

Dietrich Boles

Das Hamster-Modell

Programmieren spielend gelernt



Vorwort

Programmieranfänger leiden häufig darunter, daß sie beim Programmieren ihre normale Gedankenwelt verlassen und in eher technisch-orientierten Kategorien denken müssen, die ihnen von den Programmiersprachen vorgegeben werden. Gerade am Anfang strömen häufig so viele Neuigkeiten inhaltlicher und methodischer Art auf sie ein, daß sie leicht das Wesentliche der Programmierung, nämlich das Lösen von Problemen, aus den Augen verlieren und sich in syntaktischen und technischen Einzelheiten verirren. Der „Kampf“ mit dem Compiler bekommt somit höhere Priorität als der Programmentwurf an sich und kann frühzeitig zur Frustration führen.

Das Hamster-Modell ist mit dem Ziel entwickelt worden, dieses Problem zu lösen. Mit dem Hamster-Modell wird dem Programmieranfänger ein einfaches aber mächtiges Modell zur Verfügung gestellt, mit dessen Hilfe er Grundkonzepte der Programmierung auf spielerische Art und Weise erlernen kann. Der Programmierer steuert einen virtuellen Hamster durch eine virtuelle Landschaft und läßt ihn bestimmte Aufgaben lösen. Die Anzahl der gleichzeitig zu berücksichtigenden Einzelheiten wird im Hamster-Modell stark eingeschränkt und nach und nach erweitert.

Mit einer ähnlichen Motivation wurde in den 70er und 80er Jahren die Schildkröten-Graphik der Programmiersprache LOGO entwickelt bzw. erprobt [Ros83, Men85]. Problem der Sprache LOGO ist allerdings, daß sie sich – wenn überhaupt – nur im Ausbildungssektor nicht aber beispielsweise im industriellen Bereich durchsetzen konnte. Dem „Mutterspracheneffekt“ kommt jedoch auch beim Programmieranfänger eine wichtige Bedeutung zu: Die Muttersprache beherrscht man wesentlich besser als jede später erlernte Sprache. Aus diesem Grund wurde für das Hamster-Modell keine neue Programmiersprache entwickelt. Vielmehr wurde das Modell in die Konzepte und die Syntax der Programmiersprache Java [AG96] eingebettet. Die Sprache Java, die auch als Sprache des Internet bezeichnet wird, ist eine (relativ) einfache Sprache, die viele wichtige Programmierkonzepte enthält und sich – insbesondere im Zusammenhang mit dem rapiden Wachstum des Internet – auch im industriellen Bereich immer mehr durchzusetzen scheint.

Das Hamster-Modell wurde in einer einfachen Version zu Beginn der 80er Jahre in der GMD entwickelt [Opp83]. Zielsprache war damals die imperative Programmiersprache ELAN [KL83, KL85]. Vorlage für das Hamster-Modell war dabei „Karel der Roboter“ [PRS95]. Ich habe das imperative Hamster-Modell an die Programmiersprache Java angepaßt und um Konzepte der objektorientierten und parallelen Programmierung erweitert. Dabei werden nicht der gesamte Sprachschatz der Programmiersprache Java sondern lediglich die grundlegenden Konstrukte behandelt.

Endziel meiner Arbeiten ist ein Buch, daß in vier Teile gegliedert ist. Der erste Teil gibt eine allgemeine Einführung in die Grundlagen der Programmierung. Im zweiten Teil werden die

Konzepte der imperativen Programmierung vorgestellt. Im einzelnen werden hier Anweisungen und Programme, Prozeduren, Kontrollstrukturen, der Top-Down-Programmwurf, Variablen und Ausdrücke, Funktionen und Parameter sowie das Prinzip der Rekursion behandelt. Auf den imperativen Programmierkonzepten aufbauend wird im dritten Teil in die objektorientierte Programmierung eingeführt. Konzepte, die hier erläutert werden, sind Objekte, Klassen, Arrays, Vererbungsmechanismen, Polymorphismus und dynamisches Binden sowie Zugriffsrechte. Im dritten Teil werden die Grundlagen gelegt für die Vorstellung paralleler Programmierkonzepte im vierten Teil des Buches. In diesem Teil werden Prozesse bzw. Threads, die Inter-Prozeß-Kommunikation, die Synchronisation sowie Schedulingmechanismen behandelt. Im Anhang des Buches findet sich ein ausführliches Glossar, die genaue Syntax der verwendeten Sprache sowie Musterlösungen zu ausgewählten Übungsaufgaben.

Der aktuelle Stand des Buches sieht so aus, daß der erste Teil (Grundlagen) weitgehend fertiggestellt ist. Der zweite Teil (Imperative Programmierung) ist soweit abgeschlossen, daß damit die wesentlichen Aspekte des Programmwurfs erlernt werden können. Der dritte und vierte Teil des Buches (Objektorientierte und Parallele Programmierung) fehlen noch.

Beim Hamster-Modell steht nicht so sehr das „Learning-by-Listening“ bzw. „Learning-by-Reading“ im Vordergrund, sondern vielmehr das „Learning-by-Doing“. Aus diesem Grund enthalten die einzelnen Kapitel jeweils viele Übungsaufgaben, die intensiv bearbeitet werden sollten. Des weiteren möchte ich den Lesern bzw. Üben den ans Herz legen, sich selbst weitere Aufgaben auszudenken. Programmieren lernt man am besten durch Üben, Üben, Üben. Ich wünsche allen Lesern viel Spaß beim Programmieren Lernen mit dem Hamster.

Dietrich Boles

Inhaltsverzeichnis

I Grundlagen	1
1 Programmierung	5
1.1 Ziele der Programmierung	5
1.2 Algorithmen	5
1.3 Programme	11
2 Programmiersprachen	13
2.1 Programmierparadigmen	13
2.2 Abstraktionsebenen von Programmiersprachen	14
2.3 Syntaxdarstellungen	15
3 Programmentwicklung	19
3.1 Entwicklungsphasen	19
3.2 Entwicklungswerkzeuge	23
4 Computer	25
4.1 Arbeitsweise eines Computers	25
4.2 Aufbau eines Computers	26
4.3 Von-Neumann-Prinzipien der Rechnerarchitektur	28
4.4 Hintergrundspeicher	29
4.5 Betriebssystem	29
4.6 Dateien und Verzeichnisse	29
4.7 Window-System	30

5	Aussagenlogik	31
5.1	Aussagen	31
5.2	Operationen auf Aussagen	31
5.3	Syntax von Aussagen	32
5.4	Äquivalenz von Aussagen	33
5.5	Algebraische Eigenschaften von booleschen Operatoren	34
II	Imperative Programmierung	37
6	Grundlagen des Hamster-Modells	41
6.1	Motivation	41
6.2	Komponenten des Hamster-Modells	44
6.3	Grundlagen der Hamstersprache	46
7	Anweisungen und Programme	49
7.1	Hamster-Befehle	49
7.2	Anweisungen	52
7.3	Programme	53
7.4	Kommentare	54
7.5	Beispielprogramme	57
7.6	Übungsaufgaben	59
8	Prozeduren	61
8.1	Motivation	61
8.2	Prozedurdefinition	62
8.3	Prozeduraufruf	64
8.4	Programme (mit Prozeduren)	66
8.5	Vorteile von Prozeduren	68
8.6	Beispielprogramme	70
8.7	Übungsaufgaben	73

9	Auswahanweisungen	75
9.1	Test-Befehle	75
9.2	Boolesche Operatoren und Ausdrücke	77
9.3	Blockanweisung	82
9.4	Leeranweisung	84
9.5	Bedingte Anweisung	85
9.6	Alternativanweisung	88
9.7	Beispielprogramme	92
9.8	Übungsaufgaben	96
10	Wiederholungsanweisungen	99
10.1	Motivation	99
10.2	while-Anweisung	100
10.3	do-Anweisung	109
10.4	Beispielprogramme	112
10.5	Übungsaufgaben	116
11	Boolesche Funktionen	119
11.1	Motivation	119
11.2	Boolesche return-Anweisung	120
11.3	Definition boolescher Funktionen	121
11.4	Aufruf boolescher Funktionen	124
11.5	Seiteneffekte	129
11.6	Beispielprogramme	131
11.7	Übungsaufgaben	138
12	Programmwurf	141
12.1	Lösen von Problemen	141
12.2	Analyse	142
12.3	Entwurf	144
12.4	Implementierung	148
12.5	Test	149
12.6	Dokumentation	151
12.7	Ein weiteres Beispiel	152
12.8	Übungsaufgaben	164

13 Variablen und Ausdrücke	167
14 Prozeduren und Funktionen	169
15 Funktionsparameter	171
16 Rekursion	173
Literatur	174

Teil I

Grundlagen

Der erste Teil dieses Kurses ist mit dem Begriff *Grundlagen* überschrieben. Hier werden grundlegende Begriffe und Aspekte erläutert, die für das Verständnis der folgende Teile des Kurses notwendig sind. Die Beschreibungen sind bewußt einfach gehalten bzw. werden vereinfacht dargestellt, um Sie nicht mit im Rahmen dieses Kurses unwichtigen Details zu überhäufen und damit zu demotivieren.

Das erste Kapitel enthält eine Einführung in die Programmierung. Die Ziele der Programmierung werden erläutert und die Begriffe des Algorithmus und des Programms eingeführt. Das zweite Kapitel widmet sich den Programmiersprachen. Es werden die verschiedenen Programmierparadigmen und Abstraktionsebenen von Programmiersprachen kurz angesprochen. Detaillierter wird auf die Syntax von Programmiersprachen eingegangen. Kapitel 3 schildert den gesamten Vorgang, der notwendig ist, um ein auf einem Computer lauffähiges und korrektes Programm zu entwickeln. Mit dem Computer selbst beschäftigt sich Kapitel 4. Seine Bestandteile und Arbeitsweise werden geschildert. Eine wichtige mathematische Grundlage der Programmierung bildet die Aussagenlogik. Sie wird in Kapitel 5 kurz eingeführt.

Kapitel 1

Programmierung

1.1 Ziele der Programmierung

Die *Programmierung* ist ein Teilgebiet der Informatik, das sich im weiteren Sinne mit Methoden und Denkweisen bei der Lösung von Problemen mit Hilfe von Computern und im engeren Sinne mit dem Vorgang der Programmerstellung befaßt. Unter einem *Programm* versteht man dabei eine in einer speziellen Sprache verfaßte Anleitung zum Lösen eines Problems durch einen Computer. Programme werden auch unter dem Begriff *Software* subsumiert. Konkreter ausgedrückt ist das Ziel der Programmierung bzw. Softwareentwicklung, zu gegebenen Problemen Programme zu entwickeln, die auf Computern ausführbar sind und die Probleme korrekt und vollständig lösen, und das möglichst effizient.

Die hier angesprochenen Probleme können von ganz einfacher Art sein wie das Addieren oder Subtrahieren von Zahlen oder das Sortieren einer gegebenen Datenmenge. Komplexere Probleme reichen von der Erstellung von Computerspielen oder der Datenverwaltung von Firmen bis hin zur Steuerung von Raketen. Im Umfeld dieses Buches werden nur relativ einfache Probleme behandelt, die innerhalb weniger Minuten bzw. Stunden vom Programmierer gelöst werden können. Dahingegen kann das Lösen von komplexen Problemen Monate ja sogar Jahre dauern und den Einsatz eines ganzen Teams von Menschen erforderlich machen.

Der Vorgang des Erstellens von Programmen zu einfachen Problemen wird *Programmieren im Kleinen* genannt. Er erstreckt sich von der Analyse des gegebenen Problems über die Entwicklung einer Problemlösebeschreibung bis hin zur eigentlichen Programmformulierung und -ausführung. Die Bearbeitung komplexer Probleme umfaßt darüber hinaus weitere Phasen wie eine vorangehende Systemanalyse und die spätere Wartung der erstellten Software und ist Gegenstand des *Softwareengineerings*, einem Teilgebiet der Informatik, auf das in diesem Buch nicht näher eingegangen wird.

1.2 Algorithmen

Als *Algorithmus* bezeichnet man eine Arbeitsanleitung für einen Computer. Der Begriff des Algorithmus ist ein zentraler Begriff der Programmierung. In diesem Abschnitt wird der Begriff zunächst motiviert und dann genauer definiert. Anschließend werden verschiedene Möglichkeiten

der Formulierung von Algorithmen vorgestellt, und es wird auf die Ausführung von Algorithmen eingegangen. Algorithmen besitzen einige charakteristische Eigenschaften, die zum Abschluß dieses Abschnitts erläutert werden.

1.2.1 Arbeitsanleitungen

Wenn Sie etwas Leckeres kochen wollen, gehen Sie nach einem Rezept vor. Der Zusammenbau eines Modellflugzeugs erfordert eine Bastelanleitung. Beim Klavierspielen haben Sie ein Notenheft vor sich. Zum Skatspielen sind Spielregeln notwendig. Mit derlei Anleitungen wie Kochrezepten, Bastelanleitungen, Partituren und Spielregeln kommen Sie tagtäglich in Berührung. Wenn Sie sich den Aufbau solcher Anleitungen genauer anschauen, können Sie feststellen, daß sie alle etwas gemeinsam haben. Sie bestehen aus einzelnen Angaben (Anweisungen), die in einer angegebenen Reihenfolge ausgeführt zu dem gewünschten Ergebnis führen:

Kochrezept:

```
Zwiebel feinhacken;
Brötchen einweichen;
aus Mett, gemischtem Hack, Eiern, feingehackter Zwiebel
und eingeweichtem und gut ausgedrücktem Brötchen
einen Fleischteig bereiten;
mit Salz und Pfeffer herzhaft würzen;
Trauben waschen, halbieren und entkernen;
...
```

Teilweise sind gewisse Anweisungen in den Arbeitsanleitungen nur unter bestimmten Bedingungen auszuführen (*bedingte Anweisungen*). Ausgedrückt wird dieser Sachverhalt durch ein: *Wenn eine Bedingung erfüllt ist, dann tue dies ansonsten tue das*

Anleitung für einen Fußballschiedsrichter:

```
ein Spieler von Mannschaft A wird von einem Spieler
von Mannschaft B gefoult;
wenn das Foul im Strafraum von Mannschaft B erfolgt
dann pfeife Strafstoß,
ansonsten pfeife Freistoß
```

Darüberhinaus kommt es auch schonmal vor, daß gewisse Anweisungen in einer sogenannten *Schleife* mehrmals hintereinander ausgeführt werden sollen: *Solange eine Bedingung erfüllt ist, tue folgendes*

Anleitung beim Mensch-Ärgere-Dich-Nicht-Spiel:

```
Solange ein Spieler eine 6 würfelt, bleibt er am Zug.
```

Weiterhin fällt auf, daß zum Ausführen der Anleitungen gewisse Voraussetzungen erfüllt sein müssen: Zum Kochen werden Zutaten benötigt, Basteln ist nicht ohne Materialien möglich und zum Spielen sind Spielkarten oder Spielfiguren unabdingbar.

Zutaten beim Kochen:

```
250g Mett
```

250g gemischtes Hack
2 Eier
1 Zwiebel
1 Brötchen
Pfeffer, Salz

Ein weiteres Merkmal dieser alltäglichen Arbeitsanleitungen ist, daß sie selten exakt formuliert sind, sondern oft Teile enthalten, die unterschiedlich interpretiert werden können. Im allgemeinen sagt uns unser gesunder Menschenverstand dann, was in der speziellen Situation zu tun ist. Beim obigen Kochrezept ist bspw. die Anleitung „mit Salz und Pfeffer herzhaft würzen“ wenig präzise für jemanden, der noch nie gekocht hat.

1.2.2 Definition des Begriffs Algorithmus

Anleitungen, wie sie im letzten Abschnitt erörtert worden sind, werden von Menschen ausgeführt, um unter bestimmten Voraussetzungen zu einem bestimmten Ergebnis zu gelangen. Genauso wie Menschen benötigen auch Computer Arbeitsanleitungen, um Probleme zu lösen. Arbeitsanleitungen für einen Computer bezeichnet man als *Algorithmen*. Algorithmen weisen dabei viele Merkmale auf, die wir im letzten Abschnitt für Anleitungen für Menschen kennengelernt haben. Sie bestehen aus Anweisungen, können bedingte Anweisungen und Schleifen enthalten und operieren auf vorgegebenen Materialien, den Daten. Sie unterscheiden sich jedoch darin, daß sie wesentlich exakter formuliert sein müssen, da Computer keine Intelligenz besitzen, um mehrdeutige Formulierungen selbständig interpretieren zu können.

Damit kann der Begriff *Algorithmus* definiert werden als *eine Arbeitsanleitung zum Lösen eines Problems bzw. einer Aufgabe, die so präzise formuliert ist, daß sie von einem Computer ausgeführt werden kann.*

1.2.3 Formulierung von Algorithmen

Zur Beschreibung von Algorithmen existieren mehrere Möglichkeiten bzw. Notationen, von denen die gängigsten anhand eines kleinen Beispiels im folgenden kurz vorgestellt werden. Bei dem Beispiel geht es um die Lösung des Problems, die Summe aller Zahlen bis zu einer vorgegebenen nicht-negativen Zahl n zu berechnen. Mathematisch definiert ist also die folgende Funktion f zu berechnen:

$$f : \mathbb{N} \rightarrow \mathbb{N} \text{ mit } f(n) = \sum_{i=1}^n i \text{ für } n \in \mathbb{N}$$

1.2.3.1 Umgangssprachliche Formulierung

Arbeitsanleitungen für Menschen werden im allgemeinen umgangssprachlich formuliert. Es gibt häufig keine vorgegebenen Schemata oder Regeln. Der Mensch interpretiert die Anweisungen gemäß seines Wissens oder bereits vorliegender Erfahrungen. Auch Algorithmen lassen sich prinzipiell umgangssprachlich beschreiben. Die Beschreibung sollte jedoch so exakt sein, daß sie ohne weitergehende intellektuelle Anstrengungen in ein Programm oder eine andere Notation

übertragen werden kann. Eine umgangssprachliche Beschreibung des Algorithmus zum Lösen des Beispielproblems lautet bspw.:

Addiere für eine vorgegebene natürliche Zahl n
die Zahlen von 1 bis n . Dies ist das Resultat.

1.2.3.2 Programmablaufpläne

Eine normierte Methode zur graphischen Darstellung von Algorithmen stellen die *Programmablaufpläne* (PAP) – auch *Flußdiagramme* genannt – dar. In Abbildung 1.1 (a) werden die wichtigsten Elemente der graphischen Notation skizziert. Daneben findet sich in Abbildung 1.1 (b) ein Programmablaufplan zur Lösung des Beispielproblems.

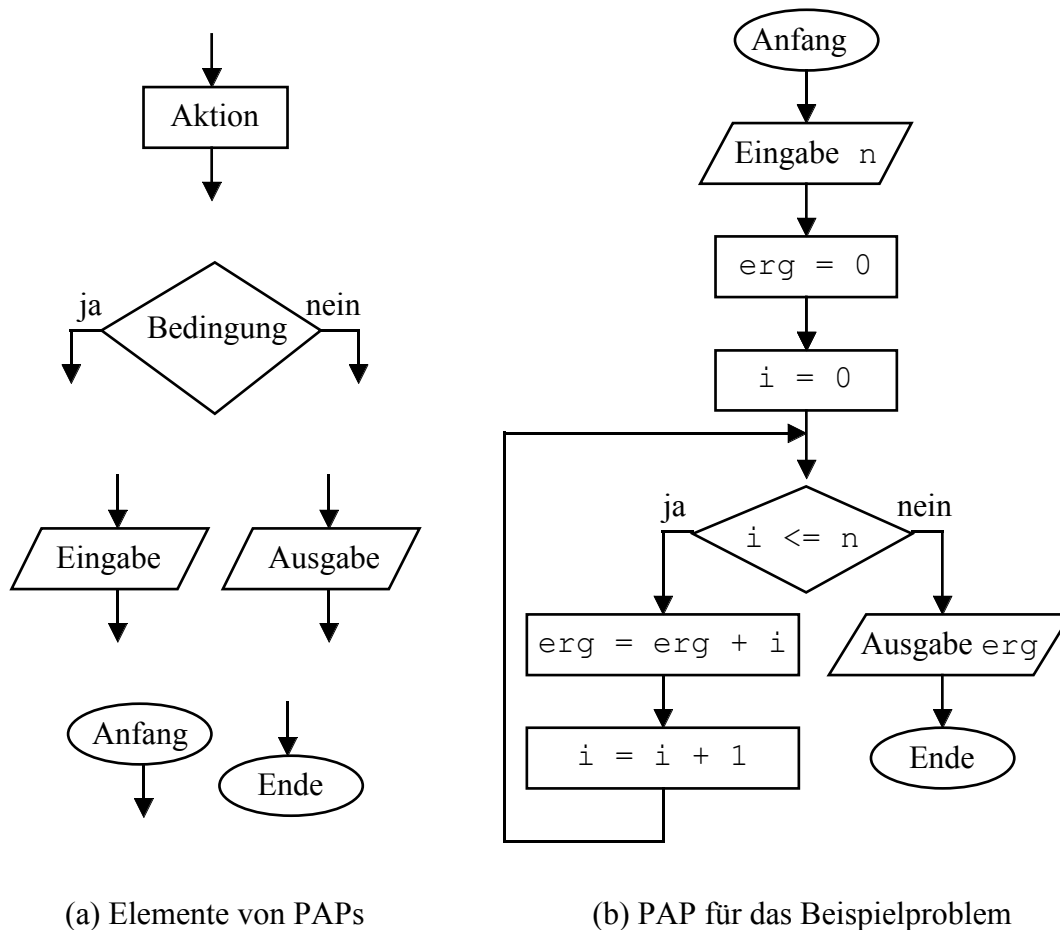


Abbildung 1.1: Programmablaufpläne

1.2.3.3 Struktogramme

Struktogramme (*Nassi-Shneiderman-Diagramme*) bieten eine weitere graphische Alternative zur Darstellung von Algorithmen. Gegenüber Programmablaufplänen sind sie im allgemeinen übersichtlicher und verständlicher. Die wichtigsten Elemente, aus denen sich Struktogramme zusam-

mensetzen, sind Abbildung 1.2 (a) zu entnehmen. In Abbildung 1.2 (b) wird eine Lösung des Beispielproblems mit Hilfe von Struktogrammen formuliert.

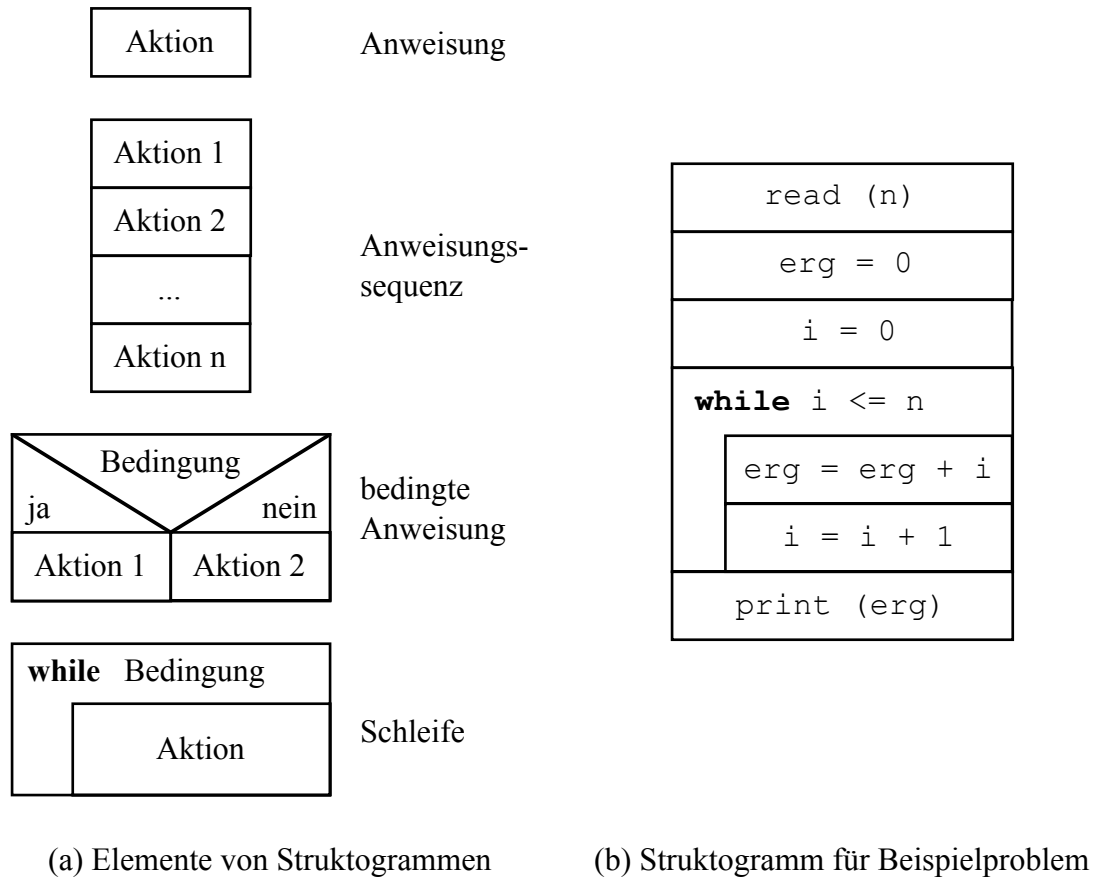


Abbildung 1.2: Struktogramme

1.2.3.4 Programmiersprache

Algorithmen lassen sich auch in der Notation einer bestimmten Programmiersprache formulieren. Folgendes an die Syntax der Programmiersprache Java angelehntes Programm löst das Beispielproblem:

```

int n = readInt();
int erg = 0;
int i = 0;
while (i <= n) {
    erg = erg + i;
    i = i + 1;
}
printInt(erg);
  
```

1.2.4 Ausführung von Algorithmen

Algorithmen stellen eine Arbeitsanleitung dar, d.h. werden sie ausgeführt, sollten sie nach Abarbeitung der einzelnen Anweisungen das erwartete Ergebnis liefern. Bei der Ausführung eines Algorithmus läuft ein sogenannter *Prozeß* ab. Dieser Prozeß wird durch einen *Prozessor* gesteuert. Bei den Arbeitsanleitungen für Menschen (siehe Abschnitt 1.2.1) ist der Mensch der Prozessor, bei Algorithmen ist dies der Computer, der die Notation, in der der Algorithmus formuliert ist, kennen und verstehen muß.

Prinzipiell kann jeder Algorithmus auch durch einen Menschen ausgeführt werden. Computer haben gegenüber uns Menschen jedoch gewisse Vorteile:

- Ihre hohe Rechengeschwindigkeit: Computer können heutzutage einige Millionen Rechenoperationen pro Sekunde ausführen.
- Ihre große Zuverlässigkeit: Computer ermüden nicht und führen fehlerlos genau die Anweisungen durch, die ihnen der Mensch vorschreibt.
- Ihre gewaltige Speicherfähigkeit: Computer können Milliarden Daten dauerhaft abspeichern und sie auch sehr schnell wiederfinden.

1.2.5 Eigenschaften von Algorithmen

Algorithmen weisen folgende Eigenschaften auf:

- Eindeutigkeit: Algorithmen liefern eine eindeutige Beschreibung zur Lösung eines gegebenen Problems. Sie dürfen keine widersprüchlichen Aussagen enthalten.
- Parametrisierbarkeit: Algorithmen lösen nicht nur genau ein spezielles Problem sondern eine Klasse von Problemen mit dem gleichen Schema. So löst der Algorithmus in Abschnitt 1.2.3 das Problem der Summenbildung von ganzen Zahlen nicht nur für eine fest vorgegebene Zahl sondern für beliebige Zahlen n . n wird auch Parameter des Algorithmus genannt.
- Finitheit: Die Beschreibung eines Algorithmus besitzt eine endliche Länge.
- Ausführbarkeit: Algorithmen dürfen keine Anweisungen enthalten, die prinzipiell nicht ausführbar sind.
- Terminierung: Algorithmen müssen nach endlich vielen Schritten terminieren, d.h. sie müssen ein Ergebnis liefern und dann anhalten.
- Determiniertheit: Algorithmen heißen determiniert, wenn sie mit gleichen Startbedingungen mehrfach ausgeführt immer die gleichen Ergebnisse liefern.
- Determinismus: Algorithmen heißen deterministisch, wenn zu jedem Zeitpunkt ihrer Ausführung höchstens eine Möglichkeit zur Fortsetzung besteht.

Die ersten drei Eigenschaften sind dabei Eigenschaften der Formulierung eines Algorithmus an sich (statische Eigenschaften), die letzten vier sind Eigenschaften der Ausführung eines Algorithmus (dynamische Eigenschaften).

Nicht alle Algorithmen erfüllen die letzten drei Eigenschaften. Steuerungsprogramme in Betriebssystemen sind häufig nicht-terminierend und aus dem Bereich der Stochastik sind bspw. nicht-determinierte und nicht-deterministische Algorithmen bekannt.

1.2.6 Praxisrelevante Eigenschaften von Algorithmen

Algorithmen sind Beschreibungen zur Lösung eines Problems bzw. einer Aufgabe. Dabei ist jedoch festzustellen, daß es zur Lösung eines Problems nicht nur immer genau einen korrekten Algorithmus gibt. Vielmehr kann sogar mathematisch bewiesen werden, daß es zu jedem Algorithmus unendlich viele verschiedene äquivalente Algorithmen gibt, also Algorithmen, die dasselbe Problem mit identischen Ergebnissen lösen.

In der Praxis ist es aber im allgemeinen nicht gleichgültig, welchen von zwei äquivalenten Algorithmen man zur Lösung eines Problems einsetzt. Kriterien, die in der Praxis bei der Findung bzw. Auswahl eines Algorithmus eine wichtige Rolle spielen, sind insbesondere:

- **Effizienz:** Algorithmen sollten so schnell wie möglich und mit möglichst wenig Ressourcen zu einem korrekten Ergebnis kommen.
- **Speicherbedarf:** Die Beschreibung eines Algorithmus sollte möglichst knapp sein, worunter die Verständlichkeit aber nicht leiden darf.
- **Erweiterbarkeit:** Algorithmen sollten ohne großen Aufwand an geänderte Anforderungen anpaßbar sein.
- **Wiederverwendbarkeit:** Algorithmen sollten so formuliert werden, daß sie nicht nur einmalig, sondern auch zur Lösung von Teilproblemen in anderen Zusammenhängen genutzt werden können.
- **Portabilität:** Algorithmen sollten nicht auf einen bestimmten Computertyp zugeschnitten sein, sondern prinzipiell auf beliebigen Computern ausgeführt werden können.
- **Zuverlässigkeit:** Algorithmen sollten das Problem korrekt und vollständig lösen. Zu beachten sind dabei insbesondere sogenannte Grenzfälle.

1.3 Programme

Als *Programm* wird ein in einer Programmiersprache formulierter Algorithmus bezeichnet. Die Formulierung von Programmen erfolgt in sehr konkreter und eingeschränkter Form:

- Programme werden im exakt definierten und eindeutigen Formalismus einer Programmiersprache verfaßt.
- Daten, auf denen Programme operieren, unterliegen einer festgelegten Darstellungsform.

Unter *Programmieren* versteht man das Erstellen von Programmen. Der gesamte Programmentwicklungsvorgang wird in Kapitel 3 ausführlich behandelt. Zuständig für die Entwicklung von Programmen ist der *Programmierer*. Die Formulierung von Programmen erfolgt in der Regel unter Verwendung der uns bekannten Zeichen wie Buchstaben, Ziffern und Sonderzeichen. Eine Programmbeschreibung wird auch als *Programmcode*, *Quellcode* oder *Source-Code* bezeichnet. Der Programmcode selbst kann im allgemeinen noch nicht direkt von einem Computer ausgeführt werden, er muß zuvor noch mit Hilfe eines Compilers in eine maschinenverständliche Form – ein *ausführbares Programm* – transformiert werden. Die Programmausführung wird durch den *Aufruf* des ausführbaren Programms gestartet.

Kapitel 2

Programmiersprachen

Eine *Programmiersprache* ist eine zum Formulieren von Programmen geschaffene künstliche Sprache. Anders als natürliche Sprachen wie Deutsch und Englisch müssen Programmiersprachen sehr exakt definiert werden, da sie ja von einem Computer, der weder Intelligenz noch Intuition noch Erfahrung besitzt, verstanden und verarbeitet werden müssen. Bei der Definition einer Programmiersprache müssen ihre Syntax und Semantik angegeben werden. Erstere beschreibt, welche Zeichenfolgen überhaupt als Programme dieser Programmiersprache gültig sind. Letztere gibt an, was korrekte Zeichenfolgen schließlich bei der Ausführung auf einem Computer für Auswirkungen haben.

2.1 Programmierparadigmen

Vielleicht fragen Sie sich jetzt: Wieso gibt es eigentlich nicht nur eine einzige Programmiersprache, mit der alle Programmierer arbeiten? Da Programmiersprachen anders als natürliche Sprachen, die sich über Jahrhunderte hinweg entwickelt haben, ja künstlich definiert werden müssen, hätte man sich doch von Anfang an auf eine einheitliche Programmiersprache festlegen können.

Zunächst kann man feststellen, daß es bestimmte Programmiersprachen gibt, die primär für Programmieranfänger definiert worden sind. Sie sind meistens sehr einfach gehalten, d.h. der Sprachumfang ist relativ gering. Sie sind leicht zu erlernen, eignen sich aber nicht besonders zum Lösen sehr komplexer Probleme. Hierfür werden sehr viel mächtigere Programmiersprachen benötigt.

Eine andere Klassifizierung unterscheidet sogenannte niedere *Maschinensprachen* (*maschinen-nahe Programmiersprachen*) und höhere *problemorientierte Programmiersprachen*. Maschinensprachen ermöglichen die Erstellung sehr effizienter Programme. Sie sind jedoch abhängig vom speziellen Computertyp. Dahingegen orientieren sich die höheren Programmiersprachen nicht so sehr an den von den Computern direkt ausführbaren Befehlen, sondern eher an den zu lösenden Problemen. Sie sind für Menschen verständlicher und einfacher zu handhaben.

Ein weiterer Grund für die Existenz der vielen verschiedenen Programmiersprachen liegt in der Tatsache, daß die zu lösenden Probleme nicht alle gleichartig sind. So werden häufig neue Programmiersprachen definiert, die speziell für bestimmte Klassen von Problemen geeignet sind.

Den höheren Programmiersprachen liegen bestimmte Konzepte zugrunde, mit denen die Lösung von Problemen gefaßt wird. Im wesentlichen lassen sich hier fünf Kategorien unterscheiden:

- Imperative Programmiersprachen: Programme bestehen aus Folgen von Befehlen (PASCAL, MODULA-2).
- Funktionale Programmiersprachen: Programme werden als mathematische Funktionen betrachtet (LISP, MIRANDA).
- Prädikative Programmiersprachen: Programme bestehen aus Fakten (gültige Tatsachen) und Regeln, die beschreiben, wie aus gegebenen Fakten neue Fakten hergeleitet werden können (PROLOG).
- Regelbasierte Programmiersprachen: Programme bestehen aus „wenn-dann-Regeln“; wenn eine angegebene Bedingung gültig ist, dann wird eine angegebene Aktion ausgeführt (OPS5).
- Objektorientierte Programmiersprachen: Programme bestehen aus Objekten, die bestimmte (Teil-)Probleme lösen und zum Lösen eines Gesamtproblems mit anderen Objekten über Nachrichten kommunizieren können (SMALLTALK).

Nicht alle Programmiersprachen können eindeutig einer dieser Klassen zugeordnet werden. So ist bspw. LOGO eine funktionale Programmiersprache, die aber auch imperative Sprachkonzepte besitzt. Java und C++ können als imperative objektorientierte Programmiersprachen klassifiziert werden, denn Java- und C++-Programme bestehen aus kommunizierenden Objekten, die intern mittels imperativer Sprachkonzepte realisiert werden.

Programmiersprachen einer Kategorie unterscheiden sich häufig nur in syntaktischen Feinheiten. Die grundlegenden Konzepte sind ähnlich. Von daher ist es im allgemeinen nicht besonders schwierig, eine weitere Programmiersprache zu erlernen, wenn man bereits eine Programmiersprache derselben Kategorie beherrscht. Anders verhält es sich jedoch beim Erlernen von Programmiersprachen anderer Kategorien, weil hier die zugrundeliegenden Konzepte stark voneinander abweichen.

2.2 Abstraktionsebenen von Programmiersprachen

Programmiersprachen sind sehr exakte künstliche Sprachen zur Formulierung von Programmen. Sie dürfen keine Mehrdeutigkeiten bei der Programmerstellung zulassen, damit der Computer das Programm auch korrekt ausführen kann. Bei der Definition einer Programmiersprache muß ihre *Lexikalik*, *Syntax*, *Semantik* und *Pragmatik* definiert werden:

- Lexikalik: Die Lexikalik einer Programmiersprache definiert die gültigen Zeichen bzw. Wörter, aus denen Programme der Programmiersprache zusammengesetzt sein dürfen.
- Syntax: Die Syntax einer Programmiersprache definiert den korrekten Aufbau der Sätze aus gültigen Zeichen bzw. Wörtern, d.h. sie legt fest, in welcher Reihenfolge lexikalisch korrekte Zeichen bzw. Wörter im Programm auftreten dürfen.

- Semantik: Die Semantik einer Programmiersprache definiert die Bedeutung syntaktisch korrekter Sätze, d.h. sie beschreibt, was passiert, wenn bspw. bestimmte Anweisungen ausgeführt werden.
- Pragmatik: Die Pragmatik einer Programmiersprache definiert ihren Einsatzbereich, d.h. sie gibt an, für welche Arten von Problemen die Programmiersprache besonders gut geeignet ist.

Die Lexikalik wird häufig mit in die Syntax mit einbezogen. Wie die Syntax einer Programmiersprache definiert werden kann, wird im nächsten Abschnitt detailliert erläutert. Die Semantik einer Programmiersprache wird in der Regel nur umgangssprachlich beschrieben, es existieren jedoch auch Möglichkeiten für eine formal saubere (mathematische) Definition. Für die Definition der Pragmatik einer Programmiersprache existiert kein bestimmter Formalismus. Sie wird deshalb umgangssprachlich angegeben.

2.3 Syntaxdarstellungen

Die Syntax einer Programmiersprache legt fest, welche Zeichenreihen bzw. Folgen von Wörtern korrekt formulierte („syntaktisch korrekte“) Programme der Sprache darstellen und welche nicht. Zur Überprüfung der syntaktischen Korrektheit eines Programms muß deshalb zuvor die Syntax der Programmiersprache formal beschrieben werden. Hierzu existieren verschiedene Möglichkeiten. In den folgenden zwei Unterabschnitten werden die zwei gängigsten vorgestellt, nämlich die *Syntaxdiagramme* und die *Backus-Naur-Form*.

Sowohl Syntaxdiagramme als auch die Backus-Naur-Form sind Techniken zur Darstellung sogenannter kontextfreier Programmiersprachen. Die meisten Programmiersprachen sind jedoch kontextsensitiv, d.h. es lassen sich nicht alle Regeln zur Beschreibung der Syntax mit den beiden Techniken beschreiben. Nicht formulierbare Eigenschaften der Syntax werden daher umgangssprachlich ergänzt.

2.3.1 Syntaxdiagramme

Bei den Syntaxdiagrammen handelt es sich um eine graphische und daher sehr übersichtliche Notation zur Definition der Syntax einer Programmiersprache. Syntaxdiagramme sind folgendermaßen definiert:

- Zur Beschreibung der Syntax einer Sprache existiert in der Regel eine Menge von Syntaxdiagrammen, die zusammen die Syntax definieren. In der Menge existiert genau ein *übergeordnetes Syntaxdiagramm*, bei dem die Definition beginnt.
- Jedes Syntaxdiagramm besitzt einen Namen (Bezeichnung).
- Jedes Syntaxdiagramm besteht aus runden und eckigen Kästchen sowie aus Pfeilen.
- In jedem rechteckigen Kästchen steht die Bezeichnung eines (anderen) Syntaxdiagramm der Menge von Syntaxdiagrammen (ein sogenanntes *Nicht-Terminalsymbol*).

- In jedem runden Kästchen steht ein Wort (*Token*, *Terminalsymbol*) der Sprache.
- Aus jedem Kästchen führt genau ein Pfeil hinaus und genau einer hinein.
- Pfeile dürfen sich aufspalten und zusammengezogen werden.
- Jedes Syntaxdiagramm besitzt genau einen Pfeil, der von keinem Kästchen ausgeht (eintretender Pfeil) und genau einen Pfeil, der zu keinem Kästchen führt (austretender Pfeil).

Token einer Programmiersprache werden durch die Lexikalik der Sprache definiert. Sie stellen die kleinsten zusammenhängenden Grundsymbole einer Sprache, bswp.:

- einfache und zusammengesetzte Symbole (+, <=, (, ...).
- Schlüsselwörter (reservierte Wörter)
- Bezeichner
- Konstanten

Trennzeichen (Zeichen für die Trennung von Token) wie Blanks, Tabulatoren oder Zeilenumbrüche sowie Kommentare werden im allgemeinen nicht in den Syntaxdiagrammen berücksichtigt.

Mit Hilfe von Syntaxdiagrammen kann festgestellt werden, ob eine bestimmte Zeichenfolge (ein Programm) syntaktisch korrekt ist. Dazu fängt man bei dem eintretenden Pfeil des übergeordneten Syntaxdiagramm an und verfolgt die Pfeile. Erreicht man ein rundes Kästchen, so muß das entsprechende Token als nächstes in der Zeichenfolge auftreten. Erreicht man ein eckiges Kästchen, springt man in das entsprechend bezeichnete Syntaxdiagramm. Existieren alternative Wege, so wählt man den entsprechenden aus. Gibt es keinen Weg durch die Syntaxdiagramme, so ist das Programm syntaktisch nicht korrekt. Nach der Abarbeitung der Zeichenfolge muß der austretende Pfeil des übergeordneten Syntaxdiagramm erreicht worden sein. Sonst ist das Programm ebenfalls nicht syntaktisch korrekt.

Das Prinzip, nach dem Syntaxdiagramme arbeiten, läßt sich durch eine Analogie veranschaulichen: In einem Zoo gibt es eine Menge von Gehegen mit verschiedenen Tieren. Die Gehege können durch Besucher auf Wegen erreicht werden. Wegen der großen Besucherzahlen dürfen dabei die Wege jeweils nur in einer Richtung begangen werden. Es existieren Kreuzungen, an denen mehrere Wege eingeschlagen werden können. In Abbildung 2.1 werden anhand eines Syntaxdiagramms (Wegeplan des Zoos) die möglichen Wege durch den Zoo veranschaulicht. Die Gehege stellen dabei die Token dar. Der Übersichtlichkeit halber ist der Plan in zwei Teilpläne (Zoo, Säugetiere) aufgeteilt.

Ein Besucher möchte nun im Zoo eine Fotoserie erstellen. Dabei muß (!) er jeweils ein Foto schießen, wenn er an einem Gehege vorbei kommt. Er orientiert sich an dem Wegeplan. Offenbar kann er die möglichen Bildsequenzen ermitteln, indem er die möglichen Wege durch den Zoo nachvollzieht. Erreicht er im Plan das eckige Kästchen *Säugetiere*, so zeigt der Teilplan *Säugetiere* den weiteren Weg, bis er diesen wieder verläßt und am Ausgang des eckigen Kästchens *Säugetiere* seinen Weg fortsetzt.

Mögliche Bildsequenzen sind zum Beispiel:

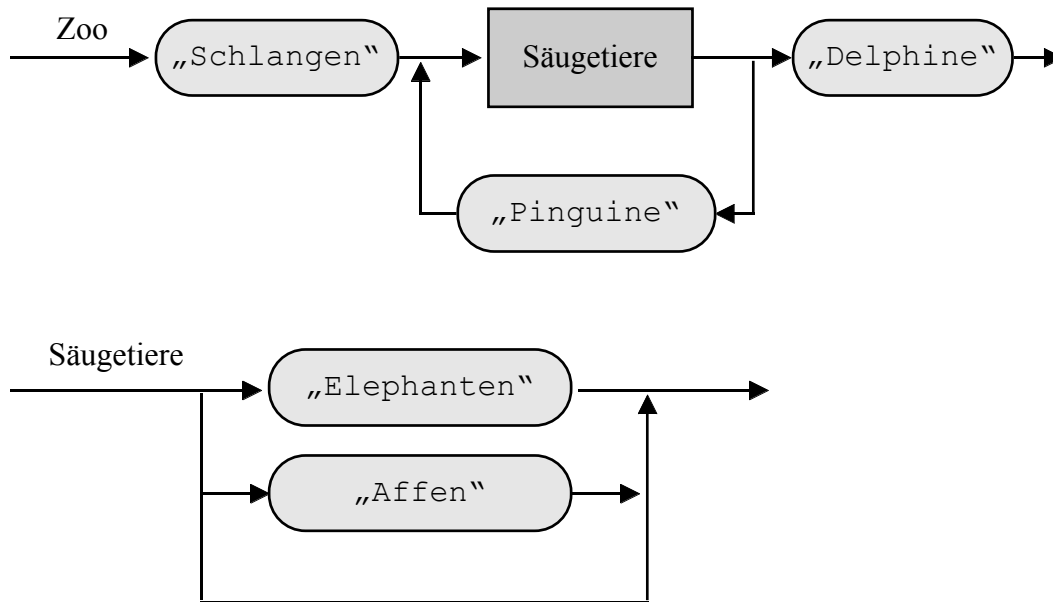


Abbildung 2.1: Wegeplan im Zoo als Syntaxdiagramm

- Schlangen Delphine
- Schlangen Elephanten Pinguine Delphine
- Schlangen Elephanten Pinguine Affen Delphine
- Schlangen Elephanten Pinguine Elephanten Pinguine Delphine

Dahingegen sind folgende Bildsequenzen nicht möglich:

- Elephanten Delphine (weil der Besucher anfangs auf jeden Fall zuerst am Schlangengehege vorbeikommt)
- Schlangen Pinguine (weil der Besucher am Ende seines Zoobesuchs auf jeden Fall am Delphingehege vorbeikommt)
- Schlangen Elephanten Affen Delphine (weil der Besucher zwischen dem Besuch des Elephanten- und Affengeheges auf jeden Fall einmal am Pinguingehege vorbeigehen muß)
- Schlangen Pinguine Schlangen Delphine (weil der Besucher nur am Anfang seines Zoobesuchs am Schlangengehege vorbeikommt)

2.3.2 Backus-Naur-Form

Bei der Backus-Naur-Form (BNF) handelt es sich um eine textuelle Beschreibungsform für die Syntax von Programmiersprachen. Das Äquivalent zu einem einzelnen Syntaxdiagramm wird hier *Produktion* oder *BNF-Regel* genannt. Produktionen besitzen eine linke und eine rechte Seite, die durch das Metazeichen $::=$ getrennt sind. Auf der linken Seite stehen jeweils einzelne

Nicht-Terminalsymbole, die immer in spitze Klammern gesetzt werden. Die rechte Seite enthält Folgen von Terminal-, Nicht-Terminalsymbolen und dem Meta-Symbol |. Terminalsymbole werden in Hochkommata gesetzt. Das Meta-Symbol | bedeutet „oder“ und entspricht einem sich aufspaltendem Pfeil in den Syntaxdiagrammen. Das „e“ (bzw. Epsilon) steht für einen leeren Weg.

Das Zoo-Beispiel aus Abbildung 2.1 wird in der BNF folgendermaßen formuliert:

```

<Zoo>                ::= "Schlangen" <Saeugetiere-und-mehr>

<Saeugetiere-und-mehr> ::= <Saeugetiere> "Pinguine" <Saeugetiere-und-mehr> |
                           <Saeugetiere> "Delphine"

<Saeugetiere>        ::= "Elefanten" |
                           "Affen" |
                           e

```

Im Laufe der Zeit haben sich einige Abkürzungsmöglichkeiten entwickelt, die zu besser lesbaren Produktionen führen:

- [...] bedeutet: Symbole oder Symbolfolgen innerhalb der Klammern können auch weggelassen werden.
- {...} bedeutet: Symbole oder Symbolfolgen innerhalb der Klammern können beliebig oft wiederholt oder auch ganz weggelassen werden.
- (...|...) bedeutet: genau ein alternatives Symbol oder eine alternative Symbolfolge innerhalb der Klammern muß auftreten.

Das Zoo-Beispiel kann damit folgendermaßen formuliert werden:

```

<Zoo>                ::= "Schlangen"
                           <Saeugetiere>
                           {"Pinguine" <Saeugetiere>}
                           "Delphine"

<Saeugetiere> ::= "Elefanten" |
                           "Affen" |
                           e

```

Kapitel 3

Programmentwicklung

3.1 Entwicklungsphasen

Ziel der Programmierung ist die Entwicklung von Programmen, die gegebene Probleme korrekt und vollständig lösen. Ausgangspunkt der Programmentwicklung ist also ein gegebenes Problem, Endpunkt ist ein ausführbares Programm, das korrekte Ergebnisse liefert. Den Weg vom Problem zum Programm bezeichnet man auch als *Problemlöse-* oder *Programmentwicklungsprozeß* oder kurz *Programmierung*. Im Rahmen dieses Kurses werden nur relativ kleine Probleme behandelt. Für diese kann der Problemlöseprozeß, der in Abbildung 3.1 skizziert wird, in mehrere Phasen zerlegt werden. Verfahren für die Bearbeitung komplexer Probleme sind Gegenstand des Softwareengineering, einem eigenständigen Teilgebiet der Informatik, auf das hier nicht näher eingegangen wird. Die im folgenden erläuterten Phasen werden in der Regel nicht streng sequentiell durchlaufen. Durch neue Erkenntnisse, aufgetretene Probleme und Fehler wird es immer wieder zu Rücksprüngen in frühere Phasen kommen.

3.1.1 Analyse

In der Analysephase wird das zu lösende Problem bzw. das Umfeld des Problems genauer untersucht. Insbesondere folgende Fragestellungen sollten bei der Analyse ins Auge gefaßt und auch mit anderen Personen diskutiert werden:

- Ist die Problemstellung exakt und vollständig beschrieben?
- Was sind mögliche Initialzustände bzw. Eingabewerte (Parameter) für das Problem?
- Welches Ergebnis wird genau erwartet, wie sehen der gewünschte Endzustand bzw. die gesuchten Ausgabewerte aus?
- Gibt es Randbedingungen, Spezialfälle bzw. bestimmte Zwänge (Constraints), die zu berücksichtigen sind?
- Lassen sich Beziehungen zwischen Initial- und Endzuständen bzw. Eingabe- und Ausgabewerten herleiten?

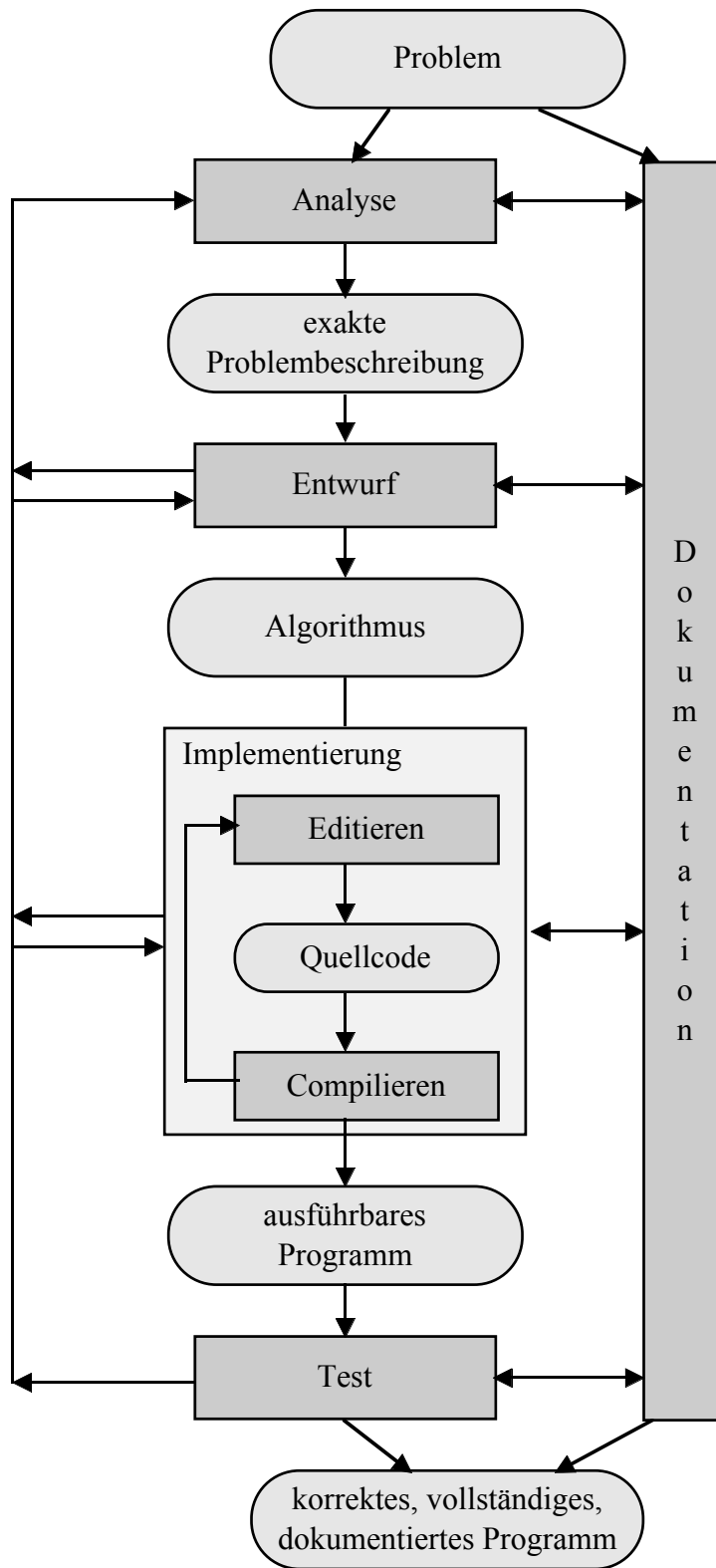


Abbildung 3.1: Programmentwicklungsphasen

Erst wenn alle diese Fragestellungen gelöst und eine exakte Problembeschreibung vorliegt, sollte in die nächste Phase verzweigt werden. Es hat sich gezeigt, daß Fehler, die aus einer nicht ordentlich durchgeführten Analyse herrühren, zu einem immensen zusätzlichen Arbeitsaufwand in späteren Phasen führen können. Deshalb sollte insbesondere in dieser Phase mit größter Sorgfalt gearbeitet werden.

3.1.2 Entwurf

Nachdem die Problemstellung präzisiert und verstanden worden ist, muß in der Entwurfsphase ein Algorithmus zum Lösen des Problems entworfen werden. Der Entwurfsprozeß kann im allgemeinen nicht mechanisch durchgeführt werden, vor allen Dingen ist er nicht automatisierbar. Vielmehr kann man ihn als einen kreativen Prozeß bezeichnen, bei dem Auffassungsgabe, Intelligenz und vor allem Erfahrung des Programmierers eine wichtige Rolle spielen. Diese Erfahrung kann insbesondere durch fleißiges Üben erworben werden.

Trotzdem können die folgenden Ratschläge beim Entwurf eines Algorithmus nützlich sein:

- Sie sollten sich nach bereits existierenden Lösungen für vergleichbare Probleme erkundigen und diese nutzen.
- Sie sollten sich nach allgemeineren Problemen umschaun, und überprüfen, ob Ihr Problem als Spezialfall des allgemeinen Problems behandelt werden kann.
- Sie sollten versuchen, das Problem in einfachere Teilprobleme aufzuteilen. Wenn eine Aufteilung möglich ist, sollten Sie den hier skizzierten Programmentwicklungsprozeß zunächst für die einzelnen Teilprobleme anwenden und anschließend die Lösungen der einzelnen Teilprobleme zu einer Lösung für das Gesamtproblem zusammensetzen.

3.1.3 Implementierung

Der Entwurf eines Algorithmus sollte unabhängig von einer konkreten Programmiersprache erfolgen. Die anschließende Überführung des Algorithmus in ein in einer Programmiersprache verfaßtes Programm wird als *Implementierung* bezeichnet. Anders als der Entwurf eines Algorithmus ist die Implementierung in der Regel ein eher mechanischer Prozeß.

Die Implementierungsphase besteht selbst wieder aus zwei Teilphasen:

- Editieren: Zunächst wird der Programmcode mit Hilfe eines Editors eingegeben und in einer Datei dauerhaft abgespeichert.
- Compilieren: Anschließend wird der Programmcode mit Hilfe eines Compilers auf syntaktische Korrektheit überprüft und – falls keine Fehler vorhanden sind – in eine ausführbare Form (ausführbares Programm) überführt. Liefert der Compiler eine Fehlermeldung, so muß in die Editierphase zurückgesprungen werden.

Ist die Compilation erfolgreich, kann das erzeugte Programm ausgeführt werden. Je nach Sprache und Compiler ist die Ausführung entweder mit Hilfe des Betriebssystems durch den Rechner selbst oder aber durch die Benutzung eines Interpreters möglich.

3.1.4 Test

In der Testphase muß überprüft werden, ob das entwickelte Programm die Problemstellung korrekt und vollständig löst. Dazu wird das Programm mit verschiedenen Initialzuständen bzw. Eingabewerten ausgeführt und überprüft, ob es die erwarteten Ergebnisse liefert. Man kann eigentlich immer davon ausgehen, daß Programme nicht auf Anhieb korrekt sind, was zum einen an der hohen Komplexität des Programmentwicklungsprozesses und zum anderen an der hohen Präzision liegt, die die Formulierung von Programmen erfordert. Insbesondere die Einbeziehung von Randbedingungen wird von Programmieranfängern häufig vernachlässigt, so daß das Programm im Normalfall zwar korrekte Ergebnisse liefert, in Ausnahmefällen jedoch versagt.

Genauso wie der Algorithmusentwurf ist auch das Testen eine kreative Tätigkeit, die viel Erfahrung voraussetzt und darüber hinaus ausgesprochen zeitaufwendig ist. Im Durchschnitt werden ca. 40 % der Programmentwicklungszeit zum Testen und Korrigieren verwendet.

Auch durch noch so systematisches Testen ist es in der Regel nicht möglich, die Abwesenheit von Fehlern zu beweisen. Es kann nur die Existenz von Fehlern nachgewiesen werden. Aus der Korrektheit des Programms für bestimmte überprüfte Initialzustände bzw. Eingabewerte kann nicht auf die Korrektheit für alle möglichen Initialzustände bzw. Eingabewerte geschlossen werden.

Im folgenden werden ein paar Teststrategien vorgestellt:

- Das Testen sollte aus psychologischen Gründen möglichst nicht nur vom Programmierer selbst bzw. allein durchgeführt werden. Häufig werten Programmierer das Entdecken von Fehlern als persönlichen Mißerfolg und sind daher gar nicht daran interessiert, Fehler zu finden. Sie lassen daher die erforderliche Sorgfalt vermissen.
- Konstruieren Sie sogenannte Testmengen, das sind Mengen von möglichen Initialzuständen bzw. Eingabewerten für das Programm. Die Testmengen sollten dabei typische und untypische Initialzustände bzw. Eingabewerte enthalten. Auf jeden Fall müssen Grenzwerte berücksichtigt werden, das sind Werte, die gerade noch als Eingabewerte zugelassen sind, wie bspw. der kleinst- oder größtmögliche Wert bei Zahlen oder der leere Text bei Texteingaben.
- Überlegen Sie sich vor der Durchführung eines Testlaufs des Programms genau, welche Ergebnisse Sie erwarten.
- Überprüfen Sie nicht nur, ob das Programm das tut, was es soll, sondern auch, ob es etwas tut, was es nicht soll.
- Gehen Sie nach dem Finden und Korrigieren eines Fehlers nie davon aus, daß nun alle Fehler beseitigt sind.
- Wenn Sie einen Fehler gefunden und beseitigt haben, müssen alle vorherigen Testläufe nochmal wiederholt werden, um sich zu vergewissern, daß sich durch die Korrektur nicht neue Fehler ins Programm geschlichen haben.

Es werden drei Klassen von Fehlern, die im Programmentwicklungsprozeß auftreten können, unterschieden: syntaktische, logische und sogenannte Laufzeitfehler. Syntaktische Fehler werden

bereits in der Implementierungsphase durch den Compiler entdeckt und sind in der Testphase nicht mehr von Interesse. Während bei logischen Fehlern das Programm normal durchläuft aber falsche Ergebnisse liefert, äußert sich ein Laufzeitfehler dadurch, daß die Ausführung des Programms abgebrochen und in der Regel durch das Laufzeitsystem eine Fehlermeldung ausgegeben wird. Klassisches Beispiel eines Laufzeitfehlers ist die Division durch den Wert Null. Laufzeitfehler können nicht bereits durch den Compiler festgestellt werden.

3.1.5 Dokumentation

Parallel zu den eigentlichen Programmentwicklungsphasen sollten alle Ergebnisse dokumentiert, d.h. schriftlich festgehalten werden. Die Dokumentation besteht also aus einer exakten Problemstellung, einer verständlichen Beschreibung der generellen Lösungsidee und des entwickelten Algorithmus, dem Programmcode sowie einer Erläuterung der gewählten Testszenarien und Protokollen der durchgeführten Testläufe. Außerdem sollten weitergehende Erkenntnisse wie aufgetretene Probleme oder alternative Lösungsansätze in die Dokumentation aufgenommen werden.

Die Dokumentation dient dazu, daß andere Personen bzw. der Programmierer selbst auch zu späteren Zeitpunkten das Programm noch verstehen bzw. den Programmentwicklungsprozeß nachvollziehen können, um bspw. mögliche Erweiterungen oder Anpassungen vornehmen oder die Lösung bei der Bearbeitung vergleichbarer Probleme wiederverwenden zu können.

3.2 Entwicklungswerkzeuge

Zur Unterstützung der Programmentwicklung und -ausführung werden eine Menge von Hilfsprogrammen eingesetzt. Diese werden auch zusammengefaßt als Programmentwicklungswerkzeuge bezeichnet. Im folgenden werden die im Rahmen dieses Kurses benötigten Werkzeuge vorgestellt:

- Editoren dienen zum Erstellen bzw. Ändern des Programmcodes.
- Compiler oder Übersetzer sind Dienstprogramme, die den Programmcode auf syntaktische Korrektheit überprüfen und in eine andere (Programmier-)Sprache bzw. Codierung transformieren. Das zu übersetzende Programm wird dabei *Quellprogramm* und das generierte Programm *Zielprogramm* genannt. Beim Zielprogramm handelt es sich im allgemeinen um ein ausführbares Programm.
- Interpreter übersetzen Quellprogramme nicht erst ganzheitlich in anschließend ausführbare Zielprogramme, sondern untersuchen den Programmcode Anweisung für Anweisung auf syntaktische Korrektheit und führen die Anweisung anschließend direkt aus. Der Vorteil von Interpretern gegenüber Compilern ist, daß noch während der Ausführung Programmteile geändert werden können, wodurch das Testen wesentlich erleichtert wird. Der Nachteil ist jedoch die langsamere Ausführungszeit.
- Debugger werden in der Testphase eingesetzt. Sie unterstützen das Erkennen, Lokalisieren und Beseitigen von Fehlern. Der Debugger verfolgt schrittweise die Ausführung des Programms und erlaubt eine jederzeitige Unterbrechung des Programmablaufs durch den Tester. Bei einer Unterbrechung kann sich der Tester genauestens über den aktuellen Zustand des Programms informieren und ihn ggf. manipulieren.

- Dokumentationshilfen sind Programme, die den Programmierer bei der Erstellung der Dokumentation unterstützen. Häufig analysieren sie den Programmcode und stellen bestimmte Programnteile nach unterschiedlichen Kriterien übersichtlich gegliedert dar.
- Das Laufzeitsystem einer Programmiersprache wird durch eine Menge von Hilfsprogrammen gebildet, die automatisch zum übersetzten Programm hinzugebunden werden. Ein Laufzeitsystem ist insbesondere bei höheren Programmiersprachen notwendig, weil bestimmte Sprachelemente bei der Übersetzung nicht direkt in die Maschinensprache transformiert werden können. Stattdessen wird vom übersetzten Programm dann das entsprechende Hilfsprogramm aufgerufen. Auch die Behandlung von Laufzeitfehlern bspw. durch geeignete Fehlermeldungen bei der Programmausführung liegt im Zuständigkeitsbereich des Laufzeitsystems.
- Programmbibliotheken bilden Sammlungen von Programmen bzw. Teilprogrammen, die direkt aus dem Programmcode des zu erstellenden Programms aufgerufen werden können, ohne sie neu implementieren zu müssen. Sie enthalten implementierte Algorithmen, die bereits von (anderen) Programmierern entwickelt und zur Verfügung gestellt wurden.

Weitere Dienstprogramme des Betriebssystems, die in der Regel an der Ausführung bzw. der Vorbereitung der Ausführung eines Programms involviert sind, die jedoch selten vom Programmierer selbst aufgerufen werden müssen, sind der Binder und der Lader. Dem Binder kommt die Aufgabe zu, das Programm mit bereits vorübersetzten Programmen, die im Programmcode aufgerufen werden wie bspw. dem Laufzeitsystem zu einer Einheit zusammenzufassen. Der Lader sorgt beim Aufruf eines ausführbaren Programms für dessen korrekten Transport in den Hauptspeicher.

Kapitel 4

Computer

Computer – auch *Rechner* genannt – sind universell verwendbare Geräte zur automatischen Verarbeitung von Daten. Der Begriff *Computer* leitet sich aus dem Lateinischen „computare“ ab, was übersetzt „berechnen“ bedeutet. Computer können jedoch nicht nur zum Rechnen eingesetzt werden, sondern eignen sich auch zur Erledigung bzw. Unterstützung anderer Aufgaben und Tätigkeiten wie zur Textverarbeitung, Bilderkennung, Maschinensteuerung und vielem mehr.

Computersysteme setzen sich zusammen aus physikalischen Geräten (*Hardware*) sowie Programmen, die auf der Hardware ausgeführt werden (*Software*).

Zur Hardware gehören dabei der Computer an sich – auch *Zentraleinheit* genannt – sowie periphere Geräte zur Dateneingabe (Tastatur, Maus, Scanner, ...), Datenausgabe (Bildschirm, Drucker, ...) und dauerhaften Datenspeicherung (Magnetplattenspeicher, Diskettenlaufwerke, CD-ROM-Laufwerke, ...).

Bei der Software wird unterschieden zwischen der *System-* und *Anwendungssoftware*. Programme, die zur Steuerung und Verwaltung des Computers notwendig sind oder häufig erforderliche Dienstleistungen erbringen wie Compiler oder Editoren, gehören zur Systemsoftware, auch *Betriebssystem* genannt, während Programme zur Lösung spezieller Benutzerprobleme, wie sie bspw. im Rahmen dieses Kurses erstellt werden, zur Anwendungssoftware gezählt werden.

4.1 Arbeitsweise eines Computers

Vereinfacht dargestellt arbeitet ein Computer nach dem in Abbildung 4.1 (oben) skizzierten Prinzip.

Zunächst muß ihm in Form eines Programms eine Arbeitsanleitung zum Bearbeiten der zu erledigenden Aufgabe übergeben werden. Der Computer führt dann dieses Programm aus. Dabei fordert er im allgemeinen bestimmte Eingabewerte wie Zahlen oder Wörter an, die er gemäß der Anleitung in Ausgabewerte umwandelt. Abbildung 4.1 enthält im unteren Teil ein konkretes Beispiel, das die Arbeitsweise eines Computers demonstriert.

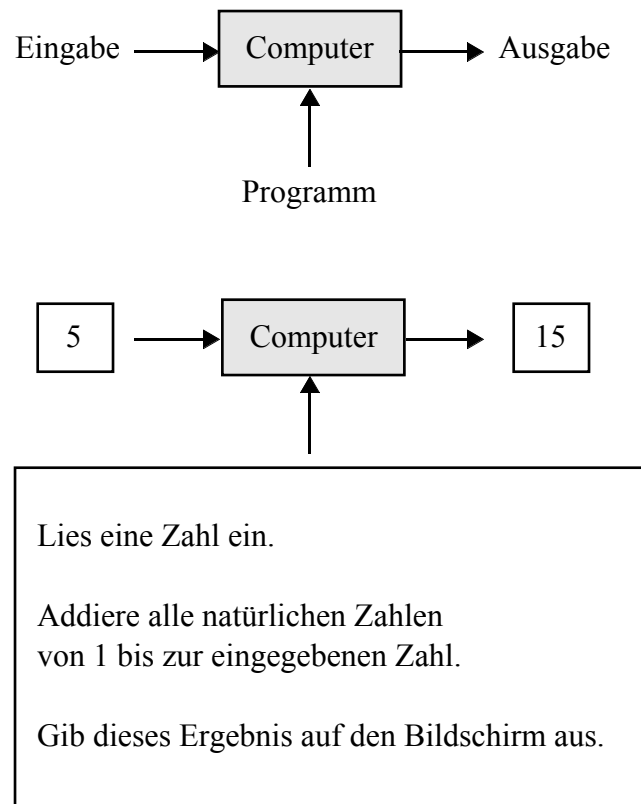


Abbildung 4.1: Arbeitsweise eines Computers

4.2 Aufbau eines Computers

Computer sind nicht alle identisch aufgebaut. Fast allen liegt jedoch die sogenannte *Von-Neumann-Rechnerarchitektur* zugrunde. Sie basiert auf einem Vorschlag von John von Neumann aus dem Jahre 1946. Nach diesem Vorschlag, der sich am biologischen Vorbild der menschlichen Informationsverarbeitung orientiert, bestehen Computer aus fünf Funktionseinheiten:

- Steuerwerk
- Rechenwerk
- Speicher
- Eingabewerk
- Ausgabewerk

Die Funktionseinheiten sind dabei miteinander verbunden. Ihr Zusammenspiel wird in Abbildung 4.2 erläutert.

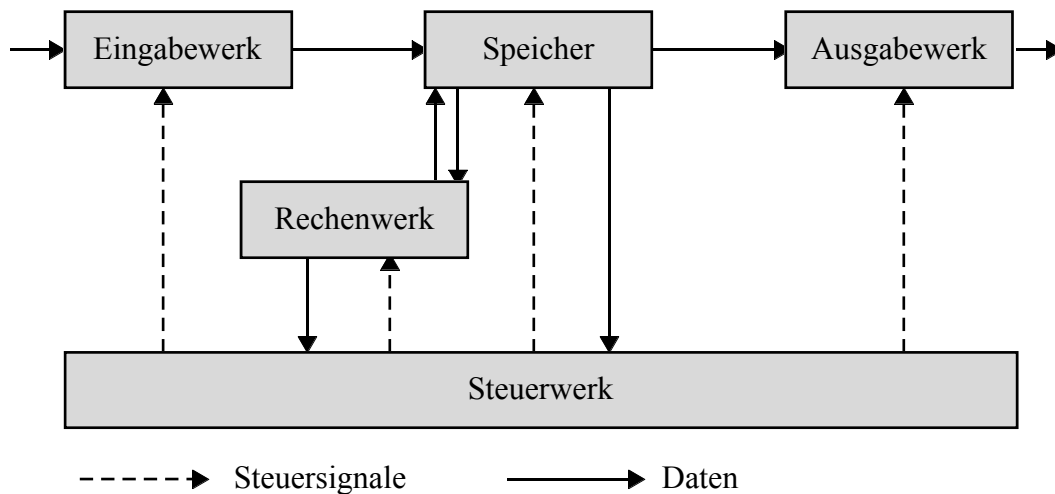


Abbildung 4.2: Von-Neumann-Rechnerarchitektur

4.2.1 Speicher

Der Speicher – auch Hauptspeicher genannt – ist die Rechnerkomponente zum Aufbewahren von auszuführenden Programmen und Daten, die bei der Ausführung benötigt bzw. berechnet werden. Speicher setzen sich aus vielen einzelnen Speicherelementen zusammen, die jeweils in der Lage sind, verschiedene Zustände anzunehmen. Heute werden fast ausschließlich sogenannte binäre Speicher mit zwei Zuständen 1 und 0 (Strom, kein Strom) eingesetzt. Ein Speicherelement speichert dann genau ein sogenanntes *Bit*. Um im Speicher abgelegt werden zu können, müssen Ihre Computerprogramme und die anfallenden Daten immer in eine Folge von Nullen und Einsen übersetzt werden. Darum müssen Sie sich jedoch nicht selbst kümmern. Diese Aufgabe übernehmen bereitgestellte Hilfsprogramme für Sie.

Der Ort im Speicher, an dem ein bestimmtes Datum abgelegt wird, wird seine *Adresse* genannt. Den Vorgang, den genauen Speicherplatz eines Datums zu finden und den Wert des gespeicherten Datums abzufragen oder zu verändern, wird als *Zugriff* bezeichnet.

4.2.2 Rechenwerk

Das Rechenwerk ist die Rechnerkomponente zum Ausführen von Operationen auf Daten. Dabei werden arithmetische und logische (boolesche) Operationen unterstützt. Arithmetische Operationen sind bspw. die Addition und Subtraktion von Zahlen, logische Operationen wie *und*, *oder*, *nicht* werden in Kapitel 5 genauer behandelt.

Das Rechenwerk besitzt verschiedene Einheiten – sogenannte *Register* – zum Zwischenspeichern der Operanden.

4.2.3 Eingabe- und Ausgabewerk

Eingabe- und Ausgabewerk bilden die Schnittstelle des Computers nach außen. Das Eingabewerk ist für die Eingabe von Daten bspw. über die Tastatur oder die Maus zuständig. Das

Ausgabewerk steuert die Ausgabe von Daten bspw. auf den Bildschirm oder den Drucker. Über die Eingabe- und Ausgabewerke wird auch der Zugriff auf den Hintergrundspeicher geregelt, auf dem der Programmcode und die ausführbaren Programme dauerhaft gespeichert werden.

4.2.4 Steuerwerk

Das Steuerwerk kann als das „Herz“ eines Computers bezeichnet werden. Es ist für die Gesamtsteuerung, d.h. die Koordination der anderen Komponenten zuständig. So teilt es bspw. dem Eingabewerk mit, an welche Adresse im Speicher bestimmte Eingabedaten abgelegt werden sollen, und informiert das Ausgabewerk darüber, bei welcher Adresse auszugebende Daten im Speicher zu finden sind.

Die Hauptaufgabe des Steuerwerks besteht in der Bearbeitung von Befehlen (Anweisungen) des auszuführenden Programms. Es holt dazu den aktuell zu bearbeitenden Befehl aus dem Speicher und interpretiert ihn. Handelt es sich bspw. um einen Additionsbefehl, dann organisiert das Steuerwerk zunächst das Laden der zwei Operanden aus dem Speicher in die Register des Rechenwerks. Anschließend teilt es dem Rechenwerk den Befehl mit und sorgt schließlich dafür, daß das berechnete Ergebnis an einer geeigneten Stelle im Speicher abgelegt wird.

Steuerwerk und Rechenwerk werden auch unter der Bezeichnung *Prozessor* zusammengefaßt.

4.3 Von-Neumann-Prinzipien der Rechnerarchitektur

Die wesentlichen Prinzipien der klassischen Von-Neumann-Rechnerarchitektur lassen sich in folgenden acht Punkten zusammenfassen:

- Ein Computer besteht aus fünf Funktionseinheiten: dem Steuerwerk, dem Rechenwerk, dem Speicher, dem Eingabewerk und dem Ausgabewerk.
- Die Struktur der Von-Neumann-Rechner ist unabhängig von den zu bearbeitenden Problemen. Zur Lösung eines Problems muß das Programm eingegeben und im Speicher abgelegt werden. Ohne Programme ist der Rechner nicht arbeitsfähig.
- Programme, Daten, Zwischen- und Endergebnisse werden im selben Speicher abgelegt.
- Der Speicher ist in gleichgroße Zellen unterteilt, die fortlaufend durchnummeriert sind. Über die Nummer (Adresse) einer Speicherzelle kann deren Inhalt gelesen oder verändert werden.
- Aufeinanderfolgende Befehle oder Anweisungen eines Programms werden in aufeinanderfolgende Speicherzellen abgelegt. Befehle werden vom Steuerwerk angesprochen. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
- Durch Sprungbefehle kann von der Bearbeitung der Befehle in der gespeicherten Reihenfolge abgewichen werden.

- Es existieren arithmetische Befehle wie Addition und Multiplikation, logische Befehle wie Vergleiche, Negation und Konjunktion, Transportbefehle z.B. zum Transportieren von Daten aus dem Speicher in das Rechenwerk, bedingte Sprünge sowie weitere Befehle wie Schiebeoperationen oder Ein-/Ausgabebefehle.
- Alle Daten (Befehle, Adressen) werden binär codiert. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (Decodierung).

4.4 Hintergrundspeicher

Der Hauptspeicher eines Computers muß in der Regel vom Prozessor sehr schnell zugreifbar sein und ist dementsprechend verhältnismäßig teuer. Zum langfristigen Speichern größer Datenmengen, die gerade nicht benötigt werden, können langsamere und billigere Speicher wie Magnetplattenspeicher, häufig auch als Festplatte bezeichnet, verwendet werden. Erst bei Bedarf werden die Daten aus diesem sogenannten *Hintergrundspeicher* in den Hauptspeicher geladen.

4.5 Betriebssystem

Als Betriebssystem oder Systemsoftware wird die Menge aller Programme eines Computersystems bezeichnet, die Routineaufgaben bewältigt und bestimmte zum Betrieb eines Rechners notwendige Verwaltungsaufgaben übernimmt. Zu diesem konkreten Aufgaben eines Betriebssystems gehören bspw. die Speicherverwaltung, die Prozessorverwaltung, die Geräteverwaltung, Sicherungsmaßnahmen und Zugriffskontrolle sowie die Kommunikation mit anderen Computersystemen. Weiterhin werden auch Compiler und andere Dienstleistungsprogramme wie Editoren, Binder und Lader zum Betriebssystem gezählt.

Ein Betriebssystem gehört in der Regel zur Grundausstattung eines Rechners. Es ist jedoch prinzipiell austauschbar. Bekannte Betriebssysteme sind MS-DOS, Windows 95, Windows NT und UNIX.

4.6 Dateien und Verzeichnisse

Dateien sind logische Behälter für Daten. Daten können dabei unterschiedlichen Typs sein (Text, Programme (Quellcode), ausführbare Programme, digitalisierte Bilder und Videos, ...). Dateien werden im allgemeinen im Hintergrundspeicher dauerhaft gespeichert. Ihre Verwaltung wird durch das Betriebssystem organisiert, das Möglichkeiten zur Manipulation von Dateien zur Verfügung stellt.

Verzeichnisse sind Hilfsmittel für eine übersichtliche Strukturierung von Dateien. In einem Verzeichnis werden im allgemeinen logisch zusammengehörende Dateien zusammengefaßt. Auch Verzeichnisse werden vom Betriebssystem verwaltet. Sie ermöglichen einem Benutzer in der Regel eine hierarchische Gliederung seiner Dateien.

4.7 Window-System

Window-Systeme werden heutzutage häufig den Betriebssystemen zugeordnet bzw. sind bereits in die Betriebssysteme integriert. Window-Systeme ermöglichen die Aufteilung des Bildschirm in mehrere Rechtecke, die sogenannten *Windows* (Fenster). Jedes Window spiegelt dabei im Prinzip einen eigenen kleinen Bildschirm wider, in dem verschiedene Programme laufen können.

Sogenannte *Window-Manager* sind spezielle Programme zur Verwaltung der Windows. Sie sorgen für eine korrekte Verteilung der Programmausgaben auf die einzelnen Windows und ermöglichen insbesondere das Anlegen, Löschen und Verändern von Windows.

Kapitel 5

Aussagenlogik

5.1 Aussagen

Eine *Aussage* – auch *boolescher Ausdruck* genannt – ist ein Satz, dem unmittelbar und eindeutig einer der Wahrheitswerte **wahr** (`true`, **T**) oder **falsch** (`false`, **F**) zugeordnet werden kann.

Bspw. bildet der Satz „Ein Tisch ist ein Möbelstück“ eine wahre Aussage, während es sich bei dem Satz „Geh nach Hause“ um keine Aussage handelt, da dem Satz kein Wahrheitswert zugeordnet werden kann.

5.2 Operationen auf Aussagen

Mit Hilfe sogenannter *logischer* oder *boolescher Operatoren* lassen sich die Wahrheitswerte von Aussagen verändern bzw. es lassen sich mehrere Aussagen miteinander verknüpfen. In der Programmierung sind dabei als boolesche Operatoren insbesondere die Negation (logische Verneinung), die Konjunktion (logisches „und“) und die Disjunktion (logisches „oder“) von Bedeutung.

5.2.1 Negation

Die Negation – im folgenden durch das Zeichen `!` repräsentiert – ist ein monadischer oder unärer Operator, d.h. sie besitzt nur einen Operanden (eine Aussage). Sie bewirkt eine Veränderung ihres Operanden derart, daß sich sein Wahrheitswert ändert. D.h. gegeben eine Aussage **P**. Besitzt **P** den Wahrheitswert **T**, dann besitzt die Aussage `!P` den Wahrheitswert **F**. Und entsprechend, besitzt **P** den Wahrheitswert **F**, dann besitzt die Aussage `!P` den Wahrheitswert **T**. `!P` ist selbst wieder eine Aussage, eine sogenannte *zusammengesetzte Aussage*.

5.2.2 Konjunktion

Die Konjunktion – im folgenden durch die Zeichenfolge `&&` ausgedrückt – ist ein dyadischer oder binärer Operator, d.h. sie benötigt zwei Operanden (Aussagen). Sie verknüpft ihre beiden Operanden derart, daß die konjugierte zusammengesetzte Aussage genau dann den Wahrheitswert

T besitzt, wenn beide Operanden den Wahrheitswert T besitzen. Besitzt einer der beiden Operanden – oder auch beide – den Wahrheitswert F, so besitzt auch die konjugierte Aussage den Wahrheitswert F.

5.2.3 Disjunktion

Die Disjunktion – im folgenden durch die Zeichenfolge $||$ ausgedrückt – ist wie die Konjunktion ein dyadischer Operator. Sie verknüpft ihre beiden Operanden (Aussagen) derart, daß die disjunctierte zusammengesetzte Aussage genau dann den Wahrheitswert F besitzt, wenn beide Operanden den Wahrheitswert F besitzen. Besitzt einer der beiden Operanden – oder auch beide – den Wahrheitswert T, so besitzt auch die konjugierte Aussage den Wahrheitswert T.

5.2.4 Wahrheitstabeln

Die Auswirkungen von booleschen Operatoren auf Aussagen können in sogenannten *Wahrheitstabeln* übersichtlich dargestellt werden. Dabei müssen immer alle Kombinationsmöglichkeiten für Wahrheitswerte ins Auge gefaßt werden. Abbildung 5.1 enthält die Wahrheitstabeln für die Negation, die Konjunktion und die Disjunktion. Dabei stehen P, Q und R in der Abbildung als Platzhalter für beliebige Aussagen.

P	!P	P	Q	P && Q	P	Q	P Q
T	F	T	T	T	T	T	T
F	T	T	F	F	T	F	T
		F	T	F	F	T	T
		F	F	F	F	F	F

Abbildung 5.1: Wahrheitstabeln für Negation, Konjunktion und Disjunktion

5.3 Syntax von Aussagen

Aus Aussagen können nun wiederum mit Hilfe der Operatoren immer komplexere zusammengesetzte Aussagen gebildet werden, in denen einfache oder zusammengesetzte Aussagen die Operanden bilden. Mit Hilfe von runden Klammern ist eine Schachtelung möglich. In Klammern gesetzte Aussagen werden dabei immer zuerst ausgewertet. Das Syntaxdiagramm in Abbildung 5.2 definiert, wie (komplexe) Aussagen bzw. boolesche Ausdrücke gebildet werden dürfen.

Gegeben seien die einfachen Aussagen P, Q und R. Dann sind bspw. folgende Zeichenfolgen syntaktisch korrekte boolesche Ausdrücke:

- P
- !P

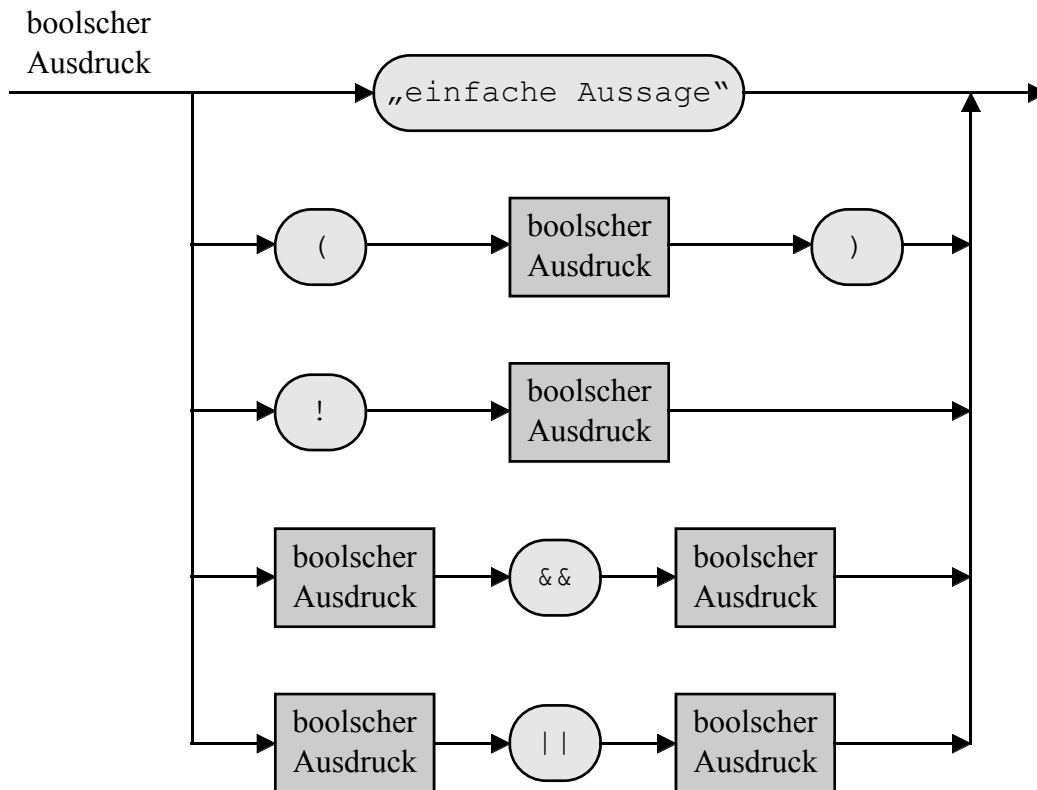


Abbildung 5.2: Syntaxdiagramm für boolesche Ausdrücke

- $P \ \&\& \ Q$
- $P \ || \ (!Q)$
- $(P \ || \ (!P \ \&\& \ Q))$
- $P \ || \ Q \ || \ R$
- $P \ || \ !(Q \ \&\& \ !R)$

5.4 Äquivalenz von Aussagen

Um boolesche Ausdrücke vergleichen zu können, wird der Begriff der *Äquivalenz von booleschen Ausdrücke* eingeführt. Es gilt: Zwei boolesche Ausdrücke sind äquivalent genau dann, wenn sie gleiche Wahrheitstabellen besitzen. Die Äquivalenz zweier boolescher Ausdrücke wird durch das Symbol \Leftrightarrow ausgedrückt. Seien P und Q Aussagen, dann sind bspw. die beiden booleschen Ausdrücke $(!P) \ \&\& \ (!Q)$ und $!(P \ || \ Q)$ äquivalent, wie die Wahrheitstabelle in Abbildung 5.3 beweist.

P	Q	!P	!Q	(!P) && (!Q)	P Q	!(P Q)
T	T	F	F	F	T	F
T	F	F	T	F	T	F
F	T	T	F	F	T	F
F	F	T	T	T	F	T

Abbildung 5.3: Äquivalente Aussagen

5.5 Algebraische Eigenschaften von booleschen Operatoren

5.5.1 Kommutativ- und Assoziativgesetz

Genauso wie bei den arithmetischen Operatoren der Addition und Multiplikation gilt auch bei den booleschen Operatoren $\&\&$ und $||$ das Kommutativ- und das Assoziativgesetz wie die Wahrheitstafeln in den Abbildungen 5.4 und 5.5 beweisen.

P	Q	P&&Q	Q&&P	P Q	Q P
T	T	T	T	T	T
T	F	F	F	T	T
F	T	F	F	T	T
F	F	F	F	F	F

Abbildung 5.4: Kommutativgesetz für boolesche Ausdrücke

5.5.2 Distributivgesetz

Des weiteren gelten für die beiden Operatoren die folgenden Distributivgesetze (P, Q und R seien Aussagen):

- $P \&\& (Q || R) \Leftrightarrow (P \&\& Q) || (P \&\& R)$
- $P || (Q \&\& R) \Leftrightarrow (P || Q) \&\& (P || R)$

Die Gültigkeit dieser Distributivgesetze geht aus den Wahrheitstafeln in Abbildung 5.6 hervor.

5.5.3 Priorität

Aus der Schule kennen Sie sicher die Regel „Punkt vor Strichrechnung“, die besagt, daß beim Rechnen die Multiplikation und Division eine höhere Priorität besitzen als die Addition und

P	Q	R	P&&Q	Q&&R	(P&&Q) &&R	P&&(Q&&R)
T	T	T	T	T	T	T
T	T	F	T	F	F	F
T	F	T	F	F	F	F
T	F	F	F	F	F	F
F	T	T	F	T	F	F
F	T	F	F	F	F	F
F	F	T	F	F	F	F
F	F	F	F	F	F	F

P	Q	R	P Q	Q R	(P Q) R	P (Q R)
T	T	T	T	T	T	T
T	T	F	T	T	T	T
T	F	T	T	T	T	T
T	F	F	T	F	T	T
F	T	T	T	T	T	T
F	T	F	T	T	T	T
F	F	T	F	T	T	T
F	F	F	F	F	F	F

Abbildung 5.5: Assoziativgesetz für boolesche Ausdrücke

Subtraktion. Eine derartige Regel gibt es auch für die booleschen Operatoren. Der Operator $!$ besitzt die höchste, der Operator $\&\&$ die zweithöchste und der Operator $||$ die niedrigste Priorität. Prioritäten kann man durch Kammersetzung beeinflussen. Das bedeutet bspw. für die vier Aussagen P, Q, R und S:

- $!P \ \&\& \ Q \ \Leftrightarrow \ (!P) \ \&\& \ Q$
- $P \ || \ Q \ \&\& \ R \ \Leftrightarrow \ P \ || \ (Q \ \&\& \ R)$
- $P \ || \ Q \ \&\& \ !R \ || \ S \ \Leftrightarrow \ (P \ || \ (Q \ \&\& \ (!R))) \ || \ S$

5.5.4 Tautologie und Widerspruch

Ein boolescher Ausdruck, der unabhängig vom Wahrheitswert der einzelnen Operanden immer den Wert T liefert, wird *Tautologie* genannt. Liefert ein boolescher Ausdruck immer den Wert F, so nennt man ihn *Widerspruch*. Wie Abbildung 5.7 zeigt, ist bspw. für eine Aussage P der boolesche Ausdruck $P \ \&\& \ (!P)$ ein Widerspruch und $P \ || \ (!P)$ eine Tautologie.

P	Q	R	$Q \mid \mid R$	$P \& \& (Q \mid \mid R)$	$P \& \& Q$	$P \& \& R$	$(P \& \& Q) \mid \mid (P \& \& R)$
T	T	T	T	T	T	T	T
T	T	F	T	T	T	F	T
T	F	T	T	T	F	T	T
T	F	F	F	F	F	F	F
F	T	T	T	F	F	F	F
F	T	F	T	F	F	F	F
F	F	T	T	F	F	F	F
F	F	F	F	F	F	F	F

P	Q	R	$Q \& \& R$	$P \mid \mid (Q \& \& R)$	$P \mid \mid Q$	$P \mid \mid R$	$(P \mid \mid Q) \& \& (P \mid \mid R)$
T	T	T	T	T	T	T	T
T	T	F	F	T	T	T	T
T	F	T	F	T	T	T	T
T	F	F	F	T	T	T	T
F	T	T	T	T	T	T	T
F	T	F	F	F	T	F	F
F	F	T	F	F	F	T	F
F	F	F	F	F	F	F	F

Abbildung 5.6: Distributivgesetze für boolesche Ausdrücke

P	$\neg P$	$P \& \& (\neg P)$	$P \mid \mid (\neg P)$
T	F	F	T
F	T	F	T

Abbildung 5.7: Tautologie und Widerspruch

Teil II

Imperative Programmierung

In diesem Teil des Buches werden die wesentlichen Sprachkonstrukte imperativer Programmiersprachen vorgestellt. Die gewählte Syntax lehnt sich dabei weitgehend an die Syntax der Programmiersprache Java an. Imperative Sprachen sind im wesentlichen dadurch gekennzeichnet, daß Programme aus Folgen von Befehlen bestehen. Eingabewerte werden in Variablen gespeichert und weiterverarbeitet. In den imperativen Programmiersprachen spiegelt sich deutlich die Architektur des Von-Neumann-Rechners wieder.

Dieser Teil des Buches ist leider noch nicht komplett fertiggestellt. Zur Zeit existieren lediglich die ersten sieben Kapitel. Trotzdem reichen die Inhalte dieser Kapitel dafür aus, erste größere Programme entwickeln zu können. Zunächst werden Sie in Kapitel 6 in die Grundlagen des Hamster-Modells eingeführt. Kapitel 7 stellt die vier Grundbefehle vor, die der Hamster kennt. Des weiteren wird auf die Zusammenfassung mehrerer Befehle zu Anweisungen und Programmen eingegangen. In Kapitel 8 wird demonstriert, wie Sie mit Hilfe der Definition von Prozeduren eigene Befehle kreieren können. Kapitel 9 beschäftigt sich mit drei Testbefehlen, die der Hamster kennt, und illustriert, wie diese Testbefehle eingesetzt werden können, um aus mehreren Anweisungsalternativen eine auszuwählen, die ausgeführt werden soll. Kapitel 10 stellt die Kontrollstruktur der Wiederholungsanweisung vor und in Kapitel 11 wird mit der Einführung boolescher Funktionen die Möglichkeit der Definition von Prozeduren erweitert um die Möglichkeit, auch eigene Testbefehle definieren zu können. Nachdem in den Kapitel 6 bis 11 die Grundlagen für die Entwicklung eigener Programme gelegt worden sind, beschäftigt sich Kapitel 12 mit einem Verfahren für eine systematische Entwicklung von Programmen.

In diesem zweiten Teil des Buches fehlen noch die Kapitel 13 bis 16. Kapitel 13 wird das Konzept der Variablen und Ausdrücke einführen, so daß der Hamster bestimmte Werte abspeichern und mit ihnen Rechnungen durchführen kann. Kapitel 14 verallgemeinert das Konzept der Prozeduren bzw. Funktionen und Kapitel 15 erweitert es um das Parameterkonzept. Kapitel 16 erläutert schließlich das Prinzip der Rekursion.

Kennern der imperativen Programmierung wird sicher auffallen, daß die Bildung komplexer Datenstrukturen durch Arrays und Verbünde in Teil II dieses Buches nicht behandelt wird, obwohl sie im Grunde genommen Teil der imperativen Programmierung ist. Grund hierfür ist der, daß in Java Arrays und Verbünde durch objektorientierte Sprachkonstrukte realisiert werden. Deshalb werden sie erst in Teil III des Buches behandelt.

Kapitel 6

Grundlagen des Hamster-Modells

6.1 Motivation

6.1.1 Machinensprachen

Wir haben in Kapitel 4 gelernt, was ein Computer ist und wie er prinzipiell funktioniert. Damit ein Computer arbeitet, müssen wir ihm Befehle mitteilen, die er ausführen kann. Diese Befehle fassen wir in Programmen zusammen. Beim Aufruf ausführbarer Programme führt der Computer die Befehle mit Hilfe des Prozessors in der angegebenen Reihenfolge aus. Anfallende Daten, wie Zwischen- oder Endergebnisse von Berechnungen legt er dabei im Speicher ab.

Der Befehlssatz eines Computers ist im allgemeinen nicht besonders umfangreich. Im großen und ganzen kann ein Computer lediglich arithmetische und logische Operationen durchführen. Prinzipiell ist es möglich, Programme direkt im Befehlssatz des Computers zu formulieren. Dies ist aber nicht besonders einfach und sehr fehlerträchtig. Programme, die direkt den Befehlssatz des Computers benutzen, heißen *Assembler-* oder *Maschinenprogramme*.

6.1.2 Höhere Programmiersprachen

Assembler- und Maschinensprachen orientieren sich sehr stark an den Eigenschaften der Computer. Diese entsprechen jedoch nicht der Art und Weise, wie wir Menschen im allgemeinen Probleme lösen bzw. Lösungsanweisungen formulieren. Aber das ist ja eigentlich genau der Grund, warum Computer überhaupt existieren: Sie sollen uns Menschen helfen, Probleme zu lösen. Wir stehen hier also vor dem Dilemma, daß auf der einen Seite wir Menschen bestimmte Probleme haben, die gelöst werden sollen, und auf der andere Seite der Computer zwar „bereit ist“, diese Probleme zu lösen, wir ihm aber Lösungsvorschriften (also Algorithmen) nicht mitteilen können, weil der Computer ganz anders „denkt“ bzw. funktioniert, als wir Menschen es gewohnt sind.

In den Anfängen des Computerzeitalters gab es daher nur wenige Experten, die überhaupt in der Lage waren, mit dem Computer zu arbeiten, die also „seine Sprache“ verstanden. Im Laufe der Zeit sind dann sogenannte *höhere Programmiersprachen* entwickelt worden, mit denen versucht wird, die Kluft zwischen der Arbeitsweise der Maschine und der Denkweise des Menschen zu verringern. Höhere Programmiersprachen sind problemorientiert, während Assembler-

bzw. Maschinensprachen maschinenorientiert sind. Das bedeutet, mit Hilfe höherer Programmiersprachen können menschliche Ideen und Konzepte zur Lösung von Problemen viel besser als Programme formuliert und dem Computer mitgeteilt werden, als dies mit Maschinensprachen möglich ist.

6.1.3 Compiler

Wie aber „versteht“ nun ein Computer Programme, die in einer höheren Programmiersprache formuliert sind? Ganz einfach: Es müssen Dolmetscher her, die die in einer höheren Programmiersprache formulierten Programme in äquivalente Maschinenprogramme übersetzen.

Prinzipiell könnte die Aufgabe des Dolmetschens von Menschen erledigt werden, nämlich von den oben angesprochenen Computerexperten. Das würde das Problem aber auch nicht lösen: Experten sind rar und nicht unmittelbar verfügbar. Von daher sind die Experten sehr schnell auf die Idee gekommen, als Dolmetscher selbst wieder die Computer zu nutzen. Sie haben (in Maschinensprache) Dolmetscherprogramme geschrieben, die sogenannten *Compiler*. Die Aufgabe eines Compilers besteht darin zu überprüfen, ob ein in einer höheren Programmiersprache formuliertes Programm korrekt ist. Falls dies der Fall ist, übersetzt der Compiler es in ein gleichbedeutendes Maschinenprogramm.

Der Begriff *korrekt* muß an dieser Stelle noch ein wenig präzisiert werden. Wozu ein Compiler fähig ist, ist die Überprüfung, ob das benutzte Vokabular und die Grammatikregeln der Programmiersprache eingehalten worden sind. Wozu er (leider) nicht fähig ist, ist die Überprüfung, ob das Programm auch das tut, was es soll, d.h. ein Compiler kann nicht überprüfen, ob das Programm das zu lösende Problem korrekt und vollständig löst. Diese Aufgabe bleibt immer noch dem Programmierer überlassen.

6.1.4 Programmiersprachen lernen

Durch die Definition höherer Programmiersprachen ist es heute nicht alleine Experten vorbehalten, Computer zum Bearbeiten und Lösen von Problemen zu nutzen. Sie werden sehen, nach dem Durcharbeiten dieses Kurses werden Sie dies auch schaffen.

Genauso wie Sie zum Beherrschen einer Fremdsprache wie Englisch oder Italienisch das Vokabular und die Grammatik dieser Sprache lernen müssen, müssen Sie auch zum Beherrschen einer Programmiersprache ihr Vokabular und die zugrundeliegenden Grammatikregeln lernen. Dabei ist das Vokabular einer Programmiersprache sehr viel geringer als das Vokabular einer Fremdsprache. Die Grammatik einer Programmiersprache ist jedoch sehr viel präziser als die Grammatik einer natürlichen Sprache. Hinzu kommt, daß zum Verständnis einer Programmiersprache bestimmte Konzepte erlernt werden müssen, die bei natürlichen Sprachen nicht existieren. Bei imperativen Programmiersprachen sind dies bspw. Variablen, Anweisungen und Prozeduren.

Stellen Sie sich vor, Sie haben sich ein englisches Wörterbuch gekauft und etwa 1000 Vokabeln und die wichtigsten Grammatikregeln gelernt. Sie fahren nach England und möchten sich mit einem Engländer unterhalten. Wie Sie sicher wissen, ist das nicht ganz so einfach möglich, Sie werden Ihre Probleme bekommen. Ganz wichtig beim Erlernen einer Fremdsprache ist das konsequente Üben. Erst nach und nach stellt sich ein Erfolgserlebnis ein. Alleine durch das bloße

Auswendiglernen von Vokabeln und Grammatikregeln schaffen Sie es nicht, mit einer Fremdsprache perfekt umgehen zu können. Sie müssen üben, üben, üben und Erfahrungen sammeln.

Dasselbe trifft auch für das Erlernen einer Programmiersprache zu. Nur durch konsequentes Üben werden Sie den korrekten Umgang mit der Programmiersprache erlernen. Als Hilfe können Sie dabei einen Compiler verwenden, der überprüft, ob Sie sich beim Formulieren eines Programmes an das Vokabular und die Grammatik der Programmiersprache gehalten haben oder nicht. Fehler in der Programmformulierung teilt Ihnen der Compiler in Form von Fehlermeldungen auf dem Bildschirm mit. Leider sind diese Fehlermeldungen nicht immer sehr präzise. Gerade am Anfang werden Sie Probleme haben, die Meldungen zu verstehen. Auch hier heißt es: üben, üben, üben und Erfahrungen damit sammeln. Bauen Sie anfangs ruhig auch mal absichtlich Fehler in Ihre Programme ein, und schauen Sie sich an, was Ihnen der Compiler dazu mitteilt.

6.1.5 Programmieren lernen

Häufig hört man Menschen, die sich ein Buch bspw. über die Programmiersprache Java gekauft und es durchgearbeitet haben, ganz stolz behaupten: Ich kann Java. Das mag sogar zutreffen, denn das Vokabular und die Grammatik von Java ist nicht besonders umfangreich. Was sie tatsächlich können, ist *syntaktisch korrekte Java-Programme schreiben*, was sie häufig jedoch leider nicht können, ist *mit Java zu programmieren*. Das Erlernen einer Programmiersprache ist in der Tat nicht besonders schwierig. Was sehr viel schwierig ist, ist das *Programmieren lernen*, d.h. das Erlernen des Programmentwicklungsprozeß:

- Wie komme ich von einem gegebenen Problem hin zu einem Programm, das das Problem korrekt und vollständig löst?
- Wie finde ich eine Lösungsidee bzw. einen Algorithmus, der das Problem löst?
- Wie setze ich den Algorithmus um in ein Programm?

Während das Erlernen einer Programmiersprache ein eher mechanischer Prozeß ist, bei dem die Verwendung eines Compilers helfen kann, ist die Programmentwicklung ein kreativer Prozeß, der Intelligenz voraussetzt. Computer besitzen keine Intelligenz, deshalb gibt es auch keine Programme, die hier weiterhelfen. An dieser Stelle sind Sie als Programmierer gefragt. Programmieren lernen bedeutet in noch stärkerem Maße als das Erlernen einer Programmiersprache: üben, üben, üben und Erfahrung sammeln. Schauen Sie sich Programme anderer Programmierer an und überlegen Sie: Wieso hat der das Problem so gelöst? Denken Sie sich selbst Probleme aus und versuchen Sie, hierfür Programme zu entwickeln. Fangen Sie mit einfachen Aufgaben an und steigern Sie nach und nach den Schwierigkeitsgrad. Ganz wichtig ist: Versuchen Sie Programmierpartner zu gewinnen, mit denen Sie Probleme und Lösungsansätze diskutieren können.

6.1.6 Sinn und Zweck des Hamster-Modells

Das Hamster-Modell ist ein einfaches Modell, bei dem nicht primär das Erlernen einer Programmiersprache sondern das Erlernen der Programmierung im Vordergrund steht, d.h. das Erlernen grundlegender Programmierkonzepte und das Erlernen des Problemlösungs- bzw. des

Programmentwicklungsprozesses. Der Lernprozeß wird im Hamster-Modell durch spielerische Elemente unterstützt. Der Hamster steuert einen *virtuellen Hamster* durch eine *virtuelle Landschaft* und läßt ihn bestimmte Aufgaben lösen.

Um den Programmieranfänger nicht zu überfordern, werden die gleichzeitig eingeführten Konzepte und Einzelheiten stark eingeschränkt und erst nach und nach erweitert. Aus diesem Grund ist dieser Kurs auch in derart viele Kapitel unterteilt. In jedem Kapitel wird ein einzelnes neues Konzept eingeführt und anhand vieler Beispiele erläutert. Es werden eine Reihe von Aufgaben gestellt, über die der Lernende motiviert werden soll zu üben. Die Wichtigkeit des Übens wurde bereits in den vergangenen Abschnitten mehrfach erwähnt und soll hier explizit nochmal herausgestellt werden: Programmieren können Sie nur durch üben lernen. Durch das Lösen der Übungsaufgaben und den Vergleich Ihrer Lösung mit den Musterlösungen im Anhang sammeln Sie Programmiererfahrung. Gehen Sie immer erst dann zum nächsten Kapitel über, wenn Sie alle Übungsaufgaben selbständig gelöst haben und sich absolut sicher sind, die in dem Kapitel eingeführten Konzepte nicht nur verstanden zu haben, sondern auch mit Ihnen umgehen und sie anwenden zu können, d.h. sie gezielt zum Lösen von Problemen einsetzen zu können.

Im Vordergrund des Hamster-Modells und dieses Kurses steht das „Learning-by-doing“ und nicht das „Learning-by-reading“ oder „Learning-by-listening“. Durch die zahlreichen vorgegebenen Übungsaufgaben wird das bekannte Anfängerproblem gelöst, daß Programmieranfänger zwar gerne programmieren üben wollen, ihnen aber keine passenden Aufgaben einfallen. Scheuen Sie sich auch nicht davor, sich selbst weitere Aufgaben zu überlegen und diese zu lösen.

Auch wenn die Programmiersprache selbst in diesem Kurs nicht im Vordergrund steht, zum Programmieren braucht man nun mal eine Programmiersprache. Deshalb werden Sie in diesem Kurs auch eine Programmiersprache erlernen, nämlich die sogenannte *Hamstersprache*. Die Hamstersprache wurde dabei gezielt ausgewählt bzw. definiert. Sie lehnt sich nämlich sehr stark an die Programmiersprache Java an. Java ist eine relativ junge Sprache, die auch als Sprache des Internet bezeichnet wird. Sie enthält viele wichtige Programmierkonzepte und scheint sich insbesondere im Zusammenhang mit dem rapiden Wachstum des Internets in vielen Bereichen – auch im industriellen Bereich – immer mehr durchzusetzen. Ich versichere Ihnen: Wenn Sie nach Ende des Kurses die Hamstersprache und die Entwicklung von Hamsterprogrammen beherrschen, werden Sie innerhalb weniger Stunden auch die Programmiersprache Java und das Entwickeln von Java-Programmen beherrschen.

6.2 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.

6.2.1 Landschaft

Die Welt, in der der Hamster lebt, wird durch eine gekachelte Ebene repräsentiert. Abbildung 6.1 zeigt eine typische Landschaftsskizze plus Legende. Die Größe der Landschaft, d.h. die Anzahl der Kacheln, ist dabei nicht explizit vorgegeben. Die Landschaft kann prinzipiell unendlich groß sein.

#	#	#	#	#	#	#	#	#	#	#	#	#			
#	o	o	#									#	<u>Symbol</u>	<u>Bedeutung</u>	
#	o	o	#					>				#	>	Hamster (Blickrichtung Ost)	
#	o	o	#	#	#							#	v	Hamster (Blickrichtung Süd)	
#								#	o	o	o	#	<	Hamster (Blickrichtung West)	
#								#	o	o	o	#	^	Hamster (Blickrichtung Nord)	
#								#	#	#	#	#	#	blockierte Kachel	
#	o											#	o	Kachel mit Körnern	
#				#	#	#						#			
#			#	o	o	o						o	#		
#	#	#	#	#	#	#	#	#	#	#	#	#			

Abbildung 6.1: 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 einzelnes Symbol `o` gekennzeichnet. Dabei sagt das `o` 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.

Auf den Kacheln der Hamster-Landschaft können weiterhin auch Mauern stehen, das 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.

6.2.2 Hamster

Im imperativen Hamster-Modell existiert immer genau ein Hamster. Der Hamster steht dabei auf einer der Kacheln der Hamster-Landschaft. 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 Zeichen 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.

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, wieviele, ist ebenfalls nicht direkt aus der Landschaftsskizze ersichtlich.

Mit Hilfe bestimmter Befehle, die im nächsten Kapitel (Kapitel 7) genauer erläutert werden, kann ein Programmierer den Hamster durch eine gegebene Hamster-Landschaft 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 Prozessoren (zunächst) keine arithmetischen und logischen Operationen ausführen kann, sondern in der Lage ist, mit einem kleinen Grundvorrat an Befehlen eine Hamster-Landschaft zu „erforschen“.

6.2.3 Hamsteraufgaben

Ihnen als Hamster-Programmierer werden nun bestimmte Aufgaben gestellt, die Sie durch die Steuerung des Hamsters durch eine Landschaft zu lösen haben. Diese Aufgaben werden im folgenden *Hamsteraufgaben* und die entstehenden Lösungsprogramme *Hamsterprogramme* genannt. Zusätzlich zu einer Aufgabe werden dabei zunächst bestimmte Hamsterlandschaften fest vorgegeben, in denen der Hamster die Aufgabe zu lösen hat.

Beispiel für eine Hamsteraufgabe: Gegeben sei die Landschaft in Abbildung 6.2. Der Hamster soll zwei beliebige Körner fressen.

#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#						v		o				#							#
#							#					#	#	#	#	#			#
#						o		o			o	#							#
#												#							#
#			#	#	#		#	#	#	#	#	#							#
#			#																#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 6.2: Hamsterlandschaft für Beispiel 1

Später – ab Kapitel 10 – werden die Landschaften zu einer Hamsteraufgabe nicht mehr fest vorgegeben. Sie werden dann nur noch durch vorgegebene Eigenschaften charakterisiert, d.h. der Hamster wird in der Lage sein, eine bestimmte Aufgabe in unterschiedlichen (aber gleichartigen) Landschaften zu lösen.

6.3 Grundlagen der Hamstersprache

6.3.1 Lexikalik

Der Zeichenvorrat, den Sie beim Erstellen von Hamsterprogrammen (und Java-Programmen) verwenden dürfen, entspricht dem 16-Bit-Zeichensatz *Unicode*. Sie brauchen an dieser Stelle nicht genau zu wissen, was das bedeutet. Alles was für Sie wichtig ist, ist daß Sie im Prinzip alle Zeichen benutzen dürfen, die Sie auf Ihrer Tastatur vorfinden.

6.3.2 Token

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 der Hamstersprache (und 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.

6.3.3 Bezeichner

Bezeichner, die zur Benennung von deklarierten Elementen (wie Prozeduren oder Variablen) verwendet werden, müssen in der Hamstersprache (und 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 der Hamstersprache (und in Java) wird streng zwischen Groß- und Kleinbuchstaben unterschieden, d.h. daß bspw. die Bezeichner `rechts` und `Rechts` unterschiedliche Bezeichner sind.

6.3.4 Schlüsselwörter

Schlüsselwörter der Hamster-Programmiersprache (und auch Java) sind reserviert, d.h. sie dürfen nicht als Bezeichner verwendet werden. Die folgende Zusammenstellung enthält alle Schlüsselwörter:

<code>abstract</code>	<code>double</code>	<code>int</code>	<code>static</code>
<code>boolean</code>	<code>else</code>	<code>interface</code>	<code>super</code>
<code>break</code>	<code>extends</code>	<code>long</code>	<code>switch</code>
<code>byte</code>	<code>final</code>	<code>native</code>	<code>synchronized</code>
<code>case</code>	<code>finally</code>	<code>new</code>	<code>this</code>
<code>catch</code>	<code>float</code>	<code>null</code>	<code>throw</code>
<code>char</code>	<code>for</code>	<code>package</code>	<code>throws</code>
<code>class</code>	<code>goto</code>	<code>private</code>	<code>transient</code>
<code>const</code>	<code>if</code>	<code>protected</code>	<code>try</code>
<code>continue</code>	<code>implements</code>	<code>public</code>	<code>void</code>
<code>default</code>	<code>import</code>	<code>return</code>	<code>volatile</code>
<code>do</code>	<code>instanceof</code>	<code>short</code>	<code>while</code>

Die booleschen Literale `true` und `false` sind formal keine Schlüsselwörter. Für sie gelten jedoch dieselben Einschränkungen.

Auch die vier Grundbefehle des Hamsters `vor`, `links_um`, `gib` und `nimm` und die drei Testbefehle `vorn_freie`, `maul_leer` und `korn_da` sowie das Wort `main` sollten Sie zunächst wie Schlüsselwörter behandeln.

Kapitel 7

Anweisungen und Programme

7.1 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()`;
- `links_um()`;
- `nimm()`;
- `gib()`;

7.1.1 Syntax

Die genaue Syntax der vier Grundbefehle des Hamster-Modells wird in Abbildung 7.1 dargestellt. Dabei ist unbedingt auf folgendes zu achten:

- Die Zeichenfolgen `vor`, `links_um`, `nimm` und `gib` dürfen nur Kleinbuchstaben enthalten. In der Hamstersprache werden Klein- und Großbuchstaben unterschieden.
- Die Zeichenfolgen `vor`, `links_um`, `nimm` und `gib` stellen jeweils ein Token dar, d.h. die Zeichenfolgen müssen immer als Ganzes auftreten, sie dürfen nicht durch Trennzeichen (Leerzeichen, Tabulator, Zeilenwechsel) unterbrochen werden.
- Vor, hinter und zwischen den runden Klammern können beliebig viele Trennzeichen stehen.
- Das Semikolon gehört zum Befehl dazu. Es darf nicht weggelassen werden.

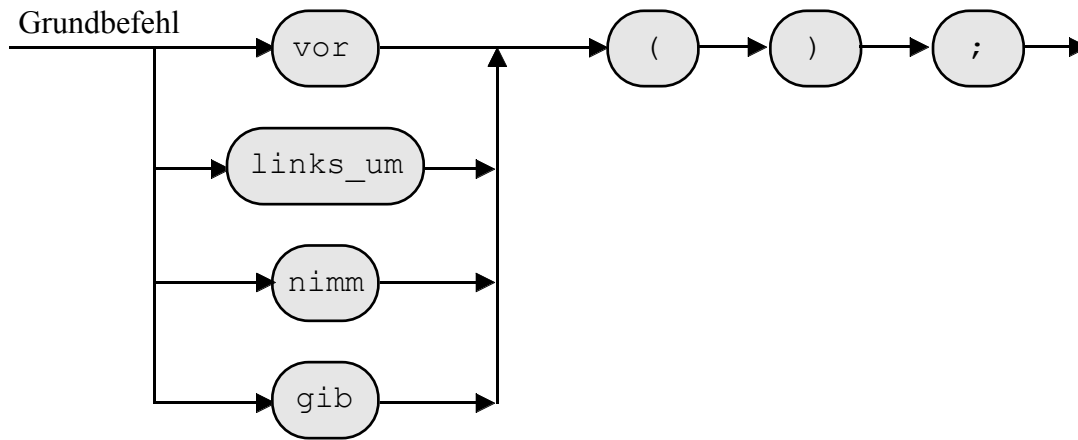


Abbildung 7.1: Syntaxdiagramm: Hamster-Grundbefehl

7.1.2 Semantik

Während die Syntax das Vokabular und die Grammatik einer Programmiersprache definiert, wird durch die Semantik die Bedeutung eines syntaktisch korrekten Programmes angegeben, d.h. es wird festgelegt, was das Programm bewirkt. Im folgenden wird dazu zunächst die Semantik der vier Grundbefehle des Hamster-Modells verbal beschrieben. Im Falle des Hamster-Modells bedeutet das, es wird definiert, wie der Hamster reagiert, wenn ihm ein Befehl “mitgeteilt” wird:

- **vor()** ;: Der Hamster hüpfte eine Kachel in seiner aktuellen Blickrichtung nach vorn.
- **links_um()** ;: Der Hamster dreht sich auf der Kachel, auf der er gerade steht, um 90 Grad nach links.
- **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! In Kapitel 9.1 werden sogenannte Testbefehle eingeführt, mit denen sich die angeführten Laufzeitfehler vermeiden lassen.

7.1.3 Beispiele

Folgender Befehl ist syntaktisch korrekt:

```
vor ( ) ;
```

Er ist auch semantisch korrekt bzgl. der in Abbildung 7.2 (links oben) dargestellten Situation. Nach seiner Ausführung ergibt sich die in Abbildung 7.2 (rechts oben) skizzierte Landschaft.

Der Befehl führt jedoch zu einem Laufzeitfehler, wenn die Landschaft in Abbildung 7.2 (links unten) die Situation skizziert, in der der Befehl ausgeführt wird.

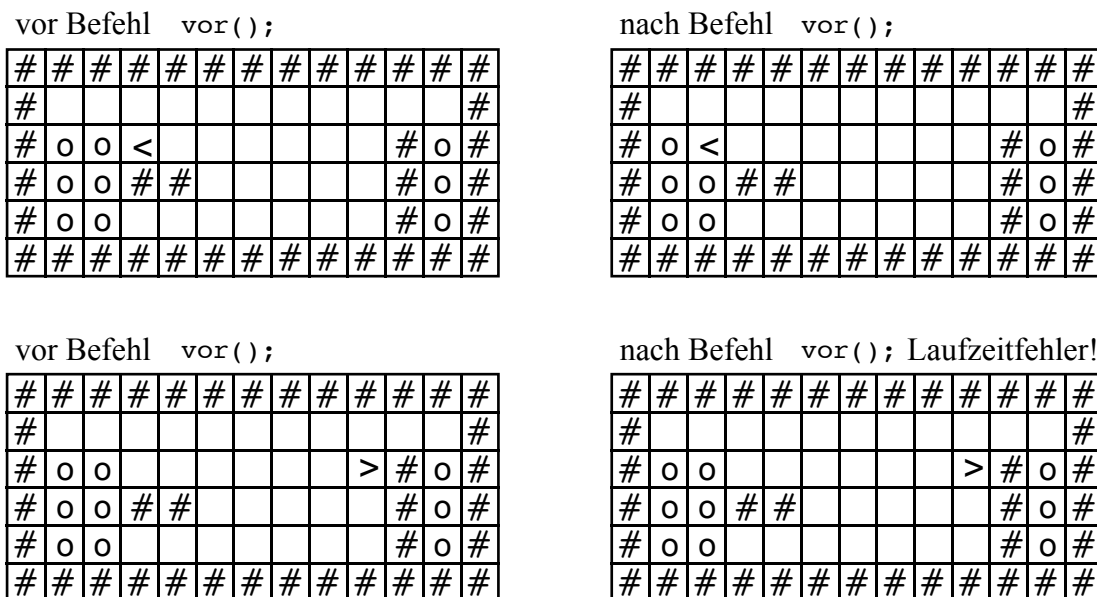


Abbildung 7.2: Auswirkung von Hamsterbefehlen

Syntaktisch nicht korrekt sind folgende Befehle:

```
n imm();
Gib();
links um();
vor()
```

7.2 Anweisungen

In imperativen Programmiersprachen werden einzelne Verarbeitungsvorschriften durch sogenannte *Anweisungen* ausgedrückt. Anweisungen, die nicht weiter zerlegt werden können, werden *elementare Anweisungen* genannt. In der Hamstersprache sind die vier Grundbefehle elementare Anweisungen. Eine Folge von Anweisungen, die nacheinander ausgeführt werden, wird als *Anweisungssequenz* bezeichnet.

7.2.1 Syntax

Die Syntax der Anweisungssequenz und Anweisung wird in Abbildung 7.3 durch Syntaxdiagramme definiert. Zwischen mehreren Anweisungen dürfen beliebig viele Trennzeichen stehen. Das Syntaxdiagramm für die Anweisung wird in späteren Kapiteln noch häufiger ergänzt.

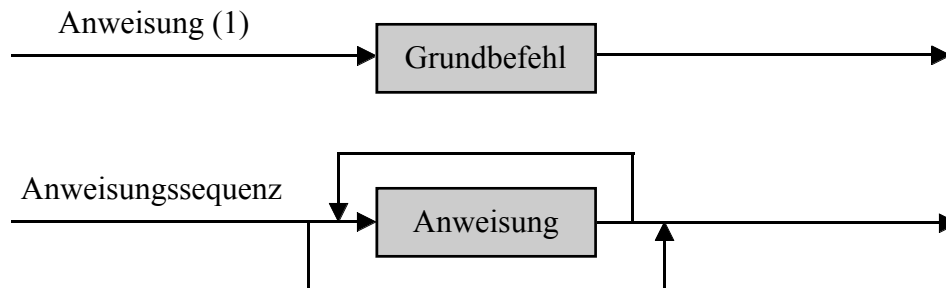


Abbildung 7.3: Syntaxdiagramm: Anweisung (1)

7.2.2 Semantik

Die einzelnen Anweisungen einer Anweisungssequenz werden in der angegebenen Reihenfolge hintereinander ausgeführt.

7.2.3 Beispiele

Das folgende Beispiel ist eine syntaktisch korrekte Anweisungssequenz:

```
vor();
links_um(); vor(); nimm(); vor(); gib();
vor(); links_um();
```

Die Anweisung bewirkt folgende Aktionen (siehe auch Abbildung 7.4): Der Hamster hüpfte zunächst eine Kachel in Blickrichtung nach vorne. Dann dreht er sich nach links um, geht in der neuen Blickrichtung wieder einen Schritt nach vorn, nimmt sich ein Korn, hüpfte noch eine Kachel weiter und legt das Korn wieder ab. Anschließend springt er wiederum eine Kachel nach vorne und dreht sich nach links.

Syntaktisch nicht korrekt ist das folgende Beispiel, weil `denke()`; kein Befehl und keine Anweisung ist:

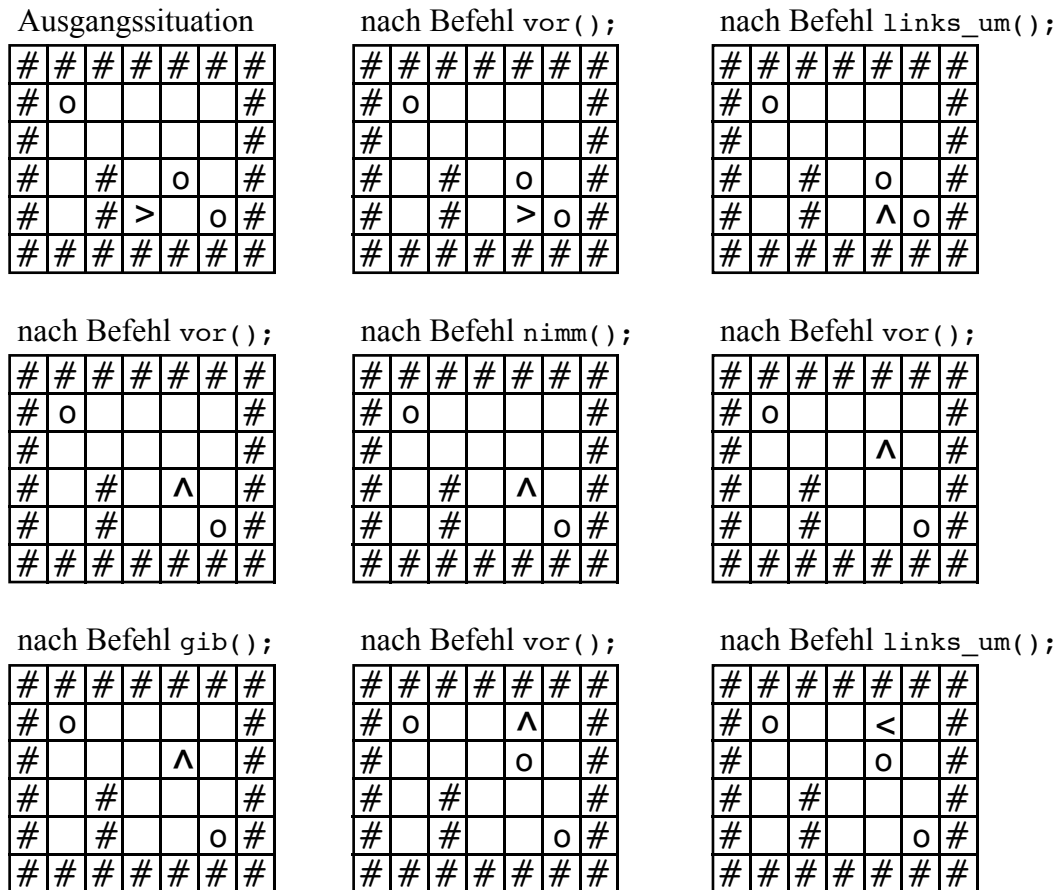


Abbildung 7.4: Auswirkung von Anweisungen

```
vor(); denke(); vor();
```

7.3 Programme

Nachdem wir nun Anweisungen kennengelernt haben, ist es nur noch ein kleiner Schritt zu definieren, was Hamster-Programme sind.

7.3.1 Syntax

Die Syntax eines Hamster-Programms wird in Abbildung 7.5 definiert. Danach setzt sich ein Hamsterprogramm aus den Schlüsselwörtern `void`, gefolgt von `main`, einem runden Klammerpaar und einem geschweiften Klammernpaar, das eine Anweisung umschließt, zusammen. Im allgemeinen wird es sich dabei bei der Anweisung um eine Anweisungssequenz handeln.

Programm (1)

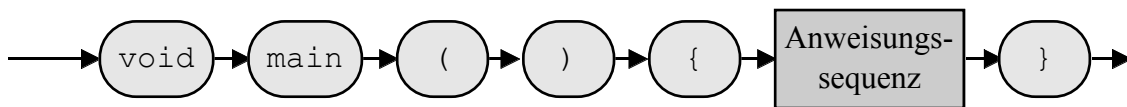


Abbildung 7.5: Syntaxdiagramm: Programm

7.3.2 Semantik

Beim Aufruf bzw. Start des Programms wird die Anweisung innerhalb der geschweiften Klammern ausgeführt.

Kümmern Sie sich zur Zeit noch nicht um die Bedeutung der anderen Bestandteile der Syntax. Sie werden in Kapitel 8.4 erläutert.

7.3.3 Beispiele

Das folgende Beispiel stellt ein syntaktisch korrektes Hamster-Programm dar:

```
void main()
{
  nimm(); vor(); gib();
}
```

Nach dem Start des Programms nimmt sich der Hamster von der Kachel, auf der er gerade steht ein Korn, hüpft anschließend eine Kachel in Blickrichtung nach vorn und legt das Korn wieder ab.

Syntaktisch nicht korrekt ist das folgende Hamster-Programm:

```
main:
{
  links_um(); links_um(); links_um();
}
```

Die Zeichenfolge `main:` ist syntaktisch nicht erlaubt. Ersetzen Sie sie durch die Zeichenfolge `void main()`. Dann ist das Programm syntaktisch korrekt und bei seinem Aufruf dreht sich der Hamster um 270 Grad gegen den Uhrzeigersinn.

7.4 Kommentare

Ziel der Hamster-Programmierung ist es, Hamster-Programme zu entwickeln, die gegebene Hamster-Aufgaben lösen. Neben ihren Eigenschaften, korrekt und vollständig zu sein, sollten

sich Hamster-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 sogenannte *Kommentare*. Sie haben auf die Steuerung des Hamsters keinerlei Auswirkungen. Alles, was sie bewirken, ist eine bessere Lesbarkeit des Programms. In der Hamstersprache gibt es zwei Typen von Kommentaren: *Zeilenkommentare* und *Bereichskommentare*.

7.4.1 Syntax

Die Syntax von Kommentaren wird in Abbildung 7.6 definiert.

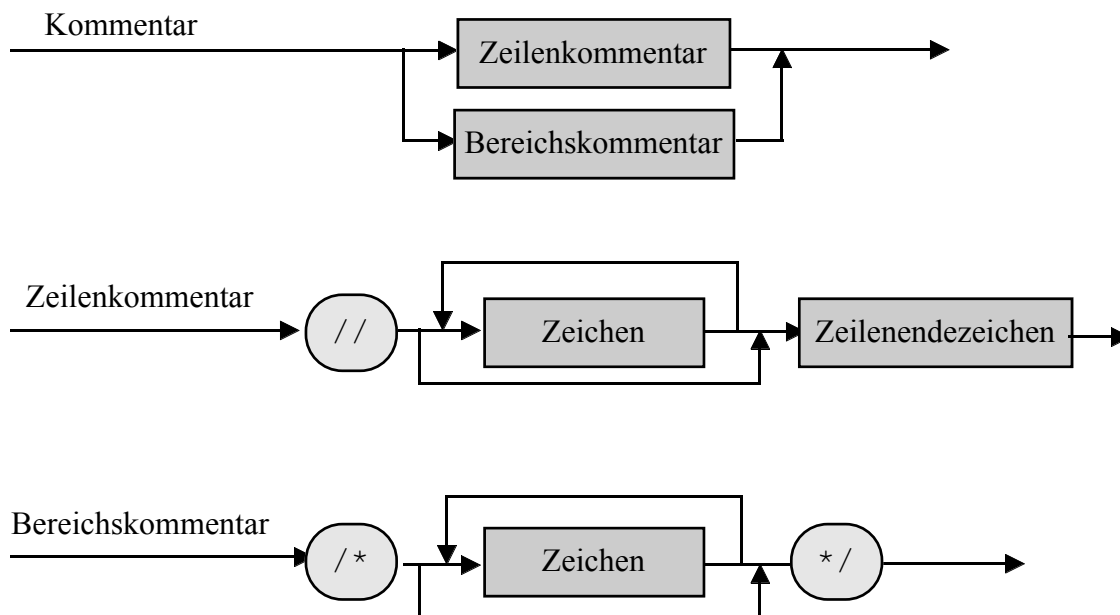


Abbildung 7.6: Syntaxdiagramm: Kommentar

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.

Kommentare können überall dort im Programm auftreten, wo auch Trennzeichen (Leerzeichen, Tabulatoren, Zeilenende) erlaubt sind.

7.4.2 Semantik

Kommentare haben für die Programmausführung keine Bedeutung. Sie bewirken keinerlei Zustandsänderung. In der Tat werden Kommentare (wie Trennzeichen übrigens auch) bereits während der lexikalischen Analyse der Compilation eines Programms entfernt, zählen also im Grunde genommen gar nicht mehr zur eigentlichen Syntax eines Programmes hinzu.

7.4.3 Beispiele

Das folgende Programm enthält einen korrekten Bereichskommentar:

```
void main()
{
    /* der Hamster soll sich einmal
       im Kreis drehen
    */
    links_um(); links_um(); links_um(); links_um();
}
```

Im folgenden Programm wird der Bereichskommentar aus dem obigen Beispiel durch einen Zeilenkommentar ersetzt:

```
void main()
{
    // der Hamster soll sich einmal im Kreis drehen
    links_um(); links_um(); links_um(); links_um();
}
```

Syntaktisch nicht korrekt ist folgendes Programm:

```
void main()
{
    // der Hamster soll sich /* einmal
       im Kreis drehen */
    links_um(); links_um(); links_um(); links_um();
}
```

Die Zeichenfolge `/*` ist Bestandteil des Zeilenkommentars. Deshalb leitet sie keinen Bereichskommentar ein.

Auch das nächste Programm enthält einen syntaktischen Fehler:

```
void main()
{
    /* der Hamster /* soll sich einmal */ im Kreis drehen */
    links_um(); links_um(); links_um(); links_um();
}
```

Bereichskommentare dürfen nämlich nicht geschachtelt werden. Bei der ersten Zeichenfolge `/*` ist der Bereichskommentar beendet, so daß die Zeichenfolge `im Kreis drehen */` nicht mehr zum Kommentar dazugehört. Die erste Zeichenfolge `/*` ist Bestandteil des Kommentars.

7.5 Beispielprogramme

In diesem Abschnitt werden einige Beispiele für Hamster-Aufgaben gegeben und eine oder mehrere Musterlösungen vorgestellt. Schauen Sie sich die Beispiele genau an und versuchen Sie, die Lösungen nachzuvollziehen.

7.5.1 Beispielprogramm 1

Aufgabe:

Gegeben sei das Hamster-Territorium in Abbildung 7.7. Der Hamster soll zwei Körner einsammeln.

#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#												#							#
#				v		o		#				#	#	#	#	#	#		#
#	#	#	#		#			#			o	#	o	o	o	o	o		#
#				o		o						#	#	#	#	#	#		#
#			#	#	#		#	#	#	#	#	#							#
#	o	o	#																#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 7.7: Hamsterlandschaft zu Beispielprogramm 1

Lösung 1:

```
void main()
{
    // nehme erstes Korn
    vor(); vor(); nimm();

    // nehme zweites Korn
    links_um(); vor(); vor(); nimm();
}
```

Lösung 2:

```
void main()
{
    // nehme erstes Korn
    links_um(); vor(); vor(); nimm();

    // nehme zweites Korn
    links_um(); links_um(); links_um();
    vor(); vor(); nimm();
}
```

7.5.2 Beispielprogramm 2

Aufgabe:

Gegeben sei das Hamster-Territorium in Abbildung 7.8 (links). Der Hamster habe vier Körner im Maul. Er soll in jeder Ecke des Territoriums eines ablegen und in seine Ausgangsposition zurückkehren. Nach Ausführung des Lösungsprogramms hat das Territorium das Erscheinungsbild in Abbildung 7.8 (rechts).

#	#	#	#	#	#	#	#
#							#
#			o	o			#
#		v	#	#	#		#
#							#
#	#	#	#	#	#	#	#

#	#	#	#	#	#	#	#
#	o					o	#
#			o	o			#
#		v	#	#	#		#
#	o					o	#
#	#	#	#	#	#	#	#

Abbildung 7.8: Hamsterlandschaft zu Beispielprogramm 2

Lösung:

```
void main()
{
    // begib dich an den Rand
    vor(); links_um();
    // laufe in die rechte untere Ecke
    vor(); vor(); vor(); vor(); gib(); links_um();
    // laufe in die rechte obere Ecke
    vor(); vor(); vor(); gib(); links_um();
    // laufe in die linke obere Ecke
    vor(); vor(); vor(); vor(); vor(); gib(); links_um();
    // laufe in die linke untere Ecke
    vor(); vor(); vor(); gib(); links_um();
    // begib dich in deine Ausgangsposition zurueck
    vor(); links_um(); vor(); links_um(); links_um();
}
```

7.5.3 Beispielprogramm 3

Aufgabe:

Der Hamster stehe vor einem Berg wie in Abbildung 7.9 skizziert. Der Hamster soll den Berg erklimmen.

Lösung:

```
void main()
{
    // laufe zum Berg
    vor();
```

#	#	#	#	#	#	#	#	#	#	#	#	#
#												#
#												#
#							#					#
#						#	#	#				#
#					#	#	#	#	#			#
#					#	#	#	#	#	#		#
#					#	#	#	#	#	#		#
#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 7.9: Hamsterlandschaft zu Beispielprogramm 3

```

// erklimme erste Stufe
links_um(); vor(); links_um(); links_um(); links_um(); vor();
// erklimme zweite Stufe
links_um(); vor(); links_um(); links_um(); links_um(); vor();
// erklimme dritte Stufe
links_um(); vor(); links_um(); links_um(); links_um(); vor();
// erklimme Gipfel
links_um(); vor(); links_um(); links_um(); links_um(); vor();
}

```

7.6 Übungsaufgaben

Nun sind Sie gefordert; denn in diesem Abschnitt werden Ihnen einige Hamster-Aufgaben gestellt, die sie selbständig zu lösen haben.

7.6.1 Aufgabe 1

Gegeben sei das Hamster-Territorium in Abbildung 7.10 (links). Dabei kann vorausgesetzt werden, daß auf allen Feldern, auf denen Körner eingezeichnet sind, jeweils genau zwei Körner liegen. Der Hamster soll alle Körner einsammeln. Nach Beendigung des Programms soll das Hamster-Territorium das in Abbildung 7.10 (rechts) skizzierte Erscheinungsbild besitzen.

#	#	#	#	#	#	#	#
#		o	o	o			#
#	v	#	#	#			#
#		o	o	o			#
#							#
#	#	#	#	#	#	#	#

#	#	#	#	#	#	#	#
#	<						#
#		#	#	#			#
#							#
#							#
#	#	#	#	#	#	#	#

Abbildung 7.10: Hamsterlandschaft zu Aufgabe 1

7.6.2 Aufgabe 2

Gegeben sei das Hamster-Territorium in Abbildung 7.11 (links). Der Hamster habe mindestens sechs Körner im Maul. Er soll auf allen für ihn erreichbaren Felder jeweils ein Korn ablegen und anschließend in seine Ausgangsposition zurückkehren, d.h. nach Beendigung des Programms soll das Hamster-Territorium das in Abbildung 7.11 (rechts) skizzierte Erscheinungsbild besitzen.

#	#	#	#	#	#	#	#
#			#			v	#
#				#			#
#					#		#
#						#	#
#	#	#	#	#	#	#	#

#	#	#	#	#	#	#	#
#			#	o	o	v	#
#				#	o	o	#
#					#	o	#
#						#	#
#	#	#	#	#	#	#	#

Abbildung 7.11: Hamsterlandschaft zu Aufgabe 2

7.6.3 Aufgabe 3

Gegeben sei das Hamster-Territorium in Abbildung 7.12. Der Hamster soll das Korn fressen.

#	#	#	#	#	#	#	#
#							#
#		#	#	#	#		#
#		#		o	#		#
#		#		#	#		#
#		#					#
#		#	#	#	#	#	#
#						<	#
#	#	#	#	#	#	#	#

Abbildung 7.12: Hamsterlandschaft zu Aufgabe 3

7.6.4 Aufgabe 4

Denken Sie sich selbst weitere Hamster-Aufgaben aus, und versuchen Sie, diese zu lösen. Viel Spaß!

Kapitel 8

Prozeduren

Das Konzept der Prozeduren ist eines der mächtigsten Konzepte imperativer Programmiersprachen. Wir werden in diesem Kapitel zunächst nur einen Teil dieses Konzeptes kennenlernen. In den Kapitel 14 Kapitel 15 und Kapitel 16 wird das Prozedurkonzept verallgemeinert und erweitert.

8.1 Motivation

Schauen Sie sich einmal das folgende Hamsterprogramm an: Der Hamster soll zwei Körner einsammeln. Die dazugehörige Landschaft wird in Abbildung 8.1 skizziert:

```
void main()
{
    vor(); vor(); nimm();
    links_um(); links_um(); links_um();
    vor(); vor();
    links_um(); links_um(); links_um();
    vor(); vor(); nimm();
}
```

Was an diesem Beispiel direkt auffällt, ist die Umständlichkeit für Sie als Programmierer, den Hamster um 90 Grad nach rechts zu drehen. Leider kennt der Hamster nur den Befehl `links_um()`; und nicht den Befehl `rechts_um()`;, so daß Sie jedesmal dreimal hintereinander den Befehl `links_um()`; ausführen müssen, um den Hamster nach rechts zu drehen. Schön wäre es, wenn wir dem Hamster einen neuen Befehl `rechts_um()`; beibringen könnten, indem wir ihm sagen: Wenn du diesen Befehl `rechts_um()`; erhältst, sollst du dreimal den Befehl `links_um()`; ausführen. Genau diesem Zweck, nämlich der Definition neuer Befehle auf der Grundlage bereits existierender Befehle und Anweisungen dient das Prozedurkonzept. Prozeduren werden manchmal auch *Unterprogramme* genannt.

Um zwei Dinge werden wir uns im folgenden kümmern: Wie werden Prozeduren und damit neue Befehle definiert und wie werden die neuen Befehle aufgerufen.

#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#						>	o				#								#
#			#	#	#	#	#				#	#		#		#		#	#
#			#			o					#	o		o		o			#
#			#								#								#
#	o	o	#	#	#		#				#	#	#	#	#	#			#
#	o	o	#	o							o		o	o	o	o	o	o	#
#			#										o	o	o	o	o	o	#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 8.1: Hamsterlandschaft zu Prozedur-Motivation

8.2 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 sogenannten *Prozedurkopf*, letzteres im sogenannten *Prozedurrumpf*.

8.2.1 Syntax

Die genaue Syntax einer Prozedurdefinition ist in Abbildung 8.2 definiert. Die Syntax wird in Kapitel 14 erweitert.

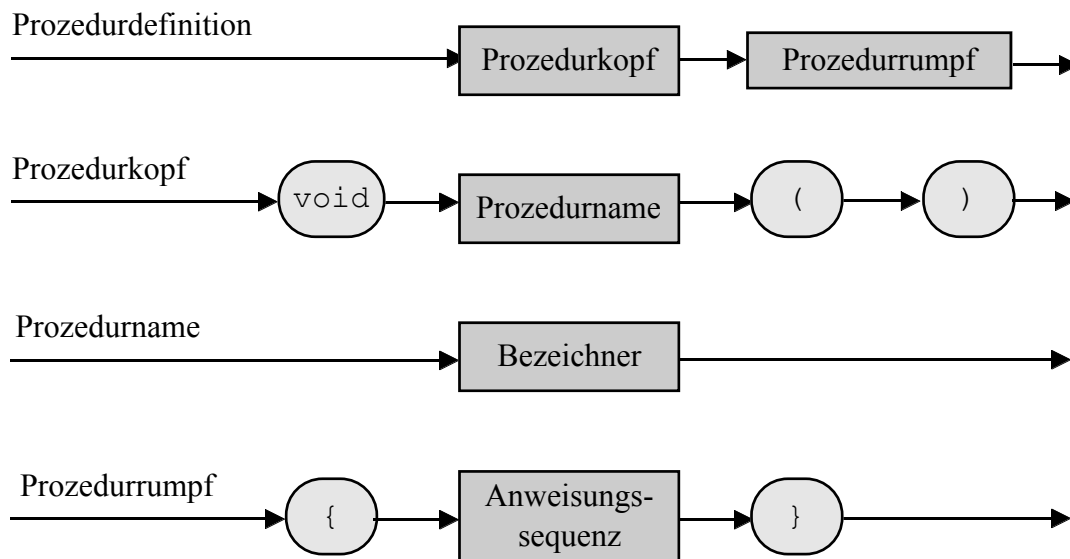


Abbildung 8.2: Syntaxdiagramm: Prozedurdefinition

Zunächst muß das Schlüsselwort `void` angegeben werden. Seine Bedeutung werden Sie in Kapitel 13 kennenlernen. Anschließend folgt ein Bezeichner (siehe auch Kapitel 6.3.3), der Prozedurname bzw. der Name des neuen Befehls. Nach dem Prozedurnamen folgt ein rundes Klammerpaar,

die den Prozedurkopf beendet. Der Prozedurrumpf beginnt mit einer öffnenden geschweiften Klammer. Er besteht aus einer Anweisung. Hierbei wird es sich im allgemeinen um eine zusammengesetzte Anweisung wie eine Sequenz handeln. Der Prozedurrumpf und damit die Prozedurdefinition endet mit einer schließenden geschweiften Klammer.

Beachten Sie folgendes: Prozedurnamen dürfen keine Schlüsselwörter sein. Außerdem müssen die Namen eindeutig sein, d.h. Sie dürfen nicht zwei Prozeduren gleich benennen. Insbesondere dürfen Sie als Prozedurnamen auch nicht die Befehlnamen der vier Grundbefehle verwenden.

Wählen Sie als Prozedurnamen immer aussagekräftige Bezeichner. Das erhöht die Lesbarkeit Ihrer Programme. Wenn Sie einen Befehl definieren, um den Hamster nach rechts zu drehen, dann nennen Sie den Befehl auch `rechts_um` und nicht `f1` oder `x2`.

Achten Sie bitte bei der Prozedurdefinition weiter auf eine gute Strukturierung und damit bessere Lesbarkeit. Wie Sie an den Beispielen unten sehen werden, können Sie zwischen die einzelnen Token wieder beliebige Trennzeichen einfügen. Der Prozedurkopf sollte möglichst in einer separaten Zeile stehen. Dasselbe gilt im allgemeinen für die einzelnen Anweisungen innerhalb des Prozedurrumpfes. Es empfiehlt sich weiterhin, die einzelnen Anweisungen des Rumpfes um zwei Spalten nach innen einzurücken und die öffnende und schließende Klammer des Prozedurrumpfes in dieselbe Spalte zu plazieren. Klammernpaare werden später noch für andere Zwecke verwendet, und man kann schnell den Überblick verlieren, ob es zu jeder öffnenden Klammer auch wieder eine schließende Klammer gibt.

8.2.2 Semantik

Durch eine Prozedurdefinition wird ein neuer Befehl vereinbart. Auf die Ausführung des Programmes hat das zunächst keinerlei Auswirkungen. Erst die Prozeduraufrufe, die im nächsten Abschnitt definiert werden, führen zu einer semantischen Beeinflussung des Programms.

8.2.3 Beispiele

Folgende Beispiele stellen gültige Prozedurdefinitionen dar:

```
void rechts_um()
{
    links_um();
    links_um();
    links_um();
}

void nimm_korn_und_lege_es_auf_naechstem_feld_wieder_ab()
{
    nimm();
    vor();
    gib();
}
```

Syntaktisch nicht korrekt sind folgende Beispiele:

```
void while() {
    vor();
}

2_vor()
{
    vor();
    vor();
}
```

Im ersten Beispiel wird als Prozedurname das Schlüsselwort `while` verwendet. Das zweite Beispiel enthält sogar zwei Fehler. Zunächst fehlt das Schlüsselwort `void`. Weiterhin ist `2_vor` kein gültiger Bezeichner, weil Bezeichner nicht mit Ziffern beginnen dürfen.

Das folgende Beispiel ist zwar syntaktisch korrekt, aber nicht besonders gut lesbar, weil zum einen ein schlechter Bezeichner gewählt wurde und zum anderen die Strukturierung zu wünschen übrig läßt:

```
void zwei_vor
()
{ links_um();
  vor();nimm();    vor();
  gib(); vor();}
```

8.3 Prozeduraufruf

Durch eine Prozedurdefinition wird ein neuer Befehl eingeführt. Ein Aufruf des neuen Befehls wird *Prozeduraufruf* genannt.

8.3.1 Syntax

Die Syntax eines Prozeduraufrufs ist in Abbildung 8.3 definiert.

Ein Prozeduraufruf entspricht syntaktisch dem Aufruf eines der vier Grundbefehle. Er beginnt mit dem Prozedurnamen. Anschließend folgen eine öffnende und eine schließende runde Klammer und ein Semikolon.

Ein Prozeduraufruf ist eine spezielle elementare Anweisung. Das Syntaxdiagramm aus Abbildung 7.3 muß also erweitert werden. Abbildung 8.3 enthält das neue Syntaxdiagramm für elementare Anweisungen. Prozeduraufrufe dürfen daher überall dort in Hamster-Programmen auftreten, wo auch andere Anweisungen wie die vier Grundbefehle stehen dürfen. Insbesondere können innerhalb von Prozedurrümpfen auch wieder (andere) Prozeduren aufgerufen werden. Achten Sie jedoch zunächst darauf, daß innerhalb eines Rumpfes einer Prozedur nicht die Prozedur

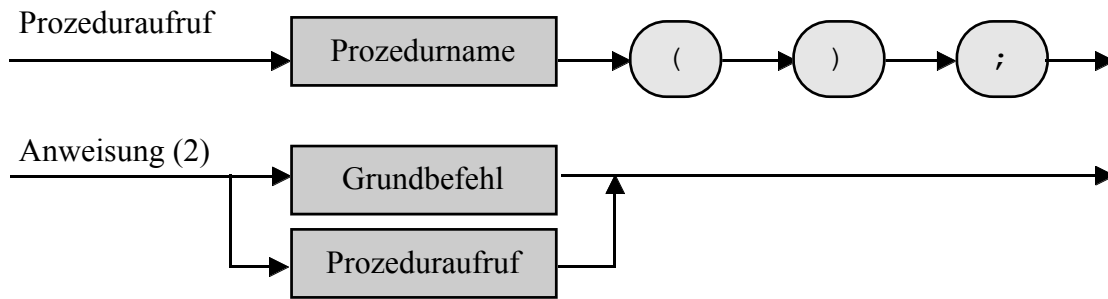


Abbildung 8.3: Syntaxdiagramm: Prozeduraufruf

selbst wieder aufgerufen wird. Prozeduren, die sich selbst aufrufen, werden *rekursive Prozeduren* genannt und in Kapitel 16 detailliert erläutert.

An dieser Stelle soll nochmal darauf hingewiesen werden, daß in der Hamstersprache Groß- und Kleinbuchstaben unterschieden werden. Das bedeutet insbesondere, daß wenn Sie eine Prozedur mit `rechts_um` benennen, Sie sie auch mit `rechts_um()`; aufrufen müssen. Der Aufruf der Anweisung `Rechts_Um()`; würde zu einer syntaktischen Fehlermeldung durch den Compiler führen, es sei denn, es ist noch eine weitere Prozedur namens `Rechts_Um` definiert.

8.3.2 Semantik

Im Prinzip entspricht ein Prozeduraufruf einem Platzhalter für den Prozedurrumpf, d.h. Sie können sich vorstellen, daß an der Stelle des Prozeduraufrufes die Anweisungen des entsprechenden Prozedurrumpfes stehen.

Eine alternative Erläuterung ist folgende: Wird irgendwo in einem Programm eine Prozedur aufgerufen, so wird bei der Ausführung des Programms an dieser Stelle der Rumpf der Prozedur, d.h. 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.

8.3.3 Beispiele

Folgendes Beispiel enthält gültige Prozedurdefinitionen für die Prozeduren `kehrt` und `rechts_um` und einen Prozeduraufruf der Prozedur `kehrt`:

```
void kehrt()
{
    links_um();
    links_um();
}
void rechts_um()
{
    kehrt();
}
```

```

links_um();
links_um();
}

```

8.4 Programme (mit Prozeduren)

Wir müssen an dieser Stelle die Definition eines Hamsterprogramms aus Kapitel 7.3 erweitern.

8.4.1 Syntax

Die nun gültige Syntax für ein Hamsterprogramm ist in Abbildung 8.4 dargestellt. In der Abbildung wird das Syntaxdiagramm „Programm“ aus Abbildung 7.5 erweitert.

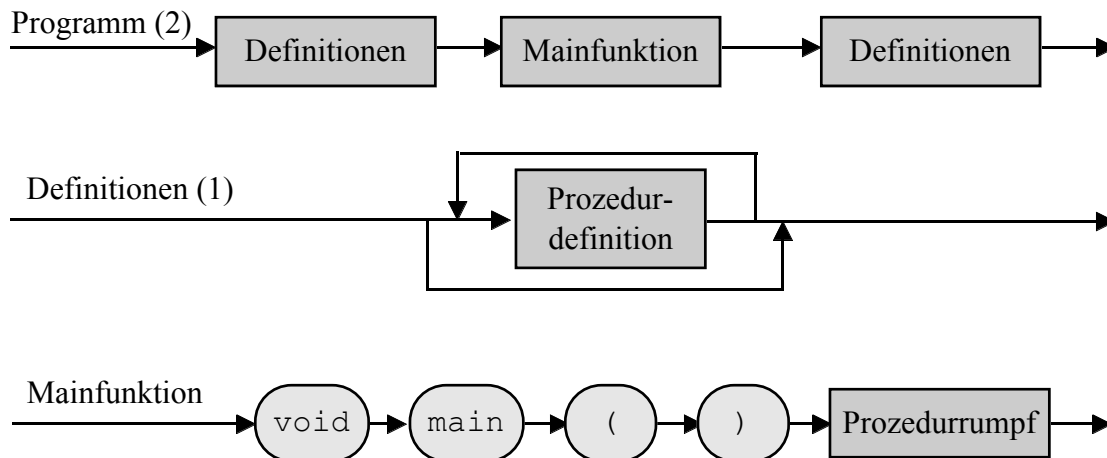


Abbildung 8.4: Syntaxdiagramm: Programm (2)

An dieser Stelle kann nun auch die Bedeutung des `main`-Teils erläutert werden. Wie Sie sicher schon festgestellt haben, handelt es sich auch hierbei um eine Prozedur, die sogenannte *main-Prozedur*. Dies ist eine besondere Prozedur. Sie wird automatisch beim Aufruf des Programms durch das Laufzeitsystem aufgerufen und darf niemals explizit durch den Programmierer aufgerufen werden.

Ein Hamsterprogramm besteht danach aus einer Menge von Prozedurdefinitionen, Dabei muß eine Prozedur den Namen `main` tragen. Die Namen der Prozeduren müssen paarweise disjunkt sein. Es dürfen innerhalb der Prozedurrümpfe nur Prozeduren aufgerufen werden, die auch definiert sind. Nicht aufgerufen werden darf jedoch die `main`-Prozedur. Der Ort einer Prozedurdefinition ist nicht festgelegt. Es spielt keine Rolle, ob die Prozedur vor oder nach einem Aufruf der Prozedur innerhalb eines Prozedurrumpfes definiert wird. Prozedurdefinitionen sind keine Anweisungen, d.h. es ist nicht erlaubt innerhalb einer Prozedurdefinition eine weitere Prozedur zu definieren, anders ausgedrückt: Prozedurdefinitionen dürfen nicht geschachtelt werden.

8.4.2 Semantik

Beim Aufruf eines Programmes wird implizit die Prozedur `main` aufgerufen. Es werden also die Anweisung(en) ausgeführt, die im Rumpf der `main`-Prozedur stehen. Nach der Ausführung der letzten Anweisung der `main`-Prozedur endet das Programm.

8.4.3 Beispiele

Das Programm aus Abschnitt 8.1 kann nun mit Hilfe von Prozeduren folgendermaßen umformuliert werden:

```
void main()
{
    vor(); vor(); nimm();
    rechts_um();
    vor(); vor();
    rechts_um();
    vor(); vor(); nimm();
}

void rechts_um()
{
    links_um();
    kehrt();
}

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

Wird das Programm aufgerufen, werden die einzelnen Anweisungen der `main`-Prozedur ausgeführt. Zunächst hüpfet der Hamster also aufgrund der `vor()`-Befehle zwei Felder in Blickrichtung nach vorne und nimmt ein Korn auf. Anschließend wird die Prozedur `rechts_um` aufgerufen, d.h. es werden die Anweisungen `links_um()` und `kehrt()` ausgeführt. Letztere Anweisung ist wiederum ein Prozeduraufruf, der dazu führt, daß zweimal der Befehl `links_um()` ausgeführt wird. Danach ist die Prozedur `kehrt` und ebenfalls die Prozedur `rechts_um` abgearbeitet, d.h. der Kontrollfluß des Programms befindet sich wieder in der `main`-Prozedur. Hier folgen zwei weitere `vor()`-Befehle und anschließend ein erneuter Aufruf der Prozedur `rechts_um`. Im Anschluß daran werden noch zwei weitere `vor()`-Befehle sowie ein `nimm()`-Befehl ausgeführt. Danach ist die `main`-Prozedur abgearbeitet und das Programm beendet.

Das folgende Beispiel enthält mehrere syntaktische Fehler:

```
void grase_feld_ab()
```

```

{
  void grase_reihe_ab()
  {
    vor();
    nimm();
  }
  vor();
  grase_reihe_ab();
  vor();
  grase_reihe_ab();
}

void kehre_zurueck()
{
  links_um();
  vor();
  rechts_um();
}

void start()
{
  grase_feld_ab();
  kehre_zurueck();
}

void kehre_zurueck()
{
  links_um();
  vor();
  rechts_um();
}

```

Zunächst fehlt die main-Prozedur. Des weiteren wird der Name `kehre_zurueck` für zwei Prozeduren verwendet. Dabei spielt es auch keine Rolle, daß beide Prozeduren identisch definiert sind. Ein dritter Fehler besteht darin, daß eine Prozedur `rechts_um` zwar aufgerufen, nicht jedoch definiert wird. Der vierte Fehler findet sich im Prozedurrumpf der Prozedur `grase_feld_ab`. Hier wird innerhalb des Prozedurrumpfes eine weitere Prozedur `grase_reihe_ab` definiert, was nicht erlaubt ist.

8.5 Vorteile von Prozeduren

Wie bereits anfangs erwähnt, ist das Prozedurkonzept eines der mächtigsten Konzepte imperativer Programmiersprachen. Prozeduren spielen beim Programmentwurf eine fundamentale Rolle. Die wichtigsten Eigenschaften und Vorteile von Prozeduren sind:

- bessere Übersichtlichkeit von Programmen,
- separate Lösung von Teilproblemen,
- Platzeinsparung,
- einfachere Fehlerbeseitigung,
- Flexibilität,
- Wiederverwendbarkeit.

Viele dieser Eigenschaften und Vorteile sind für Sie mit Ihren bisherigen Kenntnissen noch nicht unmittelbar ersichtlich. Sie werden die Vorteile jedoch in den nächsten Kapiteln, wenn die Programme größer werden, schätzen lernen.

8.5.1 Übersichtlichkeit

Durch die Nutzung von Prozeduren lassen sich Programme sehr viel übersichtlicher darstellen. Die Struktur des gewählten Lösungsalgorithmus ist besser ersichtlich. Eine Prozedur stellt eine zusammenhängende Einheit dar, die ein abgeschlossenes Teilproblem in sich löst. Der Name der Prozedur sollte möglichst aussagekräftig gewählt werden, d.h. aus dem Namen soll möglichst schon hervorgehen, was die Prozedur tut.

8.5.2 Lösung von Teilproblemen

Beim Programmwurf in Kapitel 12 werden wir sehen, daß es günstig ist, beim Lösen eines Problems dieses zunächst in Teilprobleme zu zerlegen, dann für die Teilprobleme Lösungsalgorithmen zu entwickeln und schließlich durch die Zusammenfassung der Lösungen der einzelnen Teilalgorithmen das Gesamtprobleme zu lösen. Prozeduren können dabei für die Implementierung der Teilalgorithmen eingesetzt werden. Das eigentliche Programm besteht dann (nur noch) aus Aufrufen der einzelnen Prozeduren.

8.5.3 Platzeinsparung

Besonders wertvoll sind Prozeduren, wenn dieselbe Prozedur von mehreren Programmstellen aus aufgerufen wird. Durch die Definition von Prozeduren kann hierdurch Platz bei der Formulierung von Programmen gespart werden.

8.5.4 Fehlerbeseitigung

Stellen Sie sich vor, Sie entdecken irgendwann einen logischen Fehler in einer Prozedur. Dann brauchen Sie ihn nur einmal im Prozedurrumpf zu beheben. Hätten Sie sich die Prozedur "gespart" und anstelle des Prozeduraufrufes jeweils die Anweisungen des Prozedurrumpfes an den entsprechenden Stellen explizit angeführt, dann müßten Sie den Fehler an allen diesen Stellen ändern. Dabei kann leicht auch mal eine Stelle übersehen werden, wodurch das Programm fehlerhaft bleibt bzw. wird.

8.5.5 Flexibilität und Wiederverwendbarkeit

Wie bereits erwähnt, sind in diesem Kapitel nur die fundamentalen Grundlagen des Prozedurkonzeptes eingeführt worden. In späteren Kapiteln wird das Konzept noch erweitert. In Kapitel 11 lernen Sie das *Funktionskonzept* kennen. Kapitel 14 verallgemeinert das Prozedur- und Funktionskonzept. Kapitel 15 führt sogenannte *Parameter* ein, durch die Prozeduren flexibler eingesetzt werden können. Schließlich werden Sie im dritten Teil dieses Kurses (Objektorientierte Programmierung) Methoden kennenlernen, die es ermöglichen, Prozeduren so zu definieren, daß sie von verschiedenen Programmen aus aufgerufen und auch anderen Programmierern direkt zur Verfügung gestellt werden können. Im Moment müssen Sie leider noch jede Prozedur, die in einem Programm aufgerufen wird, auch in diesem Programm definieren.

8.6 Beispielprogramme

In diesem Abschnitt werden einige Beispiele für Hamster-Aufgaben gegeben und eine oder mehrere Musterlösungen vorgestellt. Dabei werden Prozeduren eingesetzt. Schauen Sie sich die Beispiele genau an, und versuchen Sie, die Lösungen nachzuvollziehen.

8.6.1 Beispielprogramm 1

Aufgabe:

Gegeben sei das Hamster-Territorium in Abbildung 8.5. Auf den Kacheln, auf denen Körner liegen, liegt jeweils nur ein Korn. Der Hamster soll alle Körner einsammeln.

#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#								>		o		o		o					#
#		#	#						#		#		#						#
#			#					o		o		o							#
#			#						#		#		#						#
#			#	#	#		#												#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 8.5: Hamsterlandschaft zu Beispielprogramm 1

Lösung:

```
void main()
{
    ernte_reihe();
    rechts_um();
    zwei_vor();
    rechts_um();
    ernte_reihe();
}
```

```

void rechts_um()
{
    links_um(); links_um(); links_um();
}

void zwei_vor()
{
    vor(); vor();
}

void zwei_vor_und_nimm()
{
    zwei_vor(); nimm();
}

void ernte_reihe()
{
    zwei_vor_und_nimm();
    zwei_vor_und_nimm();
    zwei_vor_und_nimm();
}

```

8.6.2 Beispielprogramm 2

Aufgabe:

Gegeben sei das Hamster-Territorium in Abbildung 8.6. Der Hamster habe mindestens 31 Körner im Maul. Er soll auf allen Feldern jeweils genau ein Korn ablegen.

#	#	#	#	#	#	#	#	#	#	#
#										#
#		#	#	#	#	#	#	#		#
#										#
#		#	#	#	#	#	#	#		#
#	>									#
#	#	#	#	#	#	#	#	#	#	#

Abbildung 8.6: Hamsterlandschaft zu Beispielprogramm 2

Lösung:

```

void main()
{
    bearbeite_eine_reihe();
    // begib dich zur naechsten vollstaendigen Reihe
    links_um(); vor(); gib_und_vor(); links_um();
    bearbeite_eine_reihe();
    // bearbeite das Feld unter dir
}

```

```

    links_um(); vor(); gib(); kehrt(); vor();
    // begib dich zur naechsten vollstaendigen Reihe
    vor(); gib_und_vor(); rechts_um();
    bearbeite_eine_reihe();
}

void bearbeite_eine_reihe()
{
    gib_und_vor(); gib_und_vor(); gib_und_vor(); gib_und_vor();
    gib_und_vor(); gib_und_vor(); gib_und_vor(); gib_und_vor();
    gib();
}

void gib_und_vor()
{
    gib(); vor();
}

void kehrt()
{
    links_um(); links_um();
}

void rechts_um()
{
    kehrt(); links_um();
}

```

8.6.3 Beispielprogramm 3

Aufgabe:

Schauen Sie nochmal das Beispielprogramm 3 aus Kapitel 7.5.3 an: Der Hamster stehe vor einem Berg wie in Abbildung 7.9 skizziert; der Hamster soll den Berg erklimmen. Dort wurde eine Lösung mit eingestreuten Kommentaren gegeben. Diese Kommentare werden nun in diesem Abschnitt durch entsprechende Prozeduren ersetzt.

Lösung:

```

void main()
{
    laufe_zum_berg();
    erklimme_erste_stufe();
    erklimme_zweite_stufe();
    erklimme_dritte_stufe();
    erklimme_gipfel();
}

```

```
void erklimme_erste_stufe()
{
    erklimme_stufe();
}

void erklimme_zweite_stufe()
{
    erklimme_stufe();
}

void erklimme_dritte_stufe()
{
    erklimme_stufe();
}

void erklimme_gipfel()
{
    erklimme_stufe();
}

void erklimme_stufe()
{
    links_um(); vor();
    links_um(); links_um(); links_um(); vor();
}
```

8.7 Übungsaufgaben

Nun sind wieder Sie gefordert; denn in diesem Abschnitt werden Ihnen einige Hamster-Aufgaben gestellt, die sie selbständig zu lösen haben. Überlegen Sie jeweils, wo es sinnvoll bzw. nützlich ist, Prozeduren zu definieren und aufzurufen.

8.7.1 Aufgabe 1

Gegeben sei das Hamster-Territorium in Abbildung 8.7 (links). Der Hamster soll in allen Feldern der beiden Diagonalen jeweils genau ein Korn ablegen, so daß nach Beendigung des Programms das Hamster-Territorium das in Abbildung 8.7 (rechts) skizzierte Erscheinungsbild aufweist. Er habe anfangs mindestens 8 Körner im Maul.

8.7.2 Aufgabe 2

Die Aufgabe, die der Hamsters diesmal zu lösen hat, ist dieselbe wie in Aufgabe 1: Der Hamster soll in allen Feldern der beiden Diagonalen jeweils genau ein Korn ablegen. Nur diesmal sieht das Hamster-Territorium so aus wie in Abbildung 8.8 (links) skizziert. Der Hamster habe anfangs genau 9 Körner im Maul.

#	#	#	#	#	#
#					#
#					#
#					#
#	>				#
#	#	#	#	#	#

#	#	#	#	#	#
#	o			o	#
#		o	o		#
#		o	o		#
#	o	>		o	#
#	#	#	#	#	#

Abbildung 8.7: Hamsterlandschaft zu Aufgabe 1

#	#	#	#	#	#	#
#						#
#						#
#						#
#						#
#						#
#	>					#
#	#	#	#	#	#	#

#	#	#	#	#	#	#
#	o				o	#
#		o		o		#
#			o			#
#		o		o		#
#	o	>			o	#
#	#	#	#	#	#	#

Abbildung 8.8: Hamsterlandschaft zu Aufgabe 2

8.7.3 Aufgabe 3

Der Hamster hat Schiffbruch erlitten und ist auf einer einsamen Insel gestrandet. Er hat zum Glück noch 100 Körner dabei. Um Flugzeuge auf sich aufmerksam zu machen, will er aus den Körnern die Buchstaben SOS ablegen (siehe Abbildung 8.9) Helfen Sie ihm dabei. Der Hamster steht anfangs in der linken unteren Ecke des Territoriums mit Blickrichtung Ost.

#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#		o	o	o				o	o	o				o	o	o	#
#	o				o		o				o	o				o	#
#	o					o					o	o					#
#		o	o	o			o				o			o	o	o	#
#					o		o				o					o	#
#	o				o		o				o	o				o	#
#		o	o	o				o	o	o				o	o	o	#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 8.9: Hamsterlandschaft zu Aufgabe 3

8.7.4 Aufgabe 4

Denken Sie sich selbst weitere Hamster-Aufgaben aus, und versuchen Sie, diese zu lösen. Verwenden Sie dabei intensiv Prozeduren. Viel Spaß!

Kapitel 9

Auswahanweisungen

9.1 Test-Befehle

Mit dem Grundvorrat von vier Befehlen (`vor()`; , `links_um()`; , `gib()`; und `nimm()`;) sind Sie in der Lage, den Hamster über das Kornfeld zu bewegen und Körner aufnehmen bzw. ablegen zu lassen. Wir haben dabei gesehen, daß es Situationen gibt, die dem Hamster gar nicht gefallen:

- wenn der Hamster vor einer Mauer steht und Sie ihm den Befehl `vor()`; geben,
- wenn der Hamster keine Körner im Maul hat, er aber aufgrund Ihres Befehls `gib()`; eines ablegen soll und
- wenn der Hamster mittels des Befehls `nimm()`; ein Korn aufnehmen soll, sich aber auf dem Feld, auf dem er sich gerade befindet, gar keines liegt.

Wenn Sie den Hamster in diese für ihn unlösbaren Situationen bringen, dann ist der Hamster derart von Ihnen enttäuscht, daß er im folgenden nicht mehr bereit ist, weitere Befehle auszuführen. Um zu vermeiden, daß der Hamster in diese Situationen gelangt, werden nun drei sogenannte *Testbefehle* eingeführt. Testbefehle liefern boolesche Werte, also wahr (`true`) oder falsch (`false`):

- `vorn_frei()`
- `maul_leer()`
- `korn_da()`

9.1.1 Syntax

Die genaue Syntax der drei Testbefehle des Hamster-Modells wird in Abbildung 9.1 dargestellt. Im Unterschied zu den Grundbefehlen fehlt das abschließende Semikolon.

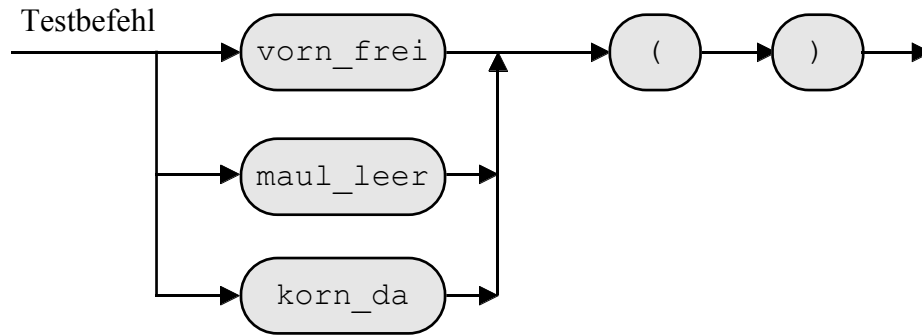


Abbildung 9.1: Syntaxdiagramm: Hamster-Testbefehl

9.1.2 Semantik

Die drei Testbefehle der Hamstersprache haben die folgende Bedeutung:

- `vorn_frei()`: Liefert den Wert `true`, falls sich auf der Kacheln in Blickrichtung vor dem Hamster keine Mauer befindet. Ist die Kachel durch eine Mauer blockiert, dann wird der Wert `false` geliefert.
- `maul_leer()`: Liefert den Wert `true`, falls der Hamster ein oder mehrere Körner im Maul hat. Befinden sich keine Körner im Maul des Hamsters, dann wird der Wert `false` geliefert.
- `korn_da()`: 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.

Beachten Sie, daß die Ausführung von Testbefehlen zunächst keine unmittelbare Auswirkung auf den Zustand des Kornfeldes hat.

9.1.3 Beispiele

Schauen Sie sich Abbildung 9.2 an. Wird dem Hamster in der in der Abbildung links dargestellten Situation der Testbefehl `vorn_frei()` gegeben, dann liefert der Testbefehl den booleschen Wert `true`. Dahingegen hat ein Aufruf des Testbefehls `vorn_frei()` in der im rechten Teil der Abbildung skizzierten Situation die Rückgabe des Wertes `false` zur Folge.

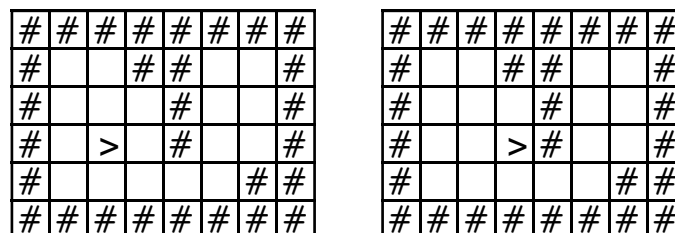


Abbildung 9.2: Testbefehle

9.1.4 Sinn und Zweck

Mit Hilfe der drei Testbefehle lassen sich die drei gefährlichen Situationen nun vorherbestimmen bzw. sie können vermieden werden:

- nur wenn die Kachel vor ihm frei ist, darf der Hamster nach vorne hüpfen, d.h. nur wenn der Testbefehl `vorn_frei()` den Wert `true` liefert, darf der Befehl `vor()`; ausgeführt werden,
- nur wenn der Hamster ein Korn im Maul hat, darf er auch eines ablegen, d.h. nur wenn der Testbefehl `maul_leer()` den Wert `false` liefert, darf der Befehl `gib()`; ausgeführt werden,
- nur wenn sich auf der Kachel, auf der der Hamster gerade steht, ein Korn befindet, darf er auch eines aufnehmen, d.h. nur wenn der Testbefehl `korn_da()` den Wert `true` liefert, darf der Befehl `nimm()`; ausgeführt werden.

Wie sich diese Sprachkonstrukte zum Abfragen einer bestimmten Situation in der Hamstersprache formulieren lassen, wird in Abschnitt 9.5 behandelt. Zuvor werden wir im nächsten Abschnitt lernen, wie sich Testbefehle mit Hilfe von booleschen Operatoren verknüpfen lassen. Des weiteren werden noch zwei neue Typen von Anweisungen eingeführt.

9.2 Boolesche Operatoren und Ausdrücke

In Kapitel 5 haben Sie die boolesche Logik kennengelernt. Sie wissen, was Aussagen bzw. boolesche Ausdrücke sind, daß Aussagen Wahrheitswerte liefern und wie sich Aussagen mit Hilfe der Konjunktion, Disjunktion und Negation verknüpfen lassen. Die drei Testbefehle `vorn_frei()`, `korn_da()` und `maul_leer()` stellen Aussagen in der Hamstersprache dar, d.h. abhängig von der Situation, in der sich der Hamster gerade befindet, liefern sie den Wert `true` oder `false`.

Darüber hinaus sind in diesem Zusammenhang die beiden Schlüsselwörter `true` und `false` der Hamstersprache von Bedeutung. Diese beiden sogenannten *booleschen Literale* repräsentieren spezielle boolesche Ausdrücke: Das boolesche Literal `true` liefert immer den Wahrheitswert `true`, das boolesche Literal `false` liefert immer den Wahrheitswert `false`.

Für die Konjunktion, Disjunktion und Negation von booleschen Ausdrücken, d.h. insbesondere der drei Testbefehle, stellt die Hamstersprache die folgenden drei booleschen Operatoren zur Verfügung:

- `!` für die Negation,
- `&&` für die Konjunktion und
- `||` für die Disjunktion,

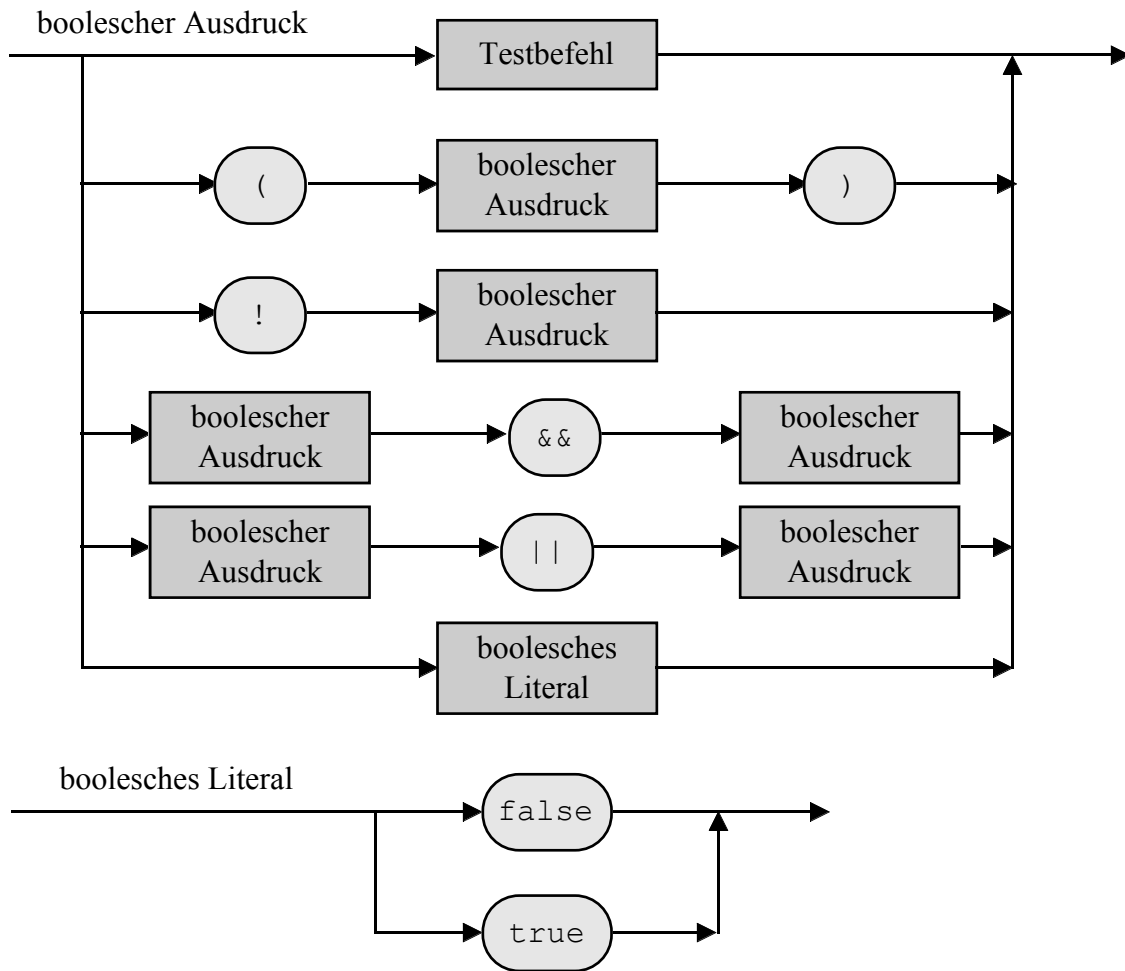


Abbildung 9.3: Syntaxdiagramm: boolescher Ausdruck

9.2.1 Syntax

Die genaue Syntax boolescher Operatoren und Ausdrücke der Hamstersprache wird in Abbildung 9.3 dargestellt.

Wie Sie in den Syntaxdiagrammen sehen, ist `!` ein monadischer Operator, `&&` und `||` sind dyadische Operatoren. Beachten Sie, daß die Zeichenfolgen `&&` und `||` nicht durch Trennzeichen unterbrochen werden dürfen. Beachten Sie weiterhin, daß auch die drei Testbefehle boolesche Ausdrücke darstellen.

9.2.2 Semantik

Die Ausführung boolescher Ausdrücke hat zunächst keine Auswirkungen auf den Zustand der aktuellen Situation. Boolesche Ausdrücke ermitteln ausschließlich Wahrheitswerte, und zwar gemäß der folgenden Regeln. Dabei seien `ba`, `ba1` und `ba2` jeweils Platzhalter für beliebige boolesche Ausdrücke:

- Der Operator `!` negiert den Wahrheitswert seines Operanden, d.h. er dreht ihn um. Liefert ein boolescher Ausdruck `bA` den Wert `true`, dann liefert `!bA` den Wert `false`. Umgekehrt, liefert `bA` den Wert `false`, dann liefert `!bA` den Wert `true`.
- Der Operator `&&` konjugiert den Wahrheitswert seiner beiden Operanden, d.h. er liefert genau dann den Wahrheitswert `true`, wenn beide Operanden den Wert `true` liefern. Liefern zwei boolesche Ausdrücke `bA1` und `bA2` beide den Wert `true`, dann liefert auch `bA1 && bA2` den Wert `true`. Liefert einer oder liefern beide Ausdrücke `bA1` oder `bA2` den Wert `false`, dann liefert `bA1 && bA2` den Wert `false`.
- Der Operator `||` disjungiert den Wahrheitswert seiner beiden Operanden, d.h. er liefert genau dann den Wahrheitswert `true`, wenn einer seiner Operanden oder seine beiden Operanden den Wert `true` liefern. Liefert einer der beiden booleschen Ausdrücke `bA1` und `bA2` oder liefern beide den Wert `true`, dann liefert auch `bA1 || bA2` den Wert `true`. Liefern beide Ausdrücke `bA1` oder `bA2` den Wert `false`, dann liefert `bA1 || bA2` den Wert `false`.
- Mit Hilfe der runden Klammern können Sie die Priorität der Operatoren beeinflussen. Ansonsten haben Klammern keine Bedeutung, was die Wertlieferung von booleschen Ausdrücken betrifft.

9.2.3 Beispiele

Beispiele für syntaktisch korrekte boolesche Ausdrücke sind:

1. `true` (boolesches Literal)
2. `vorn_frei()` (Testbefehl)
3. `!maul_leer()` (Negation)
4. `vorn_frei() && maul_leer()` (Konjunktion)
5. `vorn_frei() || korn_da()` (Disjunktion)

Schauen Sie sich nun Abbildung 9.4 (links) an. Auf jedem nicht durch eine Mauer blockiertem Feld liege mindestens ein Korn; der Hamster hat keine Körner im Maul. Bezüglich der Situation in Abbildung 9.4 (links) liefern die obigen Ausdrücke die folgenden Werte:

1. `true`
2. `true`
3. `false`
4. `true`
5. `true`

Bezüglich der in Abbildung 9.4 (rechts) dargestellten Situation (auf keinem Feld liegt ein Korn; der Hamster hat zwei Körner im Maul) liefern die obigen Ausdrücke die folgenden Werte:

#	#	#	#	#	#	#	#	#	#	#	#	#
#	0	0	0	0	0	0	0	0	0	0	0	#
#	0	0	0	0	#	#	#	#	0	0	0	#
#	0	0	0	0	0	0	0	0	0	0	0	#
#	0	>	0	0	#	#	#	#	0	0	0	#
#	0	0	0	0	0	0	0	0	0	0	0	#
#	#	#	#	#	#	#	#	#	#	#	#	#

#	#	#	#	#	#	#	#	#	#	#	#	#
#												#
#			#		#	#	#	#		#		#
#			#							#		#
#		>	#		#	#	#	#		#		#
#												#
#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 9.4: Auswirkung von booleschen Ausdrücken

1. true
2. false
3. true
4. false
5. false

Die Konjunktion, Disjunktion und Negation von Testbefehlen stellen selbst wieder boolesche Ausdrücke dar, d.h. auch folgende Konstrukte sind syntaktisch korrekte boolesche Ausdrücke, wie Sie anhand der Syntaxdiagramme in Abbildung 9.3 verifizieren können:

1. `vorn_frei() && !maul_leer()`
2. `!(vorn_frei() && korn_da())`
3. `vorn_frei() || !vorn_frei() && !maul_leer()`
4. `vorn_frei() && !korn_da() && maul_leer()`
5. `!!!vorn_frei() || (((maul_leer())))`

Bezüglich der in Abbildung 9.4 (links) bzw. (rechts) dargestellten Situation liefern diese Ausdrücke die folgenden Werte. Bevor Sie nachprüfen, ob die Lösungen korrekt sind, müssen Sie zunächst den folgenden Abschnitt über Eigenschaften der booleschen Operatoren lesen:

1. false (links) bzw. false (rechts)
2. false bzw. true
3. true bzw. true
4. false bzw. false
5. true bzw. true

9.2.4 Eigenschaften

9.2.4.1 Priorität

Von den drei booleschen Operatoren hat `!` die höchste, `&&` die zweithöchste und `||` die niedrigste Priorität. Durch Klammersetzung (runde Klammern) kann die Priorität beeinflusst werden, d.h. in runde Klammern gesetzte boolesche Ausdrücke werden immer zuerst berechnet.

Im booleschen Ausdruck `maul_leer() || !vorn_frei() && korn_da()` wird daher zunächst der Wert des Testbefehls `vorn_frei()` ermittelt. Dieser wird negiert. Anschließend wird dieser Wert mit dem Wert des Testbefehls `korn_da()` konjugiert und schließlich wird der Wert des Testbefehls `maul_leer()` mit dem konjugierten Wert disjungiert. Der boolesche Ausdruck ist also äquivalent mit dem Ausdruck `maul_leer() || ((!vorn_frei()) && korn_da())`.

Im booleschen Ausdruck `!(korn_da() && maul_leer())` wird aufgrund der Klammersetzung zunächst die Konjunktion der Werte der booleschen Ausdrücke `korn_da()` und `maul_leer()` ermittelt und dieser Wert dann negiert.

9.2.4.2 Assoziativität

Alle drei booleschen Operatoren sind linksassoziativ.

Der Ausdruck `korn_da() || vorn_frei() || maul_leer()` ist äquivalent zu `(korn_da() || vorn_frei()) || maul_leer()`, d.h. es wird zunächst der Wert von `korn_da()` ermittelt und disjunktiv mit dem Wert von `vorn_frei()` verknüpft. Dieser Wert wird anschließend mit dem Wert von `maul_leer()` disjungiert.

9.2.5 Auswertungsreihenfolge

Der Hamster ist ein sehr faules Lebewesen, d.h. er erspart sich unnötige Tests. Stellen Sie sich vor, der Hamster soll den Wert des booleschen Ausdrucks `vorn_frei() || korn_da()` ermitteln. Bisher sind wir davon ausgegangen, daß er zunächst den Wert des Testbefehls `vorn_frei()` und anschließend den Wert des Testbefehls `korn_da()` ermittelt und diese beiden Werte disjungiert. Stellen Sie sich nun vor, der Testbefehl `vorn_frei()` liefert den Wert `true`. Gemäß der Wahrheitstabelle liefert dann aber die Disjunktion unabhängig von dem Wert des Testbefehls `korn_da()` den Wert `true`. Also kann sich der Hamster die Ermittlung des Wertes des Testbefehls `korn_da()` ersparen, was er auch tut.

Ähnlich verhält es sich mit der Konjunktion. Liefert der Testbefehl `vorn_frei` den Wert `false`, dann kann sich der Hamster bei der Auswertung des Ausdrucks `vorn_frei() && (!korn_da() || maul_leer())` die Berechnung der anderen Operationen sparen, weil der boolesche Gesamtausdruck auf jeden Fall den Wert `false` liefert.

Im Moment spielt die Auswertungsreihenfolge boolescher Ausdrücke für Ihre Hamsterprogramme noch keine Rolle, aber in Kapitel 11.5 werden wir Situationen kennenlernen, wo der Beachtung dieser Auswertungsreihenfolge eine große Bedeutung zukommt. Insbesondere gilt daher in der Hamstersprache für boolesche Operatoren das Kommutativgesetz nur, was die Wertlieferung betrifft.

Zusammengefaßt lassen sich folgende Regeln definieren. Dabei seien p und q Platzhalter für beliebige boolesche Ausdrücke:

- Im Ausdruck $p \ \&\& \ q$ wird der Wert des Teilausdrucks q nur dann ermittelt, wenn der Teilausdruck p den Wert `true` liefert. In diesem Fall liefert der Gesamtausdruck den Wert von q . Liefert p den Wert `false`, dann liefert der Gesamtausdruck unmittelbar den Wert `false`.
- Im Ausdruck $p \ || \ q$ wird der Wert des Teilausdrucks q nur dann ermittelt, wenn der Teilausdruck p den Wert `false` liefert. In diesem Fall liefert der Gesamtausdruck den Wert von q . Liefert p den Wert `true`, dann liefert der Gesamtausdruck unmittelbar den Wert `true`.

9.3 Blockanweisung

Mit Hilfe der Blockanweisung lassen sich mehrere Anweisungen zu einer Einheit zusammenfassen. In Kapitel 13 werden weitere Eigenschaften der Blockanweisung angeführt.

Wir haben die Blockanweisung bereits in Kapitel 8 kennengelernt. Prozedurrümpfe werden nämlich immer durch eine Blockanweisung gebildet. Sie faßt die Anweisungen zusammen, die beim Aufruf der Prozedur ausgeführt werden sollen.

9.3.1 Syntax

Syntaktisch gesehen, handelt es sich bei einer Blockanweisung um eine zusammengesetzte Anweisung. Innerhalb von geschweiften Klammern steht eine andere Anweisung – im allgemeinen eine Anweisungssequenz. Abbildung 9.5 skizziert die genaue Syntax der Blockanweisung und erweitert das Syntaxdiagramm „Anweisung“ aus Abbildung 7.3.

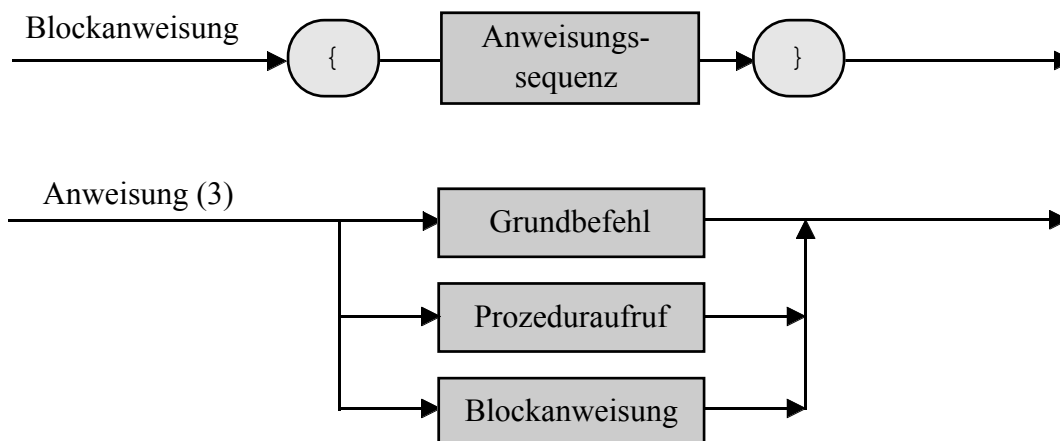


Abbildung 9.5: Syntaxdiagramm: Blockanweisung

9.3.2 Semantik

Beim Ausführen einer Blockanweisung wird die innerhalb der geschweiften Klammern stehende Anweisung ausgeführt. Eine weitere Auswirkung auf den Programmablauf hat die Blockanweisung zunächst nicht.

9.3.3 Beispiele

Das folgende Beispiel zeigt ein Hamsterprogramm, in dem Blockanweisungen zur Strukturierung eingesetzt werden.

```
void main()
{
  links_um();
  {
    laufe_zur_wand();
    links_um();
  }
  { // vier vor
    { // zwei vor
      vor(); vor();
    }
    { // zwei vor
      vor(); vor();
    }
  }
  links_um();
  {
    laufe_zur_wand();
    links_um();
  }
}

void laufe_zur_wand()
{
  vor(); vor(); vor();
}
```

Für eine bessere Übersichtlichkeit sollten Sie sich angewöhnen, die öffnende und die schließende geschweifte Klammer einer Blockanweisung in derselben Spalte zu platzieren und die (inneren) Anweisungen der Blockanweisung um zwei Spalten nach rechts einzurücken, wie im obigen Beispiel. Blockanweisungen werden später sehr häufig benutzt und treten auch ineinander verschachtelt auf. Plaziert man die zusammengehörenden Klammern untereinander, so kann man auf einen Blick feststellen, an welcher Stelle eine Blockanweisung beendet ist.

9.4 Leeranweisung

Wir werden später Situationen kennenlernen, in denen es ganz nützlich ist, Anweisungen zur Verfügung zu haben, die nichts tun bzw. bewirken. Derartige Anweisungen sind die Leeranweisungen.

9.4.1 Syntax

Syntaktisch werden Leeranweisungen durch ein einzelnes Semikolon gebildet. Abbildung 9.6 stellt die Syntax der Leeranweisung dar und erweitert das Syntaxdiagramm „Anweisung“ aus Abbildung 9.5.

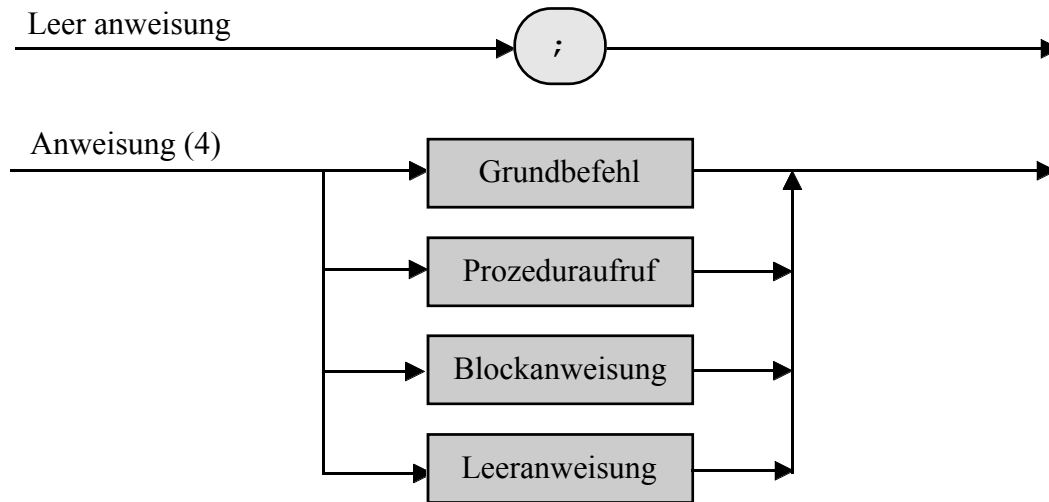


Abbildung 9.6: Syntaxdiagramm: Leeranweisung

9.4.2 Semantik

Leeranweisungen haben keinerlei Auswirkungen auf den Programmablauf.

9.4.3 Beispiele

Im folgenden Beispielpogramm sind wahllos ein paar Leeranweisungen eingestreut.

```

void main()
{
  lege_drei_koerner_ab();
  {
    ; vor(); vor();
  };
  ;; lege_drei_koerner_ab();
}
  
```

```

}

void lege_drei_koerner_ab()
{
    gib();;; gib();; gib();
}

```

Beachten Sie, daß Leeranweisungen (spezielle) Anweisungen sind und daher überall dort (aber auch nur dort!) eingesetzt werden dürfen, wo Anweisungen auftreten dürfen. Insbesondere ist es syntaktisch nicht erlaubt, ein Semikolon hinter einen Prozedurrumpf zu plazieren, wie im folgenden Beispiel:

```

void main()
{
    vor(); rechts_um(); vor(); links_um();
}; // <- syntaktischer Fehler!

void rechts_um()
{
    links_um(); links_um(); links_um();
}

```

9.5 Bedingte Anweisung

Wo und wie lassen sich Testbefehle bzw. boolesche Ausdrücke im Hamstermodell nun benutzen? Wir hatten in Abschnitt 9.1.4 gesehen, daß sich mit Hilfe der Testbefehle gefährliche Situationen vorherbestimmen und umgehen lassen. Beispielsweise soll der Befehl `vor()`; nur dann ausgeführt werden, wenn der Testbefehl `vorn_frei()` den Wert `true` liefert. Zur syntaktischen Formulierung dieser *nur wenn ..., dann ...* Beziehung existiert in der Hamstersprache die sogenannte *bedingte Anweisung*. Wie der Name schon aussagt, soll in einer bedingten Anweisung eine bestimmte Anweisung nur unter einer bestimmten Bedingung ausgeführt werden.

9.5.1 Syntax

Die bedingte Anweisung, die auch *if-Anweisung* genannt wird, ist eine zusammengesetzte Anweisung, deren genaue Syntax in Abbildung 9.7 definiert wird. Bei der Anweisung handelt es sich im allgemeinen um eine Blockanweisung. Abbildung 9.7 erweitert auch das Syntaxdiagramm „Anweisung“ aus Abbildung 9.6.

9.5.2 Semantik

Es wird zunächst wie in Abschnitt 9.2 beschrieben der boolesche Ausdruck in den runden Klammern ausgewertet. Falls dieser Ausdruck den Wert `true` liefert, d.h. die Bedingung erfüllt ist, wird die Anweisung ausgeführt. Liefert der boolesche Ausdruck den Wert `false`, dann wird die Anweisung nicht ausgeführt.

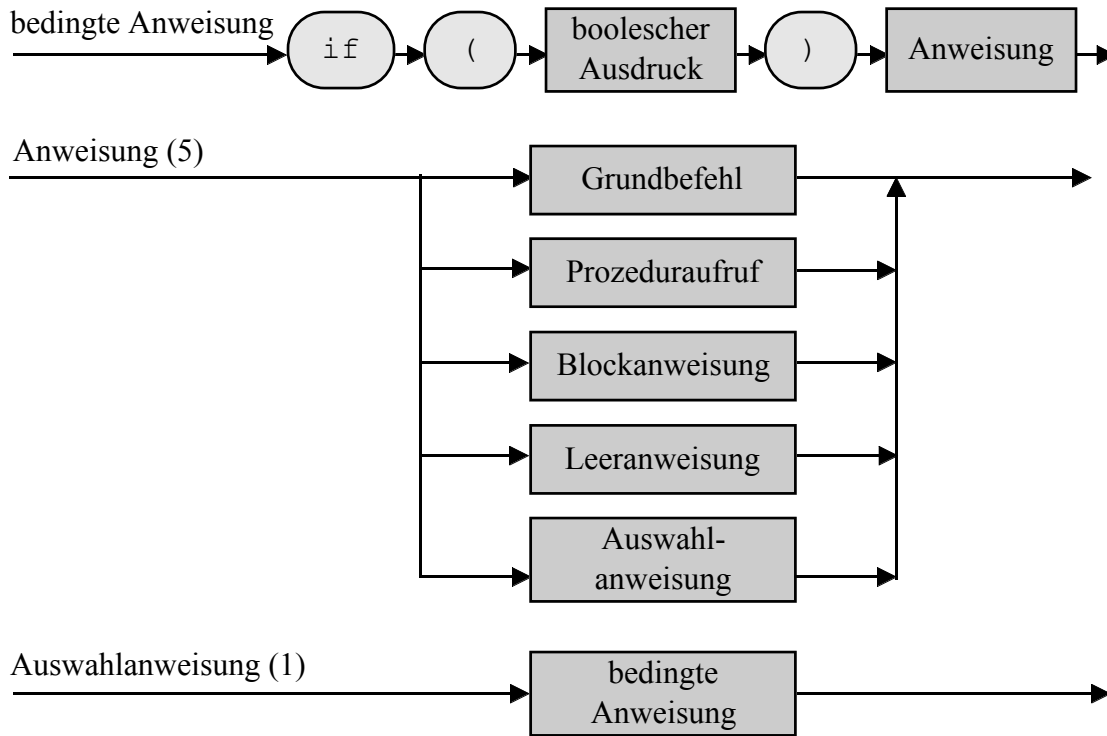


Abbildung 9.7: Syntaxdiagramm: Bedingte Anweisung

9.5.3 Beispiele

Folgende Anweisungen sind syntaktisch korrekte bedingte Anweisungen:

1. `if (vorn_frei()) vor();`
2. `if (korn_da() && vorn_frei()) { nimm(); vor(); }`
3. `if (korn_da()) if (vorn_frei()) { nimm(); vor(); }`

Bezüglich der Hamsterlandschaft in Abbildung 9.8 (links) wird der Hamster, wenn die bedingte Anweisung (1) ausgeführt wird, den Befehl `vor();` ausführen, da der Testbefehl `vorn_frei()` den Wert `true` liefert. Anders ist dies bezüglich der Situation in Abbildung 9.8 (rechts). Hier wird der Hamster bei Ausführung von Anweisung (1) nichts tun, da der Testbefehl den Wert `false` liefert.

Liegt auf der Kachel, auf der sich der Hamster in Abbildung 9.8 (links) befindet, ein Korn, dann wird der Hamster bei Aufruf der bedingten Anweisung (2), da der boolesche Ausdruck den Wert `true` liefert, die Blockanweisung ausführen, d.h. er wird ein Korn fressen und anschließend eine Kachel nach vorne hüpfen.

Die bedingte Anweisung (3) ist zur bedingten Anweisung (2) semantisch äquivalent, d.h. die beiden Anweisungen bewirken dasselbe. Anweisung (3) nutzt aus, daß die bedingte Anweisung selbst wieder eine Anweisung ist, also auch selbst wieder als Anweisung in einer bedingten Anweisung auftreten darf.

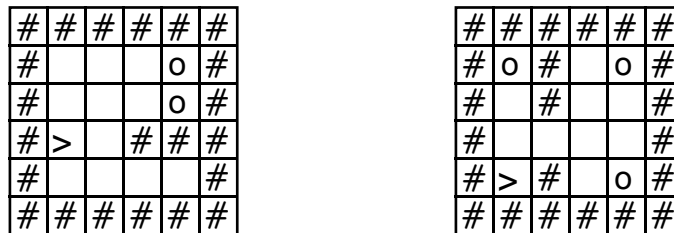


Abbildung 9.8: Bedingte Anweisungen

9.5.4 Anmerkungen

Auch wenn es sich bei der (inneren) Anweisung einer bedingten Anweisung um keine Anweisungssequenz handelt, sollten Sie sich angewöhnen, die Blockanweisung zu nutzen, um Fehlern vorzubeugen. Schauen Sie sich das folgende Beispiel an:

```
if (korn_da())
    nimm();
    vor();
```

Bei dieser Anweisung handelt es sich um eine Anweisungssequenz, die aus einer bedingten Anweisung `if (korn_da()) nimm();` und der Anweisung `vor();` besteht, d.h. nur die Anweisung `nimm();` gehört zur bedingten Anweisung, der Befehl `vor();` wird immer ausgeführt. Um solchen unter Umständen fehlerhaften Situationen vorzubeugen, die häufig dann entstehen, wenn nachträglich Anweisungen ins Programm eingefügt werden, empfiehlt sich der Einsatz der Blockanweisung.

```
if (korn_da())
{
    nimm();
    vor();
}
```

9.5.5 „Sichere“ Grundbefehle

Mit Hilfe der bedingten Anweisung lassen sich die Grundbefehle nun „sicher“ machen. Dazu müssen Sie nur die folgenden drei Prozeduren definieren und anstelle der entsprechenden Grundbefehle im Programm verwenden:

```
void sicheres_vor()
{
    if (vorn_frei())
    {
        vor();
    }
}
```

```
void sicheres_nimm()
{
    if (korn_da())
    {
        nimm();
    }
}

void sicheres_gib()
{
    if (!maul_leer())
    {
        gib();
    }
}
```

9.6 Alternativanweisung

Die bedingte Anweisung ermöglicht die optionale Ausführung einer Anweisung unter einer bestimmten Bedingung. In diesem Abschnitt wird die bedingte Anweisung durch die Alternativanweisung – auch Fallunterscheidung oder Verzweigung genannt – erweitert. Bei der Alternativanweisung können Sie nicht nur angeben, daß eine bestimmte Anweisung nur bedingt ausgeführt werden soll, sondern Sie können auch eine alternative Anweisung ausführen lassen, wenn die Bedingung nicht erfüllt ist.

9.6.1 Syntax

Die Alternativanweisung ist eine bedingte Anweisung mit einem angehängten sogenannten *else-Teil*. Dieser besteht aus dem Schlüsselwort `else` und einer Anweisung – im allgemeinen eine Blockanweisung. Die Anweisung einer Alternativanweisung, die ausgeführt wird, wenn der boolesche Ausdruck den Wert `true` liefert, wird im folgenden auch *true-Anweisung* und die Anweisung des *else*-Teils dementsprechend *false-Anweisung* genannt. Die genaue Syntax der Alternativanweisung können Sie Abbildung 9.9 entnehmen. Die Alternativanweisung ist wie die bedingte Anweisung eine Auswahlanweisung. Deshalb wird in Abbildung 9.9 das Syntaxdiagramm „Auswahlanweisung“ aus Abbildung 9.7 erweitert.

9.6.2 Semantik

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 ausgeführt; liefert der boolesche Ausdruck den Wert `false`, dann wird die *false*-Anweisung – nicht aber die *true*-Anweisung – ausgeführt.

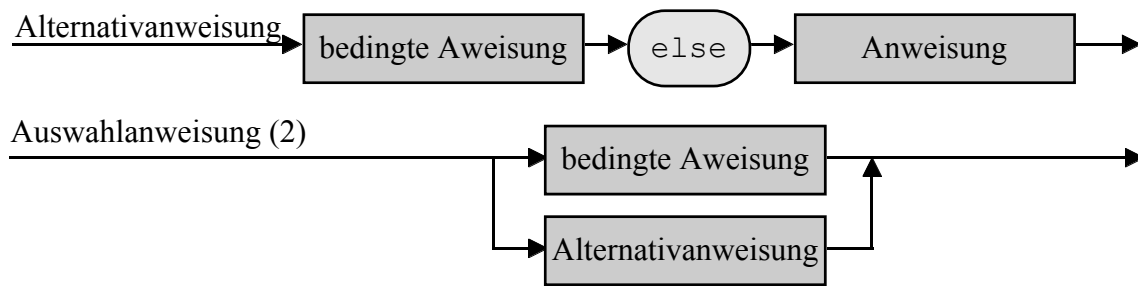


Abbildung 9.9: Syntaxdiagramm: Alternativanweisung

9.6.3 Beispiele

Die folgenden Beispiele sollen die Alternativanweisung verdeutlichen.

```
// Beispiel 1
if (vorn_frei())
{
    vor();
}
else
{
    links_um();
}
```

Steht der Hamster bei der Ausführung von Beispiel 1 nicht vor einer Mauer, d.h. der Testbefehl `vorn_frei()` liefert den Wert `true`, dann wird die `true`-Anweisung ausgeführt: der Hamster hüpft eine Kachel nach vorne. Steht er jedoch vor einer Mauer, dann wird die `false`-Anweisung ausgeführt: der Hamster dreht sich um 90 Grad nach links.

```
// Beispiel 2a
if (maul_leer())
;
else
    gib();
links_um();

// Beispiel 2b
if (!maul_leer())
{
    gib();
}
links_um();
```

In Beispiel 2a wird die `true`-Anweisung durch eine Leeranweisung gebildet, d.h. hat der Hamster kein Korn im Maul, so passiert nichts. Der `else`-Teil enthält den `gib()`-Befehl, d.h. hat der

Hamster ein Korn im Maul, so legt er eines ab. Der `links_um()`-Befehl gehört nicht mehr zur Alternativanweisung und wird in beiden Fällen ausgeführt. Beispiel 2a ist zwar syntaktisch korrekt, statt der Verwendung der Leeranweisung sollte jedoch sinnvollerweise die Bedingung negiert werden, wie in Beispiel 2b. Beispiel 2b ist semantisch äquivalent zu Beispiel 2a.

```
// Beispiel 3
if (vorn_freie())
    vor();
else if (korn_da())
    nimm();
else if (!maul_leer())
    gib();
else
    links_um();
```

Beispiel 3 zeigt, daß die false-Anweisung selbst wieder eine zusammengesetzte Anweisung, insbesondere also auch wieder eine Alternativanweisung sein darf. Auswahlanweisungen lassen sich dadurch schachteln. Wenn der Testbefehl `vorn_freie()` den Wert `true` liefert, dann wird die true-Anweisung, also der Befehl `vor()`; ausgeführt, und das Beispiel ist beendet. Wenn die Kachel vor dem Hamster allerdings durch eine Mauer blockiert ist, d.h. die Auswertung der ersten Bedingung den Wert `false` ergibt, dann wird die false-Anweisung ausgeführt. Hierbei handelt es sich wieder um eine Alternativauswahl. Liefert der Testbefehl `korn_da()` den Wert `true`, dann wird der Befehl `nimm()`; als true-Anweisung ausgeführt, und das Beispiel ist beendet. Liefert jedoch auch diese zweite Bedingung den Wert `false`, dann wird der zweite else-Teil „aktiv“. Wiederum handelt es sich bei der false-Anweisung um eine Alternativanweisung. Liefert der boolesche Ausdruck `!maul_leer` den Wert `true`, dann legt der Hamster ein Korn ab, ansonsten dreht er sich um 90 Grad nach links. Insgesamt läßt sich feststellen, daß der Hamster unabhängig von der Situation, in der er sich gerade befindet, beim Aufruf von Beispiel 3 immer genau einen Grundbefehl ausführt.

```
// Beispiel 4a
if (vorn_freie())
if (korn_da())
    nimm();
else
    vor();

// Beispiel 4b
if (vorn_freie())
{
    if (korn_da())
        nimm();
}
else
    vor();

// Beispiel 4c
```

```

if (vorn_frei())
{
  if (korn_da())
  {
    nimm();
  }
  else
  {
    vor();
  }
}

```

Beispiel 4a ist auf den ersten Blick mehrdeutig: Es ist nicht klar, ob der else-Teil zum ersten if oder zum zweiten if gehört. Dieses ist jedoch in der Hamstersprache eindeutig definiert: In geschachtelten Auswahlanweisungen gehört ein else-Teil immer zum innersten if.

In Beispiel 4a bildet der else-Teil also den else-Teil der `korn_da()`- und nicht der `vorn_frei()`-Bedingung. Die Anweisung in Beispiel 4a ist eine bedingte Anweisung, deren true-Anweisung durch eine Alternativanweisung gebildet wird. Ist die Kachel vor dem Hamster blockiert, d.h. der boolesche Ausdruck `vorn_frei()` liefert den Wert `false`, dann ist das Beispiel beendet. Wird dagegen der Wert `true` ermittelt, dann wird die (innere) Alternativanweisung ausgeführt.

Soll der else-Teil nicht zum inneren sondern zu einem äußeren if gehören, so müssen Klammern verwendet werden wie in Beispiel 4b. Die Anweisung in Beispiel 4b ist eine Alternativanweisung, deren true-Anweisung aus einer Blockanweisung besteht, die eine bedingte Anweisung enthält.

Um Mißverständnissen bzw. Fehlern vorzubeugen, sollten Sie auch bei den Alternativanweisungen die true- und false-Anweisung jeweils durch eine Blockanweisung „kapseln“. Nutzen Sie also anstelle der verkürzten Schreibweise in Beispiel 4a möglichst die etwas längere dafür aber „sicherere“ Schreibweise von Beispiel 4c. Achten Sie auch auf eine übersichtliche Einrückung der entsprechenden Teile.

9.6.4 Anmerkungen

Schauen Sie sich die folgenden beiden Anweisungen an. `bA` sei dabei Platzhalter für einen beliebigen booleschen Ausdruck, und `a1` und `a2` seien Platzhalter für beliebige Anweisungen:

```

// Anweisung 1
if (bA)
  a1;
else
  a2;

// Anweisung 2
if (bA)
  a1;
if (!bA)
  a2;

```

Sie mögen vielleicht denken, daß die beiden Anweisungen semantisch äquivalent sind. In der Tat sind sie dies auch mit unseren bisherigen Kenntnissen. Wir werden jedoch in Kapitel 11.5 sehen, daß dies nicht immer der Fall sein muß. Grund hierfür ist der, daß in Anweisung 1 der boolesche Ausdruck einmal und in Anweisung 2 der boolesche Ausdruck zweimal ausgewertet wird.

9.7 Beispielprogramme

In diesem Abschnitt werden einige Beispiele für Hamster-Aufgaben gegeben und jeweils eine oder mehrere Musterlösungen vorgestellt. Schauen Sie sich die Beispiele genau an und versuchen Sie, die Lösungen nachzuvollziehen.

9.7.1 Beispielprogramm 1

Aufgabe:

Gegeben sei das Hamster-Territorium in Abbildung 9.10. Der Hamster soll genau zwei Körner einsammeln.

#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
#																			#
#				v		o		#				#	#		#	#	#		#
#	#	#	#		#			#			o	#	o	o	o	o	o		#
#				o		o						#	#	#	#		#		#
#			#	#	#		#	#	#	#	#	#							#
#	o	o	#	o							#								#
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 9.10: Hamsterlandschaft zu Beispielprogramm 1

Lösung 1:

Bei dieser Lösung überprüft der Hamster nicht, ob er bereits in der Ausgangsstellung auf einem Kornfeld steht.

```
void main()
{
    // suche und nehme erstes Korn
    vor(); vor(); nimm();
    // suche und nehme zweites Korn
    links_um();
    vor(); vor(); nimm();
}
```

Lösung 2:

Bei dieser Lösung überprüft der Hamster, ob er bereits in der Ausgangsstellung auf einem Kornfeld steht.

```

void main()
{
  if (korn_da()) // der Hamster steht bereits auf einem Kornfeld
  {
    // nehme erstes Korn
    nimm();
  }
  else
  {
    // suche und nehme erstes Korn
    vor(); vor(); nimm();
    links_um();
  }
  // suche und nehme zweites Korn
  vor(); vor(); nimm();
}

```

9.7.2 Beispielprogramm 2

Aufgabe:

Gegeben sei das Hamster-Territorium in Abbildung 9.11. Der Hamster ist durch das viele Herumrennen so verwirrt, daß er nicht mehr weiß, wieviele Körner er im Maul hat. Falls möglich soll er in jeder Ecke des Territoriums ein Korn ablegen.

#	#	#	#	#	#	#
#						#
#						#
#						#
#						#
#	>					#
#	#	#	#	#	#	#

Abbildung 9.11: Hamsterlandschaft zu Beispielprogramm 2

Lösung 1:

Der Hamster besucht jede der vier Ecken im Territorium und legt, falls er noch ein Korn im Maul hat, jeweils eines ab.

```

void main()
{
  if (!maul_leer()) gib();
  laufe_bis_in_die_naechste_ecke()
  if (!maul_leer()) gib();
  laufe_bis_in_die_naechste_ecke()
  if (!maul_leer()) gib();
}

```

```

    laufe_bis_in_die_naechste_ecke()
    if (!maul_leer()) gib();
}

void laufe_bis_in_die_naechste_ecke()
{
    vor(); vor(); vor(); vor();
}

```

Lösung 2:

Bei dieser Lösung ist der Hamster schlauer: Wenn er gar keine Körner mehr im Maul hat, braucht er ja auch gar nicht mehr weiterzulaufen. Er überprüft also nach jedem Ablegen eines Kornes, ob es sich noch lohnt weiterzulaufen.

```

void main()
{
    if (!maul_leer())
    {
        gib();
        if (!maul_leer())
        {
            laufe_bis_in_die_naechste_ecke()
            gib();
            if (!maul_leer())
            {
                laufe_bis_in_die_naechste_ecke()
                gib();
                if (!maul_leer())
                {
                    laufe_bis_in_die_naechste_ecke()
                    gib();
                }
            }
        }
    }
}

void laufe_bis_in_die_naechste_ecke()
{
    vor(); vor(); vor(); vor();
}

```

9.7.3 Beispielprogramm 3**Aufgabe:**

Gegeben sei das Hamster-Territorium in Abbildung 9.12. Auf jedem Feld liegen ein oder zwei Körner. Der Hamster soll dafür sorgen, daß auf jedem Feld genau ein Korn liegt.

#	#	#	#	#	#	#
#	o	o	o	o	o	#
#	o	o	o	o	o	#
#	o	o	o	o	o	#
#	o	o	o	o	o	#
#	>	o	o	o	o	#
#	#	#	#	#	#	#

Abbildung 9.12: Hamsterlandschaft zu Beispielprogramm 3

Lösung:

```

void main()
{
    ueberpruefe_eine_reihe();
    links_um(); vor(); links_um();
    ueberpruefe_eine_reihe();
    rechts_um(); vor(); rechts_um();
    ueberpruefe_eine_reihe();
    links_um(); vor(); links_um();
    ueberpruefe_eine_reihe();
    rechts_um(); vor(); rechts_um();
    ueberpruefe_eine_reihe();
}

void ueberpruefe_eine_reihe()
{
    evtl_fressen(); vor();
    evtl_fressen(); vor();
    evtl_fressen(); vor();
    evtl_fressen(); vor();
    evtl_fressen();
}

void evtl_fressen()
{
    // erstmal ein Korn fressen
    nimm();
    // falls es das einzige Korn war, muss es wieder abgelegt werden
    if (!korn_da())
    {
        gib();
    }
}

void rechts_um()

```

```

{
  links_um(); links_um(); links_um();
}

```

9.8 Übungsaufgaben

Nun sind Sie wieder gefordert; denn in diesem Abschnitt werden Ihnen einige Hamster-Aufgaben gestellt, die sie selbständig zu lösen haben.

9.8.1 Aufgabe 1

Gegeben sei das Hamster-Territorium in Abbildung 9.13. Auf allen Feldern, auf denen Körner eingezeichnet sind, liegen entweder ein oder zwei Körner. Der Hamster soll drei Körner einsammeln und dabei einen möglichst kurzen Weg zurücklegen.

#	#	#	#	#	#	#
#	o		o		o	#
#		#	o	#	o	#
#	o	o	#	o		#
#		#	o	#	o	#
#	>	o		o		#
#	#	#	#	#	#	#

Abbildung 9.13: Hamsterlandschaft zu Aufgabe 1

9.8.2 Aufgabe 2

Gegeben sei das Hamster-Territorium in Abbildung 9.14. Der Hamster weiß nicht, wieviele Körner er im Maul hat. Solange er noch Körner im Maul hat (!), soll er folgendes tun: Er soll in der aktuellen Ecke ein Korn ablegen und dann in die zweite Ecke laufen. Dort soll er zwei Körner ablegen und in die dritte Ecke laufen. Dort soll er drei Körner ablegen und in die vierte Ecke laufen. Dort soll er vier Körner ablegen.

#	#	#	#	#	#	#
#						#
#		#	#	#		#
#		#	#	#		#
#		#	#	#		#
#	>					#
#	#	#	#	#	#	#

Abbildung 9.14: Hamsterlandschaft zu Aufgabe 2

9.8.3 Aufgabe 3

Der Hamster erhält dieselbe Aufgabe wie in Aufgabe 2, d.h. nach Ausführung des Programms sollen in der unteren linken Ecke ein Korn, in der unteren rechten Ecke zwei Körner, in der oberen rechten Ecke drei Körner und in der oberen linken Ecke vier Körner liegen. Nur diesmal sieht die Hamsterlandschaft anfangs ein wenig anders aus, denn in den Eckfeldern liegen bereits jeweils ein, zwei oder drei Körner (siehe Abbildung 9.15). Außerdem soll der Hamster, sobald er feststellt, daß er kein Korn mehr im Maul hat oder nachdem er in der vierten Ecke das vierte Korn abgelegt hat, zurück in seine Ausgangsposition laufen.

#	#	#	#	#	#	#
#	o				o	#
#		#	#	#		#
#		#	#	#		#
#		#	#	#		#
#	>				o	#
#	#	#	#	#	#	#

Abbildung 9.15: Hamsterlandschaft zu Aufgabe 3

9.8.4 Aufgabe 4

Denken Sie sich selbst weitere Hamster-Aufgaben aus, und versuchen Sie, diese zu lösen. Viel Spaß!

Kapitel 10

Wiederholungsanweisungen

In den bisherigen Kapiteln hatten Hamsteraufgaben immer folgende Form: Gegeben eine bestimmte Landschaft und gegeben ein bestimmtes Problem; entwickeln Sie ein Hamsterprogramm, das das Problem bzgl. der Landschaft löst. Bisherige Hamsterprogramme waren also sehr unflexibel, sie lieferten nur Lösungen für eine bestimmte fest vorgegebene Landschaft. Geringfügige Änderungen an der Landschaft konnten zu einem inkorrekten Verhalten des Programms führen. In diesem Kapitel werden wir Wiederholungsanweisungen kennenlernen, die es ermöglichen, Hamsterprogramme zu entwickeln, die gegebene Probleme für alle Landschaften eines bestimmten Landschaftstyps lösen.

10.1 Motivation

Schauen Sie sich die Hamsterlandschaft in Abbildung 10.1 (links) an.

#	#	#	#	#	#	#
#						#
#				#	o	#
#	>			#	o	#
#				#	o	#
#				#	o	#
#	#	#	#	#	#	#

#	#	#	#	#	#	#
#						#
#	o		#	#		#
#	#		#	o		#
#	o		#	#		#
#	#	^		#	o	#
#	#	#	#	#	#	#

Abbildung 10.1: Wiederholungsanweisung (Motivation)

Der Hamster soll in Blickrichtung bis zur nächsten Wand laufen. Eine korrekte Lösung dieses Problems ist das folgende Hamsterprogramm:

```
void main()
{
    vor();
    vor();
}
```

Schauen Sie sich nun die Hamsterlandschaft in Abbildung 10.1 (rechts) an. Auch für diese Situation soll ein Hamsterprogramm geschrieben werden, das den Hamster in Blickrichtung bis zur nächsten Wand laufen läßt:

```
void main()
{
    vor();
    vor();
    vor();
    vor();
}
```

Beide Programme lösen das Problem, daß der Hamster bis zur nächsten Wand laufen soll, jeweils für eine fest vorgegebene Landschaft. Viel flexibler wäre es, wenn wir ein Programm entwickeln könnten, daß beide obigen Probleme bzw. allgemein Probleme folgender Art löst: Der Hamster stehe auf einer Kachel der Hamsterlandschaft. Irgendwo in Blickrichtung vor ihm befindet sich eine Mauer. Der Hamster soll bis zu dieser Mauer laufen und dann anhalten. Zur Lösung dieses Problems benötigen wir eine Anweisung der Art: *solange vorne frei ist, hüpfе eine Kachel nach vorne*, oder allgemeiner: *solange eine bestimmte Bedingung erfüllt ist, führe eine bestimmte Aktion aus*, bzw. präziser mit den Hilfsmitteln der Hamstersprache formuliert: *solange ein boolescher Ausdruck den Wert `true` liefert, führe eine gegebene Anweisung (wiederholt) aus*. In der Hamstersprache existieren zwei Alternativen zur Formulierung derartiger Wiederholungsanweisungen, die *while-Anweisung* und die *do-Anweisung*.

10.2 while-Anweisung

Mit Hilfe der *while-Anweisung* – auch *while-Schleife* genannt – läßt sich in der Hamstersprache eine Anweisung in Abhängigkeit eines booleschen Ausdrucks beliebig oft wiederholt ausführen.

10.2.1 Syntax

Die genaue Syntax der *while-Anweisung* kann Abbildung 10.2 entnommen werden.

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

In Abbildung 10.2 wird das Syntaxdiagramm „Anweisung“ aus Abbildung 9.7 erweitert.

10.2.2 Semantik

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,

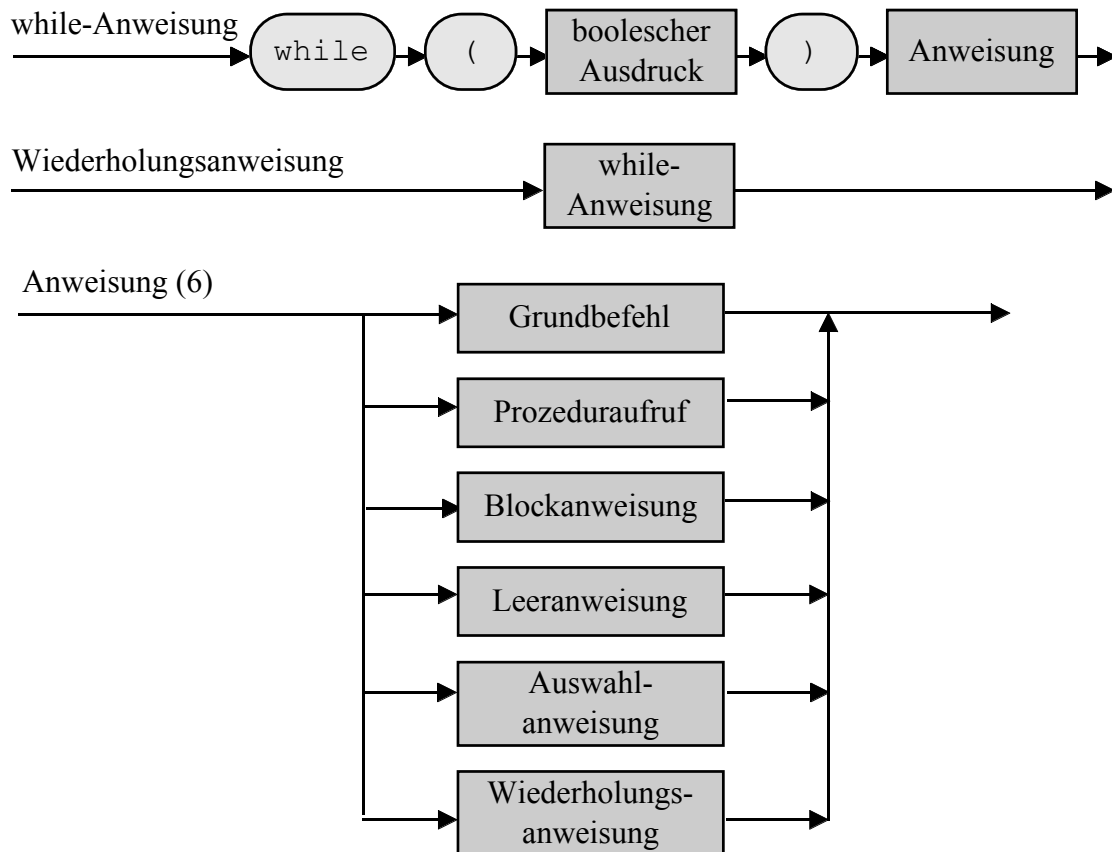


Abbildung 10.2: Syntaxdiagramm: while-Anweisung

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 solange, bis (hoffentlich) irgendwann einmal die Bedingung nicht mehr erfüllt ist.

10.2.3 Korrekte Hamsterprogramme

Im folgenden werden zu Hamsteraufgaben die Hamsterlandschaften nicht mehr explizit angegeben, sondern nur noch durch ihre charakteristischen Merkmale beschrieben. Hamsterprogramme müssen für alle (!) Landschaften korrekt arbeiten, die die Merkmale bei Start des Programms erfüllen. Arbeitet ein Hamsterprogramm auch nur für eine gegebene Landschaft, die die Merkmale erfüllt, nicht korrekt, gilt das Hamsterprogramm insgesamt als fehlerhaft.

Gegeben eine Hamsteraufgabe und eine Charakterisierung einer Hamsterlandschaft. Ein Hamsterprogramm ist korrekt (bzgl. der Hamsteraufgabe und der Landschaftscharakterisierung), wenn es alle folgenden Bedingungen erfüllt:

- Es muß syntaktisch korrekt sein.

- Es muß die Aufgabenstellung für alle sich aus der Landschaftcharakterisierung ergebenden möglichen Ausgangssituationen korrekt und vollständig lösen.
- Es darf für keine sich aus der Landschaftcharakterisierung ergebenden möglichen Ausgangssituation zu einem Laufzeitfehler führen.
- Es muß nach endlicher Zeit für alle sich aus der Landschaftcharakterisierung ergebenden möglichen Ausgangssituationen enden, es sei denn, eine Nicht-Terminierung des Programms wird in der Aufgabenstellung explizit erlaubt.

10.2.4 Beispiele

10.2.4.1 Beispiel 1

Das Hamsterprogramm für das oben skizzierte Problem, daß der Hamster irgendwo in einer Landschaft steht und bis zur nächsten Mauer laufen soll, sieht folgendermaßen aus:

```
void main()
{
  while (vorn_frei())
  {
    vor();
  }
}
```

Überprüfen wir einmal, ob das Programm für die Landschaft in Abbildung 10.1 (links) korrekt arbeitet (siehe auch Abbildung 10.3 (links)).

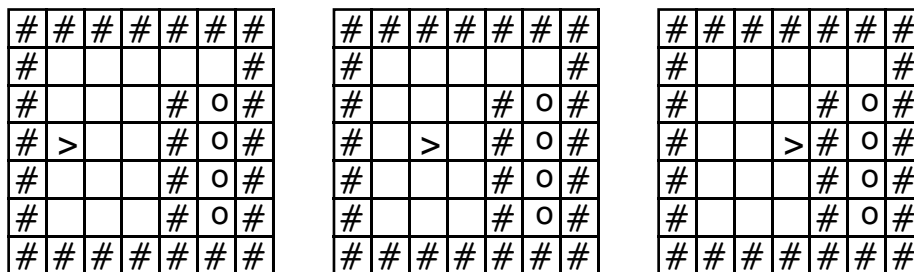


Abbildung 10.3: Typische Hamsterlandschaften zu Beispiel 1

Zunächst wird der Testbefehl `vorn_frei()` ausgewertet. Er liefert den Wert `true`. Also ist die Schleifenbedingung erfüllt. D.h. als nächstes wird die Iterationsanweisung ausgeführt. Dieses ist eine Blockanweisung, die als einzige Anweisung den Befehl `vor();` enthält. Der Hamster hüpfte eine Kachel nach vorne (siehe Abbildung 10.3 (Mitte)). Nach der Abarbeitung der Iterationsanweisung wird erneut die Schleifenbedingung überprüft. Der Testbefehl `vorn_frei()` liefert auch dieses Mal den Wert `true`, so daß der `vor();`-Befehl ein zweites Mal ausgeführt wird und sich die Situation gemäß Abbildung 10.3 (rechts) ergibt. Wiederum wird nun die Schleifenbedingung überprüft. Inzwischen steht der Hamster jedoch vor einer Mauer, so daß der Testbefehl

`vorn_frei()` diesmal den Wert `false` liefert. Damit ist die `while`-Anweisung und – weil dies die einzige Anweisung des Hamsterprogrammes war – auch das gesamte Hamsterprogramm beendet.

Auf dieselbe Art und Weise läßt sich verifizieren, daß das Programm auch für die Landschaft in Abbildung 10.1 (rechts) korrekt arbeitet. Kontrollieren Sie dies bitte selbst.

10.2.4.2 Beispiel 2

Hamsteraufgabe Der Hamster befindet sich irgendwo in einem rechteckigen durch Mauern abgeschlossenen sonst aber mauerlosen Raum unbekannter (aber endlicher) Größe. Er soll in irgendeine Ecke laufen und dann anhalten. Abbildung 10.4 enthält einige Beispiele für den skizzierten Landschaftstyp.

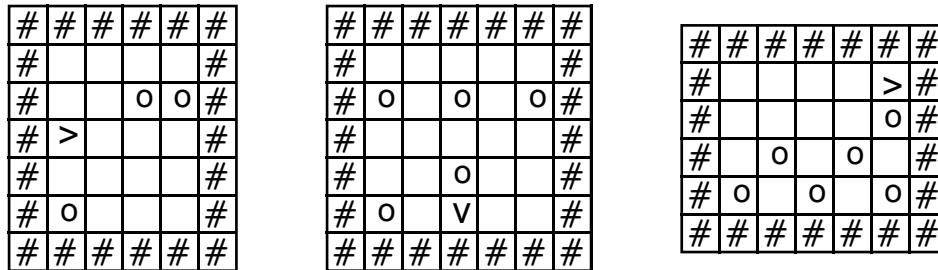


Abbildung 10.4: Typische Hamsterlandschaften zu Beispiel 2

Lösungsidee Der Hamster läuft bis zur nächsten Wand in Blickrichtung vor ihm, dreht sich um 90 Grad nach links und läuft erneut bis zur nächsten Wand in Blickrichtung vor ihm.¹

Programm

```
void main()
{
    // laufe bis zur naechsten Wand
    while (vorn_frei())
    {
        vor();
    }

    // drehe dich um 90 Grad nach links
    links_um();

    // laufe erneut bis zur naechsten Wand
    while (vorn_frei())
    {
        vor();
    }
}
```

¹Dies ist zwar nicht der schnellste Weg, aber das ist ja auch in der Aufgabe nicht verlangt!

Test Testen wir das Programm einmal für die Landschaft in Abbildung 10.4 (links) (siehe auch Abbildung 10.5 (a)). Die Schleifenbedingung der ersten `while`-Anweisung wird insgesamt viermal überprüft. Die ersten drei Male liefert sie den Wert `true`, d.h. der `vor()`-Befehl innerhalb der Iterationsanweisung wird dreimal ausgeführt. Bei der vierten Überprüfung steht der Hamster inzwischen vor der Mauer (siehe Abbildung 10.5 (b)), so daß der Testbefehl `vorn_frei()` nun den Wert `false` liefert. Damit ist die erste `while`-Anweisung des Hamsterprogramms beendet. Als nächste Anweisung folgt nun der `links_um()`-Befehl. Nach seiner Ausführung ergibt sich die Situation in Abbildung 10.5 (c). Nun folgt eine zweite `while`-Anweisung. Die Schleifenbedingung dieser `while`-Anweisung wird nun wiederholt ausgewertet, zweimal liefert sie den Wert `true`, so daß der Hamster zweimal den Befehl `vor()` ausführt. Dann steht er vor einer Mauer, genauer gesagt in einer Ecke (siehe Abbildung 10.5 (d)), die Schleifenbedingung ist nicht mehr erfüllt. Damit ist die `while`-Anweisung und – weil dies die letzte Anweisung des Hauptprogrammes war – auch das gesamte Programm beendet, und zwar korrekt: Der Hamster steht in einer Ecke.

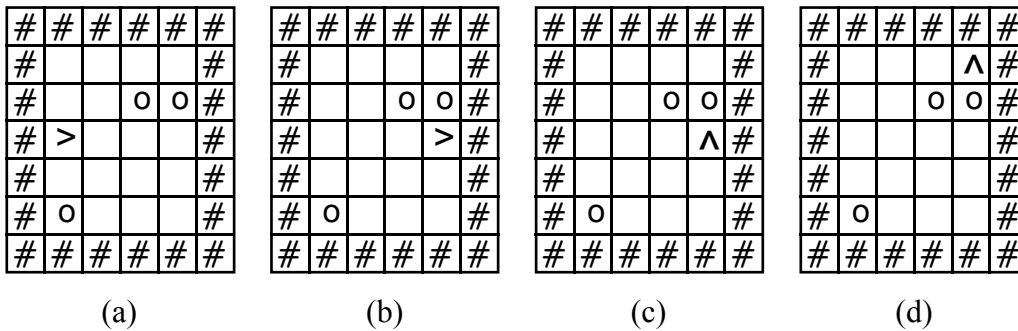


Abbildung 10.5: Test von Beispiel 2

Überprüfen Sie bitte selbständig, daß das Programm auch für die anderen vorgegebenen Landschaften aus Abbildung 10.4 korrekt arbeitet. Insbesondere ist die Abbildung rechts interessant. Hier befindet sich der Hamster bereits zu Anfang in einer Ecke.

Verbesserung Sie haben sicher schon festgestellt, daß die beiden `while`-Anweisungen des Programmes identisch sind. Aus Gründen der Übersichtlichkeit bietet es sich also an, eine Prozedur `laufe_bis_zur_naechsten_wand` zu definieren und diese zweimal aufzurufen. Damit sieht das Programm dann folgendermaßen aus:

```
void main()
{
    laufe_bis_zur_naechsten_wand();
    links_um();
    laufe_bis_zur_naechsten_wand();
}

void laufe_bis_zur_naechsten_wand()
{
    while (vorn_frei())
    {
        vor();
    }
}
```

```

    }
}

```

10.2.4.3 Beispiel 3

Hamsteraufgabe Der Hamster befindet sich wie in Beispiel 2 irgendwo in einem rechteckigen durch Mauern abgeschlossenen sonst aber mauerlosen Raum unbekannter (aber endlicher) Größe. Auf den einzelnen Kacheln kann jeweils eine beliebig große aber endliche Anzahl an Körner liegen. Der Hamster soll in eine Ecke laufen und dann anhalten. Dabei soll er alle Körner einsammeln, die er auf seinem Weg findet. Abbildung 10.4 enthält einige Beispiele für den skizzierten Landschaftstyp.

Programm

```

void main()
{
    sammle();
    laufe_bis_zur_naechsten_wand_und_sammle();
    links_um();
    laufe_bis_zur_naechsten_wand_und_sammle();
}

void sammle()
{
    while (korn_da())
    {
        nimm();
    }
}

void laufe_bis_zur_naechsten_wand_und_sammle()
{
    while (vorn_frei())
    {
        vor();
        sammle();
    }
}

```

Erläuterung Das Programm entspricht zum großen Teil dem Lösungsprogramm aus Beispiel 2. Es wird jedoch durch die Prozedur `sammle` erweitert. Wird diese ausgeführt, dann sammelt der Hamster alle Körner ein, die auf der Kachel liegen, auf der er sich gerade befindet. Die Prozedur arbeitet dabei unabhängig von der konkreten Anzahl an Körnern auf einer Kachel korrekt. Nehmen wir an, auf einer Kachel liegen sieben Körner. Dann ist die Schleifenbedingung `korn_da()` siebenmal erfüllt, d.h. es wird insgesamt siebenmal der Befehl `nimm()`; ausgeführt.

Nach dem siebten Schleifendurchlauf hat der Hamster jedoch alle Körner der Kachel aufgenommen, so daß die Schleifenbedingung bei ihrer achten Überprüfung nicht mehr erfüllt und damit die while-Anweisung beendet ist. Bei einer Anzahl von 20 Körnern auf einer Kachel wird die Iterationsanweisung genau zwanzig Mal ausgeführt; allgemein bei n Körnern genau n -mal. Beachten Sie, daß die Prozedur `sammle` auch dann korrekt arbeitet, wenn auf einer Kachel kein Korn liegt. Dann ist die Schleifenbedingung bereits bei der ersten Überprüfung nicht erfüllt und der `nimm()`;-Befehl wird kein einziges Mal ausgeführt.

Die Prozedur `sammle` wird in der Iterationsanweisung der prozedur `laufe_bis_zur_naechsten_wand_und_sammle` aufgerufen. Diese Prozedur bewirkt, daß jedes Mal, wenn der Hamster auf eine neue Kachel springen kann, er dies auch tut und diese anschließend „abgrast“. Die Iterationsanweisung der diesbezüglichen while-Anweisung innerhalb der Prozedur besteht daher aus einer Blockanweisung mit der Anweisungssequenz `vor()`; und `sammle()`;

Achtung Achten Sie darauf, daß die Prozedur `sammle` ganz zu Anfang des Programms einmal aufgerufen wird. Ohne diese Anweisung wäre das Programm nicht korrekt! Es würde nur bei solchen Landschaften korrekt arbeiten, bei denen sich auf der Ausgangskachel des Hamsters keine Körner befinden. Bei den anderen Landschaften würde der Hamster die Körner auf seiner Ausgangskachel liegenlassen und damit nicht die komplette Aufgabenstellung erfüllen.

10.2.5 Anmerkungen

10.2.5.1 Übersichtlichkeit

Um eine bessere Übersichtlichkeit ihres Programmes zu erzielen und potentielle Fehlerquellen zu vermeiden, berücksichtigen Sie bitte bei der while-Anweisung dasselbe, was bereits bei der if-Anweisung in Kapitel 9.5.4 erläutert wurde. Benutzen Sie möglichst als Iterationsanweisung die Blockanweisung. Platzieren Sie dabei die öffnende und schließende Klammer der Blockanweisung untereinander und rücken Sie die inneren Anweisungen der Blockanweisung um zwei Spalten nach rechts ein, wie in folgendem Beispiel, in dem der Hamster bis zur nächsten Mauer läuft und dabei solange möglich eine Körnerspur legt:

```
void main()
{
    while (vorn_frei())
    {
        vor();
        if (!maul_leer())
        {
            gib();
        }
    }
}
```

10.2.5.2 Geschachtelte Schleifen

Die Iterationsanweisung einer while-Anweisung kann selbst wieder eine while-Anweisung sein oder eine while-Anweisung enthalten. Im folgenden Programm sammelt der Hamster alle Körner ein, die er auf seinem Weg bis zur nächsten Wand in Blickrichtung vor ihm findet (ausgenommen die Körner auf der letzten Kachel vor der Mauer):

```
void main()
{
    while (vorn_frei())
    {
        while (korn_da())
        {
            nimm();
        }
        vor();
    }
}
```

Derartige Konstruktionen nennt man *geschachtelte Schleifen*. Da sie gerade für Programmieranfänger leicht unübersichtlich werden, sollten Sie anfangs zunächst auf die Verwendung von Prozeduren zurückgreifen:

```
void main()
{
    while (vorn_frei())
    {
        sammle();
        vor();
    }
}

void sammle()
{
    while (korn_da())
    {
        nimm();
    }
}
```

10.2.5.3 Endlosschleifen

Schauen Sie sich das folgende Hamsterprogramm an, und stellen Sie sich vor, es würde bzgl. der in Abbildung 10.6 skizzierten Hamsterlandschaft gestartet:

```

void main()
{
    while (vorn_frei())
    {
        links_um();
    }
}

```

#	#	#	#	#	#
#	#				#
#	#		o	o	#
#		>		#	#
#		o		o	#
#	o	#	#		#
#	#	#	#	#	#

Abbildung 10.6: Endlosschleifen

Die Ausführung des Programms wird niemals enden. Da sich um den Hamster herum keine Mauern befinden und der Hamster bei der Ausführung der Iterationsanweisung die Kachel nie verläßt, wird die Schleifenbedingung immer den Wert `true` liefern.

Eine `while`-Anweisung, deren Schleifenbedingung immer den Wert `true` liefert, wird Endlosschleife genannt. Da eine Endlosschleife niemals endet, endet auch niemals das Programm, in dem die Schleife aufgerufen wird. Derartige Endlos-Programme sind im allgemeinen fehlerhaft, es sei denn, in der Aufgabenstellung wird dies explizit erlaubt. Endlosschleifen treten häufig auf, weil bestimmte Anweisungen innerhalb der Iterationsanweisung vergessen werden. Überprüfen Sie deshalb Ihre Programme, ob Situationen möglich sind, die zu einer Endlosschleife führen können. Ergreifen Sie Maßnahmen, die diesen Fehlerfall beseitigen.

Im obigen Beispiel werden Sie relativ schnell merken, daß sich das Programm in einer Endlosschleife befindet; der Hamster dreht sich nämlich fortwährend im Kreis. Das muß aber nicht immer der Fall sein. Im folgenden Programm hat der Programmierer innerhalb der Iterationsanweisung der `while`-Schleife der `main`-Prozedur den Befehl `vor()`; vergessen. Befindet sich der Hamster anfangs nicht vor einer Mauer, sammelt er zunächst alle Körner ein. Danach werden jeweils abwechselnd die beiden Schleifenbedingungen überprüft, ohne das der Hamster irgendwas für Sie Sichtbares tut; `vorn_frei()` liefert immer den Wert `true`, bedingt also die Endlosschleife, und `korn_da()` liefert immer den Wert `false`, da der Hamster ja bereits anfangs alle Körner gefressen hat.

```

void main()
{
    while (vorn_frei())
    {
        // sammle
        while (korn_da())
        {

```

```

        nimm();
    }
}
}

```

10.3 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.

10.3.1 Syntax

Dem Schlüsselwort `do`, von dem die Anweisung seinen Namen hat, folgt die Iterationsanweisung. Im Gegensatz zur while-Anweisung, bei der beliebigen Anweisungstypen zugelassen sind, wird die Iterationsanweisung bei der do-Anweisung immer durch eine Blockanweisung gebildet. Hinter der Blockanweisung muß das Schlüsselwort `while` stehen. Anschließend folgt in runden Klammern ein boolescher Ausdruck – die Schleifenbedingung. Abgeschlossen wird die do-Anweisung durch ein Semikolon. Abbildung 10.7 enthält das Syntaxdiagramm für die do-Anweisung. Die do-Anweisung ist wie die while-Anweisung eine Wiederholungsanweisung. Das Syntaxdiagramm „Wiederholungsanweisung“ aus Abbildung 10.2 wird daher in Abbildung 10.7 erweitert.

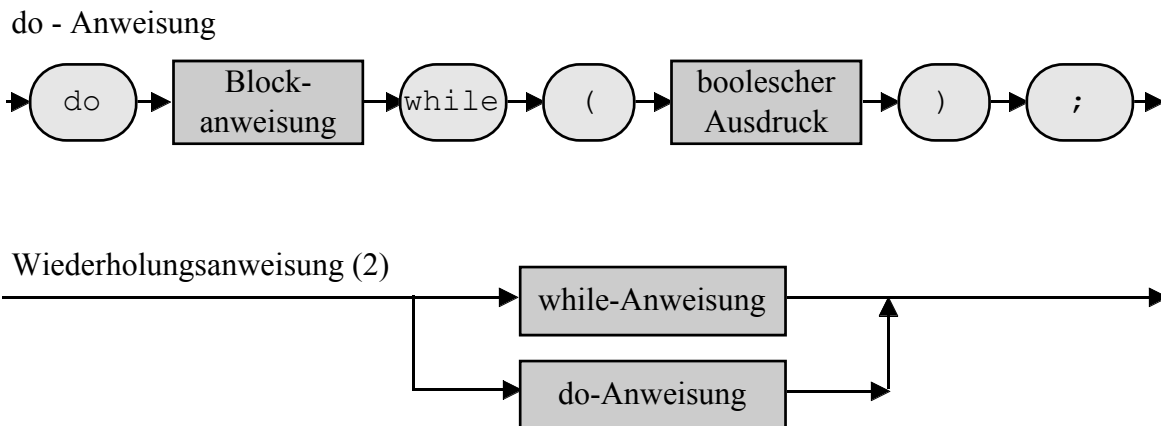


Abbildung 10.7: Syntaxdiagramm: do-Anweisung

10.3.2 Semantik

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 solange, bis irgendwann einmal die Schleifenbedingung nicht mehr erfüllt ist.

Wie Sie sehen, besteht der einzige Unterschied zwischen der `do`- und der `while`-Anweisung darin, daß bei der `do`-Anweisung die Iterationsanweisung mindestens einmal ausgeführt wird, was bei der `while`-Anweisung nicht unbedingt der Fall sein muß. In der Tat läßt sich jede `do`-Anweisung leicht durch eine `while`-Anweisung ersetzen. Sei `anw` Platzhalter für eine beliebige Anweisung und sei `ba` Platzhalter für einen beliebigen booleschen Ausdruck, dann sind die beiden folgenden Programmfragmente semantisch äquivalent, d.h. ihre Ausführungen haben dieselben Auswirkungen auf den Zustand eines Programmes:

```
// Programmfragment 1
do
{
    anw
} while (ba);

// Programmfragment 2
anw
while (ba)
{
    anw
}
```

10.3.3 Beispiele

10.3.3.1 Beispiel 1

Der Hamster habe eine bestimmte Anzahl (>0) an Körnern im Maul, die er alle ablegen soll:

```
void main()
{
    do
    {
        gib();
    } while (!maul_leer());
}
```

Da in der Aufgabenstellung vorgegeben wurde, daß die Anzahl an Körnern im Maul größer als Null ist, kann zur Lösung eine `do`-Anweisung verwendet werden. Der Hamster legt also zunächst ein Korn ab, bevor er die Schleifenbedingung ein erstes Mal testet.

10.3.3.2 Beispiel 2

Hamsteraufgabe Der Hamster befindet sich wie in Abbildung 10.8 dargestellt (die Abbildung skizziert zwei mögliche Landschaften) mit Blickrichtung Ost in der linken unteren Ecke eines rechteckigen durch Mauern abgeschlossenen sonst aber mauerlosen Raumes unbekannter Größe. Auf jeder Kachel innerhalb der Mauern befindet sich mindestens ein Korn. Der Hamster soll entlang der Mauern laufen und dabei alle Körner einsammeln. Alle Körner, die er im Maul hat, soll er anschließend in der linken unteren Ecke ablegen.

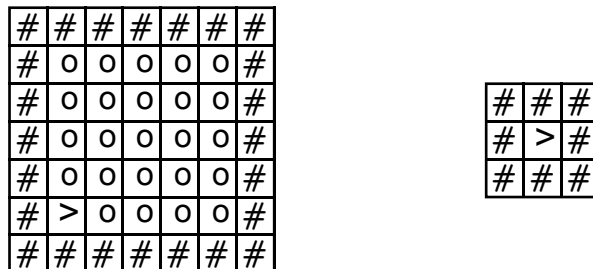


Abbildung 10.8: Typische Hamsterlandschaften zu Beispiel 2

Hamsterprogramm

```

void main()
{
    laufe_in_die_naechste_ecke_und_sammle();
    links_um();
    laufe_in_die_naechste_ecke_und_sammle();
    links_um();
    laufe_in_die_naechste_ecke_und_sammle();
    links_um();
    laufe_in_die_naechste_ecke_und_sammle();
    leg_ab();
}

void laufe_in_die_naechste_ecke_und_sammle()
{
    while (vorn_frei())
    {
        vor();
        sammle();
    }
}

void sammle()
{
    do
    {

```

```
    nimm();
} while (korn_da());
}

void leg_ab()
{
    do
    {
        gib();
    } while (!maul_leer());
}
```

Erläuterung Innerhalb der Prozedur `sammle` kann eine `do`-Anweisung benutzt werden, da laut Aufgabenstellung auf jeder Kachel mindestens ein Korn liegt und die Prozedur nur nach einem `vor()`-Befehl aufgerufen wird, d.h. der Hamster sich also bei Aufruf der Prozedur auf jeden Fall auf einer noch nicht „abgegrasten“ Kachel befindet. Dasselbe trifft auch für die Prozedur `leg_ab()` zu; denn da der Hamster mindestens einmal „sammelt“, hat er auf jeden Fall Körner im Maul, bevor die Prozedur aufgerufen wird. Für die Prozedur `laufe_in_die_naechste_ecke_und_sammle` ist die Verwendung einer `do`-Anweisung jedoch nicht möglich. Hier muß eine `while`-Anweisung benutzt werden, da die Landschaft durchaus auch aus nur einer freien Kachel bestehen kann (siehe Abbildung 10.8 (rechts)). Bei Verwendung einer `do`-Anweisung anstelle der `while`-Anweisung würde die Programmausführung bzgl. der Landschaft aus Abbildung 10.8 (rechts) zu einem Laufzeitfehler führen, was gemäß Abschnitt 10.2.3 bedeutet, daß das Programm nicht korrekt ist.

10.4 Beispielprogramme

In diesem Abschnitt werden einige Beispiele für Hamster-Programme gegeben, die Ihnen den Einsatz von Wiederholungsanweisungen demonstrieren sollen. Es werden jeweils eine oder mehrere Musterlösungen vorgestellt. Schauen Sie sich die Beispiele genau an und versuchen Sie, die Lösungen nachzuvollziehen.

10.4.1 Beispielprogramm 1

Aufgabe:

In einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern sind wahllos eine unbekannte Anzahl an Körnern verstreut (siehe Beispiele in Abbildung 10.9). Der Hamster, der sich zu Anfang in der linken unteren Ecke des Hamster-Territoriums mit Blickrichtung Ost befindet, soll alle Körner aufsammeln und dann stehenbleiben.

Lösung:

```
void main()
{
    ernte_eine_reihe_und_laufe_zurueck();
}
```

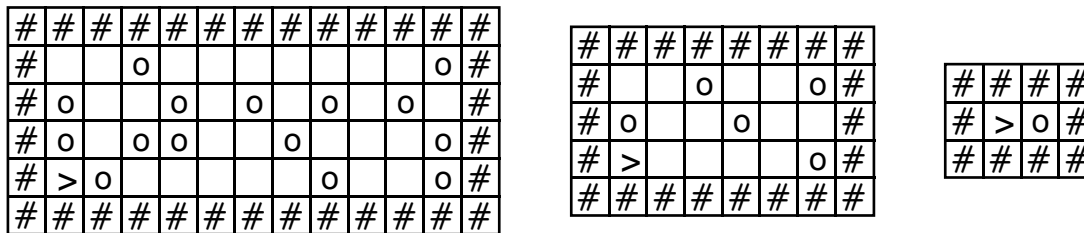


Abbildung 10.9: Typische Hamsterlandschaften zu Beispielprogramm 1

```

rechts_um();
while (vorn_frei())
{
    vor(); rechts_um();
    ernte_eine_reihe_und_laufe_zurueck();
    rechts_um();
}
}

void ernte_eine_reihe_und_laufe_zurueck()
{
    ernte_eine_reihe();
    kehrt();
    laufe_zurueck();
}

void ernte_eine_reihe()
{
    sammle();
    while (vorn_frei())
    {
        vor();
        sammle();
    }
}

void laufe_zurueck()
{
    while (vorn_frei())
    {
        vor();
    }
}

void sammle()
{
    while (korn_da())

```

```

    {
        nimm();
    }
}

void rechts_um()
{
    kehrt(); links_um();
}

void kehrt()
{
    links_um(); links_um();
}

```

10.4.2 Beispielprogramm 2

Aufgabe:

Der Hamster steht irgendwo in einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern. Auf keinem der Felder liegt ein Korn. Der Raum habe eine Mindestgröße von 4x4 Kacheln (siehe Beispiele in Abbildung 10.10). Der Hamster, der mindestens 4 Körner im Maul hat, soll in allen vier Ecken des Feldes je ein Korn ablegen.

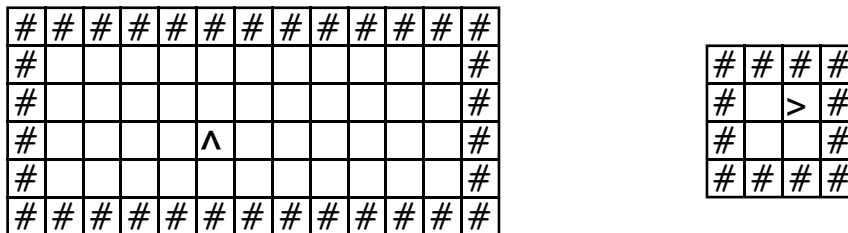


Abbildung 10.10: Typische Hamsterlandschaften zu Beispielprogramm 2

Lösung:

```

void main()
{
    begib_dich_in_eine_ecke();
    // der Hamster kann leider (noch) nicht zaehlen; er fuehrt deshalb
    // viermal dieselbe Anweisungssequenz aus
    gib(); laufe_in_die_naechste_ecke(); links_um();
    gib(); laufe_in_die_naechste_ecke(); links_um();
    gib(); laufe_in_die_naechste_ecke(); links_um();
    gib();
}

void begib_dich_in_eine_ecke()
{

```

```

    laufe_zur_naechsten_wand();
    links_um();
    laufe_zur_naechsten_wand();
    links_um();
}

void laufe_zur_naechsten_wand()
{
    while (vorn_frei())
    {
        vor();
    }
}

void laufe_in_die_naechste_ecke()
{
    laufe_zur_naechsten_wand();
}

```

10.4.3 Beispielprogramm 3

Aufgabe:

Der Hamster steht – wie in den Beispielen in Abbildung 10.11 skizziert – vor einem regelmäßigen Berg unbekannter Höhe. Er soll den Gipfel erklimmen und dann stehenbleiben.

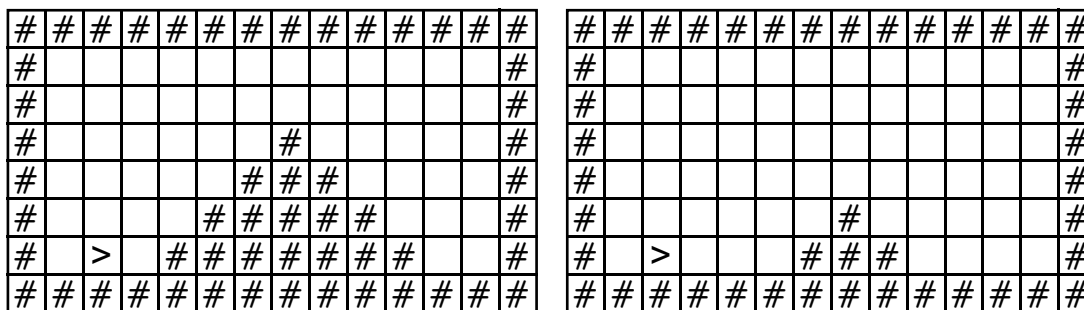


Abbildung 10.11: Typische Hamsterlandschaften zu Beispielprogramm 3

Lösung:

```

void main()
{
    laufe_zum_berg();
    erklimme_gipfel();
}

void laufe_zum_berg()
{

```

```
    while (vorn_frei())
    {
        vor();
    }
}

void erklimme_gipfel()
{
    do
    {
        erklimme_eine_stufe();
    } while (!vorn_frei());
}

void erklimme_eine_stufe()
{
    links_um();
    vor();
    rechts_um();
    vor();
}

void rechts_um()
{
    kehrt(); links_um();
}

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

10.5 Übungsaufgaben

Nun sind wieder Sie gefordert; denn in diesem Abschnitt werden Ihnen einige Hamster-Aufgaben gestellt, die sie selbständig zu lösen haben. Achten Sie darauf, daß bei den Aufgaben keine Landschaften mehr fest vorgegeben sind wie in den vergangenen Kapiteln, sondern daß nur noch spezifische Merkmale von möglichen Ausgangslandschaften angegeben werden. Ihre Hamsterprogramme müssen für alle Landschaften korrekt arbeiten, die dieser Charakterisierung entsprechen.

10.5.1 Aufgabe 1

Der Hamster steht – wie schon in Beispielprogramm 3 (siehe auch Abschnitt 10.4.3) – vor einem regelmäßigen Berg unbekannter Höhe. Es liegen keine Körner im Territorium. Der Hamster, der

anfangs 1 Korn im Maul hat, soll den Gipfel erklimmen, sich umdrehen, wieder hinabsteigen und an seiner Ausgangsposition stehenbleiben.

10.5.2 Aufgabe 2

Der Hamster steht – wie in Abbildung 10.12 skizziert – irgendwo in einem abgeschlossenen ansonsten aber mauerlosen rechteckigen Raum unbekannter Größe. Alle Kacheln sind körnerlos. Der Hamster hat mindestens so viele Körner im Maul, wie es Kacheln auf dem Feld gibt. Der Hamster soll auf allen Kacheln des Territoriums genau ein Korn ablegen und schließlich stehenbleiben. Entwickeln Sie zwei verschiedene Hamster-Programme, die diese Aufgabe lösen.

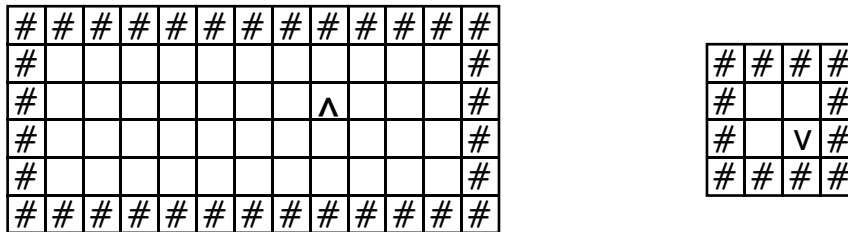


Abbildung 10.12: Typische Hamsterlandschaften zu Aufgabe 2

10.5.3 Aufgabe 3

Ändern Sie Beispielprogramm 2 (siehe Abschnitt 10.4.2) so ab, daß die Einschränkung des Hamster-Territoriums auf eine Mindestgröße von 4x4 Kacheln entfallen kann, d.h. das Programm soll auch dann korrekt arbeiten, wenn der Hamster anfangs auf dem einzig freien Feld des Territoriums steht oder wenn es lediglich eine einzige freie Reihe gibt.

10.5.4 Aufgabe 4

Denken Sie sich selbst weitere Hamster-Aufgaben aus, und versuchen Sie, diese zu lösen. Verwenden Sie dabei intensiv Wiederholungsanweisungen. Viel Spaß!

Kapitel 11

Boolesche Funktionen

11.1 Motivation

Der Hamster hat einen sehr begrenzten Grundvorrat an Befehlen (`vor()`;, `links_um()`;, `nimm()`;, `gib()`;) und Testbefehlen (`vorn_frei()`;, `maul_leer()`;, `korn_da()`). In Kapitel 8 haben Sie gelernt, wie mittels der Definition von Prozeduren dem Hamster weitere Befehle beigebracht werden können. In diesem Kapitel werden Sie einen Mechanismus kennenlernen, den Vorrat an Testbefehlen zu erweitern. Dazu werden sogenannte *boolesche Funktionen* oder *Testfunktionen* eingeführt.

Stellen Sie sich vor, Sie möchten einen neuen Testbefehl `mauer_da()` definieren, der genau dann den Wert `true` liefert, wenn sich direkt in Blickrichtung vor dem Hamster eine Mauer befindet. Im Prinzip entspricht dies dem booleschen Ausdruck `!vorn_frei()`, denn dieser liefert ja das gewünschte Ergebnis. Prozeduren können Ihnen hier nicht weiterhelfen, denn diese können ja keinen Wert liefern. Was Sie benötigen ist ein Sprachkonstrukt, über das Sie einen neuen Namen `mauer_da` für den neuen Testbefehl einführen und das immer dann, wenn Sie den Testbefehl `mauer_da()` aufrufen, den Wert des booleschen Ausdrucks `!vorn_frei()` liefert.

Etwas komplizierter ist die folgende Situation: der Hamster soll ermitteln, ob sich links von ihm eine Mauer befindet oder nicht. Es soll also ein neuer Testbefehl `links_frei()` eingeführt werden. Ein solcher Testbefehl könnte dadurch realisiert werden, daß sich der Hamster zunächst nach links umdreht. Anschließend kann er mit Hilfe des Testbefehls `vorn_frei()` überprüfen, ob die Kachel vor ihm frei ist. Falls der Aufruf des Testbefehls `vorn_frei()` den Wert `true` liefert, muß auch der Testbefehl `links_frei()` den Wert `true` liefern, für den `false`-Fall gilt entsprechendes. Zu beachten ist jedoch, daß sich der Hamster in beiden Fällen noch wieder nach rechts umdrehen muß, um in seine Ausgangsposition zurück zu gelangen.

Beide Beispiele lassen sich mit Hilfe sogenannter *boolescher Funktionen* realisieren. Bevor wir deren Syntax und Semantik kennenlernen, wird zunächst die *boolesche return-Anweisung* eingeführt.

11.2 Boolesche return-Anweisung

Boolesche return-Anweisungen werden in booleschen Funktionen zum Liefern eines booleschen Wertes benötigt. Die boolesche return-Anweisung wird in Kapitel 14 erweitert.

11.2.1 Syntax

Die Syntax der booleschen return-Anweisung ist sehr einfach: Dem Schlüsselwort `return` folgt ein boolescher Ausdruck und ein abschließendes Semikolon; Die boolesche return-Anweisung darf ausschließlich im Funktionsrumpf boolescher Funktionen (siehe Abschnitt 11.3) verwendet werden. Die genaue Syntax wird in Abbildung 11.1 verdeutlicht. In Abbildung 11.1 wird auch das Syntaxdiagramm „Anweisung“ aus Abbildung 10.2 erweitert.

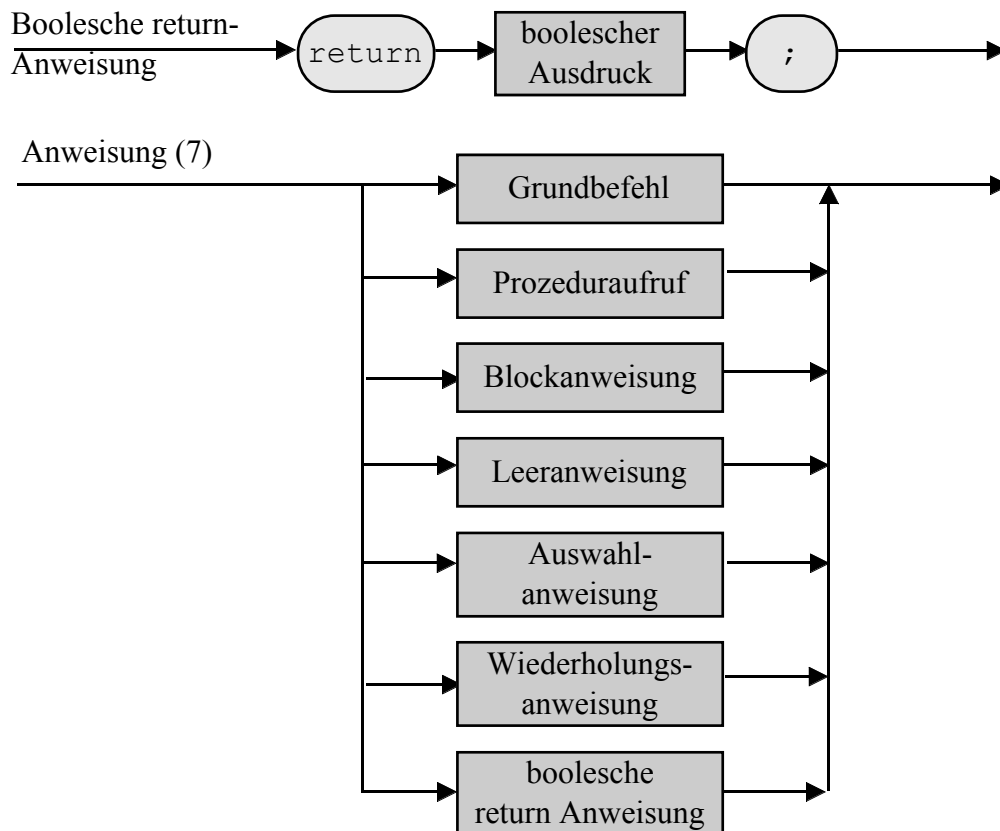


Abbildung 11.1: Syntaxdiagramm: Boolesche return-Anweisung

11.2.2 Semantik

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 sogenannter Funktionswert zurückgegeben.

11.2.3 Beispiele

Im folgenden werden einige Beispiele für syntaktisch korrekte boolesche return-Anweisungen gegeben:

- `return true;`
- `return vorn_frei();`
- `return maul_leer || vorn_frei();`
- `return (korn_da() && !vorn_frei());`

11.3 Definition boolescher Funktionen

Genauso wie Prozeduren müssen auch boolesche Funktionen definiert werden, um sie in anderen Teilen des Programmes aufrufen zu können.

11.3.1 Syntax

Die Syntax der Definition einer booleschen Funktion unterscheidet sich nur geringfügig von der Definition einer Prozedur (siehe auch Abbildung 11.2). Statt Prozedurkopf, -name und -rumpf spricht man hier von Funktionskopf, -name und -rumpf.

Anstelle des Schlüsselwortes `void` bei der Definition einer Prozedur muß bei der Definition einer booleschen Funktion das Schlüsselwort `boolean` am Anfang des Funktionskopfs stehen. Außerdem können im Funktionsrumpf boolesche return-Anweisungen verwendet werden.

Ganz wichtig bei der Definition boolescher Funktionen ist jedoch folgende Zusatzbedingung, die sich mit Hilfe von Syntaxdiagrammen nicht ausdrücken läßt und deshalb verbal ergänzt wird: In jedem möglichen Weg durch die Funktion bei ihrer Ausführung muß eine boolesche return-Anweisung auftreten. Der Wert, den der boolesche Ausdruck einer booleschen return-Anweisung liefert, ist der sogenannte *Funktionswert* der booleschen Funktion.

Boolesche Funktionen können überall dort in einem Hamsterprogramm definiert werden, wo auch Prozeduren definiert werden können.

In Abbildung 11.2 wird auch das Syntaxdiagramm „Definitionen“ aus Abbildung 8.4 erweitert.

11.3.2 Semantik

Durch die Definition einer booleschen Funktion innerhalb eines Hamsterprogrammes wird ein neuer Testbefehl eingeführt, der über den Namen der Funktion aufgerufen werden kann. Ansonsten hat die Definition einer booleschen Funktion keine direkten Auswirkungen auf die Ausführung eines Programms.

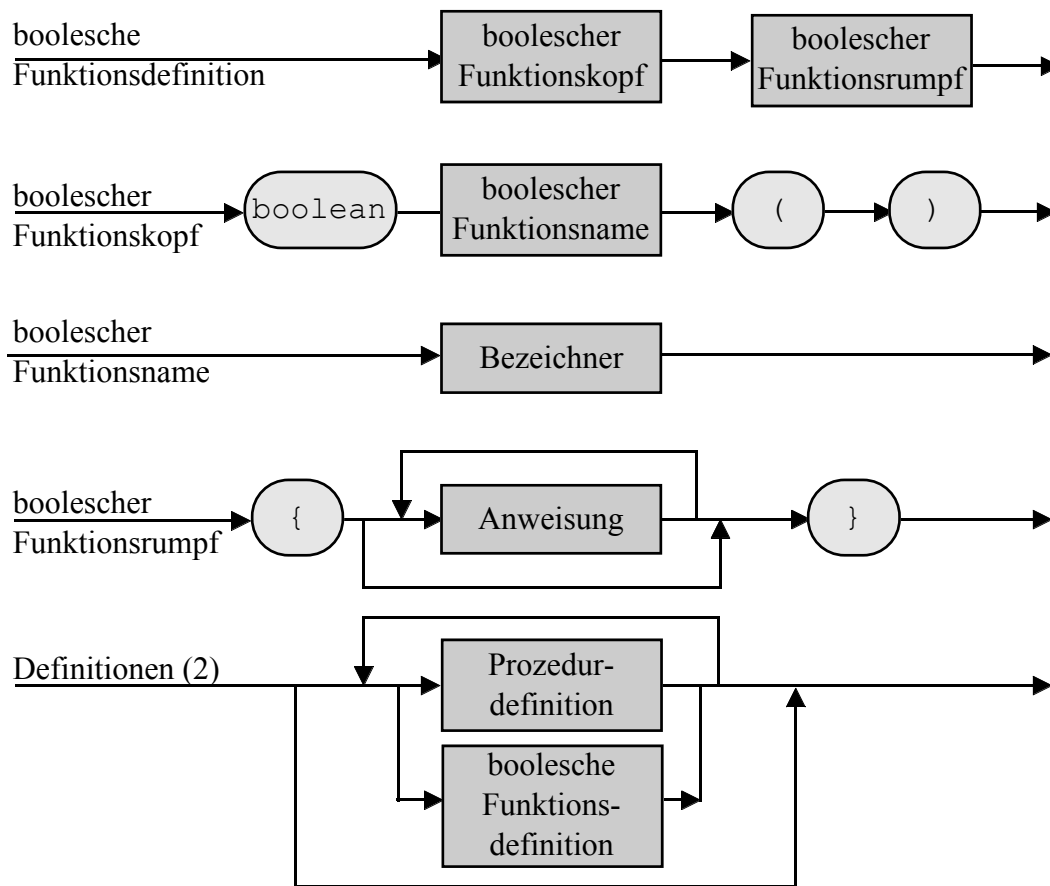


Abbildung 11.2: Syntaxdiagramm: Boolesche Funktionsdefinition

11.3.3 Beispiele

In den folgenden beiden Funktionsdefinitionen werden die im obigen Motivationsabschnitt 11.1 geschilderten Beispiele implementiert. Im Rumpf der Funktion `links_frei()` wird dabei die ebenfalls definierte Prozedur `rechts_um` aufgerufen.

```

boolean mauer_da()
{
    return !vorn_frei();
}

```

```

boolean links_frei()
{
    links_um();
    if (vorn_frei())
    {
        rechts_um();
        return true;
    }
    else

```

```
    {
        rechts_um();
        return false;
    }
}

void rechts_um()
{
    links_um(); links_um(); links_um();
}
```

Die folgende Funktionsdefinition ist dahingegen fehlerhaft, denn bei der Ausführung des else-Teils der if-Anweisung endet die Funktion, ohne daß eine boolesche return-Anweisung ausgeführt worden ist:

```
boolean nichts_geht_mehr()
{
    if (vorn_frei)
    {
        vor();
        return false;
    }
    else
    {
        links_um();
    }
}
```

Im allgemeinen entdeckt der Compiler derartige Fehler und gibt Fehlermeldungen bzw. Warnungen aus. Wir werden jedoch in Kapitel 13 Fälle kennenlernen, wo der Compiler derartige Fehler nicht erkennen kann. Daraus resultieren dann spezielle Laufzeitfehler.

Was der Compiler auch nicht „mag“, sind Anweisungen innerhalb von Funktionsrümpfen, die gar nicht erreicht werden können, weil die Funktion auf jeden Fall vorher verlassen wird. Schauen Sie sich folgendes Beispiel an:

```
boolean hinten_frei()
{
    links_um();
    links_um();
    if (vorn_frei())
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```

    }
    links_um();
    links_um();
}

```

Die Funktion `hinten_frei` wird in jedem Fall in der `if`-Anweisung verlassen, da bei ihrer Ausführung sowohl in der `true`-Blockanweisung als auch in der `false`-Blockanweisung eine boolesche `return`-Anweisung erreicht wird. Die beiden anschließenden `links_um()`-Befehle können also unmöglich erreicht werden. Auch hier liefert der Compiler im allgemeinen eine Fehlermeldung oder Warnung.

Die `if`-Anweisung im vorangehenden Beispiel ist zwar syntaktisch korrekt aber umständlich formuliert. Häufig sieht man einen derartigen Programmstil bei Programmieranfänger. Es gilt: Sei `bA` Platzhalter für einen beliebigen booleschen Ausdruck. Dann sind folgende Anweisungen semantisch äquivalent:

```

// Anweisung 1
if (bA)
{
    return true;
}
else
{
    return false;
}

// Anweisung 2
return bA;

```

Die zweite Variante ist wegen ihrer Kompaktheit besser verständlich und der ersten Variante vorzuziehen.

11.4 Aufruf boolescher Funktionen

Durch die Definition einer booleschen Funktion wird ein neuer Testbefehl eingeführt. Der Aufruf einer booleschen Funktion wird auch Funktionsaufruf genannt.

11.4.1 Syntax

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. Abbildung 11.3 definiert die genaue Syntax des Aufrufs boolescher Funktionen und erweitert das Syntaxdiagramm „boolescher Ausdruck“ aus Abbildung 9.3.

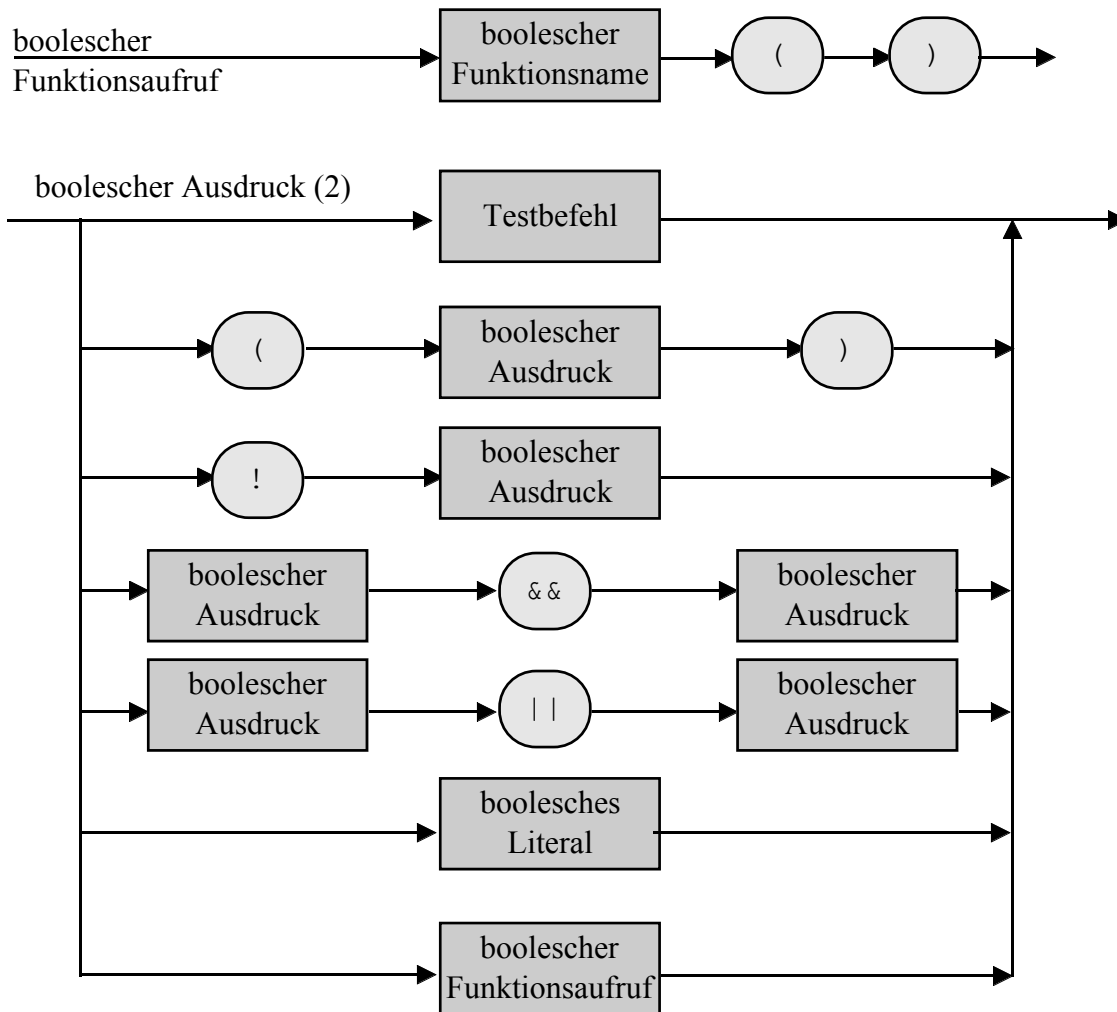


Abbildung 11.3: Syntaxdiagramm: Boolescher Funktionsaufruf

11.4.2 Semantik

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 Funktionsaufrufes zurückgesprungen. Der von der booleschen return-Anweisung gelieferte Wert (also der Funktionswert) wird dabei zur Berechnung des booleschen Ausdrucks weiterverwendet.

11.4.3 Beispiele

11.4.3.1 Beispiel 1

Schauen Sie sich folgendes Beispielprogramm an:

```

boolean mauer_da()
{
    return !vorn_frei();
}

void main()
{
    if (mauer_da())
    {
        links_um();
    }
}

```

Die boolesche Funktion `mauer_da` wird bei der Formulierung der Bedingung der `if`-Anweisung in der `main`-Prozedur benutzt. Der Hamster stehe wie in Abbildung 11.4 (links) ersichtlich auf der Hamsterlandschaft. Dann wird zunächst die Funktion `mauer_da` aufgerufen. Da die Funktion lediglich aus einer booleschen `return`-Anweisung besteht, wird deren boolescher Ausdruck ausgewertet. Es ergibt sich in der skizzierten Situation der Wert `true`, der als Funktionswert zurückgeliefert wird. Das bedeutet, die Auswahlbedingung ist erfüllt und der `links_um();`-Befehl wird ausgeführt (siehe Abbildung 11.4 (rechts)).

#	#	#	#	#	#	#	#	#	#	#
#	#				o					#
#	#		o	o			#	o		#
#				#			#	#	#	#
#		o				>	#	#	#	#
#	o	#	#						o	#
#	#	#	#	#	#	#	#	#	#	#

#	#	#	#	#	#	#	#	#	#	#
#	#				o					#
#	#		o	o			#	o		#
#				#			#	#	#	#
#		o					^	#	#	#
#	o	#	#						o	#
#	#	#	#	#	#	#	#	#	#	#

Abbildung 11.4: Aufruf boolescher Funktionen

11.4.3.2 Beispiel 2

Im folgenden Beispiel sucht der Hamster eine Nische an seiner linken Seite. Falls er eine solche findet, begibt er sich in die Nische.

```

void main()
{
    while (vorn_frei() && !links_frei())
    {
        vor();
    }
    if (links_frei())
    {
        links_um();
    }
}

```

```
        vor();
    }
}

boolean links_frei()
{
    links_um();
    if (vorn_frei())
    {
        rechts_um();
        return true;
    }
    else
    {
        rechts_um();
        return false;
    }
}

void rechts_um()
{
    links_um(); links_um(); links_um();
}
```

An diesem Beispiel können sie visuell nachvollziehen, was Sie bezüglich der Auswertungsreihenfolge von booleschen Ausdrücken in Kapitel 9.2.5 gelernt haben, daß der Hamster nämlich die Auswertung bestimmter boolescher Ausdrücke optimiert. Schauen Sie sich dazu die in Abbildung 11.5 geschilderte Ausgangssituation an. Bei der Auswertung der Schleifenbedingung der while-Anweisung innerhalb der main-Prozedur wird zunächst der Wert des Testbefehls `vorn_frei` ermittelt. Der Testbefehl liefert den Wert `true`. Also wird als nächstes der Funktionswert der Funktion `links_frei` ermittelt, der negiert den Wert des booleschen Ausdrucks der Schleifenbedingung ergibt. Zur Ermittlung des Funktionswertes wird die Funktion aufgerufen: der Hamster dreht sich um 90 Grad nach links, testet, ob nun vor ihm frei ist; das ist nicht der Fall, also dreht er sich wieder um 90 Grad nach rechts und liefert den Wert `false`. Durch die anschließende Negierung ist die Schleifenbedingung insgesamt erfüllt, und die Iterationsanweisung wird ausgeführt, d.h. der Hamster hüpft eine Kachel nach vorne. Nun wird die Schleifenbedingung ein weiteres Mal überprüft. Da dieses Mal aber bereits der Testbefehl `vorn_frei()` den Wert `false` liefert, muß die Funktion `links_frei` nicht ein zweites Mal aufgerufen werden. Die Schleifenbedingung kann nicht mehr erfüllt werden. Die Funktion `links_frei` wird dann jedoch für die Auswertung der Bedingung der if-Anweisung erneut aufgerufen. Sie liefert den Wert `true`, so daß die Bedingung erfüllt ist und sich der Hamster in die Nische begibt.

Drehen Sie nun einmal die konjugierten Ausdrücke in dem Beispiel um und führen Sie das Programm erneut für die in Abbildung 11.6 skizzierte Situation aus. Sie werden feststellen, daß der Hamster einmal mehr testet, ob sich links von ihm eine Mauer befindet.

```
void main()
```

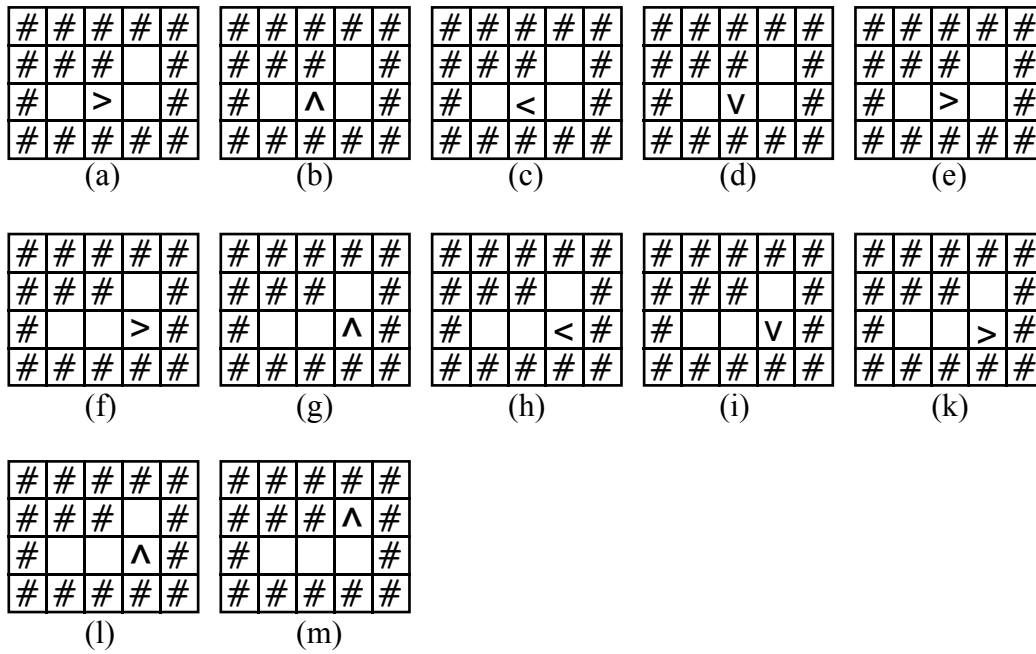


Abbildung 11.5: Auswertungsgreihenfolge boolescher Ausdrücke

```

{
  while (!links_frei() && vorn_frei())
  {
    vor();
  }
  if (links_frei())
  {
    links_um();
    vor();
  }
}

boolean links_frei()
{
  links_um();
  if (vorn_frei())
  {
    rechts_um();
    return true;
  }
  else
  {
    rechts_um();
    return false;
  }
}

```

```

void rechts_um()
{
  links_um();
  links_um();
  links_um();
}

```

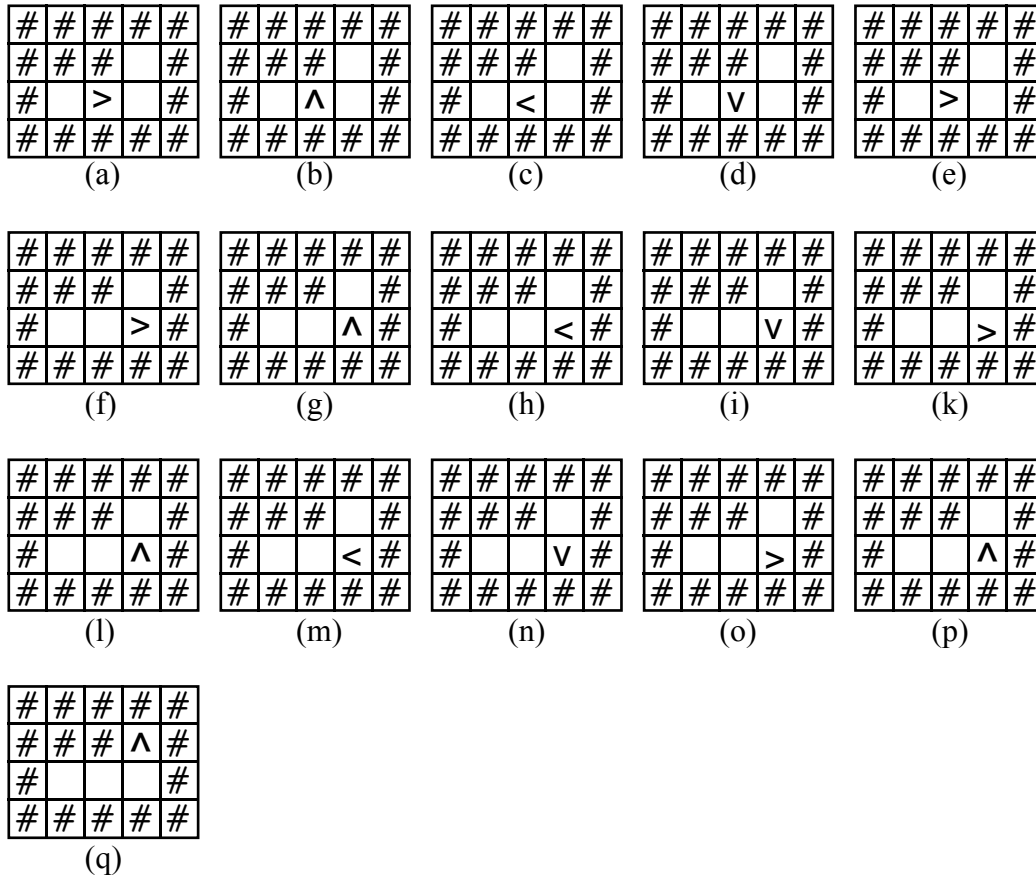


Abbildung 11.6: Auswertungsreihenfolge boolescher Ausdrücke

11.5 Seiteneffekte

Der Unterschied zwischen Prozeduren und booleschen Funktionen besteht nach außen hin darin, daß Prozeduren etwas tun, indem sie den Zustand des Hamsters (Blickrichtung, Position, Körneranzahl im Maul) oder der Hamsterlandschaft (Körneranzahl auf den einzelnen Kacheln) verändern, während boolesche Funktionen etwas berechnen, nämlich einen booleschen Wert. Zur Berechnung dieses Wertes müssen sie jedoch in der Regel intern auch etwas tun. Führt ein Funktionsaufruf nicht nur zur Berechnung eines Wertes, sondern darüber hinaus dazu, daß sich nach Beendigung der Funktion der Zustand des Hamsters bzw. der Hamsterlandschaft geändert hat, dann spricht man davon, daß die Funktion einen sogenannten *Seiteneffekt* produziert hat.

Die folgende boolesche Funktion `links_frei()` produziert beispielsweise immer einen Seiteneffekt; denn nach ihrer Ausführung hat der Hamster eine andere Blickrichtung als vorher.

```
boolean links_frei()
{
    links_um();
    return vorn_frei();
}
```

Genauso wie die drei vordefinierten Testbefehle keine Zustandsänderungen bewirken, also niemals Seiteneffekte produzieren, sollten auch boolesche Funktionen in der Regel keine Seiteneffekte hervorrufen, d.h. sollten zur Berechnung eines zu liefernden Wertes innerhalb einer Funktion Zustandsänderungen notwendig sein, so sollten diese vor dem Verlassen der Funktion wieder rückgängig gemacht werden. Seiteneffektfreie Funktionen führen zu besser lesbaren und weniger fehleranfälligen Programmen. Definieren Sie eine boolesche Funktion `links_frei` also immer folgendermaßen:

```
boolean links_frei()
{
    links_um();
    if (vorn_frei())
    {
        rechts_um();
        return true;
    }
    else
    {
        rechts_um();
        return false;
    }
}

void rechts_um()
{
    links_um(); links_um(); links_um();
}
```

Wenn Sie – aus welchem Grund auch immer – dennoch seiteneffektproduzierende Funktionen definieren, sollten Sie auf jeden Fall durch einen Kommentar darauf hinweisen.

In Kapitel 9.6.4 wurde bereits erwähnt, daß die folgenden beiden Anweisungen nicht unbedingt semantisch äquivalent sein müssen. Dabei sei `bA` Platzhalter für einen beliebigen booleschen Ausdruck, und `a1` und `a2` seien Platzhalter für beliebige Anweisungen:

```
// Anweisung 1
if (bA)
```

```

    a1;
else
    a2;

// Anweisung 2
if (bA)
    a1;
if (!bA)
    a2;

```

Mit Hilfe der Seiteneffekte läßt sich dieses Phänomen nun auch verstehen. Konkretisieren wir die Anweisungen dadurch, daß wir für **bA** die oben definierte seiteneffektproduzierende boolesche Funktion `links_frei` verwenden und die beiden Anweisungen `a1` und `a2` durch die Leeranweisung ersetzen:

```

// Anweisung 1
if (links_frei())
    ;
else
    ;

// Anweisung 2
if (links_frei())
    ;
if (!links_frei())
    ;

```

Werden die Anweisungen jeweils in der in Abbildung 11.7 (a) skizzierten Situation aufgerufen, so ergibt sich nach Ausführung von Anweisung 1 der in Abbildung 11.7 (b) und nach Ausführung von Anweisung 2 der in Abbildung 11.7 (c) dargestellte Zustand. Die beiden Zustände sind nicht gleich, die Anweisungen also nicht äquivalent.

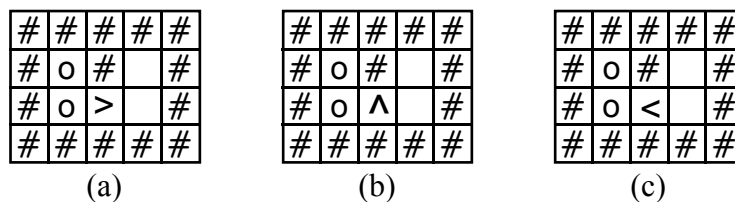


Abbildung 11.7: Seiteneffekte

11.6 Beispielprogramme

In diesem Abschnitt werden einige Beispiele für Hamster-Programme gegeben, die Ihnen den Einsatz von booleschen Funktionen demonstrieren sollen. Es werden jeweils eine oder mehrere Musterlösungen vorgestellt. Schauen Sie sich die Beispiele genau an, und versuchen Sie, die Lösungen nachzuvollziehen.

11.6.1 Beispielprogramm 1

Aufgabe:

In einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern sind wahllos eine unbekannte Anzahl an Körnern verstreut (siehe Beispiele in Abbildung 11.8). Der Hamster, der sich zu Anfang in der linken unteren Ecke des Hamster-Territoriums mit Blickrichtung Ost befindet, soll alle Körner aufsammeln und dann stehenbleiben.

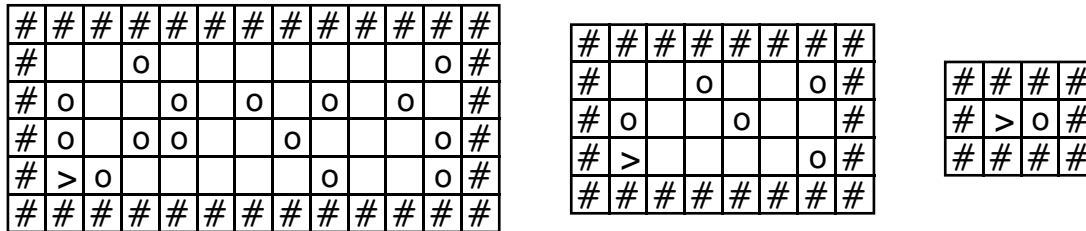


Abbildung 11.8: Typische Hamsterlandschaften zu Beispielprogramm 1

Dieses Beispielprogramm hatten wir bereits in Kapitel 10.4.1 als Beispielprogramm 1 gelöst; dort allerdings ohne den Einsatz von booleschen Funktionen. Dieses Mal lösen wir das Problem mit Hilfe boolescher Funktionen. Sie werden feststellen, daß der Algorithmus so verständlicher und besser nachvollziehbar ist.

Lösung:

```
void main()
{
    ernte_eine_reihe_und_laufe_zurueck();
    while (weitere_reihe_existiert())
    {
        begib_dich_in_naechste_reihe();
        ernte_eine_reihe_und_laufe_zurueck();
    }
}

boolean weitere_reihe_existiert() {
    rechts_um();
    if (vorn_frei()) {
        // Achtung: der Hamster muss sich wieder in seine
        //           Ausgangsstellung begeben
        links_um();
        return true;
    } else {
        links_um();
        return false;
    }
}

void ernte_eine_reihe_und_laufe_zurueck()
```

```
{
    ernte_eine_reihe();
    kehrt();
    laufe_zurueck();
}

void ernte_eine_reihe()
{
    sammle();
    while (vorn_frei())
    {
        vor();
        sammle();
    }
}

void begib_dich_in_naechste_reihe()
{
    rechts_um(); vor(); rechts_um();
}

void laufe_zurueck()
{
    while (vorn_frei())
    {
        vor();
    }
}

void sammle()
{
    while (korn_da())
    {
        nimm();
    }
}

void rechts_um()
{
    kehrt(); links_um();
}

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



```
    {
        vor();
        sammle();
    }
}

void sammle()
{
    while (korn_da())
    {
        nimm();
    }
}

// Achtung: Die Funktion produziert Seiteneffekte!
boolean begib_dich_in_naechste_reihe_von_ost()
{
    kehrt();
    // finde naechste Nische
    while (vorn_frei() && !links_frei())
    {
        vor();
    }
    if (!links_frei())
    {
        // Ende der Mulde erreicht
        return false;
    }
    else
    {
        // begib dich in naechste Reihe
        links_um();
        vor();
        rechts_um();
        return true;
    }
}

// Achtung: Die Funktion produziert Seiteneffekte!
boolean begib_dich_in_naechste_reihe_von_west()
{
    kehrt();
    // finde naechste Nische
    while (vorn_frei() && !rechts_frei())
    {
        vor();
    }
}
```

```
if (!rechts_frei())
{
    // Ende der Mulde erreicht
    return false;
}
else
{
    // begib dich in naechste Reihe
    rechts_um();
    vor();
    links_um();
    return true;
}
}
```

```
boolean links_frei()
{
    links_um();
    if (vorn_frei())
    {
        rechts_um();
        return true;
    }
    else
    {
        rechts_um();
        return false;
    }
}
```

```
boolean rechts_frei()
{
    rechts_um();
    if (vorn_frei())
    {
        links_um();
        return true;
    }
    else
    {
        links_um();
        return false;
    }
}
```

```
void rechts_um()
{
```

```

    kehrt(); links_um();
}

void kehrt()
{
    links_um(); links_um();
}

```

11.6.3 Beispielprogramm 3

Aufgabe:

Der Hamster, der genau ein Korn in seinem Maul hat, befindet sich in einem einem geschlossenen, körnerlosen Raum unbekannter Größe. Rechts von ihm befindet sich eine Wand, und vor ihm das Feld ist frei (siehe Beispiel in Abbildung 11.10). Der Hamster soll solange an der Wand entlang laufen bis er irgendwann wieder sein Ausgangsfeld erreicht.

#	#	#	#	#	#	#	#	#	#	#	#	#	#
#	#				#				#				#
#		#			#								#
#												#	#
#				#	#	#	#			#			#
#			#		#		#	#	#				#
#	>		#					#	#				#
#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 11.10: Typische Hamsterlandschaft zu Beispielprogramm 3

Lösung:

```

void main()
{
    gib(); // markiere Ausgangsposition
    vor();
    while (!korn_da()) // irgendwann kehrt der Hamster hierher zurueck
    {
        while (vorn_frei() && !rechts_frei() && !korn_da())
        {
            vor();
        }
        if (!korn_da()) // Ausgangsposition erreicht
        {
            if (rechts_frei())
            {
                rechts_um();
                vor();
            }
        }
    }
}

```

```
        else // vorne und rechts stehen Mauern
        {
            links_um();
        }
    }
}

boolean rechts_frei()
{
    rechts_um();
    if (vorn_frei())
    {
        links_um();
        return true;
    }
    else
    {
        links_um();
        return false;
    }
}

void rechts_um()
{
    links_um(); links_um(); links_um();
}
```

11.7 Übungsaufgaben

Nun sind wieder Sie gefordert; denn in diesem Abschnitt werden Ihnen einige Hamster-Aufgaben gestellt, die sie selbständig zu lösen haben.

11.7.1 Aufgabe 1

Der Hamster befindet sich irgendwo in einem quadratischen geschlossenen, körnerlosen Raum unbekannter Größe ohne innere Mauern. Der Hamster soll die beiden Diagonalen des Territoriums mit jeweils einem Korn kennzeichnen (siehe Abbildung 11.11).

11.7.2 Aufgabe 2

Ähnlich wie in Beispielprogramm 3 in Kapitel 10.4.3 steht der Hamster vor einem Berg unbekannter Höhe. Allerdings ist der Berg diesmal nicht regelmäßig eine Stufe hoch, sondern die Stufenhöhen und -längen können variieren wie in Abbildung 11.12 skizziert. Es gibt jedoch keine Überhänge! Der Hamster soll den Gipfel suchen und schließlich auf dem Gipfel anhalten.

#	#	#	#	#	#
#	o			o	#
#		o	o		#
#		o	o		#
#	o			o	#
#	#	#	#	#	#

#	#	#	#	#
#	o		o	#
#		o		#
#	o		o	#
#	#	#	#	#

Abbildung 11.11: Typische Hamsterlandschaften zu Aufgabe 1

#	#	#	#	#	#	#	#	#	#	#	#	#	#
#													#
#													#
#						#	#						#
#				#	#	#	#						#
#				#	#	#	#						#
#	>		#	#	#	#	#	#					#
#	#	#	#	#	#	#	#	#	#	#	#	#	#

#	#	#	#	#	#	#	#	#	#	#	#	#	#
#													#
#													#
#						#							#
#						#							#
#						#	#						#
#	>	#	#	#	#	#	#						#
#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 11.12: Typische Hamsterlandschaften zu Aufgabe 2

11.7.3 Aufgabe 3

Der Hamster befindet sich genau in der Mitte eines quadratischen geschlossenen, körnerlosen Raum ohne innere Mauern mit einer ungeraden Anzahl an freien Feldern pro Reihe (siehe Beispiele für Ausgangslandschaften in Abbildung 11.13). Er habe mindestens soviele Körner in seinem Maul wie freie Felder existieren. Seine Aufgabe besteht darin, mit möglichst wenigen Schritten (`vor()`;-Befehle) auf allen Feldern des Territoriums jeweils ein Korn abzulegen. Hinweis: Lassen Sie den Hamster zyklische Kreise laufen!

#	#	#	#	#	#	#
#						#
#						#
#			>			#
#						#
#						#
#	#	#	#	#	#	#

#	#	#	#	#	#	#	
#		-	-	-	-		#
#		-	-	-	-		#
#		-	-	-	-		#
#		-	-	-	-		#
#		-	-	-	-		#
#		-	-	-	-		#
#	#	#	#	#	#	#	#

Abbildung 11.13: Typische Hamsterlandschaft und Wegbeschreibung zu Aufgabe 3

11.7.4 Aufgabe 4

Denken Sie sich selbst weitere Hamster-Aufgaben aus, und versuchen Sie, diese zu lösen. Verwenden Sie dabei intensiv boolesche Funktionen. Viel Spaß!

Kapitel 12

Programmmentwurf

12.1 Lösen von Problemen

Wenn Sie die vorangegangenen Kapitel sorgfältig durchgearbeitet und insbesondere die Übungsaufgaben intensiv bearbeitet haben, sind Sie Ihrem Ziel, ein „guter“ Programmierer zu werden, schon ein ganzes Stück nähergekommen. Sie wissen nun, was Anweisungen sind, Sie können Prozeduren und boolesche Funktionen definieren und aufrufen, und Sie kennen als Kontrollstrukturen des Programmablaufs die Auswahl- und Wiederholungsanweisungen, mit denen Sie Algorithmen bzw. Programme flexibel formulieren können. In diesem Kapitel wird nun kein weiteres derartiges Sprachkonstrukt eingeführt, stattdessen werden Sie lernen, wie Sie die bisher erlernten Sprachkonstrukte einsetzen können, um ausgehend von einem gegebenen Problem ein Programm zu entwickeln, das dieses Problem korrekt und vollständig löst und dessen Lösungs-idee auch für andere verständlich ist.

Vielleicht hatten Sie beim Lösen von Übungsaufgaben insbesondere in den letzten beiden Kapiteln bestimmte Probleme; sie wußten eventuell nicht, wie Sie an die Übungsaufgabe herangehen, womit Sie anfangen sollten. Keine Angst, in diesem Kapitel werden Sie ein Verfahren kennenlernen, das Ihnen dabei hilft, derartige Übungsaufgaben systematisch zu lösen. Nichtsdestotrotz ist der Algorithmen- bzw. Programmmentwurf kein vollständig mechanisierbarer, sondern ein kreativer Prozeß, bei dem Sie Ihre Gehirnzellen aktivieren müssen. Er ist insbesondere selbst nicht algorithmisierbar und erfordert vom Programmierer Intuition, Erfahrung und Ideen.

Aber der Programmmentwurf ist vergleichbar mit vielen Problemen, mit denen Sie tagtäglich in Berührung kommen. Überlegen Sie einmal: Wie gehen Sie vor, wenn Sie ein Puzzle zusammensetzen. Ganz sicher nicht so, daß Sie einfach probieren, so lange Puzzlestücke zusammenzustecken, bis das Puzzle fertig ist. Vielmehr werden Sie zunächst die vier Eckstücke und anschließend die Randstücke suchen und damit zunächst den Rahmen zusammensetzen. Anschließend werden Sie vielleicht (falls es sich bei dem Puzzle um ein Landschaftsbild handelt) alle blauen Stücke herausuchen, die zum Himmel gehören könnten. Was Sie hier machen, nennt man *Komplexitätsreduktion*. Sie versuchen, das komplexe Gesamtproblem zunächst in weniger komplexe Teilprobleme zu zerlegen und die Teilprobleme zu lösen. Die Gesamtlösung ergibt sich dann schließlich durch die Zusammenführung der Teillösungen. Genau dieselbe Systematik liegt auch dem im folgenden beschriebenen Verfahren des Programmmentwurfs zugrunde. Man bezeichnet das Verfahren deshalb auch als *Schrittweise Verfeinerung* oder *Top-Down-Programmmentwurf*.

- Wo steht der Hamster?
 - In welche Richtung schaut der Hamster?
 - Wieviele Körner hat der Hamster im Maul?
 - Liegen irgendwo im Territorium Körner, wenn ja wieviele?
 - Befinden sich irgendwo im Territorium (noch weitere) Mauern?
 - Muß der Hamster direkt vor dem Berg stehen oder kann er sich auch einige Felder vor dem Berg befinden?
 - Wie hoch sind die einzelnen Stufen des Berges?
- bzgl. der Endsituation:
 - Wo soll der Hamster stehen?
 - In welche Richtung soll der Hamster schauen?
 - Wieviele Körner soll der Hamster im Maul haben?
 - Sollen irgendwo im Territorium Körner liegen, wenn ja wieviele?
 - bzgl. des Weges des Hamsters:
 - Darf der Hamster zwischendurch auch wieder zurückgehen?
 - Muß der Hamster immer mit einer Wand in Berührung sein (ansonsten besteht „Absturzgefahr“)?

Die ersten fünf Fragen bzgl. der Ausgangssituation und die ersten vier Fragen bzgl. der Endsituation sind übrigens typische Fragestellungen bei allen Hamsteraufgaben. Die fünf Größen Position des Hamsters, Richtung des Hamsters, Anzahl der Körner im Maul des Hamsters, Anzahl der Körner auf bestimmten Feldern des Territoriums und Mauern auf bestimmten Feldern des Territoriums beim Start eines Hamsterprogramms nennt man auch *Anfangs-* bzw. *Eingabegrößen*. Dementsprechend werden die drei Größen Position des Hamsters, Richtung des Hamsters, Anzahl der Körner im Maul des Hamsters und Anzahl der Körner auf bestimmten Feldern des Territoriums bei der Beendigung eines Hamsterprogramms als *End-* bzw. *Ausgabegrößen* bezeichnet.

Wie Sie schnell feststellen können, ist die obige Aufgabenstellung noch nicht exakt. Sie müssen also weitere Informationen einholen bzw. selbst bestimmte Größen festlegen.

Unter der Annahme, daß alle nicht genannten Anfangs- und Endgrößen beliebig sind und – außer den genannten – auch keinerlei Einschränkungen zum Lösungsweg (Weg des Hamsters) existieren, ist nun folgende Reformulierung der obigen Aufgabenstellung exakt:

Der Hamster steht mit Blickrichtung West vor einem regelmäßigen Berg unbekannter Höhe (ohne „Überhänge“!). Er muß dabei nicht unbedingt direkt vor dem Berg stehen. Die Stufen des Berges sind jeweils eine Mauer hoch. Der Hamster habe mindestens soviele Körner im Maul wie Stufen existieren. Auf dem Hamsterfeld befinden sich anfangs keine Körner. Außer den Mauern, die den Berg und den Weg zum Berg bilden, befinden sich keine weiteren Mauern im Territorium). Die Aufgabe des Hamsters besteht darin, den Gipfel des Berges zu erklimmen und auf dem Gipfel anzuhalten. Auf dem Weg zum Gipfel soll der Hamster auf jeder Stufe einschließlich dem Gipfel selbst – und nur dort! – genau ein Korn ablegen. Auf dem Weg zum Gipfel muß der Hamster

immer mit der Wand in Berührung bleiben, darf also nicht in eine Situation gelangen, in der „Absturzgefahr“ droht.

In Abbildung 12.1 sind also die Teile (a) und (b) sowohl am Anfang, zwischendurch und am Ende korrekt. Die Ausgangslandschaft (c) ist nicht korrekt (Hamster schaut nicht nach Westen; der Berg ist nicht regelmäßig). Der in Teil (d) skizzierte Snapshot von unterwegs ist nicht zulässig (es droht „Absturzgefahr“), ebenso ist in Teil (e) eine unzulässige Endsituation skizziert (auf einer Aufwärtsstufe des Berges liegt kein Korn, dafür aber auf einer Abwärtsstufe).

12.3 Entwurf

Wie bereits in Abschnitt 12.1 erwähnt, basiert der Entwurf von Algorithmen bzw. Programmen auf dem Prinzip der *Schrittweisen Verfeinerung*. Was genau ist aber unter diesem Prinzip zu verstehen? Schauen Sie sich unser Beispiel an. Eine intuitive Lösungsidee ist folgende: Der Hamster soll zunächst bis zum Berg laufen und dann den Berg erklimmen. An dem „und“ in diesem Satz erkennen Sie, daß das Problem in zwei Teilprobleme zerlegt werden kann:

- Der Hamster soll zum Berg laufen.
- Der Hamster soll den Berg erklimmen.

Damit haben wir das Problem „verfeinert“. Wir müssen nun diese beiden Teilprobleme lösen. Wenn wir anschließend die Lösungen der beiden Teilprobleme zusammensetzen, erhalten wir die Lösung des Gesamtproblems.

Übertragen wir dieses Prinzip nun auf die Programmierung. In Kapitel 8 haben Sie mit der Prozedurdefinition ein Sprachkonstrukt kennengelernt, mit dem Sie neue Befehle definieren können. Dieses Sprachkonstrukt werden wir nun verwenden, um Hamsterprogramme nach dem Prinzip der *Schrittweisen Verfeinerung* zu entwickeln. Definieren Sie dazu für jedes Teilproblem zunächst einfach eine Prozedur, ohne sich um die Implementierung zu kümmern. Das Hauptprogramm setzt sich dann einfach aus den Aufruf der Prozeduren zusammen. An dem Beispielprogramm sei dieses verdeutlicht:

```
// der Hamster soll zunaechst bis zum Berg laufen
// und dann den Berg erklimmen
void main()
{
    laufe_zum_berg();
    erklimme_den_berg();
}

// der Hamster soll zum Berg laufen
void laufe_zum_berg()
{
}

// der Hamster soll den Berg erklimmen
```

```
void erklimme_den_berg()
{
}
```

Wenden wir uns nun dem Lösen der beiden Teilprobleme zu. Beim ersten Teilproblem ist dies nicht besonders schwierig. Solange wie sich vor ihm keine Mauer befindet, soll der Hamster ein Feld nach vorne hüpfen, am Fuße des Berges legt er ein Korn ab:

```
// der Hamster soll zum Berg laufen
void laufe_zum_berg()
{
    while (vorn_frei())
    {
        vor();
    }
    gib(); // am Fusse des Berges legt der Hamster ein Korn ab
}
```

Die Lösung des zweiten Teilproblems ist noch nicht ganz so einfach zu ermitteln. Also nutzen wir das Prinzip der Verfeinerung erneut – daher der Begriff **Schrittweise Verfeinerung**. Zunächst stellen wir fest, daß die Höhe des Berges nicht festgelegt, also prinzipiell beliebig ist. Diese Feststellung deutet darauf hin, daß der Einsatz einer Wiederholungsanweisung notwendig ist. Innerhalb dieser Wiederholungsanweisung soll der Hamster jedes Mal eine Stufe erklimmen. Die Wiederholungsanweisung endet, sobald der Hamster den Gipfel erreicht hat. Wir notieren: „Der Hamster soll jeweils eine Stufe erklimmen, solange wie er den Gipfel noch nicht erreicht hat.“

Übertragen wir diese Lösungsidee nun in entsprechende Sprachkonstrukte der Programmierung. Für die Anweisung „Der Hamster soll eine Stufe erklimmen“ definieren wir eine weitere Prozedur, für die Bedingung „ist der Gipfel erreicht“ eine boolesche Funktion:

```
// der Hamster soll den Berg erklimmen
void erklimme_den_berg()
{
    do
    {
        erklimme_eine_stufe();
    } while (!gipfel_erreicht());
}

// der Hamster soll eine Stufe erklimmen
void erklimme_eine_stufe()
{
}

// hat der Hamster den Gipfel erreicht?
boolean gipfel_erreicht()
{
}
```

Wenden wir uns zunächst der Implementierung der Prozedur `erklimme_eine_stufe` zu. Diese ist nicht mehr besonders schwierig:

```
// der Hamster soll eine Stufe erklimmen
void erklimme_eine_stufe()
{
    links_um(); // nun schaut der Hamster nach oben
    vor();      // der Hamster erklimmt die Mauer
    rechts_um(); // der Hamster wendet sich wieder dem Berg zu
    vor();      // der Hamster begibt sich auf den naechsten Vorsprung
    gib();      // der Hamster legt ein Korn ab
}

// der Hamster dreht sich nach rechts um
void rechts_um()
{
    links_um(); links_um(); links_um();
}

```

Etwas schwieriger scheint die Implementierung der booleschen Funktion `gipfel_erreicht` zu sein. Wie kann der Hamster feststellen, ob er sich auf dem Gipfel befindet oder nicht? Überlegen Sie einfach: wann wird die Funktion aufgerufen? Sie wird jedesmal dann aufgerufen, nachdem der Hamster eine Stufe erklimmen hat. Steht der Hamster danach vor einer Mauer, so ist der Gipfel noch nicht erreicht. Befindet sich vor dem Hamster jedoch keine Mauer, dann steht er auf dem Gipfel. Also ist die Implementierung der booleschen Funktion doch ganz einfach:

```
// hat der Hamster den Gipfel erreicht?
boolean gipfel_erreicht()
{
    if (vorn_frei())
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

oder besser formuliert (siehe auch Kapitel 11.3.3):

```
// hat der Hamster den Gipfel erreicht?
boolean gipfel_erreicht()
{
    return vorn_frei();
}

```

Damit haben wir nun alle Prozeduren implementiert – sprich Teilprobleme gelöst – d.h. unser Programm ist fertig:

```
// der Hamster soll zunaechst bis zum Berg laufen
// und dann den Berg erklimmen
void main()
{
    laufe_zum_berg();
    erklimme_den_berg();
}

// der Hamster soll zum Berg laufen
void laufe_zum_berg()
{
    while (vorn_frei())
    {
        vor();
    }
    gib(); // am Fusse des Berges legt der Hamster ein Korn ab
}

// der Hamster soll den Berg erklimmen
void erklimme_den_berg()
{
    do
    {
        erklimme_eine_stufe();
    } while (!gipfel_erreicht());
}

// der Hamster soll eine Stufe erklimmen
void erklimme_eine_stufe()
{
    links_um(); // nun schaut der Hamster nach oben
    vor();      // der Hamster erklimmt die Mauer
    rechts_um(); // der Hamster wendet sich wieder dem Berg zu
    vor();      // der Hamster begibt sich auf den naechsten Vorsprung
    gib();      // der Hamster legt ein Korn ab
}

// der Hamster dreht sich nach rechts um
void rechts_um()
{
    links_um(); links_um(); links_um();
}

// hat der Hamster den Gipfel erreicht?
boolean gipfel_erreicht()
```

```

{
    return vorn_frei();
}

```

Sie sehen, der Entwurf eines Programms ist gar nicht so schwierig, wenn man sich strikt an das Prinzip der *Schrittweisen Verfeinerung* hält:

- Wenn das Gesamtproblem zu komplex ist, teilen Sie es in einfachere Teilprobleme auf.
- Lösen Sie die Teilprobleme:
 - Wenn ein Teilproblem zu komplex ist, teilen Sie es in (noch) einfachere Teilprobleme auf.
 - ...
 - Setzen Sie die Lösungen der Teilprobleme zu einer Lösung des (übergeordneten) Teilproblems zusammen.
- Setzen Sie die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems zusammen.

12.4 Implementierung

Ziel der Implementierungsphase ist es, den in der Entwurfsphase entwickelten Algorithmus in ein in einer Programmiersprache formuliertes Programm zu überführen und in den Rechner einzugeben. Das erste Teilziel haben wir bei den Hamsterprogrammen bereits in der Entwurfsphase erreicht. Die Möglichkeit, Prozeduren und boolesche Funktionen zu definieren und diese mit selbstgewählten Bezeichnern zu benennen, ermöglichen uns dies. Algorithmenentwurf und Programmformulierung können durch das Prinzip der *Schrittweisen Verfeinerung* wechselseitig durchgeführt werden: Zunächst wird umgangssprachlich der Lösungsalgorithmus zu einem (Teil-)Problem formuliert. Anschließend wird dieser direkt in die Syntax der Hamstersprache übertragen. Beispiel:

Problem: Der Hamster soll den Berg erklimmen.

Algorithmus: Erklimme solange eine Stufe wie der Gipfel noch nicht erreicht ist.

Implementierung:

```

// der Hamster soll den Berg erklimmen
void erklimme_den_berg()
{
    do
    {
        erklimme_eine_stufe();
    } while (!gipfel_erreicht());
}

// der Hamster soll eine Stufe erklimmen
void erklimme_eine_stufe()

```

```

{
}

// hat der Hamster den Gipfel erreicht?
boolean gipfel_erreicht()
{
}

```

Also reduziert sich die Arbeit der Implementierungsphase bei der Entwicklung von Hamsterprogrammen auf das Eingeben des entwickelten Programms in den Rechner und das anschließende Compilieren des Programmcodes.

An dieser Stelle sei auf eine ganz wichtige Eigenschaft der Programmentwicklung hingewiesen: Vermeiden Sie es, die Entwurfsphase direkt am Computer durchzuführen und den Programmcode direkt einzugeben. Nehmen Sie sich einen Stift zur Hand, und skizzieren Sie Ihre Lösungsideen zunächst auf Papier. Bei so einfachen Beispielen wie dem obigen scheint das überflüssig zu sein, aber bei komplexeren Problemen laufen Sie ansonsten in die Gefahr, sogenannten „Spaghetti-Code“ zu produzieren. Der Begriff „Spaghetti-Code“ wird dann verwendet, wenn der Programmcode nicht mehr sauber strukturiert und die Lösungsidee nur schwer nachvollziehbar ist; ein Phänomen, das häufig bei direkt in den Computer eingegebenem Programmcode festzustellen ist: Zunächst wird der Code eingegeben, dann werden Fehler entdeckt, es werden Programmteile geändert, gelöscht und hinzugefügt, und schließlich hat man ein schirr undurchdringliches Wirrwarr bzw. Chaos von Programmcode vor sich. Hüten Sie sich davor, Spaghetti-Code zu produzieren. Gute Programme sind nicht nur dadurch gekennzeichnet, daß sie korrekt ablaufen, sondern auch durch die Eigenschaft, daß andere Personen sie leicht verstehen und die Lösungsidee nachvollziehen können.

Achten Sie also im Programmcode auf Übersichtlichkeit und Strukturiertheit. Nutzen Sie die Möglichkeit, zur Lösung von Teilproblemen Prozeduren und Funktionen definieren zu können. Kommentieren Sie Prozeduren bzw. Funktionen sowie komplizierte Sachverhalte. Wählen Sie aussagekräftige Bezeichner.

Vergleichen Sie einmal das folgende Programm mit dem oben entwickelten. Auch dieses Programm löst die Problemstellung korrekt, ist aber wesentlich schwerer überschaubar, weniger verständlich und kaum nachvollziehbar:

```

void main() {
    while
        (vorn_frei()) vor(); while (!vorn_frei()) {
            links_um(); vor(); r(); vor(); }
} void r() { links_um(); links_um(); links_um(); }

```

12.5 Test

Ziel der Testphase ist die Überprüfung, ob das entwickelte Programm die Problemstellung korrekt und vollständig löst. Korrekt bedeutet, daß das Programm für zulässige Anfangsgrößen die erwarteten Endgrößen produziert, wobei eventuell vorhandene Bedingungen, die an den

Lösungsweg gestellt wurden, eingehalten werden. Vollständig bedeutet, daß das Programm für **alle** zulässigen Anfangsgrößen korrekt arbeitet. Dabei ist es im allgemeinen allerdings nicht möglich, Testläufe für alle möglichen Anfangsgrößen durchzuführen, in der Regel gibt es nämlich unendlich viele mögliche Anfangszustände. Daher besteht eine wesentliche Aufgabe der Testphase darin, sogenannte Testmengen zu bilden. Testmengen werden derart gebildet, daß die Menge der zulässigen Anfangsgrößen in (disjunkte) Teilmengen zerlegt wird, wobei alle Elemente einer Teilmenge jeweils gewisse Gemeinsamkeiten bzw. gleichartige Charakteristiken aufweisen. Aus jeder dieser Teilmengen wird ein Vertreter gewählt und in die Testmenge eingefügt. Hat man eine korrekte Zerlegung der Menge der zulässigen Anfangsgrößen gewählt, d.h. eine korrekte Testmenge gebildet, und arbeitet das Programm für alle Elemente der Testmenge korrekt, dann kann man rückschließen, daß das Programm vollständig korrekt ist. Allerdings ist die Art und Weise einer Teilmengenzerlegung der Menge der Anfangsgrößen aufgabenspezifisch und erfordert vom Programmierer Erfahrung und Intuition.

Leider wird die Testphase nicht nur von Programmieranfängern häufig vernachlässigt. Man läßt sein Programm ein- oder zweimal laufen, alles läuft glatt ab, das Programm scheint also korrekt zu sein. Hierbei handelt es sich um ein psychologisches Problem; niemand gesteht sich selbst gerne Fehler ein, also ist man auch gar nicht daran interessiert, Fehler zu finden. Daher der dringende Appell: Widmen Sie insbesondere der Testphase viel Aufmerksamkeit, und lassen Sie das Programm eventuell auch mal von anderen Personen testen.

In Kapitel 3.1.4 wurden einige Teststrategien vorgestellt, die im folgenden nun am obigen Beispiel demonstriert werden. Von besonderer Bedeutung ist dabei wie bereits erwähnt die Konstruktion von Testmengen, wobei insbesondere Grenzwerte zu berücksichtigen sind.

Notieren Sie sich beim Zusammenstellen der Testmengen nicht nur die Ausgangsgrößen, sondern jeweils auch das erwartete Ergebnis, und überprüfen Sie nach Beendigung des Programms, ob das tatsächliche Ergebnis mit dem erwarteten übereinstimmt.

Im Umfeld der Hamsterprogramme bestehen Testmengen immer aus möglichen Hamster-Territorien, auf denen das zu testende Hamsterprogramm gestartet werden könnte. Abbildung 12.2 enthält eine typische Testmenge für das obige Beispiel. Teil (a) der Abbildung zeigt eine „normale“ Hamsterlandschaft, Teil (b) skizziert den Grenzfall, daß der Hamster bereits anfangs direkt vor dem Berg steht und in Teil (c) ist der Berg nur eine einzige Stufe hoch.

Bei der Konstruktion von Testmengen für Ihre Hamsterprogramme ist es wichtig zu wissen, daß Sie natürlich nur Situationen berücksichtigen müssen, die auch mit der Aufgabenstellung konform sind. Wird Ihr Programm mit nicht-zulässigen Ausgangsgrößen gestartet, kommt es zwangsläufig zu Fehlern; aber das sind nicht Ihre Fehler, sondern die Fehler desjenigen, der das Programm gestartet hat.

Lassen Sie nun das Hamsterprogramm für alle Elemente der Testmenge laufen. Kontrollieren Sie dabei sowohl den Zustand der Landschaft bei Beendigung des Programmes (auf jeder Stufe einschließlich dem Gipfel – und nur dort – liegt genau ein Korn; der Hamster befindet sich auf dem Gipfel) als auch den Weg des Hamsters (der Hamster darf nicht in „Absturzgefahr“ kommen). Sie werden feststellen, daß das Beispielprogramm für alle Elemente der Testmenge korrekt abläuft und korrekte Ergebnisse liefert. Das Programm scheint also in der Tat die Problemstellung korrekt und – unter der Voraussetzung, daß die Testmenge korrekt gebildet wurde – vollständig zu lösen.

					#		
				#	#	#	
>			#	#	#	#	#
#	#	#	#	#	#	#	#

(a) zu Anfang

					>		
				o	#		
			o	#	#	#	
		o	#	#	#	#	#
#	#	#	#	#	#	#	#

(a) am Ende

					#		
		>	#	#	#		
#	#	#	#	#	#	#	#

(b) zu Anfang

					>		
				o	#		
		o	#	#	#		
#	#	#	#	#	#	#	#

(b) am Ende

	>		#				
#	#	#	#	#	#	#	#

(c) zu Anfang

					>		
		o	#				
#	#	#	#	#	#	#	#

(c) am Ende

Abbildung 12.2: Testmenge zu Beispiel 1

Die Konstruktion von Testmengen ist in der Regel keine triviale Aufgabe. Häufig werden bestimmte Grenzfälle einfach übersehen. Gehen Sie daher bei der Zusammenstellung der Testmengen immer sehr sorgfältig vor. Leider gibt es hierzu keine allgemeingültigen Gesetzmäßigkeiten.

Günstig ist es, wenn Sie die Testmenge nicht erst während der Testphase zusammenstellen, sondern bereits in der Analysephase. Dann können Sie bereits beim Entwurf kontrollieren, ob Ihr Algorithmus auch für Grenzwerte korrekt ist.

12.6 Dokumentation

Genauso wie die Testphase wird bei der Programmentwicklung leider auch die Dokumentation nicht nur von Programmieranfängern häufig vernachlässigt. Denken Sie bitte daran, daß es bei der Softwareentwicklung nicht nur darauf ankommt, ein Programm zu entwickeln; vielmehr sind alle Ergebnisse des gesamten Entwicklungsprozesses schriftlich festzuhalten, so daß sowohl das Ergebnis als auch der Entwicklungsprozeß auch für andere Personen leicht nachvollziehbar sind. Zur Dokumentation gehören insbesondere:

- eine exakte Problemstellung,
- eine verständliche Beschreibung des entwickelten Algorithmus,
- der Programmcode,

- die gewählte Testmenge mit Protokollen der durchgeführten Testläufe,
- eine Beschreibung von aufgetretenen Problemen,
- alternative Lösungsansätze.

12.7 Ein weiteres Beispiel

Um die Vorgehensweise der Programmentwicklung noch verständlicher zu machen, wird in diesem Abschnitt der gesamte Entwicklungsprozeß an einem weiteren Beispiel demonstriert.

12.7.1 Aufgabe

Folgende Aufgabe ist zu lösen (siehe auch Kapitel 10.4.1):

In einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern sind wahllos eine unbekannte Anzahl an Körnern verstreut. Der Hamster, der sich zu Anfang in der linken unteren Ecke des Hamster-Territoriums mit Blickrichtung Ost befindet, soll alle Körner aufsammeln und dann stehenbleiben.

12.7.2 Analyse

Die Aufgabenstellung wird präzisiert:

- bzgl. der Ausgangssituation:
 - Der Hamster steht in der unteren linken Ecke des Territoriums.
 - Der Hamster schaut nach Osten.
 - Die Anzahl an Körnern im Maul des Hamster ist nicht festgelegt.
 - Auf beliebigen Feldern in Territorium liegen beliebig viele Körner.
 - Das Territorium ist von einer rechteckigen geschlossenen Wand von Mauern umgeben; ansonsten befinden sich keine Mauern im Territorium; es existiert mindestens ein Feld im Territorium, auf dem keine Mauer steht.
- bzgl. der Endsituation:
 - Die Position des Hamsters ist nicht festgelegt.
 - Die Blickrichtung des Hamsters ist nicht festgelegt.
 - Die Anzahl an Körnern im Maul des Hamsters ist nicht festgelegt.
 - Auf keinem Feld im Territorium sollen mehr Körner liegen.
- bzgl. des Weges des Hamsters:
 - Es gibt keine Constraints bzgl. des Weges des Hamsters.

Abbildung 12.3 skizziert die gewählte Testmenge. Die Landschaft in Teil (a) hat eine ungerade die Landschaft in Teil (b) eine gerade Anzahl an Körnerreihen. Teil (c) skizziert den Grenzfall, daß das Territorium lediglich ein einziges nicht von Mauern besetztes Feld enthält.

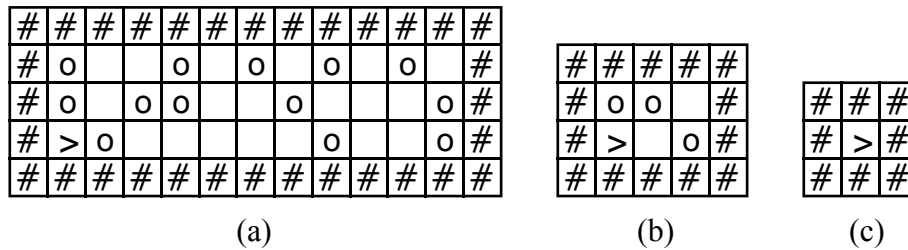


Abbildung 12.3: Testmenge zu Beispiel 2

12.7.3 Entwurf

Eine intuitive Lösungsidee hat folgende Gestalt: Der Hamster soll die Reihe „abgrasen“, in der er sich gerade befindet. Anschließend soll er testen, ob sich nördlich von ihm noch eine weitere Reihe befindet. Wenn ja, soll er sich in diese Reihe begeben und diese Reihe abgrasen. Dieser Vorgang soll solange wiederholt werden, bis der Hamster die nördliche Begrenzungswand erreicht.

In der Hamstersprache wird diese Lösungsidee folgendermaßen formuliert:

```
// der Hamster soll einzelne Koernerreihen abgrasen,
// solange wie noch weitere Reihen existieren
void main()
{
    ernte_eine_reihe();
    while (weitere_reihe_existiert())
    {
        begib_dich_in_naechste_reihe();
        ernte_eine_reihe();
    }
}

// der Hamster soll alle Koerner in einer Reihe einsammeln
void ernte_eine_reihe()
{
}

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben
void begib_dich_in_naechste_reihe()
{
}

// Ueberpruefung, ob in noerdlicher Richtung eine weitere nicht
// mit Mauern besetzte Reihe existiert
boolean weitere_reihe_existiert()
{
}
```

Während die Implementierung der Prozedur `ernte_eine_reihe` nicht besonders schwierig ist, stoßen wir bei der Implementierung der Prozedur `begib_dich_in_naechste_reihe` und der booleschen Funktion `weitere_reihe_existiert` auf ein Problem. Es macht nämlich einen Unterschied, ob der Hamster eine Reihe von links oder von rechts abgrast. Mit den bisher kennengelernten Sprachkonstrukten können wir diesen Konflikt nicht lösen. Das war auch der Grund, warum die Aufgabe in Kapitel 10.4.1 relativ umständlich derart gelöst worden ist, daß der Hamster nach dem Abgrasen einer Reihe zunächst die Reihe wieder zurücklaufen mußte.

Wir können zwar unsere erste Lösungsidee beibehalten, müssen jedoch bei der Übertragung der Idee in die Hamstersprache sorgfältiger die verschiedenen Richtungsalternativen unterscheiden:

```
// der Hamster soll einzelne Koernerreihen abgrasen,
// solange wie noch weitere Reihen existieren; er unterscheidet
// dabei, ob er die Reihen nach Osten oder nach Westen hin abgrast
void main()
{
    ernte_eine_reihe_nach_osten();
    while (weitere_reihe_links_vom_hamster_existiert())
    {
        begib_dich_links_um_in_naechste_reihe();
        ernte_eine_reihe_nach_westen();
        if (weitere_reihe_rechts_vom_hamster_existiert())
        {
            begib_dich_rechts_um_in_naechste_reihe();
            ernte_eine_reihe_nach_osten();
        }
    }
}

// der Hamster soll alle Koerner in einer Reihe einsammeln;
// er laeuft dabei von Westen nach Osten
void ernte_eine_reihe_nach_osten()
{
}

// der Hamster soll alle Koerner in einer Reihe einsammeln;
// er laeuft dabei von Osten nach Westen
void ernte_eine_reihe_nach_westen()
{
}

// Ueberpruefung, ob in noerdlicher Richtung (vom Hamster aus
// gesehen links) eine weitere nicht mit Mauern besetzte
// Reihe existiert
boolean weitere_reihe_links_vom_hamster_existiert()
{
}
```

```

// Ueberpruefung, ob in noerdlicher Richtung (vom Hamster aus
// gesehen rechts) eine weitere nicht mit Mauern besetzte
// Reihe existiert
boolean weitere_reihe_rechts_vom_hamster_existiert()
{
}

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben; vom Hamster aus gesehen, liegt diese Reihe
// links von ihm
void begib_dich_links_um_in_naechste_reihe()
{
}

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben; vom Hamster aus gesehen, liegt diese Reihe
// rechts von ihm
void begib_dich_rechts_um_in_naechste_reihe()
{
}

```

Bei der Implementierung der beiden Prozeduren `ernte_eine_reihe_nach_osten` und `ernte_eine_reihe_nach_westen` stellt sich schnell heraus, daß beide dieselbe Gestalt haben. Also wird eine Prozedur `ernte_eine_reihe` definiert und durch die beiden Prozeduren aufgerufen. Die Implementierung der Prozedur `ernte_eine_reihe` ist dabei relativ problemlos:

```

// der Hamster soll alle Koerner in einer Reihe einsammeln;
// er laeuft dabei von Westen nach Osten
void ernte_eine_reihe_nach_osten()
{
    ernte_eine_reihe();
}

// der Hamster soll alle Koerner in einer Reihe einsammeln;
// er laeuft dabei von Osten nach Westen
void ernte_eine_reihe_nach_westen()
{
    ernte_eine_reihe();
}

// der Hamster soll alle Koerner in einer Reihe einsammeln
void ernte_eine_reihe()
{
    sammle();
    while (vorn_frei())
    {
        vor();
    }
}

```

```

        sammle();
    }
}

// der Hamster sammelt alle Koerner eines Feldes ein
void sammle()
{
    while (korn_da())
    {
        nimm();
    }
}

```

Auch die Implementierung der beiden booleschen Funktionen ist geradlinienförmig. Es muß einfach nur getestet werden, ob sich links bzw. rechts vom Hamster eine Mauer befindet:

```

// Ueberpruefung, ob in noerdlicher Richtung (vom Hamster aus
// gesehen links) eine weitere nicht mit Mauern besetzte
// Reihe existiert
boolean weitere_reihe_links_vom_hamster_existiert()
{
    return !links_frei();
}

// Ueberpruefung, ob in noerdlicher Richtung (vom Hamster aus
// gesehen rechts) eine weitere nicht mit Mauern besetzte
// Reihe existiert
boolean weitere_reihe_rechts_vom_hamster_existiert()
{
    return !rechts_frei();
}

// Ueberpruefung, ob sich links vom Hamster eine Mauer befindet
boolean links_frei()
{
    links_um();
    if (vorn_frei())
    {
        rechts_um();
        return true;
    }
    else
    {
        rechts_um();
        return false;
    }
}

```

```

}

// Ueberpruefung, ob sich rechts vom Hamster eine Mauer befindet
boolean rechts_frei()
{
    rechts_um();
    if (vorn_frei())
    {
        links_um();
        return true;
    }
    else
    {
        links_um();
        return false;
    }
}

// drehe dich um 90 Grad nach rechts
void rechts_um()
{
    kehrt(); links_um();
}

// drehe dich um 180 Grad
void kehrt()
{
    links_um(); links_um();
}

```

Die beiden Prozeduren zum Wechseln der Reihe werden nur aufgerufen, wenn die Reihe in nördlicher Richtung auch frei ist. Sie sind also besonders einfach zu implementieren:

```

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben; vom Hamster aus gesehen, liegt diese Reihe
// links von ihm
void begib_dich_links_um_in_naechste_reihe()
{
    links_um();
    vor();
    links_um();
}

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben; vom Hamster aus gesehen, liegt diese Reihe
// rechts von ihm
void begib_dich_rechts_um_in_naechste_reihe()

```

```

{
  rechts_um();
  vor();
  rechts_um();
}

```

Damit sind alle Prozeduren bzw. booleschen Funktionen implementiert, das Hamsterprogramm ist fertiggestellt.

12.7.4 Implementierung

In der Implementierungsphase wird der Programmcode nun mit einem Editor in den Rechner eingegeben und auf syntaktische Fehler untersucht.

12.7.5 Test

In der Testphase wird das Programm auf allen Landschaften der Testmenge (siehe Abbildung 12.3 gestartet. In den Fällen (a) und (c) liefert es auch die erwarteten korrekten Ergebnisse. In Fall (b) gerät das Programm jedoch in eine Endlosschleife: Die beiden Reihen werden ununterbrochen zyklisch vom Hamster durchlaufen. Also ist das Programm nicht korrekt!

Es muß nun untersucht werden, woran der Fehler liegt: Der Hamster erntet die untere Reihe ab und testet die Bedingung der while-Schleife. Die boolesche Funktion liefert den Wert `true`, d.h. die while-Schleife wird betreten. Also begibt sich der Hamster in die obere Reihe und erntet. Anschließend wird die Bedingung der if-Anweisung überprüft. Die boolesche Funktion liefert den Wert `false`, weil sich rechts vom Hamster eine Mauer befindet. Da die if-Anweisungen keinen else-Teil enthält, wird als nächstes wieder die Bedingung der while-Schleife überprüft. Eigentlich müßte diese nun den Wert `false` liefern, weil ja alle Reihen abgegrast sind. Tut sie aber leider nicht. Der Grund hierfür ist der, daß als Vorbedingung für die boolesche Funktion `weitere_reihe_links_vom_hamster_existiert`, die in der Bedingung der while-Schleife aufgerufen, angenommen wird, daß der Hamster gerade in Blickrichtung Osten schaut. Diese Vorbedingung ist aber nicht erfüllt; denn ein Richtungswechsel wird nur im true-Teil der if-Anweisung, nicht aber im false-Teil der if-Anweisung vorgenommen. Dies haben wir bei der obigen Lösung schlicht vergessen. Wir müssen das Hauptprogramm dementsprechend korrigieren:

```

// der Hamster soll einzelne Koernerreihen abgrasen,
// solange wie noch weitere Reihen existieren; er unterscheidet
// dabei, ob er die Reihen von Osten oder von Westen aus abgrast
void main()
{
  ernte_eine_reihe_nach_osten();
  while (weitere_reihe_links_vom_hamster_existiert())
  {
    begib_dich_links_um_in_naechste_reihe();
    ernte_eine_reihe_nach_westen();
    if (weitere_reihe_rechts_vom_hamster_existiert())

```

```

{
  begib_dich_rechts_um_in_naechste_reihe();
  ernte_eine_reihe_nach_osten();
}
else
{
  kehrt();
}
}
}

```

Nach der Compilierung muß nun das Programm erneut mit allen (!) Hamsterlandschaften der Testmenge getestet werden. Dieses Mal liefert es tatsächlich in allen Fällen die erwarteten Ergebnisse (siehe auch die Ausschnitte des Testlaufprotokolls in Abbildung 12.4).

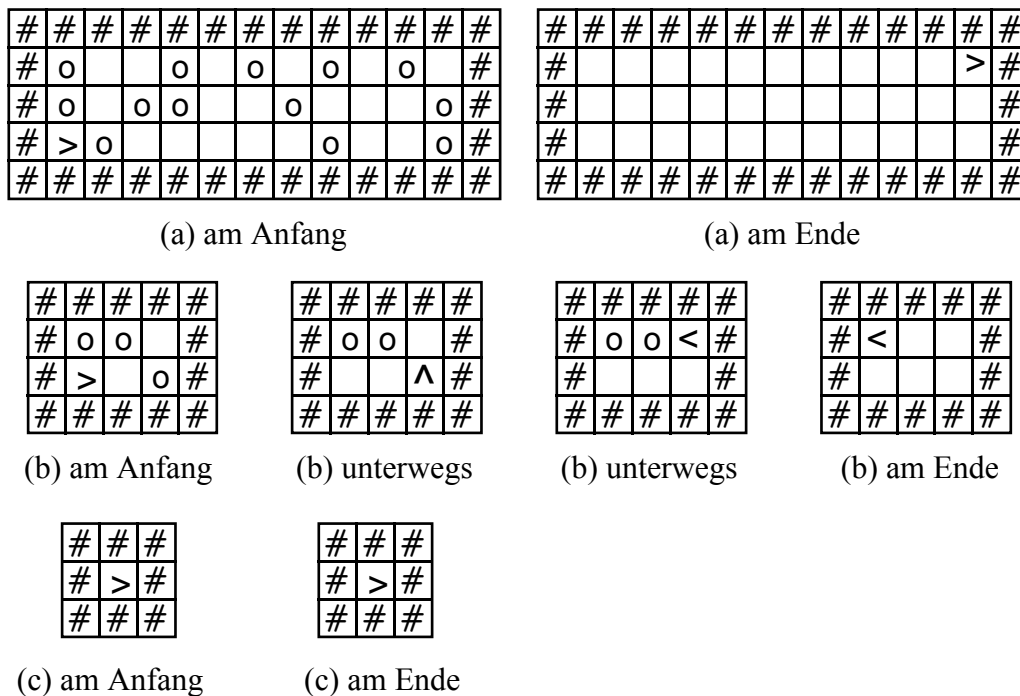


Abbildung 12.4: Protokoll der Testläufe zu Beispiel 2

12.7.6 Dokumentation

Der Vollständigkeit halber werden im folgenden nochmal alle Bestandteile der Dokumentation zusammengestellt.

12.7.6.1 Problemstellung

In einem rechteckigen geschlossenen Raum unbekannter Größe ohne innere Mauern sind wahllos eine unbekannte Anzahl an Körnern verstreut. Der Hamster, der sich zu Anfang in der linken

unteren Ecke des Hamster-Territoriums mit Blickrichtung Ost befindet, soll alle Körner aufsammeln und dann stehenbleiben.

Die präzisierte Problemstellung lautet:

- bzgl. der Ausgangssituation:
 - Der Hamster steht in der unteren linken Ecke des Territoriums.
 - Der Hamster schaut nach Osten.
 - Die Anzahl an Körnern im Maul des Hamster ist nicht festgelegt.
 - Auf beliebigen Feldern in Territorium liegen beliebig viele Körner.
 - Das Territorium ist von einer rechteckigen geschlossenen Wand von Mauern umgeben; ansonsten befinden sich keine Mauern im Territorium; es existiert mindestens ein Feld im Territorium, auf dem keine Mauer steht.
- bzgl. der Endsituation:
 - Die Position des Hamsters ist nicht festgelegt.
 - Die Blickrichtung des Hamsters ist nicht festgelegt.
 - Die Anzahl an Körnern im Maul des Hamsters ist nicht festgelegt.
 - Auf keinem Feld im Territorium sollen mehr Körner liegen (d.h. der Hamster soll alle Körner, die anfangs im Territorium liegen, einsammeln)
- bzgl. des Weges des Hamsters:
 - Es gibt keine Constraints bzgl. des Weges des Hamsters.

12.7.6.2 Lösungsidee

Der Hamster soll die Reihe „abgrasen“, in der er sich gerade befindet. Anschließend soll er testen, ob sich nördlich von ihm noch eine weitere Reihe befindet. Wenn ja, soll er sich in diese Reihe begeben und diese Reihe abgrasen. Dieser Vorgang soll solange wiederholt werden, bis der Hamster die nördliche Begrenzungswand erreicht.

12.7.6.3 Programmcode

```
// der Hamster soll einzelne Koernerreihen abgrasen,
// solange wie noch weitere Reihen existieren; er unterscheidet
// dabei, ob er die Reihen von Osten oder von Westen aus abgrast
void main()
{
  ernte_eine_reihe_nach_osten();
  while (weitere_reihe_links_vom_hamster_existiert())
  {
    begib_dich_links_um_in_naechste_reihe();
    ernte_eine_reihe_nach_westen();
  }
}
```

```
    if (weitere_reihe_rechts_vom_hamster_existiert())
    {
        begib_dich_rechts_um_in_naechste_reihe();
        ernte_eine_reihe_nach_osten();
    }
    else
    {
        kehrt();
    }
}
}

// der Hamster soll alle Koerner in einer Reihe einsammeln;
// er laeuft dabei von Westen nach Osten
void ernte_eine_reihe_nach_osten()
{
    ernte_eine_reihe();
}

// der Hamster soll alle Koerner in einer Reihe einsammeln;
// er laeuft dabei von Osten nach Westen
void ernte_eine_reihe_nach_westen()
{
    ernte_eine_reihe();
}

// der Hamster soll alle Koerner in einer Reihe einsammeln
void ernte_eine_reihe()
{
    sammle();
    while (vorn_frei())
    {
        vor();
        sammle();
    }
}

// der Hamster sammelt alle Koerner eines Feldes ein
void sammle()
{
    while (korn_da())
    {
        nimm();
    }
}

// Ueberpruefung, ob in noerdlicher Richtung (vom Hamster aus
```

```
// gesehen links) eine weitere nicht mit Mauern besetzte
// Reihe existiert
boolean weitere_reihe_links_vom_hamster_existiert()
{
    return !links_frei();
}

// Ueberpruefung, ob in noerdlicher Richtung (vom Hamster aus
// gesehen rechts) eine weitere nicht mit Mauern besetzte
// Reihe existiert
boolean weitere_reihe_rechts_vom_hamster_existiert()
{
    return !rechts_frei();
}

// Ueberpruefung, ob sich links vom Hamster eine Mauer befindet
boolean links_frei()
{
    links_um();
    if (vorn_frei())
    {
        rechts_um();
        return true;
    }
    else
    {
        rechts_um();
        return false;
    }
}

// Ueberpruefung, ob sich rechts vom Hamster eine Mauer befindet
boolean rechts_frei()
{
    rechts_um();
    if (vorn_frei())
    {
        links_um();
        return true;
    }
    else
    {
        links_um();
        return false;
    }
}
```

```
// drehe dich um 90 Grad nach rechts
void rechts_um()
{
    kehrt(); links_um();
}

// drehe dich um 180 Grad
void kehrt()
{
    links_um(); links_um();
}

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben; vom Hamster aus gesehen, liegt diese Reihe
// links von ihm
void begib_dich_links_um_in_naechste_reihe()
{
    links_um();
    vor();
    links_um();
}

// der Hamster soll sich in die naechste Reihe in noerdlicher
// Richtung begeben; vom Hamster aus gesehen, liegt diese Reihe
// rechts von ihm
void begib_dich_rechts_um_in_naechste_reihe()
{
    rechts_um();
    vor();
    rechts_um();
}
```

12.7.6.4 Testmenge mit Protokollen der Testläufe

Die Testmenge sowie Ausschnitte aus den Ergebnisse der Testläufe des Programms finden sich in Abbildung 12.4.

12.7.6.5 Aufgetretene Probleme

Bei der Lösung der Hamsteraufgabe sind keine nennenswerten Probleme aufgetreten.

12.7.6.6 Alternative Lösungsansätze

Eine alternative Lösungsidee ist die, daß der Hamster nach dem Abgrasen einer Reihe zunächst jeweils wieder die Reihe zurückläuft, bevor er sich in die nächste Reihe begibt (siehe auch

Kapitel 10.4.1). Bei dieser Lösungsidee ist das Hauptprogramm ein wenig verständlicher, weil nicht zwischen den beiden Fällen „grase eine Reihe nach Osten hin ab“ und „grase eine Reihe nach Westen hin ab“ unterschieden werden muß. Die Lösung ist aber nicht besonders effizient.

12.8 Übungsaufgaben

Nun sind wieder Sie gefordert; denn in diesem Abschnitt werden Ihnen einige Hamster-Aufgaben gestellt, die sie selbständig zu lösen haben. Entwickeln Sie Ihre Programme dabei nach dem in diesem Kapitel erläuterten Verfahren, und halten Sie die Ergebnisse den Dokumentationsrichtungen entsprechend fest.

12.8.1 Aufgabe 1

Der Hamster steht in einem durch Mauern abgeschlossenen Raum unbekannter Größe. Er hat den Anfang einer Körnerspur entdeckt, die sich durch sein Territorium zieht. Die Körnerspur kreuzt sich nirgends und zwischen zwei parallelen Reihen, in denen die Spur verläuft, ist immer eine Reihe frei (siehe auch die Landschaft in Abbildung 12.5). Außer den Körnern der Spur befinden sich keine weiteren Körner im Territorium. Die Aufgabe des Hamsters besteht darin, alle Körner einzusammeln und am Ausgangspunkt der Spur (dort wo der Hamster anfangs steht) abzulegen.

#	#	#	#	#	#	#	#	#	#	#	#	#	
#	o	o	o	o	o	o	o	o				#	
#	o				#			o		#		#	
#	o	o	o	o		o	o	o	#	#	#	#	
#		#	#	o		o				o	#	#	
#			>	o	o	#	o		o	o	o	#	#
#						o	o	o					#
#	#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 12.5: Typische Ausgangssituation zu Aufgabe 1

12.8.2 Aufgabe 2

In Aufgabe 2 hat der Hamster eine ähnliche Aufgabe zu lösen wie in Aufgabe 1. Nur die Anfangsbedingungen sind etwas erschwert. Wiederum steht der Hamster in einem durch Mauern abgeschlossenen Raum unbekannter Größe. Er hat eine Körnerspur entdeckt (nicht unbedingt den Anfang!), die sich durch sein Territorium zieht. Im Gegensatz zu Aufgabe 1 kann die Spur jedoch verzweigen. Es gibt allerdings keine „Rundwege“. Die Voraussetzung, daß zwischen zwei Reihen der Körnerspur immer eine Reihe frei ist, hat auch in Aufgabe 2 Bestand; ebenfalls die Annahme, daß sich außer den Körnern der Spur keine weiteren Körner im Territorium befinden. Der Hamster soll alle Körner fressen. Er muß aber anschließend nicht unbedingt zum Ausgangspunkt zurücklaufen. Abbildung 12.6 skizziert eine typische Ausgangssituation.

#	#	#	#	#	#	#	#	#	#	#	#	#
#	o	o	>	o	o	o	o	o	o	o		#
#	o				#	o		o		#		#
#	o	o	o	o	o		o	o	#	#	#	#
#		#	#	o		o	o			o	#	#
#			o	o	#	o		o	o	o	#	#
#	o	o	o		o	o	o	o		o		#
#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 12.6: Typische Ausgangssituation zu Aufgabe 2

12.8.3 Aufgabe 3

Der Hamster steht am Anfang eines Labyrinths (siehe Abbildung 12.7). Dieses besteht aus Gängen, die jeweils so breit sind, wie eine Mauer. Die Gänge können verzweigen, es gibt jedoch keine Rundgänge. Der Hamster „riecht“, daß sich irgendwo im Labyrinth ein Korn befindet. Da er Hunger hat, versucht er natürlich, daß Korn zu finden und zu fressen.

#	#	#	#	#	#	#	#	#	#	#	#	#
#					#							#
#		#	#			#	#		#	#	#	#
#			#	#		#					o	#
#	#				#		#	#		#	#	#
#	#	#	#		#		#	#		#	#	#
#	>				#							#
#	#	#	#	#	#	#	#	#	#	#	#	#

Abbildung 12.7: Typische Ausgangssituation zu Aufgabe 3

12.8.4 Aufgabe 3

Denken Sie sich selbst weitere Hamster-Aufgaben aus, und versuchen Sie, diese zu lösen. Viel Spaß!

Kapitel 13

Variablen und Ausdrücke

Kapitel 14

Prozeduren und Funktionen

Kapitel 15

Funktionsparameter

Kapitel 16

Rekursion

Literaturverzeichnis

- [AG96] K. Arnold and J. Gosling. *Die Programmiersprache Java*. Addison-Wesley, 1996.
- [KL83] L. Kligen and J. Liedtke. *Programmieren mit ELAN*. Teubner Verlag, 1983.
- [KL85] L. Kligen and J. Liedtke. *ELAN in 100 Beispielen*. Teubner Verlag, 1985.
- [Men85] K. Menzel. *LOGO in 100 Beispielen*. Teubner Verlag, 1985.
- [Opp83] L. Oppor. Das Hamster-Modell. Interner Bericht, GDM St. Augustin, 1983.
- [PRS95] R. Pattis, J. Robert, and M. Stehlek. *Karel the Robot: A gentle Introduction to the Art of Programming*. Wiley, 1995.
- [Ros83] P. Ross. *LOGO programming*. Addison-Wesley, 1983.