

```

let rec binomial (n, k) = match (n, k) with
| (0, k) -> 1
| (n, k) when (k <= 0 || k >= n) -> 1
| (n, k) -> binomial(n-1,k) + binomial(n-1, k-1);

```

Die Übertragung in ein Java-Programm lautet dann:

```

class Binomialkoeffizienten {

    public static int binomialkoeff (int n, int k){
        if (n == 0 || k <=0 || k >= n) return 1;
        else return binomialkoeff (n-1, k) + binomialkoeff(n-1, k-1);
    }
    public static void main (String[] args) {
        System.out.println(binomialkoeff(10, 7));
    }
}

```

Prinzipiell besteht zwischen rekursiven Methoden in Java und rekursiven Funktionen in Ocaml somit nur ein syntaktischer Unterschied. Einige Kleinigkeiten sollten jedoch erwähnt werden:

- Der Vergleichsoperator „==“ in Java: Das einfache Gleichheitszeichen = wird in Java für die Zuweisung von Ausdrücken zu Variablen verwendet(siehe Abschnitt (3)). Sollen dagegen die Werte zweier Ausdrücke auf Gleichheit überprüft werden, so ist das doppelte Gleichheitszeichen == zu verwenden. Direkt können in Java jedoch nur Ausdrücke, deren Werte Grundtypen (d.h. **boolean**, **int**, **double**, **char**) sind, verglichen werden (dazu später mehr).
- Um rekursive Methoden formulieren zu können ist die BNF-Regel für den Methodenrumpf in folgender Weise zu erweitern:

```

<Methodenrumpf>      ::= <Rückgabeeinweisung> | <Methodenaufruf>
                        | <Bedingte Anweisung>

```

## Konditionaloperator

Der Konditionaloperator stellt eine weitere Möglichkeit dar, Alternativen in Programmen zu implementieren. Es handelt sich hier jedoch nicht um eine Anweisung, sondern um einen Ausdruck. Das bedeutet beispielsweise, dass **return**-Anweisungen nicht mehr in den jeweiligen Alternativzweigen auftreten, sondern der bedingte Ausdruck als ganzes zurückgegeben wird. Der Konditionaloperator ist, ähnlich wie die bedingte Anweisung, ein (nichtstrikt) dreistelliger (oder ternärer) Operator, dessen Argumente wie bei der bedingten Anweisung strukturiert sind: Das 1. Argument bildet die Bedingung, das 2. Argument das Ergebnis des Operators bei erfüllter Bedingung und das dritte Argument das Ergebnis bei nicht erfüllter Bedingung. Mit dem Konditionaloperator lassen sich rekursive Aufrufe noch „funktionaler“ schreiben, wie folgendes Beispiel zeigt:

```

public static int binomialkoeff (int n, int k){
    return ((n == 0 || k <=0 || k >= n)
        ? 1
        : binomialkoeff (n-1, k) + binomialkoeff(n-1, k-1));
}

```

Das angegebene Programm zeigt die syntaktische Struktur des Konditionaloperators:

```

<Konditionaloperator>      ::= <Boolescher Ausdruck> '?' <True-Fall> ':' <False-Fall>

```

Hierbei müssen die Typen der beiden letzten Argumente natürlich identisch sein.

## 3 Zuweisungsorientierte Kontrollstrukturen in Java

Bisher wurden die Programmiersprache Java nur insoweit erläutert, als funktionale Sprachelemente umgesetzt werden sollen. Java ist jedoch im Kern keine funktionale, sondern eine objektorientierte Sprache. Bevor wir uns mit den damit zusammenhängenden Begriffen (Objekt, Klasse, etc.) beschäftigen, sollen die Konzepte der zuweisungsorientierten Programmierung erläutert werden, deren Grundkonzepte ebenfalls Bestandteil von Java sind. Die zuweisungsorientierte Beschreibung von Abläufen stellt eine Alternative zur funktionalen Beschreibung von Algorithmen dar. Vom theoretischen Standpunkt sind funktionale und zuweisungsorientierte Beschreibung vollkommen gleichwertig. Unter praktischen Gesichtspunkten ergeben sich jedoch Unterschiede: Während die funktionale Modellierung eine kompakte und algorithmisch durchsichtige Implementierung von Algorithmen erlaubt, kann mit Hilfe zuweisungsorientierter Methoden ein Algorithmus häufig effizienter umgesetzt werden.

### 3.1 Variablen

Das entscheidende Konzept, das die zuweisungsorientierte Programmierung von der funktionalen unterscheidet, ist das Konzept der Variable. Bildhaft kann man sich eine Variable als Behälter vorstellen, der mit Werten gefüllt werden kann; diese Werte können natürlich jederzeit aktualisiert werden. Variablen werden deshalb idealerweise dazu verwendet, Zwischenergebnisse zu speichern oder sie weiter zu reichen. Ein derartiges Zwischenergebnis kann ein boolescher Wert, eine Zahl, oder auch kompletter Baum sein. Je nach Datentyp ist deshalb der Speicherbedarf verschieden: Ein boolescher Wert benötigt sicherlich weniger Speicher als ein binärer Baum. Deshalb müssen Variablen deklariert werden: Dabei ist mindestens der Name und der Typ der Variablen anzugeben. Variablen im strengen Sinn der zuweisungsorientierten Programmierung treten nur innerhalb von Methoden auf. Entsprechende Konstrukte, die sich auf die ganze Klasse beziehen nennt man Attribute; sie sind Gegenstand des Abschnitts (4).

#### Variablendeklaration und Zuweisung

Wie erwähnt müssen Variablen deklariert werden, d.h. es muss der Variablen ein Name und ein Datentyp zugeordnet werden, wie in folgendem Beispiel illustriert:

```
int ganzeZahl;  
double gleitpunktzahl;  
String zeichenkette, nochEineZeichenkette;
```

Listing 4: "Variablendeklaration"

Es lassen sich also auch Variablen gleichen Typs in einer Anweisung deklarieren, wie bei den Zeichenketten illustriert. Insgesamt erkennt man folgende BNF-Regeln für die Variablendeklaration:

$\langle \text{Variablendeklaration} \rangle ::= \langle \text{Typ} \rangle \langle \text{Variablenbezeichner} \rangle \mid (', ' \langle \text{Variablenbezeichner} \rangle)^*$

Zwar ist es in Java möglich an (fast) beliebigen Stellen Variablen zu deklarieren, doch empfiehlt es sich aus Gründen der Übersichtlichkeit Variablen stets zu Beginn eines Methodenrumpfes festzulegen.

Nachdem eine Variable deklariert wurde, kann sie mit Werten belegt werden (anschaulich wird der Behälter mit einem Wert gefüllt). Dies geschieht durch die Zuweisung, deren Syntax in einfacher Weise durch die Regel

$\langle \text{Zuweisung} \rangle ::= \langle \text{Variablenbezeichner} \rangle '=' \langle \text{Ausdruck} \rangle ';'$

gegeben ist. Im Falle des Programmbruchstücks (4) könnten möglich Zuweisungen lauten:

```
ganzeZahl = 5;
gleitpunktzahl = 3.14159;
zeichenkette = "Irgendein_Text";
```

Im Laufe eines Programms lässt sich der Wert einer Variablen durch eine erneute Zuweisung beliebig oft überschreiben. Dadurch unterscheidet sich der Variablenbegriff der Informatik von dem der Mathematik. Der Vollständigkeit halber sollte nicht unerwähnt bleiben, dass Deklaration und erstmalige Wertzuweisung, die Initialisierung, in einer so genannten Deklarationsanweisung gekoppelt werden kann:

```
int ganzeZahl = 5;
double gleitpunktzahl = 3.14159;
String zeichenkette = "Irgendein_Text";
```

Die entsprechende BNF-Regel lautet dann:

<Deklarationsanweisung> ::= <Variablendeklaration> '=' <Ausdruck> ';'

In Java wird jede Variable nach der Deklaration implizit, d.h. ohne Zutun des Programmierers initialisiert, in einen definierten Anfangszustand versetzt. Beispielsweise werden ganze Zahlen mit 0 initialisiert. Dennoch sollte jede Variable vor der Verwendung in einem Ausdruck explizit vom Programmierer belegt werden.

### Verbundanweisungen und lokale Variablen

Das zentrale Konzept zur Strukturierung von Programmen in Java sind Blockstrukturen, die einen modularen Programmaufbau ermöglichen. Neben Klassendeklarationen, die auch als Blockstruktur angesehen werden können (siehe Abschnitt (4)), sind vor allem Anweisungsblöcke oder Verbundanweisungen für die Programmierung interessant. Ein Block besteht aus einem Paar geschweiften Klammern und kann Variablendeklarationen und Programmanweisungen beinhalten. Blöcke selbst sind Programmanweisungen, d.h. in einem Block können weitere Blöcke auftreten. Durch die Schachtelung von Blöcken ist eine Strukturierung des Programmtextes möglich.

Fassen wir die bisherigen Syntaxregeln für Anweisungen zusammen, so erhalten wir folgendes System von Regeln:

```
<Anweisung> ::= <Zuweisung> | <Deklarationsanweisung>
               | <Bedingte Anweisung> | <Verbundanweisung> | <Rückgabeanweisung>
<Verbundanweisung> ::= '{' <Anweisungssequenz> '}'
<Anweisungssequenz> ::= (<Anweisung>)*
```

Werden innerhalb eines Blocks Variablen deklariert, so bezeichnet man diese als lokale Variablen. Lokale Variablen sind nur innerhalb des Blocks verwendbar, in dem sie deklariert werden. Ein Zugriff von aussen ist nicht zulässig, egal ob von einem umschließenden oder einem separaten Block aus.

```
int variable = 0;
{
    int lokaleVariable = 1;
}
variable = lokaleVariable; // FEHLER: Zugriff auf lokale Variable von aussen!
```

## Lebensdauer und Gültigkeitsbereich

Die Lebensdauer einer Variablen wird bestimmt durch den Ort ihrer Deklaration im Programmtext. Die Lebensdauer einer lokalen Variable endet bei Verlassen des umschließenden Blockes (im Normalfall am Blockende) bzw. der Methode. Nach Verlassen des Blockes steht der Speicherbereich der lokalen Variable wieder für andere Verwendungszwecke zur Verfügung. Auf eine Variable kann während ihrer Lebensdauer innerhalb ihres Gültigkeitsbereiches zugegriffen werden. Dieser umfasst – je nach Variablenart – einen bestimmten Block, wobei u.U. in einem inneren Block eine Variable gleichen Namens deklariert werden kann, die in diesem Abschnitt die äussere Variable verschattet. In Java dürfen lokale Variablen sich nicht gegenseitig verschatten, aber sie können Klassen- und Instanzvariablen verschatten.

```
int variable = 0;
{
    variable = 1;
}
{
    int variable = 1; // FEHLER: Verschattung in Java nicht erlaubt!
}
```

Wichtig: Begriffe wie Lokalität, Gültigkeit und Lebensdauer beziehen sich vor allem auf den Programmtext und weniger auf die Programmausführung. Geschachtelte Methodenaufrufe z.B. sind uneingeschränkt möglich, da jeder Methodenaufruf als separater Block abläuft. Man spricht dabei auch von einer Inkarnation der Methode. Alle Inkarnationen einer Methode besitzen für ihre lokalen Variablen jeweils eigene Speicherbereiche. Lokale Variablen werden also zwischen den Inkarnationen nicht geteilt, sondern jede Inkarnation besitzt eigene „Exemplare“ der lokalen Variablen. Dies ist insbesondere von Bedeutung für die Rekursion.

## 3.2 Anweisungen

Anweisungen dienen zur Steuerung des Ablaufs von Java-Programmen. Anders als Ausdrücke können Anweisungen keinem Datentyp zugeordnet werden, d.h. eine Anweisung liefert keinen Rückgabewert. Anweisungen ändern lediglich die Belegung der einzelnen Variablen und stellen den Ablaufrahmen für die Auswertung von Ausdrücken dar. Neben den schon zuvor verwendeten bedingten Anweisungen (2.3) und Zuweisungsanweisungen sind die Wiederholungsanweisungen die wesentlichen Anweisungsarten. In Java existieren noch andere Anweisungen für spezielle Aufgaben wie z.B. zur Fehlerbehandlung. Diese Anweisungen werden erst in späteren Kapiteln eingeführt und daher hier nicht weiter behandelt.

Einzelne Anweisungen werden durch einen Strichpunkt abgeschlossen. Sind mehrere Anweisungen in einem Block zusammengefasst, ist mit dem Abschluss der letzten Anweisung innerhalb des Blocks der gesamte Block abgeschlossen, d.h. der Block selbst muss nicht mehr mit einem Strichpunkt abgeschlossen werden.

Die Abarbeitung von Anweisungen erfolgt sequentiell, die Anweisungen werden also der Reihe nach ausgeführt. Kann eine Anweisung selbst weitere Anweisungen enthalten, so ist eine Schachtelung von Anweisungen möglich. In diesem Fall werden die Unteranweisungen nach den Regeln ausgeführt, die für die jeweilige umfassende Anweisung gelten. Ein Block von Anweisungen wird dabei wie eine einzelne Anweisung behandelt.

### Schleifen

Wiederholungsanweisungen bzw. Schleifen erlauben die wiederholte Ausführung einer Anweisung bzw. Anweisungsfolge. Ähnlich wie bei der bedingten Anweisung gilt auch hier, dass Anweisungsfolgen, die wiederholt ausgeführt werden sollen, in einem Block zusammengefasst werden müssen. Die zu wiederholende Anweisung wird auch als Schleifenkörper bezeichnet. In Java gibt es drei verschiedene Wiederholungsanweisungen, die sich in erster Linie dadurch unterscheiden, wie und wann die Anzahl der Wiederholungsschritte

bzw. Iterationen festgelegt wird. Generell bieten alle Schleifen die Möglichkeit, eine Wiederholungsbedingung zu formulieren, die ähnlich wie die Terminierungsbedingung bei der Rekursion vor bzw. nach jeder Iteration überprüft wird.

Eine einfache Schleife, bei der die Überprüfung der Wiederholungsbedingung vor der Ausführung des Schleifenkörpers stattfindet, hat folgende Form:

<while-Anweisung> ::= 'while' <Bedingung> <Anweisung>

Die Anweisung kann natürlich im Sinne der Verbundanweisung aus mehreren Teilanweisungen zusammengesetzt sein. Vor jeder Iteration innerhalb einer while-Anweisung wird zunächst die Wiederholungsbedingung ausgewertet. Ergibt sich als Resultat false, dann ist die Wiederholungsbedingung nicht erfüllt. Die Ausführung der Schleife ist damit beendet, d.h. auch der Schleifenkörper wird nicht mehr ausgeführt. Ist die Wiederholungsbedingung aber erfüllt, so findet die Iteration statt und der Schleifenkörper wird ausgeführt.

Das folgende Beispiel berechnet den größten gemeinsamen Teiler zweier Zahlen:

```
public static int ggT(int a, int b){
    int a_neu = a;
    int b_neu = b;
    while (a_neu!=b_neu) {
        if (a_neu > b_neu) {
            a_neu = a_neu - b_neu;
        }
        else {
            b_neu = b_neu - a_neu;
        }
    }
    return a_neu;
}
```

Listing 5: "ggT zweier Zahlen"

Wichtig: Ist die Wiederholungsbedingung immer erfüllt, so ergibt sich eine sog. Endlos-Schleife, bei der unendlich viele Iterationen ausgeführt werden. Damit terminiert auch das Programm nie. In Programm (5) beispielsweise erhält man für  $a > 0$  und  $b = 0$ , eine nichtterminierende Methode.

## Zählschleife

Eine zweite Schleifenvariante ist die for-Anweisung, die speziell für Felddurchläufe und andere zählergesteuerte Wiederholungen geeignet ist:

<for-Anweisung> ::= 'for' '(' [<Typ>]<Zuweisung>; <Bedingung>; <Zuweisung> ')' <Anweisung>

Die einzelnen Elemente der runden Klammern haben dabei folgende Bedeutung: Zunächst kann ein geeigneter Schleifenzähler deklariert und initialisiert werden; anschließend folgt die Abbruchbedingung und als letztes Element folgt eine Zuweisung um den Schleifenzähler hochzuzählen (Fortschaltung). Initialisierung und Fortschaltung sind jeweils einzelne Anweisungen, die auch weggelassen werden können (die Strichpunkte als Begrenzer der Wiederholungsbedingung müssen aber auf alle Fälle gesetzt werden). Die Wiederholungsbedingung und der Schleifenkörper sind analog zur while-Anweisung definiert. Die Ausführung einer for-Anweisung beginnt mit der Ausführung der Initialisierung; Hier können Variablen mit geeigneten Werten belegt werden, es können aber auch neue Variablen deklariert werden. Eine solche Variable ist dann nur innerhalb der gesamten for-Anweisung gültig. Die anschließenden Iterationsschritte der for-Anweisung laufen ähnlich ab wie bei der while-Anweisung: Zunächst wird die Wiederholungsbedingung ausgewertet: Liefert die Auswertung das Resultat false, so ist die for-Anweisung beendet, und der Schleifenkörper wird nicht