



Kapitel 2

Das imperative Hamster-Modell

Computer können heutzutage zum Lösen vielfältiger Aufgaben genutzt werden. Die Arbeitsanleitungen zum Bearbeiten der Aufgaben werden ihnen in Form von Programmen mitgeteilt. Diese Programme, die von Programmierern entwickelt werden, bestehen aus einer Menge von Befehlen bzw. Anweisungen, die der Computer ausführen kann. Die Entwicklung solcher Programme bezeichnet man als *Programmierung*.

Das Hamster-Modell ist ein spezielles didaktisches Modell zum Erlernen der Programmierung. Im Hamster-Modell nimmt ein virtueller Hamster die Rolle des Computers ein. Diesem Hamster können ähnlich wie einem Computer Befehle erteilt werden, die dieser ausführt.

Ihnen als Programmierer werden bestimmte Aufgaben gestellt, die sie durch die Steuerung des Hamsters zu lösen haben. Derartige Aufgaben werden im Folgenden *Hamster-Aufgaben* genannt. Zu diesen Aufgaben müssen Sie in der *Hamster-Sprache* – eine Programmiersprache, die fast vollständig der Programmiersprache Java entspricht – Programme – *Hamster-Programme* genannt – entwickeln, die die Aufgaben korrekt und vollständig lösen. Die Aufgaben werden dabei nach und nach komplexer. Zum Lösen der Aufgaben müssen bestimmte Programmierkonzepte eingesetzt werden, die im Hamster-Modell inkrementell eingeführt werden.

Das ursprüngliche Hamster-Modell wird in dem Buch „Programmieren spielend gelernt mit dem Java-Hamster-Modell“ [Bol08b] eingeführt. In dem Buch – auch Band 1 der Java-Hamster-Bücher genannt – werden die Konzepte der imperativen Programmierung vorgestellt, weshalb das dort beschriebene Hamster-Modell im Folgenden auch als *imperatives Hamster-Modell* bezeichnet wird. Zur Vermittlung der Konzepte der objektorientierten Programmierung wird das imperative Hamster-Modell im folgenden Kapitel 3 leicht verändert. Das geänderte Hamster-Modell wird als *objektorientiertes Hamster-Modell* bezeichnet. Es basiert dabei auf dem imperativen Hamster-Modell, genauso wie die objektorientierten Programmierkonzepte, die Sie in diesem Buch – Band 2 der Java-Hamster-Bücher – kennen lernen, auf den imperativen Programmierkonzepten des ersten Bandes basieren.

Für diejenigen Leser, die den ersten Band der Java-Hamster-Bücher nicht durchgearbeitet haben, werden in diesem Kapitel die Grundlagen des imperativen Hamster-Modells sowie die wichtigsten Konzepte der imperativen Programmierung zusammengefasst.

2.1 Komponenten des Hamster-Modells

Die Grundidee des Hamster-Modells ist ausgesprochen einfach: Sie als Programmierer müssen einen (virtuellen) Hamster durch eine (virtuelle) Landschaft steuern und ihn gegebene Aufgaben lösen lassen.

2.1.1 Landschaft

Die Welt, in der der Hamster lebt, wird durch eine gekachelte Ebene repräsentiert. Abbildung 2.1 zeigt eine typische Hamsterlandschaft – auch Hamster-Territorium genannt – inklusive Legende. Die Größe der Landschaft, d.h. die Anzahl der Kacheln, ist dabei nicht explizit vorgegeben. Die Landschaft ist beliebig aber nie unendlich groß.

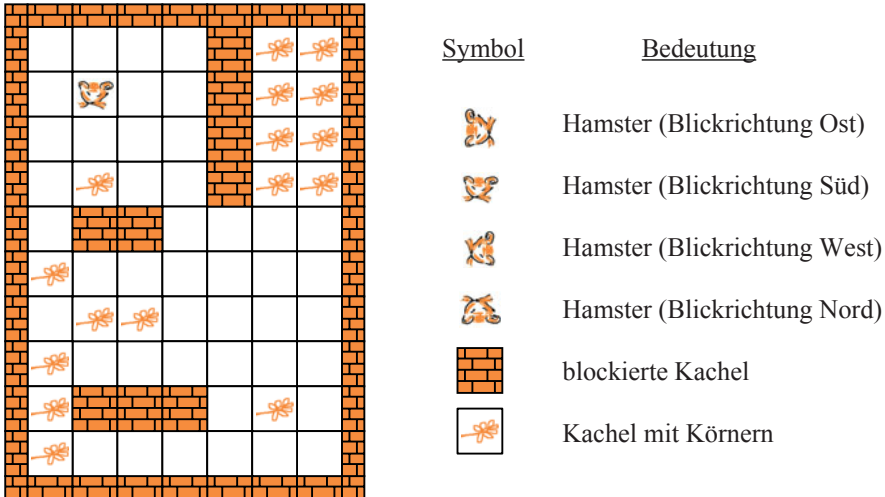


Abbildung 2.1: Komponenten des Hamster-Modells

Auf einzelnen Kacheln können ein oder mehrere Körner liegen. Kacheln, auf denen sich Körner befinden, sind in den Landschaftsskizzen durch ein spezielles Symbol gekennzeichnet. Dabei sagt das Symbol nur aus, dass auf der Kachel mindestens ein Korn liegt. Die genaue Anzahl an Körnern auf einem Feld geht aus der Landschaftsskizze nicht direkt hervor.¹

Auf den Kacheln des Hamster-Territoriums können weiterhin auch Mauern stehen, was bedeutet, dass diese Kacheln blockiert sind. Der Hamster kann sie nicht betreten. Es ist nicht möglich, dass sich auf einer Kachel sowohl eine Mauer als auch Körner befinden. Das Territorium ist immer vollständig von Mauern umgeben.

2.1.2 Hamster

Im imperativen Hamster-Modell existiert immer genau ein Hamster. Der Hamster steht dabei auf einer der Kacheln des Hamster-Territoriums. Diese Kachel darf nicht durch eine Mauer blockiert sein, sie kann jedoch Körner enthalten.

Der Hamster kann in vier unterschiedlichen Blickrichtungen (Nord, Süd, West, Ost) auf den Kacheln stehen. Je nach Blickrichtung wird der Hamster durch unterschiedliche Symbole repräsentiert.

Wenn der Hamster auf einer Kachel steht, auf der auch Körner liegen, wird in der Skizze das Kornsymbol nicht angezeigt, d.h. es kann aus der Skizze nicht direkt abgelesen werden, ob sich der Hamster auf einer Körnerkachel befindet.

¹ Aus Gründen eines besseren Verständnisses wird die Anzahl an Körnern auf einer Kachel in den Abbildungen der folgenden Kapitel ab und zu als Zahl hinzugefügt.

Körner können sich nicht nur auf einzelnen Kacheln, sondern auch im Maul des Hamster befinden. Ob der Hamster Körner im Maul hat und wenn ja, wie viele, ist ebenfalls nicht direkt aus der Landschaftsskizze ersichtlich.

Mit Hilfe bestimmter Befehle, die in den kommenden Abschnitten genauer erläutert werden, kann ein Programmierer den Hamster durch ein gegebenes Hamster-Territorium steuern. Der Hamster kann dabei von Kachel zu Kachel hüpfen, er kann sich drehen, Körner fressen und Körner wieder ablegen. Sie können sich den Hamster quasi als einen virtuellen Prozessor vorstellen, der im Gegensatz zu realen Computer-Prozessoren (zunächst) keine arithmetischen und logischen Operationen ausführen kann, sondern in der Lage ist, mit einem kleinen Grundvorrat an Befehlen ein Hamster-Territorium zu „erkunden“.

2.1.3 Grundlagen der Programmiersprache Java

Der Zeichenvorrat (die Lexikalik), den Sie beim Erstellen von Hamster-Programmen verwenden dürfen, entspricht dem 16-Bit-Zeichensatz *Unicode*.

Die *Token* einer Sprache, auch lexikalische Einheiten genannt, sind die Wörter, auf denen sie basiert. Wenn Sie Ihr Programm compilieren, teilt der Compiler Ihren Quellcode in Token auf und versucht herauszufinden, welche Anweisungen, Bezeichner und andere Elemente der Quellcode enthält. Token müssen in Java durch Wortzwischenräume voneinander getrennt werden. Zu den Wortzwischenräumen zählen Leerzeichen, Tabulatoren, Zeilenvorschub- und Seitenvorschubzeichen. Diese im Folgenden kurz als *Trennzeichen* bezeichneten Zeichen haben ansonsten keine Bedeutung.

Bezeichner, die zur Benennung von deklarierten Elementen (wie Prozeduren oder Variablen) verwendet werden, müssen in Java mit einem Buchstaben, einem Unterstrich (`_`) oder einem Dollarzeichen (`$`) beginnen, dem weitere Buchstaben, Unterstriche und Ziffern folgen können. Bezeichner dürfen beliebig lang sein.

In Java wird streng zwischen Groß- und Kleinbuchstaben unterschieden, d.h. dass bspw. die Bezeichner `rechts` und `Rechts` unterschiedliche Bezeichner sind.

Schlüsselwörter sind reservierte Wörter einer Programmiersprache. Sie dürfen nicht als Bezeichner verwendet werden.

2.2 Anweisungen und Programme

Programme setzen sich aus einer Menge von Befehlen bzw. Anweisungen zusammen.

2.2.1 Hamster-Befehle

Die Aufgabe eines Hamster-Programmierers besteht darin, den Hamster durch eine Landschaft zu steuern, um dadurch gegebene Hamster-Aufgaben zu lösen. Zur Steuerung des Hamsters müssen ihm Anweisungen in Form von Befehlen gegeben werden. Der Hamster besitzt dabei die Fähigkeit, vier verschiedene Befehle zu verstehen und auszuführen:

- `vor()` ;: Der Hamster hüpfte eine Kachel in seiner aktuellen Blickrichtung nach vorne.

- `linksUm()` ;: Der Hamster dreht sich auf der Kachel, auf der er gerade steht, um 90 Grad entgegen dem Uhrzeigersinn.
- `nimm()` ;: Der Hamster frisst von der Kachel, auf der er sich gerade befindet, genau ein Korn, d.h. anschließend hat der Hamster ein Korn mehr im Maul und auf der Kachel liegt ein Korn weniger als vorher.
- `gib()` ;: Der Hamster legt auf der Kachel, auf der er sich gerade befindet, genau ein Korn aus seinem Maul ab, d.h. er hat anschließend ein Korn weniger im Maul, und auf der Kachel liegt ein Korn mehr als vorher.

Wie Sie vielleicht schon festgestellt haben, können bei den Befehlen `vor`, `nimm` und `gib` Probleme auftreten:

- Der Hamster bekommt den Befehl `vor()` ; und die Kachel in Blickrichtung vor ihm ist durch eine Mauer blockiert.
- Der Hamster bekommt den Befehl `nimm()` ; und auf der Kachel, auf der er sich gerade befindet, liegt kein einziges Korn.
- Der Hamster bekommt den Befehl `gib()` ; und er hat kein einziges Korn im Maul.

Bringen Sie den Hamster in diese für ihn unlösbaren Situationen, dann ist der Hamster derart von Ihnen enttäuscht, dass er im Folgenden nicht mehr bereit ist, weitere Befehle auszuführen. Derartige Fehler werden *Laufzeitfehler* genannt. Laufzeitfehler können im Allgemeinen nicht schon durch den Compiler entdeckt werden, sondern treten erst während der Ausführung eines Programmes auf. Programme, die zu Laufzeitfehlern führen können, sind nicht korrekt!

2.2.2 Anweisungen

In imperativen Programmiersprachen werden Verarbeitungsvorschriften durch so genannte *Anweisungen* ausgedrückt. Anweisungen, die nicht weiter zerlegt werden können, werden *elementare Anweisungen* genannt. In der Hamster-Sprache sind die vier Grundbefehle elementare Anweisungen. Eine Folge von Anweisungen, die nacheinander ausgeführt werden, wird als *Anweisungssequenz* bezeichnet. Die einzelnen Anweisungen einer Anweisungssequenz werden in der angegebenen Reihenfolge hintereinander ausgeführt.

2.2.3 Programme

Ein Hamster-Programm setzt sich aus den Schlüsselwörtern `void`, gefolgt von `main`, einem runden Klammernpaar und einem geschweiften Klammernpaar, das eine Anweisungssequenz umschließt, zusammen. Beim Aufruf bzw. Start des Programms werden die Anweisungen der Anweisungssequenz innerhalb der geschweiften Klammern hintereinander ausgeführt.

2.2.4 Kommentare

Ziel der Programmierung ist es, Programme zu entwickeln, die gegebene Aufgaben lösen. Neben ihren Eigenschaften, korrekt und vollständig zu sein, sollten sich Programme durch eine weitere

Eigenschaft auszeichnen; sie sollten gut verständlich sein. Das bedeutet, die Lösungsidee und die Realisierung sollte auch von anderen Programmierern mühelos verstanden und nachvollzogen werden können, um bspw. das Programm später noch zu erweitern oder in anderen Zusammenhängen wiederverwenden zu können.

Diesem Zweck der Dokumentation eines Programms dienen so genannte *Kommentare*. Sie haben auf die Steuerung des Hamsters keinerlei Auswirkungen. Alles, was sie bewirken, ist eine bessere Lesbarkeit des Programms. In Java gibt es zwei Typen von Kommentaren: *Zeilenkommentare* und *Bereichskommentare*.

- Zeilenkommentare beginnen mit zwei Schrägstrichen `//` und enden am nächsten Zeilenende. Den Schrägstrichen können beliebige Zeichen folgen.
- Bereichskommentare beginnen mit der Zeichenkombination `/*` und enden mit der Zeichenkombination `*/`. Dazwischen können beliebige Zeichen stehen. Bereichskommentare können sich auch über mehrere Zeilen erstrecken.

2.2.5 Beispielprogramm

Das folgende Hamster-Programm bewirkt, dass der Hamster in dem in Abbildung 2.2 skizzierten Territorium zwei Körner frisst:

```
void main() {

    // friss erstes Korn
    vor();
    vor();
    nimm();

    // friss zweites Korn
    linksUm();
    vor();
    vor();
    nimm();
}
```

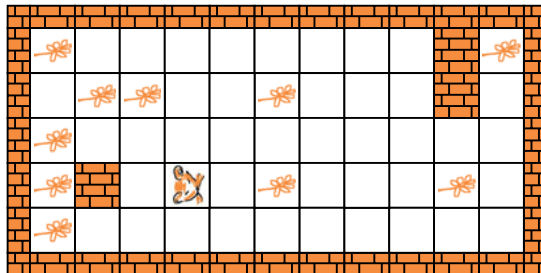


Abbildung 2.2: Hamster-Territorium zum Beispielprogramm

2.3 Prozeduren

Prozeduren dienen zur Vereinbarung neuer Befehle. Diesbezüglich sind zwei Aspekte zu betrachten: die Definition von Prozeduren und deren Aufruf, d.h. Ausführung.

2.3.1 Prozedurdefinition

Durch eine *Prozedurdefinition* wird ein neuer Befehl vereinbart. In der Definition muss zum einen angegeben werden, wie der Befehl heißt (*Prozedurname*), und zum anderen muss festgelegt werden, was der Hamster tun soll, wenn er den neuen Befehl erhält. Ersteres erfolgt im so genannten *Prozedurkopf*, letzteres im so genannten *Prozedurrumpf*.

Im Prozedurkopf muss zunächst das Schlüsselwort `void` angegeben werden. Anschließend folgt ein Bezeichner, der Prozedurname bzw. der Name des neuen Befehls. Nach dem Prozedurnamen folgt ein rundes Klammernpaar, das den Prozedurkopf beendet. Der Prozedurrumpf beginnt mit einer öffnenden geschweiften Klammer, der eine Anweisungssequenz folgt. Der Prozedurrumpf und damit die Prozedurdefinition endet mit einer schließenden geschweiften Klammer.

Auch das oben eingeführte `main`-Konstrukt ist im Prinzip eine Prozedur. Sie wird automatisch beim Start eines Programms durch das Laufzeitsystem aufgerufen. Die Definition einer neuen Prozedur darf vor oder nach der Definition der `main`-Prozedur erfolgen.

2.3.2 Prozeduraufruf

Durch eine Prozedurdefinition wird ein neuer Befehl eingeführt. Ein Aufruf des neuen Befehls wird *Prozeduraufruf* genannt. Ein Prozeduraufruf entspricht syntaktisch dem Aufruf eines der vier Grundbefehle des Hamsters. Er beginnt mit dem Prozedurnamen. Anschließend folgen eine öffnende und eine schließende runde Klammer und ein Semikolon.

Wird irgendwo in einem Programm eine Prozedur aufgerufen, so werden bei der Ausführung des Programms an dieser Stelle die Anweisung(en) des Prozedurrumpfes ausgeführt. Der Kontrollfluss des Programms verzweigt beim Prozeduraufruf in den Rumpf der Prozedur, führt die dortigen Anweisungen aus und kehrt nach der Abarbeitung der letzten Anweisung des Rumpfes an die Stelle des Prozeduraufrufs zurück. Durch Aufruf der *return-Anweisung* (`return;`) kann eine Prozedur vorzeitig verlassen werden.

2.3.3 Beispielprogramm

Im folgenden Hamster-Programm wird eine Prozedur `rechtsUm` definiert und aufgerufen, die bewirkt, dass sich der Hamster dreimal nach links dreht, was einer Rechtsdrehung um 90 Grad entspricht. Der Hamster frisst in dem in Abbildung 2.3 skizzierten Territorium zwei Körner:

```
void rechtsUm() { // Prozedurdefinition
    linksUm();
    linksUm();
    linksUm();
}
```

```

void main() {           // main-Prozedur

    // friss erstes Korn
    rechtsUm();          // Prozeduraufruf
    vor();
    vor();
    nimm();

    // friss zweites Korn
    rechtsUm();          // Prozeduraufruf
    vor();
    vor();
    nimm();
}

```

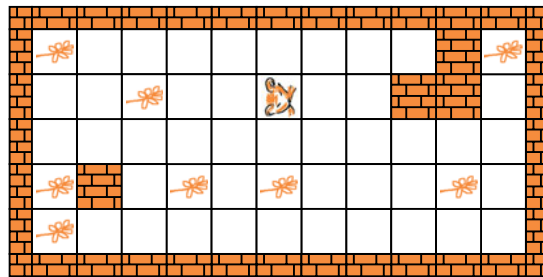


Abbildung 2.3: Hamster-Territorium zum Beispielprogramm

2.4 Auswahlanweisungen

Auswahlanweisungen dienen dazu, bestimmte Anweisungen nur unter bestimmten Bedingungen ausführen zu lassen.

2.4.1 Testbefehle

Um Laufzeitfehler zu vermeiden, die ja bspw. dadurch entstehen können, dass Sie dem Hamster den Befehl vor geben, obwohl er vor einer Mauer steht, existieren drei so genannte *Testbefehle*. Testbefehle liefern boolesche Werte, also wahr (*true*) oder falsch (*false*):

- *vornFrei()*: Liefert den Wert *true*, falls sich auf der Kachel in Blickrichtung vor dem Hamster keine Mauer befindet. Ist die Kachel durch eine Mauer blockiert, dann wird der Wert *false* geliefert.
- *maulLeer()*: Liefert den Wert *false*, falls der Hamster ein oder mehrere Körner im Maul hat. Befinden sich keine Körner im Maul des Hamsters, dann wird der Wert *true* geliefert.
- *kornDa()*: Liefert den Wert *true*, falls auf der Kachel, auf der der Hamster gerade steht, ein oder mehrere Körner liegen. Befindet sich kein Korn auf der Kachel, dann wird der Wert *false* geliefert.

2.4.2 Boolesche Operatoren und Ausdrücke

Die drei Testbefehle stellen einfache *boolesche Ausdrücke* dar. *Ausdrücke* sind Verarbeitungsvorschriften, die einen Wert berechnen und liefern. Boolesche Ausdrücke liefern einen booleschen Wert. Durch die Verknüpfung boolescher Ausdrücke mittels boolescher Operatoren lassen sich zusammengesetzte boolesche Ausdrücke bilden. Die Programmiersprache Java stellt drei boolesche Operatoren zur Verfügung:

- Der Operator `!` (Negation) negiert den Wahrheitswert seines Operanden, d.h. er dreht ihn um. Liefert ein boolescher Ausdruck `bA` den Wert `true`, dann liefert der boolesche Ausdruck `!bA` den Wert `false`. Umgekehrt gilt, liefert `bA` den Wert `false`, dann liefert `!bA` den Wert `true`.
- Der Operator `&&` (Konjunktion) konjugiert die Wahrheitswerte seiner beiden Operanden, d.h. er liefert genau dann den Wahrheitswert `true`, wenn beide Operanden den Wert `true` liefern.
- Der Operator `||` (Disjunktion) disjüngiert die Wahrheitswerte seiner beiden Operanden, d.h. er liefert genau dann den Wahrheitswert `true`, wenn einer seiner Operanden oder seine beiden Operanden den Wert `true` liefern.

Der Operator `!` hat eine höhere Priorität als der Operator `&&`, der wiederum eine höhere Priorität als der Operator `||` besitzt. Durch Einschließen von booleschen Ausdrücken in runde Klammern können Sie die Abarbeitungsfolge der Operatoren beeinflussen.

2.4.3 Blockanweisung

Mit Hilfe der *Blockanweisung* lassen sich mehrere Anweisungen zu einer Einheit zusammenfassen. Syntaktisch gesehen handelt es sich bei einer Blockanweisung um eine zusammengesetzte Anweisung. Innerhalb von geschweiften Klammern steht eine andere Anweisung – im Allgemeinen eine Anweisungssequenz.

Beim Ausführen einer Blockanweisung werden die innerhalb der geschweiften Klammern stehenden Anweisungen ausgeführt.

2.4.4 Bedingte Anweisung

Die *bedingte Anweisung*, die auch *if-Anweisung* genannt wird, ist eine zusammengesetzte Anweisung. Die bedingte Anweisung wird eingeleitet durch das Schlüsselwort `if`. Anschließend folgt innerhalb eines runden Klammernpaares ein boolescher Ausdruck und danach eine Anweisung. Bei dieser Anweisung, die im Folgenden *true-Anweisung* genannt wird, handelt es sich im Allgemeinen um eine Blockanweisung.

Beim Ausführen einer bedingten Anweisung wird zunächst der boolesche Ausdruck innerhalb der runden Klammern ausgewertet. Falls dieser Ausdruck den Wert `true` liefert, d.h. die Bedingung erfüllt ist, wird die *true-Anweisung* (daher der Name) ausgeführt. Liefert der boolesche Ausdruck den Wert `false`, dann wird die *true-Anweisung* nicht ausgeführt.

2.4.5 Alternativanweisung

Die *Alternativanweisung* ist eine bedingte Anweisung mit einem angehängten so genannten *else-Teil*. Dieser besteht aus dem Schlüsselwort `else` und einer Anweisung (*false-Anweisung*) – im

Allgemeinen eine Blockanweisung. Die Alternativanweisung ist wie die bedingte Anweisung eine Auswahlanweisung.

Wird eine Alternativanweisung ausgeführt, dann wird zunächst der Wert der Bedingung (boolescher Ausdruck) ermittelt. Ist die Bedingung erfüllt, d.h. liefert der boolesche Ausdruck den Wert `true`, dann wird die `true`-Anweisung nicht aber die `false`-Anweisung ausgeführt. Liefert der boolesche Ausdruck den Wert `false`, dann wird die `false`-Anweisung nicht aber die `true`-Anweisung ausgeführt.

2.4.6 Beispielprogramm

Im folgenden Beispielprogramm führt der Hamster die Blockanweisung mit dem `vor`- und dem `linksUm`-Befehl nur aus, wenn sich vor ihm keine Mauer befindet. Anschließend überprüft er, ob sich auf der aktuellen Kachel ein Korn befindet und gleichzeitig die Kachel vor ihm frei ist. Dann und nur dann nimmt er das Korn und springt eine Kachel nach vorne. Im anderen Fall dreht er sich um 90 Grad nach links.

```
void main() {  
    if (vornFrei()) { // bedingte Anweisung  
        vor();  
        linksUm();  
    }  
  
    if (kornDa() && vornFrei()) { // Alternativanweisung  
        nimm();  
        vor();  
    } else {  
        linksUm();  
    }  
}
```

2.5 Wiederholungsanweisungen

Wiederholungsanweisungen – auch *Schleifenanweisungen* genannt – dienen dazu, bestimmte Anweisungen mehrmals ausführen zu lassen, solange eine bestimmte Bedingung erfüllt wird.

2.5.1 while-Anweisung

Die *while-Anweisung* ist eine zusammengesetzte Anweisung. Nach dem Schlüsselwort `while` steht in runden Klammern ein boolescher Ausdruck, die so genannte *Schleifenbedingung*. Anschließend folgt die Anweisung, die eventuell mehrfach ausgeführt werden soll. Sie wird auch *Iterationsanweisung* genannt. Hierbei handelt es sich im Allgemeinen um eine Blockanweisung.

Bei der Ausführung einer *while-Anweisung* wird zunächst überprüft, ob die Schleifenbedingung erfüllt ist, d.h. ob der boolesche Ausdruck den Wert `true` liefert. Falls dies nicht der Fall ist, ist die *while-Anweisung* unmittelbar beendet. Falls die Bedingung erfüllt ist, wird die Iterationsanweisung einmal ausgeführt. Anschließend wird die Schleifenbedingung erneut ausgewertet. Falls sie immer

noch erfüllt ist, wird die Iterationsanweisung ein weiteres Mal ausgeführt. Dieser Prozess (Überprüfung der Schleifenbedingung und falls diese erfüllt ist, Ausführung der Iterationsanweisung) wiederholt sich so lange, bis (hoffentlich) irgendwann einmal die Bedingung nicht mehr erfüllt ist.

2.5.2 do-Anweisung

Bei Ausführung der `while`-Anweisung kann es vorkommen, dass die Iterationsanweisung kein einziges Mal ausgeführt wird; nämlich genau dann, wenn die Schleifenbedingung direkt beim ersten Test nicht erfüllt ist. Für solche Fälle, bei denen die Iterationsanweisung auf jeden Fall mindestens einmal ausgeführt werden soll, existiert die *do-Anweisung* – auch *do-Schleife* genannt.

Dem Schlüsselwort `do`, von dem die Anweisung ihren Namen hat, folgt die Iterationsanweisung. Hinter der Iterationsanweisung muss das Schlüsselwort `while` stehen. Anschließend folgt in runden Klammern ein boolescher Ausdruck – die Schleifenbedingung. Abgeschlossen wird die `do`-Anweisung durch ein Semikolon.

Bei der Ausführung einer `do`-Anweisung wird zunächst einmal die Iterationsanweisung ausgeführt. Anschließend wird die Schleifenbedingung überprüft. Ist sie nicht erfüllt, d.h. liefert der boolesche Ausdruck den Wert `false`, dann endet die `do`-Anweisung. Ist die Bedingung erfüllt, wird die Iterationsanweisung ein zweites Mal ausgeführt und danach erneut die Schleifenbedingung ausgewertet. Dieser Prozess wiederholt sich so lange, bis irgendwann einmal die Schleifenbedingung nicht mehr erfüllt ist.

2.5.3 Beispielprogramm

Der Hamster steht – wie in den Beispielen in Abbildung 2.4 skizziert – vor einem regelmäßigen Berg unbekannter Höhe. Er soll den Gipfel erklimmen und dort anhalten.

```
void main() {
    laufeZumBerg();
    erklimmeGipfel();
}

void laufeZumBerg() {
    while (vornFrei()) { // while-Schleife
        vor();
    }
}

void erklimmeGipfel() {
    do { // do-Schleife
        erklimmeEineStufe();
    } while (!vornFrei());
}

void erklimmeEineStufe() {
    linksUm();
    vor();
    rechtsUm();
    vor();
}
```

```

}

void rechtsUm() {
    linksUm();
    linksUm();
    linksUm();
}

```

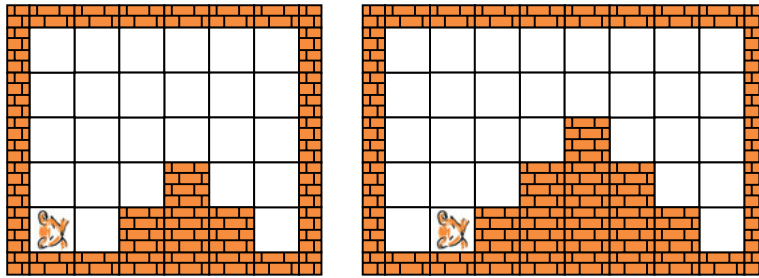


Abbildung 2.4: Hamster-Territorien zum Beispielprogramm

2.6 Boolesche Funktionen

Während Prozeduren dazu dienen, den Befehlsvorrat des Hamsters zu erweitern, dienen *boolesche Funktionen* dazu, neue Testbefehle einzuführen.

2.6.1 Boolesche return-Anweisung

Boolesche return-Anweisungen werden in booleschen Funktionen zum Liefern eines booleschen Wertes benötigt.

Die Syntax der booleschen *return*-Anweisung ist sehr einfach: Dem Schlüsselwort *return* folgt ein boolescher Ausdruck und ein abschließendes Semikolon. Boolesche *return*-Anweisungen sind spezielle Anweisungen, die ausschließlich im Funktionsrumpf boolescher Funktionen verwendet werden dürfen.

Die Ausführung einer booleschen *return*-Anweisung während der Ausführung einer booleschen Funktion führt zur unmittelbaren Beendigung der Funktionsausführung. Dabei wird der Wert des booleschen Ausdrucks als so genannter *Funktionswert* zurückgegeben.

2.6.2 Definition boolescher Funktionen

Die Syntax der Definition einer booleschen Funktion unterscheidet sich nur geringfügig von der Definition einer Prozedur. Statt Prozedurkopf, -name und -rumpf spricht man hier von *Funktionskopf*, *Funktionsname* und *Funktionsrumpf*.

Anstelle des Schlüsselwortes *void* bei der Definition einer Prozedur muss bei der Definition einer booleschen Funktion das Schlüsselwort *boolean* am Anfang des Funktionskopfes stehen.

Ganz wichtig bei der Definition boolescher Funktionen ist jedoch folgende Zusatzbedingung: In jedem möglichen Weg durch die Funktion muss eine boolesche `return`-Anweisung auftreten!

Boolesche Funktionen können überall dort in einem Hamster-Programm definiert werden, wo auch Prozeduren definiert werden können.

2.6.3 Aufruf boolescher Funktionen

Eine boolesche Funktion darf überall dort aufgerufen werden, wo auch einer der drei vordefinierten Testbefehle aufgerufen werden darf. Der Aufruf einer booleschen Funktion gilt also als ein spezieller boolescher Ausdruck. Der Funktionsaufruf erfolgt syntaktisch durch die Angabe des Funktionsnamens gefolgt von einem runden Klammernpaar.

Wird bei der Berechnung eines booleschen Ausdrucks eine boolesche Funktion aufgerufen, so wird in deren Funktionsrumpf verzweigt und es werden die dortigen Anweisungen aufgerufen. Wird dabei eine boolesche `return`-Anweisung ausgeführt, so wird der Funktionsrumpf unmittelbar verlassen und an die Stelle des Funktionsaufrufs zurückgesprungen. Der von der booleschen `return`-Anweisung gelieferte Wert (also der Funktionswert) wird dabei zur Berechnung des booleschen Ausdrucks weiterverwendet.

2.6.4 Beispielprogramm

Im folgenden Programm sucht der Hamster auf der rechten Seite eine Nische. Findet er eine, begibt er sich hinein. Beim Suchen benutzt er eine boolesche Funktion `rechtsFrei`.

```
void kehrt() {
    linksUm();
    linksUm();
}

void rechtsUm() {
    kehrt();
    linksUm();
}

boolean rechtsFrei() { // Definition einer booleschen Funktion
    rechtsUm();
    if (vornFrei()) {
        linksUm();
        return true; // boolesche return-Anweisung
    } else {
        linksUm();
        return false; // boolesche return-Anweisung
    }
}

void main() {
    while (vornFrei() &&
           !rechtsFrei()) { // Aufruf einer booleschen Funktion
        vor();
    }
}
```

```
    }  
    if (rechtsFrei()) { // Aufruf einer booleschen Funktion  
        rechtsUm();  
        vor();  
    }  
}
```

2.7 Variablen und Ausdrücke

Durch die Einführung von Variablen und Ausdrücken bekommt der Hamster ein „Gedächtnis“ und lernt rechnen.

2.7.1 Datentypen

Ein *Datentyp* repräsentiert Werte eines bestimmten Typs. Im imperativen Hamster-Modell werden die Datentypen `boolean` und `int` genutzt. Der Datentyp `boolean` repräsentiert boolesche Werte, also `true` und `false`. Der Datentyp `int` repräsentiert ganze Zahlen zwischen -2^{31} und $2^{31} - 1$.

2.7.2 Variablen

Variablen sind Speicherbereiche („Behälter“), in denen Werte abgespeichert werden können. Vor ihrer Benutzung müssen sie definiert werden. Bei der Definition einer Variablen wird ihr ein Name – ein beliebiger Bezeichner – zugeordnet. Außerdem wird durch die Angabe eines Datentyps festgelegt, welche Werte die Variable speichern kann.

Die folgenden Anweisungen definieren eine Variable `frei` zum Speichern von booleschen Werten sowie eine Variable `anzahlKoerner` zum Speichern von ganzen Zahlen. Der Variablen `frei` wird der Initialwert `false` zugewiesen, der Variablen `anzahl` der Wert 13.

```
boolean frei = false;  
int anzahl = 13;
```

2.7.3 Ausdrücke

Ausdrücke sind spezielle Programmierkonstrukte zum Berechnen und Liefern eines Wertes. Boolesche Ausdrücke haben Sie bereits kennen gelernt. Sie liefern boolesche Werte. Zur Berechnung boolescher Ausdrücke können auch `boolean`-Variablen hinzugezogen werden. Enthält ein boolescher Ausdruck den Namen einer booleschen Variablen, dann wird bei der Auswertung des booleschen Ausdrucks an der entsprechenden Stelle der Wert berücksichtigt, der aktuell in der Variablen gespeichert ist.

Einen weiteren Typ von Ausdrücken stellen *arithmetische Ausdrücke* dar. Sie berechnen und liefern Werte vom Typ `int`, also ganze Zahlen. Arithmetische Ausdrücke lassen sich auf folgende Art und Weise bilden:

- **int-Literale:** int-Literale werden durch Zeichenfolgen beschrieben, die aus dezimalen Ziffern (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) bestehen. Dabei gilt die Einschränkung, dass einer Zahl ungleich 0 keine „0“ vorangestellt werden darf. Gültige int-Literale sind also: 0, 2, 4711, 1234560789, ...
- **int-Variablenname:** Der Name einer int-Variablen in einem arithmetischen Ausdruck repräsentiert den aktuell in der Variablen gespeicherten Wert.
- **Unäre arithmetische Operatoren:** Die Zeichen „+“ und „-“ kennzeichnen Vorzeichen von arithmetischen Ausdrücken. Die unären arithmetischen Operatoren sind rechtsassoziativ und besitzen die höchste Priorität aller arithmetischen Operatoren.
- **Binäre arithmetische Operatoren:** Es existieren insgesamt fünf binäre arithmetische Operatoren, mit denen jeweils zwei andere arithmetische Ausdrücke (die Operanden) verknüpft werden:
 - „+“: liefert als Wert die Summe seiner beiden Operanden (Addition)
 - „-“: liefert als Wert die Differenz seiner beiden Operanden (Subtraktion)
 - „*“: liefert als Wert das Produkt seiner beiden Operanden (Produkt)
 - „/“: liefert als Wert den Quotient seiner beiden Operanden; dabei werden entstehende Nachkommastellen ignoriert (*ganzzahlige Division*); z.B. $7/3 = 2$
 - „%“: liefert als Wert den Rest einer ganzzahligen Division (*Modulo-Operator*); z.B. $7\%3 = 1$. Zwischen der Ganzzahldivision und der Restbildung besteht folgende Beziehung: Seien x und y arithmetische Ausdrücke, dann gilt $((x/y) * y) + (x\%y) = x$.

Die binären arithmetischen Operatoren sind linksassoziativ. Die Operatoren „*“, „/“ und „%“ besitzen eine höhere Priorität als die Operatoren „+“ und „-“ („Punkt-vor-Strich-Rechnung“).
- **Klammern:** Zum Bilden von (komplexen) arithmetischen Ausdrücken können arithmetische Ausdrücke in Paare von runden Klammern gesetzt werden. Dadurch lässt sich die Priorität von arithmetischen Operatoren beeinflussen.

2.7.4 Zuweisung

Mit Hilfe einer *Zuweisungsanweisung* – kurz auch *Zuweisung* genannt – können Variablen neue Werte zugewiesen werden. Die alten Werte gehen dabei verloren.

Syntaktisch wird die Zuweisung so gebildet, dass dem Namen der betroffenen Variablen das Zeichen `=` – der *Zuweisungsoperator* – und anschließend ein Ausdruck folgt. Bei booleschen Variablen muss das ein boolescher Ausdruck sein, bei int-Variablen ein arithmetischer Ausdruck.

Bei der Ausführung einer Zuweisung wird zunächst der Ausdruck ausgewertet und anschließend der berechnete Wert der Variablen auf der linken Seite des Zuweisungsoperators zugewiesen. Die Zuweisungsanweisung `zahl = zahl + 1`; erhöht bspw. den Wert einer int-Variablen `zahl` um den Wert 1.

2.7.5 Vergleichsausdrücke

Vergleichsausdrücke sind boolesche Ausdrücke, die zum Vergleichen arithmetischer Ausdrücke dienen. Sie liefern boolesche Werte nach folgenden Gesetzmäßigkeiten: Seien x und y zwei arithmetische Ausdrücke, dann gilt:

- $x == y$ liefert genau dann den Wert `true`, falls der Wert, den x liefert, gleich dem Wert ist, den y liefert (Gleichheitsoperator).
- $x != y$ liefert genau dann den Wert `true`, falls der Wert, den x liefert, ungleich dem Wert ist, den y liefert (Ungleichheitsoperator).
- $x < y$ liefert genau dann den Wert `true`, falls der Wert, den x liefert, kleiner ist als der Wert, den y liefert (Kleineroperator).
- $x <= y$ liefert genau dann den Wert `true`, falls der Wert, den x liefert, kleiner oder gleich dem Wert ist, den y liefert (Kleinerleichoperator).
- $x > y$ liefert genau dann den Wert `true`, falls der Wert, den x liefert, größer ist als der Wert, den y liefert (Größeroperator).
- $x >= y$ liefert genau dann den Wert `true`, falls der Wert, den x liefert, größer oder gleich dem Wert ist, den y liefert (Größergleichoperator).

Die Vergleichsoperatoren sind linksassoziativ. Die Operatoren $<$, $<=$, $>$ und $>=$ haben eine höhere Priorität als die Operatoren $==$ und $!=$. Weiterhin ist zu beachten, dass die Vergleichsoperatoren eine niedrigere Priorität besitzen als die arithmetischen Operatoren und eine höhere Priorität als der Zuweisungsoperator.

2.7.6 Gültigkeitsbereiche von Variablen

Variablen lassen sich innerhalb von Prozeduren und Funktionen definieren. In diesem Fall nennt man sie *lokale* Variablen. Variablen, die außerhalb von Prozeduren oder Funktionen definiert werden, heißen *globale* Variablen.

Als *Gültigkeitsbereich* einer Variablen wird der Teil eines Programmes bezeichnet, in dem eine Variable genutzt werden kann, d.h. in dem der Name einer Variablen verwendet werden darf. Der Gültigkeitsbereich einer lokalen Variablen erstreckt sich von der Variablendefinition folgenden Anweisung bis zum Ende desselben Blockes und umschließt alle inneren Blöcke. Der Gültigkeitsbereich einer globalen Variablen umfasst das gesamte Hamster-Programm. Im Gültigkeitsbereich einer Variablen darf keine weitere Variable mit demselben Namen definiert werden.

2.7.7 Lebensdauer von Variablen

Während der Gültigkeitsbereich einer booleschen Variablen zur Compilierzeit von Bedeutung ist, ist die *Lebensdauer* einer booleschen Variablen eine Größe, die zur Laufzeit Relevanz besitzt. Sie ist definiert als die Zeitspanne, während der im Hauptspeicher Speicherplatz für eine Variable reserviert ist. Es gilt dabei: Die Lebensdauer einer globalen Variablen umfasst die gesamte Ausführungszeit eines Hamster-Programms. Die Lebensdauer einer lokalen Variablen beginnt bei ihrer Definition und endet nach der vollständigen Abarbeitung des Blocks, in dem sie definiert wurde.

2.7.8 Beispielprogramm

Im folgenden Programm läuft der Hamster bis zur nächsten Wand. Anschließend dreht er sich um und läuft zum Ausgangsfeld zurück. Die Anzahl der gelaufenen Schritte merkt er sich dabei in einer globalen Variablen `anzahl`.


```
int anzahl = 0; // globale Variable

void laufeBisZurWand() {
    while (vornFrei()) {
        vor();
        anzahl = anzahl + 1; // Zuweisung
    }
}

void laufeZurueckZumAusgangspunkt() {
    while (anzahl > 0) { // Vergleichsausdruck
        vor();
        anzahl = anzahl - 1;
    }
}

void main() {
    laufeBisZurWand();
    linksUm();
    linksUm();
    laufeZurueckZumAusgangspunkt();
}
```

2.8 int-Funktionen

Boolesche Funktionen liefern einen Wert vom Typ `boolean`. Dementsprechend sind *int-Funktionen* Funktionen, die einen Wert vom Typ `int`, also eine ganze Zahl, liefern.

2.8.1 Funktionsdefinition

Die Definition einer `int`-Funktion unterscheidet sich nur dadurch von der Definition einer booleschen Funktion, dass im Funktionskopf das Schlüsselwort `boolean` gegen das Schlüsselwort `int` ausgetauscht werden muss und die im Funktionsrumpf obligatorischen `return`-Anweisungen anstelle eines booleschen Wertes einen Wert vom Typ `int` liefern müssen.

2.8.2 Funktionsaufruf

Der Aufruf von `int`-Funktionen entspricht einem speziellen arithmetischen Ausdruck. `int`-Funktionen dürfen also überall dort aufgerufen werden, wo arithmetische Ausdrücke stehen dürfen. Der Aufruf einer `int`-Funktion erfolgt wie der Aufruf einer booleschen Funktion syntaktisch durch die Angabe des Funktionsnamens gefolgt von einem runden Klammernpaar. Beim Aufruf einer `int`-Funktion wird in den Funktionsrumpf verzweigt und die dortigen Anweisungen werden ausgeführt. Als Wert des arithmetischen Ausdrucks wird in diesem Fall der Wert genommen, den die Funktion berechnet und mit Hilfe einer `return`-Anweisung geliefert hat.

2.8.3 Verallgemeinerung des Funktionskonzeptes

Neben den Datentypen `boolean` und `int` gibt es in Java und in anderen Programmiersprachen weitere Datentypen, auf deren Einführung im Hamster-Modell zunächst verzichtet wird, für die das Funktionskonzept aber entsprechend gilt.

Funktionen können prinzipiell wie Prozeduren auch in Form von Anweisungen aufgerufen werden. Der gelieferte Funktionswert wird dann einfach ignoriert.

Prozeduren werden ab jetzt als Spezialform von Funktionen aufgefasst. Sie liefern keinen Wert, was durch das Schlüsselwort `void` bei ihrer Definition ausgedrückt wird.

Der Gültigkeitsbereich von Funktionen erstreckt sich über ein gesamtes Programm. Insbesondere dürfen Funktionen auch schon vor ihrer Definition aufgerufen werden.

Wenn sich nach der Ausführung einer Funktion der Programmzustand verändert hat (bspw. die Position oder Blickrichtung des Hamsters), spricht man von einem *Seiteneffekt*, den die Funktion produziert hat.

2.8.4 Beispielprogramm

Der Hamster bekommt die Aufgabe, so viele Körner abzulegen, wie insgesamt auf seinen vier Nachbarfeldern liegen (natürlich nur, wenn er auch genügend Körner im Maul hat). Er nutzt dabei eine Funktion `koernerVorne()`, die ohne Seiteneffekte die Anzahl an Körnern liefert, die sich auf der Kachel vor dem Hamster befindet.

```
// liefert die Anzahl an Koernern, die sich auf dem Feld vor dem
// Hamster befindet; produziert dabei keine Seiteneffekte
int koernerVorne() { // Definition einer int-Funktion
    if (!vornFrei()) {
        return 0;
    }

    vor();
    int anzahl = 0; // lokale Variable
    while (kornDa()) {
        nimm();
        anzahl = anzahl + 1;
    }

    // um Seiteneffekte zu vermeiden, muessen nun noch die Koerner
    // wieder abgelegt werden und der Hamster muss auf
    // seine alte Kachel zurueckkehren
    int zurueck = anzahl;
    while (zurueck > 0) {
        gib();
        zurueck = zurueck - 1;
    }
    kehrt();
    vor();
    kehrt();
}
```

```

    // nun kann die Anzahl an Koernern zurueckgeliefert werden
    return anzahl;
}

void kehrt() {
    linksUm();
    linksUm();
}

void main() {

    // ermittelt die Anzahl an Koernern vor sich
    int nachbarKoerner = koernerVorne(); // Aufruf einer int-Funktion
    int drehungen = 0;

    // dreht sich in die anderen Richtungen und addiert die
    // entsprechenden Koerneranzahlen
    while (drehungen < 3) {
        linksUm();
        nachbarKoerner = nachbarKoerner +
            koernerVorne(); // Aufruf einer int-Funktion
        drehungen = drehungen + 1;
    }

    // legt entsprechend viele Koerner ab, solange er noch
    // welche im Maul hat
    while (nachbarKoerner > 0 && !maulLeer()) {
        gib();
        nachbarKoerner = nachbarKoerner - 1;
    }
}

```

2.9 Funktionsparameter

Das Parameterkonzept erhöht die Flexibilität von Prozeduren und Funktionen.

2.9.1 Formale Parameter

Parameter sind lokale Variablen von Funktionen, die dadurch initialisiert werden, dass der Funktion bei ihrem Aufruf ein entsprechender Initialisierungswert für die Variable übergeben wird.

Parameter werden im Funktionskopf definiert. Zwischen die beiden runden Klammern wird eine so genannte *formale Parameterliste* eingeschoben. Die Parameterliste besteht aus keiner, einer oder mehreren durch Kommata getrennten Parameterdefinitionen. Eine Parameterdefinition hat dabei eine ähnliche Gestalt wie die Variablendefinition. Was fehlt, ist ein expliziter Initialisierungsausdruck. In der Tat handelt es sich bei einem Parameter auch um eine ganz normale Variable. Sie ist lokal bezüglich des Funktionsrumpfes. Ihr können im Funktionsrumpf ihrem Typ entsprechend Werte zugewiesen werden und sie kann bei der Bildung von typkonformen Ausdrücken innerhalb des Funk-

tionsrumpfes eingesetzt werden. Man nennt die Parameter innerhalb der Funktionsdefinition auch *formale Parameter* oder *Parametervariablen*.

2.9.2 Aktuelle Parameter

Der Funktionsaufruf wird durch die Angabe einer *aktuellen Parameterliste* zwischen den runden Klammern erweitert. Die durch Kommata getrennten Elemente dieser Liste werden als *aktuelle Parameter* bezeichnet. Hierbei handelt es sich um Ausdrücke.

2.9.3 Parameterübergabe

Bezüglich der Definition von Funktionen mit (formalen) Parametern und dem Aufruf von Funktionen mit (aktuellen) Parametern sind folgende zusätzliche Bedingungen zu beachten:

- Die Anzahl der aktuellen Parameter beim Aufruf einer Funktion muss gleich der Anzahl der formalen Parameter der Funktionsdefinition sein. (Ausnahme: so genannte *varargs*-Parameter)
- Für alle Parameter in der angegebenen Reihenfolge muss gelten: Der Typ eines aktuellen Parameters muss konform sein zum Typ des entsprechenden formalen Parameters (boolesche Ausdrücke sind konform zum Typ `boolean`, arithmetische Ausdrücke sind konform zum Typ `int`).

Wird eine Funktion mit Parametern aufgerufen, so passiert folgendes: Die aktuellen Parameter – hierbei handelt es sich ja um Ausdrücke – werden berechnet, und zwar immer von links nach rechts, falls es sich um mehr als einen Parameter handelt. Für jeden formalen Parameter der formalen Parameterliste wird im Funktionsrumpf eine lokale Variable angelegt. Diese Variablen werden anschließend – bei Beachtung der Reihenfolge innerhalb der Parameterlisten – mit dem Wert des entsprechenden aktuellen Parameters initialisiert. Man spricht in diesem Zusammenhang auch von *Parameterübergabe*: Der Wert eines aktuellen Parameters wird beim Aufruf einer Funktion einem formalen Parameter der Funktion als Initialisierungswert übergeben.

2.9.4 Überladen von Funktionen

Eigentlich müssen Funktionsnamen in einem Programm eindeutig sein. Zwei oder mehrere Funktionen dürfen jedoch denselben Namen besitzen, falls sich ihre formalen Parameterlisten durch die Anzahl an Parametern oder die Typen der Parameter unterscheiden. Man nennt dieses Prinzip auch *Überladen* von Funktionen. Die tatsächlich aufgerufene Funktion wird dann beim Funktionsaufruf anhand der Anzahl bzw. Typen der aktuellen Parameterliste bestimmt.

2.9.5 Beispielprogramm

Im folgenden Beispielprogramm wird eine Funktion `vor` mit einem formalen Parameter `anzahl` vom Typ `int` definiert. Die Funktion überlädt dabei die Funktion, die den Hamster-Befehl `vor` repräsentiert. Der Wert von `anzahl` gibt an, um wie viele Kacheln der Hamster nach vorne springen soll. Der Wert braucht dabei erst zur Laufzeit beim Aufruf der Funktion angegeben werden. In der `main`-Funktion wird die Funktion `vor` einmal mit dem Wert 4 aufgerufen, d.h. der Hamster soll vier

Kacheln nach vorne springen. Beim zweiten Aufruf wird als Wert die Anzahl an Körnern, die sich im Maul des Hamsters befinden, übergeben. Entsprechend oft hüpfet der Hamster nach vorne.

```
// der Hamster huepft anzahl-mal vor, maximal aber bis zur Mauer
void vor(int anzahl) { // formaler Parameter
    while (vornFrei() && anzahl > 0) {
        vor();
        anzahl = anzahl - 1;
    }
}

// liefert seiteneffektfrei die Koerneranzahl im Maul des Hamsters
int koernerImMaul() {
    int anzahl = 0;
    while (!maulLeer()) {
        gib();
        anzahl = anzahl + 1;
    }
    int ergebnis = anzahl;

    // Koerner wieder fressen
    while (anzahl > 0) {
        nimm();
        anzahl = anzahl - 1;
    }
    return ergebnis;
}

void main() {
    vor(4); // aktueller Parameter
    linksUm();
    vor(koernerImMaul()); // aktueller Parameter
}
```

2.10 Rekursion

Funktionen, die sich selbst aufrufen, bezeichnet man als *rekursive Funktionen*. Die rekursive Funktion `hinUndZurueck` im folgenden Programm bewirkt, dass der Hamster bis zur nächsten Wand und anschließend zurück zu seinem Ausgangspunkt läuft.

```
void hinUndZurueck() {
    if (vornFrei()) {
        vor();
        hinUndZurueck(); // rekursiver Funktionsaufruf
        vor();
    } else {
        kehrt();
    }
}
```

```
void kehrt() {  
    linksUm();  
    linksUm();  
}  
  
void main() {  
  
    // der Hamster laeuft zur naechsten Wand und zurueck  
    hinUndZurueck();  
}
```

Objektorientierte Programmierung spielend gelernt mit
dem Java-Hamster-Modell

Boles, D.; Boles, C.

2014, XII, 523 S. 92 Abb., Softcover

ISBN: 978-3-658-04802-0