x \cdot fak(x-1)) \qquad (\text{da } W_{[x/2]}(x = 0) = FALSE) \\
&= W_{[x/2]}(x) \cdot W_{[x/2]}(fak(x-1)) \\
&= W_{[x/2]}(x) \cdot W(fak((x-1)[x/2])) \\
&= 2 \cdot W(fak(1))
\end{aligned}
$$

Rekursion

- Fortsetzung:

3) Aufruf $fak(1)$, d.h. $\sigma = [x/1]$

$$
\begin{aligned}
& W_{[x/1]}(fak(x)) \\
&\quad = W_{[x/1]}(\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \cdot fak(x-1) \textbf{ endif}) \\
&\quad = W_{[x/1]}(x \cdot fak(x-1)) \qquad (\text{da } W_{[x/1]}(x = 0) = FALSE) \\
&\quad = W_{[x/1]}(x) \cdot W_{[x/1]}(fak(x-1)) \\
&\quad = 1 \cdot W(fak(0))
\end{aligned}
$$

4) Aufruf $fak(0)$, d.h. $\sigma = [x/0]$

$$
\begin{aligned}
& W_{[x/0]}(fak(x)) \\
&\quad = W_{[x/0]}(\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \cdot fak(x-1) \textbf{ endif}) \\
&\quad = W_{[x/0]}(1) \qquad (\text{da } W_{[x/0]}(x = 0) = TRUE) \\
&\quad = 1
\end{aligned}
$$

Rekursion

- Fortsetzung:

  5) Einsetzen von 4) in 3) ergibt: $W(fak(1)) = 1 \cdot 1 = 1$

  6) Einsetzen von 5) in 4) ergibt: $W(fak(2)) = 2 \cdot 1 = 2$

  7) Einsetzen von 6) in 5) ergibt: $W(fak(3)) = 3 \cdot 2 = 6$

- Die verschiedenen Aufrufe einer rekursiven Funktion während der Auswertung eines gegebenen Aufrufs nennt man auch *Inkarnationen* der Funktion

Rekursion und Terminierung

Mit unserer Formalisierung können wir übrigens Terminierung sehr sauber fassen:

- Ein (funktionaler) Algorithmus f mit Rumpf r *terminiert* für eine gegebene Parameterbelegung σ, wenn die Bestimmung des Wertes $W(r)$ bzgl. σ in endlich vielen Schritten einen definierten Wert ergibt
- Speziell bei rekursiven Algorithmen ist dies nicht immer offensichtlich und muss notfalls bewiesen werden (typischerweise durch Induktion)

Rekursion und Terminierung

Beispiel

- Behauptung: $W(fak(n))$ *ergibt sich für ein beliebiges* $n \in \mathbb{N}$ *in endlich vielen Auswertungsschritten*
- Induktionsanfang $n = 0$: $W(fak(0))$: $W_{[x/0]}(fak(x))$ prüfen
  Wie wir im Beispiel vorher gesehen haben, geht diese Auswertung in endlich vielen Schritten.
- Induktionsschritt $n \to n + 1$:
  Induktionsvoraussetzung: $W(fak(n))$ ist in endlich vielen Schritten auswertbar.
  $W(fak(n+1)) = W_{[x/n+1]}(fak(x)) = W_{[x/n+1]}(x) \cdot W_{[x/n+1]}(fak(x-1))$
  $= (n+1) \cdot W(fak(x[x/n+1]-1)) = (n+1) \cdot W(fak(n))$
  Nach IV ist $W(fak(n))$ in endlich vielen Schritten auswertbar.

Rekursion

Weitere Beispiele für rekursive (funktionale) Algorithmen

- Summenformel

function $summeBis(n : \mathbb{N}_0) \to \mathbb{N}_0$
output Summe von 0 bis n
body **if** $n = 0$ **then** 0 **else** $n + summeBis(n-1)$ **endif**

- Fibonacci-Zahlen

function $fib(x : \mathbb{N}_0) \to \mathbb{N}_0$
output x-te Fibonacci-Zahl
body **if** $x = 0 \vee x = 1$ **then** 1 **else** $fib(x-1) + fib(x-2)$ **endif**

Von der Theorie zur Praxis

Zunächst eine kurze Zusammenfassung:

- Wir hatten begonnen mit der Formalisierung der als gegeben angenommenen Grunddatentypen und deren Grundoperationen (zunächst theoretisch mit Hilfe des Modul-Konzepts, anschließend haben wir uns die primitiven Typen und deren Operationen in Java angesehen)
- Das Konzept der Ausdrücke (Terme) folgte:
  - wir haben die Struktur (Syntax) definiert
  - wir haben definiert, wie Ausdrücke interpretiert werden können (Semantik)
  - dadurch bekamen wir die Möglichkeit, den funktionalen Zusammenhang zwischen Ein- und Ausgabedaten zu spezifizieren, also funktionale Algorithmen zu entwerfen
  - wir haben uns zudem angesehen, wie einfache Ausdrücke in Java aussehen

Von der Theorie zur Praxis

- Diese funktionalen Algorithmen wurden als Funktionen notiert
- Wir erweiterten das Modulkonzept indem wir nun auch diese eigenen Funktionen in Modulen zugelassen haben
  - dadurch stehen die selbst-definierten Funktionen auch wieder anderen Algorithmen zur Benutzung zur Verfügung, d.h. sie können in anderen Algorithmen (Funktionen) verwendet/aufgerufen werden (wie die Grundoperationen)
  - Module sind damit Einheiten, die spezielle Funktionalitäten bereitstellen, dienen also zur Strukturierung von komplexeren Programmen und ermöglichen die Wiederverwendung von Algorithmen (Code)

Das Konzept der Ausdrücke wurde erweitert um bedingte Ausdrücke, was wiederum Rekursion ermöglichte.

Von der Theorie zur Praxis

- Ausgehend von den theoretischen Konzepten, wollen wir uns noch kurz anschauen, wie diese in einer konkreten Sprache umgesetzt sind
- Java ist hier eigentlich ein schlechtes Beispiel, da Java hauptsächlich dem imperativen Paradigma folgt
- Dennoch kann man die einzelnen Konzepte in Java veranschaulichen
- Basierend auf den einfachen Java-Ausdrücken, die wir schon kennen, werden wir das nun tun

Funktionen in Java

- In Java heißen Funktionen (übrigens wie das imperative Pendant der Prozeduren) *Methoden*
- Die Funktion *strecke*, die im Modul *Bewegung* definiert ist, lässt sich in Java wie folgt notieren:

```
/**
   * Berechnung der Strecke, die ein K&ouml;rper mit einer gegeben Masse,
   * der ein gegebene Zeit lang mit einer auf ihn einwirkenden
   * konstanten Kraft bewegt wird, zur&uuml;cklegt.
   * @param   m die Masse
   * @param   t die Zeit
   * @param   k die Kraft
   * @return  die Strecke, die der K&ouml;rper zur&uuml;cklegt.
   */

public static double strecke(double m, double t, double k) {
     return (k * t * t) / (2 * m);
}
```

Funktionen in Java

Erklärungen:

```
public static double strecke(double m, double t, double k) {
     return (k * t * t) / (2 * m);
}
```

- `double strecke(double m, double t, double k)` spezifiziert die *Signatur* der Methode (Funktion)
- Das Schlüsselwort **`public`** ist zunächst nicht wichtig; es bedeutet intuitiv, dass diese Methode (Funktion) von anderen Modulen aus verwendbar ist (d.h. es gibt auch Methoden, die das nicht sind)
- Das Schlüsselwort **`static`** ist zunächst ebenso nicht wichtig; es zeigt an, dass es sich um einen rein imperativen (bzw. funktionalen) Algorithmus handelt

Funktionen in Java

Erklärungen (cont.):

```
public static double strecke(double m, double t, double k) {
      return (k * t * t) / (2 * m);
}
```

- In den Klammern `{` und `}` ist der Rumpf der Methode (Funktion) plaziert
  - Hier sollte laut unserer Theorie einfach ein Ausdruck stehen
  - Tatsächlich steht hier genau genommen ein Befehl, den die JVM ausführt: das Schlüsselwort **`return`** beendet die Methode und gibt den Wert des Ausdrucks, der im Anschluss steht als Rückgabewert zurück
  - Der Befehl **`return`** `( k * t * t) / (2 * m);` simuliert also sozusagen das funktionale Konzept: er weist an, den Ausdruck nach **`return`** auszuwerten und dieser Wert ist der (Rückgabe-)Wert der Methode (dieser Wert ist übrigens entsprechend unserer Formalisierung berechenbar für eine gegebene Variablenbelegung)

Funktionen in Java

- Funktionen in Java sind also als Methoden umgesetzt
- Methoden sind eigentlich imperative Prozeduren (siehe später), die eine Reihe von Anweisungen enthalten, d.h. das Konzept der Funktion in Reinform existiert in Java nicht
- Der wesentliche Unterschied zwischen Prozeduren und Funktionen ist, dass Funktionen keine Nebeneffekte haben, sondern direkt den Zusammenhang zwischen Ein- und Ausgabe berechnen
- Eine Funktion in Java ist daher eine Methode, die nur aus einer (oder bei Fallunterscheidung) mehreren `return`-Anweisung(en) besteht und keine Nebeneffekte hat

Funktionen in Java

- Weiteres Beispiel:

  Die Funktion *arbeit* aus dem Modul *Bewegung*, lässt sich (auf Basis der Methode `strecke`) in Java wie folgt notieren:

```
/**
   * Berechnung der Arbeit, die ein K&ouml;rper mit einer gegeben Masse,
   * der ein gegebene Zeit lang mit einer auf ihn einwirkenden
   * konstanten Kraft bewegt wird, leistet.
   * @param   m die Masse
   * @param   t die Zeit
   * @param   k die Kraft
   * @return  die Arbeit, die der K&ouml;rper leistet.
   */

public static double arbeit(double m, double t, double k) {
     return k * strecke(m,t,k);
}
```

Module in Java

- Auch das Modulkonzept ist in Java nicht explizit umgesetzt
- Es ist vielmehr ein Nebenprodukt des Klassenkonzepts, bzw. das Klassenkonzept kann auch für die Implementierung von Modulen verwendet werden
- Ein Modul `MyModul` in Java ist eine Vereinbarung der *Klasse* `MyModul` in der Textdatei `MyModul.java`, in der nur statische Elemente (also Elemente mit dem Schlüsselwort **`static`**) vorkommen
- Die Menge der Operationen besteht aus den vereinbarten (statischen) Methoden

Module in Java

```
/**
  * Umsetzung des Moduls Bewegung.
  */
public class Bewegung {

    /** ... */
    public static double strecke(double m, double t, double k) {
        return ( k * t * t) / (2 * m);
    }


    /** ... */
    public static double endgeschwindigkeit(double m, double t, double k) {
        return (k/m) * t;
    }

    /** ... */
    public static double arbeit(double m, double t, double k) {
        return k * strecke(m,t,k);
    }
}
```

Module in Java

- Es gibt in Java einige Klassen, die nichts anderes als Module in unserem Sinne sind, d.h. die eine Menge von (ausschließlich statischen) Operationen über einer Menge von Sorten zur Verfügung stellen
- Ein sehr nützliches solches Modul ist z.B. die Klasse `Math`:
  The class `Math` *contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.* [. . .]
- Schauen Sie sich doch einfach mal die Dokumentation der Klasse (des Moduls) an unter:
  `https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html`

Module in Java

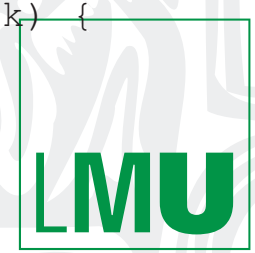
- Ein (statische) Methode `m` der Klasse (des Moduls) `K` wird übrigens mit `K.m` bezeichnet
- Beispiel: `Math.pow` bezeichnet die Methode `pow` der Klasse `Math`, gegeben durch

$$\textbf{double} \times \textbf{double} \to \textbf{double} \qquad \text{mit } (x, y) \mapsto x^y$$

- Folgende Variante der Methode `strecke` nutzt die Methode `Math.pow` beispielhaft

```
/** ... */

    public static double streckeVariante(double m, double t, double k) {
         return ( k * Math.pow(t,2)) / (2 * m);
    }
```



Bedingte Ausdrücke in Java

- Bedingte Ausdrücke gibt es in Java, allerdings in etwas anderer Notation: `<Bedingung> ? <Dann-Wert> : <Sonst-Wert>` (mit rechtsassoziativer Bindung)
- `<Bedingung>` ist ein Ausdruck vom Typ **`boolean`**, die Ausdrücke `<Dann-Wert>` und `<Sonst-Wert>` haben einen beliebigen aber (wie vorher) den selben Typ
- Beispiel

```
/**
  * Berechnung des Absolutbetrags einer ganzen Zahl.
  * @param   x die ganze Zahl
  * @return  der Absolutbetrag von x.
  */

public static int abs(int x) {
     return (x>=0) ? x : -x;
}
```

Bedingte Ausdrücke in Java

- Alternativ kann ein bedingter Ausdruck auch durch eine Fallunterscheidung mit **`return`**-Anweisung(en) simuliert werden (dies ist i.A. die häufiger verwendete Schreibweise):

```
/**
   * ...
   */

public static int absAlternative(int x) {
    if(x>=0) return x; else return -x;
}
```

- Achtung: in Java fehlt das Schlüsselwort **`then`**
- Zur besseren Lesbarkeit kann man sog. Blockklammern setzen:

```
if(x>=0) { return x; } else { return -x; }
```

(was das genau ist, lernen wir bald kennen)

Rekursion in Java

- Mit Hilfe der bedingten Ausdrücke kann man in Java entsprechend rekursive Funktionen implementieren, z.B.:

```
/**
 * Berechnet die x-te Fibonacci-Zahl
 * @param x eine nat&uuml;rliche Zahl
 * @return die x-te Fibonacci-Zahl
 */
public static int fib(int x) {
     return (x==0 | x==1) ? 1 : (fib(x-1) + fib(x-2));
}
```

bzw. entsprechend

```
public static int fibVariante(int x) {
         if(x==0 | x==1) { return 1; }
         else { return fib(x-1) + fib(x-2); }
}
```

Wrap-up

- Sie können jetzt rein funktionale Programme in Java schreiben
- Das ist noch nicht viel, aber es ist auch nicht wenig
- Wichtig dabei: unsere theoretischen Konzepte, insbesondere die Semantik der Ausdrücke und deren Werte, gelten 1-zu-1 für Java (und übrigens auch in anderen Sprachen)!!!
- Wenn Sie verstanden haben, was bei so einer Auswertung abläuft, haben Sie eines der wesentlichen Geheimnisse der funktionalen Programmierung verstanden
- Andere Programmiersprachen (z.B. SML oder C) schreiben diese Konzepte (wenn überhaupt) einfach nur anders auf, das Kernprinzip aber bleibt gleich

Beispiele

- Berechne für zwei Eingaben a und b vom Typ $\mathbb{B}$, wie oft davon $TRUE$ vorhanden ist
- Algorithmus:

function $wieOftTrue(a : \mathbb{B}, b : \mathbb{B}) \rightarrow \mathbb{N}_0$
  output Wieviel von a und b sind $TRUE$
  body
    if $a \wedge b$ **then** 2 **else**
      if $a \vee b$ **then** 1 **else** 0 **endif**
    endif

- Java Programm:

```
public static int wieOftTrue(boolean a, boolean b) {
      if(a && b) return 2;
      else if(a || b) return 1;
      else return 0;
}
```

Alternativ mit verschachtelten bedingten Ausdrücken:

```
public static int wieOftTrueVariante(boolean a, boolean b) {
      return (a && b) ? 2 : ((a || b) ? 1 : 0);
```

Beispiele

- Gegeben: Abfahrtszeit (Stunden und Minuten) und Ankunftszeit (Stunden und Minuten) eines Zuges.
- Gesucht: Berechne die Fahrtzeit des Zuges.
- Zur Vereinfachung: es darf angenommen werden, dass der Zug nicht länger als 24 Stunden fährt.
- Datenmodellierung:
  - Abfahrtszeit ist durch ein Paar von ganzen Zahlen (ab_s, ab_m) gegeben, wobei ab_s die Stunden und ab_m die Minuten repräsentiert,
    z.B. 17:23 ist repräsentiert durch das Paar $(17, 23)$.
  - Ankunftszeit analog.
  - Ausgabe ist die Fahrtzeit in Minuten.

Beispiele

Lösungsidee:

- Fall 1: die Fahrt geht nicht über Mitternacht.
  - Beispiel: 12:10 bis 17:50 ergibt $17 - 12 = 5$ Stunden, $50 - 10 = 40$ Minuten bzw. $5 \cdot 60 + 40 = 340$ Minuten.
  - Beispiel: 12:30 bis 18:10 ergibt ebenfalls 340 Minuten.
  - Idee: Wandle die Zeiten in Minuten um und ziehe die Ankunftszeit von der Abfahrtszeit ab.
  - Beispiel: 12:10 bis 17:50:
    Verwandle 17:50 in $17 \cdot 60 + 50 = 1070$ um und 12:10 in $12 \cdot 60 + 10 = 730$ um.
    Differenz ergibt $1070 - 730 = 340$.
  - Analog mit 12:30 bis 18:10
    $(18 \cdot 60 + 10) - (12 \cdot 60 + 30) = 340$
  - Lösung für Fall 1:

$$(an_s \cdot 60 + an_m) - (ab_s \cdot 60 + ab_m)$$

Beispiele

Lösungsidee (cont.):

- Fall 2: die Fahrt geht über Mitternacht
  - Beispiel: 22:10 bis 02:50 ergibt 280 Stunden, aber leider nicht $\underbrace{(2 \cdot 60 + 50)}_{170} - \underbrace{(22 \cdot 60 + 10)}_{1330} = -1160$ Minuten
  - Beobachtung: wir müssten zur Ankunftszeit 24 Stunden (1440 Minuten) hinzurechnen, dann ginge es mit der Formel aus Fall 1
  - Also:
    $(2 \cdot 60 + 50) + 1440 = 1610$ und $1610 - 1330 = 280$
  - Analog mit 22:30 bis 03:10
    $(3 \cdot 60 + 10 + 1440) - (22 \cdot 60 + 30) = 1630 - 1350$
  - Lösung für Fall 1:
    $$(an_s \cdot 60 + an_m + 1440) - (ab_s \cdot 60 + ab_m)$$

Beispiele

- Fehlt nur noch: wie prüfen wir, ob die Fahrt über Mitternacht geht?
- Offenbar gilt bei Fahrt über Mitternacht: $an_s < ab_s$ oder, wenn $an_s = ab_s$, dann $an_m < ab_m$
- Algorithmus:

function $fahrzeit(ab_s : \mathbb{N}_0, ab_m : \mathbb{N}_0, an_s : \mathbb{N}_0, an_m : \mathbb{N}_0) \to \mathbb{N}_0$
  output Fahrzeitberechnung f
  body
    if $an_s < ab_s \vee (an_s = ab_s \wedge an_m < ab_m)$
    then $(an_s \cdot 60 + an_m + 1440) - (ab_s \cdot 60 + ab_m)$
    else $(an_s \cdot 60 + an_m) - (ab_s \cdot 60 + ab_m)$
    endif

Beispiele

- Java Programm:

```
public static int fahrtzeitInMinuten(int abS, int abM, int anS, int anM) {
     if((anS < abS) || (anS == abS && anM < abM)) {
          // ueber Mitternacht
          return (anS * 60 + anM + 1440) - (abS * 60 + abM);
     } else {
          // nicht ueber Mitternacht
          return (anS * 60 + anM) - (abS * 60 + abM);
     }
}
```

Beispiele

- Berechne für ein $n \in \mathbb{N}_0$ die Anzahl der Stellen (Ziffern) von n.
- Beispiel: Anzahl der Stellen der Zahl 1234 ist 4
- Beobachtung:
  - Frage: Welche Funktion bildet die einstelligen Zahlen $0, \ldots, 9$ auf den selben Wert ab?
  - Antwort: Die ganzzahlige Division durch 10, nämlich auf 0.
    Die zweistellige Zahl 42 wird dagegen auf 4 abgebildet.
    Die dreistellige Zahl 792 wird auf 79 abgebildet.
  - Allgemeines Schema:
    Die ganzzahlige Division durch 10 ergibt bei 1-stelligen Zahlen 0 und bei mehrstelligen Zahlen wird die letzte Stelle abgeschnitten.

Beispiele

- Algorithmus:

function $stellen(x : \mathbb{N}_0) \rightarrow \mathbb{N}$
  output Anzahl Stellen einer Zahl f
  body
    if $DIV(n, 10) = 0$
    then 1
    else $1 + stellen(DIV(n, 10))$
    endif

- Java Programm:

```
public static int stellen(int n) {
      if(n / 10 == 0) {
            return 1;
      } else {
            return 1 + stellen(n / 10);
      }
}
```