Rekursion

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

Algorithmus 12 (Fakultät)

$$FAK : \mathbb{N}_0 \to \mathbb{N}_0$$

mit

$$FAK(x) = \textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } x \cdot FAK(x-1) \textbf{ endif}$$

berechnet die Fakultät einer natürlichen Zahl

- 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. $\sigma = [x/3]$

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

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

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

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)
- Beispiel (Skizze):
  Behauptung: $W(FAK(n))$ *ergibt sich für ein beliebiges* $n \in \mathbb{N}$ *in endlich vielen Auswertungsschritten*
  Induktionsanfang $W(FAK(0))$: $W_{[x/0]}(FAK(x))$ prüfen
  Induktionsschritt $W(FAK(n))$: $W_{[x/n]}(FAK(x))$ auf $W_{[x/n]}(x) \cdot W_{[x/n]}(FAK(x-1))$ zurückführen (für $W(FAK(n-1))$ gilt IV)

Rekursion

Weitere Beispiele für rekursive (funktionale) Algorithmen

- Summenformel

  Algorithmus 13 (Summe)

  $$Summe : \mathbb{N}_0 \to \mathbb{N}_0$$

  mit

  $$Summe(x) = \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } x + Summe(x-1) \textbf{ endif}$$

- Fibonacci-Zahlen

  Algorithmus 14 (Fibonacci-Zahl)

  $$FIB : \mathbb{N}_0 \to \mathbb{N}_0$$

  mit

  $$FIB(x) = \textbf{if } x = 0 \vee x = 1 \textbf{ then } 1 \textbf{ else } FIB(x-1) + FIB(x-2) \textbf{ endif}$$

Von der Theorie zur Praxis

Zunächst eine kurze Zusammenfassung:

- Wir starteten das Kapitel 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 eigene 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 theoretsichen 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 in Algorithmus 8 gegeben 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);
}
```

(Sie erinnern sich sicher an die Konvention, dass Methoden mit kleinen Buchstaben beginnen!!!)

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
- 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 (nämlich `( k * t * t) / (2 * m)`) stehen
  - Tatsächlich steht hier eine Anweisung (imperativ!!!): das Schlüsselwort **`return`** beendet die Methode und gibt den Wert des Ausdrucks, der im Anschluss steht als Rückgabewert zurück
  - Die Anweisung **`return`** `( k * t * t) / (2 * m);` simuliert also sozusagen das funktionale Konzept: sie 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)
  - Eine Anweisung endet immer mit einem **`;`**

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

Funktionen in Java

- Weiteres Beispiel:

  Die Funktion *Arbeit*, die in Algorithmus 9 gegeben ist, 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 Sorten wird nicht explizit angegeben
- Die Menge der Operationen besteht aus den vereinbarten (statischen) Methoden
- Beispiel: wir könnten die Operationen `strecke` und `arbeit` z.B. in ein Modul (eine Klasse) `Physik` packen (siehe nächste Folie)

Module in Java

```
/**
   * Ein Modul, dass Operationen f&uuml;r physikalische
   * Berechnungen zur Verf&uuml;gung stellt.
   */
public class Physik {

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

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

(Datei `Physik.java`)

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 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 am häufigeren 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. (siehe Algorithmus 14):

```
/**
 * 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

- Nun ist es soweit: Sie können theoretisch 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, d.h. letztlich die Auswertung von Funktionen, gelten 1-zu-1 für Java (und übrigens auch andere Sprachen)!!!
- Wenn Sie verstanden haben, was bei so einer Auswertung abläuft, haben Sie eines der wesentlichen Geheimnisse der funktionaler Programmierung verstanden
- Und Sie erahnen vielleicht jetzt auch, dass eine andere Programmiersprache (z.B. SML oder C) diese Konzepte (wenn überhaupt) einfach nur anders aufschreibt, dass aber der Kern gleich bleibt