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

    $$\mathtt{IF}\ b\ \mathtt{THEN}\ a_1\ \mathtt{ELSE}\ a_2\ \mathtt{ENDIF}$$

    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.

Beispiel

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

FUNCTION $abs(x : \mathbb{Z}) \to \mathbb{N}_0$
OUTPUT Absolutbetrag
BODY **IF** $x \geq 0$ **THEN** x **ELSE** $-x$ **ENDIF**

Mit impliziter Sortenanpassung: Der Vorzeichenoperator hat die Signatur $\mathbb{Z} \to \mathbb{Z}$, d.h. sowohl der Ausdruck x als auch $-x$ muss implizit nach $\mathbb{N}_0$ gecasted werden.

Beispiel

Wir wollen die der Größe nach mittlere von drei natürlichen Zahlen $x, y, z \in \mathbb{N}_0$ bestimmen:

FUNCTION $mitte(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z}) \to \mathbb{Z}$

  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

Wert bedingter 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 $W(u) = W(a_2)$
- Die rekursive Definition von $W(u)$ kann entsprechend ergänzt werden.

Beispiel

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

$W(abs(-3)) = W(\texttt{\textbf{IF}}\ x \geq 0\ \texttt{\textbf{THEN}}\ x\ \texttt{\textbf{ELSE}}\ -x\ \texttt{\textbf{ENDIF}})$

Dazu: $W(x \geq 0) = (x \geq 0)[x/-3] = x[x/-3] \geq 0[x/-3]$
$= -3 \geq 0 = \mathit{FALSE}$

Also:
$W(\texttt{\textbf{IF}}\ x \geq 0\ \texttt{\textbf{THEN}}\ x\ \texttt{\textbf{ELSE}}\ -x\ \texttt{\textbf{ENDIF}}) = W(-x) = -x[x/-3] = --3 = 3$

Rekursion

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

FUNCTION $fak(x : \mathbb{N}_0) \rightarrow \mathbb{N}_0$
  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.

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

$$
\begin{aligned}
& W_{[x/3]}(fak(x)) \\
&\quad = W_{[x/3]}(\texttt{IF}\ x = 0\ \texttt{THEN}\ 1\ \texttt{ELSE}\ x \cdot fak(x-1)\ \texttt{ENDIF}) \\
&\quad = W_{[x/3]}(x \cdot fak(x-1)) \qquad (\text{da } W_{[x/3]}(x = 0) = FALSE) \\
&\quad = W_{[x/3]}(x) \cdot W_{[x/3]}(fak(x-1)) \\
&\quad = 3 \cdot W(fak((x-1)[x/3])) \\
&\quad = 3 \cdot W(fak(2))
\end{aligned}
$$

- Fortsetzung:

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

$$
\begin{aligned}
& W_{[x/2]}(fak(x)) \\
& \quad = W_{[x/2]}(\textbf{IF } x = 0 \textbf{ THEN } 1 \textbf{ ELSE } x \cdot fak(x-1) \textbf{ ENDIF}) \\
& \quad = W_{[x/2]}(x \cdot fak(x-1)) \qquad (\text{da } W_{[x/2]}(x = 0) = \mathit{FALSE}) \\
& \quad = W_{[x/2]}(x) \cdot W_{[x/2]}(fak(x-1)) \\
& \quad = W_{[x/2]}(x) \cdot W(fak((x-1)[x/2])) \\
& \quad = 2 \cdot W(fak(1))
\end{aligned}
$$

- Fortsetzung:

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

$$\begin{aligned}
&W_{[x/1]}(\mathit{fak}(x))\\
&\quad = W_{[x/1]}(\mathbf{IF}\ x = 0\ \mathbf{THEN}\ 1\ \mathbf{ELSE}\ x \cdot \mathit{fak}(x-1)\ \mathbf{ENDIF})\\
&\quad = W_{[x/1]}(x \cdot \mathit{fak}(x-1)) \qquad (\text{da } W_{[x/1]}(x=0) = \mathit{FALSE})\\
&\quad = W_{[x/1]}(x) \cdot W_{[x/1]}(\mathit{fak}(x-1))\\
&\quad = 1 \cdot W(\mathit{fak}(0))
\end{aligned}$$

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

$$\begin{aligned}
&W_{[x/0]}(\mathit{fak}(x))\\
&\quad = W_{[x/0]}(\mathbf{IF}\ x = 0\ \mathbf{THEN}\ 1\ \mathbf{ELSE}\ x \cdot \mathit{fak}(x-1)\ \mathbf{ENDIF})\\
&\quad = W_{[x/0]}(1) \qquad (\text{da } W_{[x/0]}(x=0) = \mathit{TRUE})\\
&\quad = 1
\end{aligned}$$

- Fortsetzung:

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

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

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

- 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

Behauptung: $W(fak(n))$ ergibt sich für ein beliebiges $n \in \mathbb{N}$ in endlich vielen Auswertungsschritten

Induktionsanfang $n = 0$:
$W(fak(0)) = W_{[x/0]}(fak(x))$. Diese Auswertung geht in endlich vielen Schritten (siehe Schritt 4) in obigem Ablauf-Beispiel).

Induktionsschritt $n \rightarrow n + 1$:
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, und damit ist es auch $(n + 1) \cdot W(fak(n))$.

Weitere **Beispiele** für rekursive (funktionale) Algorithmen

- Summenformel

  FUNCTION $\mathit{summeBis}(n : \mathbb{N}_0) \rightarrow \mathbb{N}_0$
  OUTPUT Summe von 0 bis n
  BODY **IF** $n = 0$ **THEN** 0 **ELSE** $n + \mathit{summeBis}(n - 1)$ **ENDIF**

- Fibonacci-Zahlen

  FUNCTION $\mathit{fib}(x : \mathbb{N}_0) \rightarrow \mathbb{N}_0$
  OUTPUT x-te Fibonacci-Zahl
  BODY **IF** $x = 0 \vee x = 1$ **THEN** 1 **ELSE** $\mathit{fib}(x - 1) + \mathit{fib}(x - 2)$ **ENDIF**

Von der Theorie zur Praxis

Zunächst eine kurze Zusammenfassung:

- Wir haben zunächst die als gegeben angenommenen Grunddatentypen und deren Grundoperationen eingeführt (als Module) und uns deren Umsetzung in Java angeschaut (primitive/atomare Datentypen).
- Das Konzept der Ausdrücke (Terme) folgte:
  - wir haben die Struktur (Syntax) definiert,
  - wir haben definiert, wie Ausdrücke interpretiert werden können (Semantik),

  ... in Theorie und Praxis (einfache Ausdrücke in Java).
- Durch Ausdrücke können wir nun den funktionalen Zusammenhang zwischen Ein- und Ausgabedaten zu spezifizieren, also funktionale Algorithmen zu entwerfen.

Von der Theorie zur Praxis

- Das Modulkonzept wurde erweitert, um diese funktionalen Algorithmen als Funktionen in Modulen zu definieren (implementieren).
- Die selbst-definierten Funktionen stehen damit auch anderen Algorithmen (Funktionen) zur Benutzung zur Verfügung und können dort 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).
- Die Einführung bedingter Ausdrücke ermöglichte schließlich Rekursion.

Wie geht es weiter?

- Wir wollen uns noch kurz anschauen, wie diese theoretischen Konzepte (Modul, Funktion, etc.) in einer konkreten Sprache (Java) umgesetzt sind.
- Java ist hier eigentlich ein schlechtes Beispiel, da Java hauptsächlich dem imperativen Paradigma folgt.
- Basierend auf den einfachen Java-Ausdrücken, die wir schon kennen, werden wir das nun dennoch tun.

Funktionen in Java

- In Java heißen Funktionen (im Übrigen wie ihre imperativen Pendants, die Prozeduren) *Methoden*.
- Die Funktion *strecke* aus dem oben angegebenen Modul `BEWEGUNG` lässt sich in Java wie folgt notieren:

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

- Wir erinnern uns vage, daran, dass wir sauber kommentieren sollen und tun dies gleich ordentlich mit Javadoc Kommentaren:

- Das Ergebnis:

```
/**
  * Berechnung der Strecke, die ein K&ouml;rper mit einer
  * gegeben Masse, der eine gegebene Zeit lang mit einer auf
  * ihn einwirkenden konst. 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);
}
```

Erklärungen:

```java
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` können Sie zunächst ignorieren; es bedeutet, dass diese Methode (Funktion) von anderen Modulen aus verwendbar ist.
- Das Schlüsselwort `static` können Sie zunächst ebenfalls ignorieren; es zeigt an, dass es sich um einen rein imperativen (bzw. funktionalen) Algorithmus handelt.

Funktionen in Java

- In den Klammern { und } ist der Rumpf der Methode (Funktion) platziert.
  - 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 eine Methode, die aus mind. einer **`return`**-Anweisung(en) besteht und keine Nebeneffekte hat.

- 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);
}
```

- Das Modulkonzept ist in Java durch Klassen umgesetzt.
- Klassen haben allerdings eigentlich einen anderen (Haupt-)Zweck.
- Ein Modul `MyModul` in Java ist eine Vereinbarung der *Klasse* `MyModul` in der Textdatei `MyModul.java`.
- Damit eine Klasse ein Modul in unserem Sinne darstellt, dürfen nur statische Methoden mit dem Schlüsselwort **`static`** vorkommen.
- Die Menge der Sorten wird nicht explizit angegeben, 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);
    }
}
```

- Es gibt in Java einige Klassen, Module in unserem Sinne sind, d.h. die eine Menge von Operationen (als statischen Methoden) zur Verfügung stellen.
- Ein sehr nützliches solches Modul ist z.B. die Klasse `Math` für grundlegende mathematische Funktionen (Exponent, Logarithmus, Sinus/Cosinus, etc.).
- Schauen Sie sich doch einfach mal die Dokumentation der Methoden 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

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

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

```
/** ... */
public static double strecke1(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) identischen Typ.

Bedingte Ausdrücke in Java

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 fehlen die Schlüsselwörter **`then`** und **`endif`**; zur besseren Lesbarkeit setzt man daher gerne sog. Blockklammern:

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

(was das genau ist, lernen wir bald kennen)

- 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 kennen nun alle Konzepte, um rein funktionale Algorithmen zu entwerfen.
- Sie wissen auch, wie Sie diese als Java-Programme schreiben können.
- Unsere theoretischen Konzepte, insbesondere die Semantik der Ausdrücke und deren Werte, gelten für Java genauso wie für alle anderen Sprachen.
- Wenn Sie verstanden haben, was bei der Auswertung eines Ausdrucks abläuft (z.B. wenn eine Funktion aufgerufen wird), haben Sie das Grundprinzip der funktionalen Programmierung verstanden.
- Dieses Kernprinzip gilt in allen Programmiersprachen.

Aufgabe: berechne für zwei Eingaben *a* und *b* vom Typ $\mathbb{B}$, wie oft davon *TRUE* vorhanden ist.

Algorithmus:

```
FUNCTION  wieOftTrue(a : 𝔹, b : 𝔹) → ℕ₀
  OUTPUT  Wieviel von a und b sind TRUE
  BODY
    IF  a ∧ b  THEN  2  ELSE
      IF  a ∨ 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

Aufgabe:

- 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: (an_s, an_m).
- Ausgabe ist die Fahrtzeit in Minuten.

Lösungsidee:

- Fall 1: die Fahrt geht nicht über Mitternacht.
  - Idee: Wandle die Zeiten (Stunden, Minuten) 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 $1070 - 730 = 340$.
  - Lösung für Fall 1:

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

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 $1610 - 1330 = 280$
  - Lösung für Fall 2:
    $$(an_s \cdot 60 + an_m + 1440) - (ab_s \cdot 60 + ab_m)$$

- 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

Java Programm:

```
/** ... */
public static int fahrtzeit(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

Aufgabe:

Berechne für ein $n \in \mathbb{N}_0$ die Anzahl ihrer Stellen (Ziffern), z.B. die Zahl 1234 hat 4 Stellen.

Idee:

Die ganzzahlige Division durch 10 ergibt bei 1-stelligen Zahlen 0 und bei mehrstelligen Zahlen wird die letzte Stelle abgeschnitten.

Algorithmus:

FUNCTION $stellen(x : \mathbb{N}_0) \to \mathbb{N}$
  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);
     }
}
```

Kapitel 4: Variablen, Anweisungen, Prozeduren ii

Road Map

- Wir wenden uns jetzt dem imperativen Paradigma zu.
- Zur Erinnerung: imperative Algorithmen werden typischerweise als Folge von *Anweisungen* formuliert.
- Diese Anweisung haben meist *Nebeneffekte*, bei denen Größen (meist in Form von *Variablen*) geändert werden.
- Wir werden sehen, dass einige der funktionalen Konzept der vergangenen Kapitel entsprechende imperative Pendants haben und dass sich beide Konzepte sehr gut miteinander vereinbaren lassen.

Variablen

- Im vorherigen Kapitel haben wir Ausdrücke nur mit Operationssymbolen (Literale und mehrstelligen Operatoren) gebildet; als Variablen haben wir nur die Eingabevariablen der Algorithmen (Funktionen) zugelassen.
- Diese Einschränkung über der Menge der für Ausdrücke zur Verfügung stehenden Variablen V geben wir jetzt auf:
- Wir erlauben nun auch weitere Variablen in Ausdrücken, allerdings müssen diese vorher *vereinbart*, also bekannt, sein.

- Wozu sind Variablen gut?
- Um diese Frage zu beantworten, betrachten wir zunächst ein sehr einfaches Beispiel.
- Als Vorwarnung: es geht nicht darum, ob das, was wir da machen, besonders sinnvoll ist oder nicht.
- Es geht darum, dass wir nun mit einem anderen Paradigma an die Lösung eines Problem herangehen.

Variablen

- Unser Beispiel ist folgender (in funktionaler Darstellung gegebener) Algorithmus:
  - Berechne die Funktion $f(x)$ für $x \neq -1$ mit $f : \mathbb{R} \to \mathbb{R}$ gegeben durch

$$f(x) = \left(x + 1 + \frac{1}{x + 1} \right)^2 \quad \text{für } x \neq -1$$

  - Eine imperative Darstellung erhält man z.B. durch Aufteilung der Funktionsdefinition in mehrere *Anweisungen*, die nacheinander auszuführen sind:

$$y_1 = x + 1;$$
$$y_2 = y_1 + \tfrac{1}{y_1};$$
$$y_3 = y_2 * y_2;$$
$$f(x) = y_3.$$

Variablen

- Intuition des Auswertungsvorgangs der imperativen Darstellung:
  - y_1, y_2 und y_3 repräsentieren drei Zettel.
  - Auf diese Zettel werden der Reihe nach Rechenergebnisse geschrieben (Werte verändert $\Rightarrow$ *Nebeneffekte*).
  - Bei Bedarf wird der Wert auf dem Zettel abgelesen.
- Formal steckt hinter dieser Intuition eine Substitution:
  - x wird beim Aufruf der Funktion wie bisher durch den Eingabewert substituiert.
  - y_1 wird mit dem Wert des Ausdrucks $x + 1$ substituiert wobei x bereits substituiert wurde (der Wert von y_1 ist damit beim Aufruf von f wohldefiniert).
  - Mit y_2 und y_3 kann man analog verfahren.

- Bei genauerer Betrachtung:
  - Nachdem der Wert von y_1 zur Berechnung von y_2 benutzt wurde, wird er im folgenden nicht mehr benötigt.
  - Eigentlich könnte der Zettel nach dem Verwenden (Ablesen). *radiert* und für die weiteren Schritte wiederverwendet werden
  - In diesem Fall kämen wir mit einem Zettel y aus:

    $$y = x + 1;$$
    $$y = y + \frac{1}{y};$$
    $$y = y * y;$$
    $$f(x) = y.$$

  Hier wird das Konzept der Nebeneffekte vielleicht noch klarer: der Wert von y verändert sich mehrmals.

Beobachtung 1:

- Es gibt also offenbar *radierbare* Zettel (*Variablen*) und *nicht-radierbare* Zettel (*Konstanten*[7]).

Beobachtung 2:

- Der Wert von *f* wurde jetzt nicht durch einen (auswertbaren) Ausdruck angegeben, sondern durch eine Menge von Anweisungen, die sequentiell abzuarbeiten sind.
- Das ist genau genommen ein imperativer Algorithmus, den wir nun als *Prozedur* (imperatives Pendant zur Funktion) darstellen.

[7] Nicht zu verwechseln mit Literalen!

- In unserer Modul-Schreibweise (Pseudo-Sprache) würden wir diese Prozedur für f wie folgt notieren:

Mit mehreren Variablen:

PROCEDURE $fBerechnen1(x : \mathbb{R}) \rightarrow \mathbb{R}$
  OUTPUT Berechnung der Funktion f
  PRE $x \neq 1$
  BODY
    VAR $y_1, y_2, y_3 \in \mathbb{R}$;
    $y_1 \leftarrow x + 1$;
    $y_2 \leftarrow y_1 + 1/y_1$;
    $y_3 \leftarrow y_2 \cdot y_2$;
    RETURN y_3;

Mit einer Variablen:

PROCEDURE $fBerechnen2(x : \mathbb{R}) \rightarrow \mathbb{R}$;
  OUTPUT Berechnung der Funktion f
  PRE $x \neq 1$
  BODY
    VAR $y \in \mathbb{R}$
    $y \leftarrow x + 1$;
    $y \leftarrow y + 1/y$;
    $y \leftarrow y \cdot y$;
    RETURN y;

In beiden Algorithmen benutzen wir wieder eine implizite Sortenanpassung.

Prozeduren

- Der einzige Unterschied zu unseren bisherigen Fuktionen:
  - Der Name **`PROCEDURE`** anstelle von **`FUNCTION`** zeigt an, dass es sich um eine Prozedur, also einem imperativen Algorithmus handelt (alles andere von Signatur über Eingabevariablen gilt analog).
  - Im Prozedur-Rumpf (nach **`BODY`**) stehen jetzt eine Menge von Anweisungen.
  - Bei diesen Anweisung handelt es sich um Anweisungen mit Variablen (wir erlauben Analoges mit Konstanten — siehe folgende Folien).
  - Die letzte Anweisung beginnt mit dem Schlüsselwort **`RETURN`**, gefolgt von einem Ausdruck a; dies bedeutet, dass der Wert von a als Ergebnis der Prozedur zurück gegeben werden soll.

Anweisungen mit Variablen und Konstanten

- Variablen und Konstanten können *deklariert* werden,
  z.B. $\texttt{VAR}\ y_1, y_2, y_3 \in \mathbb{R}$ in *fBerechnen*1
  bzw. $\texttt{VAR}\ y \in \mathbb{R}$ in *fBerechnen*2.
  - Intuitiv wird dabei ein *leerer Zettel* (Speicherzelle) angelegt.
  - Formal bedeutet die Deklaration einer Variablen/Konstanten v, dass der Bezeichner v zu der Menge der zur Verfügung stehenden Variablen V, die wir in Ausdrücken (im Rumpf) verwenden dürfen, hinzugefügt wird.
  - Der Typ der Variable muss bei der Deklaration angegeben werden.
- Um anzuzeigen, dass wir Konstanten deklarieren, verwenden wir das Schlüsselwort **CONST** statt **VAR**.
- Wir fassen Variablen- und Konstantendeklarationen formal ebenfalls als Anweisungen auf.

Anweisungen mit Variablen und Konstanten

- Achtung: Variablen sind (bei uns) Ausdrücke und haben daher (bei uns) einen Typ!
- Besonderheit: Wir fordern, dass zu Beginn einer Prozedur alle Variablen bekannt gemacht werden müssen (das ist nicht in allen Programmiersprachen so, bspw. auch in Java nicht; es erleichtert uns aber das Leben etwas — die nachfolgenden Formalisierungen sind aber leicht erweiterbar).
- Damit enthält die Menge V der in Ausdrücke im Prozedurrumpf erlaubten Variablen alle Eingabe-Variablen (im Prozedurkopf) und alle im Rumpf unter **`VAR`** deklarierten Variablen bzw. unter **`CONST`** deklarierten Konstanten.

Anweisungen mit Variablen und Konstanten

- Deklarierte Variablen und Konstanten können *intialisiert* werden, d.h. erstmalig einen (Wert eines) Ausdruck(s) zugewiesen bekommen,
  z.B. $y_1 \leftarrow x + 1$ in *fBerechnen*1.
  - Intuitiv wird durch $v \leftarrow a$ der Ausdruck a auf den Zettel v (in die Speicherzelle) geschrieben.
  - Dadurch wird der Variablen/Konstanten $v \in V$ der Wert des Ausdrucks a zugewiesen (dies nennt man auch *Wertzuweisung*).
  - Formal vereinbart $v \leftarrow a$ die Substitution $[v/a]$.
- Die Schreibweise $x \leftarrow y$ weist also der Variablen/ Konstanten x den Wert des Ausdrucks y zu; wir fordern sinnvollerweise, dass x und y den selben Typ haben.

- Der Wert einer Variablen kann später auch noch durch eine weitere *Wertzuweisung* an einen Ausdruck (mit gleichem Typ) verändert werden,
  z.B. $y \leftarrow 1/y$ in *fBerechnen*2.
- Dabei kann man auch auf den alten Wert des Zettels zurückgreifen (ihn vor dem radieren ablesen und sich merken), d.h. der neue Wert kann vom alten abhängig sein.
  - Intuitiv wird der alte Wert auf dem Zettel radiert und der Wert des neuen Ausdrucks auf den Zettel geschrieben.
  - Formal also eine erneute Substitution.

Grundsätzlich gilt wie oben besprochen:

- *Konstanten* können nur einmal verändert (initialisiert) werden.
- *Variablen* können beliebig oft durch Wertzuweisungen verändert werden.

- Wir illustrieren noch kurz die Verwendung von Konstanten an der Funktion f von oben, z.B. indem wir y_1, y_2, y_3 als Konstanten verwenden, da sie nur einmal initialisiert und dannach nicht mehr verändert werden:

Mit Konstanten:

PROCEDURE *fBerechnen*3($x : \mathbb{R}$) $\to \mathbb{R}$
  OUTPUT Berechnung der Funktion f
  PRE $x \neq 1$
  BODY
    CONST $y_1, y_2, y_3 \in \mathbb{R}$;
    $y_1 \leftarrow x + 1$;
    $y_2 \leftarrow y_1 + 1/y_1$;
    $y_3 \leftarrow y_2 \cdot y_2$;
    RETURN y_3;

Mit Variablen *und* Konstanten:

PROCEDURE *fBerechnen*4($x : \mathbb{R}$) $\to \mathbb{R}$;
  OUTPUT Berechnung der Funktion f
  PRE $x \neq 1$
  BODY
    VAR $v \in \mathbb{R}$;
    CONST $c \in \mathbb{R}$;
    $v \leftarrow x + 1$;
    $v \leftarrow v + 1/v$;
    $c \leftarrow v \cdot v$;
    RETURN c;

- Die *Prozedur* dient (ähnlich wie die Funktion) zur *Abstraktion* von Algorithmen (genauer: von den einzelnen Schritten eines Algorithmus).
- Wie bei Funktionen wird durch Parametrisierung von der Identität der Daten abstrahiert:
  - Die Berechnungsvorschriften werden mit abstrakten (Eingabe-) Parametern (Variablen) formuliert (bei Funktionen wird u.a. aus diesen Variablen der Rumpf gebildet).
  - Konkrete Eingabedaten bilden die aktuellen (Parameter-) Werte mit denen die Eingabedaten dann beim Aufruf substituiert werden.

Funktionen und Prozeduren

- Durch Spezifikation des (Ein- / Ausgabe-) Verhaltens wird von den Implementierungsdetails abstrahiert.
- Dies hat folgende Vorteile:
  - Örtliche Eingrenzung (*Locality*): Die Implementierung einer Abstraktion kann verstanden oder geschrieben werden, ohne die Implementierungen (also die einzelnen Anweisungen) anderer Abstraktionen kennen zu müssen.
  - Änderbarkeit (*Modifiability*): Jede Abstraktion kann reimplementiert (z.B. die Anweisungen verändert) werden, ohne dass andere Abstraktionen geändert werden müssen.
  - Wiederverwendbarkeit (*Reusability*): Die Implementierung einer Abstraktion kann beliebig wiederverwendet werden.

Funktionen und Prozeduren

- Funktionen und Prozeduren haben also gewisse Gemeinsamkeiten: eine Funktion kann man als Prozedur bezeichnen, nicht jede Prozedur ist jedoch eine Funktion.
- Eine Funktion stellt nur eine Abbildung von Elementen aus dem Definitionsbereich auf Elemente aus dem Bildbereich dar, es werden aber keine Werte verändert.
- Im imperativen Paradigma können Werte von Variablen verändert werden (durch Anweisungen), dies kann Effekte auf andere Bereiche eines Programmes haben.
- Treten in einer Prozedur solche Seiteneffekte auf, kann man nicht mehr von einer Funktion sprechen.
- Eine Funktion kann man also als eine Prozedur ohne Seiteneffekte auffassen.

- Das beliebte „erste“ Beispiel für ein Java Programm verwendet übrigens eine Prozedur `main` in einem Modul `HelloWorld`:

```
public class HelloWorld {

  public static void main(String[] args) {
    // Hier passiert der Seiteneffekt:
    System.out.println("Hello, World!");
  }
}
```

- Der Seiteneffekt ist die Ausgabe von `"Hello World!"` auf der Kommandozeile.