Informatik 1

Volker Aurich

Heinrich-Heine-Universität Düsseldorf

WS 2012/2013

Inhaltsverzeichnis

Kapitel 0

Einführung

An Hand einiger Beispiele sollen typische Fragestellungen der Informatik im Zusammenhang mit dem Begriff Algorithmus skizziert werden. Es sollen lediglich einige Ideen vermittelt werden; die Darstellung ist nicht immer formal präzise. Eine Präzisierung erfolgt später.

Wir werden die Sprechweisen der naiven Mengenlehre verwenden. Die Bildung einer Menge ist ein Abstraktionsprozeß: Man faßt einige Objekte unseres Denkens zu einem neuen Objekt zusammen, das man eine Menge nennt. Die zusammengefaßten Objekte heißen die Elemente der Menge.

Beispiele:

- Definition durch explizite Angabe der Elemente: $\{1, 2, 3, 4, 5\}$
- $\emptyset$ ist die leere Menge, die kein Element besitzt.
- Definition durch Angabe einer Eigenschaft, welche die Elemente der Menge und nur diese haben; alle anderen Eigenschaften der Elemente, in denen sie sich durchaus unterscheiden können, werden weggelassen: $\{x : x$ ist eine Primzahl $\}$.

Eine schlampige, aber oft suggestive Art, Mengen anzugeben, ist folgende: Man gibt einige Elemente an, aus denen man ein Bildungsgesetz für sämtliche Elemente der Menge erraten kann, z.B. $\{1, \ldots, n\}$. Aber Vorsicht! Das ist mathematisch gesehen Unfug. So könnte z.B. $\{2, 4, \ldots\}$ die Menge der geraden Zahlen oder die Menge der Zweierpotenzen sein.

$x \in M$ bedeute, daß x Element der Menge M ist; $x \notin M$ bedeute, daß $x \in M$ nicht gilt. Zwei Mengen M und N heißen *gleich* (in Zeichen: $M = N$), wenn sie dieselben Elemente haben (Extensionalitätsaxiom). Es kommt also nicht auf die Art ihrer Beschreibung an. Es ist z.B. $\{2, 3, 5, 7\} = \{3, 7, 2, 5, 2\} = \{x : x \leq 10$ und x ist Primzahl$\}$.

Sind $M_1, \ldots, M_n$ Mengen, so seien $M_1 \cup \ldots \cup M_n := \{x : x \in M_1$ oder $x \in M_2$ oder $\ldots$ oder $x \in M_n\} = \{x :$ es gibt mindestens ein j mit $x \in M_j\}$ und $M_1 \cap \ldots \cap M_n := \{x : x \in M_1$ und $x \in M_2$ und $\ldots$ und $x \in M_n\} = \{x :$ für jedes j gilt $x \in M_j\}$. Dabei ist unter der logischen Verknüpfung *oder* das nicht ausschließende *oder* zu verstehen, also nicht *entweder-oder*.

Eine *Abbildung* $f\colon A \to B$ einer Menge A in eine Menge B ist eine Vorschrift, die jedem Element $a \in A$ genau ein Element $b \in B$ zuordnet; dieses b wird mit $f(a)$ bezeichnet. Umgekehrt bezeichnet $f^{-1}(b)$ für jedes $b \in B$ die Menge derjenigen $a \in A$, für die $f(a) = b$ ist. $f^{-1}(b)$ kann leer sein oder auch aus mehr als einem Element bestehen. Ist $f^{-1}(b)$ für kein $b \in B$ leer, so nennt man die Abbildung *surjektiv*. Hat $f^{-1}(b)$ für kein $b \in B$ mehr als ein Element, so nennt man die Abbildung *injektiv*. Eine Abbildung, die zugleich surjektiv und injektiv ist, heißt *bijektiv*, oder mit anderen Worten: wenn es zu jedem $b \in B$ genau ein $a \in A$ mit $f(a) = b$ gibt, so ist f bijektiv. Eine bijektive Abbildung $f\colon A \to A$ heißt eine *Permutation* von A. Der Begriff *Funktion* wird oft synonym zu Abbildung verwendet, manchmal aber auch speziell für Abbildungen mit Werten in $\mathbb{R}$ reserviert.

0.1 Vereinbarungen

$\mathbb{N}$ sei die Menge der natürlichen Zahlen, also $\mathbb{N} = \{1, 2, 3, \ldots\}$.
$\mathbb{N}_0$ entstehe aus $\mathbb{N}$ durch Hinzunahme der Null, also $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$.
$\mathbb{Z}$ sei die Menge der ganzen Zahlen, also $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$.
$\mathbb{Q}$ sei die Menge der rationalen Zahlen; das sind diejenigen Zahlen, die sich als Bruch $\frac{m}{n}$ darstellen lassen, wobei m und n ganze Zahlen und $n \neq 0$ sind.
$\mathbb{R}$ sei die Menge der reellen Zahlen; das sind diejenigen Zahlen, die sich als (endlicher oder unendlicher) Dezimalbruch darstellen lassen.

Eine präzise Definition erhalten Sie in den Mathematikvorlesungen.

0.2 Vollständige Induktion

Mit der üblichen Ordnungsrelation ist $\mathbb{N}$ wohlgeordnet, d.h. jede nichtleere Teilmenge von $\mathbb{N}$ hat ein kleinstes Element.

0.2.1 Prinzip der vollständigen Induktion Für jedes $n \in \mathbb{N}$ sei $P(n)$ eine Aussage. Um zu zeigen, dass alle Aussagen $P(n)$ richtig sind, genügt es, Folgendes zu zeigen:
Induktionsanfang: $P(1)$ richtig ist.
Induktionsschluss: Für jedes $n \in \mathbb{N}$ ist die Aussage $P(n+1)$ richtig, sofern $P(n)$ richtig ist.

Beweis. Sei X die Menge der natürlichen Zahlen j, für die $P(j)$ falsch ist.
Annahme: X ist nicht leer. Dann hat X ein kleinstes Element x. Es kann nicht gleich 1 sein, weil $P(1)$ nach Voraussetzung richtig ist. Also ist $x > 1$. Es hat folglich einen Vorgänger in $\mathbb{N}$, nämlich $n = x - 1$, und diese Zahl n liegt nicht in X, weil x das kleinste Element von X ist und $n < x$ gilt. Somit ist $P(n)$ richtig, und nach dem Induktionsschluss ist auch $P(n+1) = P(x)$ richtig. Dies ist ein Widerspruch zur Definition von x. Folglich ist die Annahme, dass X nicht leer ist, falsch. q.e.d.

Bemerkungen: Man kann dieses Beweisprinzip auf diverse Weisen modifizieren. Z.B. kann man den Induktionsanfang statt für 1 für ein $m > 1$ durchführen; dann kann man natürlich auch nur auf die Richtigkeit der $P(n)$ mit $n \geq m$ schließen. Oder man

kann im Induktionsschluss die Induktionsvoraussetzung, dass $P(n)$ richtig ist, durch die umfassendere ersetzen, dass $P(j)$ für alle $j \leq n$ richtig ist.

Als Folgerung erhält man die Methode der rekursiven Definition.

0.2.2 Prinzip der rekursiven Definition Definiere $a_1 \in \mathbb{R}$ und gib an, wie man $a_{n+1} \in \mathbb{R}$ berechnet, wenn man a_n kennt. Dann sind die Zahlen a_j für alle $j \in \mathbb{N}$ definiert.

Beweis. Setze $P(j)$ gleich der Aussage: a_j *ist definiert.*

0.2.3 Beispiele

0.2.3.1 Fakultätsfunktion Definiere $f(1) := 1$ und $f(n+1) := (n+1)f(n)$ für jedes $n \in \mathbb{N}$. Dann ist $f(n)$ für alle $n \in \mathbb{N}$ definiert. Wenn man $f(n)$ für kleine n rekursiv ausrechnet, wird man schnell vermuten, dass f die Fakultätsfunktion ist, dass also $f(n) = n! := \prod_{j=1}^{j=n} j$ ist. Mit Hilfe vollständiger Induktion lässt sich dies leicht beweisen. $P(n)$ sei die Aussage: $f(n) = n!$. Es gilt $f(1) = 1 = 1!$; das ist der Induktionsanfang. Und weiter gilt $f(n+1) = (n+1)f(n) = (n+1)\prod_{j=1}^{j=n} j = \prod_{j=1}^{j=n+1} j = (n+1)!$; das ist der Induktionsschluss. q.e.d.

0.2.3.2 Fibonacci-Zahlen Die Folge der Fibonaccizahlen ist definiert durch $F_0 = 0, F_1 = 1$ und die Rekursionsgleichung $F_n = F_{n-1} + F_{n-2}$ für jedes $n \in \mathbb{N}$ mit $n \geq 2$. Diese einfache Rekursionsgleichung tritt immer wieder mal in Anwendungen auf. Es gibt eine explizite, analytische Formel, die den Wert von F_n in Abhängigkeit angibt. Sie ist aber nicht so leicht herzuleiten wie die im ersten Beispiel.

0.3 Das ggT-Problem

Um grundlegende Begriffe im Umfeld des Algorithmusbegriffs zu motivieren, betrachten wir ein einfaches zahlentheoretisches Problem, das z.B. bei kryptografischen Verfahren vorkommt. Es geht uns dabei nicht um die Konstruktion eines möglichst effizienten Lösungsverfahrens, sondern um die Illustration wichtiger Grundbegriffe.

Zur Erinnerung: $x \in \mathbb{Z}$ heißt ein Teiler von $a \in \mathbb{Z}$ (kurz: $x|a$ oder x *teilt* a), wenn es ein $y \in \mathbb{Z}$ gibt, so daß $x \cdot y = a$ gilt. Ist x ein Teiler von a und von b, so heißt x ein gemeinsamer Teiler von a und b. Und x heißt ein größter gemeinsamer Teiler (kurz: ggT) von a und b, wenn x ein gemeinsamer Teiler von a und b ist und wenn $x \geq z$ für jeden gemeinsamen Teiler z von a und b gilt.

Problem:
Bestimme für je zwei natürliche Zahlen a und b ihren größten gemeinsamen Teiler.

Die Untersuchung des Problems gliedert sich in zwei Teile:
1. Existenz einer Lösung: Gibt es stets einen ggT?
2. Berechnung des ggT.

Zu 1: Zwei Zahlen $a, b \in \mathbb{Z}$, die nicht beide 0 sind, besitzen stets einen ggT; denn einerseits gibt es stets einen gemeinsamen Teiler, nämlich 1, andererseits gibt es höchstens endlich viele gemeinsame Teiler x, weil ja $|x| \leq |a|$, wenn $a \neq 0$, oder $|x| \leq |b|$, wenn $b \neq 0$. Unter diesen endlich vielen gemeinsamen Teilern gibt es genau eine größte Zahl; sie ist der ggT.

Zu 2: Damit haben wir auch gleich ein Verfahren zur Berechnung des ggT zweier natürlicher Zahlen a und b: Überprüfe der aufsteigenden Reihe nach die Zahlen $x \in \{1, \ldots, \min\{a, b\}\}$, ob sie sowohl a als auch b teilen. Die größte davon ist der ggT.

Allerdings ist diese Verfahren langsam, wenn $|a|$ und $|b|$ ähnlich groß sind; denn die Anzahl der durchgeführten Operationen liegt in der Größenordnung von $\min\{a, b\}$.

Die Idee für ein in solchen Fällen schnelleres Verfahren beruht auf folgendem Hilfsatz.

0.3.1 Hilfsatz *a, b und x seien ganze Zahlen. x ist genau dann ein gemeinsamer Teiler von a und b, wenn x ein gemeinsamer Teiler von a und $b - a$ ist.*

Beweis. $\Rightarrow$: Ist x ein Teiler von a und b, so gibt es ein $a' \in \mathbb{Z}$ mit $a = xa'$ und ein $b' \in \mathbb{Z}$ mit $b = xb'$. Dann gilt $b - a = xb' - xa' = x(b' - a')$ und $b' - a' \in \mathbb{Z}$; also teilt x auch $b - a$.
$\Leftarrow$: Ist x ein Teiler von a und $b - a$, so gibt es ein $a' \in \mathbb{Z}$ mit $a = xa'$ und ein $c \in \mathbb{Z}$ mit $b - a = xc$. Dann gilt $b = b - a + a = xc + xa' = x(c + a')$ und $c + a' \in \mathbb{Z}$; also teilt x auch b.

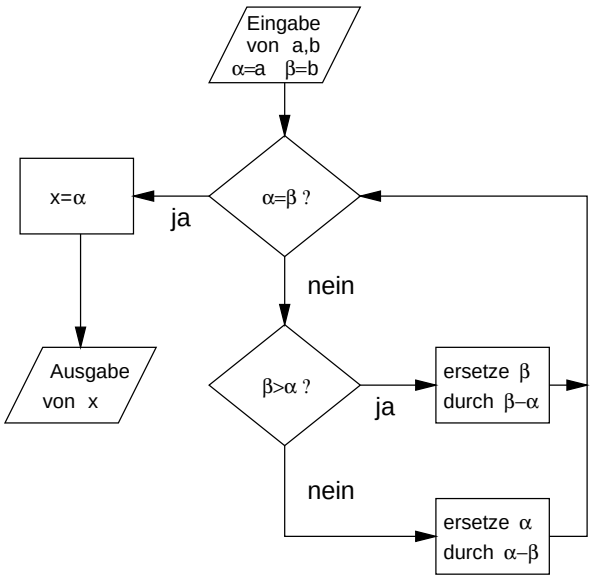
0.3.2 Ein Algorithmus zur Lösung des ggT-Problems

Eingabe: Zwei natürliche Zahlen a und b.
Ausgabe: Eine natürliche Zahl. x

Setze $\alpha = a$ und $\beta = b$.

1 Wenn $\alpha = \beta$, dann setze $x := \alpha$ und beende die Rechnung, sonst führe Schritt 2 aus.

2 Wenn $\beta > \alpha$, dann ersetze β durch $\beta - \alpha$ und führe Schritt 1 aus, sonst ersetze α durch $\alpha - \beta$ und führe Schritt 1 aus.



0.3.3 Satz *Für je zwei natürliche Zahlen a und b berechnet obiger Algorithmus eine natürliche Zahl x, die ggT von a und b ist.*

Beweis. Zwei Dinge sind zu zeigen:

a) Für jede Eingabe zweier natürlicher Zahlen a und b *terminiert* der Algorithmus, d.h. er stoppt nach der Ausführung endlich vieler Schritte.
b) Stoppt der Algorithmus, so ist x ein ggT von a und b; d.h. der Algorithmus ist *korrekt*.

Beweis von a): Bei jeder Ausführung von Schritt 2 wird entweder α oder β um mindestens 1 verkleinert. Beide Zahlen bleiben aber positiv. Folglich kann Schritt 2 nur endlich oft durchlaufen werden. Weil Schritt 2 stets ausgeführt wird, wenn in Schritt 1 die Bedingung $\alpha = \beta$ verletzt ist, muß nach endlich vielen Iterationen $\alpha = \beta$ gelten und der Algorithmus terminieren.

Beweis von b): Nach dem Hilfsatz 0.3.1 wissen wir, daß eine natürliche Zahl x genau dann ein Teiler der Zahlen α und β vor Durchlauf von Schritt 2 ist, wenn x auch ein Teiler der Zahlen α und β nach Durchlauf von Schritt 2 ist. Das bedeutet, daß bei Durchlauf von Schritt 2 der ggT von α und β sich nicht ändert (obwohl sich α oder β ändert). Nach a) stoppt der Algorithmus nach endlich vielen Schritten, und es gilt dann $\alpha = \beta$. Folglich ist $x = \alpha$ der ggT von α und β und nach obigen Überlegungen auch von den Eingabewerten a und b. q.e.d.

Bemerkungen Einen formal präziseren Beweis führt man mit Hilfe vollständiger Induktion nach der Anzahl der Iterationen von Schritt 2. Der obige Algorithmus ist kein besonders guter Algorithmus zur Berechnung des ggT; er dient hier nur als Demonstrationsbeispiel für diverse Begriffe und Phänomene. Wenn eine der beiden Zahlen klein und die andere groß ist, benötigt obiger Algorithmus viele Rechenschritte. Das kann man offensichtlich vermeiden, wenn man als Rechenoperation die Division mit Rest zur Verfügung hat (Euklidischer Algorithmus).

0.3.4 Verhalten des Algorithmus bei ganzzahligen Eingabewerten

Die Eingabewerte seien $a \leq 0$ und $b > 0$: Wegen $a \neq b$ wird Schritt 2 mindestens einmal ausgeführt. Gilt vor der Ausführung von Schritt 2 $\alpha < 0$ und $\beta > 0$, so bleibt α unverändert und β wird vergrößert! Deswegen kann durch weitere Ausführungen von Schritt 2 nie die Abbruchbedingung $\alpha = \beta$ erreicht werden. *Der Algorithmus terminiert nicht!* Er liefert keinen Ausgabewert.

Der Fall $a > 0$ und $b \leq 0$ geht analog.

Die Eingabewerte a und b seien beide ≤ 0: Ist $a \neq b$, so wird Schritt 2 ausgeführt. Dabei wird eine der beiden Zahlen α oder β positiv, während die andere unverändert, also negativ bleibt. Somit befindet man sich in obiger Situation; *der Algorithmus terminiert nicht.*

Ist $a = b \leq 0$, so terminiert der Algorithmus; der Ausgabewert $x = a$ ist aber nicht der ggT von a und b, weil z.B. 1 ein Teiler von a und b ist, der größer als x ist. Der Ausgabewert ist keine korrekte Lösung des ggT-Problems.

0.4 Ein Sortier-Problem

Sortiere n reelle Zahlen $a_1, \ldots, a_n$ der Größe nach, d.h. finde eine Permutation $f\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$, für die $a_{f(1)} \leq \ldots \leq a_{f(n)}$ gilt.

0.4.1 Ein einfaches Sortierverfahren

Eingabe: Eine Folge $a_1, \ldots, a_n$ von n reellen Zahlen.
Ausgabe: Die Folge der Werte $f(1), \ldots, f(n)$ einer Permutation $f\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$.

$I := \{1, \ldots, n\}$

Solange $I \neq \emptyset$, führe folgendes aus:

bestimme ein $j \in I$, so daß $a_j \leq a_i$ für jedes $i \in I$; $\star$
entferne j aus I;
gib j aus.

0.4.2 Bemerkung Obiges Verfahren enthält eine nicht präzise bestimmte Stelle, nämlich die mit $\star$ bezeichnete Zeile. Denn es kann mehrere Indizes j mit $a_j \leq a_i$ für jedes $i \in I$ geben. Es ist nicht angegeben, wie einer dieser Indizes ausgewählt wird. Je nach Auswahl gibt das Verfahren eine andere Permutation von $\{1, \ldots, n\}$ aus. Das Verfahren ist nicht wohldefiniert. Es gibt aber einen einfachen Ausweg: ersetze die Zeile $\star$ durch folgende:

wähle das kleinste $j \in I$, für das $a_j \leq a_i$ für jedes $i \in I$ gilt;

0.5 Der Begriff des Algorithmus

0.5.1 Definition *Ein **Algorithmus** ist eine Beschreibung, wie aus gewissen Eingabedaten durch elementare Verarbeitungsschritte Ausgabedaten erzeugt werden. Dabei müssen folgende Bedingungen erfüllt sein:*

0.5.1.1 Finitheit: *Die Beschreibung ist ein endliches Stück Text; insbesondere werden nur endlich viele elementare Verarbeitungsschritte erwähnt.*

0.5.1.2 Effektivität: *Die elementaren Verarbeitungsschritte müssen - zumindest prinzipiell - von einer Maschine oder einem Menschen (ohne Einsatz von Intelligenz) ausführbar sein.*

Bemerkung Folgender Schritt ist nicht für beliebige Werte von x effektiv: Wenn die Dezimalbruchentwicklung von x eine 1 enthält, dann stoppe.

In der theoretischen Informatik führt man einfache Maschinenmodelle ein (z.B. Turingmaschinen oder unbegrenzte Registermaschinen), deren elementare Verarbeitungsschritte durch kurze Textstücke bezeichnet werden. Ein Algorithmus ist dann eine endliche Folge solcher Textstücke; er wird auch Programm genannt. Für reale Anwendungen sind diese Maschinen zu primitiv; Programme zur Lösung selbst einfachster Probleme sind meist extrem lang und unübersichtlich. Das gilt auch für die Maschinensprache der üblichen Prozessoren. Deshalb verwendet man höhere Programmiersprachen (wie C, PASCAL, FORTRAN, COBOL, C++, JAVA). Das sind zwar auch formale Sprachen, die aus einigen wenigen Textbausteinen aufgebaut sind. Sie enthalten aber Beschreibungen von Verarbeitungsschritten, die der Umgangssprache ähnlich

sind, und ihre elementaren Verarbeitungsschritte sind teilweise wesentlich komplexer als die der Maschinensprache der Prozessoren. Dadurch werden die Programme viel kürzer und für den Menschen leichter verständlich.

Um einen in einer höheren Programmiersprache geschriebenen Algorithmus auf einem Rechner ablaufen lassen zu können, muß er erst in die Maschinensprache des Rechners übersetzt werden. Das wird von Compilern oder Interpretern bewerkstelligt (später mehr).

In der Literatur werden Algorithmen oft nicht in einer konkreten Programmiersprache beschrieben, sondern in irgendeiner erfundenen Pseudoprogrammiersprache, für die es keinen Compiler gibt, die jedoch einfacher als wirkliche Programmiersprachen ist, aber trotzdem viele übliche Verarbeitungsschritte enthält wie z.B. Fallunterscheidungen und Schleifen. Wir werden folgende Konstrukte verwenden.

`if` *Aussage* `then` *Anweisungen1*
`else` *Anweisungen2*
`endif`

Dies soll folgendes bedeuten: Wenn *Aussage* wahr ist, wird der Teil *Anweisungen1*, aber nicht der Teil *Anweisungen2* ausgeführt; ansonsten wird umgekehrt der Teil *Anweisungen2*, aber nicht der Teil *Anweisungen1* ausgeführt. Der Teil *Anweisungen2* darf leer sein; dann wird die Zeile mit `else` einfach weggelassen.

`while` *Aussage* `do`
Anweisungen
`endwhile`

Dies soll folgendes bedeuten: Solange *Aussage* wahr ist, wird der Teil *Anweisungen* ausgeführt.

`foreach` $x \in X$ `do`
Anweisungen
`endfor`

Dies soll folgendes bedeuten: Für jedes einzelne Element x in der Menge X wird der Teil *Anweisungen* genau einmal ausgeführt. Vorsicht! Damit ist nicht festgelegt, in welcher Reihenfolge die Elemente von X ausgewählt werden. In Programmen für reale Rechner muß das natürlich spezifiziert sein. Deshalb sind die `for`-Konstrukte wirklicher Programmiersprachen eingeschränkter. Um nur die Grundidee eines Algorithmus zu beschreiben, ist oft eine genaue Festlegung der Reihenfolge nicht nötig. Man sollte sich aber immer vor Augen halten, daß das Ergebnis des Algorithmus sehr wohl von der Reihenfolge abhängen kann, und daher überprüfen, inwieweit dies für die jeweilige Anwendung relevant ist.

Zur Trennung einzelner Verarbeitungsschritte werden wir wie in C das Semikolon verwenden. Leerzeichen und Einrückungen dienen nur zur besseren Übersichtlichkeit.

0.5.2 Vereinbarung Wie wir beim ggT-Algorithmus in 0.3 gesehen haben, ist die Angabe der zugelassenen Eingabedaten für das Verhalten des Algorithmus wichtig. *Wir vereinbaren deshalb, daß die Angabe der zulässigen Eingabe- und Ausgabedaten ein Bestandteil jedes Algorithmus ist.*

Die Ausgabe eines Datums w mit anschließender Beendigung des Algorithmus wird oft durch `return` w bezeichnet.

0.5.3 Bemerkung Im üblichen Sprachgebrauch geht man mit dem Begriff Algorithmus etwas schlampig um. Man bezeichnet damit verschiedene Dinge:

- den Text der Beschreibung des Verfahrens (die Syntax der Beschreibung)
- das beschriebene Verfahren (die Semantik der Beschreibung)
- die Ausführung des Verfahrens (den Rechenprozeß)

0.5.4 Definition

a) Ein Algorithmus *terminiert* für ein zulässiges Eingabedatum x, wenn seine Ausführung bei Eingabe von x nach Durchlauf endlich vieler Verarbeitungsschritten stoppt.

b) Ein Algorithmus heißt *terminierend*, wenn seine Ausführung für jedes zulässige Eingabedatum nach Durchlauf endlich vieler Verarbeitungsschritte stoppt.

c) Ein Algorithmus heißt *deterministisch*, wenn zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besteht.

0.5.5 Bemerkung zu b) In praktischen Anwendungen wird man meist wollen, daß die eingesetzten Algorithmen terminierend sind. Es gibt aber Ausnahmen, z.B. einen Kommandozeileninterpreter (Shell) oder ein ganzes Betriebssystem. Nicht terminierende Algorithmen spielen auch in der theoretischen Informatik eine Rolle.

0.5.6 Bemerkung zu c) In dieser Vorlesung werden wir nur deterministische Algorithmen betrachten. Nichtdeterministische spielen eine wichtige Rolle in der theoretischen Informatik (Komplexitätstheorie), aber auch in Anwendungen z.B. als stochastische Suchverfahren, mit denen extrem komplexe Probleme zumindest approximativ gelöst werden können.

0.6 Algorithmen als Realisierung von Abbildungen

Durch die Ausführung eines terminierenden, deterministischen Algorithmus ALG wird aus einem zulässigen Eingabedatum x ein Ausgabedatum y erzeugt. Der Algorithmus berechnet also eine Abbildung $f\colon E \to A$, wobei E die Menge der zulässigen Eingabedaten und A die Menge der Ausgabedaten ist. Diese funktionale Betrachtungsweise findet sich in vielen Programmiersprachen, so z.B. in C, wo jedes Programm aus einer Ineinandersetzung von Funktionen besteht. Es ist daher üblich, den Namen des Algorithmus und die von ihm realisierte Funktion synonym zu verwenden; beispielsweise bezeichnet `ALG(x,y)` also das Ausgabedatum (auch Rückgabwert genannt) des Algorithmus `ALG` bei Eingabe von x und y.

0.7 Rekursive Algorithmen

Ein Algorithmus heißt *rekursiv*, wenn er in seiner Beschreibung selbst als Verarbeitungsschritt vorkommt und ausgeführt wird.

Wir geben ein einfaches Beispiel dafür. Man erinnere sich an die übliche Definition der Fakultät $n!$ einer natürlichen Zahl n, nämlich $n! = n \cdot (n-1) \cdot \ldots \cdot 1 = \prod_{j=1}^{n} j$. Daran sieht man, daß für jedes $n \in \mathbb{N}$ mit $n > 1$ die Gleichung $n! = n \cdot (n-1)!$ gilt. Sie kann man zum Ausgangspunkt eines rekursiven Algorithmus zur Berechnung der Fakultätsfunktion machen.

0.7.1 Algorithmus RekFak(n)

```
if n = 1 then return 1;
         else return n· RekFak(n − 1);
endif
```

Bei Ausführung des Algorithmus mit einem Eingabewert n entstehen also weitere, ineinander verschachtelte Aufrufe des Algorithmus, die jeweils durch die Anweisung `return` beendet werden und dabei ihr Resultat an die aufrufende Instanz zurückgeben. Damit der Algorithmus terminiert, muß die Rekursionstiefe d.h. die Anzahl der ineinander verschachtelten Aufrufe beschränkt sein. Dies wird dadurch erreicht, daß der Eingabewert bei jedem weiteren Aufruf um 1 kleiner wird, bis schließlich die Bedingung $n = 1$ erfüllt ist und somit kein weiterer Selbstaufruf des Algorithmus mehr erfolgt. So eine Abbruchbedingung ist essentiell für jeden rekursiven Algorithmus, damit er terminiert. Verfolgen Sie einmal im Detail die Arbeitsweise für kleine Eingabewerte n.

Viele Probleme lassen sich sehr elegant durch rekursive Algorithmen lösen. Deshalb ermöglichen viele Programmiersprachen auch die Implementierung rekursiver Algorithmen. Auch für theoretische Untersuchungen, insbesondere für asymptotische Abschätzungen der Rechenzeit, sind rekursive Algorithmen gut geeignet. In praktischen Implementierungen jedoch braucht jeder Aufruf des Algorithmus eine gewisse Zeit für die rechnerinterne Organisation. Deshalb ergeben sich bei hoher Rekursionstiefe oft unerquicklich große Rechenzeiten, und man bevorzugt nichtrekursive Algorithmen, die jedoch meist komplizierter und unübersichtlicher sind.

0.8 Algorithmisch unlösbare Probleme, nicht berechenbare Funktionen

Für Anwendungen ist es wichtig zu wissen, ob ein gegebener Algorithmus für gewisse Eingabedaten terminiert und ob die durch den Algorithmus realisierte Abbildung mit einer gewünschten übereinstimmt (Korrektheit des Algorithmus). Statt dies in jedem Einzelfall mit Ad-hoc-Methoden zu beantworten zu versuchen, könnte man auch einen Algorithmus suchen, der diese Fragen beantwortet.

0.8.1 Das Halteproblem Gibt es einen Algorithmus H mit folgender Eigenschaften:

a) Die zulässigen Eingabedaten E sind die Paare (P, x), deren erste Komponente P ein Algorithmus und deren zweite Komponente x ein zulässiges Eingabedatum für P ist.

b) Die Menge der Ausgabedaten ist $\{0, 1\}$.

c) Für jedes zulässige Eingabedatum (P, x) terminiert H mit dem Ausgabedatum 1, wenn P bei Eingabe von x terminiert, und mit 0, wenn P bei Eingabe von x nicht terminiert. Mit anderen Worten realisiert H die Funktion

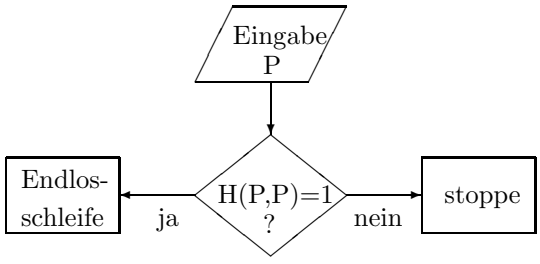
$$f\colon E \to \{0,1\}, (P,x) \mapsto \begin{cases} 1, & \text{wenn } P \text{ bei Eingabe von } x \text{ terminiert} \\ 0 & \text{sonst} \end{cases}$$

In der Theorie der Berechenbarkeit zeigt man, daß es so einen Algorithmus H nicht gibt. Die Funktion f ist nicht algorithmisch berechenbar. Etwas verständlicher ausgedrückt: Es gibt kein C-Programm, das von jedem C-Programm feststellen kann, ob es zu gegebenen Eingabedaten terminiert oder nicht.

Auch die Korrektheit von Algorithmen ist nicht algorithmisch entscheidbar. Diese Aussagen schließen aber nicht aus, daß für Teilklassen von Algorithmen das Halte- oder das Korrektheitsproblem algorithmisch entscheidbar ist.

0.8.2 Grobe Skizze der Beweisidee für die Unlösbarkeit des Halteproblems

Der Beweis wird indirekt geführt. Und zwar zeigen wir, daß die Annahme, es gäbe einen Algorithmus, der das Halteproblem löst, zu einem Widerspruch führt. Weil man aus etwas Richtigem durch korrekte Schlüsse nichts Falsches folgern kann (*tertium non datur*), muß also die Annahme falsch sein. Wir machen also folgende Annahme: Es gibt einen Algorithmus H, der das Halteproblem löst.
E sei die Menge der Algorithmen P, für die jedes Textstück eine zulässige Eingabe ist. Wir konstruieren einen neuen Algorithmus G, der die Algorithmen in E als zulässige Eingabedaten besitzt, auf folgende Weise:



Wie verhält sich G bei Eingabe von G?

1.Fall: G terminiert bei Eingabe von G, also $H(G, G) = 1$. Dann gerät G in eine Endlosschleife, d.h. G terminiert nicht. Das ist ein Widerspruch!

2.Fall: G terminiert bei Eingabe von G nicht, also $H(G, G) = 0$. Dann stoppt G. Das ist wieder ein Widerspruch! q.e.d.

0.9 Komplexitätsbetrachtungen

In praktischen Anwendungen genügt es nicht, für ein Problem einen Lösungsalgorithmus zu finden; der Algorithmus muß auch ausreichend effizient sein, d.h. seine Ausführungszeit und der Speicherplatzbedarf dürfen die durch die Anwendung vorgegebenen Grenzen nicht überschreiten. Der oben angeführte Algorithmen zum Sortieren 0.4.1 ist z.B. nicht sehr effizient. Um den Rechenaufwand genauer zu analysieren, ersetzen wir die Zeile $\star$ durch eine `foreach`-Anweisung und erhalten folgenden Sortieralgorithmus.

0.9.1 Ein Sortieralgorithmus

Eingabe: Eine Folge $a_1, \ldots, a_n$ von n reellen Zahlen.
Ausgabe: Die Folge der Werte $f(1), \ldots, f(n)$ einer Permutation
$f\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$.
$I := \{1, \ldots, n\}$;
`while` $I \neq \emptyset$ `do`
setze $J := I$ und $m := \infty$;
`foreach` $j \in J$
`if` $a_j < m$ `then` setze $j_{min} := j$ und ersetze m durch a_j;
`endforeach`
entferne j_{min} aus I;
gib j_{min} aus;
`endwhile`
stoppe

Als Maß für die Rechenzeit wählen wir die Anzahl der Vergleiche zwischen zwei der a_i oder zwischen einem der a_i und m. Beim l-ten Durchlauf der äußeren Schleife werden $n - l + 1$ Vergleiche durchgeführt, insgesamt also $n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$. Die Anzahl der Vergleiche wächst also in Abhängigkeit der Anzahl n der zu sortierenden Zahlen wie n^2. Es gibt bessere Sortieralgorithmen, deren Rechenzeit nur wie $n \log n$ wächst. Für große n ist $n \log n$ viel kleiner als n^2. Ein quadratisches Wachstum ist noch recht harmlos. Es gibt viel schlimmere Fälle.

0.9.2 Ein Partitionsproblem

Gegeben seien n reelle Zahlen $a_1, \ldots, a_n$. Gibt es eine Teilmenge J von $\{1, \ldots, n\}$, so daß $\sum_{j \in J} a_j = \sum_{i \notin J} a_i$?

0.9.3 Ein Algorithmus zur Lösung des Partitionsproblems

Eingabe: n reelle Zahlen $a_1, \ldots, a_n$
Ausgabe: Eine Teilmenge J von $\{1, \ldots, n\}$
Setze $s := a_1 + \ldots + a_n$.
`foreach` Teilmenge J von $\{1, \ldots, n\}$ `do`
berechne $s_J := \sum_{j \in J} a_j$;
`if` $s_J = \frac{1}{2}s$ `then` gib J aus und stoppe;
`endif`
`endfor`
gib den Text *Es gibt keine gesuchte Partition* aus und stoppe.

Wir lassen momentan offen, wie man alle Teilmengen J von $\{1,\ldots,n\}$ bestimmen kann; dafür gibt es aber einen einfachen Algorithmus. Was uns hier interessiert, ist nur, daß $\{1,\ldots,n\}$ genau 2^n Teilmengen hat. Im ungünstigsten Fall wird also die Schleife 2^n-mal durchlaufen; die Laufzeit wächst also exponentiell! Es ist nicht bekannt, ob es einen Algorithmus zur Lösung des Partitionsproblems gibt, dessen Rechenzeit $T(n)$ durch ein Polynom in n nach oben beschränkt ist, also für große n viel kleiner als 2^n ist. Präziser formuliert: Es ist nicht bekannt, ob es ein Polynom p, ein positives $c \in \mathbb{R}$ und ein $n_0 \in \mathbb{N}$ gibt, so dass für jedes $n \in \mathbb{N}$ mit $n \geq n_0$ die Ungleichung $T(n) \leq c \cdot |p(n)|$ gilt.

Untersuchungen dieser Art werden in der Komplexitätstheorie durchgeführt. Dabei interessiert man sich vor allem für das asymptotische Wachstum der Rechenzeit, wenn die Größe der Eingabe gegen Unendlich strebt.

0.9.4 Wachstumsklassen

$f\colon \mathbb{N} \to \mathbb{R}$ sei eine nichtnegative Funktion. Dann definiert man

$$O(f) := \{\, g\colon \mathbb{N} \to \mathbb{R} \mid \exists\, c > 0\ \exists k\ \in \mathbb{N}\ \forall\, n > k \quad 0 \leq g(n) \leq c \cdot f(n) \,\}$$

d.h. $O(f)$ ist die Menge aller Funktionen $g\colon \mathbb{N} \to \mathbb{R}$ derart, dass ein $c > 0$ und ein $k \in \mathbb{N}$ existieren, so dass für jedes $n \in \mathbb{N}$, das größer als k ist, $0 \leq g(n) \leq c \cdot f(n)$ gilt.

Analog definiert man

$$\Omega(f) := \{\, g\colon \mathbb{N} \to \mathbb{R} \mid \exists\, c > 0\ \exists k\ \in \mathbb{N}\ \forall\, n > k \quad c \cdot f(n) \leq g(n) \,\}$$

d.h. $\Omega(f)$ ist die Menge aller Funktionen $g\colon \mathbb{N} \to \mathbb{R}$ derart, dass ein $c > 0$ und ein $k \in \mathbb{N}$ existieren, so dass für jedes $n \in \mathbb{N}$, das größer als k ist, $c \cdot f(n) \leq g(n)$ gilt.

Etwas unpräzise, aber suggestiv formuliert ist $O(f)$ die Menge der Funktionen, die asymptotisch höchstens so schnell wie f wachsen, und $\Omega(f)$ die Menge der Funktionen, die asymptotisch mindestens so schnell wie f wachsen. Und $\Theta(f) := O(f) \cap \Omega(f)$ ist die Menge der Funktionen, die asymptotisch so wie f wachsen.

Um die Geschwindigkeit von Algorithmen abzuschätzen und zu vergleichen, versucht man, die Rechenzeit T eines Algorithmus (als Funktion der Größe $n \in \mathbb{N}$ der Eingabe) in Wachstumsklassen bekannter, einfacher Funktionen f einzuordnen. So haben wir oben gezeigt, dass die Anzahl $T(n)$ der Vergleiche, die der einfache Sortieralgorithmus 0.9.1 zum Sortieren von n Zahlen benötigt, in $\Theta(n^2)$ liegt. Wie dort schon erwähnt gibt es effizientere Sortieralgorithmen, für deren Rechenaufwand T gilt $T \in O(n \log n)$. Mehr dazu in einem späteren Kapitel.

Kapitel 1

Grundlegende Strukturen eines C-Programms

C ist eine Programmiersprache, die in den frühen 70ern von Dennis Ritchie erfunden und auf einem UNIX-System implementiert wurde. Sie hatte als Vorgänger BCPL von Martin Richards (1965-67) und B (1970) von Ken Thompson (das interpretiert wurde) und wurde von Brian Kernighan und Dennis Ritchie bis zu einem de facto-Standard 1978 (K&R-C) weiterentwickelt. Für viele Rechnerplattformen wurden C-Compiler entwickelt. Leider unterschieden sie sich meist mehr oder weniger stark in ihrem Verhalten, auch weil in der Definition des K&R-C Lücken waren. Daher wurde jahrelang um eine vernünftige Vereinheitlichung gerungen, bis schließlich 1989 vom American National Standards Institute eine Normierung vorgenommen wurde, die als ANSI-C bekannt ist und 1999 zu ANSI-C99 erweitert wurde. Wir werden uns im folgenden auf ANSI-C konzentrieren.

Die Bedeutung von C liegt in seiner weiten Verbreitung und seiner engen Verflechtung mit dem UNIX-Betriebssystem. Es ist keine schöne Sprache, die nach höheren Gesichtspunkten konstruiert wurde, sondern eher eine fast natürlich gewachsene. C enthält höhere, abstrakte Strukturelemente, läßt aber auch sehr hardwarenahe Programmierung zu. Dies ist zwar oft recht praktisch, aber auch gefährlich, weil fehlerträchtig. Andere Sprachen wie PASCAL, MODULA, JAVA sind so konstruiert, daß ein solch fehlerträchtiger Programmierstil garnicht möglich ist. Allerdings wird es damit aber auch schwierig, systemnah zu programmieren. C dagegen ist dafür gut geeignet; viele UNIX-Varianten sind in C programmiert (z.B. LINUX). Dies ist sicher einer der Gründe für die weite Verbreitung von C.

C besteht aus nur wenigen Schlüsselwörtern; es enthält im Gegensatz zu anderen Programmiersprachen (wie PASCAL oder BASIC) keine Ein-Ausgabe-Routinen oder höhere mathematischen Funktionen. Solche Dinge sind in Bibliotheken ausgelagert, die nach Bedarf an den Objektcode gelinkt werden. Dadurch erhält C eine große Flexiblität (aber auch eine gewisse Fehlerträchtigkeit).

C hat die folgenden 32 Schlüsselwörter, die in keinem anderen Sinn verwendet werden dürfen:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Achtung! Bei C wird Groß- und Kleinschreibung unterschieden! Die Schlüsselwörter werden alle klein geschrieben.

Die offizielle ANSI-Normierung von C verwendet für die Grammatikregeln eine EBNF-ähnliche Notation (siehe Kapitel 3). Weil die ANSI-C-Normierung anfangs oft schwer zu verstehen ist, werden wir zunächst umgangssprachliche Beschreibungen geben.

1.1 Prinzipielle Gestalt eines C-Programms

Ein Programm ist ein Stück Text in einer formalen Sprache (siehe Kapitel 3), das durch einen sogenannten Compiler in die Sprache des jeweiligen Prozessors im Rechner, die Maschinensprache, übersetzt wird. Ein *Compiler* ist selbst ein Programm in Maschinensprache, das den Rechner dazu bewegt, den Programmtext (die Quelldatei) einzulesen und daraus ein lauffähiges Programm in Maschinensprache, den sogenannten Objektcode oder Binärcode, zu erzeugen. Während Programme in höheren Programmiersprachen weitgehend rechnerunabhängig geschrieben werden können, sind Programme in Maschinensprache völlig vom Prozessortyp und der Rechnerarchitektur abhängig.

In der Programmiersprache C ist ein Programm einen Folge von Deklarationen und Definitionen von Funktionen und Variablen.

Eine *Funktion* ist ein Programmteil, den man als eigenen Algorithmus auffassen kann; einer Funktion kann man Datenwerte als Eingabe übergeben, und sie kann Ausgabewerte zurückgeben. Eine Funktion besteht aus einer Deklaration, in der für den Compiler angegeben wird, welche Art von Daten als Ein- und Ausgabewerte zulässig sind, und einer Definition, in der durch eine Folge von Anweisungen festgelegt wird, was die Funktion tut. Es ist zugelassen, daß eine Funktion keinen Eingabewert und/oder keinen Rückgabewert hat; Funktionen ohne Rückgabewert heißen in anderen Sprachen manchmal Prozeduren. Eine Deklaration oder Definition einer Funktion darf nicht innerhalb der Definition einer Funktion stehen.

Ein C-Programm muss genau eine Funktion namens `main` enthalten. Mit dem Objektcode dieser Funktion beginnt der Rechner die Ausführung des Programms. Gibt es keine Funktion dieses Namens, so kann der Compiler daraus kein lauffähiges Programm machen.

Ein C-Programm hat also typischerweise folgende Gestalt.

globale Deklarationen von Variablen und Funktionen
Initialisierung globaler Variablen

function1( )
{
    lokale Deklarationen und Definitionen von Variablen
    Anweisungen
}

function2( )
{
    lokale Deklarationen und Definitionen von Variablen
    Anweisungen
}

main( )
{
    lokale Deklarationen und Definitionen von Variablen
    Anweisungen
}

Variablen sind mit Namen versehene Speicherplätze, in denen Daten abgelegt werden. Durch die Deklaration von Variablen wird dem Compiler gesagt, um welche Art von Daten es sich handelt. Dabei wird aber nicht unbedingt schon tatsächlich Speicherplatz reserviert. Dies geschieht erst durch die Definition (Genaueres später). Deklarationen und Definitionen, die außerhalb jeder Funktion stehen, gelten für das gesamte Programm; solche, die innerhalb einer Funktion stehen, sind lokal und gelten nur innerhalb des kleinsten Blockes, der durch ein Klammerpaar { } begrenzt wird.

1.1.1 Beispiel

```
#include <stdio.h>
int main(void) {
   printf("hello world\n");
   return 0;
}
```

Dieses Programm enthält im wesentlichen nur die Definition einer einzigen Funktion, nämlich `main`. Wegen des Wortes `void` im Argument ist sie als Funktion ohne Eingabeparameter definiert. Sie enthält als einzige Anweisung den Aufruf der Funktion `printf`, einer Funktion aus der Standardbibliothek von C, die in diesem Fall einfach den im Argument angegebenen Textstring ausdruckt. Dabei ist `\n` ein Steuerzeichen, das einen Zeilenvorschub bewirkt. Die Zeile `return 0` bewirkt lediglich, daß bei Programmende dem Betriebssystem als Rückgabewert eine `0` zurückgegeben wird als Zeichen dafür, daß das Programm ordnungsgemäß beendet wurde. Läßt man diese Zeile weg, so wird sich eventuell der Compiler beschweren, weil die Funktion `main` per Definition einen Integerwert zurückgibt.

Damit der Compiler weiß, wie er den Aufruf einer Funktion übersetzen soll, muß im Quelltext bereits vor dem Aufruf einer Funktion eine Deklaration oder die Definition der Funktion stehen. Bei kleineren Programmen verzichtet man meist auf gesonder-

te Deklarationen und gibt einfach direkt die Definitionen an; infolgedessen steht die Funktion `main` am Ende. Benutzt man jedoch Funktionen aus irgendwelchen Bibliotheken, so müssen sie, bevor man sie aufruft, deklariert werden. Zu diesem Zweck gibt es zu jeder Bibliothek eine sogenannte Headerdatei, in der die Deklarationen ihrer Funktionen (und anderes) stehen. Diese Headerdatei hängt man vor den eigenen Programmtext. Weil die Headerdateien recht umfangreich sein können, würden die Programme ziemlich unübersichtlich. Dem kann man begegnen, indem man eine C-spezifische Sonderheit ausnutzt, nämlich den C-Präprozessor. Bei jedem Compileraufruf wird zunächst der Präpozessor aufgerufen, der in dem Quelltext nach gewissen Regeln Textersetzungen vornimmt. Die Befehle für den Präprozessor fangen alle mit einem `#` am Zeilenanfang an. Einer dieser Befehle lautet `#include <`*datei*`>`. Er bewirkt, daß der Präprozessor genau an dieser Stelle den Inhalt der Datei mit Namen *datei* einfügt. Diese Datei muß in einem Ordner stehen, der standardgemäß durchsucht wird (ansonsten muß man den vollen Pfadnamen in Anführungszeichen statt in spitzen Klammern angeben). Die Headerdateien der mit dem Compiler mitgelieferten Bibliotheken stehen üblicherweise in Ordnern namens `include`, die vom Compiler automatisch durchsucht werden. Es ist üblich, daß die Namen von Headerdateien die Endung `.h` haben. In obigem Beispiel steht in `stdio.h` die Deklaration der Funktion `printf`. Diese Funktion ist sehr wichtig, um Zahlen und Text auszudrucken. Ihre Benutzung erfordert aber einige Übung.

1.1.2 Beispiel

```
#include <stdio.h>

int quadrat(int x) {
   return x*x;
}

int main(void) {
   int i;
   i=3;
   printf("Das Quadrat von %d ist %d\n",i,quadrat(i));
   return 0;
}
```

In diesem Beispiel wird eine Funktion `quadrat` definiert, die als Eingabe und Ausgabe jeweils einen Wert vom Typ Integer hat. Innerhalb der Funktion wird der Eingabewert mit sich selbst multipliziert und mit dem Befehl `return` zurückgegeben. In `main` wird eine Variable vom Typ Integer definiert, dann wird ihr der Wert 3 zugewiesen und anschließend die Funktion `printf` aufgerufen. Dabei ist das erste Argument ein sogenannter Formatstring, in dem einerseits Text angegeben ist, andererseits aber auch zwei Platzhalter `%d` für zwei Zahlenwerte vom Typ Integer. Die beiden Werte sind in den beiden folgenden Argumenten angegeben, und zwar einmal der Wert der Variablen `i` und zum anderen der Rückgabewert der Funktion `quadrat`, wenn sie mit dem Wert von `i` als Eingabewert aufgerufen wird.

Für die Funktionen in der Standardbibliothek gibt es meist eine Online-Hilfe; auf UNIX-Systemen gibt man dazu auf der Kommandozeile `man` *Funktionsname* ein. Man muß also den Namen der Funktion wissen; weiß man ihn nicht genau, hilft es

manchmal, den Namen einer ähnlichen Funktion anzugeben, weil in den Manualseiten oft Querverweise enthalten sind, oder es mit dem Kommando `apropos` *Stichwort* zu probieren, wobei man als Stichwort irgendwelche ähnliche Begriffe angibt. Für die GNU-Programme hat sich ein anderes Online-Hilfe-System eingebürgert, das man mit `info` oder `info` *programmname*, also z.B. `info gcc`, aufruft (die Bedienung ist etwas gewöhnungsbedürftig).

1.1.3 Die Compilierung von Programmen Wir betrachten momentan nur UNIX-Systeme, insbesondere LINUX-Rechner. Wenn das Programm lediglich aus einer einzigen Quelldatei namens `prog.c` besteht, lautet der Compileraufruf im einfachsten Fall `cc prog.c`. Das entstehende Binärobjekt heißt dann (aus historischen Gründen) `a.out`, was nicht besonders sinnvoll ist. Um ihm einen sinnvollen Namen zu geben, verwendet man die Option `-o` *name*, also z.B. `cc prog.c -o prog`. In den Übungen werden wir den (kostenlosen) GNU-C-Compiler verwenden, der üblicherweise `gcc` heißt. Um `prog.c` zu übersetzen, gibt man also

```
gcc prog.c -o prog
```

ein. Der Compiler ruft nach dem Übersetzen automatisch den Linker auf, der Funktionen aus der Standardbibliothek, die im Programm benutzt werden, an den Objektcode bindet und ein ausführbares Binärobjekt namens `prog` erzeugt.

Bei komplexeren Programmen muß man beim Compilieren meist noch etliche Bibliotheken und andere Dinge als Optionen angeben; vielfach besteht ein Programm auch aus mehreren Quelldateien, sogenannten Moduln, die getrennt übersetzt und dann erst gelinkt werden. Um in solchen Fällen nicht immer komplizierte Compileraufrufe eintippen zu müssen, gibt es auf UNIX-Systemen ein sehr leistungsfähiges Werkzeug namens `make`. In einer Datei namens `Makefile` wird festgelegt, wie die Compilierung des gesamten Programms abzulaufen hat. Durch Aufruf von `make` wird dann die Compilierung tatsächlich durchgeführt. Das Programm `make` ist dabei so intelligent, daß es nur die Teile neu übersetzt, die seit der letzten Compilierung eine Änderung erfahren haben (das funktioniert natürlich nur dann richtig, wenn man im `Makefile` angegeben hat, in welcher Abhängigkeit die einzelnen Quelldateien voneinander stehen). Das Schreiben und Verstehen raffinierter Makefiles ist eine mühsame Kunst. Für unsere Zwecke in dieser Vorlesung werden aber einfache, leicht verständliche Makefiles völlig ausreichen. Am allereinfachsten ist es, überhaupt kein Makefile zu haben. Als Default wird dann bei Aufruf von `make prog` das Kommando `CC prog.c -o prog` ausgeführt, wobei CC eine Umgebungsvariable ist, der man mit CC=gcc den Wert gcc gibt.

Mit den Optionen `-ansi` und `-Wall` kann man den GNU-C-Compiler dazu bewegen, streng auf die Einhaltung der ANSI-Norm zu achten und alles, was ihm merkwürdig vorkommt, als Warnung mitzuteilen. Es empfiehlt sich, diese Optionen beim Compilieren zu verwenden, weil man auf diese Weise oft auf inhaltliche Programmierfehler mehr oder weniger direkt hingewiesen wird, obwohl keine syntaktischen Fehler vorliegen. Der Compileraufruf lautet dann `gcc prog.c -o prog -ansi -Wall`.

Unter anderen Betriebssystemen als UNIX gibt es andere Entwicklungsumgebungen (die jedoch meist nicht kostenlos sind). Für WINDOWS gibt es aber das Softwarepaket `cygwin`, das eine UNIX-ähnliche Umgebung mit der bash-Shell, dem GNU-C-Compiler und dem make-Tool emuliert. Man kann es im Internet von der Adresse

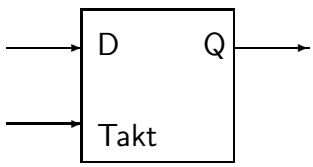
`www.cygwin.com` herunterladen. Eine Alternative ist, per Software einen virtuellen Rechner zu emulieren, auf dem auch ein anderes Betriebssystem laufen kann. Solch ein virtueller Rechner wird z.B. durch die Software `VMware player` realisiert. Sie kann bei `www.vmware.com` gratis heruntergeladen werden kann. Im Internet findet man Images von Linux-Distributionen, die in dem Fenster des `VMware player` einen Linux-Rechner booten.

1.1.4 Kommentierung Es ist eine alte Programmierererfahrung, daß man Programme, die man selbst geschrieben hat und gut verstanden hat, oft nach relativ kurzer Zeit keineswegs mehr auf Anhieb versteht. Noch viel schwerer sind Programme zu verstehen, die andere geschrieben haben. Deshalb sollte man es sich unbedingt zur Pflicht machen, in die Programme sog. Kommentare einzuflechten, die erklären, welchen Schritt des gesamten Algorithmus die jeweilige Programmzeile durchführen soll. Solche kommentierende Texte müssen mit `/*` beginnen und mit `*/` enden; dann werden sie nämlich beim Compilieren ignoriert. Achtung! Man kann solche Kommentare nicht ineinander schachteln, wie man es von der Klammerung arithmetischer Ausdrücke gewohnt ist. Die Zeichen `/*` und `*/` sind eher als Schalter zu verstehen, mit denen man von Programmtext auf Kommentartext und wieder zurückschalten kann. **Kommentieren Sie Ihre Programme ausführlich!**

1.2 Speicher

Übliche Rechner haben einen internen Arbeitsspeicher, der meist nicht allzu groß ist, aber sehr schnelle Zugriffe erlaubt, und größere, aber langsamere Massenspeicher, die oft als externe, eigene Geräte angeschlossen sind. Diese Massenspeicher sind typischerweise magnetische oder optische Platten oder Magnetbänder. Üblicherweise werden mehrere Daten in eine sogenannte Datei (file) zusammengefaßt, die dann unter einem Namen in dem Speicher abgelegt wird. Eine Datei ist rein programmiertechnisch gesehen einfach eine mit Namen versehene Folge von Einzeldaten, auf die nacheinander, jedoch nicht in beliebiger Reihenfolge zugegriffen werden kann.

Die Arbeitsspeicher bestehen aus elektronischen Halbleiterschaltungen. Die einfachsten Grundbausteine sind Schaltungen, die genau zwei stabile Zustände annehmen können (wie ein Kippschalter), sog. *Flip-Flops*. In einem Flip-Flop kann man also eine Alternativentscheidung *ja* oder *nein* speichern. Wir werden die beiden stabilen Zustände mit 0 und 1 bezeichnen. Eine Entscheidung zwischen 0 und 1 bezeichnet man als ein Bit. Ein Flip-Flop ist also ein Speicher für 1 Bit. Es gibt unterschiedliche Versionen von Flip-Flops.



D-Flip-Flop

Eine einfache ist das D-Flip-Flop. Es hat zwei stabile Zustände; in dem einen ist die

Spannung am Ausgang 0 Volt; in dem anderen ist sie 5 Volt. An dem Eingang D dürfen 0 Volt oder 5 Volt anliegen. Wird zu einem Zeitpunkt t der Takteingang von 0 Volt auf 5 Volt geschaltet, so wechselt das Flip-Flop in den Zustand, in dem die Spannung am Ausgang Q gleich der Spannung ist, die zum Zeitpunkt t an D anlag. Es bleibt in diesem Zustand bis zum nächsten 0 Volt-5 Volt-Wechsel am Takteingang.

Bei magnetischen Speichermedien und dynamischen Halbleiterspeichern liegt physikalisch gesehen kein reines bistabiles Verhalten vor (aber immerhin gibt es Hysteredeeffekte). Durch Einsatz von Schwellwertentscheidungen kann man sie aber trotzdem zur Speicherung einzelner Bits verwenden.

Organisation des Arbeitsspeichers Bei den meisten Rechnersystemen werden immer acht 1-Bit-Speicherzelle zusammengefaßt zu einem Speicherplatz für ein Byte (1 Byte = 8 Bit). Und diese 1-Byte-Speicherplätze werden durchnumeriert beginnend bei 0. Die Nummer eines Speicherplatzes heißt seine *Adresse*. Mit Hilfe dieser Adresse wählt der Prozessor die Speicherplätze aus, um ihren Inhalt zu lesen oder zu ändern.

1.3 Variablen

1.3.1 Definition *Ein **Objekt** besteht aus*

*1. einem **Speicherbereich** d.h. einer Folge von Bytes mit aufeinanderfolgenden Adressen*

*2. dem **Typ** (oder Datentyp), der angibt, welche Art von Datum in dem Speicherbereich abgelegt werden darf*

*3. dem **Wert** des Datums, das in dem Speicherbereich abgelegt ist.*

Während der Ausführung eines Programms kann es Phasen geben, während denen der Wert eines Objekts nicht definiert ist, z.B. nach einer Definition ohne Initialisierung wie in Beispiel 1.1.2 nach Ausführung der Definition `int i;` und vor der Initialisierung `i=3;`.

1.3.2 Definition *Eine **Variable** ist ein Objekt, das mit einem Namen bezeichnet ist, wobei der Name ein Bezeichner (identifier) sein muß; das ist ein Wort, das folgenden Bedingungen genügt:*
a) Das Alphabet besteht aus den Buchstaben (ohne Umlaute), den Ziffern und dem Unterstrich _ .

b) Das erste Zeichen ist keine Ziffer.

c) Das Wort ist nicht eines der C-Schlüsselwörter.

1.3.3 Bemerkungen

1) Laut ANSI-Norm müssen nur die ersten 31 Zeichen bei der Unterscheidung von Bezeichnern berücksichtigt werden.

2) In der ANSI-Norm kommt das Wort Variable nicht vor, nur Objekte mit Bezeichnern. Es ist bei höheren Programmiersprachen aber allgemein üblich, den Begriff Variable zu verwenden.

3) Wie wir später noch sehen werden, kann man Objekte auch durch Wörter bezeichnen, die nicht identifier im obigen Sinne sind. Jede Bezeichnung für ein Objekt wird von Kernighan & Ritchie ein **lvalue** genannt.

1.3.4 Deklaration und Definition von Variablen

In der Deklaration einer Variablen werden ihr Name und ihr Typ vereinbart. Die einfachste Form einer Variablendeklaration ist folgende:

Typ Variablenname;

also beispielsweise

```
int x;
```

um `x` als Variable vom Typ Integer zu deklarieren. Man kann mehrere Variablen vom gleichen Typ auf einmal deklarieren, indem man ihre Namen durch Kommas getrennt hintereinander schreibt, z.B.

```
int x,y;
```

Achtung! Die Deklaration muß mit einem Semikolon enden.

Durch eine Deklaration wird einer Variablen nicht notwendig ein Speicherplatz reserviert; falls dies der Fall ist, spricht man von einer *Definition* der Variablen. Meist sind Deklarationen auch Definitionen, zumindest für die folgenden arithmetischen Datentypen. Die gesamte Beschreibung der Begriffe Deklaration und Definition im ANSI-Standard umfaßt viele Regeln und ist etwas unübersichtlich.

Durch eine Definition wird einer Variablen nicht unbedingt ein Wert zugewiesen. Die Zuweisung eines Anfangswertes heißt *Initialisierung*. Sie kann in manchen Fällen mit der Definition verbunden werden, z.B.

```
int x=7;
```

1.3.5 Type Qualifiers

Bei der Deklaration einer Variablen kann eines der Schlüsselwörter `const, volatile, extern, static, auto, register` vor den Typ geschrieben werden. Damit werden dem Compiler gewisse Hinweise gegeben.

`const` bedeutet, daß der Wert dieser Variablen nicht durch Anweisungen des Programms geändert werden kann. Falls man dies doch tut, gibt der Compiler eine Warnung aus. `volatile` weist den Compiler darauf hin, daß der Wert dieser Variablen unvorhergesehen durch äußere Ereignisse sich ändern kann (z.B. der Inhalt von Registern in Peripheribausteinen wie Uhren oder Schnittstellen) und daß er deshalb auf manche Optimierungen verzichten muß, um sicherzustellen, daß immer der neueste Wert verwendet wird.

`extern` weist den Compiler darauf hin, daß die Definition der Variablen nicht in dieser Quelldatei erfolgt, sondern in einer Datei, die später vom Linker daran gebunden wird. Eine Deklaration mit dem Qualifier `extern` ist nie eine Definition.

`static` hat unterschiedliche Bedeutungen in Abhängigkeit von der Stelle, wo die Deklaration steht. Der wichtigste Fall ist der, daß eine Variable außerhalb jeder Funktion mit dem Zusatz `static` definiert wird (sog. modulglobale Variablen). Dann kann diese Variable an jeder Stelle innerhalb derselben Quelldatei verwendet werden; man kann auf sie aber nicht aus anderen Quelldateien heraus zugreifen, auch wenn dort eine entsprechende Deklaration mit dem Zusatz `extern` steht. Die Variable ist eben nur innerhalb der Quelldatei, in der sie definiert ist, sichtbar. Auch wenn in anderen Quelldateien eine Variable desselben Namens definiert ist, wird der Linker die beiden als unterschiedliche Variablen behandeln. *Es empfiehlt sich, modulglobale Variablen stets als* `static` *zu deklarieren, sofern sie nicht in anderen Quelldateien verwendet werden.* Dadurch vermeidet man Kollisionen mit Variablen gleichen Namens in anderen

Programmmoduln oder Bibliotheken, die daran gebunden werden.

Die Qualifier `auto` und `register` sind nicht sehr wichtig; wir übergehen sie hier.

1.4 Arithmetische Datentypen

1.4.1 Definition *Ein **Datentyp** besteht aus*
1. einem Namen
2. einer Menge von Werten
3. einer Menge von Operationen (Abbildungen).

Die arithmetischen Datentypen von C dienen zur Darstellung gewisser Bereiche ganzer oder rationaler Zahlen. Es ist $2^{32} = 4\,294\,967\,296$.

1.4.2 Ganzzahltypen

Name	üblicher Speicherplatz (keine ANSI-Vorschrift)	minimaler Wertebereich nach ANSI
`unsigned char`	1 Byte	$\{0, \ldots, 255\}$
`signed char`	1 Byte	$\{-127, \ldots, 127\}$
`char`	1 Byte	$\{0, \ldots, 255\}$ oder $\{-127, \ldots, 127\}$
`int`	2 oder 4 Byte	$\{-2^{15}+1, \ldots, 2^{15}-1\}$
`unsigned int`	2 oder 4 Byte	$\{0, \ldots, 2^{16}-1\}$
`short`	2 Byte	$\{-2^{15}+1, \ldots, 2^{15}-1\}$
`unsigned short`	2 Byte	$\{0, \ldots, 2^{16}-1\}$
`long`	4 Byte	$\{-2^{31}+1, \ldots, 2^{31}-1\}$
`unsigned long`	4 Byte	$\{0, \ldots, 2^{32}-1\}$

Häufig findet man folgende Implementierungen:

`signed char`	1 Byte	$\{-128, \ldots, 127\}$
`int`	4 Byte	$\{-2^{31}, \ldots, 2^{31}-1\}$
`unsigned int`	4 Byte	$\{0, \ldots, 2^{32}-1\}$
`long`	4 Byte	$\{-2^{31}, \ldots, 2^{31}-1\}$

Der negative Wertebereich ist also um eine Zahl größer als der positive; das ist aber keine ANSI-Norm! Die implementierungsabhängigen Grenzen der Wertebereiche sind in der Datei `limits.h` festgelegt. Um maschinenunabhängig programmieren zu können, gibt es den `sizeof`-Operator, der die Länge des belegten Speicherbereichs ausgibt, z.B. ist `sizeof(int)` auf heutigen Rechnern meist gleich 4.

Die Menge der Operatoren der Ganzzahltypen besteht aus

- den binären Operatoren `+` (Addition), `-` (Subtraktion), `*` (Multiplikation), `/` (Division), `%` (Modulodivision). Dabei handelt es sich um die üblichen Operationen in $\mathbb{Z}$, deren Resultat jedoch modulo 2^n gerechnet wird, um wieder in dem Wertebereich des jeweiligen Typs zu landen; dabei ist $n = 8$ für die `char`-Typen, $n = 16$ für die `short`-typen und $n = 32$ für die `long`-Typen.
- den unitären Operatoren `-` (Vorzeichen), `++` (Inkrement), `--` (Dekrement)

- den Bit-Operationen `&` (AND), `|` (OR), `^` (EXOR), `~` (NEG), `<<` und `>>` (Shift).

Wegen der Bedeutung verweisen wir auf die einschlägige Literatur.

1.4.3 Gleitpunkttypen : `float, double, long double`

In einer Variablen, deren Typ ein Gleitpunkttyp ist, können Zahlen in Gleitpunktdarstellung abgespeichert werden. Die grobe Idee ist folgende: Verwende Gleitpunktdarstellungen $\pm\mu \cdot g^{\alpha}$ und stelle die Mantisse und den Exponenten sowie das Vorzeichen mit einigen Bytes dar. (Genaueres im nächsten Kapitel.)

Vorsicht! Was in Büchern dazu gesagt wird, stimmt oft nicht mit der ANSI-Norm überein. Gemäß ANSI-Norm müssen in der Datei `float.h` etliche implementierungsabhängige Parameter angegeben sein, und sie müssen gewissen Mindestanforderungen genügen, z.B.

	`float`	`double`	`long double`
kleinster Exponent bez. Basis 10	≤ -37	≤ -37	≤ -37
größter Exponent bez. Basis 10	$\geq +37$	$\geq +37$	$\geq +37$
mindestens q Dezimalstellen genau	$q \geq 6$	$q \geq 10$	$q \geq 10$

Eine die ANSI/IEEE-Norm für double-precision normalized numbers erfüllende Implementierung kann mit $g = 2$ und 8 Bytes pro `double` erreicht werden; die Mantissenlänge entspricht dabei 15 Dezimalstellen und der Exponentenbereich (bez. der Basis 10) dem Intervall $[-307, 308]$.

Als Operatoren für Gleitpunkttypen stehen `+,-,*,/,++,--` zur Verfügung.

1.5 Eingabe von Zahlen mit der Funktion `scanf`

Einem C-Programm ist während seiner Ausführung ein Standardeingabekanal `stdin` zugeordnet, aus dem es Daten einlesen kann. Normalerweise ist `stdin` die Tastatur; man kann aber beim Aufruf des Programm die Standardeingabe auch umlenken, z.B. zu einer Datei, indem man auf der Kommandozeile des Terminals *programmname* `<` *dateiname* eintippt.

Die Funktion `scanf` liest Zeichenreihen aus der Standardeingabe und konvertiert sie in Werte vorgegebenen Typs.

Syntax: `scanf(`*formatstring, argumentliste*`);`

formatstring ist im einfachsten Fall eine Zeichenkette, die mit `"` anfängt und endet und beschreibt, wie die Wörter in der Standardeingabe interpretiert werden sollen. Beispiel: Der Formatstring `"%d %d %lf"` bedeutet, daß die ersten drei Wörter in der Standardeingabe der Reihe nach als Dezimaldarstellungen zweier ganzer Zahlen und einer Gleitpunktzahl interpretiert werden sollen. Dabei sind Wörter Zeichenreihen, die keine Zwischenraumzeichen (white space characters) enthalten, aber durch solche getrennt sind.

argumentliste enthält die Adressen von Objekten, in denen die eingelesenen Werte gespeichert werden sollen. Reihenfolge und Typ müssen mit den Angaben im Formatstring übereinstimmen. Die Adresse eines Objekts kann man dadurch angeben, daß man vor einen Objektbezeichner das Zeichen `&` setzt; im Falle einer Variablen namens

`var` bezeichnet also `&var` ihre Adresse.

Beispiel:

```
int a,b;
double x;
scanf("%d %d %lf",&a,&b,&x);
```

Nach Eingabe von `3 4 1.5` hat `a` den Wert 3, `b` den Wert 4 und `x` den Wert $1,5$. Achtung! Die Eingabe von der Tastatur wird meist zwischengespeichert, bis ein RETURN eingegeben wird.

Beim Einlesen von Daten des Typs `char` gibt es (mindestens) zwei Möglichkeiten:

```
char c;
scanf("%c",&c);
scanf(" %c",&c);
```

Die erste `scanf`-Zeile liest das nächste Zeichen aus der Standardeingabe, egal was es ist, auch Leerzeichen. Die zweite überliest Leerzeichen und weist `c` das nächste Non-White-Space-Zeichen als Wert zu.

Bezüglich weiterer Einzelheiten verweisen wir auf die Manualseiten von `scanf`.

Die Funktion `scanf` hat einen **Rückgabewert** vom Typ `int`. Er gibt an, wieviele Eingabefelder eingelesen wurden. Ist er kleiner als die im Formatstring angegebene Anzahl, ist ein Fehler beim Einlesen der Daten aufgetreten, z.B wenn in einer Dezimaldarstellung fälschlicherweise ein Buchstabe stand. Der Rückgabewert kann auch den speziellen Wert `EOF` haben, der in der Headerdatei `stdio.h` definiert ist; er gibt an, daß `scanf` bereits vor dem Einlesen eines Datums auf eine `End-of-File`-Markierung gestoßen ist. Dies ist der Fall, wenn in der Standardeingabe `Ctrl-D` (unter Unix, `Ctrl-Z` unter DOS) eingegeben wird oder wenn beim Umlenken der Standardeingabe in eine Datei tatsächlich das Dateiende erreicht ist.

1.6 Ausdrücke

Ausdrücke (expressions) sind Zeichenketten, deren Aufbau rekursiv definiert wird. Variablen und allgemeine Objektbezeichner sind Ausdrücke, ebenso Zahlenkonstanten und Funktionsaufrufe. Davon ausgehend wird rekursiv definiert, durch welche Operatorzeichen und Klammern man Ausdrücke verbinden darf, um wiederum Ausdrücke zu erhalten. Die formalen Regeln in der ANSI-C-Normierung sind für den Anfänger etwas unübersichtlich. Wir geben hier deshalb eine informelle, aber unvollständige Übersicht darüber, wie in C Ausdrücke (expressions) gebildet werden können.

Jedem Ausdruck wird ein Datentyp und ein Wert zugeordnet. Dies geschieht natürlich ebenfalls rekursiv über den Aufbau des Ausdrucks. Diese rekursive Berechnung nennt man die Auswertung des Ausdrucks. Meist betrachtet man Ausdrücke mit arithmetischen Typen; es gibt aber auch andere, z.B. haben Aufrufe von Funktionen ohne Rückgabewert den Typ `void`; solchen Funktionsausdrücken wird dann kein Wert zugeordnet. Auch in manch anderen Fällen ist man gar nicht so sehr an dem Wert des Ausdrucks, sondern eher an Nebeneffekten (side effects), die beim Auswerten auftreten, interessiert. Man denke an die Funktion `scanf`, die bei Aufruf Daten einliest. Ein

typisches Anwendungsbeispiel werden wir weiter unten angeben.

1.6.1 Arithmetische Ausdrücke werden wie in der Mathematik üblich gebildet; dabei können Variablen und Zahlenkonstanten mit arithmetischen Operatoren verbunden und durch runde Klammern hierarchisch geordnet werden. C benützt ähnliche Präzedenzregeln wie in der Mathematik üblich, also binden z.B. der Multiplikationsoperator `*` und der Divisionsoperator `/` stärker als `+` und `-`. Im Zweifelsfall verwende man Klammern.

Jedem arithmetischen Ausdruck wird ein Typ und ein Zahlenwert zugeordnet, der wie in der Mathematik üblich berechnet wird. Achtung! Kommen in einem arithmetischen Ausdruck Objekte mit unterschiedlichen arithmetischen Typen vor, werden bei der Auswertung des Ausdrucks die Werte der Teilausdrücke, wenn nötig, automatisch umgerechnet in Werte eines anderen Typs. Dabei kann es zu Ungenauigkeiten kommen, wenn der Wertebereich des urspünglichen Typs nicht im Wertebereich des neuen Typs enthalten ist; so sind z.B. große `long`-Werte nicht immer exakt als `float` darstellbar. Genaueres dazu später (Typkonversion). Zahlenkonstanten in Gleitpunktform werden als Werte vom Typ `double` interpretiert, ansonst als Werte eines `int`-Typs.

Beispiele: `x + 2*x` hat den dreifachen Wert der Variablen `x`. Sind `x` eine Variable vom Typ `double` und `n` eine Variable vom Typ `int`, so hat der Ausdruck `x + n` den Typ `double`.

1.6.2 Zuweisungsausdrücke (assignement expressions) haben die Gestalt

objektbezeichner `=` *expression*

Häufig ist *objektbezeichner* der Name einer Variablen.
Auswertung: Der Datentyp ist der des bezeichneten Objektes, und der Wert ist der des rechts stehenden Ausdrucks nach einer eventuellen Umwandlung in den Typ des linksstehenden Objekts. Überdies ersetzt dieser Wert auch den bisherige Wert des bezeichneten Objekts. Dies nennt man einen Nebeneffekt (side effect), der aber meist der Hauptzweck eines Zuweisungsausdrucks ist.

Beispiel: Der Ausdruck `y=y+1` sieht zwar wie eine mathematische Gleichung aus, die durch keine reelle Zahl `y` erfüllbar ist; seine Auswertung hat jedoch einen sehr sinnvollen Nebeneffekt. Denn zunächst wird `y+1` ausgewertet, und dann wird der Wert von `y` durch den Wert des Ausdrucks `y+1` ersetzt. Anschließend hat `y` also einen Wert, der um 1 höher als der bisherige ist.

Weil solche Ausdrücke häufig auftreten, gibt es abkürzende Schreibweisen:
`y+=`*expression* statt `y=y+`*expression* und entsprechend für andere binäre Operatoren wie `-,*,/,&,|,^`.

1.6.3 Gleichheitsausdrücke (equality expressions) haben die Gestalt

expression1 `==` *expression2*

oder

expression1 `!=` *expression2*

Die erste Form testet die beiden Operanden *expression1* und *expression2* auf Gleichheit, die zweite auf Ungleichheit.
Auswertung: Der Wert ist vom Typ `int`, und zwar bekommt der Ausdruck den Wert 1, wenn die Werte der Operanden in der angebenen Relation stehen, und 0 sonst.

Vorsicht! Es ist ein häufiger Fehler, `=` statt `==` zu schreiben! Da dies kein Syntaxfehler ist, kann es der Compiler nicht als Fehler erkennen (höchstens vermuten und warnen). Die Konsequenzen dieses Fehlers wirken oft verwirrend, z.B. weil einige Programmteile nie ausgeführt werden. Beachte, daß der Zuweisungsausdruck `x=7` den Wert 7 hat, während der Gleichheitsausdruck `x==7` den Wert 0 oder 1 hat. Häufig gibt es einen einfachen Trick, diese Art von Fehler zu vermeiden: Soll ein Ausdruck auf Gleichheit mit einer Zahlenkonstante getestet werden, so schreibe man die Zahlenkonstante auf die linke Seite. Vergißt man nun eines der beiden Gleichheitszeichen, so wird der Compiler einen Fehler melden, weil man einer Zahlenkonstante keinen Wert zuweisen kann. So ist z.B. `7==x` ein korrekter Gleichheitsausdruck, dagegen ist `7=x` ein Syntaxfehler.

1.6.4 Ungleichungsausdrücke (relational expressions) haben die Gestalt

expression1 vergleichsoperator expression2

Dabei ist *vergleichsoperator* eines der Wörter `<`, `>`, `<=`, `>=`, mit denen die mathematischen Vergleichsoperatoren $<, >, \leq, \geq$ bezeichnet werden.
Auswertung: Der Wert ist vom Typ `int`; er ist 1, wenn die Werte der Operanden *expression1* und *expression2* in der entsprechenden Relation stehen, und 0 ansonsten.

1.6.5 Komma-separierte Ausdrücke haben die Gestalt

`(`*expression_1*`, ... ,`*expression_k*`)`

Auswertung: Die aufgelisteten Ausdrücke werden nacheinander in der angegebenen Reihenfolge ausgewertet. Der Wert und Datentyp des letzten Ausdrucks ist zugleich der Wert und Datentyp des gesamten Ausdrucks.

Für $k = 1$, erhält man die Aussage, daß man wieder einen Ausdruck erhält, wenn man um einen Ausdruck runde Klammern setzt.

1.7 Typkonversion

Bei der Auswertung von Ausdrücken kann es vorkommen, daß Teilausdrücke unterschiedlichen Typs verknüpft werden sollen. Dabei ist es notwendig, den Wertebereich eines Typs - wenn möglich - in den eines anderen Typs abzubilden. Dies wird bei arithmetischen Ausdrücken in C weitgehend automatisch vorgenommen und zwar meistens so, wie man es auch selbst vornehmen würde. Es gibt aber Fälle, wo dies nicht zum erwünschten Ergebnis führt.

Beispiel:
Will man die Häufigkeit bestimmen, mit der bei n-fachem Würfeln eine 3 vorkommt, so ist es naheliegend, das folgende Programmfragment zu verwenden.

```
int Spiele, Anzahl3;  /* Spiele gibt an, wie oft gewuerfelt wurde */
                      /* Anzahl3 gibt an, wie oft eine 3 vorkam */
double Haeufigkeit;
....                  /* hier werden die Werte von Spiele und Anzahl3 */
....                  /* eingegeben */
....
Haeufigkeit = Anzahl3/Spiele;    /* Vorsicht! Das geht schief! */
```

Der Zuweisungsaudruck wird folgendermaßen ausgewertet: Zunächst werden `Spiele`

und `Anzahl3` ausgewertet. Weil beide vom Typ `int` sind, wird bei der folgenden Auswertung von `Anzahl3/Spiele` die Division ganzzahlig ausgeführt; wenn der Würfel einigermaßen fair ist, wird nicht allen bei Spielen die Drei gewürfelt worden sein, und dann bekommt `Anzahl3/Spiele` den Wert 0. Bei der Auswertung des Zuweisungsausdrucks `Haeufigkeit = Anzahl3/Spiele` wird dann der `int`-Wert 0 in den `double`-Wert 0 umgewandelt. Das ändert aber nichts daran, daß dies nicht das gewünschte Ergebnis ist (es erklärt aber, warum sich schon mancher Anfänger gewundert hat, daß die von ihm berechneten Histogramme alle konstant 0 sind).

Wie macht man es richtig? Man muß erzwingen, daß bereits vor Ausführung der Division eine Umwandlung in den Typ `double` erfolgt. Dazu bedient man sich des sogenannten **Cast-Operators**. Er hat die Gestalt `(T)`, wobei `T` der Typ ist, in den umgewandelt werden soll. Er wird einfach vor den Ausdruck gestellt, dessen Typ gewandelt werden soll. In obigem Beispiel kann man z.B. den Zähler wandeln:

```
Haeufigkeit = (double)Anzahl3/Spiele;
```

Dies hat zur Folge, daß `Anzahl` auch zu `double` gewandelt wird und eine Division von zwei `double`-Werten ausgeführt wird.

Ähnlich ist es bei Ausdrücken der Gestalt *expression*`/2.0`, wobei *expression* ein Ausdruck von Ganzzahltyp ist. Dadurch daß der Nenner eine Konstante in Gleitpunktdarstellung ist, der der Typ `double` zugeordnet ist, wird der Zähler auch nach `double` gewandelt und erst dann die Division ausgeführt. Der gesamte Ausdruck hat dann den Typ `double`. Dagegen hat der Ausdruck *expression*`/2` denselben Ganzzahltyp wie *expression*, und die Division wird ganzzahlig durchgeführt.

1.8 Anweisungen

Anweisungen (statements) sind in C gewisse Zeichenketten, die alle mit einem Semikolon enden. Sie stellen die elementaren Verarbeitungsschritte dar. Ausdrücke sind keine Anweisungen. Allerdings kann man durch Anfügen eines Semikolons daraus Anweisungen gewinnen.

1.8.1 Ausdrucksanweisungen (expression statements) haben die Gestalt

expression`;`

Bedeutung: Der Ausdruck *expression* wird ausgewertet.
Der Sinn einer Ausdruckssanweisung besteht nur darin, daß die Auswertung des Ausdrucks *expression* durchgeführt wird und dabei Nebeneffekte auftreten, an denen man interessiert ist. Der bei weitem wichtigste Fall ist der, daß der Ausdruck ein Zuweisungsausdruck ist.

1.8.2 Zuweisungsanweisungen (assignement statements) haben die Gestalt

objektbezeichner `=` *expression*`;`

Bedeutung: Zuerst wird der Ausdruck *expression* ausgewertet, dann wird dem bezeichneten Objekt der Wert des rechts stehenden Ausdrucks zugeordnet; dabei wird eine Umwandlung von dem Typ von *expression* in den Typ des links stehenden Objekts durchgeführt.

Beispiel für Zuweisungsanweisungen:

```
int x, y=2;
double a,b;
x=(y+1)/2;
a=(y+1)/2;
b=(y+1.0)/2;
printf("x=%d  a=%f  b=%f\n",x,a,b);
```

Dadurch bekommt `x` den Wert 1; denn `y+1` hat den Wert 3, und die anschließende Division wird ganzzahlig durchgeführt. Ebenso hat `a` den Wert 1, weil die Umwandlung in den Typ `double` erst nach der Auswertung der rechten Seite erfolgt. Dagegen bekommt `b` den Wert 1.5; denn wegen der Zahlenkonstante `1.0` in Gleitpunktform wird der Wert von `y` bereits vor der Addition in einen `double`-Zahlenwert umgerechnet.

1.8.3 Anweisungsblöcke Mehrere Anweisungen können zu einem Anweisungsblock (compound statement) zusammengefaßt werden, indem man sie nacheinander auflistet und diese Liste mit geschweiften Klammern { und } einschließt. Am Beginn eines Blockes (gleich hinter der Klammer {) darf man Definitionen von Variablen einfügen, die dann nur innerhalb des Blockes gelten und außerhalb nicht sichtbar sind.

1.8.4 Wiederholungsanweisungen (iteration statements)
1.8.4.1 `while`-Schleifen haben die Gestalt

`while (`*expression*`)` *anweisungsblock*

Bedeutung: Der Ausdruck *expression* wird ausgewertet; ist sein Wert gleich 0, so fährt die Programmausführung mit den Anweisungen, die nach dem Block *anweisungsblock* kommen, fort. Ist sein Wert ungleich 0, so werden die Anweisungen in *anweisungsblock* ausgeführt; anschließend wird der Ausdruck *expression* erneut ausgewertet und wie oben verfahren. Der Block *anweisungsblock* wird also so lange ausgeführt, solange *expression* einen Wert ungleich 0 hat.

Beispiel:

```
int i=1;
int summe=0;
while (i<=100) {
   summe=summe+i;
   i=i+1;
}
printf("Die Summe der Zahlen 1 bis 100 ist %d\n",summe);
```

Das folgende Beispiel benutzt auf sehr typische Weise einen Nebeneffekt bei der Auswertung der Abbruchbedingung:

```
int x;
int summe=0;
while (EOF!=scanf("%d",&x))  summe+=x;
printf("Die Summe der eingegebenen Zahlen ist %d\n",summe);
```

Bei der Auswertung der Abbruchbedingung `EOF!=scanf("%d",&x)` wird die Funktion `scanf("%d",&x)` aufgerufen und dadurch ein weiteres Datum eingelesen und

der Variablen `x` zugewiesen.

Beispiel einer Endlosschleife:
`while(1)` *anweisungsblock*

1.8.4.2 `do-while`-Schleifen haben die Gestalt

`do` *anweisungsblock* `while (`*expression*`);`

Bedeutung: Die Anweisungen im Block *anweisungsblock* werden ausgeführt, dann wird *expression* ausgewertet; ist sein Wert gleich 0, so wird mit der Ausführung des weiteren Programms fortgefahren, ansonsten wird erneut *anweisungsblock* ausgeführt, anschließend wiederum *expression* ausgewertet und wie oben verfahren. Im Gegensatz zur `while`-Schleife wird *anweisungsblock* immer mindestens einmal ausgeführt.

Beispiel:

```
int i=0;
int summe=0;
do {
   summe=summe+i;
   i=i+1;
} while (i<=100);
printf("Die Summe der Zahlen 1 bis 100 ist %d\n", summe);
```

1.8.4.3 `for`-Schleifen haben die Gestalt

`for(`*initialisierung*`;`*bedingung*`;`*inkrement*`)` *anweisungsblock*

initialisierung, *inkrement* und *bedingung* sind Ausdrücke.
Bedeutung: Die obige `for`-Schleife bewirkt dasselbe wie die folgende Folge von Anweisungen:

initialisierung`;`
`while (`*bedingung*`) {`
    anweisungsblock
    inkrement`;`
`}`

Ein typisches Beispiel:

```
int i;
int summe=0;
for(i=1;i<=100;i++) summe=summe+i;
printf("Die Summe der Zahlen 1 bis 100 ist %d\n",summe);
```

1.8.5 Auswahlanweisungen (selection statements)

1.8.5.1 Die `if`-Anweisung hat die Gestalt

`if (`*expression*`)` *anweisungsblock*

oder

`if(`*expression*`))` *anweisungsblock* `else` *anweisungsblock2*

Bedeutung: Der Ausdruck *expression* wird ausgewertet; ist sein Wert ungleich 0, so wird *anweisungsblock*, aber nicht *anweisungsblock2* ausgeführt; ist er gleich 0, so wird *anweisungsblock2*, aber nicht *anweisungsblock* ausgeführt.

1.8.5.2 Die `switch`-Anweisung dient dazu eine Wahl zwischen mehreren Möglichkeiten zu treffen, ohne dafür eine Folge von `if`-Anweisungen hinschreiben zu müssen. Sie hat typischerweise die Gestalt

```
switch(expression) {
      case constant-expression_1:
                                    anweisungen_1
                                    break;
      case constant-expression_2:
                                    anweisungen_2
                                    break;
      .
      .
      .
      case constant-expression_k:
                                    anweisungen_k
                                    break;
      default:
                                    anweisungen
}
```

Der `default`-Teil darf auch fehlen. Die `case`-Marken *constant-expression_j* sind Ausdrücke, die keine Variablen oder Objektbezeichner enthalten, so daß ihr Wert schon beim Compilieren bestimmt werden kann.

Bedeutung: Der Wert von *expression* wird der Reihe nach mit den Werten der Ausdrücke in den `case`-Marken verglichen. Bei der ersten Übereinstimmung werden die folgenden Anweisungen bis zum nächsten `break` ausgeführt; dann wird die Ausführung der `switch`-Anweisung beendet. Gibt es keine Übereinstimmung mit dem Wert einer der `case`-Marken, so wird mit der Ausführung der Anweisungen nach der `default`-Marke fortgefahren. Gibt es keinen `default`-Teil, so wird stattdessen die Ausführung der `switch`-Anweisung beendet.

Bemerkung: Meist wird man jeden Einzelfall mit `break` beenden; es gibt aber sinnvolle Ausnahmen, z.B. wenn für mehrere Werte von *expression* dieselbe Aktion ausgeführt werden soll.

Beispiel: Innerhalb eines Programms, das einen Taschenrechner simuliert, könnte folgendes Programmsegment vorkommen. Man beachte, wie `char`-Konstanten angegeben werden können.

```
char operator;
int a,b,ergebnis;

switch(operator) {
     case '+':
             ergebnis=a+b;
             break;
     case '*':
             ergebnis=a*b;
             break;
     case '-':
```

```
            ergebnis=a-b;
            break;
    case '/':
            ergebnis=a/b;
            break;
    default:
            printf("Den eingegebenen Operator gibt es nicht!\n");
}
```

1.9 Typdefinitionen

Eine **Typdefinition** hat die Gestalt

`typedef T` *identifier*`;`

wobei `T` ein Datentyp und *identifier* ein Bezeichner sind.

Bedeutung: *identifier* ist ein Name für den Typ `T`, der völlig äquivalent zu `T` verwendet werden kann. Durch `typedef` wird kein neuer Typ geschaffen!

Typische Anwendungen:

- Suggestive Bezeichnung für komplizierte Typen
- Manchmal haben die Parameter und Rückgabewerte von Funktionen in Bibliotheken Typen, die von der Rechnerplattform abhängen. Dann steht in der Funktionsdeklaration als Typ eine Name, der mit `typedef` in einer Headerdatei definiert wird; z.B. wird der Name `size_t` in `stdlib.h` definiert. Weitere Beispiele findet man in `time.h`.

Beispiel:

```
typedef unsigned char PIXEL;
PIXEL grauwert;
```

1.9.1 Aufzählungstypen

Aufzählungstypen haben als Wertebereich eine Teilmenge des Wertebereichs eines `int`-Typs; welcher verwendet wird, ist von der ANSI-Norm nicht festgelegt. Jedes Element des Wertebereichs hat einen oder mehrere Bezeichner zugeordnet, die z.B. bei Zuweisungen oder Vergleichen benützt werden können.

Typischerweise hat die Deklaration eines Aufzählunstyps die Gestalt

`enum` *etikett* {*liste*}`;`

Dabei ist *etikett* ein Bezeichner, der das Etikett (tag) des Aufzählungstyps genannt wird, und *liste* eine durch Kommata separierte Aufzählung von Bezeichnern oder Zuweisungsausdrücken der Form *identifier*`=`*constant_expression*.

Bedeutung: Den in *liste* angegebenen Bezeichnern werden der Reihenfolge nach die Wert `0,1,2,...` zugeordnet, außer wenn explizit durch einen Zuweisungsungsausdruck *identifier*`=`*constant_expression* eine andere Zuordnung angegeben ist.

Beispiele:

```
enum farbe { rot, gelb, gruen, blau };
enum farbe { blau=3, gelb=1, gruen=2, rot=0 };
enum schalter { aus=0, ein=1, off=0, on=1 };
```

Sowohl durch die erste als auch durch die zweite Zeile werden den Namen `rot, gelb, gruen, blau` der Reihe nach die Werte `0,1,2,3` zugeordnet. Die letzte Zeile zeigt, daß mehrere Namen denselben Wert zugeordnet bekommen können (aber nicht umgekehrt).

`enum` *etikett* ist die Typbezeichnung des Aufzählungstyps, die z.B. in der Deklaration von Variablen verwendet werden kann:

`enum` *etikett variablenname*`;`

Beispiel: `enum farbe hintergrund;`

Achtung! Weder beim Compilieren noch zur Laufzeit wird geprüft, ob einer Variablen vom Aufzählungstyp tatsächlich nur Werte aus dem deklarierten Wertebereich zugewiesen werden. Man kann mit einem Objekt vom Aufzählungstyp alles machen, was man mit Objekten vom Typ `int` machen darf. Es bleibt der Selbstdisziplin des Programmierers überlassen, konsequent als Werte nur die in der Deklaration vereinbarten Namen zu verwenden.

Ein Aufzählungstyp ist also kein wirklich eigener Typ!

Kapitel 2

Codierung und Compilation

Um Information maschinell verarbeiten zu können, muss sie so dargestellt werden, dass die Maschine damit etwas anfangen kann. Töne und Bilder werden z.B. in elektrische Spannungen umgewandelt, um sie verstärken und übertragen zu können. Will man statt analoger Elektronik digitale Elektronik einsetzen, so muss man natürlich auch entsprechend die zu verarbeitenden Daten anders repräsentieren, nämlich indem man sie durch Zeichenketten über einem endlichen Alphabet codiert.

2.1 Wörter

Ein *Alphabet* A ist einfach eine endliche, nicht leere Menge. Ihre Elemente werden Zeichen oder Symbole genannt. Ein *Wort* der Länge $n \in \mathbb{N}$ über einem Alphabet A ist anschaulich eine Folge $a_1 \dots a_n$ von n Zeichen aus A. Statt Wort sagt man auch *Zeichenkette* oder *String*. Mathematisch ist ein Wort eine Abbildung $f\colon \{1, \dots, n\} \to A$. In der Analysis schreibt man sie meist als n-Tupel $(a_1, \dots, a_n)$, wobei $a_j = f(j)$ für $j \in \{1, \dots, n\}$ ist. In der Theorie der formalen Sprachen stellt man sich die Elemente von A als irgendeine Art von Buchstaben vor und die Wörter über A als Zeichenreihen $a_1 \dots a_n$.

A^n sei die Menge der Wörter der Länge $n \in \mathbb{N}$ über A. Wir identifizieren A^1 mit A. Außerdem definieren wir noch ϵ als das leere Wort, das kein Zeichen enthält; formal ist es die (eindeutig bestimmte) Abbildung $f\colon \emptyset \to A$. Sei $A^0 := \{\epsilon\}$. Dann ist $A^* = \bigcup_{n \in \mathbb{N}_0} A^n$ die Menge aller Wörter über A und $A^+ = \bigcup_{n \in \mathbb{N}} A^n$ die Menge aller nichtleeren Wörter.

Als *Konkatenation* oder *Verkettung* bezeichnet man die Abbildung

$$A^* \times A^* \to A^*, (a, b) \mapsto ab,$$

wobei ab dasjenige Wort ist, das entsteht, wenn man die beiden Wörter hintereinanderschreibt; sind $a = \alpha_1 \dots \alpha_n$ und $b = \beta_1 \dots \beta_m$, so ist $ab = \alpha_1 \dots \alpha_n \beta_1 \dots \beta_m$.

2.2 Codierungen

2.2.1 Definition *M und N seien Mengen.*

a) *Eine **Codierung** φ von M **in** N ist eine injektive Abbildung $\varphi\colon M \to N$.*

b) *Eine **Codierung** φ von M **über** einem Alphabet A ist eine injektive Abbildung $\varphi\colon M \to A^*$.*

c) *Eine **binäre Codierung** ist eine Codierung über dem Alphabet $\{0,1\}$.*

d) *Die Werte $\varphi(m), m \in M$, einer Codierung $\varphi\colon M \to N$ heißen ihre **Codewörter**.*

2.2.2 Beispiel ASCII = American Standard Code of Information Interchange
Es handelt sich um eine bijektive Abbildung $\varphi\colon Z \to \{\texttt{0}, \texttt{1}, \ldots, \texttt{127}\}$, wobei die Menge Z aus den kleinen und großen Buchstaben, den Ziffern `0` bis `9`, einigen Sonderzeichen und einige Steuerzeichen besteht. Man nennt Z die **Menge der ASCII-Zeichen**. Vorsicht! Es gibt länderspezifische Modifikationen und Erweiterungen auf mehr als 127 Zeichen (Unicode, UTF-8-Codierung).
Einige ASCII-Zeichen und ihre Codewörter:

`0` $\mapsto$ 48, `1` $\mapsto$ 49, ..., `9` $\mapsto$ 57
`A` $\mapsto$ 65, `B` $\mapsto$ 66, ..., `Z` $\mapsto$ 90
`a` $\mapsto$ 97, `b` $\mapsto$ 98, ..., `z` $\mapsto$ 122
CR (Wagenrücklauf, carridge return) $\mapsto$ 13
LF (Zeilenvorschub, line feed) $\mapsto$ 10

2.2.3 Beispiel Morse-Code
Die Menge der Buchstaben, Ziffern und Satzzeichen wird über dem ternären Alphabet $\{Punkt, Strich, Pause\}$ codiert. Für die akustische Übermittlung ersetzt man einen Punkt durch einen Ton einer gewissen Länge und einen Strich durch denselben Ton, allerdings etwa dreimal so lang. Die Codewörter nennt man Morsezeichen. Sie sind nicht alle gleich lang; vielmehr werden häufigen Buchstaben kurze Morsezeichen zugeordnet und selteneren längere. So wird z.B. `e` durch einen einzigen Punkt . und `q` durch `--.-` codiert.

Codierungen werden für sehr unterschiedliche Zielsetzungen eingesetzt; dementsprechend werden sie auch völlig verschieden konstruiert.

2.2.4 Anwendungsbezogene Repräsentation von Daten
z.B. ASCII-Zeichen durch Zahlen, Zahlen durch Ziffernfolgen, die Ziffern `0` und `1` durch zwei Spannungswerte oder durch zwei Töne oder durch zwei unterschiedliche Phasenlagen eines Tones. In vielen Fällen handelt es sich um simple Bijektionen eines Alphabets auf ein anderes. Um Daten auf Digitalrechnern verarbeiten zu können, muß man sie binär codieren.

2.2.4.1 Grundprinzip der digitalen Informationsverarbeitung: Stelle alle Daten als endliche Folge von Nullen und Einsen dar und verarbeite nur diese 0-1-Folgen.

2.2.4.2 Bemerkung Es lassen sich keineswegs alle Daten, die man in Anwendungen verarbeiten möchte, als endliche 0-1-Folgen darstellen. Oft will man Meßdaten

von Sensoren verarbeiten (Mikrofon, Kamera usw.), und diese Meßwerte sind Spannungswerte, die mit der Zeit variieren. Üblicherweise modelliert man Spannungs- und Zeitbereiche als Intervalle in $\mathbb{R}$; die Meßdaten sind also reelle Funktionen und lassen sich i.a. nicht durch endliche 0-1-Folgen darstellen. Man kann sie höchstens approximativ dadurch beschreiben. Es ist nicht trivial, daß die approximative Beschreibung so gut gewählt werden kann, daß sie für die jeweilige Anwendung ausreicht. Der Beweis solcher Aussagen ist mathematisch anspruchsvoll.

2.2.5 Datenkompression

Um Speicherplatz oder Übertragungszeit zu sparen, bemüht man sich, Daten in möglichst knapper Form zu codieren. Man unterscheidet verlustfreie Kompressionverfahren, bei denen die ursprünglichen Daten aus den codierten wieder vollständig rekonstruiert werden können, und verlustbehaftete, bei denen die Daten nur annähernd rekonstruiert werden können, aber die entstehenden Fehler so geartet sind, daß sie in der jeweiligen Anwendung keine große Rolle spielen.

2.2.6 Fehlerkorrektur

Bei der Speicherung oder Übertragung von Daten können Fehler auftreten. Um sie korrigieren zu können, verwendet man fehlerkorrigiernde Codierungen. Das sind Codierungen, die den ursprünglichen Daten zusätzliche Information auf geschickte Weise so hinzufügen, daß eine Korrektur von Fehlern mit hoher Wahrscheinlichkeit möglich ist.

2.2.7 Verschlüsselung

Um zu verhindern, daß Daten von Unbefugten gelesen werden, wendet man kryptografische Codes an, deren Decodierung ohne spezielles Vorwissen kaum möglich ist. Eine damit verwandte Variante sind steganografische Codierungen, mit denen man Daten in anderen (z.B. in Bildern oder Musik) verstecken kann.

2.3 Codierung von Zahlen

2.3.1 g-adische Entwicklung natürlicher Zahlen

Sei $g \in \mathbb{N}, g \geq 2$. Dann gibt es zu jedem $n \in \mathbb{N}$ eindeutig bestimmte Zahlen $k \in \mathbb{N}$ und $\alpha_0, \ldots, \alpha_k \in \{0, 1, \ldots, g-1\}$, so daß $n = \sum_{j=0}^{k} \alpha_j g^j$ und $\alpha_k \neq 0$.

Beweis: in den Mathematikvorlesungen.

2.3.2 g-adische Darstellung natürlicher Zahlen

A sei ein g-elementiges Alphabet, $g \geq 2$, und $\chi\colon A \to \{0, 1, , \ldots, g-1\}$ sei eine Bijektion. $\psi\colon A^* \to \mathbb{N}_0$ sei definiert durch $\psi(a_k \ldots a_0) = \sum_{j=0}^{k} \chi(a_j)\ g^j$. Nach 2.3.1 ist ψ surjektiv d.h. für jedes $n \in \mathbb{N}$ ist $\psi^{-1}(n) \neq \emptyset$. Jedes Wort $a \in \psi^{-1}(n)$ heißt eine g-adische Darstellung von n. Jede Zahl $n \in \mathbb{N}$ hat unendlich viele g-adische Darstellungen; man kann nämlich beliebig viele führende Nullen (d.h. $\chi^{-1}(0)$) hinzufügen. Nach 2.3.1 gibt es zu jedem $n \in \mathbb{N}$ aber nur eine g-adische Darstellung $a_k \ldots a_0$ mit $\chi(a_k) \neq 0$.

Welche Zahlen haben eine g-adische Darstellung der festen Länge $L \in \mathbb{N}$ (einschließlich führender Nullen)?

Für jedes Wort $a_{L-1} \dots a_0 \in A^L$ gilt $0 \leq \sum_{j=0}^{L-1} \chi(a_j) \cdot g^j \leq \sum_{j=0}^{L-1} (g-1) \cdot g^j = (g-1)\frac{g^L-1}{g-1} = g^L - 1$, also $\psi(A^L) \subset \{0, \dots, g^L - 1\}$.
Umgekehrt folgt aus 2.3.1, daß jede Zahl in $\{0, \dots, g^L - 1\}$ genau eine g-adische Darstellung der Länge L hat. Folglich ist die Einschränkung $\psi | A^L \to \{0, \dots, g^L - 1\}$ der Abbildung ψ auf A^L eine Bijektion.

2.3.3 Spezialfälle

a) Dezimaldarstellung:
$g = 10,\ A = \{\texttt{0}, \dots, \texttt{9}\},\ \chi(x) =$ die durch die Ziffer x bezeichnete Zahl.

b) Dual- oder Binärdarstellung:
$g = 2,\ A = \{\texttt{0,1}\},\ \chi(x) =$ die durch die Ziffer x bezeichnete Zahl. Dann ist $\psi | \{\texttt{0,1}\}^n \to \{0, \dots, 2^n - 1\}$ eine Bijektion, mit der Wörter der Länge n über $A = \{\texttt{0,1}\}$ mit den ganzen Zahlen zwischen 0 und $2^n - 1$ identifiziert werden. In der n-stelligen Dualdarstellung $a_{n-1} \dots a_0$ der Zahl $\sum_{j=0}^{n-1} a_j 2^j$ heißt a_{n-1} das *höchstwertige Bit (MSB = Most Significant Bit)* und a_0 das *niedrigstwertige Bit (LSB = Least Significant Bit).*

c) Hexadezimaldarstellung:
$g = 16,\ A = \{\texttt{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}\}$. Dabei nimmt man `A,B,C,D,E,F` als Symbole für die Zahlen $10, 11, 12, 13, 14, 15$; die Abbildung χ ist also dadurch definiert, daß sie die Symbole `0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F` der Reihen nach auf die Zahlen $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$ abbildet. Die Codierung ψ liefert eine Bijektion von A^2 auf $\{0, \dots, 255\}$; jede ganze Zahl zwischen 0 und 255 hat eine zweistellige Hexadezimaldarstellung.
Bemerkung: Das Wort Hexadezimal ist eine Mischung aus Griechisch und Latein; sprachlich sauberer wäre es, von der Hexagesimal- oder der Sedezimaldarstellung zu sprechen.

d) Oktaldarstellung:
$g = 8, A = \{\texttt{0,1,2,3,4,5,6,7}\}, \chi(x) =$ die durch die Ziffer x bezeichnete Zahl.

Zur Unterscheidung obiger Darstellungen werden oft gewisse Sonderzeichen vorangestellt, z.B. zur Kennzeichnung von Hexadezimalzahlen \$ oder in C die beiden Zeichen `0x`.

2.3.4 Darstellung nichtnegativer ganzer Zahlen im Rechner

Während in Programmtexten Dezimal-, Dual- und Hexdezimaldarstellungen vorkommen, werden im Objektcode intern nahezu immer Dualdarstellungen unterschiedlicher Länge n verwendet. Und zwar werden die Koeffizienten der Dualdarstellung der Länge n als eine 0-1-Folge der Länge n aufgefasst, die als Folge von n Bits abgepeichert werden. Dabei spielen vor allem die Fälle $n = 8, 16$ und 32 eine wichtige Rolle. In C entsprechen sie den Typen `unsigned char, unsigned short` und `unsigned long`, welche 1, 2 bzw. 4 Byte Speicherplatz belegen.
Vorsicht! In welcher Reihenfolge die Bits und Bytes im Speicher abgelegt werden, ist prozessorabhängig.

Die üblichen Rechenoperationen werden im Rechner immer modulo 2^n ausgeführt, d.h. von dem Ergebnis in $\mathbb{Z}$ wird nur der Rest bei Divison durch 2^n verwendet. Man

muss sich selbst überlegen, ob der Wertebereich eines Typs groß genug ist, um das Ergebnis einer Operation in $\mathbb{Z}$ unverfälscht darstellen zu können.

2.3.5 Zweierkomplementdarstellung negativer ganzer Zahlen

Am Einfachsten ist es, die Darstellung nichtnegativer, ganzer Zahlen durch ein zusätzliches Bit zu erweitern, welches das Vorzeichen angibt. Dies führt jedoch bei der Implementierung der Addition positiver und negativer Zahlen zu Fallunterscheidungen. Um sie zu vermeiden, wird eine andere Codierung gewählt. Für $x \in \mathbb{Z}$ gilt $x = (2^n + x)$ modulo 2^n, also insbesondere $x = (2^n - |x|)$ modulo 2^n für negative x. Die Zahl $2^n - |x|$ nennt man das n-stellige *Zweierkomplement* von $x \in \mathbb{Z}$. Damit erhält man eine bijektive Abbildung

$$\{-2^n, \ldots, -1\} \to \{0, \ldots, 2^n - 1\}, \quad x \mapsto 2^n - |x| = 2^n + x$$

welche die Addition und Subtraktion modulo 2^n respektiert. Damit hat man Addition und Subtraktion modulo 2^n für die Zahlen in $\{-2^n, \ldots, -1\}$ auf die der Zahlen in $\{0, \ldots, 2^n - 1\}$ zurückgeführt. Um gleichzeitig mit positiven und negativen Zahlen rechnen zu können, beschränkt man sich auf den Wertebereich $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$. Die Abbildung

$$\varphi\colon \{-2^{n-1}, \ldots, 2^{n-1} - 1\} \to \{0, \ldots, 2^n - 1\}, \quad x \mapsto (2^n + x) \bmod 2^n$$

ist bijektiv und respektiert die Addition und Subtraktion modulo 2^n. Für die negativen x ist sie gerade die Zweierkomplementbildung und für die nichtnegativen x die identische Abbildung. Das höchstwertige Bit von $\varphi(x)$ gibt das Vorzeichen von x an; es ist genau dann 1, wenn x negativ ist. Aus mathematischer Sicht ist es einfach so, dass man in der Quotientengruppe $\mathbb{Z}/2^n\mathbb{Z}$ rechnet und dabei das eine Mal das Repräsentantensystem $\{-2^{n-1}, \ldots, 2^{n-1} - 1\}$ und das andere Mal das Repräsentantensystem $\{0, \ldots, 2^n - 1\}$ für die Restklassen verwendet.

Die Umkehrabbildung von φ kann leicht berechnet werden. Sei $y \in \{0, \ldots, 2^n - 1\}$. Ist $y < 2^{n-1}$, so ist $\varphi^{-1}(y) = y$. Ist $y \geq 2^{n-1}$, so ist $\varphi^{-1}(y) = -(2^n - y)$, also gerade das negative Zweierkomplement; insbesondere ist das Zweierkomplement $2^n - y$ dann der Betrag der durch y dargestellten negativen Zahl.

Das Zweierkomplement von $x \in \{0, \ldots, 2^n - 1\}$ kann mit Hilfe der n-stelligen Dualdarstellung von x leicht berechnet werden. Man braucht nur das sogenannte *Einserkomplement* zu bilden, d.h. alle Bits umzukippen, und dann 1 zu addieren, modulo 2^n natürlich.

2.3.6 g-adische Entwicklung positiver reeller Zahlen

Sei $g \in \mathbb{N}, g \geq 2$. Dann gibt es zu jedem $r \in \mathbb{R}, r \geq 0$, ein $k \in \mathbb{N}$, Zahlen $a_0, \ldots, a_k \in \{0, \ldots, g - 1\}$ und eine Folge $(b_n)_{n \in \mathbb{N}}$ von Zahlen in $\{0, \ldots, g - 1\}$, so daß

$$r = \sum_{j=0}^{k} a_j \, g^j \; + \; \sum_{n=1}^{\infty} b_n \, g^{-n}.$$

k und $a_0, \ldots, a_k$ und die b_n sind nicht notwendigerweise eindeutig.
Es ist $\sum_{j=0}^{k} a_j \, g^j \in \mathbb{N}_0$ und $\sum_{n=1}^{\infty} b_n \, g^{-n} \in [0, 1]$.

Spezialfall: $g = 10$. Dann erhält man die sogenannte Dezimalbruchentwicklung.

Um daraus eine Codierung zu gewinnen, die in der Praxis verwendbar ist, beschränkt man sich auf reelle Zahlen r, für die nur endlich viele der b_n ungleich 0 sind. Die übliche Schreibweise ist $a_k a_{k-1} \dots a_0.b_1 b_2 \dots b_m$, wenn $b_n = 0$ für $n > m$. Im Deutschen verwendet man statt des Punktes ein Komma; in Programmiersprachen dagegen immer den Punkt.

Die Zahlen, die sich so darstellen lassen, sind rational; aber bei fest gewähltem g lassen sich nicht alle rationalen Zahlen so darstellen; so hat z.B. $\frac{1}{3}$ keine endliche Dezimalbruchentwicklung.

2.3.7 Gleitpunktdarstellungen

Sei $g \in \mathbb{N}, g \geq 2$. Schreibt man ein $r \in \mathbb{R}, r > 0$, in der Form $r = \mu \cdot g^\alpha$ mit $\mu \in \mathbb{R}, \mu > 0$, und $\alpha \in \mathbb{Z}$, so nennt man μ die Mantisse und α den Exponenten. So eine Darstellung von r heißt eine Gleitpunktdarstellung, wenn die Mantisse durch ihre g-adische Entwicklung angegeben ist. Man spricht von **normalisierter Gleitpunktdarstellung**, wenn α so gewählt ist, daß $\mu \in [1, g[$ liegt.

Beispiel: $g = 10, r = 3.717 \cdot 10^5 = 0.3717 \cdot 10^6$. Andere Schreibweisen: `3.717E+5` oder `0.3717E+6`.

Viele Compiler halten sich an die ANSI/IEEE-Norm 754. Sie verwenden normalisierte Gleitpunktdarstellungen bezüglich der Basis $g = 2$. Beim C-Typ `double` wird die Mantisse als 52-stellige Dualzahl dargestellt, wobei die führende 1 weggelassen wird. Statt des Exponenten α wird die sogenannte Charakteristik $\alpha + 1023$ verwendet, die eine 11-stellige Dualzahl ist. Für den zugelassenen Bereich des Exponenten $\alpha \in \{-1023, \dots, 1023\}$ ist die Charakteristik nicht negativ. Das Vorzeichen der Zahl wird separat als 1 Bit gespeichert. In IEEE 754 werden noch weitere Formate unterschiedlicher Genauigkeit festgelegt. Außerdem wurde noch ein Pseudowert `NaN` (Not a Number) eingeführt, der undefinierten Ausdrücken wie $\frac{0}{0}$ zugeordnet wird. Eine Besonderheit ist die Darstellung der 0; sie wird dadurch charakterisiert, dass Mantisse und Charakteristik beide 0 sind.

Führt man mit solchen Gleitpunktzahlen Rechenoperationen aus, so kommt es vor, dass das exakte Ergebnis keine Zahl ist, die wieder als Gleitpunktzahl des vorgegebenen Formats dargestellt werden kann. Dann wird das Ergebnis zu einer darstellbaren Zahl gerundet. IEEE 754 gibt Regeln an, wie das geschehen soll. Ein weiterer Rundungsfehler tritt auf, wenn ein Programm Dezimalbrüche einliest, die dann in die interne duale Gleitpunktdarstellung umgewandelt werden; denn aus einer Dezimaldarstellung endlicher Länge kann eine Dualdarstellung unendlicher Länge werden. So hat z.B. der Dezimalbruch 0.1 die unendliche, periodische Dualbruchentwicklung $0.0\overline{0011}$. Beweis: Der Wert des Dualbruchs $0.0\overline{0011}$ errechnet sich (in üblicher Dezimalschreibweise) als

$$\begin{aligned} \frac{1}{2} \cdot \frac{3}{16} \cdot \sum_{n=0}^{\infty} \frac{1}{16^n} &= \frac{1}{2} \cdot \frac{3}{16} \cdot \frac{1}{1 - \frac{1}{16}} \\ &= \frac{1}{2} \cdot \frac{3}{16} \cdot \frac{16}{15} \\ &= \frac{1}{10} \end{aligned}$$

Während das Rechnen mit Ganzzahltypen im Rechner abgesehen von Wertebereichs-

überschreitungen exakt erfolgt, ist das Rechnen mit Gleitpunktzahlen im Rechner stets mit Rundungsfehlern verbunden. Deshalb sollte man auch vermeiden, die Werte von Variablen mit Gleitpunkttyp auf Gleichheit zu vergleichen; stattdessen sollte man lieber mit Ungleichungen prüfen, ob die Werte innerhalb eines geeigneten Intervalls liegen. Insbesondere sollte man eine Überprüfung der Gleichheit oder Ungleichheit von Gleitpunktvariablen nicht als Abbruchbedingung für eine Schleife verwenden. So wird das folgende Programm nicht nach 10 Schleifendurchläufen enden. Probieren Sie es aus! Wenn man den Ausdruck `(x!=1)` durch `(x<1)` ersetzt, wird es das Gewünschte tun.

```
#include <stdio.h>

int main(void) {
   double x=0;
   int i=0;

   while(x!=1) {
      printf("%d    %f\n",i,x);
      x+=0.1;
      i+=1;
      getchar();
   }
   return 0;
}
```

2.4 Codierung von Programmtexten

Ein Programmtext besteht aus sogenannten Terminalsymbolen, die gemäß den Regeln der verwendeten Programmiersprache zusammengestellt werden. Meist besteht die Menge der Terminalsymbole aus einer Teilmenge der ASCII-Zeichen sowie einigen Wörtern über der Menge der ASCII-Zeichen, den sogenannten Schlüsselwörtern (key words) der Programmiersprache wie `if, while, for` usw. Konkrete Zahlenwerte werden meist in einer der oben angegebenen oder ähnlichen Weisen dargestellt. Außerdem kommen meist noch Steuerzeichen hinzu wie `LF, CR, TAB`, die nicht zur Programmiersprache gehören, sondern nur zur übersichtlichen Visualisierung des Programmtextes im Editor dienen.

Fazit: Ein Programmtext (oder Quelldatei, source file) ist eine Folge von ASCII-Zeichen.

Mit Hilfe der ASCII-Codierung φ (siehe 2.2.2) macht man aus einem Programmtext eine Folge von Zahlen aus $\{0, \ldots, 127\}$. Jede dieser Zahlen wird im Rechner in einem Byte abgespeichert. Einige Firmen verwendeten auch andere Codierungen (z.B. den EBCDI-Code).

1 Byte enthält 8 Bit; also kann man in einem Byte ein beliebiges 8-Tupel aus $\{0,1\}^8$ speichern. Üblicherweise wählt man fallende Indizierung d.h. man schreibt das 8-Tupel als $b_7b_6b_5b_4b_3b_2b_1b_0$ mit $b_j \in \{0,1\}$ und faßt es als Dualdarstellung der Zahl $\sum_{j=0}^{7} b_j\, 2^j$ auf. Das Bit b_7 heißt das höchstwertige Bit oder MSB (=Most Significant

Bit), das Bit b_0 das niedrigstwertige Bit oder LSB (=Least Significant Bit).

Wie wir schon in 2.3.2 gesehen haben, ist die Abbildung

$$\{0,1\}^8 \to \{0,\ldots,255\}, \quad b_7 \ldots b_0 \mapsto \sum_{j=0}^{7} b_j \, 2^j$$

bijektiv. Vermöge dieser Abbildung identifiziert man die möglichen Inhalte eines 1-Byte-Speichers mit den Zahlen $\{0,\ldots,255\}$. In diesem Sinne kann man die Codes der ASCII-Zeichen eines Programmtextes in jeweils einem Byte speichern (mit MSB= 0).

2.5 Grobe Gliederung des Übersetzungsvorgangs

Wie kann man den Rechner dazu bringen, daß er den in einem Programmtext (Quelldatei) beschriebenen Algorithmus ausführt? Man muß den Programmtext in eine andere Sprache übersetzen, nämlich in die Sprache des Prozessors im Rechner, die sogenannte Maschinesprache. Während Programme in höheren Programmiersprachen weitgehend rechnerunabhängig geschrieben werden können, sind Programme in Maschinensprache völlig vom Prozessortyp und der Rechnerarchitektur abhängig. Solche Übersetzer sind selbst Programme in Maschinesprache, die den Rechner dazu bewegen, die Quelldatei einzulesen und daraus ein lauffähiges Programm in Maschinensprache, den sogenannten Objektcode, zu erzeugen. Sie heißen *Compiler*.

2.5.1 Lexikalische Analyse (Scanning): Der Programmtext in der Quelldatei wird Zeichen für Zeichen durchlaufen. Dabei werden syntaktisch überflüssige Steuerzeichen (mehrfache Leerzeichen, CR usw.) entfernt, und es werden die Terminalsymbole (insbesondere die Schlüsselwörter wie z.B. `if, while` usw.) erkannt und auf eine systematische Weise neu bezeichnet (durch sogenannte Token), so daß die maschinelle Weiterverarbeitung erleichtert wird.

2.5.2 Syntaktische Analyse (Parsing): Es wird untersucht, ob die in 2.5.1 erzeugte Folge von Terminalsymbolen (oder Token) ein nach den Regeln der jeweiligen Programmiersprache syntaktisch korrekt gebildeter Programmtext ist. Gleichzeitig wird ein Ableitungsbaum erstellt, der angibt, wie die Terminalsymbole mit Regeln der Programmiersprache abgeleitet werden können. Außerdem wird eine Symboltabelle angelegt, in der die Namen von Variablen, Marken usw. sind, aufgeführt sind.

2.5.3 Semantische Analyse und Maschinencodeerzeugung Den im Programmtext genannten Daten (Eingabe-, Ausgabedaten, Zwischenergebnisse) werden Speicherplätze zugewiesen, deren Adressen in der Symboltabelle an der entsprechenden Stelle eingetragen werden. Dann wird der in 2.5.2 erstellte Ableitungsbaum durchlaufen, und dabei werden die vorkommenden syntaktischen Strukturen in eine Folge von Anweisungen in Maschinensprache übersetzt. Dieses übersetzte Programm wird auch Objektcode oder Binärobjekt genannt.

2.5.4 Optimierung Der in 2.5.3 erzeugte Maschinencode wird hinsichtlich Laufzeit und/oder Speicherbedarf verbessert.

2.6 Bibliotheken (Libraries)

Programmteile, die sehr häufig und in vielen Programmen vorkommen (wie Ein- und Ausgaberoutinen), sind - bereits fertig compiliert - in sogenannten Libraries zusammengefaßt und tragen einen (möglichst unverwechselbaren) Namen. In eigenen Programmen kann man sie unter Verwendung dieses Namens einbauen. Allerdings wird ihr Maschinencode nicht vom Compiler eingesetzt. Das macht erst der Linker.

2.7 Binder (Linker)

Dem Linker übergibt man ein compiliertes Programm (oder auch mehrere) und den Namen von denjenigen Bibliotheken, aus denen Routinen verwendet werden. Beim *statischen Binden* setzt er für die Routinenamen den entsprechenden Objektcode ein. Beim *dynamischen Binden* setzt er nur einen Verweis ein, wo sich die benötigte Bibliothek befindet. Der Objektcode der gewünschten Routine wird erst während des Programmlaufs eingesetzt, wenn er wirklich benötigt wird. Dadurch wird zwar die Programmausführung ein klein wenig langsamer, dafür ist aber der Maschinencode des dynamisch gebundenen Programms viel kürzer als der des statisch gebundenen.

2.8 Lader (Loader)

Soll ein fertig gebundenes Programm ausgeführt werden, so schreibt der Lader den Programmcode in den Arbeitsspeicher und übergibt die Kontrolle seiner Ausführung dem Betriebssystem.

2.9 Interpreter, Scriptsprachen

Ein Compiler übersetzt einen Programmtext als ganzes in eine Folge von Befehlen der Maschinensprache, den sog. Objektcode, der dann anschließend vom Rechner ausgeführt werden kann. Ein Interpreter hingegen bearbeitet den Quellcode strikt Zeile für Zeile; jede Zeile wird gescanned, geparsed, übersetzt, und der enstehende Maschinencode wird ausgeführt, bevor die nächste Zeile angeschaut wird. Dieses Vorgehen hat Vor- und Nachteile. Einerseits kann man sofort mit der Programmausführung beginnen, ohne erst die Compilation des gesamten Programms abwarten zu müssen, andererseits ist die Programmausführung viel langsamer als bei einem bereits compilierten Programm; außerdem ist eine gute Optimierung des Objektcodes kaum möglich. Für manche Anwendungen sind Interpreter aber sehr geeignet, z.B. um Kommandos abzuarbeiten, die interaktiv von einem Menschen eingegeben werden (Kommandointerpreter, Shells). Die meisten Shells verstehen eine mehr oder weniger raffinierte Programmiersprache, mit der oft komplexe Programmabläufe und Dateimanipulationen, die in grafischen Benutzeroberflächen undenkbar sind, recht einfach durchführt werden können. Zwei bekannte, sehr komfortable Shells sind `bash` und `tcsh`.

Es gibt auch eigenständige Programmiersprachen, die von vornherein für interpretative Abarbeitung konzipiert wurden, wie BASIC, Tcl/Tk, Perl. Darin geschriebene Programme laufen ohne weitere Vorbereitung auf jedem Rechner, auf dem ein entsprechender Interpreter vorhanden ist. Dagegen erweist es sich in der Praxis oft als schwierig, Programme auf Rechnerplattformen, für die sie nicht entwickelt wurden, zu compilieren.

Einen Mittelweg versuchte die Firma Sun mit der Entwicklung von JAVA zu gehen. Dabei handelt es sich um eine Programmiersprache, die compiliert wird, allerdings nicht in den Maschinencode des jeweiligen Rechners, sondern in den Code eines Interpreters, der Java Virtual Machine. Die (nicht ganz so neue) Idee ist, einerseits durch die Compilation in einen einfachen, schnell interpretierbaren Code eine schnelle Abarbeitung zu erreichen, andererseits eine gewisse Plattformunabhängigkeit dadurch zu erreichen, daß nur die Java Virtual Machine auf unterschiedlichen Rechnern implementiert werden muß. In der Praxis erweist sich das als ein mehr oder weniger gut geglückter Kompromiß.

Kapitel 3

Formale Sprachen

Gemäß unserer Definition 0.5.1 ist die syntaktische Beschreibung eines Algorithmus ein endliches Textstück. Umgangssprachliche Texte können von Maschinen (bisher) nur sehr schlecht verstanden werden. Deshalb verwendet man stark eingeschränkte, formalisierte Sprachen.

3.1 Kontextfreie Sprachen

3.1.1 Definition *Eine* ***formale Sprache über dem Alphabet*** A *ist eine Teilmenge von* A^*.

Zur Verarbeitung durch Maschinen sind nicht alle formalen Sprachen gleich gut geeignet; es ist sicher günstig, wenn die Sprache aus einigen wenigen Wörtern durch einfache Regeln aufgebaut werden kann. Im Zusammenhang mit Programmiersprachen spielen die Sprachen mit kontextfreier Grammatik eine wichtige Rolle.

3.1.2 Definition *Eine* ***kontextfreie Grammatik*** *oder* ***Chomsky-Grammatik vom Typ 2*** *ist eine Quadrupel* $\mathcal{G} = (A, V, P, S)$ *mit folgenden Eigenschaften:*

1) A *ist ein Alphabet; die Elemente von* A *werden Terminalsymbole genannt.*

2) V *ist eine endliche Menge; die Elemente von* V *werden syntaktische Variablen genannt.*

3) $S \in V$ *heißt das Startsymbol.*

4) $A \cap V = \emptyset$

5) P *ist eine endliche Teilmenge von* $V \times (A \cup V)^*$*; die Elemente von* P *werden Produktionen genannt. Für* $(v, w) \in P$ *schreibt man meist* $v \to w$.

3.1.3 Definition $\mathcal{G} = (A, V, P, S)$ *sei eine kontextfreie Grammatik.*

a) *Man sagt, ein Wort* $b \in (A \cup V)^*$ *entstehe aus einem Wort* $a \in (A \cup V)^*$ *durch einem direkten* ***Ableitungsschritt*** *vermöge* P *(kurz:* $a \overset{P,1}{\to} b$*), wenn gilt:*

1) *a hat die Gestalt $a = a_1 v a_2$ mit $a_1 \in (A \cup V)^*, v \in V, a_2 \in (A \cup V)^*$.*

2) *b hat die Gestalt $b = a_1 w a_2$ mit $w \in (A \cup V)^*$.*

3) *P enthält die Regel $v \to w$.*

b) *Man sagt, ein Wort $b \in (A \cup V)^*$ kann mit P aus einem Wort $a \in (A \cup V)^*$ abgeleitet werden, wenn es endlich viele Wörter $w_1, \ldots, w_n \in (A \cup V)^*$ gibt derart, daß $w_1 = a, w_n = b$ und für jedes $j \in \{1, \ldots, n-1\}$ das Wort w_{j+1} aus w_j durch einen direkten Ableitungsschritt entsteht d.h. wenn $a \overset{P,1}{\to} w_2 \overset{P,1}{\to} \ldots \overset{P,1}{\to} w_{n-1} \overset{P,1}{\to} b$ gilt.*
$a \overset{P}{\to} b$ bedeute, daß b mit P aus a abgeleitet werden kann.

c) $\mathcal{L}(\mathcal{G}) := \{x \in A^* : x$ kann mit P aus S abgeleitet werden$\}$ *heißt **die von $\mathcal{G}$ erzeugte Sprache**.*

d) *Eine Sprache $L \subset A^*$ heißt **kontextfrei**, wenn es eine kontextfreie Grammatik $\mathcal{G} = (A, V, P, S)$ mit $L = \mathcal{L}(\mathcal{G})$ gibt.*

3.1.4 Beispiel Seien $A = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}$ und $D := \{w \in A^* : w = \mathtt{0}$ oder das erste Zeichen von w ist ungleich $\mathtt{0}\}$. Die Elemente von D sind gerade die Dezimaldarstellungen der Zahlen in $\mathbb{N}_0$ ohne führende Nullen.

3.1.4.1 Lemma *D ist eine kontextfreie Sprache, die von folgender Grammatik $\mathcal{G} = (A, V, P, S)$ erzeugt wird:*

A sei wie oben die Menge der Ziffern $\mathtt{0}$ bis $\mathtt{9}$ und V sei die Menge $\{S, X, N, Z\}$. Die Menge P bestehe aus folgenden Produktionen:

1) $S \to \mathtt{0}$ 2) $S \to X$ 3) $X \to N$ 4) $X \to XZ$

5) $Z \to \mathtt{0}$ und $Z \to N$

6) $N \to \mathtt{1}$, $N \to \mathtt{2}$, $N \to \mathtt{3}$, $N \to \mathtt{4}$, $N \to \mathtt{5}$, $N \to \mathtt{6}$, $N \to \mathtt{7}$, $N \to \mathtt{8}$, $N \to \mathtt{9}$

Beweis. Sei $L = \mathcal{L}(\mathcal{G})$. Zu zeigen sind $D \subset L$ und $L \subset D$.

1. Teil: Zeige $D \subset L$.
Sei $w \in D$. Zu zeigen ist $w \in L$. Der Beweis wird durch vollständige Induktion nach der Länge n von w geführt.

Induktionsanfang: $n = 1$. Es ist $w = \mathtt{0}$ oder $w = \mathtt{1}$ oder ... $w = \mathtt{9}$. Das Wort $w = \mathtt{0}$ läßt sich mit der Produktion 1 in einem Schritt aus S ableiten. Ein Wort $w = a \in A$ mit $a \neq \mathtt{0}$ läßt sich in drei Schritten aus S ableiten; es gilt nämlich $S \overset{Prod.2}{\to} X \overset{Prod.3}{\to} N \overset{Prod.6}{\to} a$. Also gilt $w \in L$.

Induktionsschluß: Zu zeigen ist, daß für jedes $n \in \mathbb{N}$ die folgende Induktionsvoraussetzung die Induktionsbehauptung impliziert.

Induktionsvoraussetzung: Für jedes $v \in D$ der Länge n gilt $v \in L$.

Induktionsbehauptung: Für jedes $v \in D$ der Länge $n + 1$ gilt $v \in L$.

Beweis des Induktionsschlußes: Sei $w \in D$ mit Länge $n + 1$. Das Wort v entstehe aus w durch Weglassen des letzten Zeichens. Dann gilt $v \in D$, denn v hat die Länge $n \geq 1$ und das erste Zeichen von v ist nicht $\mathtt{0}$. Nach Induktionsvoraussetzung gilt $v \in L$. Es gibt also eine Ableitung $S \overset{P}{\to} v$. Wegen $v \neq \mathtt{0}$ kann der erste

Schritt dieser Ableitung nicht die Produktion 1 sein. Er muß also die Produktion $S \to X$ sein; denn es gibt keine andere mit linker Seite S. Mit den restlichen Schritten wird v aus X abgeleitet. Sei a das letzte Zeichen von w. Ersetze nun den ersten Schritt $S \to X$ durch $S \overset{Prod.2}{\to} X \overset{Prod.4}{\to} XZ \overset{Prod.5}{\to} X\mathtt{0}$, falls $a = \mathtt{0}$, und durch $S \overset{Prod.2}{\to} X \overset{Prod.4}{\to} XZ \overset{Prod.5}{\to} XN \overset{Prod.6}{\to} Xa$, falls $a \neq \mathtt{0}$, und füge die Ableitung von v aus X hinten an. Das ergibt eine Ableitung von $w = va$ aus S. Also ist $w \in L$.

2. Teil: Zeige $L \subset D$.
Sei $w \in L$. Zu zeigen ist $w \in D$. Wegen $L \subset A^*$ genügt es zu zeigen, daß w nicht das leere Wort ist und das erste Zeichen von w ungleich $\mathtt{0}$ ist, falls $w \neq \mathtt{0}$.
$w \neq \epsilon$ gilt, weil keine Produktion das leere Wort als rechte Seite hat. Sei $S \to w_1 \to w_2 \to \ldots \to w_n = w$ eine Ableitung von w aus S. Ist der erste Ableitungsschritt die Produktion 1, also $S \to \mathtt{0}$, so ist $n = 1$ und $w = \mathtt{0}$, und $w \in D$. Anderenfalls ist der erste Ableitungsschritt die Produktion 2, also $S \to X$, und somit $w_1 = X$, und es genügt, folgende Behauptung zu zeigen.
Behauptung: Kann $v \in A^*$ aus X abgeleitet werden, so ist das erste Zeichen von v nicht $\mathtt{0}$.
Beweis der Behauptung: Nur die Produktionen 3 und 4 haben X auf der linken Seite. Durch Anwendung von 4 kann X nicht eliminiert werden, sondern bleibt als erstes Zeichen des entstehenden Wortes. X kann nur dadurch durch Terminalzeichen ersetzt werden, daß erst die Produktion 3, also $X \to N$, und später eine der in 6 aufgeführten Produktionen angewendet wird. Folglich kann das erste Zeichen von v nicht $\mathtt{0}$ sein.
q.e.d.

3.1.5 Beispiel Die Menge der arithmetischen Ausdrücke in n Variablen $x_1, \ldots, x_n$ mit den Operationssymbolen $+, -, *, /$ und korrekter Klammerung ist eine kontextfreie Sprache, die von der Grammatik (A, V, P, S) erzeugt wird, für die gilt: $A = \{x_1, \ldots, x_n, +, -, *, /, (,)\}, V = \{S\}$ und $P = \{S \to S + S,\ S \to S - S,\ S \to S * S,\ S \to S/S,\ S \to (S),\ S \to x_1, \ldots, S \to x_n\}$.

3.1.6 Programmiersprachen Kontextfreie Grammatiken werden zur Definition von Programmiersprachen benutzt. Idealerweise bilden die Syntaxregeln einer Programmiersprache eine kontextfreie Grammatik, und die Programme, die in dieser Sprache geschrieben werden können, sind einfach die Wörter der durch die Grammatik definierten Sprache. In der theoretischen Informatik untersucht man, für welche formalen Sprachen es einen Automaten gibt, der entscheiden kann, ob ein Wort zu der Sprache gehört oder nicht. Die meisten Programmiersprachen enthalten jedoch Regeln, die nicht kontextfrei beschreibbar sind. Statt zu einer allgemeineren kontextsensitiven Grammatik, die meist unübersichtlicher ist, überzugehen, wählt man lieber eine kontextfreie Teilbeschreibung der Sprache und fügt weitere umgangssprachliche Forderungen hinzu.

Aber auch kontextfreie Grammatiken können sehr viele Produktionen enthalten und unübersichtlich werden. Deshalb hat man sich andere Arten der Beschreibung von Grammatiken ausgedacht, die wir kurz skizzieren. Um eine Sprache zu beschreiben, muß man sich einer anderen Sprache bedienen; diese nennt man eine *Metasprache*. Um Verwirrungen zu vermeiden, muß klar sein, welche Zeichen zu welcher Sprache gehören. Schreibt man z.B. die Produktionen der Grammatik in der Form $v \to w$, wie wir es getan haben, so ist der Pfeil ein Zeichen der Metasprache.

3.2 Die Backus-Naur-Form (BNF)

Die BNF verwendet folgende Metazeichen:

::=	Definitionszeichen
|	Alternativzeichen
<*string*>	Bezeichnung für syntaktische Variablen, wobei *string* eine beliebige Kette von Buchstaben und Ziffern ist. Dadurch kann man den Variablen Namen geben, die ihre inhaltliche Bedeutung widerspiegeln.

A sei das Alphabet der Terminalzeichen, V die Menge der syntaktischen Variablen (die alle die Form <*string*> haben).

Die BNF verwendet zwei Arten von Regeln:

1) <*string*> ::= a, wobei $a \in (A \cup V)^*$.

2) <*string*> ::= $a_1|a_2|\ldots|a_n$, wobei $n \in \mathbb{N}, a_1, \ldots, a_n \in (A \cup V)^*$.

Eine *BNF-Beschreibung* einer kontextfreien Sprache L besteht aus einem Quadrupel (A, V, R, S), wobei A das Terminalalphabet, V die Menge der syntaktischen Variablen (in obiger Form), R eine endliche Menge von BNF-Regeln der Gestalt 1 oder 2 und $S \in V$ das sogenannte Startsymbol sind. Einer BNF-Beschreibung (A, V, R, S) ordnet man eine Grammatik $(A, V, P(R), S)$ zu, indem man die BNF-Regeln in R durch Produktionen ersetzt, die zur Menge $P(R)$ zusammengefaßt werden. Dies geschieht auf folgende Weise:

Ersetze jede BNF-Regel <*string*> ::= a durch die Produktion <*string*>$\rightarrow a$
und jede BNF-Regel <*string*> ::= $a_1|a_2|\ldots|a_n$ durch die endlich vielen Produktionen <*string*>$\rightarrow a_1$,..., <*string*>$\rightarrow a_n$.

Die von der Grammatik $(A, V, P(R), S)$ erzeugte Sprache heißt dann die von (A, V, R, S) erzeugte Sprache.

3.2.1 Beispiel Die Menge der Dezimaldarstellungen der Zahlen in $\mathbb{N}_0$ ohne führende Null hat folgende BNF-Beschreibung (A, V, R, S).

$A = \{0,\ldots,9\}$, $V = \{$<*Zahl*>,<*X*>,<*Nichtnullziffer*>,<*Ziffer*>$\}$, $S :=$<*Zahl*>.
Die Menge der Regeln R besteht aus:

```
          <Zahl> ::= 0 |<X>
             <X> ::= <Nichtnullziffer>|<X><Ziffer>
        <Ziffer> ::= 0 |<Nichtnullziffer>
<Nichtnullziffer> ::= 1|2|3|4|5|6|7|8|9
```

Das ist schon viel übersichtlicher und verständlicher als die Beschreibung 3.1.4.

3.3 EBNF

Wie in diesem Beispiel kommt es häufig vor, daß eine Regel mehrfach angewendet werden muß, um an einer Stelle in einem Wort ein Zeichen (oder Zeichen einer gewissen Art) mehrfach hinzuschreiben. Zur Vereinfachung und besseren Übersichtlichkeit verwendet man oft eine Erweiterung der BNF-Notation, die sogenannte EBNF-Notation

(Extended BNF), die von N. Wirth eingeführt wurde. Wir geben zunächst die *Syntax* an (etwas modifiziert gegenüber Wirths EBNF).

Als Metazeichen werden verwendet:

`::= | { } [ ] ( )`

A sei das Terminalalphabet, V die Menge der syntaktischen Variablen.

3.3.1 Definition *Die Menge der EBNF-Terme ist die kleinste Teilmenge $\mathcal{T}(A,V)$ von $(A \cup V \cup \{\text{Metazeichen}\})^*$, die folgende Eigenschaften hat:*

a) $V \subset \mathcal{T}(A,V)$

b) $A^* \subset \mathcal{T}(A,V)$

c) *Sind $\alpha_1, \ldots, \alpha_n \in \mathcal{T}(A,V)$, so sind auch $(\alpha_1 | \alpha_2 | \ldots \alpha_n)$ und $\alpha_1 \ldots \alpha_n$ in $\mathcal{T}(A,V)$.*

d) *Ist $\alpha \in \mathcal{T}(A,V)$, so sind (α), $[\alpha]$ und $\{\alpha\}$ in $\mathcal{T}(A,V)$.*

Überlegen Sie sich, daß $\mathcal{T}(A,V)$ wohldefiniert ist und daß die folgende Aussage gilt.

3.3.2 Lemma Jeder EBNF-Term läßt sich durch endlichfaches Anwenden von a) bis d) konstruieren.

3.3.3 Definition *Eine **EBNF-Beschreibung** ist ein Quadrupel (A,V,R,S), wobei A das Terminalalphabet, V die Menge der syntaktischen Variablen, $S \in V$ das Startsymbol und R eine endliche Menge von Regeln der Form `v ::=` α mit $v \in V$ und $\alpha \in \mathcal{T}(A,V)$ sind.*

3.3.4 Bemerkung Die Syntax der EBNF von Wirth unterscheidet sich von unserer etwas; so wird z.B. `=` statt `::=` verwendet, die Regeln enden mit einem Punkt und die syntaktischen Variablen sind Zeichenketten ohne Leerzeichen, die nicht in spitze Klammern eingeschlossen sind. Wir wollen uns hier jedoch mehr an der Syntax der BNF orientieren.

Wir kommen jetzt zur *Semantik*, d.h. wir geben an, welche Sprache durch eine EBNF-Beschreibung beschrieben wird. Diese Sprache wird induktiv über den Aufbau der EBNF-Terme konstruiert. Die Potenzmenge einer Menge M bzeichnen wir mit $\mathcal{P}(M)$.

3.3.5 Definition *(A,V,R,S) sei eine EBNF-Beschreibung.*
Dann wird die Abbildung $\sigma\colon \mathcal{T}(A,V) \to \mathcal{P}(A^*)$ folgendermaßen definiert.

a) *Für $w \in A^*$ sei $\sigma(w) = \{w\}$.*

b) *Für $v \in V$ sei*

$$\sigma(v) = \begin{cases} \emptyset, \text{ falls } v \text{ nicht als linke Seite einer Regel vorkommt,} \\ \sigma(\alpha_1) \cup \ldots \cup \sigma(\alpha_n) \text{ sonst,} \end{cases}$$

wobei $\alpha_1, \ldots, \alpha_n$ die rechten Seiten derjenigen Regeln in R sind, deren linke Seite v ist.

c) *Für $\alpha_1, \ldots, \alpha_n \in \mathcal{T}(A,V)$ seien*

$$\begin{aligned} \sigma((\alpha_1 | \ldots | \alpha_n)) &= \sigma(\alpha_1) \cup \ldots \cup \sigma(\alpha_n) \\ \sigma(\alpha_1 \ldots \alpha_n) &= \{w_1 \ldots w_n \colon w_i \in \sigma(\alpha_i)\} \end{aligned}$$

d) *Für $\alpha \in \mathcal{T}(A,V)$ seien*

$$\begin{aligned} \sigma((\alpha)) &= \sigma(\alpha) \\ \sigma([\alpha]) &= \sigma(\alpha) \cup \{\epsilon\} \\ \sigma(\{\alpha\}) &= \sigma(\alpha)^* \end{aligned}$$

d.h. $\sigma(\{\alpha\})$ besteht aus allen Wörtern, die sich als Verkettung von endlich vielen Wörtern aus $\sigma(\alpha)$ darstellen lassen, einschließlich des leeren Wortes.

3.3.6 Bemerkung Daß durch obige Definition tatsächlich jedem EBNF-Term $\alpha \in \mathcal{T}(A,V)$ auf eindeutige Weise eine Sprache $\sigma(\alpha)$ zugeordnet wird, bedarf eines Beweises! Ein formal präziser Beweis ist gar nicht so einfach zu führen. Wir verzichten hier auf einen Beweis, weil in konkreten Fällen die Bestimmung von $\sigma(\alpha)$ intuitiv meist klar ist.

3.3.7 Definition *Ist (A,V,R,S) eine EBNF-Beschreibung, so heißt $\sigma(S)$ die von ihr erzeugte Sprache.*

3.3.8 Beispiel Seien $A = \{0,\ldots,9\}$,
$V = \{$<Zahldarstellung>, <Nichtnullziffer>$\}$, $S =$<Zahldarstellung>.

<Zahldarstellung> ::= 0|<Nichtnullziffer>{(0|1|2|3|4|5|6|7|8|9)}
<Nichtnullziffer> ::= (1|2|3|4|5|6|7|8|9)

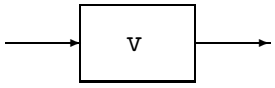
$\sigma(S)$ ist die Menge der Dezimaldarstellungen ohne führende Nullen der Zahlen in $\mathbb{N}_0$.

3.4 Syntaxdiagramme

Zur grafischen Verdeutlichung von EBNF-Regeln verwendet man Syntaxdiagramme. Wir wollen dies nur skizzieren und verzichten auf eine formale Präzisierung.

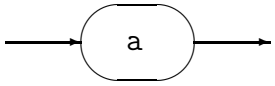
Jedem EBNF-Term a wird ein Diagramm D_a zugeordnet, das genau einen Eingangs- und einen Ausgangspfeil hat, und zwar:

a) $v \in V$ wird



zugeordnet.

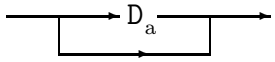
b) $a \in A^*$ wird



zugeordnet.

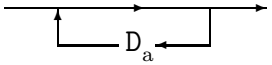
c) Ist $a \in \mathcal{T}(A,V)$, so wird (a) das gleiche Diagramm wie a zugeordnet.

[a] wird



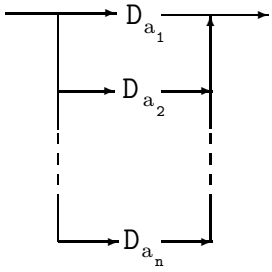
zugeordnet.

{a} wird



zugeordnet, wobei D_a das a zugeordnete Diagramm ist (das ja genau einen Eingangs- und einen Ausgangspfeil hat).

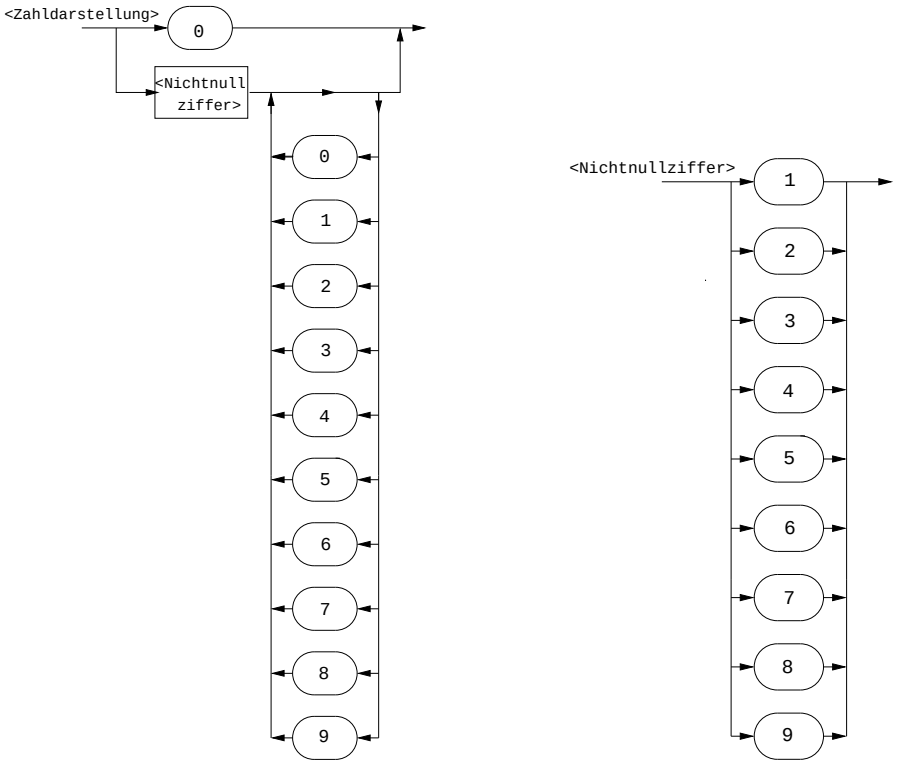
d) Sind $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathcal{T}(A, V)$ und $\mathtt{D}_{\mathtt{a}_1}, \ldots, \mathtt{D}_{\mathtt{a}_n}$ die zugeordneten Diagramme, so wird $\mathtt{a}_1 \ldots \mathtt{a}_n$ das Diagramm $\rightarrow \mathtt{D}_{\mathtt{a}_1} \rightarrow \mathtt{D}_{\mathtt{a}_2} \rightarrow \ldots \rightarrow \mathtt{D}_{\mathtt{a}_n}$ zugeordnet und $\mathtt{a}_1|\mathtt{a}_2|\ldots|\mathtt{a}_n$ das Diagramm



Schließlich wird die EBNF-Regel v `::=` `a` dargestellt durch das Syntaxdiagramm $\overset{v}{\rightarrow} \mathtt{D}_{\mathtt{a}} \rightarrow$.

Die Wörter der einem Term `a` zugeordneten Sprache erhält man, indem man das Diagramm $\mathtt{D}_{\mathtt{a}}$ auf allen Wegen von Eingang zum Ausgang (immer in Pfeilrichtung) durchläuft und dabei jeweils die Symbole in den durchlaufenden Kästchen nebeneinander schreibt.

3.4.1 Beispiel Die in 3.3.8 angegebene EBNF-Beschreibung der Dezimaldarstellung der Zahlen in $\mathbb{N}_0$ wird durch folgende Syntaxdiagramme dargestellt.



3.5 Reguläre Ausdrücke, Dateinamenmuster

Es gibt etliche kommandozeilen-orientierte UNIX-Programme, die Texte auf unterschiedlichste Weise verändern, zerlegen oder aufgrund von Textstücken irgendwelche Kommandos ausführen. Beispiele sind `grep`, das in Texten nach Wörtern oder Wortmustern suchen kann, oder der Stream-Editor `sed`, mit dem Textveränderungen nach vorgegebenen Regeln ohne manuelle Interaktion oder grafische Darstellung vorgenommen werden können. Noch universeller ist das Programm `awk`, mit dem man Texte nach vorgegebenen Regeln zerlegen und Aktionen auslösen kann.

Um solchen Programmen mitzuteilen, mit welcher Menge von Wörtern es irgendetwas tun soll, verwendet man reguläre Ausdrücke. Das sind Ausdrücke aus Metazeichen, die ähnlich gebildet werden wie obige EBNF-Terme. Und wie jedem EBNF-Term wird auch jedem regulären Ausdruck eine Menge von Wörtern über dem Terminalalphabet zugeordnet. Leider gibt es keinen Konsens, welche Metazeichen in welcher Bedeutung verwendet werden. Das kann je nach Programm ein wenig variieren.

In der theoretischen Informatik wird gezeigt, dass mit regulären Ausdrücken genau die formalen Sprachen beschrieben werden, die von endlichen Automaten erkannt werden.

Viele Kommandozeileninterpreter (Shells) lassen zu, dass man eine Menge von Dateinamen durch eine Art von regulärem Ausdruck angibt (die allerdings nicht reguläre Ausdrücke im Sinne regulärer Sprachen sind). Bei der `bash` z.B. sind als Metazeichen u.a. zugelassen: `*,?,[,],!`. Ihre Bedeutung unterscheidet sich ein wenig von der in den regulären Ausdrücken für `grep` usw. Wir beschreiben sie informell, indem wir angeben, welche Arten von Zeichenketten für die einzelnen Ausdrücke eingesetzt werden.

`*`	Kann durch jeden Namen aus 0 oder mehr Zeichen ersetzt werden.
`?`	Kann und muss durch ein einzelnes Zeichen ersetzt werden.
`[`*liste*`]`	Kann durch jedes der in *liste* aufgeführten Zeichen ersetzt werden. Dabei kann die Liste auch als Intervall von ASCII-Zeichen angegeben werden, z.B. `[a-z]` oder `[D-K]` oder `[0-9]`.
`[!`*liste*`]`	Kann durch jedes Zeichen ersetzt werden, das nicht in dem angegebenen Bereich liegt.

`Beispiele`

`*.jpg` ist die Menge aller Dateinamen, die `.jpg` enden.
Somit listet `ls *.jpg` alle Dateien auf, deren Name auf `.jpg` endet.

`?[0-9][0-9]*` ist die Menge aller Dateinamen, die mit irgendeinem Zeichen beginnen, dem zwei Ziffern folgen.

Kapitel 4

Zeiger und Arrays

4.1 Zeiger

Ein Zeigertyp (pointer type) wird von einem anderen Datentyp `T`, dem sogenannten referenzierten Datentyp, abgeleitet und wird als Zeiger auf `T` (pointer to `T`) bezeichnet.

Syntaktische Bezeichnung: `T *`

Der Wert eines Objekts vom Typ "Zeiger auf `T`" ist ein Verweis auf ein Objekt vom Typ `T`, das sogenannte referenzierte Objekt.

Die Definition einer Variablen vom Typ "Zeiger auf `T`" hat die Gestalt

`T *`*variablenname*`;`

Beispiel: Durch `int *p;` wird `p` als ein Zeiger auf `int` deklariert.

Wie soll man sich so einen Verweis auf ein Objekt vom Typ `T` vorstellen?
Da der Typ `T` bereits bei der Deklaration festgelegt wird, genügt es, als Verweis die Anfangsadresse des referenzierten Objekts anzugeben; denn die Länge des vom referenzierten Objekt belegten Speicherbereichs läßt sich aus seinem Typ bestimmen.

Achtung! Man sollte den Wert eines Objekts vom Zeigertyp nicht einfach als eine Zahl, die eine Adresse darstellt, betrachten. Denn wie das referenzierte Objekt im Speicher tatsächlich repräsentiert wird, kann vom Rechner oder Prozessor abhängen. In den einzelnen Bytes steht nicht immer dasselbe drin! Ein typisches Beispiel ist die Darstellung von Zahlen des Typs `unsigned short`. Dafür wird die 16-stellige Dualdarstellung der Zahl benutzt; durch die ANSI-Norm ist aber nicht völlig festgelegt, wie sie im Speicher abgelegt ist. Es gibt prozessorabhängige Unterschiede. Man nennt die 8 höherwertigen Bit der Dualdarstellung das High-Byte und die 8 niedrigerwertigen Bit das Low-Byte. Intel-Prozessoren erwarten im Speicher das Low-Byte auf der niedrigeren Adresse a und das High-Byte auf der höheren Adresse $a + 1$; Prozessoren anderer Firmen wie Motorola dagegen genau umgekehrt. Ein direkter Zugriff auf die beiden Bytes liefert also je nach Rechner unterschiedliche Ergebnisse, ein Zugriff über Zeiger dagegen nicht, weil der Compiler an die Architektur des jeweiligen Rechners angepaßt ist und weiß, wo sich High- und Low-Byte befinden.

4.1.1 Nullzeiger

Zeiger können den Wert `NULL` haben. Solche Zeiger heißen Nullzeiger. *Sie weisen auf kein Objekt!*. Funktionen, die einen Zeiger auf ein Objekt zurückgeben, können Nullzeiger verwenden, um anzuzeigen, daß die erwünschte Aktion nicht erfolgreich durchgeführt werden konnte (dies tut z.B. `fopen`).

4.1.2 Der Adreßoperator oder Referenzoperator

Ist *object* die Bezeichnung eines Objekts vom Typ `T`, so ist `&`*object* ein Verweis auf dieses Objekt. Somit ist `&`*object* ein Wert, der einer Variablen vom Typ `T *` zugewiesen werden kann.

Beispiel:

```
int x=7;
int *p;
p=&x;
```

Eine typische Anwendung des Adreßoperators ist die Parameterübergabe an Funktionen, beispielsweise muß man der Funktion `scanf` nicht die Variablennamen als Argumente geben, sondern entsprechende Verweise auf die Variablen. Die Gründe dafür werden wir später erklären.

4.1.3 Der Dereferenzoperator

Ist *p* ein Objekt vom Typ "Zeiger auf `T`", so bezeichnet `*p` das referenzierte Objekt.

Beispiel 1:

```
int x=7;
int *p;
p=&x;      /* jetzt ist *p dasselbe Objekt wie x und hat den Wert 7 */
*p=3;      /* jetzt hat x den Wert 3 */
```

Beispiel 2:

```
int x=7;
int *p;
int **q;   /* q ist vom Typ "Zeiger auf den Typ 'Zeiger auf int'" */
p=&x;
q=&p;
**q=3;     /* jetzt hat x den Wert 3 */
```

Achtung! Bevor man `*p` einen Wert zuweisen darf, muß erst `p` einen Wert haben, sonst weiß der Compiler ja nicht, welchem Objekt ein Wert zugewiesen werden soll. In Beispiel 1 haben wir es richtig gemacht.

Fehlerhaftes Beispiel:

```
int *p;
*p=3;     /* Fehler! Weil p noch keinen Wert hat. */
```

4.2 Arrays

Ein **Arraytyp** wird von einem beliebigen Datentyp `T` abgeleitet. Ein **Objekt vom Typ "Array von `T`"** ist eine Folge von Objekten vom Typ `T`, die im Arbeitsspeicher unmittelbar hintereinander abgelegt sind. Diese Objekte heißen die **Elemente des Arrays**. Ihre Anzahl heißt die **Länge des Arrays**; sie wird bei der Definition des Arrays fest vorgegeben und kann während der Laufzeit des Programms nicht geändert werden. Der **Wert eines Arrays** ist die Folge der Werte der Elemente. Der Wertebereich des Typs "Array von `T`" ist also $W_{\mathtt{T}}^{Länge}$, wobei $W_{\mathtt{T}}$ der Wertebereich des Typs `T` ist. Allerdings gibt es dabei eine Einschränkung; es ist nämlich zulässig, daß während des Programmlaufs Elemente eines Arrays nicht initialisiert sind, also noch keinen Wert zugewiesen bekommen haben.

Die Syntax der Deklaration von Variablen vom Typ "Array von `T`" geben wir hier nicht vollständig an, sondern nur für die wichtigsten Fälle. Wir unterschlagen im wesentlichen die Fälle, in denen `T` ein Funktionstyp oder selbst wieder ein Arraytyp ist.

4.2.1 Die Deklaration einer Variablen vom Typ "Array von `T`", wenn `T` ein sogenannter *skalarer Typ* ist d.h. ein arithmetischer Typ oder ein Zeigertyp, hat die Gestalt:

`T` *arrayname*`[`*constant_expression*`];`

wobei *constant_expression* ein konstanter Ausdruck von Ganzzahltyp ist, der die Länge des Arrays angibt. Diese Form ist sogar eine Definition der Variablen, denn sie bewirkt, daß der entsprechende Speicherplatz (=*constant_expression*·`sizeof(T)` Bytes) reserviert wird. Eine reine Deklaration ohne Angabe der Arraylänge kann nur bei gleichzeitiger Initialsierung (siehe unten) verwendet werden oder in der Form `extern T` *arrayname*`[];` mit dem Zusatz `extern`, mit dem angezeigt wird, dass die Definition des Arrays in einer anderen Datei steht (siehe Kapitel 7).

Beispiele:

```
int vektor[10];     /* vektor ist ein Array aus 10 int-Zahlen */
double *a[10];      /* a ist ein Array aus 10 Zeigern auf double */
```

Bemerkung: `double (*a)[10]` dagegen erklärt a als Zeiger auf ein Array von 10 `double`-Zahlen. Solche Fälle wollen wir zunächst nicht betrachten.

4.2.2 Initialisierung globaler Variablen vom Arraytyp

Eine Variable vom Typ "Array von `T`", die global d.h. außerhalb jeder Funktion deklariert wird, kann gleichzeitig mit der Deklaration initialisiert werden (was impliziert, daß es sich dann stets um eine Definition handelt). Dies geschieht folgendermaßen:

`T` *arrayname*`[`$arraylaenge_{opt}$`] = {`*initialisierungsliste*`};`

Dabei sind *arraylaenge* ein konstanter Ausdruck von Ganzzahltyp, der die Länge des Arrays angibt, aber auch ganz entfallen kann, und *initialisierungsliste* eine durch Kommata getrennte Folge von konstanten Ausdrücken vom Typ `T`.

Bedeutung: Den Elementen des Arrays werden der Reihe nach die Werte in der Initialisierungsliste zugewiesen. Dabei gilt: Ist die Länge des Arrays größer als die Anzahl der Ausdrücke in der Initialisierungsliste, so wird den restlichen Elementen der Wert

0 zugewiesen. Ist die Länge des Arrays in der Deklaration nicht festgelegt, so wird sie automatisch gleich der Anzahl der Ausdrücke in der Initialisierungsliste gesetzt.

Beispiele:

```
int a[] = {1,2,3};   /* a hat die Laenge 3 */
int b[4] = {1,2,3};  /* b hat die Laenge 4; das 4-te Element ist 0 */
```

4.2.3 Der Subscript-Operator

Die Elemente eines Arrays `a` der Länge L werden der Reihe nach mit den Zahlen $0, \ldots, L-1$ durchnummeriert, d.h. das erste Element erhält den Index 0 und das letzte den Index $L-1$. Ist *expression* ein Ausdruck von Ganzzahltyp, dessen Wert n in $\{0, \ldots, L-1\}$ liegt, so bezeichnet `a[`*expression*`]` das $(n+1)$-te Element des Arrays. Diese Bezeichnung kann wie jeder andere Objektbezeichner in Ausdrücken und Anweisungen verwendet werden. `[ ]` heißt Subscript-Operator.

Vorsicht! In C-Programmen wird während der Laufzeit nicht nachgeprüft, ob der Index in einem Subscript-Operator im zulässigen Bereich $\{0, \ldots, L-1\}$ liegt! Es wird einfach an der entsprechenden Stelle in den Speicher gegriffen. Das kann zu überraschenden Effekten führen! Manchmal erhält man auch Fehlermeldungen vom Betriebssystem, weil der Adreßraum des Prozesses verlassen wird. Oft liest oder schreibt man aber unbeabsichtigt in den Speicherplatz anderer Objekte.

Beispiel:

```
int a[10];
int i;
for(i=0;i<10;i++) a[i] = i;
for(i=1;i<10;i++) a[i-1] = a[i];  /* verschiebt die Werte nach links */
```

4.3 Strings

4.3.1 Definition *Ein **String** ist eine Folge von Objekten vom Typ* char*, die im Speicher unmittelbar hintereinander abgelegt sind und für deren Werte gilt: Der Wert des letzten Objekts (das mit der größten Adresse) ist 0; alle anderen haben einen Wert ungleich 0. Mit 0 ist hier nicht die Ziffer* 0 *gemeint, sondern der Zahlenwert 0 aus dem Wertebereich des Typs* char*, den man in C auch mit* '\0' *bezeichnet.*

*Ein **Zeiger auf einen String** ist ein Zeiger vom Typ "Zeiger auf* char*", der auf das erste Objekt (das mit der niedrigsten Adresse) weist.*

*Die **Länge eines Strings** ist die Anzahl der Objekte vor dem mit Wert 0, und der **Wert eines Strings** ist die Folge der Werte der Objekte vor dem mit Wert 0.*

Strings dienen dazu, Textstücke oder besser gesagt ASCII-Zeichenketten zu speichern. Sie können in Objekten vom Typ "Array von `char`" abgelegt werden. Dafür gibt es eine spezielle Art der Initialisierung solcher Arrays, indem man bei der Deklaration als Wert eine ASCII-Zeichenkette zuweist, die in Anführungszeichen `"` eingeschlossen ist.

Beispiel:

```
char s[10] = "abc";
```

Dadurch wird `s` als Variable vom Typ "Array von char" der Länge 10 definiert und außerdem den ersten drei Elementen der Reihe nach als Werte die Zeichen `a,b,c` zugewiesen und den restlichen der Wert 0 (nicht die Ziffer `0`). Bei Arrays ohne Längenangabe gibt es eine Besonderheit:

```
char s[] = "abc";
```

definiert `s` als Variable vom Typ "Array von char" der **Länge 4**, weist den ersten drei Elementen der Reihe nach die Werte `'a','b','c'` zu und dem vierten den Wert 0.

Beispiel:

```
char s[] = "hallo";
char *str;
str = &s[0];    /* jetzt ist str ein Zeiger auf den String */
```

Man kann Strings nicht nur mit Hilfe von Arrays, sondern auch anders mit Hilfe von Speicherallozierung erzeugen; das werden wir aber erst später besprechen. In der Standardbibliothek von C gibt es etliche Funktionen zur Manipulation von Strings wie Kopieren (`strcpy`), Vergleichen (`strcmp`), Aneinanderhängen (`strcat`) oder Ausgabe von Zeichen in einen String (`sprintf`) und Lesen von Zeichen aus einem String (`sscanf`). Die Funktion `strlen` gibt die Länge eines Strings zurück. Außerdem gibt es Funktionen, um Strings, die Zahlen darstellen, in Werte eines arithmetischen Typs umzuwandeln, z.B. `atoi, atof, strtod, strtol, strtoul`. Dabei soll die Namensgebung eine Gedächtnisstütze sein; `a` oder `str` steht für "String", to für "Umwandlung nach" und `i,f,d,l,ul` für `int,floating point = double, double, long int, unsigned long int`. Die Funktion `atof` wandelt in einen `double`-Wert, nicht in einen `float`-Wert.

4.4 Eingabeparameter für `main`

Die Funktion `main` kann auf zwei Arten definiert werden:

1. ohne Eingabeparameter:

`int main(void)` *anweisungsblock*

Manchmal findet man die alte K&R-Schreibweise

`main()` *anweisungsblock*

2. mit zwei Eingabeparametern:

`int main(int argc, char *argv[])` *anweisungsblock*

Die beiden Parameter dürfen auch andere Namen haben; es ist aber allgemein üblich, sie so zu nennen (argument counter und argument values). Die entsprechenden Argumente werden von der Systemumgebung (z.B. einer Shell) bei Programmstart an die Funktion `main` übergeben. Wie das passiert ist, ist in der ANSI-Norm nicht näher festgelegt. Nur einige wenige Eigenschaften müssen erfüllt sein. Dazu gehören:

- Der Wert von `argc` ist nicht negativ.
- `argv[argc]` ist ein Nullzeiger.

- Ist der Wert von `argc` positiv, so sind `argv[0]`,...,`argv[argc-1]` Zeiger auf Strings, und der Wert des Strings, auf den `argv[0]` zeigt, ist der Programmname.

4.4.1 Eine typische Anwendung von `argc` und `argv`

Ein Programm soll von der Kommandozeile einer Shell aufgerufen werden, wobei gleich einige Optionen oder Parameter in der Kommandozeile übergeben werden sollen. Man denke an Beispiele wie `ls -l` oder `gcc prog.c -o prog`.

Auf den meisten Rechnerplattformen liegt folgende Situation vor:

- Der Wert von `argc` ist gleich der Anzahl der durch Leerzeichen getrennten Wörter in der Kommandozeile. Bei Eingabe von `gcc prog.c -o prog` z.B. ist der Wert von `argc` gleich 4.
- Die Länge von `argv[]` ist gleich dem Wert von `argc + 1`. Die in `argv[]` gespeicherten Zeiger weisen auf Strings, deren Werte der Reihe nach die Wörter in der Kommandozeile sind. `argv[argc]` ist ein Nullzeiger.

Beispiel:
Bei der Kommandozeile `gcc prog.c -o prog` weist `argv[0]` auf den String `gcc`, `argv[1]` auf `prog.c` , `argv[2]` auf `-o` und `argv[3]` auf `prog`, und `argv[4]` ist ein Nullzeiger.

4.5 Zeigerarithmetik

In C darf man mit Zeigern einige Rechenoperationen durchführen. Dies ist ein sehr mächtiges Werkzeug, jedoch bei unvorsichtigem Gebrauch auch eine reiche Fehlerquelle und daher auch häufig ein Punkt der Kritik an C. In der Konzeption manch anderer Sprachen (wie JAVA) wurde daher bewußt auf solche Möglichkeiten verzichtet. Wir erläutern hier nur die wichtigsten Operationen.

`p` sei ein Objekt vom Typ "Zeiger auf `T`". Das referenzierte Objekt `*p` belege einen Speicherbereich mit der Anfangsadresse α. Außerdem sei *expr* ein Ausdruck von Ganzzahltyp mit Wert j.

4.5.1 Addition und Subtraktion

`p+(`*expr*`)` bzw. `p-(`*expr*`)` ist ein Verweis auf ein Objekt vom Typ `T`, dessen Speicherbereich die Anfangsadresse $\alpha + j \cdot$`sizeof(T)` bzw. $\alpha - j \cdot$`sizeof(T)` hat. `*(p+(`*expr*`))` bzw. `*(p-(`*expr*`))` bezeichnet dieses Objekt.

4.5.2 Der Subskript-Operator

Der Ausdruck `p[`*expr*`]` wird vom Compiler durch den Ausdruck `*(p+(`*expr*`))` ersetzt; er bezeichnet also ebenfalls das Objekt vom Typ `T` mit Adresse $\alpha + j \cdot$`sizeof(T)`.

Beispiel 1:

```
char a[5]="hallo";
char *zeiger;
zeiger=a;
zeiger[1]='e';    /* jetzt hat a den Wert "hello" */
```

Den gleichen Effekt erzielt das folgende Programmfragment

```
char a[5]="hallo";
char *zeiger;
zeiger=&a[4];
zeiger[-3]='e';
```

Beispiel 2:

Manchmal ist es lästig, daß die Indizierung von Arrays immer bei 0 beginnt. Will man z.B. eine zu Null symmetrische Indizierungmenge $\{-n, \ldots, n\}$, so kann man dies mit Hilfe von Zeigern erreichen.

```
double Array[2*n+1];
double *SymFeld;
SymFeld=&Array[n];   /* SymFeld ist symmetrisch indizierbar */
```

4.5.3 Warnung Weder bei der Compilierung noch während der Laufzeit des Programms wird überprüft, ob man mit den Zeigern auf Objekte zugreift, die vorher definiert wurden. Es wird einfach an der angegebenen Adresse in den Speicher gegriffen und das, was dort steht, als ein Objekt von entsprechenden Typ interpretiert. Wenn man Glück hat, wird auf eine Adresse zugegriffen, die dem Prozess vom Betriebssystem nicht zugeteilt wurde; dann wird der Programmlauf vom Betriebssystem mit einer Fehlermeldung abgebrochen. Wenn man Pech hat, wird aber auf zulässige Adressen zugegriffen, und was dann passiert, hängt von den Daten ab, die zufälligerweise dort im Speicher stehen. Daher können solche Fehler oft unbemerkt bleiben oder zu kuriosen, datenabhängigen Effekten führen. Solche Fehler sind oft schwer zu finden. Daher sollte schon beim Schreiben des Programms sehr präzise darauf achten, daß solche Bereichsüberschreitungen in keiner Situation auftreten können.

4.6 Vergleich von Arrays und Zeigern

In C gibt es eine enge Beziehung zwischen Array und Zeigern; es gibt aber Unterschiede, die man stets beachten sollte.

Eine Variable vom Typ "Array von `T`" wird vollständig beschrieben durch

- den Typ `T` der Elemente
- die Anzahl L der Elemente
- die Adresse des ersten Elements
- den Variablennamen.

Innerhalb von Ausdrücken wird der Variablenname bei der Auswertung durch einen Zeiger, der auf das erste Element zeigt, ersetzt. Deshalb sind die beiden folgenden Programmfragmente äquivalent.

```
int a[10];          int a[10];
int *p;             int *p;
p=&a[0];            p=a;
```

`a` ist aber kein Bezeichner für einen Zeiger; Anweisungen wie `a=p;` sind falsch! Die Adresse des ersten Elementes von `a` kann nicht verändert werden.

Arrays haben einen ganz großen Nachteil: Ihre Länge muß bereits im Quellcode festgelegt werden; sie kann nicht während des Programmlaufs geändert werden. Bereits beim Compilieren wird für Arrays Speicherplatz reserviert; es ist nicht möglich, dies erst während des Programmlaufs zu tun. Dies hat gravierende Nachteile in Situationen, wo die maximale Größe der zu verarbeitenden Datenmenge nicht a priori begrenzt ist. Möchte man z.B. ein Programm schreiben, das Bilder verarbeitet, so ist es meist am einfachsten, das gesamte Bild in ein Array einzulesen. Durch die Arraygröße ist dann aber von vorn herein die maximale Bildgröße, die man verarbeiten kann, begrenzt.

Einen Ausweg schaffen Funktionen, mit denen man während des Programmlaufs vom Betriebssystem Speicher anfordern kann (memory allocation). Diese Funktionen geben einen Zeiger auf den Beginn des zur Verfügung gestellten Speicherbereichs zurück. Mit Hilfe des Subskript-Operators kann man dann auf den Speicherbereich zugreifen. Dazu später mehr.

Kapitel 5

Funktionen

ANSI-C führt eine Syntax für die Deklarationen und Definitionen von Funktionen ein, die sich erheblich von der älterer C-Versionen (wie Kernighan&Ritchie) unterscheidet. Wir geben sie hier nicht in voller Allgemeinheit an, sondern beschränken uns auf die wichtigsten Fälle.

5.1 Prototypen

Ein Prototyp ist eine Deklaration einer Funktion von folgender Gestalt

returntype identifier`(`*parameter_type_list*`);`

Dabei ist *identifier* der Name einer Funktion, *returntype* der Typ des Rückgabewerts der Funktion und *parameter_type_list* eine Beschreibung der Eingabeparameter der Funktion, die weiter unten erläutert wird.

Der Rückgabetyp *returntype* darf kein Arraytyp oder Funktionstyp sein. Meist ist er ein arithmetischer Typ, ein Zeigertyp oder das Schlüsselwort `void`. Die Angabe von `void` als Rückgabetyp bedeutet, daß die Funktion keinen Wert zurückgibt.

parameter_type_list kann ebenfalls das Schlüsselwort `void` sein, was bedeutet, daß die Funktion keine Eingabeparameter hat. Ansonsten ist *parameter_type_list* eine endliche Folge von Deklarationen der Eingabeparameter, die durch Kommas getrennt werden. Dabei gibt es zwei Möglichkeiten:

- Es werden nur die Typen der Eingabeparameter angegeben, z.B.

  ```
  double Dreiecksflaeche(double,double);
  ```

- Es werden zusätzlich Bezeichner angegeben, z.B.

  ```
  double Dreiecksflaeche(double seite, double hoehe);
  ```

  Mit diesen Bezeichnern kann man die Bedeutung der Parameter angeben, ansonsten spielen sie aber keine Rolle.

Eine Besonderheit liegt bei Funktionen vor, deren Parameteranzahl variabel ist, z.B. `printf` mit dem Prototyp `int printf( const char *format, ...);`

5.2 Funktionstypen

Die Angabe eines Typs für den Rückgabewert einer Funktion und der Typen der Eingabeparameter bezeichnet man als Funktionstyp. Als Bezeichner für Funktionstypen verwendet man

returntype (*parameter_type_list*)

mit den Bezeichnungen von 5.1, wobei für *parameter_type_list* die Form ohne Bezeichner der Parameter gewählt wird.

Beispiele:

```
double (double)
double (double, double)
int *(void)
```

5.3 Funktionsdefinitionen

Die gebräuchlichste Gestalt von Funktionsdefinitionen ist folgende

returntype identifier(*parameter_type_list*) *anweisungsblock*

Der Anfang wird also wie beim Prototyp in 5.1 gebildet (ohne Semikolon!), allerdings mit der Einschränkung, daß für die Parameter Bezeichner angegeben werden müssen. Denn unter diesen Namen können die Eingabeparameter im sogenannten Funktionsrumpf, der aus *anweisungsblock* besteht, verwendet werden. Selbstverständlich müssen die Typen des Rückgabewerts und der Eingabeparameter mit den im Prototyp angegebenen übereinstimmen.

Beispiel:

```
double quadrat( double );       /* Prototyp */

double quadrat( double x ) {    /* Definition */
   return x*x;
}
```

Achtung! Die Definition einer Funktion darf nicht innerhalb einer anderen Funktion stehen.

5.3.1 Zulässige Typen für die Eingabeparameter

Meist verwendet man Parameter von arithmetischem Typ oder Zeigertyp. Es sind jedoch auch Arraytypen zugelassen, allerdings gibt es dabei einige Besonderheiten, die wir weiter unten erläutern. Laut ANSI-Standard sind - grob gesagt - alle Typen zugelassen, deren Werte man durch eine Zuweisungsanweisung einem Objekt zuweisen kann. Bis auf Arraytypen und Funktionstypen trifft das für alle Typen, die wir bisher betrachtet haben, zu (ANSI 3.3.2.2, Constraints, letzter Satz).

5.4 Funktionsaufruf und Parameterübergabe

5.4.1 Funktionsaufrufe haben die Gestalt

function_name (*expression_list*)

function_name ist der Name einer Funktion, wie er im Prototyp und der Definition festgelegt ist. *expression_list* ist eine durch Kommas getrennte Folge von Ausdrücken, die als die **Argumente** des Funktionsaufrufs bezeichnet werden. Ihre Anzahl und ihre Typen müssen mit denen der Eingabeparameter in der Deklaration der Funktion bis auf die bei Zuweisungen übliche Typkonversion übereinstimmen.

Bedeutung: Ein Funktionsaufruf ist ein Ausdruck, der wie jeder andere Ausdruck verwendet werden kann. Die Auswertung geschieht folgendermaßen. Zunächst wird für jeden Eingabeparameter der Funktion ein neues Objekt mit demselben Typ und Namen wie in der Funktionsdefinition erzeugt und mit dem Wert des entsprechenden Arguments des Funktionsaufrufes initialisiert. Dann springt die Programmausführung an den Anfang des Codes, den der Compiler aus dem in der Definition angegebenen Funktionsrumpf erzeugt hat, und führt ihn aus, bis eine `return`-Anweisung oder das Ende des Funktionsrumpfes erreicht wird. Falls die `return`-Anweisung einen Ausdruck enthält, wird sein Wert bestimmt und dem Funktionsaufruf als Wert zugewiesen. Dann werden die den Eingabeparametern entsprechenden Objekte und eventuell weitere Objekte, die bei der Ausführung des Funktionsaufrufes erzeugt wurden, gelöscht. Sie existieren also nur während der Auswertung des Funktionsaufrufes (außer sie sind als `static` deklariert); daher heißen sie **lokale Variablen**. Umgekehrt sind Objekte gleichen Namens, die vor Beginn der Auswertung des Funktionsaufrufs eventuell existieren, während der Auswertung des Funktionsaufrufs nicht bekannt und nicht ansprechbar. Insbesondere macht es nichts aus, wenn die Namen der Eingabeparameter in der Definition der Funktion auch als Namen anderer Variablen außerhalb des Funktionsrumpfes vorkommen; für die Funktionsauswertung werden neue Objekte gleichen Namens erzeugt und die bisherigen versteckt. Dadurch sind rekursive Funktionsaufrufe und rekursive Funktionsdefinitionen möglich.

5.4.2 Besonderheiten der Parameterübergabe
In C werden wie oben beschrieben bei einem Funktionsaufruf die Werte der Argumente neu geschaffenen Objekten zugewiesen. Deshalb bezeichnet man die Parameterübergabe als **call by value**.

Beispiel:

```
int quadrat(int x) {
   x=x*x;
   return x;
}

int main(void) {
   int x=3;
   printf("%d\n",quadrat(x));
   printf("%d\n",x),
   return 0;
}
```

Dieses Programm gibt erst eine `9` und dann eine `3` aus; denn die in `main` definierte Variable `x` wird durch Aufruf von `quadrat(x)` nicht geändert!

Manchmal möchte man aber durch einen Funktionsaufruf die Werte von Variablen außerhalb der Funktion verändern. Das ginge, wenn zur Parameterübergabe die Methode **call by reference** verwendet würde d.h. wenn nicht der Wert des Arguments, sondern ein Verweis darauf übergeben würde. Dies kann man bei C mit Hilfe des Adreßoperators `&` leicht simulieren.

Beispiel:

```
void quadrat(int *p) {
   *p = (*p)*(*p);
}

int main(void) {
   int x=3;
   quadrat(&x);
   printf("%d",x);
return 0;
}
```

Achtung! In C gibt es bei der Parameterübergabe eine Ausnahme von der call-by-value-Methode! Und zwar werden **Parameter von Arraytyp nicht mit call-by-value übergeben**. Stattdessen wird ein Verweis auf das erste Element des Arrays übergeben. In einer Funktionsdefinition sind die Parameterdeklarationen `T a[n]` und `T a[]` und `T *a` völlig äquivalent. In jedem Fall wird der Eingabeparameter `a` vom Compiler durch einen Zeiger auf `T` ersetzt und dann wird call-by-value gemacht d.h. der Zeiger erhält bei einem Funktionsaufruf als Wert einen Verweis auf das erste Element des Arrays, das als Argument übergeben wird. Nach den Regeln für den Subscriptoperator werden die Elemente des Argumentarrays auch mit Hilfe dieses Zeigers korrekt angesprochen.

5.5 Rekursive Funktionsdefinitionen und Funktionsaufrufe

Infolge der in 5.4 beschriebenen Auswertung eines Funktionsaufrufs ist es möglich,

a) daß im Argument eines Funktionsaufrufs wieder ein Aufruf derselben Funktion vorkommt,

b) daß im Funktionsrumpf der Definition einer Funktion ein Aufruf eben dieser Funktion vorkommt (dann muß allerdings eine Begrenzung der Schachtelungstiefe eingebaut sein).

Beispiel zu a):

```
int doppelt(int x) {
   return 2*x;
}
```

```
int main(void) {
   int x=1;
   printf("%d", doppelt(doppelt(doppelt(x))));
   return 0;
}
```

Die Ausgabe dieses Programms ist `8`.

Beispiel zu b):

```
int potenz(int n) {
   if(n==0) return 1;
   else return 2*potenz(n-1);
}

int main(void) {
   int n=3;
   printf("%d", potenz(n));
   return 0;
}
```

Die Ausgabe dieses Programms ist `8`.

5.6 Zeiger auf Funktionen

Eine Funktion wird vom Compiler in ein Stück Maschinensprachecode übersetzt, das in einem Speicherbereich abgelegt ist. Darin gibt es eine bestimmte Adresse, zu der die Programmausführung bei einem Funktionsaufruf hinspringt. Diese Einsprungadresse kann einem Zeiger zugewiesen werden, wenn er als ein Zeiger auf eine Funktion von entsprechendem Typ deklariert ist. Ein Zeiger `funcptr` auf eine Funktion vom Typ *returntype*(*parameter_list*) wird folgendermaßen deklariert

returntype `(*funcptr)` (*parameter_list*)

Der Aufruf der referenzierten Funktion erfolgt dann durch den Ausdruck

`(*funcptr)`(*parameter_list*).

Beispiel:

```
int potenz(int n) {
   if(n==0) return 1;
   else return 2*potenz(n-1);
}

int main(void) {
   int n=3;
   int (*potzeiger)(int n);
   potzeiger = potenz;
   printf("%d\n", (*potzeiger)(n));
   return 0;
}
```

Funktionszeiger spielen eine wichtige Rolle, wenn man Algorithmen so implementieren will, dass sie mit Daten unterschiedlichen Typs funktionieren. Dann kann man für eine im Algorithmus verwendete Operation je nach Datentyp eine andere Funktion verwenden, indem man einfach die Operation über einen Funktionszeiger aufruft, der auf die jeweils richtige Funktion weist. In der Mathematik pflegt man seit Jahrhunderten z.B. die Addition ganz unterschiedlicher Objekte wie Zahlen, Vektoren, Matrizen, Gruppenelementen o.ä. immer mit demselben Zeichen + zu bezeichen, obwohl sich die Operationen im Einzelnen deutlich unterscheiden. Diese Vorgehensweise ist ein wichtiger Bestandteil eines Programmierstils, den man objektorientiert nennt. Programmiersprachen wie C++ und Java enthalten spezielle Sprachkonstrukte, die diese Denkweise unterstützen und den expliziten Umgang mit Zeigern vermeiden. Seltsamerweise ist aber in Java das eben erwähnte Überladen des +-Operators nicht möglich.

Kapitel 6

Speicherverwaltung und Structs

Bisher können wir Objekte nur dadurch erzeugen, daß wir im Quelltext des Programms entsprechende Definitionen hinschreiben. Das ist für viele Anwendungen nicht sehr geeignet. Denn vielfach ergibt sich erst während des Programmlaufs in Abhängigkeit von eingegebenen Daten die Notwendigkeit, neue Objekte zu erzeugen und nicht mehr benötigte zu löschen. Insbesondere bei Programmen, die sehr lange laufen wie Betriebssysteme, Dämone (Hintergrundprozesse wie inetd, sendmail), ist es unbedingt erforderlich, nicht mehr benötigte Objekte zu vernichten, um den von ihnen belegten Speicherplatz wieder verwenden zu können.

6.1 Funktionen zur Speicherverwaltung

Die Prototypen der folgenden Funktionen stehen in der Headerdatei `stdlib.h`, die mit einer `#include`-Anweisung in jedes Programm eingelesen werden muß, in dem diese Funktionen benutzt werden.

Die Eingabeparameter sind teilweise vom Typ `size_t`, der in `stddef.h` als ein `unsigned int`-Typ definiert ist. Die genaue Definition ist vom Rechnersystem abhängig.

6.1.1 Die `malloc`-Funktion

Prototyp:

```
void *malloc(size_t size);
```

Beschreibung: Durch Aufruf von `malloc` wird vom Betriebssystemkern ein zusammenhängender (bisher nicht verwendeter) Speicherbereich von `size` Bytes reserviert (malloc ist eine Abkürzung von memory allocation). Der Rückgabewert von `malloc` ist ein Zeiger, der auf das erste Byte (das mit der niedrigsten Adresse) des reservierten Speicherbereichs weist. Er weist jedoch auf kein Objekt von einem speziellen Typ; deshalb ist er als Zeiger auf `void` deklariert. Er kann durch eine `cast`-Operation in einen Zeiger auf irgendeinen anderen Typ verwandelt werden. Falls der gewünschte Speicherbereich von `malloc` nicht reserviert werden konnte, gibt `malloc` einen Null-

zeiger zurück. Daher kann man einen Fehler beim Reservieren von Speicher daran erkennen, daß der Ausdruck `malloc(size)==NULL` den Wert 1 hat. Achtung! `malloc` schreibt keine Initialwerte in den reservierten Speicherbereich.

Beispiel: Reservierung von Speicher für 1000 Objekte vom Typ `int`

```
int *p;
p = (int *)malloc(1000*sizeof(int));
if (p==NULL) printf("Fehler");
```

6.1.2 Die `calloc`-Funktion

Prototyp:

```
void *calloc( size_t anzahl, size_t size );
```

Beschreibung: `calloc` reserviert einen Speicherbereich für ein Array von `anzahl` Objekten, deren Größe jeweils `size` Bytes ist, also insgesamt `anzahl * size` Bytes. Außerdem wird in alle diese Bytes der Wert 0 geschrieben. Der Rückgabewert ist wie bei `malloc` erklärt.

6.1.3 Die `realloc`-Funktion

Prototyp:

```
void *realloc( void *zeiger, size_t size );
```

Beschreibung: `zeiger` muß ein Zeiger sein, der vorher bei einem Aufruf von `malloc`, `calloc` oder `realloc` zurückgegeben wurde und nicht mit der Funktion `free` (siehe unten) geändert wurde. `realloc` ändert die Größe des Speicherbereichs, auf den `zeiger` weist, auf `size` Bytes und erhält dabei den bisherigen Inhalt, soweit die neue Größe es zuläßt. Dabei kann es passieren, daß ein völlig anderer Speicherbereich reserviert und der bisherige Inhalt dorthin kopiert wird. Der Rückgabewert ist ein Zeiger auf den neuen Bereich oder ein Nullzeiger, wenn der gewünschte Bereich nicht reserviert werden konnte.

6.1.4 Die `free`-Funktion

Prototyp:

```
void free( void *zeiger );
```

Beschreibung: `zeiger` muß ein Zeiger auf einen Speicherbereich sein, der mit `malloc`, `calloc` oder `realloc` reserviert wurde. Dieser Speicherbereich wird dann durch den Aufruf von `free` wieder freigegeben. Ist `zeiger` ein Nullzeiger, so führt `free` keine Aktion aus. In allen anderen Fällen ist das Verhalten nicht definiert. **Achtung!** `free` gibt keine Erfolgs- oder Fehlermeldung zurück! Man muß selbst für die korrekten Zeiger sorgen!

6.1.5 Speichermüllbeseitigung (garbage collection)

Wird häufig Speicher alloziert und teilweise wieder freigegeben, so kann es passieren, daß der Arbeitspeicher in viele kleine Bereiche fragmentiert wird, die abwechselnd belegt und frei sind. Dann kann man keinen längeren Speicherbereich mehr reservieren, obwohl noch genügend viele Bytes frei sind. Man muß dann eine sog. Garbage Collection durchführen, d.h. die belegten Speicherbereiche umkopieren, so daß sie lückenlos hintereinander liegen. Dies muß man in C selbst explizit programmieren; C hat kei-

ne eingebaute Garbage Collection. Zur Unterstützung gibt es einige Funktionen wie `memcpy` und `memmove`. Trotzdem ist dies mühsam und unbefriedigend. Manche andere Programmiersprachen wie z.B. Java haben deshalb eine eingebaute Garbage Collection. Das ist allerdings auch nicht immer so vorteilhaft, wie man zunächst denkt; denn die Garbage Collection kann durchaus zu merklichen Zeitverzögerungen (bis in den Sekundenbereich) führen, was in manchen Phasen eines Programmlaufes sehr störend sein kann. Wenn man die Garbage Collection selbst durchführen muß, hat man immerhin den Vorteil, einen geeigneten Zeitpunkt dafür zu wählen oder sie abzubrechen.

6.2 Structs

Ein Array bietet die Möglichkeit, mehrere Objekte desselben Typs zu einer Einheit zusammenzufassen. In vielen Anwendungen ergibt sich jedoch auf natürliche Weise der Wunsch, Objekte unterschiedlichen Typs zu einer Einheit zusammenzufassen, beispielsweise die Einträge auf einer Karteikarte. Dafür gibt es in in den meisten Programmiersprachen spezielle Datentypen; in C heißen sie structs, in anderen Sprachen oft records (auf deutsch vielleicht Datensätze).

6.2.1 Definition eines `struct`-Typs

Syntax: `struct` *identifier*`{` *declaration_list*`};`

Bedeutung: *identifier* ist ein Bezeichner, der als Etikett (tag) des definierten `struct`-Typs dient; der Name des `struct`-Typs ist `struct` *identifier* (wie bei Aufzählungstypen). *declaration_list* ist eine Folge von Variablendefinitionen. Diese Variablen bilden die Komponenten des `struct`-Typs.

6.2.1.1 Beispiel:

```
struct Karteikarte {
   char          Name[100];
   unsigned long Matrikelnummer;
   char          Adresse[255];
   double        Vordiplomnote;
};
```

Ein Struct darf als Komponenten wieder Structs haben.

6.2.1.2 Beispiel:

```
struct adresse {
   char Strasse[50];
   char Wohnort[50];
   char  PLZ[6];
};
struct Karteikarte {
   char           Name[100];
   unsigned long  Matrikelnummer;
   struct adresse Adresse;
   double         Vordiplomnote;
};
```

6.2.2 Deklaration einer Variablen von `struct`-Typ

Die übliche Syntax für Variablendeklarationen ist auch hier gültig:

`struct` *etikett variablenname* ;

Dadurch wird eine Variable mit Namen *variablenname* deklariert, deren Typ ein Struct mit Etikett *etikett* ist.

6.2.3 Simultane Deklaration eines `struct`-Typs und einer Variablen gleichen Typs

Syntax: `struct` *etikett* {*declaration_list*} *variablenname*;

oder

`struct` {*declaration_list*} *variablenname*;

Bedeutung: Im ersten Fall wird ein `struct`-Typ mit Etikett *etikett* und den durch *declaration_list* festgelegten Komponenten und zugleich eine Variable dieses Typs mit Namen *variablenname* deklariert. Im zweiten Fall ebenso, nur hat der `struct`-Typ kein Etikett (und läßt sich daher nicht in weiteren Deklarationen verwenden).

6.3 Der Punktoperator und der Pfeiloperator

variablenname sei eine Variable vom Typ `struct` *etikett*.

Mit Hilfe des **Punktoperators** kann man aus *variablenname* und den Komponentennamen, wie sie in der Deklaration von `struct` *etikett* vorkommen, Bezeichnungen für die Komponentenobjekte bilden:

variablenname . *componentenname*

Beispiel 1:

```
struct Karteikarte {
   char          Name[30];
   unsigned long Matrikelnummer;
   char          Adresse[255];
   double        Vordiplomnote;
}

struct Karteikarte student;
strcpy(student.Name,"Aurich");
student.Matrikelnummer = 123456789;
```

Beispiel 2:

```
struct punkt_3d {double x,y,z;} p;
p.x = 1;
p.y = 0;
p.z = 1.5;
```

Der Punktoperator läßt sich auch anwenden, wenn der Variablenname durch eine andere Bezeichnung des Objekts ersetzt wird.

Beispiel 3:

```
struct punkt_3d {double x,y,z;};
struct punkt_3d p, *q;
q = &p;
(*q).x = 1;
(*q).y = 0;
(*q).z = 1.5;
```

Die Klammern sind nötig, weil der Punktoperator eine höhere Priorität als * hat. Diese Art, die Komponenten zu bezeichnen, ist korrekt, aber unüblich. Meist verwendet man den **Pfeiloperator**:
Ist `q` ein Zeiger auf ein Objekt von `struct`-Typ und `component` der Name einer Komponente, so bezeichnet `q->component` ebenso wie `(*q).component` das Objekt, das in dieser Komponente gespeichert ist. Der Pfeil `->` besteht aus dem Minuszeichen und dem Größerzeichen.

Beispiel 4:

```
struct punkt_3d {double x,y,z;};
struct punkt_3d p, *q;
q = &p;
q->x = 1;
q->y = 0;
q->z = 1.5;
```

6.4 Structs in Ausdrücken

6.4.1 Zuweisungen

Im Gegensatz zu Arrays kann man in ANSI-C Objekten von `struct`-Typ als Ganzes den Wert eines Objekts von demselben Typ zuweisen wie in folgendem Beispiel:

```
struct punkt_3d { double x,y,z;};
struct punkt_3d p,q;
p.x = p.y = p.z = 0;
q = p;                  /* jetzt sind alle Komponenten von q gleich 0 */
```

6.4.2 Initialisierung

Variablen von `struct`-Typ können wie andere Variablen bei ihrer Definition initialisiert werden; die Syntax ist analog zu der Initialisierung von Arrays.

Beispiel: Wir verwenden den Typ `struct Karteikarte` aus 6.2.1.2.

```
struct Karteikarte prof = { "Mustermann",
                            123456789,
                            { "Musterstrasse 1",
                              "Musterdorf",
                              "40225"
                            },
                            1.7
};
```

6.4.3 Structs als Eingabeparameter von Funktionen

In ANSI-C werden Structs mit call-by-value übergeben (im Gegensatz zu Arrays!). Der Wert eines Arguments von `struct`-Typ wird als Ganzes der entsprechenden lokalen Variablen zugewiesen. Dieser Kopiervorgang kostet bei größeren Structs natürlich viel Zeit. Deshalb pflegt man meist den Eingabeparameter nichts als `struct`, sondern als Zeiger auf `struct` zu erklären.

6.4.4 Structs als Rückgabewert von Funktionen

Der Rückgabewert einer Funktion darf von `struct`-Typ sein. Meist wird man aber stattdessen einen Zeiger auf den `struct`-Typ wählen.

6.4.5 Bemerkung: Structs dürfen der Elementtyp eines Arrays sein, und umgekehrt kann ein Array als Komponente eines Struct autreten.

6.5 Structs und `typedef`

`typedef` und `struct`-Definitionen lassen sich auf eine etwas unübersichtliche Weise mischen. Am leichtesten versteht man dies an einem Beispiel.

```
struct punkt {double x,y;};
```

definiert einen `struct`-Typ mit Typnamen `struct punkt`. Der Bezeichner `punkt` ist das Etikett (tag) des Structs.

```
typedef struct punkt punkt_t;
```

definiert `punkt_t` als einen Typnamen, der zu `struct punkt` äquivalent ist. Sowohl

```
punkt_t p;
```

als auch

```
struct punkt p;
```

definieren `p` als Variable vom Typ `struct punkt`.

Man kann auch `typedef` und die Definition eines Struct-Typs mischen, z.B. in folgender Weise

```
typedef struct {double x,y;} punkt_t;
```

Verwirrend wird es, wenn man für das Struct-Etikett, eine Struct-Komponente oder den neuen Typnamen denselben Bezeichner wählt, was möglich ist. Vermeiden Sie das!

6.6 Erzeugung neuer Structobjekte

Um während des Programmlaufs neue Objekte erzeugen zu können, die bisher noch nicht existiert haben, fordert z.B. mit `malloc` Speicherplatz von der Größe des zu schaffenden Objekts an und lässt einen Zeiger darauf weisen, über den man dann das Objekt scheiben und lesen kann.

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Karteikarte {
   char          Name[30];
   unsigned long Matrikelnummer;
   char          Adresse[255];
   double        Vordiplomnote;
};

typedef struct Karteikarte karteikarte_t;

int main() {
   karteikarte_t *student;

   student = (karteikarte_t *)malloc(sizeof(karteikarte_t));
   if(NULL==student) { printf("Fehler\n"); return -1; }

   strcpy(student->Name,"Aurich");
   student->Matrikelnummer = 123456789;

   return 0;
}
```

Kapitel 7

Module und die Sichtbarkeit von Definitionen

7.1 Programmmodule

Größere Programme bestehen oft aus mehreren Quelldateien, die getrennt compiliert und dann zusammengebunden werden. Die einzelnen Quelldateien werden Programmmodule genannt. Jedes Modul ist eine Folge von Deklarationen und Definitionen von Variablen und Funktionen. Die Option `-c` sagt dem C-Compiler `gcc`, daß er nicht den Linker aufrufen soll. Besteht also ein Programm aus zwei Quelldateien `modul1.c` und `modul2.c`, so werden sie einzeln compiliert mit

```
gcc modul1.c -c -o modul1.o
gcc modul2.c -c -o modul2.o
```

Die erzeugten Programme `modul1.o` und `modul2.o` sind i.a. keine ausführbaren Programme. Sie müssen erst noch gebunden werden. Dies geht z.B. mit

```
gcc modul1.o modul2.o -o programmname
```

Bei Unix ist die Endung `.o`, bei DOS/Windows die Endung `.obj` üblich; sie steht für Objektcode.

7.1.1 Vorteile der Verwendung von Programmmodulen

- Übersichtliche Strukturierung größerer Programme.
- Bei Änderungen müssen nur die geänderten Module neu compiliert werden und dann der Linker aufgerufen werden. Dadurch wird die Zeit zur Erstellung des neuen Programms verkürzt. Dies wird von `make` unterstützt, wenn man das `Makefile` entsprechend gestaltet.
- Die Module können von unterschiedlichen Personen geschrieben werden.

7.1.2 Probleme

Der Compiler übersetzt jedes Modul separat; er hat dabei kein Wissen über die anderen Module. Wie geht der Compiler mit Funktions- und Variablennnamen um, die in

diesem Modul nicht definiert sind? In welcher Weise löst der Linker diese undefinierten Namen auf?

Prinzipieller Lösungsansatz: Beim Compilieren eines Moduls werden zwei Tabellen erzeugt; die eine enthält die Namen von allen Objekten, die in diesem Modul definiert sind und zur Verwendung in anderen Moduln freigegeben sind (die exportierten Symbole), und Verweise auf diese sog. exportierten Objekte; die andere Tabelle enthält alle Namen, die in diesem Modul vorkommen, aber nicht definiert sind (unresolved globals). Diese Tabellen sind Bestandteil der erzeugten `.o`-Dateien. Der Linker versucht nun, anhand dieser Tabellen alle unaufgelösten Namen durch gleichnamige Verweise auf exportierte Objekte zu ersetzen. Tritt dasselbe Symbol in mehreren der angegebenen `.o`-Dateien auf, so wird eine Fehlermeldung ausgegeben.

7.1.3 Linken von Bibliotheken

Achtung! Der Linker geht auf eine andere Weise vor, wenn er unaufgelöste Namen durch exportierte Symbole einer statischen Bibliothek auflösen will. Die Liste der auf der Kommandozeile angegebenen Objektdateien und Bibliotheken geht er immer von links nach rechts durch. Stößt er dabei auf eine statisch zu bindende Bibliothek, so versucht er nur, die bisher gefundenen Namen durch exportierte Symbole dieser Bibliothek aufzulösen. Die unaufgelösten Namen in den folgenden Dateien werden dabei nicht aufgelöst. **Deshalb ist die Reihenfolge, in der Dateien dem Linker genannt werden, wichtig!** Statische Bibliotheken haben üblicherweise die Endung `.a` (die an Archiv erinnern soll).

Beispiel: `modul1.o` sei eine Objektdatei, die den unaufgelösten Namen `xyz` enthalte. `modul2.a` und `modul3.a` seien statische Bibliotheken, die jeweils das Symbol `xyz` exportieren.

`gcc modul2.a modul1.o` ergibt die Fehlermeldung: `undefined symbol _xyz`
`gcc modul1.o modul2.a modul3.a` resolviert `xyz` in `modul1.o` mit dem entsprechenden Objekt in `modul2.a`
`gcc modul1.o modul3.a modul2.a` resolviert `xyz` in `modul1.o` mit dem entsprechenden Objekt in `modul3.a`.

Programmierhinweis: Schreiben Sie die zu linkenden statischen Bibliotheken immer an das Ende der Kommandozeile, mit der sie den Linker oder Compiler aufrufen.

Anders wird mit dynamischen Bibliotheken verfahren! Ihre Symboltabelle wird zur Auflösung unaufgelöster Namen in allen angegebenen Objektdateien verwendet. Dynamische Bibliotheken haben üblicherweise die Endung `.so` (einer Abkürzung von shared object).

7.1.4 Bemerkungen zu `.o`- und `.a`-Dateien

`.o`-Dateien werden mit dem C-Compiler mit der Option `-c` erzeugt. Beim Binden wird eine `.o`-Datei vollständig in die erzeugte Datei kopiert. Statisch zu bindende Bibliotheken, also `.a`-Dateien, werden aus mehreren `.o`-Dateien mit Hilfe des Programms `ar` (eine Abkürzung für Archivieren) erzeugt. Beim Binden kopiert der Linker aus einer `.a`-Datei nur die Teile heraus, die er zur Auflösung unaufgelöster Namen braucht.

7.2 Sichtbarkeit von Funktionsdefinitionen

Die Definition einer Funktion darf nicht innerhalb der Definition einer anderen Funktion stehen. Jede Funktionsdefinition ist innerhalb des Moduls, in dem sie steht, **sichtbar** d.h bei einem Funktionsaufruf gleichen Namens verwendet der Compiler diese Funktionsdefinition. Außerdem wird ein Verweis auf sie in die Symboltabelle des compilierten Codes des Moduls eingetragen, so daß sie der Linker zur Auflösung unaufgelöster Namen verwenden kann.

7.2.1 Deklaration von Funktionen als `static`

Dieser Eintrag in die Tabelle exportierter Objekte kann dadurch verhindert werden, daß vor die Definition und Deklaration der Funktion das Schlüsselwort `static` geschrieben wird. Die Funktionsdefinition ist dann nur noch innerhalb des Moduls sichtbar, in dem sie vorkommt. Es gibt keine Möglichkeit, zur Auswertung eines gleichnamigen Funktionsaufrufs in einem anderen Modul diese Funktionsdefinition zu verwenden.

Programmierhinweis: Deklarieren Sie alle Funktionen außer `main` als `static`, sofern sie nicht tatsächlich in anderen Modulen verwendet werden.

7.2.2 Deklaration von Funktionen als `extern`

Ist eine Funktion in einem Modul nicht definiert, so darf sie dort nur aufgrufen werden, wenn ihr Prototyp in dem Modul aufgeführt ist. Es ist ratsam, deutlich zu machen, daß es sich um eine externe Funktion handelt, indem man vor den Prototyp das Schlüsselwort `extern` setzt. Meist geschieht dies in einer Headerdatei mit der Endung `.h`.

7.3 Sichtbarkeit von Variablen

Eine Variable heißt **global** oder besser **modul-global**, wenn ihre Definition nicht innerhalb einer Funktion steht, ansonsten heißt sie **lokal**. Zu den lokalen Variablen gehören die Variablen, die innerhalb einer Funktion definiert sind, und die Variablen, die den Eingabeparametern einer Funktion zugeordnet sind und bei einem Funktionsaufruf angelegt werden, um die Werte der Argumente zu übernehmen.

7.3.1 Lokale Variablen

Eine lokale Variable ist nur innerhalb des kleinsten Anweisungsblocks, in dem sie deklariert ist, sichtbar. Dies bedeutet folgendes: Innnerhalb des Blockes interpretiert der Compiler den Variablennamen immer als Bezeichnung eben dieser Variablen. Außerhalb des Blockes jedoch nicht! Dort interpretiert er den Variablennamen als Bezeichnung für ein anderes Objekt, das außerhalb des Blockes definiert ist.

7.3.2 Globale Variablen

Eine modul-globale Variable ist in dem gesamten Modul sichtbar, in dem sie deklariert ist. Dies bedeutet, daß der Compiler den Variablennamen als Bezeichnung eben dieser Variablen interpretiert außer innerhalb von Anweisungsblöcken, in denen eine lokale Variable gleichen Namens deklariert ist. Dort wird der Name immer als Bezeichnung der lokalen und nicht der globalen Variablen interpretiert.

Außerdem wird eine modul-globale Variable in die Tabelle exportierter Objekte des Moduls eingetragen und somit wirklich global und potentiell sichtbar in anderen Modulen. Dies kann dadurch verhindert werden, daß der Variablendeklaration das Schlüsselwort `static` vorangestellt wird. Dann ist diese Variable nur modul-global und kann in anderen Modulen nicht verwendet werden. Kommt ihr Name in anderen Modulen vor, so bezeichnet er andere Objekte.

7.3.3 Deklaration externer globaler Variablen

Ist eine Variable in einem Modul nicht definiert, so darf sie dort nur verwendet werden, wenn sie als extern deklariert wird. Dies geschieht dadurch, daß vor ihre Deklaration das Schlüsselwort `extern` gesetzt wird. So eine Deklaration ist nie eine Definition (d.h. es wird kein Speicherplatz reserviert), sondern sie teilt lediglich dem Compiler mit, welchen Typ die Variable hat und daß er sie in die Tabelle unaufgelöster Namen des Moduls eintragen soll.

7.3.4 Beispiel 1

```
#include <stdio.h>

int f(int n) {
   return n*n;
}

int main(void) {
   int j=2;
   printf("%d\n",j);
   printf("%d\n",f(j));
   printf("%d\n",j);
   return 0;
}
```

Dieses Programm gibt die Zahlen `2 4 2` in dieser Reihenfolge aus.

7.3.5 Beispiel 2

```
#include <stdio.h>

int f(int j) {
   return j*j;
}

int main(void) {
   int j=2;
   printf("%d\n",j);
   printf("%d\n",f(j));
   printf("%d\n",j);
   return 0;
}
```

Dieses Programm gibt ebenfalls die Zahlen `2 4 2` aus. Daß der Eingabeparameter der Funktion `f` denselben Namen wie die Variable `j` in `main` hat, spielt keine Rolle, weil bei der Auswertung des Funktionsaufrufs für den Eingabeparameter ein neues Objekt angelegt wird und die in `main` definierte Variable `j` während der Zeit der Auswertung des Funktionsaufrufs nicht sichtbar ist.

7.3.6 Beispiel 3

```
#include <stdio.h>

int f(int n) {
   int j=3;
   return n*n;
}

int main(void) {
   int j=2;
   printf("%d\n",j);
   printf("%d\n",f(j));
   printf("%d\n",j);
   return 0;
}
```

Dieses Programm gibt ebenfalls die Zahlen `2 4 2` aus. Denn die im Funktionsrumpf von `f` definierte Variable `j` ist ein anderes Objekt als die in `main` definierte. Durch die Initialisierung `j=3` in der Definition von `f` wird der Wert der in `main` definierten Variablen `j` nicht beeinflußt.

7.3.7 Beispiel 4

```
#include <stdio.h>

int j=2;

int f(int n) {
   j=3;
   return n*n;
}

int main(void) {
   printf("%d\n",j);
   printf("%d\n",f(j));
   printf("%d\n",j);
   return 0;
}
```

Dieses Programm gibt die Zahlen `2 4 3` aus. Denn die Variable `j` ist jetzt außerhalb jeder Funktion definiert, also modul-global und somit innerhalb der Funktion `f` sichtbar. Deshalb wird durch die Zuweisung `j=3` in der Definition von `f` ihr Wert verändert.

Programmierhinweis: Verwenden Sie, soweit es möglich ist, immer lokale Variablen! Globale Variablen sind meist unnötig und erschweren die Fehlersuche, weil sie ja im gesamten Programm geändert werden können. Zumindest sollte man sie möglicht als `static` deklarieren, damit ihre Sichtbarkeit auf ein Modul beschränkt bleibt.

Um an eine Funktion Daten zu übergeben, sollte man Eingabeparameter und nicht globale Variablen verwenden, weil dann bei jedem Funktionsaufruf an den Argumenten sofort zu sehen ist, welche Werte übergeben werden.

Bemerkung: Neuere Sprachen als C haben meist ein detailliertes Konzept für die Sichtbarkeit von Funktionen und Variablen, indem sie z.B. eine feinere Steuerung erlauben, welche Objekte des einen Moduls in einem anderen (und nicht gleich in allen anderen) verwendet werden dürfen und umgekehrt aus welchem anderen Modul ein Objekt eines bestimmten Namens importiert werden soll.

7.4 Lebensdauer von Variablen

7.4.1 Globale Variablen

Für globale Variablen reserviert der Compiler einen festen Speicherbereich. Sie existieren daher ab dem Start der Programmausführung als Objekte im Speicher. Sie sind aber nicht unbedingt initialisiert.

7.4.2 Lokale Variablen

Lokalen Variablen wird eine der Speicherklassen `auto, static, register` zugeordnet, indem man eines dieser Schlüsselwörter vor ihre Deklaration setzt. Steht keines davor, haben sie die Speicherklasse `auto`.

Für lokale Variablen der Klasse **auto** reserviert der Compiler keinen dauerhaften Speicherplatz. Vielmehr werden jedesmal, wenn ein Funktionsaufruf ausgewertet werden muß, alle lokalen Variablen, die in der Funktion als `auto` deklariert sind, als neue Objekte im Speicher angelegt. Am Ende der Auswertung des Funktionsaufrufs werden diese Objekte wieder aus dem Speicher entfernt; sie exitieren dann nicht mehr.

Lokale Variablen der Klasse **static** dagegen existieren während der gesamten Programmausführung als Objekte im Speicher. Allerdings wird bei jedem Aufruf der Funktion, in der sie definiert sind, dasselbe Objekt verwendet; es wird also nicht wie für `auto`-Variablen bei jedem Funktionsaufruf ein neues Objekt angelegt.

Kapitel 8

Abstrakte Datentypen und Datenstrukturen

In 1.4 haben wir eine Menge zusammen mit einigen Operationen als Datentyp bezeichnet. Die bisher betrachteten Datentypen waren alle schon in C vorgegeben. Datentypen, die unabhängig von einer Programmiersprache beschrieben sind, nennt man abstrakte Datentypen. Um diesen Begriff näher zu erläutern, betrachten wir analoge Begriffsbildungen in der Mathematik.

Erinnerung an einige Begriffe aus der Mathematik

Abstrakte Gruppe

Die folgende Menge von Axiomen heißt abstrakte Gruppe oder Gruppenstruktur:

(A1) G ist eine Menge.

(A2) $f\colon G \times G \to G$ ist eine Abbildung.

(A3) $\forall x \forall y \forall z\ f(x, f(y, z)) = f(f(x, y), z)$ (Assoziativität)

(A4) $\exists\, e \in G\ (\forall x f(e, x) = x$ und $\forall x \exists x'\ f(x', x) = e)$ (Existenz von Linkseins und Linksinversem)

Gruppe

Ein Paar (G, f) heißt ein Modell für eine abstrakte Gruppe, wenn es die Axiome (A1) bis (A4) erfüllt. Üblicherweise nennt man ein solches Modell einfach eine Gruppe.

Ein Axiomensystem kann auch auf andere abstrakte Strukturen zurückgreifen, z.B. bei Vektorraumstrukturen auf Körper und abelsche Gruppen.

8.1 Abstrakte Datentypen

8.1.1 Definition *Ein **Abstrakter Datentyp** (kurz: **ADT**) ist ein Tripel (*Sorten, Methoden, Axiome*) mit folgenden Komponenten:*

Sorten *ist eine Menge von Bezeichnern für Mengen (man spricht auch von Arten oder Modes).*

Methoden *ist eine Menge von Bezeichnern für Abbildungen mit Angabe von Quelle und Ziel.*

Axiome *ist eine Menge von Axiomen, die Eigenschaften der bezeichneten Mengen und Abbildungen beschreiben.*

Bemerkung Unter den Sorten sollte man sich Wertebereiche für Objekte in einem Programm vorstellen und unter den Methoden Operationen, die man mit diesen Objekten durchführen will.

8.1.2 Definition *Ein **Modell** für einen ADT ist eine Zuordnung, die den Sorten und Methoden des ADT konkrete Mengen und Abbildungen so zuordnet, daß die Axiome des ADT erfüllt sind. Dabei muß die Gleichheit von Mengen eventuell nur bis auf eine Bijektion erfüllt sein. Hat ein ADT bis auf Isomorphie (d.h. bis auf Bijektionen der Mengen) nur ein Modell, so heißt er **monomorph**, sonst **polymorph**.*

Bemerkung Eine Gruppenstruktur kann man als ADT und jede Gruppe als Modell dafür auffassen.

8.1.3 Sinn und Zweck abstrakter Datentypen

In Gestalt eines ADT kann man erwünschte Eigenschaften von Objekten und Operationen auf implementierungsunabhängige Weise beschreiben. Wie ein Programmierer Objekte und Operationen mit diesen Eigenschaften implementiert, ist egal. Er soll nur ein Programmmodul erstellen, das die Funktionen enthält, die solche Objekte erzeugen (und eventuell auch wieder aus dem Speicher entfernen) und die gewünschten Operationen durchführen. Als Anwender darf man nur diese Funktionen aufrufen; ihre Implementierung bleibt in dem Modul versteckt. Dadurch ist es leichter, Programmierfehler zu finden, und man kann jederzeit das Modul gegen ein anderes austauschen, das eine völlig andere, vielleicht effizientere Implementierung enthält.

8.2 Datenstrukturen

Für den Programmierer stellt sich die Aufgabe, mit Hilfe der jeweiligen Programmiersprache Objekte zu konstruieren, mit denen man die gewünschten Operationen durchführen kann. Ein ähnliches Problem stellt sich bei der mathematischen Modellierung physikalischer Größen. Aus Experimenten weiß man, wie sich physikalische Größen unter gewissen Transformationen des Bezugssystems verändern. Gesucht sind mathematische Objekte, die das gleiche Transformationsverhalten haben. Der Mathematiker konstruiert sich solche Objekte (z.B. Tensoren, Differentialformen), indem er sie aus einfacheren (z.B. Zahlen) aufbaut (meist als Elemente von Räumen von Abbildungen).

Um in der Informatik geeignete Objekte für gewünschte Operationen zu finden, benutzt man Datenstrukturen. Eine allgemein verbindliche, präzise Definition von Datenstruktur gibt es nicht. Wir geben eine informelle Definition.

8.2.1 Definition *Unter einer **Datenstruktur** versteht man eine Methode,*

- *in einer Programmiersprache aus Objekten, deren Typen in der Sprache vorgegeben sind, kompliziertere Objekte zu konstruieren (so ein Objekt nennt man manchmal eine Inkarnation der Datenstruktur),*
- *und diesen Objekten Werte zuzuordnen.*

Der Sinn besteht zum einen darin, andere Wertebereiche zu realisieren als die der in der Sprache vorgegebenen Datentypen; zum anderen darin, durch gewisse Zusatzstrukturen dieser neuen Objekte die Implementierung gewünschter Operationen zu erleichtern.

Objekte mit einem in der Sprache vorgegebenen Typ kann man als Datenstruktur auffassen. Arrays vom Typ T sind bereits aus einfacheren aufgebaut, sind also schon eine etwas kompliziertere Datenstruktur. Als Wertebereich kann man W_{T}^{L} wählen, wobei W_{T} der Wertebereich des Typs `T` und L die Länge des Arrays sind.

Nichttriviale typische Beispiele für Datenstrukturen verwenden meist `struct`-Datentypen, die durch Zeiger miteinander verkettet sind.

Oft werden auch noch einige Operationen zu den Datenstrukturen gezählt, insbesondere Konstruktoren und Destruktoren zur Erzeugung und Vernichtung entsprechender Objekte.

8.2.2 Definition *Eine **Implementierung eines abstrakten Datentyps** ist eine Zuordnung, die jedem Sortenbezeichner eine Datenstruktur und jedem Methodenbezeichner die Definition einer Funktion (oder Prozedur in anderen Sprachen als C) oder einen Operator zuordnet, so daß man durch Übergang zu den Wertebereichen der Datenstrukturen und den durch die Funktionsdefinitionen gegebenen Abbildungen ein Modell des ADT erhält.*

Vergleicht man einen ADT mit den in C vorgegebenen Datentypen, so stellt man fest, daß die Methoden des ADT die in C vorgegebenen Operatoren verallgemeinern und die Datenstrukturen die Wertebereiche der in C vorgegebenen Datentypen.

8.2.3 Bemerkung Im üblichen Sprachgebrauch werden obige Begriffe nicht scharf gegeneinander abgegrenzt. Oft wird zwischen einem ADT und einem Modell des ADT nicht unterschieden, insbesondere wenn er monomorph ist. Auch die Grenzen zwischen Modell, Implementierung und Datenstruktur sind fließend. Das liegt vor allem daran, daß man die Axiome eines ADT etwas lax interpretiert. Wenn dort z.B. von natürlichen oder reellen Zahlen die Rede ist, werden darunter oft stillschweigend nur die Zahlen verstanden, die sich durch einen geeigneten arithmetischen Datentyp der Programmiersprache darstellen lassen. Außerdem zählt man zur Implementierung eines ADT meist auch Konstruktoren und Destruktoren zur Erzeugung und Vernichtung von Objekten sowie das Abfangen von Fehlersituationen, die im ADT nicht auftreten, wohl aber in der Praxis z.B. wegen Speicherplatzmangels. Deshalb können in einer Implementierung mehr Operationen als im ADT vorkommen.

8.3 Der Abstrakte Datentyp **Boolean**

(benannt nach G. Boole, 1815-1864)

SORTEN: B

METHODEN: $\mathsf{not}\colon B \to B, \quad \mathsf{or}\colon B \times B \to B, \quad \mathsf{and}\colon B \times B \to B$

AXIOME: Es gibt ein Element $\mathsf{F} \in B$, so daß gilt

$\mathsf{F} \neq \mathsf{not}(\mathsf{F})$

$\mathsf{or}^{-1}(\mathsf{F}) = \{(\mathsf{F}, \mathsf{F})\}$

$\mathsf{and}^{-1}(\mathsf{F}) = (\{\mathsf{F}\} \times B) \cup (B \times \{\mathsf{F}\})$

Man könnte noch weitere Abbildungen hinzunehmen, z.B. xor, nand, nor.

8.3.1 Ein Modell für den ADT Boolean

Seien $B = \{0, 1\}$ und $\mathsf{F} = 0$. Die Abbildungen seien durch folgende Tabellen definiert.

not	0	1
	1	0

and	0	1
0	0	0
1	0	1

or	0	1
0	0	1
1	1	1

Interpretiert man die Werte 0 als *falsch* und 1 als *wahr*, so erhält man die üblichen logischen Operationen *nicht*, *und* und *oder*.

8.3.2 Implementierung in C

Wähle B als Wertebereich des Aufzählungstyps `enum boolean {false,true}`.
Dann realisieren die Operatoren `!`, `&&` und `||` die Abbildungen not, and bzw. or.

Bemerkung Die obige Definition des ADT Boolean liefert einen *polymorphen* ADT. Denn man kann weitere, nicht isomorphe Modelle angeben. Man kann aber Monomorphie erzwingen, indem man ein weiteres Axiom hinzunimmt, nämlich daß die Menge B aus genau zwei Elementen besteht.

8.3.3 Ein weiteres Modell für den ADT Boolean

Seien $B = \mathbb{Z}$ und $\mathsf{F} = 0$. Die Abbildungen not, and, or seien definiert durch

$\mathsf{not}(0) = 1$ und $\mathsf{not}(n) = 0$ für jedes $n \in \mathbb{Z} \setminus \{0\}$,

$\mathsf{and}(0, n) = \mathsf{and}(n, 0) = 0$ für jedes $n \in \mathbb{Z}$ und $\mathsf{and}(n, m) = 1$ für $(n, m) \neq (0, 0)$,

$\mathsf{or}(n, m) = 0$ für $(n, m) = (0, 0)$ und $\mathsf{or}(n, m) = 1$ sonst.

Ein ähnliches Modell ist in C bereits implementiert:

B = Wertebereich von `int`

not = Operator `!`

and = Operator `&&`

or = Operator `||`

8.4 Der Raum der Abbildungen $\{1, \ldots, N\} \to W$

8.4.1 Beschreibung des ADT

SORTEN:	$W, \mathcal{F}$
METHODEN:	$\mathsf{eval}\colon \mathcal{F} \times \{1, \ldots, N\} \to W$ und $\mathsf{def}\colon \mathcal{F} \times \{1, \ldots, N\} \times W \to \mathcal{F}$
AXIOME:	(A1) $\mathcal{F}$ ist die Menge der Abbildungen $f\colon \{1, \ldots, N\} \to W$
	(A2) Für jedes $(f, n) \in \mathcal{F} \times \{1, \ldots, N\}$ gilt $\mathsf{eval}(f, n) = f(n)$.
	(A3) Für $(f, n, w) \in \mathcal{F} \times \{1, \ldots, N\} \times W$ ist
	$\mathsf{def}(f, n, w)(n) = w$
	$\mathsf{def}(f, n, w)(j) = f(j)$ für $j \neq n$.

8.4.2 Bemerkungen

1. eval gibt den Funktionswert an einer Stelle zurück; def definiert ihn um.

2. Die Abbildungen $f\colon \{1, \ldots, N\} \to W$ haben viele Namen, je nach Anwendungsgebiet. In der Mathematik schreibt man sie meist als N-Tupel $(f(1), \ldots, f(N))$ oder als Folge und $\mathcal{F}$ als W^N; bei formalen Sprachen schreibt man sie als Wörter $f(1) \ldots f(N)$ fester Länge N über dem Alphabet W; in der Informatik bezeichnet man sie auch als lineare Liste fester Länge N mit Elementen aus W.

3. Die obige Axiomenmenge hat nicht isomorphe Modelle, weil W nicht näher festgelegt ist, Ist aber W gewählt, so sind eval und def eindeutig festgelegt.

4. Operationen, die auf W definiert sind, induzieren Operationen auf $\mathcal{F}$, indem man sie einfach punktweise definiert. Ist W z.B. der Wertebereich eines arithmetischen Datentyps, so kann man noch Axiome für die punktweise Multiplikation, Addition, Minimum- und Maximumbildung hinzufügen.

8.4.3 Eine Implementierung in C

`TYP` sei ein in C vorgegebener Datentyp mit dem Wertebereich $W_{\texttt{TYP}}$.

Dem Sortenbezeichner W ordnen wir die Datenstruktur *Objekt vom Typ* `TYP` mit dem Wertebereich $W_{\texttt{TYP}}$ zu.

Dem Sortenbezeichner $\mathcal{F}$ ordnen wir die Datenstruktur *Array von* `TYP` *der Länge* N zu; bezeichnet `a` ein Objekt dieser Datenstruktur (also ein Array von `TYP` der Länge N), so sei sein Wert die Abbildung $f\colon \{1, \ldots, N\} \to W_{\texttt{TYP}}$, die durch $f(j) = \texttt{a[}j-1\texttt{]}$ gegeben ist. Damit ist das Axiom (A1) erfüllt.

Implementierung von eval

```
TYP function_eval( TYP a[], int n ) {
   return a[n-1];
}
```

Implementierung von def

```
void function_def( TYP a[], int n, TYP wert, TYP b[] ) {
   int i;
   for(i=0;i<n;i++) b[i]=a[i];
   b[n-1]=wert;
}
```

Kapitel 9

Listen

Anschaulich ist eine Liste ein Speicher, der Objekte eines vorgegebenen Datentyps in einer gewünschten Reihenfolge enthält und einige Operationen wie Einfügen, Löschen, Lesen zuläßt. Es ist mühsam und nicht sehr übersichtlich, Listen als ADT zu axiomatisieren. Üblicherweise definiert man Listen als mathematische Modelle oder gleich als Datenstruktur.

9.1 Ein Modell für eine Liste über einer Menge W mit direktem Zugriff

W heißt die Wertemenge der Listenelemente. $\mathcal{L} := W^*$ heißt die Menge der Inhalte der Liste; ϵ sei das leere Wort.
$P := \mathbb{N}$ heißt die Menge der Listenpositionen.
Die Menge der Listenoperationen bestehe aus den folgenden, teilweise nur partiell definierten Abbildungen:

1. `make_empty`: $\mathcal{L} \to \mathcal{L}, L \mapsto \epsilon$
2. `is_empty`: $\mathcal{L} \to \{\texttt{true}, \texttt{false}\}$, `is_empty`$(L) =$ `true`, falls $L = \epsilon$, und `false` sonst.

Im folgenden sei $L = w_1 \dots w_n \in \mathcal{L}$, wobei $n = 0$ als $L = \epsilon$ zu verstehen ist.

3. `insert`: $\mathcal{L} \times P \times W \to \mathcal{L}$,

$$\texttt{insert}(L, j, w) := \begin{cases} ww_1 \dots w_n & , \text{ falls } j = 1 \\ w_1 \dots w_n w & , \text{ falls } j = n+1 \\ w_1 \dots w_{j-1} w w_j \dots w_n & , \text{ falls } 1 < j \leq n \\ \text{nicht definiert} & , \text{ falls } j > n+1 \end{cases}$$

4. `delete`: $\mathcal{L} \times P \to \mathcal{L}$,

$$\texttt{delete}(L,j) := \begin{cases} w_2 \dots w_n & , \text{ falls } j = 1 \text{ und } n \geq 1 \\ w_1 \dots w_{n-1} & , \text{ falls } j = n \text{ und } n \geq 1 \\ w_1 \dots w_{j-1} w_{j+1} \dots w_n & , \text{ falls } 1 < j < n \\ \text{nicht definiert} & , \text{ falls } j > n \end{cases}$$

5. `read`: $\mathcal{L} \times P \to W$,

$$\texttt{read}(L,j) := \begin{cases} w_j & , \text{ falls } 1 \leq j \leq n \\ \text{nicht definiert} & , \text{ falls } j > n \end{cases}$$

6. `retrieve`: $\mathcal{L} \times W \to P \cup \{0\}$,

$$\texttt{retrieve}(L,w) := \begin{cases} \min J & \text{falls } J := \{j : w_j = w\} \neq \emptyset \\ 0 & \text{sonst} \end{cases}$$

9.2 Ein Modell für eine Liste über einer Menge W mit Zugriff über einen Zeiger

Oft will man sich die Position des letzen Listenzugriffs als Bestandteil der Liste merken. Dann definiert man z.B. $\mathcal{L} := \{(u,j) \in W^* \times \mathbb{N} : j \leq 1 + \text{länge}(u)\}$ als die Menge der Listenzustände. Die Menge der Listenoperationen enthält dann meist die folgenden, teilweise nur partiell definierten Abbildungen:

1. `make_empty`: $\mathcal{L} \to \mathcal{L}, (u,j) \mapsto (\epsilon, 1)$
2. `is_empty`: $\mathcal{L} \to \{\texttt{true}, \texttt{false}\}$, $\texttt{is\_empty}(L) = \texttt{true}$, falls $L = \epsilon$, und $\texttt{is\_empty}(L) = \texttt{false}$ sonst.

Im folgenden sei $L = (u,j) = (u_1 \dots u_n, j) \in \mathcal{L}$, eventuell $n = 0$, also $u = \epsilon$.

3. `insert`: $\mathcal{L} \times W \to \mathcal{L}$,

$$\texttt{insert}(L,w) := \begin{cases} (wu_1 \dots u_n, j) & , \text{ falls } j = 1 \\ (u_1 \dots u_n w, j) & , \text{ falls } j = n+1 \\ (u_1 \dots u_j w u_{j+1} \dots u_n, j) & , \text{ falls } 1 < j \leq n \end{cases}$$

4. `delete`: $\mathcal{L} \to \mathcal{L}$,

$$\texttt{delete}(L) := \begin{cases} (u_2 \dots u_n, j) & , \text{ falls } j = 1 \text{ und } n \geq 1 \\ (u_1 \dots u_{n-1}, j) & , \text{ falls } j = n \text{ und } n \geq 1 \\ (u_1 \dots u_{j-1} u_{j+1} \dots u_n, j) & , \text{ falls } 1 < j < n \\ L & , \text{ falls } j = n+1 \end{cases}$$

5. `read`: $\mathcal{L} \to W$,

$$\texttt{read}(L) := \begin{cases} u_j & , \text{ falls } 1 \leq j \leq n \\ \text{nicht definiert} & , \text{ falls } j = n+1 \end{cases}$$

6. `reset`: $\mathcal{L} \to \mathcal{L}, L = (u, j) \mapsto (u, 1)$

7. `forward`: $\mathcal{L} \to \mathcal{L}$,

$$\texttt{forward}(L) = \begin{cases} (u, j+1) \ , & \text{falls } j \leq n \\ L \qquad\quad , & \text{falls } j = n+1 \end{cases}$$

8. `at_end`: $\mathcal{L} \to \{\texttt{true}, \texttt{false}\}$,

$$\texttt{at\_end}(L) = \begin{cases} \texttt{true} & \text{falls } j = n+1 \\ \texttt{false} & \text{sonst} \end{cases}$$

Manchmal nimmt man auch noch folgende Abbildungen hinzu.

9. `back`: $\mathcal{L} \to \mathcal{L}$,

$$\texttt{back}(L) = \begin{cases} (u, j-1) \ , & \text{falls } j > 1 \\ L \qquad\quad , & \text{falls } j = 1 \end{cases}$$

10. `at_begin`: $\mathcal{L} \to \{\texttt{true}, \texttt{false}\}$,

$$\texttt{at\_begin}(L) = \begin{cases} \texttt{true} & \text{falls } j = 1 \\ \texttt{false} & \text{sonst} \end{cases}$$

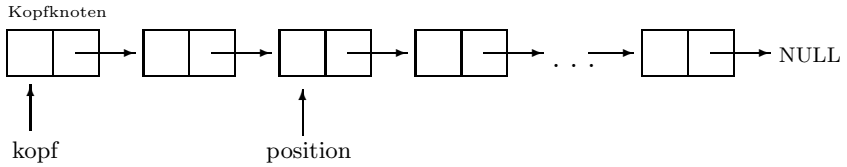
Man überlege sich, wie man eine `retrieve`-Funktion wie in dem vorigen Modell definiert.

9.3 Die Datenstruktur *Einfach verkettete Liste*

W sei der Wertebereich des Datentyps `TYP` der Listenelemente.

Den einfach verketteten Listen liegt folgende Idee zugrunde: Jede Komponente des Listeninhalts wird in einem Struct verpackt, der außer dem Wert aus W auch noch einen Zeiger enthält, der wieder auf so einen Struct zeigen kann. Eine Liste wird durch eine Menge von solchen Structs, die durch die Zeiger zu einer Folge verkettet sind, dargestellt. Der Zeiger des letzten Listenelements ist ein NULL-Zeiger. Der Position j in dem Modell 9.2 entspricht dann ein Zeiger, den wir mit `position` bezeichnen. Die Implementierung der `insert`- und `delete`-Funktion wird etwas einheitlicher und übersichtlicher, wenn man vor die eigentliche Liste ein sog. Kopfelement setzt, dessen Datenkomponente keine Rolle spielt. Denn dann zeigt der Positionszeiger stets auf einen Listenelement-Struct, und die Operationen `insert, delete, read` bearbeiten das darauffolgende Listenelement. Man benötigt dann keine Sonderbehandlung für die leere Liste.

Einfach verkettete Liste

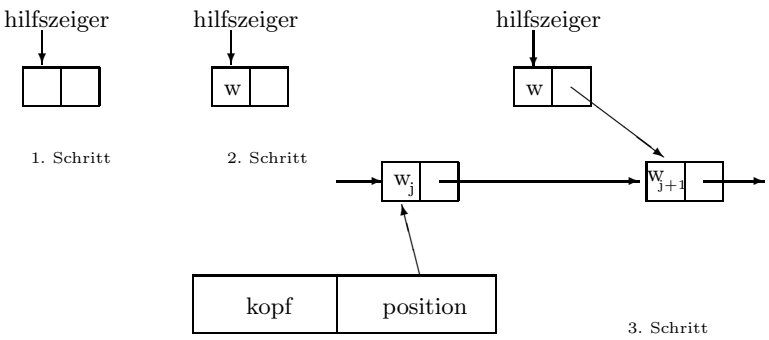


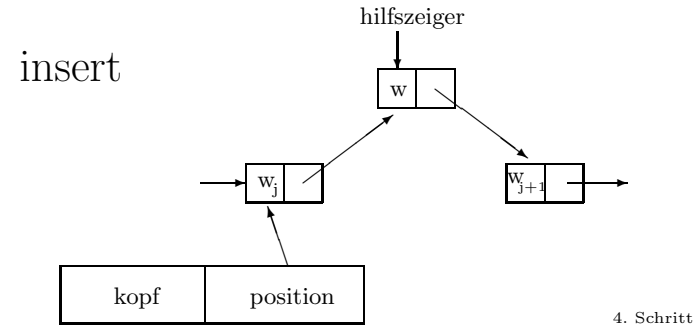
9.3.1 Eine Implementierung in C

```
#include <stdio.h>
#include <stdlib.h>

#define TYP int      /* oder irgendetwas anderes */


/************ Listenelement *************/

struct listenelement {
   TYP wert;
   struct listenelement *nachfolger;
};

typedef struct listenelement listenelement_t;

/************ Listentyp ***************/

typedef struct {
   listenelement_t *kopf;
   listenelement_t *position;
}  liste_t;

/************* Boolean ****************/

typedef enum {false=0, true=1, fehlerhaft=0, fehlerfrei=1} boolean;

/************ make_empty ***************/
/*     !ohne Speicherfreigabe!         */

void make_empty(liste_t *zeiger) {
   zeiger->kopf->nachfolger = NULL;
   zeiger->position = zeiger->kopf;
}

/************ is_empty ****************/

boolean is_empty(liste_t liste) {
   if(liste.kopf->nachfolger == NULL) return true;
   else return false;
}

/************ insert ****************/

boolean insert(liste_t liste, TYP w) {
   listenelement_t *hilfszeiger;
   hilfszeiger = (listenelement_t *)malloc(sizeof(listenelement_t));
```

```
   if(hilfszeiger==NULL) return fehlerhaft;
   hilfszeiger->wert = w;
   hilfszeiger->nachfolger = liste.position->nachfolger;
   liste.position->nachfolger = hilfszeiger;
   return fehlerfrei;
}

/************* delete ***************/

boolean delete(liste_t liste) {
   listenelement_t *hilfszeiger;
   hilfszeiger = liste.position->nachfolger;
   if(hilfszeiger==NULL) return fehlerhaft;
   liste.position->nachfolger = hilfszeiger->nachfolger;
   free(hilfszeiger);
   return fehlerfrei;
}

/************* read *****************/

boolean read(liste_t liste, TYP *w) {
   if(liste.position->nachfolger == NULL) return fehlerhaft;
   *w = liste.position->nachfolger->wert;
   return fehlerfrei;
}

/************* reset ****************/

void reset(liste_t *zeiger) {
   zeiger->position = zeiger->kopf;
}

/************* forward *************/

boolean forward(liste_t *zeiger) {
   if(zeiger->position->nachfolger == NULL) return fehlerhaft;
   zeiger->position = zeiger->position->nachfolger;
   return fehlerfrei;
}

/************* at_end **************/

boolean at_end(liste_t liste) {
   if(liste.position->nachfolger == NULL) return true;
   else return false;
}
```

```
/************ at_begin ************/

boolean at_begin(liste_t liste) {
   if(liste.position == liste.kopf) return true;
   else return false;
}

/************ back ***************/

boolean back(liste_t *zeiger) {
   listenelement_t *hilfszeiger;
   hilfszeiger = zeiger->position;
   if(hilfszeiger==zeiger->kopf) return fehlerhaft;
   zeiger->position = zeiger->kopf;
   while(zeiger->position->nachfolger != hilfszeiger)
      zeiger->position = zeiger->position->nachfolger;
   return fehlerfrei;
}
```

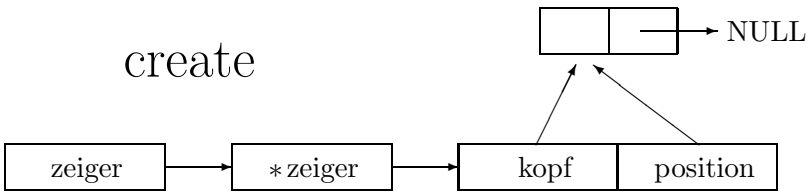




Wir geben zwei Beispiele für Funktionen an, die eine neue, leere Liste, die nur aus einem Kopfelement besteht, erzeugen. Die erste Funktion ist wahrscheinlich leichter verständlich; sie gibt einen Struct vom Typ `liste_t`, der die neu erzeugte Liste beschreibt, zurück. Fehler bei der Speicherallozierung werden nicht abgefangen.

```
/*********** Konstruktor 1.Version **************/

liste_t create(void) {
   listenelement_t *hilfszeiger;
   liste_t *listenzeiger;

   listenzeiger = (liste_t *)malloc(sizeof(liste_t));
   hilfszeiger = (listenelement_t *)malloc(sizeof(listenelement_t));

   hilfszeiger->nachfolger = NULL;
   listenzeiger->kopf = hilfszeiger;
   listenzeiger->position = hilfszeiger;
   return *listenzeiger;
}
```

Die zweite Konstruktorfunktion ist vom Typ her ähnlich zu den obigen Funktionen für die Listenoperationen. Ihr Rückgabewert vom Typ `boolean` gibt an, ob die Erzeugung einer neuen Liste erfolgreich verlaufen ist oder nicht. Die Funktion alloziert Speicher für ein Kopfelement und für einen Struct vom Typ `liste_t` und initialisiert die Komponenten so, daß sie eine leere Liste beschreiben. Die Adresse des Listenstructs soll dem aufrufenden Programm mit Hilfe des Arguments der Konstruktorfunktion mitgeteilt werden. Die Adresse kann in einem Objekt vom Typ `liste_t *` abgespeichert werden. Also muß der Konstruktorfunktion die Adresse eines Zeigers auf `liste_t` als Argument übergeben werden, d.h. der Eingabeparamater der Konstruktorfunktion muß ein Zeiger auf einen Zeiger auf `liste_t` sein.



```
/*********** Konstruktor 2.Version ***************/

boolean create(liste_t **zeiger) {
   listenelement_t *hilfszeiger;

   hilfszeiger = (listenelement_t *)malloc(sizeof(listenelement_t));
   if(hilfszeiger==NULL) return fehlerhaft;
   hilfszeiger->nachfolger = NULL;
   *zeiger = (liste_t *)malloc(sizeof(liste_t));
   if(*zeiger==NULL) return fehlerhaft;
   (*zeiger)->kopf = hilfszeiger;
   (*zeiger)->position = hilfszeiger;
   return fehlerfrei;
}
```

Dieser Konstruktor kann dann z.B. folgendermaßen verwendet werden (ohne Fehlerbehandlung):

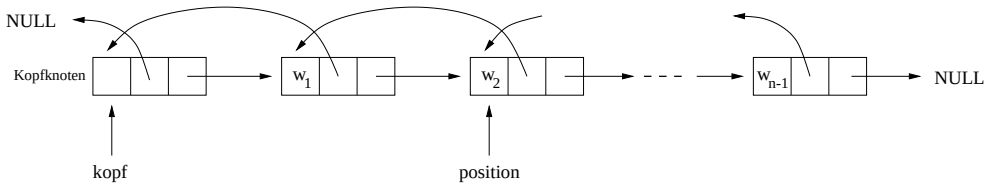
```
liste_t *ListenZeiger;
create(&ListenZeiger);
```

Dann bezeichnet `*ListenZeiger` den neu erzeugten Listenstruct. Man kann ihn natürlich auch einer Variablen vom Typ `liste_t` als Wert zuweisen:

```
liste_t neue_Liste;
liste_t *ListenZeiger;
create(&ListenZeiger);
neue_Liste = *ListenZeiger;
```

9.4 Die Datenstruktur *Doppelt verkettete Liste*

Verwendet man einfach verkettete Listen, so ist es leicht, von einem Element zum nächsten zu gehen, aber aufwendig, zum vorhergehenden zurückzugehen. Denn man muß vom Anfang der Liste an suchen. Dasselbe Problem tritt auf, wenn man bei den Operationen `insert, delete, read` dasjenige Listenelement, auf welches der Positionszeiger weist, bearbeitet und nicht, wie wir es oben gemacht haben, das darauf folgende. Deshalb verwendet man oft doppelt verkettete Listen, bei denen jedes Listenelement einen Zeiger auf das nachfolgende und einen auf das vorhergehende enthält.



Doppelt verkettete Liste

9.4.1 Eine Implementierung einiger Listenoperationen

```
#include <stdio.h>
#include <stdlib.h>

#define TYP int      /* oder irgendetwas anderes */

/*********** Listenelement *************/

struct listenelement {
   TYP wert;
   struct listenelement *vorgaenger;
   struct listenelement *nachfolger;
};

typedef struct listenelement listenelement_t;

/************ Listentyp ***************/

typedef struct {
   listenelement_t *kopf;
   listenelement_t *position;
} liste_t;

/************* Boolean ****************/

typedef enum {false=0, true=1, fehlerhaft=0, fehlerfrei=1} boolean;


/********** insert ***********/

boolean insert(liste_t liste, TYP w) {
   listenelement_t *hilfszeiger;
   hilfszeiger = (listenelement_t *)malloc(sizeof(listenelement_t));
   if(hilfszeiger==NULL) return fehlerhaft;
   hilfszeiger->wert = w;
   hilfszeiger->nachfolger = liste.position->nachfolger;
   hilfszeiger->vorgaenger = liste.position;
   if(liste.position->nachfolger != NULL)
      liste.position->nachfolger->vorgaenger = hilfszeiger;
   liste.position->nachfolger = hilfszeiger;
   return fehlerfrei;
}

/*********** delete ***********/

boolean delete(liste_t liste) {
   listenelement_t *hilfszeiger;
```

```
   hilfszeiger = liste.position->nachfolger;
   if(hilfszeiger==NULL) return fehlerhaft;
   liste.position->nachfolger = hilfszeiger->nachfolger;
   if(hilfszeiger->nachfolger != NULL)
      hilfszeiger->nachfolger->vorgaenger = liste.position;
   free(hilfszeiger);
   return fehlerfrei;
}

/*********** back ************/

boolean back(liste_t *zeiger) {
   if(zeiger->position->vorgaenger == NULL) return fehlerhaft;
   zeiger->position = zeiger->position->vorgaenger;
   return fehlerfrei;
}
```

9.5 Bemerkungen

9.5.1 Listen dienen dazu, sich eine Menge von Daten zu merken und einige einfache Operationen damit durchführen zu können wie Hinzufügen und Wegnehmen eines Datums oder Feststellen, ob ein Datum in der Liste vorkommt. Eine Liste ist also eine Datenstruktur, mit welcher der ADT Menge implementiert werden kann, allerdings mit dem Unterschied, daß in einer Liste ein Wert mehrfach vorkommen kann, in einer Menge jedoch nicht. Nicht alle Mengenoperationen sind gleich effizient implementierbar. So ist z.B. die Überprüfung, ob ein vorgelegter Wert in der Liste vorkommt, i.a. nur dadurch durchzuführen, daß man die Liste durchläuft. Deshalb werden Listen meist nur in speziellen Situationen eingesetzt wie z.B. als Stapel oder Warteschlangen, wo die Suche nach einem Wert keine Rolle spielt (siehe die nächsten Abschnitte). Muß man jedoch öfters nach Werten suchen, sind andere Datenstrukturen günstiger wie z.B. Suchbäume, bei denen die Datenwerte allerdings eine Zusatzstruktur haben müssen, so daß sie angeordnet werden können.

9.5.2 Man kann natürlich das Kopfelement mit leerem Datenfeld weglassen und außerdem die Operationen `insert, delete, read` direkt an der Stelle, auf die der Positionszeiger weist, vornehmen. Dann muß man jedoch mehr Sonderfälle behandeln.

9.5.3 Ist man bei der Suche nach Programmfehlern gezwungen, sich den Binärcode der Listendaten anzusehen, so ist es unangenehm, wenn wie in obigen Implementierungen die Zeiger in einem Listenelement nach dem Datenwert kommen. Will man nämlich manuell in der Liste entlanglaufen, so muß man bei jedem Listenelement um `sizeof(TYP)` Bytes weitergehen, um den nächsten Zeiger zu finden. Speichert man jedoch in jedem Listenelement den Nachfolgerzeiger als erste Komponente, also vor dem Datenwert ab, so ist der Wert des Nachfolgerzeigers direkt die Adresse, unter welcher der Nachfolgerzeiger des nächsten Listenelements zu finden ist.

Kapitel 10

Schlangen

Wenn zwei Geräte oder zwei Prozesse im Rechner Daten austauschen, sind häufig die Geschwindigkeiten, mit denen der eine Partner Daten liefert und der andere sie liest und verarbeitet, sehr unterschiedlich. So kann z.B. ein Drucker die Daten meist nicht so schnell verarbeiten, wie der Rechner sie liefert. Um den Rechner nicht zum Warten zu zwingen, schaltet man einen Pufferspeicher dazwischen, der nach dem Warteschlangen- oder FIFO-Prinzip (First In - First Out) arbeitet. So ein Pufferspeicher wird auch häufig eingesetzt, wenn ein UNIX-Prozeß einem anderen Daten übergibt, z.B. wenn der Standardausgabekanal des einen in den Standardeingabekanal des anderen umgelenkt wird; man nennt solche Zwischenspeicher*pipes*. Auch in Algorithmen werden Warteschlangen verwendet, z.B. bei der Breitensuche in Graphen (siehe später).

10.1 Der abstrakte Datentyp **Warteschlange (FIFO, queue) über einer Menge** W

SORTEN:	S
METHODEN:	$\mathsf{enqueue}\colon S \times W \to S, \quad \mathsf{dequeue}\colon S \to S, \mathsf{front}\colon S \to W$
	$\mathsf{is\_empty}\colon S \to \{\mathtt{true}, \mathtt{false}\}, \mathsf{make\_empty}\colon S \to S$
AXIOME:	(A1) Für jedes $s \in S$ gilt $\mathsf{is\_empty}(\mathsf{make\_empty}(s)) = \mathtt{true}$
	(A2) Für alle $s \in S$ und $w \in W$ gilt
	$\mathsf{is\_empty}(\mathsf{enqueue}(s, w)) = \mathtt{false}$
	(A3) Für alle $s \in S$ und $w \in W$ gilt
	$\mathsf{dequeue}(\mathsf{enqueue}(s, w)) = \begin{cases} s \text{ , falls } \mathsf{is\_empty}(s) = \mathtt{true} \\ \mathsf{enqueue}(\mathsf{dequeue}(s), w) \text{ sonst} \end{cases}$
	(A4) Für alle $s \in S$ und $w \in W$ gilt
	$\mathsf{front}(\mathsf{enqueue}(s, w)) = \begin{cases} w \text{ , falls } \mathsf{is\_empty}(s) = \mathtt{true} \\ \mathsf{front}(s) \text{ sonst} \end{cases}$

Anschauliche Interpretation: $\mathsf{enqueue}(s, w)$ fügt w am Ende der Warteschlange an. $\mathsf{front}(s)$ gibt das Element am Kopf der Warteschlange aus. $\mathsf{dequeue}(s)$ entfernt das Element am Kopf der Warteschlange.

In der Praxis denkt man meist weniger an den ADT, sondern eher an das folgende Modell.

10.2 Ein Modell des ADT Schlange über W

Seien $S = W^*$ und ϵ das leere Wort.

$\mathsf{enqueue}\colon S \times W \to S, (s, w) \mapsto sw$.

$\mathsf{dequeue}\colon S \to S$ sei definiert durch

$$\mathsf{dequeue} = \begin{cases} w_2 \dots w_n & , \text{ wenn } s = w_1 \dots w_n, n \geq 1 \\ \epsilon & , \text{ wenn } s = \epsilon \end{cases}$$

$\mathsf{front}\colon S \to W$ sei definiert durch

$$\mathsf{front}(s) = \begin{cases} w_1 & , \text{ wenn } s = w_1 \dots w_n, n \geq 1 \\ \text{irgendein } w \in W & , \text{ wenn } s = \epsilon \end{cases}$$

$\mathsf{is\_empty}\colon S \to \{\mathtt{true}, \mathtt{false}\}$ sei definiert durch

$$\mathsf{is\_empty}(s) = \begin{cases} \mathtt{true} & , \text{ wenn } s = \epsilon \\ \mathtt{false} & , \text{ sonst} \end{cases}$$

$\mathsf{make\_empty}\colon S \to S$, $\mathsf{make\_empty}(s) = \epsilon$ für jedes $s \in S$.

Man rechne nach, daß alle Axiome erfüllt sind.

10.3 Implementierung als Liste

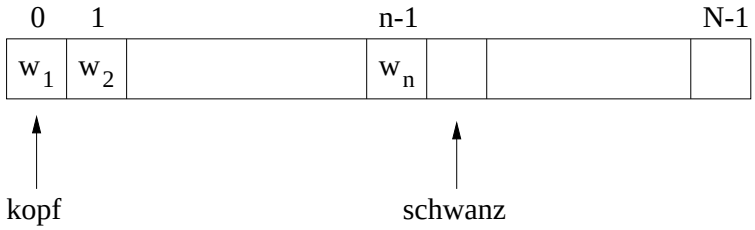
Schlangen lassen sich leicht als einfach verkettete Listen realisieren. Die Operation enqueue hängt das neue Element an das Ende der Liste an. Der Positionszeiger weist immer auf das Ende der Liste. Die Operation dequeue entfernt das erste Element der Liste, front liest den Datenwert des ersten Elementes.

10.4 Implementierung als zirkuläres Array

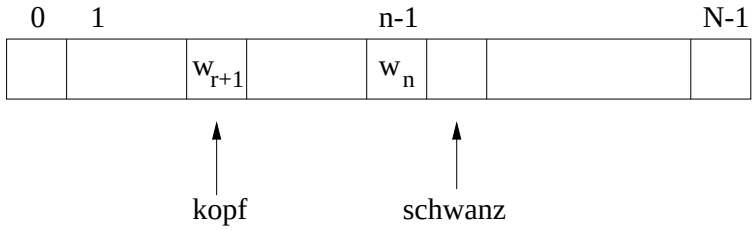
In praktischen Anwendungen beschränkt man sich oft auf Schlangen, deren Länge eine vorgegebene Schranke `MAXLENGTH` nicht überschreitet. Zur Speicherung verwendet man meist ein Array. Außerdem muß man sich zwei Indizes `kopf` und `laenge` merken, welche die Position des Schlangenanfangs und der Anzahl der Elemente der Schlange angeben. Das mit `kopf` indizierte Element des Arrays sei dasjenige, welches mit der

front-Operation ausgelesen wird. Zur Abkürzung bezeichnen wir `MAXLENGTH` mit `N` und setzen `schwanz` = `kopf`+`laenge`.

Beginnt man mit einer leeren Schlange und liest `n < N` Werte ein d.h. bildet man $S =$ enqueue(enqueue... (enqueue$(\epsilon, w_1), w_2) \ldots, w_n)$, so erhält man folgenden Zustand

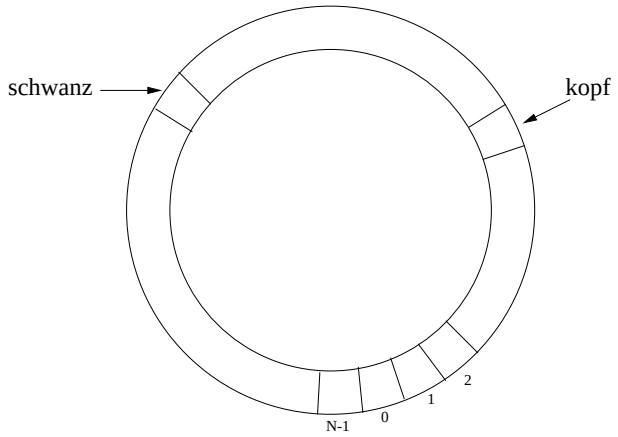


Liest man nun mit dequeue `r < n` Werte aus, so erhält man



Problem: Nachdem `r` Zeichen mit dequeue entfernt sind, sind zwar `N-n+r` Plätze frei; ab der Position `schwanz` kann man jedoch nur noch `N-n` Werte in dem Array speichern.

Lösung: Verwende ein sogenanntes zirkuläres Array, d.h. schreibe die Datenwerte, die hinter `schwanz` im Array keinen Platz mehr finden, in die freien Plätze vor `kopf`. Anschaulich bedeutet dies, daß man sich das Array zu einem Kreis zusammengebogen vorstellt. Alle Indexoperationen werden modulo `N` ausgeführt; ist z.B. `schwanz = n` und werden `N-n` weitere Elemente in die Schlange eingegeben, so ist dann `schwanz = N-n+n` modulo `N` = 0.



Die leere Schlange ist durch `laenge`=0 gekennzeichnet und die Schlange maximaler Länge `MAXLENGTH` durch `laenge`=`MAXLENGTH`. Anhand der Indizes `kopf` und `schwanz`

kann man die leere und die volle Liste nicht unterscheiden, weil in beiden Fällen `kopf=schwanz` gilt.

10.4.1 Eine Implementierung in C

W sei der Wertebereich eines Datentyps `TYP`, der durch eine `#define`-Anweisung festgelegt wird.

```
#include <stdio.h>
#include <stdlib.h>

#define TYP int         /* oder irgendetwas anderes */
#define MAXLENGTH 1000  /* oder irgendetwas anderes */

/**** Definition des Typs Schlange ****/

typedef struct {
   TYP data[MAXLENGTH];
   int kopf;
   int laenge;
} schlange_t;

typedef enum {false=0, true=1, fehlerfrei=0, fehlerhaft=1} boolean;

/******* make_empty *******/

void make_empty(schlange_t *zeiger) {
   zeiger->laenge = 0;
}

/******** is_empty ********/

boolean is_empty(schlange_t *zeiger) {
   if(zeiger->laenge == 0) return true;
   else return false;
}

/********** front *********/

boolean front(schlange_t *zeiger, TYP *wert) {
   if(zeiger->laenge != 0) {
      *wert = zeiger->data[zeiger->kopf];
      return fehlerfrei;
   }
   else return fehlerhaft;
}
```

```
/********** enqueue *******/

boolean enqueue(schlange_t *zeiger, TYP *wert) {
   if(zeiger->laenge < MAXLENGTH) {
      zeiger->data[zeiger->kopf + zeiger->laenge] = *wert;
      zeiger->laenge = (zeiger->laenge + 1);
      return fehlerfrei;
   }
   else return fehlerhaft;
}

/********** dequeue *******/

boolean dequeue(schlange_t *zeiger) {
   if(zeiger->laenge != 0) {
      zeiger->kopf = (zeiger->kopf + 1)%MAXLENGTH;
      zeiger->laenge -= 1;
      return fehlerfrei;
   }
   else return fehlerhaft;
}

/******* Konstruktor ******/

boolean create(schlange_t **dzeiger) {
   *dzeiger = (schlange_t *) malloc(sizeof(schlange_t));
   if(*dzeiger == NULL) return fehlerhaft;
   else {
      (*dzeiger)->kopf = 0;
      (*dzeiger)->laenge = 0;
      return fehlerfrei;
   }
}

/******* Destruktor *******/

void destroy(schlange_t *zeiger) {
   free(zeiger);
}
```

Kapitel 11

Stapel

In vielen Algorithmen wird ein Zwischenspeicher benötigt, in den man Werte einlesen und dann in der umgekehrten Reihenfolge wieder auslesen kann. Das ist das LIFO-Prinzip (Last In - First Out). Der zuletzt eingegebene Wert wird als erster wieder herausgeholt, wie bei einem Stapel Tabletten in einem Röhrchen, wo immer nur die oberste hingelegt oder weggenommen werden kann.

11.1 Der abstrakte Datentyp **Stapel** (oder Keller, LIFO, Stack) über einer Menge W

SORTEN:	S
METHODEN:	$\mathsf{push}\colon S \times W \to S, \quad \mathsf{pop}\colon S \to S, \quad \mathsf{top}\colon S \to W$
	$\mathsf{is\_empty}\colon S \to \{\mathtt{true},\mathtt{false}\}, \quad \mathsf{make\_empty}\colon S \to S$
AXIOME:	(A1) Für jedes $s \in S$ gilt $\mathsf{is\_empty}(\mathsf{make\_empty}(s)) = \mathtt{true}$
	(A2) Für alle $s \in S$ und $w \in W$ gilt
	$\mathsf{is\_empty}(\mathsf{push}(s, w)) = \mathtt{false}$
	(A3) Für alle $s \in S$ und $w \in W$ gilt
	$\mathsf{pop}(\mathsf{push}(s, w)) = s$
	(A4) Für alle $s \in S$ und $w \in W$ gilt
	$\mathsf{top}(\mathsf{push}(s, w)) = w$

11.2 Ein Modell des ADT **Stapel über W**

Seien $S = W^*$ und ϵ das leere Wort.
$\mathsf{push}\colon S \times W \to S, (s, w) \mapsto sw$.

pop: $S \to S$ sei definiert durch

$$\mathsf{pop}(s) = \begin{cases} w_1 \dots w_{n-1} , & \text{wenn } s = w_1 \dots w_n, n \geq 1 \\ \epsilon , & \text{wenn } s = \epsilon \end{cases}$$

top: $S \to W$ sei definiert durch

$$\mathsf{top}(s) = \begin{cases} w_n , & \text{wenn } s = w_1 \dots w_n, n \geq 1 \\ \text{irgendein } w \in W , & \text{wenn } s = \epsilon \end{cases}$$

is_empty: $S \to \{\texttt{true}, \texttt{false}\}$ sei definiert durch

$$\mathsf{is\_empty}(s) = \begin{cases} \texttt{true} , & \text{wenn } s = \epsilon \\ \texttt{false} , & \text{sonst} \end{cases}$$

make_empty: $S \to S$, $\mathsf{make\_empty}(s) = \epsilon$ für jedes $s \in S$.

Man rechne nach, daß alle Axiome erfüllt sind.

11.3 Implementierung als Liste

Stapel lassen sich als einfach verkettete Listen realisieren. Es ist allerdings nicht günstig, die Operationen push und pop durch Anhängen bzw. Löschen eines Elementes am Ende der Liste zu verwirklichen. Denn bei der Operation pop muß nach dem Entfernen des letzten Elementes der Positionszeiger um eine Position nach links verschoben werden. Diese back-Operation ist aber bei einfach verketteten Listen relativ aufwendig. Deshalb ist es besser, wenn man push und pop durch das Einfügen bzw. Entfernen des Elementes am Kopf der Liste realisiert. Denn dann muß der Positionszeiger nie verändert werden, er zeigt immer auf das Kopfelement der Liste und ist somit sogar überflüssig.

11.4 Implementierung höhenbeschränkter Stapel als Array

Als Höhe eines Stapels (oder Länge oder Tiefe (eines Kellers)) bezeichnen wir die minimale Anzahl von pop-Operationen, die ausgeführt werden müssen, um ihn leer zu machen.

In dem Modell 11.2 ist die Höhe des Stapels s gleich der Länge des Wortes s.

In praktischen Anwendungen beschränkt man sich oft auf Stapel, deren Höhe eine vorgegebene Schranke `MAXHEIGHT` nicht überschreitet. Dann kann man im wesentlichen das Modell 11.2 implementieren, wobei man sich auf Wörter einer Länge $\leq$ `MAXHEIGHT` beschränkt. Jedes Wort wird durch ein Array der Länge `MAXHEIGHT` und einen `int`-Wert, der die Länge des Wortes angibt, beschrieben.

11.4.1 Eine Implementierung in C

W sei der Wertebereich eines Datentyps `TYP`, der durch eine `#define`-Anweisung festgelegt wird.

```
#include <stdio.h>
#include <stdlib.h>

#define MAXHEIGHT 1000      /* oder irgendetwas anderes */
#define TYP int             /* oder irgendetwas anderes */

/********* Stapeltyp **********/

typedef struct {
   TYP data[MAXHEIGHT];
   int hoehe;
} stapel_t;

typedef enum { false=0, true=1, fehlerfrei=0, fehlerhaft=1 } boolean;

/********* make_empty *********/

void make_empty(stapel_t *zeiger) {
   zeiger->hoehe = 0;
}

/********* is_empty ***********/

boolean is_empty(stapel_t *zeiger) {
   if(zeiger->hoehe == 0) return true;
   else return false;
}

/************ top *************/

boolean top(stapel_t *zeiger, TYP *w) {
   if(zeiger->hoehe > 0) {
      *w = zeiger->data[(zeiger->hoehe)-1];
      return fehlerfrei;
   }
   else return fehlerhaft;
}

/************ pop *************/

void pop(stapel_t *zeiger) {
   if(zeiger->hoehe > 0) zeiger->hoehe -= 1;
}

/*********** push *************/

boolean push(stapel_t *zeiger, TYP *w) {
```

```
   if(zeiger->hoehe < MAXHEIGHT) {
      zeiger->data[zeiger->hoehe] = *w;
      zeiger->hoehe += 1;
      return fehlerfrei;
   }
   else return fehlerhaft;
}

/******** Konstruktor *********/

boolean create(stapel_t **zeiger) {
   *zeiger = (stapel_t *)malloc(sizeof(stapel_t));
   if(*zeiger == NULL) return fehlerhaft;
   else {
      (*zeiger)->hoehe = 0;
      return fehlerfrei;
   }
}

/******** Destruktor **********/

void destroy(stapel_t *zeiger) {
   free(zeiger);
}
```

11.5 Beispiele für die Verwendung von Stapeln

11.5.1 Auswertung von Funktionsaufrufen

Üblicherweise verwendet jedes Mikroprozessorsystem einen Stapel, um verschachtelte Funktionsaufrufe zu realisieren. Bei einem Funktionsaufruf werden die neuen lokalen Variablen auf den Stapel gelegt und am Ende des Aufrufs wieder in umgekehrter Reihenfolge entfernt. Während der Funktionsausführung braucht man aber Schreib- und Leseoperationen, die im ADT Stapel nicht vorgesehen sind.

11.5.2 Interrupts

Ähnlich wird auf Maschinensprachebene bei sogenannten Interrupts verfahren. Das sind Unterbrechungen der momentanen Programmausführung durch äußere Signale, deren Auftreten nicht unbedingt vorhersehbar ist (z.B. wenn Peripheriegeräte eine Bedienung anfordern). Dann müssen die momentanen Inhalte der Rechenspeicher im Prozessor (Register) vorübergehend gerettet werden, damit nach Beendigung der Interruptbehandlung das ursprüngliche Programm fortgesetzt werden kann. Dies geschieht meist dadurch, daß man die Registerinhalte auf einen Stapel schreibt. Daher besitzen übliche Mikroprozessoren bereits hardwaremäßig ein Stackpointerregister (also einen Speicher für einen Zeiger auf einen Stapel) zur einfachen Realisierung eines Stapels.

11.5.3 Kellerautomaten

In der Theorie der formalen Sprachen kommen hypothetische Maschinen, sogenannte Kellerautomaten, vor, die als wesentlichen Bestandteil einen Stapel besitzen.

11.5.4 Durchlaufen von Graphen und Bäumen

Graphen und Bäume sind Datenstrukturen, die in vielen Anwendungen eingesetzt werden; wir werden sie in späteren Kapiteln einführen. Zum systematischen Durchlaufen ihrer Bestandteile, den Knoten und Kanten, benutzt man ein Verfahren, das auf einem Stapel basiert und Tiefensuche genannt wird; alternativ kann man auch eine Schlange verwenden und erhält dann die sogenannte Breitensuche.

Kapitel 12

Objektorientierte Programmierung

Zunächst soll skizziert werden, wie aus den Problemen, die bei prozeduraler Programmierung auftreten, die Grundideen der objektorientierten Programmierung entstanden sind.

12.1 Prozedurale Programmierung

Typische prozedurale Programmiersprachen wie C, Pascal, Fortran wurden so entworfen, daß sich die elementaren Einzelschritte von Algorithmen möglichst unmittelbar durch Konstrukte der Sprache ausdrücken lassen. Dabei dachte man vor allem an Algorithmen, die numerische Berechnungen anstellen. Dementsprechend sind die Programme in solchen Sprachen eine Sammlung von Prozeduren oder Funktionen. Die Datenobjekte sind eigentlich nur Speicher, in denen Daten stehen, die von den Funktionen verarbeitet werden. Die prozedurale Programmierung stellt die Prozeduren völlig in den Vordergrund. Ein Programm wird also wesentlich durch die Anordnung der Prozeduren strukturiert.

Für kleinere Programme und numerische Algorithmen ist dieser Zugang natürlich und zufriedenstellend. Für komplexere Projekte und für Aufgaben, die nicht numerisch beschrieben sind wie z.B. die Gestaltung graphischer Benutzeroberflächen, ist der Einsatz prozeduraler Programmierung mit typischen Problemen behaftet:

1. Man benötigt viele Variablen, oft auch globale; dadurch kommt es leicht zu unbeabsichtigten Verwechslungen, insbesondere zu unerwünschten Kollisionen zwischen lokalen und globalen Variablen gleichen Namens. Dies wird bei C dadurch begünstigt, daß globale und lokale Variablen meist an weit voneinander entfernten Stellen des Quelltextes definiert sind.

2. Die Anzahl der Funktionen kann recht groß werden, und es ist schwierig, durch entsprechende Namensgebung deutlich zu machen, welche Funktion welche Da-

ten wie verarbeitet. Dies trifft vor allem auf C-Programme zu, in denen meist intensiver Gebrauch von fremden Funktionsbibliotheken gemacht wird.

12.2 Modulare Programmierung

Die Grundidee der modularen Programmierung ist die **Abkapselung** (encapsulation, information hiding) von Daten und Funktionen zur Verarbeitung eben dieser Daten in jeweils ein Modul. Z.B. könnte man zur Realisierung eines Stapels eine geeignete Datenstruktur und die üblichen Stapelfunktionen in ein Modul packen.

Ein modulares Programm ist eine Abfolge von solchen Modulen. Die Daten und die Funktionen eines Moduls können in anderen Modulen nicht gesehen oder verwendet werden, außer wenn dies explizit durch entsprechende Export- und Import-Anweisungen erlaubt wird.

C ermöglicht modulare Programmierung, indem es erlaubt, ein Programm in mehrere Quelldateien zu unterteilen und durch die Verwendung der Modifier `extern` und `static` die Sichtbarkeit von Daten und Funktionen zu steuern. Allerdings ist diese Steuerung recht grob, weil man z.B. nur festlegen kann, daß eine Variable außerhalb des eigenen Moduls in allen anderen sichtbar ist; eine Beschränkung der Sichtbarkeit auf einige bestimmte Module ist nicht möglich.

Andere Sprachen wie Modula-2 dagegen unterstützen modulare Programmierung in dem Sinne, daß sie spezielle Konstrukte zur Definition von Modulen und zur Kontrolle der Sichtbarkeit von Daten und Prozeduren enthalten und überdies erzwingen, das Programm modular zu strukturieren.

12.3 Abstrakte Datentypen

Ein Modul besteht aus Datenobjekten und Funktionen, die darauf definiert sind, in ähnlicher Weise, wie abstrakte Datentypen aus Sorten und Methoden bestehen. Man kann Module in etwa als Implementierung von abstrakten Datentypen auffassen.

Durch modulare Programmierung kann man also eine auf Typen basierende Strukturierung des Programms einführen. Allerdings unterscheiden sich diese abstrakten Typen im Gebrauch von den in der Sprache vorhandenen elementaren Typen, weil sie nicht vom Compiler unterstützt werden. So kann man z.B. nicht einfach Variablen oder Funktionsargumente mit diesen Typen definieren. Man beachte, daß die in C übliche Praxis, mit `typedef` und Structs neue Typen zu definieren, nicht das gleiche, sondern viel weniger ist, denn dabei bleiben die zugehörigen Funktionen völlig unberücksichtigt. Die Idee von abstrakten Datentypen lebt aber gerade von der unauflösbaren Verbindung der Datenobjekte und der Operationen auf ihnen. Die Programmiersprache sollte daher so gestaltet sein, daß sie bereits durch ihre Syntax erzwingt, Daten eines Typs möglichst nur mit den Methoden dieses Typs zu bearbeiten. Und der Compiler sollte dies nachprüfen können. Dies ist bei C absolut nicht der Fall. Wie gut ein C-Programm den obigen Ideen entspricht, hängt allein von der Disziplin des Programmierers ab. Die Sprache erlaubt z.B. durch die cast-Operation fast jeden Unfug.

Die Idee abstrakter Datentypen, die vom Programmierer definiert werden, läßt sich mit modularer Programmierung nur unvollständig umsetzen. Denn ein Modul beschreibt nur eine spezielle Instanz eines abstrakten Datentyps und nicht den abstrakten Datentyp selbst. Ein Modul, das einen Stapel mit Elementtyp `int` realisiert, kann nicht verwendet werden, um einen Stapel mit anderem Elementtyp zu realisieren. Prozedurale Sprachen haben eben keine syntaktischen Konstrukte, mit denen man das abstrakte Schema eines Stapel definieren kann, dessen Elementtyp erst bei der Erzeugung einer konkreten Instanz festgelegt wird. C ist zwar so flexibel, daß es geschickten Programmierern gelingt, solche Ideen umzusetzen (z.B. mit Hilfe von Zeigern auf Funktionen), nur handelt es sich dabei um proprietäre Lösungen, die nicht unbedingt übersichtlich und leicht verständlich sind.

12.4 Objektorientierte Programmierung

Objektorientierte Programmierung erweitert das Konzept der Abkapselung durch die Konzepte Abstraktion, Vererbung und Polymorphismus.

Unter **Abstraktion** versteht man die Möglichkeit, eine abstrakte Beschreibung von Daten und Operationen auf diesen Daten in einen Programmteil, eine sogenannte Klasse, zu packen. Die Klassen kann man sich als benutzerdefinierte abstrakte Datentypen vorstellen. Die Operationen einer Klasse werden als ihre Methoden bezeichnet.

Unter **Vererbung** (inheritance) versteht man die Möglichkeit, eine Klasse B von einer Klasse A abzuleiten, indem zu den Daten und Methoden von A noch weitere hinzugenommen werden. Die Subklasse B beschreibt sozusagen eine Teilmenge der Superklasse. Dies erspart, denselben Code an mehreren Programmstellen schreiben zu müssen, wie das bei modularer Programmierung nötig ist. Dadurch werden die Programme wesentlich übersichtlicher und gleichzeitig weniger anfällig gegenüber inkonsistenten Änderungen.

Unter **Polymorphismus** versteht man zum einen die Möglichkeit, eine Methode einer Subklasse anders zu implementieren als die gleichnamige Methode der Superklasse (Überschreiben einer Methode), zum anderen die Möglichkeit, Methodenbezeichner zu **überladen**; damit ist gemeint, daß zur Unterscheidung von Methoden nicht nur ihre Namen herangezogen werden (wie bei C), sondern auch ihre Signatur d.h. Anzahl und Typ der Argumente. Dies ist aus der Mathematik wohlvertraut, z.B bezeichnet das Pluszeichen `+` völlig unterschiedliche Operationen: Vorzeichen (unär), Addition in $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ (binär), Addition von Vektoren oder Matrizen (binär). Achtung! JAVA gestattet zwar das Überladen von Methoden, aber nicht das Überladen der elementaren Operatoren wie `+`; C++ dagegen schon.

So wie man in C-Programmen Objekte von einem vorgegebenen Typ erzeugen kann, kann man Objekte einer Klasse erzeugen. Zur Erzeugung wird eine spezielle Methode der Klasse, ein Konstruktor, aufgerufen, die Speicherplatz für die in der Klasse aufgeführten Daten reserviert und eventuell eine Initialsierung vornimmt.

Der innere Aufbau einer Klasse ähnelt einem prozeduralen Programm. Eine Klasse besteht aus Definitionen von Variablen und von Funktionen, die Methoden genannt werden. In JAVA oder C++ ist die Syntax der Methoden sehr ähnlich zu C. Jedes

Programm in C++ oder JAVA enthält genau eine Klasse, in der es eine Methode namens `main` gibt. Die Ausführung eines Programms beginnt immer mit der Ausführung dieser Methode.

Die Sichtbarkeit jeder Methode kann durch Modifier gesteuert werden; sie kann z.B. auf ein Objekt beschränkt werden oder auf die ganze Klasse, eventuell einschließlich von Subklassen, oder sogar auf alle Klassen ausgedehnt werden. Ähnlich kann man die Sichtbarkeit der Daten kontrollieren.

Objektorientiertes Programmieren erfordert eine andere Denkweise als prozedurales Programmieren. Ein prozedurales Programm ist eine hierarchische Abfolge von Funktionsaufrufen; ein objektorientiertes Programm ist eine Hierarchie von Klassen und Objekten, durch die eine Abfolge von Methodenaufrufen definiert wird.

12.5 Objektorientierung in JAVA

Ein JAVA-Programm ist eine Folge von Klassen, abstrakten Klassen, Interfaces und Generics. Eine Klasse kann man als Implementierung eines abstrakten Datentyps auffassen, während abstrakte Klassen und Interfaces eher abstrakten Datentypen entsprechen und Generics Klassen sind, denen man einen Datentyp als Parameter übergeben kann, so dass man Algorithmen so implementieren kann, dass sie mit unterschiedlichen Datentypen arbeiten können (z.B. Sortieralgorithmen). Wir geben einen ersten Überblick.

12.5.1 Die allgemeine Form einer Klasse

```
class classname {
        // deklariere Variablen
        type var1;
        type var2;
        //...
        type varN;

        // deklariere Methoden
        returntype method1(parameterlist) {
                // Methodenkörper
        }
        returntype method2(parameterlist) {
                // Methodenkörper
        }
        //...
        returntype methodM(parameterlist) {
                // Methodenkörper
        }
}
```

Eine Klasse ist ähnlich aufgebaut wie eine prozedurales Programm. Die Variablen sind Datenobjekte, und die Methoden sind Prozeduren zur Verarbeitung von Daten ähnlich wie die Funktionen in einem C-Programm. Auch syntaktisch sind Methoden

den C-Funktionen sehr ähnlich; mehr dazu später. In einem JAVA-Programm gibt es genau eine Klasse, die eine Methode namens `main` enthält. Mit dieser Methode beginnt die Programmausführung.

Die Sichtbarkeit der Variablen und Methoden einer Klasse außerhalb von ihr kann mit den Schlüsselwörtern `public` und `private`, die vor die Deklaration gesetzt werden, gesteuert werden.

12.5.2 Vererbung und Polymorphismus Man kann eine Klasse A durch Hinzunahme von Variablen und Methoden zu einer Klasse B erweitern. B heißt dann eine Subklasse der Superklasse A. Die syntaktische Form von B ist dieselbe wie die obige Klassendefinition, lediglich die Kopfzeile ist abgeändert.

`class` *B* `extends` *A* {
    //Körper der Klasse B
}

Eine Subklasse kann i.a. auf die Variablen und Methoden der Superklasse zugreifen; durch den Modifier `private` vor einer Variablendeklaration in der Superklasse kann dies aber verhindert werden. Durch `protected` wird erreicht, dass eine Komponente in jeder Subklasse, aber nicht in anderen Klassen sichtbar ist.

Eine Subklasse kann Methoden der Superklasse durch eine eigene Definition überschreiben; auch dies kann durch den Modifier `final` vor der Methodendeklaration in der Superklasse verhindert werden. Achtung! In JAVA kann eine Klasse nur Subklasse einer einzigen anderen Klasse sein! Damit werden von vorn herein Konflikte bei der Vererbung von gleichnamigen Methoden vermieden.

12.5.3 Abstrakte Klassen Eine **abstrakte Klasse** ist wie eine Klasse aufgebaut; nur muss nicht jede ihrer Methoden implementiert sein, d.h. einige Methodenkörper können leer sein. Abstrakte Klassen nehmen also eine Zwischenstellung zwischen Abstrakten Datentypen und den Implementierungen abstrakter Datentypen ein. Subklassen abstrakter Klassen werden wie bei nicht abstrakten Klassen gebildet.

`abstract class` *classname* {
    //Klassenkörper
}

12.5.4 Schnittstellen,Interfaces Dass eine Klasse nur Subklasse einer einzigen anderen Klasse sein kann, ist manchmal eine unbequeme Einschränkung. Sie kann man umgehen durch sogenannte Interfaces. Sie sind syntaktisch wie eine abstrakte Klassen aufgebaut, in der alle Methoden nur deklariert, aber nicht implementiert sind, also leere Methodenkörper haben. Im Unterschied zu einer abstrakten Klasse können die deklarierten Variablen zwar initialisiert, aber nicht verändert werden; sie sind also praktisch Konstanten. Die allgemeine Form ist folgende.

`interface` *interfacename* {
    // deklariere Variablen
    type var1 = value1;
    type var2 =value2;
    //...
    type varN = valueN; // deklariere Methoden
    returntype method1(parameterlist);

```
    returntype method2(parameterlist);
    //...
    returntype methodM(parameterlist);
}
```

Interfaces kann man grob gesprochen als abstrakte Datentypen ohne Axiome auffassen. Ein Interface kann von einer oder mehreren Klassen implementiert werden, d.h. die Klassen enthalten Implementierungen der im Interface aufgeführten Methoden. Eine Klasse kann mehrere Interfaces implementieren; zu Konflikten zwischen Methoden gleichen Namens und gleicher Signatur in den unterschiedlichen Interfaces kann es nicht kommen, da die Implementierung ja erst in der Klasse selbst angegeben ist. Die allgemeine Form einer Klasse, die eine andere Klasse erweitert und mehrere Interfaces implementiert, ist folgende.

```
class classname extends superclassname implements interfacename {
    // Klassenkörper
}
```

Auch eine abstrakte Klasse kann ein oder mehrere Interfaces implementieren.

12.5.5 Objekte Klassen kann man als Implementierungen abstrakter Datentypen auffassen. Deshalb kann man Objekte erzeugen, deren Typ eine Klasse ist, sog. Instanzen einer Klasse. So ein Objekt besteht aus neuem Speicherplatz für die Variablen der Klasse und Referenzen auf die Methoden. Obwohl JAVA den Begriff Zeiger vermeidet, werden Objekte intern durch Referenzen, also Zeiger, adressiert. Allerdings kann man mit diesen Zeigern nicht wie bei C herumrechnen; es gibt keine Zeigerarithmetik.

12.5.6 Pakete, Packages

So wie man in C Funktionen zu Bibliotheken zusammenfassen kann, kann man in JAVA Klassen zu sogenannten Paketen (Packages, Klassenbibliotheken) zusammenfassen. Dadurch dass dann die Daten und die zugehörigen Methoden immer zusammenbleiben, gewinnt man gegenüber C-Bibliotheken eine bessere Strukturierung und Klarheit, welche Methode mit welchen Daten arbeiten darf. Andererseits ergibt sich dadurch insgesamt eine riesige Anzahl von Methoden, die man kennen muss. Diese Vielzahl an Methoden ist für einen Anfänger schwer zu überblicken, vor allem wenn die Methoden bei Vererbung überschrieben wurden und sich vielleicht nicht erwartungsgemäß verhalten. Ein C-Programm besteht demgegenüber meist aus einer relativ kleinen Anzahl von Bibliotheksfunktionen, aus denen alles zusammengesetzt wird. Für den Einstieg ins Programmieren ist dies leichter.

Die Sichtbarkeit von Klassen und Schnittstellen in anderen Paketen kann durch die Schlüsselwörter `public` und `private` gesteuert werden.

Eine Entwicklungsumgebung für `JAVA` enthält viele, großen Klassenbibliotheken. Um da den Überblick zu behalten, ist man fast dazu gezwungen, eine gute grafische Programmentwicklungsumgebung wie NetBeans oder Eclipse zu benutzen. Mit ein paar man-pages wie bei C kommt man nicht mehr aus.

12.5.7 UML-Diagramme

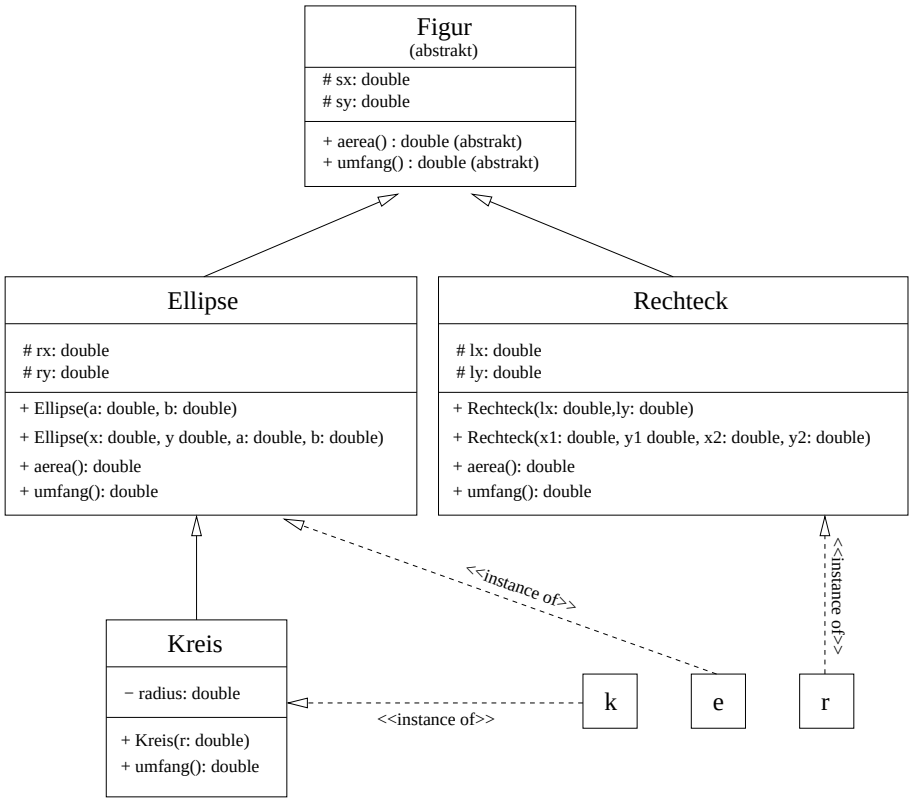
Zur Beschreibung objektorientierter Sprachen wurde die Unified Modeling Language, kurz UML, entwickelt. Klassen und Objekte werden durch Rechtecke dargestellt, in denen die Komponenten aufgelistet werden. Durch Voranstellen eines der Zeichen

`+,-,#` werden die Zugriffsmodifier `public, private, protected` angegeben. Vererbung wird durch einen Pfeil von der Subklasse zur Superklasse symbolisiert, und dass ein Objekt eine Instanz einer Klasse ist, durch einen gestrichelten Pfeil von dem Objekt zur Klasse.

12.5.8 Ein Programmbeispiel: `FigurTest`

Das folgende Beispiel soll einen ersten Eindruck geben, wie ein einfaches JAVA-Programm aus mehreren Klassen aufgebaut ist. Darin kommen einige Konstrukte vor, die wir bisher nicht besprochen haben, deren prinzipielle Bedeutung leicht ersichtlich ist. Mehr dazu wird im nächsten Kapitel gesagt.

Zur Darstellung ebener Figuren wird zunächst eine Klasse `Figur` definiert, die als abstrakt erklärt ist, weil sie nicht implementierte Methoden enthält. Von dieser Klasse werden Subklassen abgeleitet, in denen die Methoden dann implementiert werden. Die Klasse `Kreis` ist eine Subklasse der Klasse `Ellipse`, die wiederum eine Subklasse der Klasse `Figur` ist. Weil die Berechnung des Umfangs einer Ellipse auf ein elliptisches Integral führt, das nicht elementar ausgedrückt werden kann, wurde die Methode `umfang()` nur als Ausgabe einer Fehlermeldung implementiert. In der Subklasse `Kreis` wird sie dann durch die Berechnung des Kreisumfangs überladen. Die Programmausführung beginnt mit der Methode `main` in der Klasse `FigurTest`. Die gesamte Datei muß denselben Basisnamen wie die Klasse, in der `main` steht, haben und die Endung `java`; sie muß also zwingend `FigurTest.java` heißen.



```
// JAVA - Testprogramm

// Definiere eine abstrakte Klasse,
// deren Methoden nicht implementiert sind.
//
abstract class Figur {
   protected double sx, sy;        // Schwerpunkt
   public abstract double area();
   public abstract double umfang();
}

// Definiere die Klasse Ellipse als Subklasse von Figur.
// Die Methode umfang() besteht nur aus einer Fehlermeldung,
// weil zur exakten Berechnung ein elliptisches Integral noetig ist.
//
class Ellipse extends Figur {
   protected double rx, ry;

   public Ellipse( double x, double y, double a, double b ) {
      sx = x; sy = y;
      rx = a; ry = b;
   }
   public Ellipse( double a, double b ) {
      sx = 0; sy = 0;
      rx = a; ry = b;
   }
   public double area() {
      return Math.PI*rx*ry;
   }
   public double umfang() {
      System.out.println( "Nicht richtig implementiert!" );
      return -1;
   }
}

// Definiere die Klasse Kreis als Subklasse von Ellipse.
// Die Methode umfang() der Klasse Ellipse wird ueberschrieben.
//
class Kreis extends Ellipse {
   private double radius;

   public Kreis( double x, double y, double r ) {
      super(x,y,r,r);
      radius = r;
   }

   public Kreis( double r ) {
```

```
      super(r,r);
      radius = r;
   }

   public double umfang() {
      return 2*Math.PI*radius;
   }
}

// Definiere die Klasse Rechteck als Subklasse von Figur.
// Die Methode umfang() der Klasse Figur wird ueberschrieben,
// ebenso die Methode area().
//
class Rechteck extends Figur {
   protected double x1,y1,x2,y2;
   protected double lx,ly;

   public Rechteck( double x1, double y1, double x2, double y2 ) {
      sx = 0.5*(x1+x2); sy = 0.5*(y1+y2);
      lx = Math.abs(x2-x1); ly = Math.abs(y2-y1);
      this.x1 = x1; this.y1 = y1;
      this.x2 = x2; this.y2 = y2;
   }
   public Rechteck( double lx, double ly ) {
      sx = 0; sy = 0;
      this.lx = Math.abs(lx); this.ly = Math.abs(ly);
      x1 = -0.5*this.lx; y1 = -0.5*this.ly;
      x2 =  0.5*this.lx; y2 =  0.5*this.ly;
   }
   public double umfang() {
      return 2*(lx+ly);
   }
   public double area() {
      return lx*ly;
   }
}

// Die folgende Klasse enthaelt die Methode main();
// mit ihr beginnt die Programmausf"uhrung.
// Ihr Name muss mit dem Namen der Datei uebereinstimmen
// (ohne Endung .java)
//
public class FigurTest {
   public static void main( String args[] ) {
     Kreis k = new Kreis(1);
     Ellipse e = new Ellipse(1,2);
     Rechteck r = new Rechteck(2,3);
```

```
    System.out.println("Kreisflaeche = " + k.area() );
    System.out.println("Kreisumfang  = " + k.umfang());
    System.out.println("Ellipsenflaeche = " + e.area());
    System.out.println("Rechteckflaeche = " + r.area() );
    System.out.println("Rechteckumfang  = " + r.umfang());
  }
}
```

Kapitel 13

JAVA-Grundlagen

JAVA ist eine Programmiersprache, die zwar objektorientiert ist, aber an vielen Stellen die Objektorientiertheit durchbricht und C oder C++ ähnelt. Wir werden daher hier vor allem Unterschiede zu C erwähnen. Leider entstehen bei JAVA durch die nicht konsequente Objektorientierung einige unerwartete, unschöne Artefakte. Gegenüber C, das recht schlank ist, was die Schlüsselwörter und die Standardfunktionen angeht, enthält schon die Standardbibliothek von JAVA eine Unzahl von Klassen und Methoden, wodurch die Sprache recht unübersichtlich und für Anfänger ungeeignet wird. Ob JAVA für die Verarbeitung großer Datenmengen geeignet ist und ob die Grafik in JAVA in der Praxis einem Vergleich mit Grafikbibliotheken in C++ standhält, mag jeder für sich selbst entscheiden.

13.1 Kompilieren und Ausführen eines Programms

Ein JAVA-Programm besteht aus einer oder mehreren Textdateien, die Definitionen von Klassen, abstrakten Klassen, Interfaces oder Generics enthalten.

Wir betrachten zunächst den Fall, dass das Programm aus nur einer Datei besteht. In dieser Datei muss es genau eine Klassendefinition geben, die eine Methode namens `main` enthält. Bei Start des Programms beginnt die Programmausführung mit dieser Methode. Damit dies möglich ist, muss diese Methode mit den Modifiern `public static` deklariert sein; sie bedeuten, dass diese Methode auch außerhalb des Klassenkörpers zu sehen ist und dass sie aufgerufen werden kann, ohne dass zuvor ein Objekt vom Typ der Klasse erzeugt wurde.

Innerhalb einer Datei darf es nur eine Klasse geben, die als `public` erklärt ist. Und die Datei muss im Gegensatz zu C-Programmen zwingend denselben Namen wie dieser Klasse und überdies die Extension `java` haben.

13.1.1 Beispiel 1

```
public class HelloWorld {
   public static void main(String args[]) {
      System.out.println("Hello World!");
   }
}
```

Dieser Programmtext muss in einer Datei namens `HelloWorld.java` abgespeichert werden. Wie schon früher erwähnt wird ein JAVA-Programm nicht in ein ausführbares Programm in Maschincode, sondern zunächst in einen sog Bytecode übersetzt. Dies geschieht mit der Kommandozeile

`javac` *progname*`.java`

In obigem Fall also durch

```
javac HelloWorld.java
```

Dabei wird für jede Klasse in dem Programmtext eine eigene Datei erzeugt, die den Bytecode dieser Klasse enthält und als Namen den Namen der Klasse mit der Extension `class` hat. In obigem Beispiel ensteht also die Datei `HelloWorld.class`.

Dieser Bytecode ist unabhängig von der verwendeten Rechnerplattform. Um ihn auszuführen, braucht man die sog. Java Virtual Machine (JVM); das ist ein Programm, das den Bytecode Zeile für Zeile interpretiert und ausführt. Die JVM ist ein lauffähiges Programm in der Maschinensprache des verwendeten Rechners, also plattformabhängig. Der Bytecode eines JAVA-Programms ist jedoch auf jedem Rechner ausführbar, für den es eine JVM gibt. Die Ausführung des Bytecodes *progname*`.class` durch die JVM wird mit der folgenden Kommandozeile angestoßen

`java` *progname*

In obigem Beispiel also durch

```
java HelloWorld
```

Das obige Beispiel enthält nur eine Klasse, und diese Klasse besteht aus nur einer Methode, nämlich `main`. Dieser Methode können wie der C-Funktion `main` Argumentparameter von der Kommandozeile übergeben werden, und zwar als Komponenten eines Arrays namens `args`, dessen Elemente Objekte vom Typ `String` sind. In obigem Beispiel werden diese Argumentparameter nicht weiter verwendet. Es wird lediglich der Text *Hello World!* ausgegeben. Dazu wird die Methode `println` des vordefinierten Objekts `System.out` aufgerufen. An der Namensgebung wird deutlich, dass `System.out` eine Komponente der Klasse `Systems` ist. Die Methode `println` erkennt automatisch, dass sie als Argument eine Zeichenkette übergeben bekommt, und druckt sie aus.

In JAVA ist ein Array ein Objekt einer Klasse, die ein Methode `length` besitzt, die die Länge des Arrays zurück gibt. Deshalb braucht man nicht wie in C eine Variable `argc`, um zu wissen, wieviele Strings auf der Kommandozeile übergeben wurden. Vorsicht! `args[0]` ist nicht wie in C der Programmname, sondern der erste String hinter dem Programmaufruf. Dies kann man durch folgendes Beispiel demonstrieren.

13.1.2 Beispiel 2

```
public class HelloYou {
   public static void main(String args[]) {
      System.out.println("Hello " + args[0]);
   }
}
```

Nach Kompilation ergibt der Kommandozeilenaufruf `java HelloYou Volker` die Ausgabe *Hello Volker*. Diesmal war das Argument von `println` die Verkettung der Zeichenkette `"Hello "` mit dem String `args[0]`. Wie man sieht, ist in JAVA der Operator `+` überladen, so dass man ihn nicht nur für die Addition von Zahlen, sondern auch für die Verkettung von Strings benutzen kann.

13.2 Datentypen

JAVA verwendet zwei Arten von Datentypen, nämlich Referenztypen und primitive (oder native oder einfache oder eingebaute) Typen. Die *Referenztypen* sind im Wesentlichen die Klassen oder Schnittstellen (Interfaces). Sie heißen Referenztypen, weil Objekte von diesem Typ durch eine Referenz angesprochen werden, d.h. anschaulich gesprochen durch die Anfangsadresse des Speicherbereichs, in dem das Objekt liegt. Eine Variable mit so einem Typ hat als Wert diese Referenz, sprich Speicheradresse. In C wäre das ein Zeiger auf das Objekt.

Die primitiven Datentypen dienen zur Darstellung von Zahlen. Ihre Instanzierungen werden nicht als Objekte behandelt und fallen somit aus der objektorientierten Sicht heraus. Sie wurden eingeführt, um numerische Rechnungen effizienter durchführen zu können, als es mit Objekten möglich wäre. Variablen von so einem primitiven Typ werden wie in C verwendet.

13.2.1 Primitive Datentypen

Es gibt folgende primitive Datentypen.

Name	Speicherplatz	Wertebereich
`boolean`	1 Byte	{`true, false`}
`char`	2 Byte	16-Bit-Unicode-Zeichen
`byte`	1 Byte	$\{-128, \ldots, 127\}$
`short`	2 Byte	$\{-2^{15}, \ldots, 2^{15}-1\}$
`int`	4 Byte	$\{-2^{31}, \ldots, 2^{31}-1\}$
`long`	8 Byte	$\{-2^{63}, \ldots, 2^{63}-1\}$
`float`	4 Byte	
`double`	8 Byte	

Eine Besonderheit ist der Typ `boolean`, den es in C ja nicht gibt.

Achtung! Es gibt keinen 1-Byte-Datentyp ohne Vorzeichen, also kein Analogon zu `unsigned char` in C. Das ist manchmal recht nervig, besonders weil etliche Operationen den Typ `byte` automatisch zu `short` casten und zwar vorzeichengerecht, so dass die höherwertigen Bits eventuell den Wert 1 bekommen, was man möglicherweise gar nicht will.

Achtung! Der Typ `char` ist in JAVA kein 8-Bit-Zahltyp wie in C, sondern umfasst 16 Bit zur Darstellung von Unicode-Zeichen, einer Erweiterung des ASCII-Codes.

Ansonsten kann man mit diesen Datentypen so wie in C rechnen; insbesondere gibt es auch die in C üblichen Operatoren für Ganzzahltypen. Die primitiven Datentypen passen eigentlich nicht recht in ein objektorientiertes Konzept, weil die Operationen nicht mit den Daten zu Objekten zusammengepackt sind. Sie sind wohl ein Zugeständnis an die Rechengeschwindigkeit.

13.2.2 Klassentypen

Von einer Klasse kann man Instanzen erzeugen, d.h. Speicherplatz für die Datenkomponenten der Klasse und die Referenzen auf ihre Methoden reservieren. So eine Instanz nennt man dann ein Objekt, dessen Typ die gegebene Klasse ist.

Eine Variable vom Typ *classname* mit Namen *varname* wird folgendermaßen definiert

classname varname`;`

Damit wird Speicherplatz für einen Zeiger geschaffen, der auf ein Objekt vom Typ *classname* verweist. Ein Objekt vom Typ *classname* wird damit nicht erzeugt. Dies geschieht durch

varname `= new` *classname*`();`

Beispielsweise definiert das folgende Programmfragment das Element einer einfach verketteten Liste mit einem String als Datum:

```
class ListenElement {
   String datum;
   ListenElement nachfolger;
}
```

Eine einfach verkettete Liste kann man so darstellen (ohne Methoden)

```
class Liste {
   private ListenElement kopf;
   private ListenElement position;
}
```

Der `new`-Operator erzeugt eine Instanz der Klasse *classname*, d.h. er beschafft Speicher für die Variablen der Klasse und für Referenzen auf ihre Methoden, und anschließend wird die Adresse dieses Speicherberichs der Variablen *varname* als Wert zugewiesen. In C werden mit `malloc` auf ähnliche Weise Objekte generiert und ihre Adresse einem Zeiger zugeordnet.

Die Definition einer Variablen und die Erzeugung eines durch sie referenzierten Objekts kann man zusammenfassen zu folgender Zeile

classname varname `= new` *classname*`();`

In JAVA kann gleichzeitig mit der Erzeugung eines Objekts eine Initialisierung von Variablen des Objekts erfolgen; dazu kann man innerhalb der Klasse spezielle Methoden, sog. Konstruktoren, definieren. Jede Klasse *classname* hat automatisch den Konstruktor *classname*`()`, der zwar ein Objekt erzeugt, aber keine Initialisierung seiner Komponenten vornimmt. Man kann jedoch diesen Konstruktor überladen, indem man ihm Initialisierungswerte als Argumente mitgibt.

Beispielsweise kann man in obiger Definition eines Listeenelements einen Konstruktor hinzufügen, der die Datenkomponente durch einen vorgegebenen String initialisiert.

```
class ListenElement {
   String datum;
   ListenElement nachfolger;

   ListenElement(String d) { datum = d; }
}
```

Ein neues Listenelement mit Datum *Hallo* erzeugt man mit

```
ListenElement NeuesElement = new ListenElement("Hallo");
```

13.2.3 Ummantelung und Autoboxing

Die Verwendung primitiver Datentypen widerspricht eigentlich der Idee der Objektorientierung. Sie führt auch zu Problemen, wenn Methoden Klassenobjekte als Argumente erwarten und zurückgeben. Deshalb gibt es in JAVA sogenannte Ummantelungsklassen (wrapper classes) `Byte, Short, Integer, Long, Double, Float, Boolean, Character`, die als Datum einen Wert vom entsprechenden primitiven Datentyp enthalten. Außerdem enthalten sie noch einige Methoden zur Umwandlung von primitiven Datentypen in Wrapperklassen und zurück. Seit Java 1.5 nimmt der Compiler teilweise auch eine automatische Umwandlung vor (Autoboxing). Einige typische Beispiele:

```
int a = 3;
Integer A = 4;
Integer B = a;
Integer C = new Integer(a);
int b = B.intValue();
```

Autoboxing kann jedoch insbesondere im Zusammenhang mit dem Operator `==` zu sehr verwirrendem Verhalten führen (siehe Übungsaufgabe und z.B. Ullenboom: Java ist auch eine Insel, S. 200f).

Ummantelungsklassen sind nicht dafür gedacht, die primitiven Datentypen zu ersetzen, weil der übersetzte Programmcode ineffizienter wird als mit primitiven Datentypen.

13.3 Methoden

Funktionen werden in JAVA Methoden genannt. Die Deklaration entspricht der von Funktionen in C:

returntype methodname(parameterlist);

parameterlist ist eine durch Kommata getrennte Aufzählung der Argumente, die jeweils aus dem Argumenttyp und dem Argumentnamen bestehen (wie bei Variablendefinitionen).

Vor die Deklaration einer Methode kann das Schlüsselwort `static` gesetzt werden. Dies bedeutet dann, dass die Methode aufgerufen werden darf, ohne dass ein Objekt

vom Typ der Klasse erzeugt wurde. Der Aufruf erfolgt mit dem Bezeichner *classname.methodname*. Methoden, die nicht als `static` erklärt sind, können erst aufgerufen werden, wenn zuvor ein Objekt dieser Klasse erzeugt wurde. Der Aufruf erfolgt dann auch mit dem Bezeichenr *objectname.methodname*.

Als **Signatur** einer Methode bezeichnet man die Zusammenfassung des Namens und der Liste der Argumenttypen. Der Typ des Rückgabewertes gehört nicht dazu! Methoden mit unterschiedlicher Signatur werden als verschieden angesehen; zur Unterscheidung wird also nicht wie in C nur der Name herangezogen. Methoden mit gleichem Namen, aber verschiedenen Argumenttypen sind unterschiedliche Funktionen. Dies sieht man z.B. an unterschiedlichen Konstruktoren einer Klasse, die alle den Namen *classname* haben, sich aber in den Argumenten unterscheiden.

Die **Übergabe der Argumente** erfolgt bei primitiven Typen durch *call-by-value*, für alle anderen Typen durch *call-by-reference*. Bei Objekten wird also die Speicheradresse übergeben. In C deklariert man solche Argumente als Zeiger.

Da eine Methode nur einen Rückgabewert hat, ist zunächst nicht klar, wie man bewirken kann, dass eine Methode mehrere Objekte zurückgeben kann. Eine Möglichkeit besteht darin, die Objekte als Komponenten eines umfassenden Objekts zu definieren, in C als Komponenten eines Structs. In C ist es aber auch sehr üblich, dass die Rückgabe durch Funktionsargumente erfolgt, denen man bei Funktionsaufruf Zeiger auf diese Objekte übergibt. Dann kennt die Funktion den Speicherort, an dem die Objekte stehen und kann sie verändern (oder vielleicht auch erst erzeugen). Will man dies in JAVA machen, so muss man die Objekte in sogenannte Container- oder Behälterobjekte einpacken, die man als Argument übergibt. Dies sind Objekte, die nur ein anderes als Komponente enthält. Weil die Parameterübergabe per Referenz erfolgt, wird durch so ein Behälterobjekt eigentlich nur wie in C ein Zeiger auf den Objektzeiger emuliert. Man sieht: Das Konzept von Zeigern ist so inhärent, dass man zwar das Wort, aber nicht die Idee vermeiden kann.

13.3.1 Kontrollstrukturen

Die Körper von Methoden in JAVA sind den Körpern von C-Funktionen sehr ähnlich, weil es die in C üblichen Kontrollstrukturen zur Formulierung der Funktionskörper gibt auch in JAVA gibt, inbesondere die Verzweigungsstrukturen `if-else` und `switch` und die Schleifen mit `for`, `while-do` und `do-while` sowie die Abbruchanweisungen `break` und `continue`. Im Gegensatz zu C darf die Schleifenvariable auch erst in der `for`-Schleife deklariert werden, was recht praktisch ist:

`for(int i=0; i<10; i++)` *Schleifenkörper*;

13.4 Strings

Es gibt in JAVA eine Klasse mit Namen `String` im Paket `java.lang`. Ein Objekt vom Typ `String` enthält eine Zeichenkette aus Unicodezeichen vom Typ `char` und etliche Methoden zur Stringverarbeitung. Man beachte, dass man ein Objekt vom Typ `String` zwar initialisieren, aber anschließend nicht mehr verändern kann!

Hier einige Methoden:

`int length()` gibt die Länge zurück.
`char charAt(int index)` gibt das Zeichen an der Stelle *index* zurück.
`boolean equals(String str)` gibt `true` zurück, wenn das Objekt *str* die gleiche Zeichenkette enthält, sonst `false`.

13.4.1 Beispiel

```
public class Stringtest {
  public static void main(String args[]) {
    String s1 = new String("Hallo");
    String s2 = "Hello";

    if(s1.equals(s2)) System.out.println("s1 und s2 sind gleich");
    else System.out.println("s1 und s2 sind verschieden");
    if(s1.equals(args[0])) System.out.println("s1 ist gleich der Eingabe");
    if(s2.equals(args[0])) System.out.println("s2 ist gleich der Eingabe");
  }
}
```

Hier kann man auch gleich verschiedene Arten der Konstruktion mit Initialisierung sehen. Das Argument von `main` ist ein Array von Objekten vom Typ `String`. Die Komponenten können wie in C durch Indizierung angesprochen werden. Allerdings ist das Element mit Index 0 die erste Zeichenkette auf der Kommandozeile nach dem Programmaufruf; in C wäre es das Element mit Index 1.

Den Operator `+` kann man in JAVA zwar nicht durch eine eigene Methode überladen; er ist aber trotzdem bereits überladen, weil er nicht nur zur Addition von Zahlen dient, sondern auch zum Hintereinanderhängen von Strings.

Jedes Objekt in JAVA besitzt eine Methode `toString()`, die eine Zeichenkette zur Beschreibung des Objekts ausgibt. In von der Klasse `Object` abgeleiteten Klassen wird diese Methode meist überschrieben und gibt eine klassenspezifische, verständlichere Beschreibung aus.

Häufig will man einer Zahl, die als String gegeben ist, ihren Wert zuordnen. Dafür gibt es folgende Methoden, die alle statisch sind und ein Objekt vom jeweiligen Ummantelungstyp zurückgeben: `parseByte(String str)`, `parseShort(String str)`, `parseInt(String str)`, `parseLong(String str)`, `parseFloat(String str)`, `parseDouble(String str)`. Achtung! Diese Methoden gehören nicht zur Klasse `String`, sondern zur jeweiligen Ummantelungsklasse des Rückgabeobjekts. Wegen des Autoboxing kann man den Rückgabewert auch direkt einer Variablen des entsprechenden primitiven Datentyps zuweisen.

13.4.2 Beispiel

```
public class LiesInt {
  public static void main(String args[]) {
    Integer N;

    N=Integer.parseInt(args[0]);
    System.out.println(N);
```

```
    int n;
    n=Integer.parseInt(args[0]);
    System.out.println(n);
  }
}
```

Die in einem Objekt vom Typ `String` gespeicherte Zeichenkette kann nach der Initialisierung nicht mehr geändert werden. Anders ist dies bei Objekten vom Typ `StringBuffer`. Sie verhalten sich im Wesentlichen wie `String`-Objekte, lassen aber eine Veränderung der in ihnen enthaltenenen Zeichenkette zu.

13.5 Arrays

Wie in C ist auch in JAVA ein Array ein Datentyp, der mehrere Objekte gleichen Typs zu einer Einheit zusammenfasst. Gegenüber C gehören aber auch noch einige Methoden dazu, insbesondere auch Konstruktoren. Ein Array ist also kein primitver Datentyp, sondern ein Referenztyp. Die Deklaration einer Array-Variable kann folgende äquivalente Gestalten haben, wobei *type* der Elementtyp ist, der auch ein primitiver Datentyp sein kann.

type arrayname`[];`
type`[]` *arrayname*`;`

Erzeugt wird ein Array mit *anzahl* Komponenten durch

arrayname `= new` *type*`[`*anzahl*`];`

Deklaration und Erzeugung können in einer Zeile erfolgen:

type arrayname`[] = new` *type*`[`*anzahl*`];`

Dagegen ist *type arrayname*`[100];` nicht erlaubt. Allerdings kann man wie in C ein Array bei der Deklaration schon mit Werten initialisieren z.B.

`int[]` *arrayname* `= {1, 2, 3, 4};`

Achtung! Die Initialisierung erfolgt erst zur Laufzeit und kostet folglich Zeit.

Jedes Array-Objekt hat eine Komponente *arrayname*.`length`, deren Wert die Länge des Arrays ist.

Die Elemente eines Arrays der Länge n sind wie in C mit den Zahlen von 0 bis $n-1$ durchnummeriert. Der Zugriff auf das n-te Element eines Arrays erfolgt wie in C durch *arrayname*`[n]`. Bei Zugriffen auf Array-Elemente wird während der Laufzeit nachgeprüft, ob der Index des Zugriffs innerhalb des Indexbereichs des Arrays liegt. Wenn nicht, wird eine sogenannte Ausnahme (Exception) erzeugt, mit der das Programm den Fehler behandeln kann. Diese Überprüfung der Arraygrenzen kosten allerdings Zeit, und man kann sie nicht abschalten. Man erinnere sich, dass bei C keine solche Überprüfung stattfindet, was zu undurchsichtigen Fehlern während der Laufzeit führen kann (man kann aber durch Linken mit der Bibliothek `efence` ein solche Überprüfung erzwingen).

Außerdem gibt es für Arrays noch etliche Methoden, mit denen Operationen auf

Arrays durchgeführt werden können wie z.B Kopieren, Füllen, Sortieren.

Eine typische `for`-Schleife, die alle Indizes eines Arrays durchläuft, sieht folgendermaßen aus:

```
for( int n = 0; n < arrayname.length; n++) {
      // Schleifenkörper
}
```

Für so eine Schleife, die alle Elemente eines Arrays durchläuft, gibt es in JAVA (erst ab Version 1.5) ein spezielles Schleifenkonstrukt:

```
for( elementtype varname : arrayname)
```

also z.B.

```
double[] Preise = { 0.99, 3.68, 7.89};
for( double x : Preise) {
  System.out.println(x);
}
```

Echte mehrdimensionale Arrays gibt es in JAVA nicht; stattdessen verwendet man ein eindimensionales Array `A`, dessen Elementtyp wieder ein Arraytyp ist. Dabei kann die Länge (nicht der Elementtyp!) der Arrays, die die einzelnen Elemente von `A` bilden variieren; denn `A` enthält an jeder Stelle eigentlich nur eine Referenz auf das jeweilige Element. Solche Konstruktionen sind auch in C wohlbekannt; dort bildet man ein Array aus Zeigern auf jeweils ein Array, das man mit `malloc` erzeugt hat. Der Nachteil solcher Konstruktionen ist, dass die Elemente so eines "mehrdimensionalen" Arrays nicht unbedingt hintereinander in einem zusammenhängenden Speicherbereich liegen. Deshalb ist bei großen mehrdimensionalen Arrays (wie z.B. Bildern) eher anzuraten, eine großes eindimensionales Array zu erzeugen, in das man mit selbst berechneten Indizes greift.

13.6 Pakete, Packages

Will man die in einem Programm definierten Klassen auch für andere Programme als Klassenbibliothek zur Verfügung stellen, so schreibt man jede Klassendefinition in eine separate Datei, deren Basisname der Klassenname und deren Endung `java` bzw. nach der Übersetzung `class` ist. Damit die Klasse sichtbar ist, muß sie als `public` deklariert werden. Pro Datei darf nur eine Klasse `public` sein. Diese Klassendateien werden alle in einen Ordner kopiert. In obigem Beispiel `FigurTest` 12.5.8 deklarieren wir jede der Klassen `Figur, Ellipse, Kreis, Rechteck` als `public` und schreiben sie in separate Dateien `Figur.java, Ellipse.java, Kreis.java, Rechteck.java`, die wir alle in einem Ordner abspeichern. Wird der Ordner z.B. `Figuren` genannt, so fügt man in jede der Dateien in diesem Ordner als erste Zeile `package Figuren;` ein, und in die Datei `FigurTest.java` schreibt man `import Figuren.*;` als erste Zeile.

Dann kann man das Programm `FigurTest` wie gewohnt übersetzen und ausführen. Jedes andere Programm kann aber auch die im Ordner `Figuren` enthaltenen Klassen benützen.

13.6.1 Beispiel für eine Klassenbibliothek

```
// Datei Figur.java
//-----------------

package Figuren;

// Definiere eine abstrakte Klasse,
// deren Methoden nicht implementiert sind.
//
public abstract class Figur {
   protected double sx, sy;        // Schwerpunkt
   public abstract double area();
   public abstract double umfang();
}
```

```
// Datei Ellipse.java
//-------------------

package Figuren;

// Definiere die Klasse Ellipse als Subklasse von Figur.
// Die Methode umfang() besteht nur aus einer Fehlermeldung,
// weil zur exakten Berechnung ein elliptisches Integral noetig ist.
//

public class Ellipse extends Figur {
   protected double rx, ry;

   public Ellipse( double x, double y, double a, double b ) {
      sx = x; sy = y;
      rx = a; ry = b;
   }
   public Ellipse( double a, double b ) {
      sx = 0; sy = 0;
      rx = a; ry = b;
   }
   public double area() {
      return Math.PI*rx*ry;
   }
   public double umfang() {
      System.out.println( "Nicht richtig implementiert!" );
      return -1;
   }
}
```

```
// Datei Kreis.java
//-----------------

package Figuren;

// Definiere die Klasse Kreis als Subklasse von Ellipse.
// Die Methode umfang() der Klasse Ellipse wird ueberschrieben.
//
public class Kreis extends Ellipse {
   privat double radius;

   public Kreis( double x, double y, double r ) {
      super(x,y,r,r);
      radius = r;
   }

   public Kreis( double r ) {
      super(r,r);
      radius = r;
   }

   public double umfang() {
      return 2*Math.PI*radius;
   }
}
```

```
// Datei Rechteck.java
//--------------------------------

package Figuren;

// Definiere die Klasse Rechteck als Subklasse von Figur.
// Die Methode umfang() der Klasse Ellipse wird ueberschrieben,
// ebenso die Methode area().
//
public class Rechteck extends Figur {
   protected double x1,y1,x2,y2;
   protected double lx,ly;

   public Rechteck( double x1, double y1, double x2, double y2 ) {
      sx = 0.5*(x1+x2); sy = 0.5*(y1+y2);
      lx = Math.abs(x2-x1); ly = Math.abs(y2-y1);
      this.x1 = x1; this.y1 = y1;
      this.x2 = x2; this.y2 = y2;
   }
   public Rechteck( double lx, double ly ) {
      sx = 0; sy = 0;
      this.lx = Math.abs(lx); this.ly = Math.abs(ly);
```

```
      x1 = -0.5*this.lx; y1 = -0.5*this.ly;
      x2 =  0.5*this.lx; y2 =  0.5*this.ly;
   }
   public double umfang() {
      return 2*(lx+ly);
   }
   public double area() {
      return lx*ly;
   }
}

// Datei FigurTest.java
//---------------------

import Figuren.*;

// Die folgende Klasse enthaelt die Methode main();
// mit ihr beginnt die Programmausf"uhrung.
// Ihr Name muss mit dem Namen der Datei uebereinstimmen
// (ohne Endung .java)
//
public class FigurTest {
   public static void main( String args[] ) {
     Kreis k = new Kreis(1);
     Ellipse e = new Ellipse(1,2);
     Rechteck r = new Rechteck(2,3);

     System.out.println("Kreisflaeche = " + k.area() );
     System.out.println("Kreisumfang  = " + k.umfang());
     System.out.println("Ellipsenflaeche = " + e.area());
     System.out.println("Rechteckflaeche = " + r.area() );
     System.out.println("Rechteckumfang  = " + r.umfang());
   }
}
```

13.7 Eingabe von der Konsole

In den früheren Versionen von JAVA gab es keine Möglichkeit, Zahlenwerte aus einem Stream (z.B. der Tastatur) so einfach einzulesen, wie es in C mit der Funktion `fscanf` geht. Deshalb findet man in den Lehrbüchern meist eigenerstellte Klasse dafür. Ab Version 1.5 gibt es die Klasse `Scanner` im Paket `java.util`, mit der die Zeichenfolge des im Konstruktor angegebenen Streams nach gewissen Regeln zerlegt werden kann. Damit wird dann eine einfache Eingabe z.B. von Zahlen möglich. Diese Klasse hat sehr viele Methoden zum Einlesen von unterschiedlicher Zahltypen. Exemplarisch betrachten wir eine Schleife zum Einlesen von Integers. Die Methode `hasNextInt()` liefert einen Wert vom Typ `boolean` zurück, der angibt, ob in dem Stream als nächstes die Darstellung einer ganze Zahl kommt. Mit `nextInt()` wird dann diese Zahl eingelesen und ihr Wert als Objekt vom Typ `Integer` zurückgegeben. Wegen des Autoboxing

kann er auch direkt einer primitiven Integer-Variablen zugewiesen werden kann.

13.7.1 Beispiel 1 für Konsoleneingabe

```
import java.io.*;
import java.util.Scanner;
public class inputloop1 {
  public static void main( String args[] ) {
    Scanner scanner = new Scanner( System.in );
    while(scanner.hasNextInt()) {
      System.out.println(scanner.nextInt());
    }
  }
}
```

13.7.2 Beispiel 2 für Konsoleneingabe

```
import java.io.*;
import java.util.Scanner;
public class inputloop2 {
  public static void main( String args[] ) {
    Scanner scanner = new Scanner( System.in );
    while(scanner.hasNextInt()) {
      int n=scanner.nextInt();
      System.out.println(n);
    }
  }
}
```

`System.in` und `System.out` sind Objekte, die in etwa den Streams `stdin` und `stdout` bei C entsprechen. Wie wir das beim Aufruf von C-Programmen schon gemacht haben, kann man der Shell auf der Kommandozeile mit Hilfe der Zeichen `<` und `>` sagen, dass die Streams nicht mit der Konsolen-Ein-Ausgabe verbunden sein sollen, sondern aus bzw. in eine Datei umgelenkt werden sollen. So werden z.B. mit der Kommandozeile `java inputloop1 < ein.txt > aus.txt` Zahlen aus der Datei `ein.txt` gelesen und in die Datei `aus.txt` ausgegeben.

13.8 Ausnahmeverarbeitung

Fehler, die während der Laufzeit auftreten, werden Ausnahmen (exceptions) genannt. Zum einen gibt es Fehler, die in der Java Virtual Machine selbst auftreten und zum Abbruch des Programms führen. Zum anderen können Methoden Ausnahmen erzeugen, wenn bei ihrer Ausführung nicht die gewünschten Operationen ausgeführt werden können, z.B. wenn die Allozierung von Speicher nicht möglich ist oder durch Null geteilt werden soll. Für solche Ausnahmen kann man im Programm speziellen Code vorsehen, der beim Auftreten der Ausnahme ausgeführt wird.

In C wird der häufig der Rückgabewert einer Funktion verwendet, um anzuzeigen, dass bei ihrer Ausführung ein Problem aufgetreten ist. Häufig wird dann ein Nullzeiger zurückgegeben; man denke an `malloc`. An jeder solchen Stelle muss man dann

Programmcode zur Behandlung dieser Ausnahmesituation einbauen. Das kann schnell etwas unübersichtlich werden. Daher hat man in JAVA eine etwas anderes Konzept gewählt.

In JAVA erzeugt jedes Auftreten einer Ausnahme ein Objekt einer Subklasse der Klasse `Exception`. Man definiert sich dann Programmblöcke, deren Ausführung bei Auftreten einer Ausnahme unterbrochen wird; stattdessen wird das Programm dann je nach Typ der Ausnahme mit einer speziellen Fehlerbehandlung fortgeführt. Mit dem Schlüsselwort `try` wird der zu überwachende Programmblock gekennzeichnet, mit `catch` die jeweilige Fehlerbehandlung.

```
try {
      // der zu überwachende Programmblock
}
catch(exception_type1 excOb) {
      // Ausnahmebehandlung für exception_type1
}
catch(exception_type1 excOb) {
      // Ausnahmebehandlung für exception_type2
}
finally {
      // Programmcode, der immer ausgeführt wird.
}
```

Wenn im `try`-Block eine Ausnahme auftritt, geht die Programmausführung zum entsprechenden `catch`-Block über, der für die Behandlung des Ausnahmetyps zuständig ist. Falls kein entsprechender `catch`-Block da ist, wird die Ausnahmebehandlung der JVM überlassen, was meist zum Abbruch des Programms mit einer entsprechenden Fehlermeldung führt. Sowohl wenn das Programm den `try`-Block ohne Ausnahme abgearbeitet hat, als auch wenn eine Ausnahme aufgetreten ist und durch einen `catch`-Block behandelt wurde, wird anschließend die Programmausführung mit dem Code hinter dem letzten `catch`-Block fortgesetzt. Der `finally`-Block ist optional; wenn er vorhanden ist, wird er also in jedem Fall ausgeführt, egal ob eine Ausnahme aufgetreten ist oder nicht, außer natürlich, wenn eine Ausnahme aufgetreten ist, die nicht behandelt werden konnte und an die JVM weitergegeben wurde.

`try/catch`-Blöcke können ineinander geschachtelt werden. Ausnahmen, die im inneren `catch`-Block nicht behandelt werden konnten, werden an den äußeren `catch`-Block weitergegeben, nachdem eine eventuell vorhandener `finally`-Block ausgeführt wurde.

Man kann eigene Ausnahmeklassen definieren. Und man kann innerhalb einer Methode mit dem Schlüsselwort `throw` auch eigene Ausnahmen generieren. In der Deklaration einer Methode muss dann angegeben werden, welche Ausnahmen sie erzeugen kann:

```
returntype methodname( argument_list ) throws exceptions_list {
      // Methodenkörper
}
```

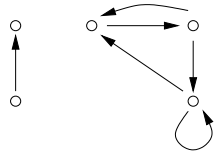
Kapitel 14

Graphen

14.1 Gerichtete Graphen

14.1.1 Definition *Ein **gerichteter Graph** (directed graph, digraph) G ist ein Paar (V, E), wobei V eine endliche Menge und $E \subset V \times V$ sind. Die Elemente von V heißen die **Knoten** (vertices) von G, die Elemente von E die **Kanten** (edges). Ein Graph (V, E) heißt Teilgraph eines Graphen (V', E'), wenn $V \subset V'$ und $E \subset E'$ gilt.*

Anschauliche Darstellung: Die Knoten werden als kleine Kreise oder Kreisscheiben gezeichnet und jede Kante (u, v) als Pfeil von dem u zugeordneten Kreis zu dem v zugeordneten.



Sprechweisen:
Für $(u, v) \in E$ sagt man auch, daß von u nach v eine Kante gehe oder daß v zu u adjazent sei. u heißt der Anfangspunkt der Kante (u, v) und v der Endpunkt. Es ist zugelassen, daß Anfangs- und Endpunkt übereinstimmen. Zwischen zwei Knoten $u, v \in V$ gibt es höchstens zwei Kanten, nämlich die Kante (u, v), die von u nach v geht, und die Kante (v, u), die von v nach u geht.
Für $u \in V$ seien

$$\text{indegree}(u) = \#\{(v, u) : (v, u) \in E\} \text{ der Eingangsgrad von } u$$

$$\text{outdegree}(u) = \#\{(u, v) : (u, v) \in E\} \text{ der Ausgangsgrad von } u$$

14.1.2 Beispiele

14.1.2.1 Transportnetze: Die Knoten sind Orte, an denen irgendwelche Materialien (z.B. Wasser, Öl, Strom) verbraucht oder erzeugt werden. Eine Kante vom Knoten u zum Knoten v gibt es genau dann, wenn eine direkter Transportweg von u nach v existiert.

14.1.2.2 Ablaufpläne: Temporale oder kausale Abhängigkeiten von Ereignissen oder Aktionen lassen sich als Graph darstellen. Die Aktionen sind die Knoten. Von einer Aktion A zu einer Aktion B gibt es genau dann eine Kante, wenn die Aktion B nach der Aktion A erfolgt bzw. wenn die Aktion A die Aktion B verursacht.

14.1.2.3 Formale Sprachen: Die Knoten sind die Wörter über einem Alphabet. Die Kantenmenge E besteht aus allen Paaren (u, v), für die das Wort v durch einen direkten Ableitungsschritt der Grammatik aus dem Wort u abgeleitet werden kann.

14.2 Ungerichtete Graphen

14.2.1 Definition *Ein **ungerichteter Graph** (undirected graph) G ist ein Paar (V, E), wobei V eine endliche Menge und E eine Menge von zweielementigen Teilmengen von V ist. V heißt die Knotenmenge und E die Kantenmenge. Jede Kante $e \in E$ hat die Gestalt $\{u, v\}$ mit $u, v \in V$ und $u \neq v$; die Knoten u, v heißen die Endpunkte der Kante. Ein Graph (V, E) heißt Teilgraph eines Graphen (V', E'), wenn $V \subset V'$ und $E \subset E'$ gilt.*

Der Unterschied zu gerichteten Graphen besteht darin, daß die Reihenfolge der Endpunkte einer Kante keine Rolle spielt und daß keine Kante von einem Knoten zu demselben Knoten zugelassen ist.
Achtung! Es ist üblich, auch bei ungerichteten Graphen die Kanten mit (u, v) statt mit $\{u, v\}$ zu bezeichnen. Man muß nur daran denken, daß (u, v) und (v, u) dieselbe Kante bezeichnen.

Sprechweisen: Statt $e = \{u, v\} \in E$ sagt man auch, u und v sind adjazent oder u und v sind inzident mit e oder u und v liegen auf e. Für $u \in V$ heißt

$$\text{degree}(u) = \#\{e \in E : u \text{ ist inzident mit } e\}$$

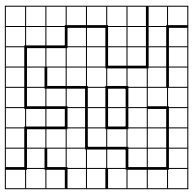
der Grad von u.

14.2.2 Beispiele

14.2.2.1 Bahnnetze oder allgemein Transportnetze mit symmetrischen Transportwegen.

14.2.2.2 Gitter: Die Knotenmenge V sei eine Teilmenge von $\mathbb{Z}^d$. Zwischen zwei Knoten $u, v \in V$ gibt es genau dann eine (ungerichtete) Kante, wenn sich u und v nur in einer Komponente unterscheiden und zwar um den Betrag 1. Die Zahl d wird die Dimension des Gitters genannt.

14.2.2.3 Labyrinthe: Einen Teilgraph eines zweidimensionalen Gitters wollen wir Labyrinth nennen. In Beispielen für Labyrinthe wird die Knotenmenge meist ein Quadrat $\{0, \ldots, N-1\} \times \{0, \ldots, N-1\} \subset \mathbb{Z}^2$ sein. Die Knoten stelle man sich als Karos eines karierten Stück Papiers vor. Kanten können nur zwischen Karos existieren, die eine Seite gemeinsam haben. Statt Kanten als Pfeile darzustellen, werden Trennwände eingezeichnet, wenn zwischen benachbarten Karos keine Kante existiert.



14.2.3 Definition

a) *Jeder ungerichtete Graph $G = (V, E)$ hat eine **gerichtete Version** $G' = (V', E')$, wobei $V' = V$ und $E' = \{(u, v) \in V^2 : \{u, v\} \in E\}$. Das heißt, jede ungerichtete Kante $\{u, v\}$ von G wird durch die beiden gerichteten Kanten (u, v) und (v, u) ersetzt.*

b) *Ist $G = (V, E)$ ein gerichteter Graph, so ist seine **ungerichtete Version** der ungerichtete Graph $G' = (V', E')$, wobei $V' = V$ und $E' = \{\{u, v\} : (u, v) \in E, u \neq v\}$. Das heißt, man vergißt die Richtungen der Kanten und läßt Schleifen weg.*

14.3 Wege in Graphen

$G = (V, E)$ sei ein Graph. Kanten werden mit (u, v) bezeichnet, auch im ungerichteten Fall. Ein **Weg** (path) der Länge $n \in \mathbb{N} \cup \{0\}$ von $v_0 \in V$ nach $v_n \in V$ ist eine Folge $(v_0, \ldots, v_n)$ in V, so daß $(v_{j-1}, v_j) \in E$ für jedes $j \in \{1, \ldots, n\}$. Der Knoten v_0 wird der **Anfangspunkt** und v_n der **Endpunkt** des Weges genannt. Stimmen Anfangs- und Endpunkt eines Weges überein, so heißt der Weg geschlossen. Sind die Knoten eines Weges $(v_0, \ldots, v_n)$ außer eventuell v_0 und v_n paarweise verschieden, so heißt der Weg **einfach**. Ein einfacher, geschlossener Weg in einem gerichteten Graphen wird ein (gerichteter) **Kreis** genannt. In ungerichteten Graphen werden einfache, geschlossene Wege nur dann Kreise genannt, wenn ihre Länge > 2 ist; sonst würde durch jede Kante ein Kreis erzeugt.

Existiert ein Weg von u nach v, so heißt v von u aus **erreichbar**. Ist in G jeder Knoten von jedem anderen aus erreichbar, so heißt G **zusammenhängend**, falls G ungerichtet ist, und **stark zusammenhängend**, wenn G gerichtet ist. Für $u \in V$ heißt $Z(u) = \{v \in V : v \text{ ist von } u \text{ aus erreichbar }\}$ die **Zusammenhangskomponente** von u. Ist G gerichtet, so heißt $\{v \in V : v \in Z(u) \text{ und } u \in Z(v)\}$ die **starke Zusammenhangskomponente** von u. Im ungerichteten Fall stimmen die Zusammenhangskomponente und die starke Zusammenhangskomponente überein. Die Relation $u \sim v$ sei dadurch definiert, daß u von v und v von u aus erreichbar ist. Dann ist $\sim$ eine Äquivalenzrelation auf V; die Äquivalenzklassen sind gerade die starken Zusammenhangskomponenten.

Wichtige Fragen bei Anwendungen sind beispielsweise, ob ein Graph zusammenhängend ist oder ob er einen Kreis enthält. Solche Eigenschaften werden topologisch genannt. Man benötigt effiziente Algorithmen, wie man beispielsweise alle Knoten finden kann, die von einem vorgegebenen Knoten aus erreichbar sind.

14.4 Suchstrategien in Graphen

14.4.1 Typische Aufgaben

14.4.1.1 Finde alle Knoten eines Graphen, die von einem vorgegebenen Knoten aus erreichbar sind.

14.4.1.2 Entscheide für zwei gegebene Knoten u und v, ob $v \in Z(u)$ ist.

14.4.1.3 Finde zu gegebenen Knoten u und $v \in Z(u)$ einen Weg von u nach v.

Bei vielen Algorithmen läuft man längs der Kanten in einem Graphen umher. Oft will man sich merken, ob man schon einmal bei einen Knoten war. Dazu muß man in dem Knoten irgendeine Markierung anbringen. In theoretischen Algorithmen kümmert man sich nicht darum, wie das realisiert werden kann, sondern spricht nur davon, daß man einen Knoten markiert. In der Praxis hängt es von der Implementierung ab, wie eine Markierung realisiert werden kann; man könnte z.B. jeden Knoten durch einen Struct darstellen, in dem eine Komponente für die Markierung vorgesehen ist, oder die Knoten durchnumerieren und sich die Markierung in einem Array merken. Wir nehmen immer stillschweigend an, daß zu Beginn der Ausführung eines Algorithmus alle Knoten unmarkiert seien.

Zur Lösung obiger Aufgaben werden vor allem zwei Strategien eingesetzt, die man Tiefensuche und Breitensuche nennt. Die Namen werden verständlich, wenn man die Reihenfolge, in der die Knoten durchlaufen werden, betrachtet. Diese unterschiedliche Reihenfolge kommt dadurch zustande, daß die Tiefensuche einen Stapel benutzt, während bei der Breitensuche stattdessen eine Schlange verwendet wird. Die Tiefensuche kann auch rekursiv formuliert werden; dann ist der Stapel nicht explizit sichtbar, sondern verbirgt sich in der Verschachtelung der Funktionsaufrufe. Wir werden die Algorithmen in C-ähnlicher Syntax formulieren. Znächst wenden wir uns der Aufgabe 14.4.1.1 zu.

14.4.2 Algorithmus Rekursive_Tiefensuche

Eingabe: Ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $u \in V$.

Ausgabe: Die Knoten in der Zusammenhangskomponente $Z(u)$ von u.

Rekursive_Tiefensuche(G, u) {
  markiere den Knoten u und gib ihn aus ;
  `for ( ` $(u, v) \in E$ `) {`
    `if(`v ist nicht markiert`)` führe Rekursive_Tiefensuche(G, v) aus;
  }

}

14.4.3 Satz: Obiger Algorithmus terminiert und ist korrekt.

Beweis: Weil bei jedem erneuten Aufruf des Algorithmus ein bisher noch nicht markierter Knoten übergeben und markiert wird und die Knotenmenge endlich ist, muß der Algorithmus nach endlich vielen Schritten stoppen.
Für die Korrektheit ist zu zeigen, daß die ausgegebene Knotenmenge gerade $Z(u)$ ist. Es ist klar, daß jeder von Rekursive_Tiefensuche(G, u) ausgegebene Knoten von u aus erreichbar ist. Sei umgekehrt $x \in Z(u)$. Dann gibt es einen Weg $(u_0, \ldots, u_n)$ von $u_0 = u$ nach $x = u_n$. Sei $j = \max\{i : u_i \text{ wird markiert}\}$. Zu zeigen ist $j = n$.
Annahme: $j < n$. Weil u_j markiert wird, wird Rekursive_Tiefensuche(G, v) für alle $v \in V$ aufgerufen, die adjazent zu u_j sind, also auch für u_{j+1}. Bei Ausführung von Rekursive_Tiefensuche(G, u_{j+1}) wird dann u_{j+1} markiert im Widerspruch zur Definition von j. Also ist die Annahme falsch, und es gilt $j = n$. q.e.d.

Man kann die Rekursion vermeiden und die Tiefensuche iterativ formulieren, allerdings muß man dann explizit einen Stapel einführen und selbst verwalten.

14.4.4 Algorithmus Iterative_Tiefensuche

Eingabe: Ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $u \in V$.

Ausgabe: Die Knoten in der Zusammenhangskomponente $Z(u)$ von u.

```
Iterative_Tiefensuche(G, u) {
    erzeuge einen leeren Stapel stapel über der Knotenmenge V;
    markiere den Knoten u und gib ihn aus;
    push(stapel,u);
    while(is_empty(stapel)==false) {
          v=top(stapel);
          pop(stapel);
          for ( (v,w) ∈ E ) {
                if(w ist nicht markiert) {
                      markiere den Knoten w und gib ihn aus;
                      push(stapel,w);
                }
          }
    }

}
```

Der Beweis, daß dieser Algorithmus korrekt ist, verläuft völlig analog zum Beweis der Korrektheit der rekursiven Tiefensuche.

Ersetzt man den Stapel durch eine Schlange, so erhält man die sogenannte Breitensuche.

14.4.5 Algorithmus Breitensuche

Eingabe: Ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $u \in V$.

Ausgabe: Die Knoten in der Zusammenhangskomponente $Z(u)$ von u.

```
Breitensuche(G, u) {
    erzeuge eine leere Schlange schlange über der Knotenmenge V;
    markiere den Knoten u und gib ihn aus;
    enqueue(schlange,u);
    while(is_empty(schlange)==false) {
          v=front(schlange);
          dequeue(schlange);
          for ( (v,w) ∈ E ) {
                if(w ist nicht markiert) {
                      markiere den Knoten w und gib ihn aus;
                      enqueue(schlange,w);
                }
          }
    }

}
```

14.4.6 Satz: Obiger Algorithmus terminiert und ist korrekt.

Beweis: Weil bei jedem Lauf durch die `for`-Schleife ein bisher noch nicht markierter Knoten markiert wird und die Knotenmenge endlich ist, muß der Algorithmus nach endlich vielen Durchläufen die Schleife verlassen und stoppen.
Für die Korrektheit ist zu zeigen, daß die ausgegebene Knotenmenge gerade $Z(u)$ ist. Für $j \in \mathbb{N} \cup \{0\}$ sei V_j die Menge aller $v \in V$, die man von u aus durch einen Weg $(u = u_0, \ldots, u_n = v)$ mit $n \leq j$ erreichen kann. Dann ist $\bigcup_{j=0}^{\infty} V_j$ die Menge aller von u aus erreichbaren Knoten. Es ist also zu zeigen, daß der Algorithmus für jedes j alle Knoten in V_j markiert und ausgibt. Wir zeigen dies durch vollständige Induktion nach j.
Induktionsanfang $j = 0$: Es gilt $V_0 = \{u\}$ und u wird markiert und ausgegeben.
Induktionsschluß $j \mapsto j + 1$: Wenn $V_{j+1} \neq V_j$, so gibt es einen Knoten $v \in V_{j+1} \setminus V_j$ und einen Weg $(v_0, \ldots, v_{j+1})$ mit $v_0 = u$ und $v_{j+1} = v$. Folglich ist $v_j \in V_j$, und nach der Induktionsvoraussetzung wird v_j markiert. Nachdem der Algorithmus v_j markiert hat, gibt er v_j in die Schlange. Da der Algorithmus nicht stoppt, bevor die Schlange leer ist, wird v_j irgendwann aus der Schlange genommen. Die anschließende `for`-Schleife erstreckt sich auch über die Kante (v_j, v_{j+1}). Falls v_{j+1} noch nicht markiert ist, wird v_{j+1} markiert und ausgegeben. q.e.d.

14.4.7 Bemerkung Die obigen Algorithmen für die Tiefensuche und die Breitensuche sind nicht völlig determiniert, weil in der `for`-Schleife nicht festgelegt ist, in welcher Reihenfolge die Kanten überprüft werden. Je nachdem wie man diese Reihenfolge wählt, ändert sich auch die Reihenfolge, in welcher der Algorithmus die Knoten von $Z(u)$ ausgibt. Dies kann man sehr schön an folgendem Beispiel nachvollziehen.

14.4.8 Beispiel: Suche im Labyrinth

In nebenstehenden Labyrinth soll die Zusammenhangskomponente des Karos sp in der Mitte bestimmt werden. Von jedem Karo gehen höchstens vier Kanten aus, und zwar in die Richtungen `rauf, links, runter, rechts`. In dieser Reihenfolge sollen auch die Kanten in den `for`-Schleifen der Algorithmen überprüft werden. Man überlege sich, daß die rekursive Tiefensuche bzw. die Breitensuche die Karos in den folgenden Reihenfolgen ausgibt.

a	b	c	d	e
f	g	h	i	j
k	l	sp	m	n
o	p	q	r	s
t	u	v	w	x

Tiefensuche: `sp h c b a d e l k p u t v q w x s n m`

Breitensuche: `sp h l q m c k p v n b d u w s a e t x`

Man überlege, wie sich die Reihenfolgen ändern, wenn die Kanten in der Reihenfolge `links, rechts, rauf, runter` durchlaufen werden. Vergleichen Sie die Reihenfolgen bei der rekursiven Tiefensuche und der iterativen Tiefensuche gemäß 14.4.4.

Bei Labyrinthen hat man oft einen Startpunkt oder Eingang `sp` und einen Zielpunkt oder Ausgang `zp` gegebenen und soll entscheiden, ob es einen Weg von `sp` nach `zp` gibt, und falls ja, einen solchen Weg angeben. Es handelt sich dann also um die Aufgaben 14.4.1.2 und 14.4.1.3. Beide lassen sich durch Modifikation obiger Suchalgorithmen lösen.

Ob `zp` von `sp` aus erreichbar ist, läßt sich leicht feststellen: man muß nur überprüfen, ob der Suchalgorithmus den Knoten `zp` ausgibt. Sobald er dies tut, kann man ihn abbrechen.

14.4.8.1 Algorithmus Rekursive_Labyrinthsuche

Eingabe: Ein Labyrinth $G = (V, E)$, ein Startknoten `sp` $\in V$ und ein Zielknoten `zp` $\in V$.

Ausgabe: `erreichbar`, wenn der Zielknoten erreichbar ist; sonst nichts.

Rufe Rekursive_Labyrinthsuche$(G,$ `sp`$)$ auf.

Rekursive_Labyrinthsuche(G, u) {
    markiere den Knoten u;
    `if(` u`==zp )` gib `erreichbar` aus und stoppe;
    `for (` $(u, v) \in E$ `) {`
        `if(`v ist nicht markiert`)` führe Rekursive_Labyrinthsuche(G, v) aus;
    }
}

Um einen Weg von `sp` nach `zp` explizit anzugeben, erweitern wir die Tiefensuche ein wenig, indem wir nicht nur die Knoten ausgeben, die wir beim Vorwärtsgehen (zu noch nicht markierten Knoten) besuchen, sondern auch die Knoten, die wir beim Zurückgehen aus Sackgassen besuchen, bis wir bei `zp` angelangt sind. Diese Folge ist dann ein Weg von `sp` nach `zp`. Allerdings enthält er überflüssige Teile, die dadurch entstehen, daß man in eine Sackgasse hinein- und anschließend wieder heraugeht. Diese Teile kann man leicht entfernen, wenn man die Knoten, die man beim Vorwärtsgehen besucht, auf einen Stapel legt und beim Zurückgehen wieder vom Stapel nimmt. Wenn der Algorithmus auf den Zielknoten stößt, enthält der Stapel einen Weg von `zp` nach `sp`. Sein Inhalt muß also nur ausgelesen und invertiert werden (was mit einem weiteren Stapel erfolgen kann).

Schon in der griechischen Mythologie wurde von Ariadne eine analoge Methode ersonnen, um Theseus zu ermöglichen, in das Labyrinth von Knossos einzudringen und den Minotaurus zu töten, ein Ungeheuer, dem König Minos, der Vater von Ariadne, jährlich sieben Jünglinge und sieben Jungfrauen aus Athen zum Fraß vorwarf. Damit Theseus wieder aus dem Labyrinth des Minotaurus herausfindet, gab Ariadne ihm ein Garnknäuel, das dieser beim Vorwärtsgehen abrollte und beim Zurückgehen wieder aufrollte. Daher spricht man von dem Ariadnefaden. (Trotzdem wurde sie später von Theseus auf Naxos zurückgelassen.)

14.4.8.2 Algorithmus zur Suche eines Lösungsweges

Eingabe: Ein Labyrinth $G = (V, E)$, ein Startknoten `sp` $\in V$ und ein Zielknoten `zp`.

Ausgabe: Ein Weg von `sp` nach `zp`, falls existent.

Erzeuge einen leeren Stapel `stapel` über der Knotenmenge V;
rufe Wegsuche$(G,$ `sp`$)$ auf.

```
Wegsuche(G, u) {
    markiere den Knoten u;
    push(stapel,u);
    if(u==zp ) {
          gib stapel in umgekehrter Reihenfolge aus;
          stoppe;
    }
    for ( (u, v) ∈ E ) {
          if( v ist nicht markiert) { führe Wegsuche(G, v) aus; }
    }
    pop(stapel);
}
```

14.5 Allgemeine Form der Tiefen- und Breitensuche

Bei dem Beispiel 14.4.8 der Labyrinthsuche wurden die Algorithmen zur Tiefensuche dadurch abgeändert, daß an einigen Stellen zusätzliche Abfragen oder Ausgaben eingefügt wurden. Allgemein kann man die Strategien der Tiefensuche oder Breitesuche einsetzen, um die Knoten eines zusammenhängenden Graphen in einer gewissen Reihenfolge zu durchlaufen und dabei irgendwelche Operationen sei es an den besuchten Knoten oder den durchlaufenen Kanten vorzunehmen, eventuell auch durch eine Bedingung die Suche vorzeitig abzubrechen.

Bei der rekursiven Tiefensuche sind vier Stellen prädestiniert, um solche Zusatzprozeduren einzufügen: wenn man eine Kante beim Vorwärtsgehen durchläuft, wenn man einen neuen Knoten besucht, wenn man eine Kante beim Zurückgehen wieder rückwärts durchläuft und wenn man beim Zurücklaufen eine Knoten zum zweiten Mal besucht. Wir wollen die eingefügten Prozeduren entsprechend Kantenvorarbeit, Knotenvorarbeit, Kantennacharbeit und Knotennacharbeit nennen. Damit erhält die rekursive Tiefensuche folgende Form.

14.5.1 Algorithmusstrategie Rekursive Tiefensuche

Eingabe: Ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $sp \in V$.

Ausgabe: erfolgt anwendungsabhängig in den Prozeduren `Knotenvorarbeit`, `Kantenvorarbeit`, `Knotennacharbeit` und `Kantennacharbeit`

Rufe Rekursive_Tiefensuche(G, sp) auf.

```
Rekursive_Tiefensuche(G, u) {
    markiere den Knoten u;
    führe Knotenvorarbeit(u) aus;
    for ( (u,v) ∈ E ) {
        if(v ist nicht markiert) {
            führe Kantenvorarbeit(u,v) aus;
            führe Rekursive_Tiefensuche(G,v) aus;
            führe Kantennacharbeit(u,v) aus;
        }
    }
    führe Knotennacharbeit(u) aus;
}
```

Bei der iterativen Tiefensuche oder Breitensuche durchläuft man Kanten nur beim Vorwärtsgehen, es gibt kein Zurückgehen. Dementsprechend werden nur die zwei Prozeduren `Knotenarbeit` und `Kantenarbeit` eingefügt.

14.5.2 Algorithmusstrategie Iterative Tiefensuche

Eingabe: Ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $u \in V$.

Ausgabe: erfolgt anwendungsabhängig in den Prozeduren `Kantenarbeit` und `Knotenarbeit`

```
Iterative_Tiefensuche(G, u) {
    erzeuge einen leeren Stapel stapel über der Knotenmenge V;
    markiere den Knoten u;
    push(stapel,u);
    while(is_empty(stapel)==false) {
        v=top(stapel);
        pop(stapel);
        führe Knotenarbeit(v) aus;
        for ( (v,w) ∈ E ) {
            if(w ist nicht markiert) {
                markiere den Knoten w;
                führe Kantenarbeit(v,w) aus;
                push(stapel,w);
            }
        }
    }

}
```

14.5.3 Algorithmusstrategie Breitensuche

Eingabe: Ein gerichteter oder ungerichteter Graph $G = (V, E)$ und ein Knoten $u \in V$.

Ausgabe: erfolgt anwendungsabhängig in den Prozeduren `Kantenarbeit` und `Knotenarbeit`

```
Breitensuche(G, u) {
   erzeuge eine leere Schlange schlange über der Knotenmenge V;
   markiere den Knoten u;
   enqueue(schlange,u);
   while(is_empty(schlange)==false) {
         v=front(schlange);
         dequeue(schlange);
         führe Knotenarbeit(v) aus;
         for ( (v,w) ∈ E ) {
               if(w ist nicht markiert) {
                     markiere den Knoten w;
                     führe Kantenarbeit(v,w) aus;
                     enqueue(schlange,w);
               }
         }
   }

}
```

14.5.4 Satz

14.5.4.1 Falls durch die Prozeduren `Knotenvorarbeit`, `Knotennacharbeit`, `Kantenvorarbeit`, `Kantennacharbeit` kein vorzeitiges Ende bewirkt wird, markiert die rekursive Tiefensuche bei Eingabe von (G, u) alle Knoten von G, die von u aus erreichbar sind, und führt für alle diese Knoten v sowohl die `Knotenarbeit(`v`)` als auch die `Kantennacharbeit(`v, w`)` für alle Kanten (v, w) in G durch.

14.5.4.2 Falls durch die Prozeduren `Kantenarbeit` und `Knotenarbeit` kein vorzeitiges Ende bewirkt wird, markiert die iterative Tiefensuche bei Eingabe von (G, u) alle Knoten von G, die von u aus erreichbar sind, und führt für alle diese Knoten v die `Knotenarbeit(`v`)` durch.

14.5.4.3 Falls durch die Prozeduren `Kantenarbeit` und `Knotenarbeit` kein vorzeitiges Ende bewirkt wird, markiert die Breitensuche bei Eingabe von (G, u) alle Knoten von G, die von u aus erreichbar sind, und führt für alle diese Knoten v die `Knotenarbeit(`v`)` durch.
Beweis: wie in 14.4.3 und 14.4.6.

14.6 Darstellung von Graphen

Die Darstellung von Graphen im Rechner kann je nach Anwendung sehr unterschiedlich aussehen. Ist man nur an topologischen Eigenschaften eines Graphen interessiert, so wird man die Knoten einfach durchnumerieren und für die Kanten eine Darstellung suchen, die möglichst wenig Platz braucht oder mit der die eingesetzten Algorithmen möglichst effizient arbeiten können. In anderen Fällen will man in den Knoten oder

auch den Kanten zusätzliche Informationen speichern. Dann wird man eventuell für jeden Knoten einen Struct verwenden, wie wir es bereits bei Listen getan haben. Die Kanten könnte man dann durch Zeiger repräsentieren. Bei speziellen Graphen wie z.B. Bäumen wird dies auch oft gemacht. Bei allgemeinen Graphen ist diese Darstellung oft ungeschickt, z.B. wenn der Ausgangsgrad der Knoten stark variiert oder wenn es nur relativ wenig Kanten gibt; denn dann wird durch die Zeiger unnötig viel Platz benötigt.

14.6.1 Beispiel: Labyrinth

Bei einem Labyrinth liegt die spezielle Situation vor, daß von jedem Knoten höchtens 4 Kanten ausgehen können und zwar nur zu unmittelbaren Gitternachbarn. Es genügt also, sich für jede der vier möglichen Richtungen zu einem Nachbarn zu merken, ob zu ihm eine Kante führt. Diese Information kann man in einem Array von `boolean` der Länge 4 oder einfach mit 4 Bit codieren. Ist das Labyrinth rechteckig mit der Knotenmenge $[0, N_x - 1] \times [0, N_y - 1] \subset \mathbb{Z}^2$, so kann man diese Information in einer Matrix abspeichern, deren Einträge 4-Bit-Zahlen sind (oder Bytes, bei denen die oberen vier Bit keine Bedeutung haben). Zur Implementierung von Matrizen gibt es in den meisten Programmiersprachen mehrdimensionale Arrays oder ähnliche Strukturen. In C gibt es nicht wirklich mehrdimensionale Arrays; man kann aber Arrays bilden, deren Elementtyp wieder ein Arraytyp ist.

14.6.2 Adjazenzmatrizen

Die Kantenmenge E kann als Teilmenge von $V \times V$ durch die charakteristische Funktion $\chi_E \colon V \times V \to \{0, 1\}$ beschrieben werden. Weil V endlich ist, kann man χ_E als Matrix $A = (a_{uv})_{u,v \in V}$ schreiben, indem man $a_{uv} = \chi_E(u, v) = 1$ setzt, wenn $(u, v) \in E$, und 0 sonst.

Da V endlich ist, kann man eine bijektive Abbildung $\{0, \ldots, |V|-1\} \to V$ wählen und dementsprechend die Adjazenzmatrix in einem zweidimensionalen Array `a[|V|][|V|]` speichern; dabei bezeichne $|V|$ die Anzahl der Elemente von V und $|E|$ die von E.

Diese Darstellung eines Graphen $G = (V, E)$ ist sinnvoll, wenn der Graph **dicht** (dense) ist d.h. wenn $|E|$ in der Größenordnung von $|V|^2$ liegt, wenn also G viele Kanten hat. Ist dagegen G **dünn** (sparse) d.h. ist $|E|$ viel kleiner als $|V|^2$, so verschwendet die Adjazenzmatrix unnötig viel Platz. Auch die Suche nach den Kanten, die von einem Knoten ausgehen, kann unnötig lange dauern. Man verwendet dann besser Adjazenzlisten.

14.6.3 Adjazenzlisten

Die Idee besteht darin, jede Zeile der Adjazenzmatrix als Liste darzustellen. Für jeden Knoten $u \in V$ legt man eine Liste $L(u)$ an, die genau die Knoten $v \in V$ enthält, zu denen von u aus eine Kante führt. Die Reihenfolge ist beliebig. Dann numeriert man V mit einer bijektiven Abbildung $\{0, \ldots, |V| - 1\} \to V$ durch und schreibt dementsprechend in ein Array der Länge $|V|$ Zeiger, die auf die jeweilige Liste $L(u), u \in V$, weisen. Diese Darstellung ist insbesondere für dünne Graphen geeignet.

Kapitel 15

Bäume

15.1 Freie Bäume

15.1.1 Definition

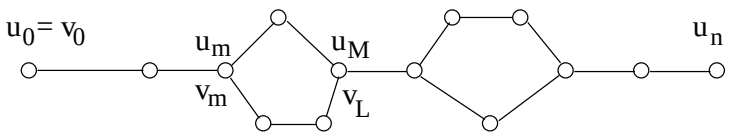
a) *Ein Graph heißt* ***kreisfrei*** *oder* ***azyklisch****, wenn in ihm kein Kreis existiert.*

b) *Ein kreisfreier, ungerichteter Graph heißt ein* ***Wald****.*

c) *Ein zusammenhängender, kreisfreier, ungerichteter Graph heißt ein* ***freier Baum****.*

15.1.2 Lemma Ein ungerichteter Graph $G = (V, E)$ ist genau dann ein freier Baum, wenn je zwei Knoten in G durch genau einen einfachen Weg miteinander verbunden sind.

Beweis $\Leftarrow$: Es ist klar, daß G zusammenhängend ist; es ist nur noch zu zeigen, daß G kreisfrei ist. Angenommen, es gäbe einen Kreis $(v_0, \ldots, v_n)$ in G. Dann ist $n \geq 3$, und (v_0, v_1) und $(v_n, v_{n-1}, \ldots, v_1)$ sind zwei unterschiedliche einfache Wege von v_0 nach v_1 im Widerspruch zur Voraussetzung.
$\Rightarrow$: Angenommen, es gäbe zwei unterschiedliche einfache Wege $(u_0, \ldots, u_n)$ und $(v_0, \ldots, v_l)$ von $v_0 = u_0$ nach $v_l = u_n$. Seien $m = \max\{j : \text{für } 0 \leq i \leq j \text{ gilt } u_i = v_i\}$ und $M = \min\{j : m < j \text{ und es gibt } r > m \text{ mit } u_j = v_r\}$. Es gibt ein $L > m$ mit $v_L = u_M$. Weil $L > m+1$ oder $M > m+1$ ist, hat $(u_m, u_{m+1}, \ldots, u_M, v_{L-1}, \ldots, v_m)$ eine Länge ≥ 3 und ist somit ein Kreis. Das ist ein Widerspruch zur Voraussetzung.



15.2 Wurzelbäume

15.2.1 Definition *Ein* ***Wurzelbaum*** *ist ein freier Baum B, in dem ein Knoten w vor den anderen ausgezeichnet ist, also ein Paar (B, w). Der ausgezeichnete Knoten*

*w heißt die **Wurzel** des Baumes.*

(B, w) sei ein Wurzelbaum. Nach 15.1.2 ist jeder Knoten v durch einen eindeutigen einfachen Weg $(v_0, v_1, \ldots, v_{n-1}, v)$ mit der Wurzel $w = v_0$ verbunden. Jeder Knoten v_i heißt ein **Vorgänger** (ancestor) von v, und v heißt **Nachfolger** (descendant) jedes Knoten v_i. Der Knoten v_{n-1} heißt der **Elternknoten** oder **Vater** (Mutter?) (parent) von v, und v heißt **Kind** (child) oder **Sohn** (Tochter?) von v_{n-1}. Die Wurzel ist der einzige Knoten, der keinen Elternknoten hat. Ein Knoten ohne Kind heißt ein **Blatt** (leaf) oder **äußerer Knoten**; die anderen Knoten heißen **innere Knoten**. Knoten mit demselben Elternknoten heißen **Geschwister** oder **Brüder** (Schwestern?) (sibling).

Der **Teilbaum mit Wurzel** v ist der Baum, der aus v und allen Nachfolgern von v und den Kanten zwischen ihnen besteht und die Wurzel v hat.

Die **Tiefe** n eines Knotens v ist die Länge des (eindeutig bestimmten) Weges von der Wurzel w zu v. Die **Höhe** des Baumes ist das Maximum T der Tiefen seiner Knoten. **Achtung:** Manchmal wird in der Literatur auch $T + 1$ als Höhe bezeichnet!

Ein Wurzelbaum wird auf natürliche Weise ein gerichteter Graph, indem man jede Kante $\{u, v\}$ durch die gerichtete Kante (u, v) ersetzt, wenn u der Elternknoten von v ist.

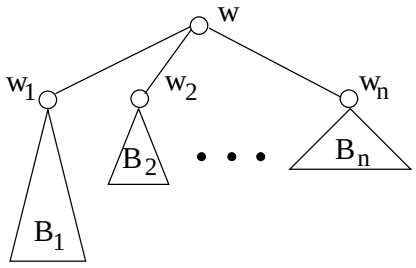
Die Anzahl der Kinder eines Knotens heißt der **Grad des Knotens**. Vorsicht! Der so definierte Grad stimmt nicht mit dem Grad des Knotens in dem freien Baum B überein; der ist um 1 größer.

Alternativ zu obiger Definition kann man Wurzelbäume auch induktiv definieren.

15.2.2 Induktive Definition von Wurzelbäumen

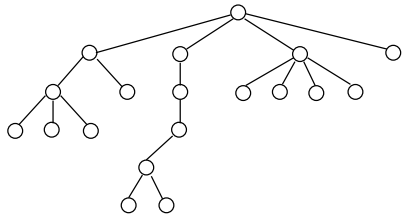
1) Ist B ein freier Baum, der nur einen Knoten w hat, so ist (B, w) ein Wurzelbaum.

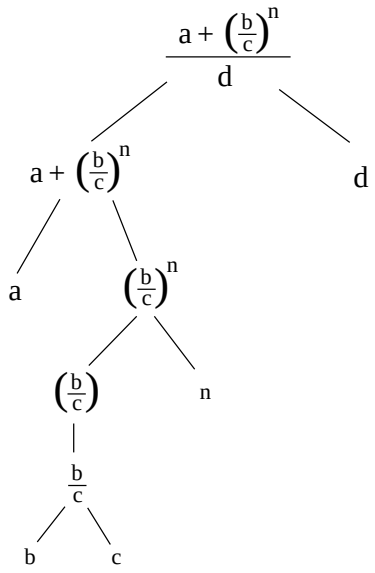
2) Sind $(B_1, w_1), \ldots, (B_n, w_n)$ Wurzelbäume mit paarweise disjunkten Knotenmengen $V_1, \ldots, V_n$ und Kantenmengen $E_1, \ldots, E_n$, so ist (B, w) ein Wurzelbaum, wobei $B = (V, E)$ mit $V = V_1 \cup \ldots \cup V_n \cup \{w\}, w \notin V_1 \cup \ldots \cup V_n, E = E_1 \cup \ldots \cup E_n \cup \{(w, w_1), \ldots, (w; w_n)\}$. Das heißt: man verbindet einen neuen Knoten w durch neue Kanten mit den Wurzeln der Bäumen $B_1, \ldots, B_n$.

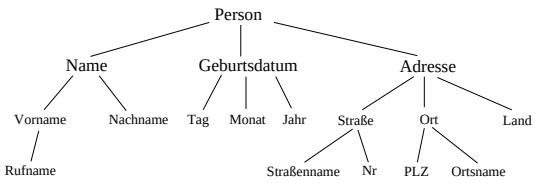


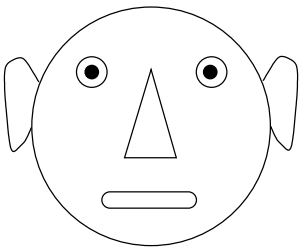
15.2.3 Beispiele

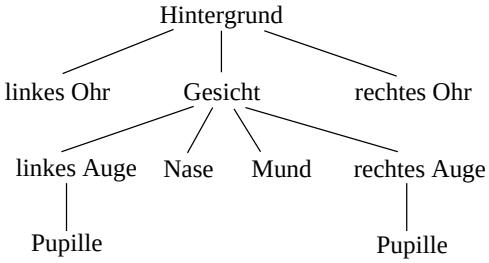
Üblicherweise zeichnet man die Wurzel oben und die Knoten mit zunehmender Tiefe darunter auf horizontalen Etagen.
Wurzelbäume sind gut geeignet zur Darstellung hierarchisch strukturierter Daten.











Söhne verdecken partiell ihren Vater.

15.3 Spannbäume

15.3.1 Definition *G sei ein ungerichteter Graph. Ein* ***Spannbaum*** *von G ist ein Teilgraph von G, der ein Baum ist und alle Knoten von G enthält.*

15.3.2 Beispiele

15.3.2.1 G sei ein Kommunikationsnetz, in dem jeder Knoten Nachrichten empfangen, versenden und weiterleiten kann. Um zu vermeiden, daß Nachrichten auf verschiedenen Wegen mehrfach übertragen werden oder gar kreisen, kann man einen Spannbaum von G bestimmen und dann die Nachrichten nur längs der Wege in diesem Spannbaum befördern. Nach 15.1.2 gibt es in einem Baum zwischen zwei Knoten genau einen einfachen Weg, so daß die erwähnten Probleme vermieden werden.

15.3.2.2 Ein kreisfreies Labyrinth aus $N \times N$ Karos, bei dem alle Positionen vom Startpunkt aus erreichbar sind, ist ein Spannbaum des $N \times N$-Gitters.

15.3.3 Konstruktion von Spannbäumen mit Tiefensuche

$G = (V, E)$ sei eine zusammenhängender, ungerichteter Graph. Ein **DFS-Spannbaum** wird mit Hilfe der Tiefensuche-Strategie 14.5.1 konstruiert, wenn man folgendes definiert:

- Zu Beginn ist die Kantenmenge E_{DFS} des zu konstruierenden Baumes leer.
- Es gibt keine Knotenarbeit und keine Kantennacharbeit.
- Die `Kantenvorarbeit(`u, v`)` besteht darin, die Kante (u, v) zu E_{DFS} hinzuzunehmen.

Ist $w \in V$ und wird dieser Algorithmus mit der Eingabe (G, w) aufgerufen, so stoppt er nach endlich vielen Schritten, und (V, E_{DFS}) ist dann ein Spannbaum von G.

Beweis: Nach 14.5.4 wird jeder Knoten $v \in V$ markiert. Dies geht nur dadurch, daß Rekursive_Tiefensuche(G, v) für jeden Knoten v ausgeführt wird; folglich wird auch die direkt davor stehende `Kantenvorarbeit(`u, v`)` ausgeführt. Infolgedessen ist jeder Knoten $v \neq w$ durch einen Weg, dessen Kanten in E_{DFS} liegen, mit w verbunden. Kreise können dabei nicht entstehen, weil die `Kantenvorarbeit(`u, v`)` nicht ausgeführt wird, wenn u und v bereits beide markiert sind. q.e.d.

Bemerkung Der konstruierte DFS-Spannbaum hängt von der Wahl des Startknotens w und der Reihenfolge, in der die Kanten in der `for`-Schleife durchlaufen werden, ab.

15.3.4 Konstruktion von Spannbäumen mit Breitensuche

$G = (V, E)$ sei eine zusammenhängender, ungerichteter Graph. Ein **BFS-Spannbaum** wird mit Hilfe der Breitensuche-Strategie 14.5.3 konstruiert, wenn man folgendes definiert:

- Zu Beginn ist die Kantenmenge E_{BFS} des zu konstruierenden Baumes leer.
- Es gibt keine Knotenarbeit.
- Die Prozedur `Kantenarbeit(`u, v`)` besteht darin, die Kante (u, v) zu E_{BFS} hinzuzunehmen.

Ist $w \in V$ und wird dieser Algorithmus mit der Eingabe (G, w) aufgerufen, so stoppt er nach endlich vielen Schritten, und (V, E_{BFS}) ist dann ein Spannbaum von G.

Beweis: Sehr ähnlich zu obigem.

15.3.5 Bemerkungen

1. Wählt man den Startknoten w als Wurzel des konstruierten Spannbaumes, so ist stets u der Elternknoten von v, wenn `Kantenvorarbeit(`u, v`)` bzw. `Kantenarbeit(`u, v`)` ausgeführt wird.

2. Es gibt noch andere Algorithmen, um Spannbäume mit besonderen Eigenschaften zu konstruieren. Ist z.B. jeder Kante von G eine nicht nichtnegative Zahl (Länge oder Kosten) zugeordnet, so sucht man oft **minimale Spannbäume**: das sind Spannbäume, bei denen die Summe der Gewichte ihrer Kanten möglichst klein ist.

15.3.6 Beispiel: Rechtschreibprüfung
A sei ein Alphabet mit k Elementen. Die Wörter $V = \cup_{n=0}^{N} A^n$ mit maximaler Länge N wählt man als Knotenmenge. Und die Kantemenge definiert man als $E = \{(u, v) \in V \times V$: es gibt ein $a \in A$ mit $v = ua\}$. Jede Kante entspricht also dem Anhängen eines Buchstabens. Mit diesem Buchstaben markiert man die Kante. (V, E) ist ein Wurzelbaum, dessen Wurzel das leere Wort ist und dessen innere Knoten genau k Kinder haben. Ist nun eine Sprache $L \subset A^*$ über A, deren Wörter höchstens die Länge N haben, so markiert man alle Knoten von V, die in L liegen, als korrekt. Während nun ein Wort buchstabenweise eingetippt wird, kann man beginnend bei der Wurzel den Kanten folgen, die mit dem jeweils eingetippten Buchstabern markiert sind, und überprüfen, ob man zu einem Knoten gelangt, der als korrektes Wort markiert ist. Eine Weiterentwicklung sind die *tries* genannten Bäume.

15.4 Binäre Bäume

15.4.1 Definition *Ein **binärer Baum** oder **Binärbaum** ist ein Wurzelbaum mit folgenden Eigenschaften:*

1. *Jeder Knoten hat höchstens zwei Kinder.*
2. *Jeder Knoten außer der Wurzel ist mit* links *oder* rechts *markiert.*
3. *Die Kinder eines Knotens sind stets unterschiedlich markiert.*

*Ein binärer Baum heißt **voll** (full), wenn keiner seiner Knoten nur ein Kind hat, und er heißt **vollständig** (complete), wenn er voll ist und alle Blätter die gleiche Tiefe haben.*

Bemerkungen: Hat ein Knoten nur ein Kind, so kann es mit links oder rechts markiert sein. Bei einer grafischen Darstellung zeichnet man natürlich das mit links markierte Kind links von seinem Geschwisterknoten. Hat ein Knoten keinen Geschwisterknoten, so macht man durch die Richtung der Kante, die von seinem Elternknoten zu ihm führt, deutlich, ob er mit links oder rechts markiert ist.

Ist v ein Knoten in dem Wurzelbaum (B, w), so bezeichne (B_v, v) den Teilbaum mit Wurzel v. Hat die Wurzel w einen linken (bzw. rechten) Sohn v, so heißt (B_v, v) der **linke Teilbaum** (bzw. **rechte Teilbaum**) von (B, w). Hat w keinen linken bzw. rechten Sohn, so wird als linker bzw. rechter Teilbaum der leere Baum definiert.

15.4.2 Beispiel: Codebäume
Jedem binären Präfixcode kann man einen Binärbaum zuordnen, mit dem die Decodierung der Codewörter vereinfacht wird.
Sei $C \subset \{0, 1\}^*$ ein Präfixcode. N sei die maximale Codewortlänge. Wähle einen vollständigen Binärbaum B der Höhe N, in dem jedes Blatt die Tiefe N hat. Jede Kante zu einem linken Kind sei mit 0, jede Kante zu einem rechten Kind mit 1 markiert. Jeder Knoten v sei mit demjenigen Wort $\varphi(v) \in \{0, 1\}^*$ markiert, welches dadurch entsteht, daß man die Kantenmarkierungen auf dem (eindeutig bestimmten) Weg von der Wurzel w nach v hintereinander schreibt. Es sei $B_C = (V, E)$ der kleinste

Teilbaum von B, für den $C \subset \varphi(V)$ gilt. Den Codewörtern in C entsprechen dann genau die Blätter von B_C.

Um festzustellen, ob ein Wort $\alpha_1 \dots \alpha_k \in \{0,1\}^*$ zu C gehört, braucht man nur in B_C von der Wurzel w aus entlang den mit $\alpha_1, \dots, \alpha_k$ markierten Kanten zu laufen. Kommt man nach dem k-ten Schritt zu einem Blatt, so ist das Wort in C, sonst nicht. Dadurch wird die Decodierung einfacher, als wenn man mit allen Wörtern in C vergleicht. Eine analoge Überlegung kann man natürlich auch für Präfixcodierungen über Alphabeten mit mehr als 2 Zeichen anstellen.

Ganz allgemein sind binäre Bäume geeignete Datenstrukturen, um Daten in einer Hierarchie abzuspeichern, die durch sukzessive Entscheidungen zwischen zwei Alternativen gegeben ist.

15.4.3 Beispiel: Heaps, Prioritätswarteschlangen

15.4.3.1 Definition *K sei eine geordnete Menge.*
*Eine **Max-Heap** (bzw. **Min-Heap**) über K besteht aus einem Binärbaum $B = (V, E)$ und einer Abbildung $\sigma\colon V \to K$, so daß stets $\sigma(v) \leq \sigma(u)$ (bzw. $\sigma(v) \geq \sigma(u)$) gilt, wenn v ein Kind von u ist.*

K heißt Schlüsselmenge, $\sigma(v)$ der Schlüssel (key) von v. Der Schlüssel in der Wurzel eines Max-Heaps ist also immer maximal. Für Heaps betrachtet man folgende Operationen:

- Füge zu einem Heap einen neuen Knoten x mit einem Schlüssel $\sigma(x)$ so hinzu, daß wieder ein Heap entsteht.
- Entferne die Wurzel des Heaps und baue den verbleibenden Graphen so um, daß wieder ein Heap entsteht.

Für beide Operationen gibt es effiziente Algorithmen, für die die Anzahl der nötigen Schlüsselvergleiche in O(Höhe des Heaps) liegen. Deshalb ist ein Heap eine Datenstruktur, die für schnelle Sortierverfahren (Heapsort) und für die Implementierung von Prioritätswarteschlangen geeignet ist.

15.4.3.2 Der ADT Prioritätswarteschlange

P sei eine geordnete Menge (die Menge der Prioritäten, z.B. $P = \mathbb{Z}$). W sei eine Menge (der Wertebereich der Daten).

SORTEN:	S
METHODEN:	enqueue: $S \times (W \times P) \to S$, dequeue: $S \to S$, max: $S \to W$
	is_empty: $S \to \{\texttt{true},\texttt{false}\}$, make_empty: $S \to S$

Die Axiome beschreiben wir nur informell. Die Funktion enqueue speichert ein weiteres Datum aus W zusammen mit einer Priorität in der Schlange. Die Funktion max gibt den Wert eines Datums zurück, das in der Schlange gespeichert ist und dessen Priorität größer oder gleich der aller anderen Daten in der Schlange ist. Die Funktion dequeue entfernt dieses Datum aus der Schlange.

Stapel und Schlangen sind Spezialfälle; man braucht nur die Prioritäten bei der Eingabe der Daten streng monoton wachsen bzw. streng monoton fallen zu lassen. Bei der Implementierung einer allgemeinen Prioritätswarteschlange ist es aber vorteilhaft, den

Inhalt der Prioritätswarteschlange nicht durch eine Liste, sondern durch einen Heap darzustellen.

15.4.4 Darstellung von Binärbäumen

Explizite Darstellung: Die Knoten werden als Objekte eines Struct-Typs bzw. einer Klasse dargestellt, und die Kanten als zwei Komponenten des Knotenobjekts, die auf auf Knotenobjekte (die Söhne) weisen.

Implizite Darstellung: Die Knoten werden als Elemente eines Arrays `a[]` dargestellt; die Kanten werden implizit durch die Reihenfolge definiert:

- `a[1]` enthält die Wurzel.
- das linke Kind von `a[i]` ist in `a[2*i]`, das rechte in `a[2*i+1]`.

15.4.5 Durchlaufen eines Binärbaums

Binärbäume sind zusammenhängende Graphen und können folglich mit Tiefen- oder Breitensuche durchlaufen werden, wobei man bei der Wurzel beginnt. Weil ein Baum B keine Kreise enthält, kann man sich die Markierung der Knoten ersparen. Außerdem gehen von jedem Knoten v höchstens zwei Kanten zu Nachfolgern aus, so daß man die `for`-Schleife in zwei Funktionsaufrufe auflösen kann, welche den linken bzw. rechten Teilbaum von B_v durchsuchen. Die Tiefensuche-Strategie in 14.5.1 erhält dann folgende Gestalt.

Rekursive Tiefensuche

Eingabe: Ein Binärbaum $B = (V, E)$ mit Wurzel w.

Ausgabe: erfolgt anwendungsabhängig in den Prozeduren `Knotenarbeit_1`, `Knotenarbeit_2`, `Knotenarbeit_3`, `Kantenvorarbeit` und `Kantennacharbeit`

Rufe Rekursive_Tiefensuche(B, w) auf.

```
Rekursive_Tiefensuche(B, v) {
    führe Knotenarbeit_1(v) aus;
    if(v hat einen linken Sohn v_l) {
            führe Kantenvorarbeit(v, v_l)   aus;
            führe Rekursive_Tiefensuche(B_{v_l}, v_l) aus;
            führe Kantennacharbeit(v, v_l)   aus;
    }
    führe Knotenarbeit_2(v) aus;
    if(w hat einen rechten Sohn v_r) {
            führe Kantenvorarbeit(v, v_r)   aus;
            führe Rekursive_Tiefensuche(B_{v_r}, v_r) aus;
            führe Kantennacharbeit(v, v_r)   aus;
    }
    führe Knotenarbeit_3(v) aus;

}
```

Häufig sind den Kanten keine Daten zugeordnet, die bearbeitet werden, sondern es werden nur die Daten in den Knoten bearbeitet. Dann vereinfacht man obiges Schema zu dem folgenden.

15.4.5.1 Rekursives Traversieren eines Binärbaumes

Eingabe: Ein Binärbaum $B = (V, E)$ mit Wurzel w.
Ausgabe: erfolgt anwendungsabhängig in den Bearbeitungsprozeduren.

Rufe Durchlaufe(B, w) auf.

Durchlaufe(B, v) {
  Bearbeitung_1 von v;
  `if(`v hat einen linken Sohn v_l`)` Durchlaufe(B_{v_l}, v_l);
  Bearbeitung_2 von v;
  `if(`w hat einen rechten Sohn v_r`)` Durchlaufe(B_{v_r}, v_r);
  Bearbeitung_3 von v;
}

Wird nur eine der drei Bearbeitungen von v durchgeführt, so gibt man den Durchlaufstrategien entsprechende Namen:

Preorder-Durchlauf oder **WLR-Durchlauf**: Nur Bearbeitung_1 findet statt.

Inorder-Durchlauf oder **LWR-Durchlauf**: Nur Bearbeitung_2 findet statt.

Postorder-Durchlauf oder **LRW-Durchlauf**: Nur Bearbeitung_3 findet statt.

15.5 Binäre Suchbäume

15.5.1 Aufgabe: S sei eine Menge. Konstruiere eine Datenstruktur zur Darstellung endlicher Teilmengen $Y \subset S$, so daß die folgenden Operationen effizient durchführbar sind:

- search(x, Y) stelle fest, ob $x \in Y$.
- insert(x, Y) konstruiere die Datenstruktur, die $Y \cup \{x\}$ darstellt.
- delete(x, Y) konstruiere die Datenstruktur, die $Y \setminus \{x\}$ darstellt.

Eine Datenstruktur mit diesen Operationen wird oft **Lexikon** genannt.

Es gibt unterschiedliche Ansätze für die Konstruktion solcher Datenstrukturen. Welche man wählt, hängt davon ab, ob in der jeweiligen Anwendung zusätzliche Eigenschaften der Elemente von S gegeben sind, mit denen es möglich wird, obige Operationen effizienter zu gestalten als im allgemeinen Fall. Wir betrachten hier die search-Operation.

Im Allgemeinen kann man die Operation search nur dadurch realisieren, dass man die Elemente von Y nacheinander darauf überprüft, ob sie gleich x sind. Sind sie z.B. als Folge $y_1, \ldots, y_n$ gegeben, so führt man die folgende lineare Suche aus.

15.5.2 Algorithmus: Lineare Suche

LinSuche$(x, y_1, \ldots, y_n)$ {
    $j = 1$;
    while$(j \leq n)$ {
        if$(x = y_j)$ gib j aus und stoppe;
        $j = j + 1$;
    }
    gib aus, daß es kein j gibt;
}

Im besten Fall braucht die lineare Suche nur 1 Vergleich, wenn nämlich $x = y_1$ ist, und im schlechtesten Fall n Vergleiche, nämlich wenn x in der Folge gar nicht vorkommt.

Häufig ist auf der Menge S eine Ordnungsrelation $<$ gegeben; dann läßt sich ein Lexikon mit Hilfe eines binären Suchbaums realisieren.

15.5.3 Definition *Ein **binärer Suchbaum** ist ein binärer Baum B, auf dessen Knotenmenge V eine Abbildung $\sigma\colon V \to S$ in eine geordnete Menge S gegeben ist, so daß für jeden Knoten $v \in V$ und alle Knoten u_l im linken Teilbaum von B_v und alle Knoten u_r im rechten Teilbaum von B_v gilt:*

$$\sigma(u_l) \leq \sigma(v) \leq \sigma(u_r)$$

Die Elemente von S nennt man Schlüssel und die Abbildung σ Schlüsselbelegung. $\sigma(V)$ ist die dargestellte endliche Teilmenge Y von S. Die Suche in einem Suchbaum (B, w) nach einem Knoten mit vorgegebenem Schlüssel $x \in S$ kann leicht rekursiv realisiert werden.

15.5.4 Algorithmus search in binären Suchbäumen

Rufe search$(x, (B, w))$ auf.

search$(x, (B, v))$ {
    `if(`$\sigma(v)$`==`x`)` gib v aus und stoppe`;`
    `if(`$x < \sigma(v)$ und v hat einen linken Sohn u_l`)` search$(x, (B_{u_l}, u_l))$`;`
    `else if(`$x > \sigma(v)$ und v hat einen rechten Sohn u_r`)` search$(x, (B_{u_r}, u_r))$`;`
    `else` gib aus, daß es keinen Knoten mit Schlüssel x gibt;
}

Die Anzahl der Schlüsselvergleiche, die dieser Algorithmus ausführt, ist durch die Höhe des Baumes nach oben beschränkt. Im Extremfall kann ein binärer Baum in eine lineare Liste ausarten, und dann läuft der Algorithmus einfach an dieser Liste entlang, artet also in eine lineare Suche aus und braucht im ungünstigsten Fall genau so viele Schlüsselvergleiche, wie Knoten im Baum sind. Gelingt es aber, Suchbäume zu konstruieren, die ausgeglichen sind in dem Sinne, daß die Tiefe ihrer Blätter möglichst wenig schwankt, dann wird der Suchalgorithmus wesentlich schneller sein. Man kann sich überlegen, dass dann die Anzahl der Schlüsselvergleiche in der Größenordnung von $\log n$ liegt, wenn n die Anzahl der Knoten ist. Das eigentliche Problem dabei besteht darin, die Operationen insert und delete so zu gestalten, daß sie wieder einen möglichst ausgeglichenen Suchbaum erzeugen und trotzdem schnell sind. 1962 konstruierten Adelson, Velskii und Landis die später nach ihnen benannten AVL-Suchbäume. Seit-

her wurden noch weitere, für unterschiedliche Anwendungen geeignete Suchbaumtypen entwickelt.

Eine besonders günstige Situation liegt vor, wenn die Elemente der zu durchsuchenden Menge Y als aufsteigend sortierte Folge $y_1, \ldots, y_n$ vorliegt. Dann kann man mit einem einfachen Divide-et-Impera-Ansatz die Suche wesentlich beschleunigen: Ist $x = y_{\lfloor \frac{n}{2} \rfloor}$, so ist man fertig. Ist $x < y_{\lfloor \frac{n}{2} \rfloor}$, so kann x höchstens in der linken Hälfte der Schlüsselfolge sein, ansonsten in der rechten. Erreicht die Rekursion eine Teilfolge der Länge 1 und ist x nicht gleich dem einen verbleibenden Element, so kommt x in der Folge $y_1, \ldots, y_n$ nicht vor. In jedem Fall erfolgt der Abbruch der Rekursion.

Wir stellen den Algorithmus als Funktion dar, die mit $\mathsf{BinSuche}(x, y_1, \ldots, y_n)$ aufgerufen wird, wobei $y_1, \ldots, y_n$ eine aufsteigend sortierte Folge ist.

15.5.5 Binäre Suche

Eingabe: Eine aufsteigend sortierte Folge $y_l, \ldots, y_r$ und ein Schlüssel x.

Ausgabe: Ein Index j mit $x = y_j$ oder die Meldung `"x kommt nicht vor"`

BinSuche$(x, y_l, \ldots, y_r)$ {
    $n = r - l + 1$;
    if$(x = y_{l+\lfloor \frac{n}{2} \rfloor})$ gib $l + \lfloor \frac{n}{2} \rfloor$ aus und stoppe;
    else if$(n = 1)$ gib aus `"x kommt nicht vor"` und stoppe;
    else if$(x < y_{l+\lfloor \frac{n}{2} \rfloor})$ $\mathsf{BinSuche}(x, y_l, \ldots, y_{l+\lfloor \frac{n}{2} \rfloor - 1})$;
        else $\mathsf{BinSuche}(x, y_{l+\lfloor \frac{n}{2} \rfloor}, \ldots, y_r)$;
}

Die größte Zahl an Vergleichen benötigt der Algorithmus, wenn x in der Folge nicht vorkommt. Bei jedem Rekursionsschritt wird die Länge der noch zu durchsuchenden Teilfolge mindestens halbiert. Spätestens nach $\lceil \log_2 n \rceil$ Schritten hat die zu durchsuchende Teilfolge nur noch die Länge 1. Somit werden insgesamt höchstens $\lceil \log_2 n \rceil + 1$ Schlüsselvergleiche durchgeführt.

15.5.6 Rechenaufwand Der Rechenaufwand der binären Suche im schlechtesten Fall liegt in $O(\log n)$.

In dem JAVA-Paket `java.util` gibt es die Klassen `Arrays` und `Collections`, die Methoden für die binäre Suche haben.

Kapitel 16

Sortieren

16.1 Grundbegriffe

16.1.1 Aufgabenstellung beim Sortieren

Gegeben sei eine Folge von Datensätzen $D_1, \ldots, D_n$ und eine geordnete Menge S mit der Ordnungsrelation $\leq$. Zu jedem Datensatz D_j gehöre ein sogenannter Schlüssel $a_j \in S$ (der meist in einer Komponente des Datensatzes D_j abgespeichert ist).
Gesucht ist eine monoton wachsende Anordnung $a_{\pi(1)} \leq \ldots \leq a_{\pi(n)}$ der Schlüssel $a_1, \ldots, a_n$, wobei π eine Permutation von $\{1, \ldots, n\}$ ist.
Man nennt dann $D_{\pi(1)}, \ldots, D_{\pi(n)}$ eine nach den Schlüsseln a_j sortierte Folge.

16.1.2 Bemerkungen

1. Typische Beispiele für die Schlüsselmenge S sind $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ oder A^* d.h die Menge der Wörter über einem Alphabet A; dabei ist auf A irgendeine Ordnung gegeben, und auf S wird die induzierte lexikografische Ordnung genommen.

2. Für die prinzipielle Lösung der Sortierverfahren spielt die spezielle Gestalt der Datensätze und Schlüssel keine Rolle. Man kann daher (wie wir es bisher auch getan haben) o.B.d.A $S = \mathbb{N}$ annehmen und den sonstigen Inhalt der Datensätze vernachlässigen. In der Praxis muß man jedoch auf die Größe der Datensätze und die Art der Schlüssel Rücksicht nehmen, weil sie die Zeit der Kopier- und Vergleichsoperationen beeinflussen. Man wird oft mit Zeigern auf die Datensätze und nicht mit den Datensätzen selbst arbeiten.

3. Nach IBM-Angaben wird 25% der Rechenzeit in kommerziellen Systemen für Sortieren verwendet. Das liegt daran, daß Sortieren als Basisoperation in vielen Algorithmen vorkommt; insbesondere kann die Suche von Datensätzen durch vorheriges Sortieren wesentlich beschleunigt werden.

16.1.3 Eigenschaften von Sortierverfahren

Interne Sortierverfahren sind darauf ausgelegt, daß sämtliche Datensätze im Arbeitsspeicher (RAM) des Rechners liegen, also insbesondere schnell in beliebiger Reihenfolge auf die Datensätze zugegriffen werden kann.
Externe Sortierverfahren sind darauf ausgelegt, daß die Datensätze auf externen

Datenträgern (Magnetbändern, Festplatten, Disketten) liegen, die nur sequentiell oder blockweise gelesen werden können.

Ein Sortierverfahren heißt **am Ort** (in place, in situ), wenn neben dem Speicherbedarf für die Eingabe nur noch ein konstanter d.h. von der Größe der Eingabe unabhängiger Speicherplatz benötigt wird.

Ein Sortierverfahren heißt **stabil**, wenn die relative Position der Datensätze mit gleichem Schlüssel erhalten bleibt d.h. wenn aus $a_i = a_j$ und $i < j$ stets $\pi(i) < \pi(j)$ folgt.

16.2 Überblick über Sortierverfahren

16.2.1 Der Begriff des Rechenaufwands

Sei S eine Schlüsselmenge mit Ordnungsrelation $<$.

16.2.1.1 Definition Ist `A` ein Sortieralgorithmus, und ist $a_1, \ldots, a_n$ eine Folge von Schlüsseln, so sei $T_{\mathtt{A}}(a_1, \ldots, a_n)$ die Anzahl der Vergleiche von zwei Schlüsseln, die der Algorithmus beim Sortieren der Folge $a_1, \ldots, a_n$ durchführt.

In der Praxis kann der wirkliche Zeitaufwand stark schwanken, weil z.B. der Vergleich von Strings viel aufwendiger als der von Integers ist. Und auch die Anzahl der Kopieroperationen von Datensätzen (bzw. von Zeigern auf Datensätze) kann eine Rolle spielen. Soll eine große Zahl von Datensätzen sortiert werden, so ist auch der Speicherbedarf des Verfahrens wichtig und somit insbesondere, ob es am Ort arbeitet. Trotzdem teilt man die Sortieralgorithmen erst mal nach dem asymptotischen Verhalten der benötigten Schlüsselvergleiche ein.

16.2.1.2 Definition Für einen Sortieralgorithmus `A` und $n \in \mathbb{N}$ definiert man als

- **Rechenaufwand im schlechtesten Fall (worst case)**

$$T_{\mathtt{A}}^{worst}(n) = \max\{T_{\mathtt{A}}(a_1, \ldots, a_n) : a_j \in S\}$$

- **Rechenaufwand im besten Fall (best case)**

$$T_{\mathtt{A}}^{best}(n) = \min\{T_{\mathtt{A}}(a_1, \ldots, a_n) : a_j \in S\}$$

Der Rechenaufwand im besten und im schlechtesten Fall kann sehr verschieden sein. Es zeigt sich aber, dass einige Sortierverfahren in praktischen Anwendungen recht schnell sind, obwohl ihr Rechenaufwand im schlechtesten Fall hoch ist. Daher hat man den Begriff des mittleren Rechenaufwandes geprägt.

16.2.1.3 Definition

Seien `A` ein Sortieralgorithmus und $a_1, \ldots, a_n$ eine Folge paarweise verschiedener Schlüssel. Dann heißt

$$T_{\mathtt{A}}^{average}(n) = \frac{1}{n!} \sum_{\pi} T_{\mathtt{A}}(a_{\pi(1)}, \ldots, a_{\pi(n)})$$

der **mittlere Rechenaufwand** oder **Rechenaufwand im Mittel** von `A`, wobei sich die Summe über alle Permutationen π von $\{1, \ldots, n\}$ erstreckt.

$T_{\mathtt{A}}^{average}(n)$ ist also der arithmetische Mittelwert der Rechenzeiten für alle möglichen Anordnungen der n Schlüssel $a_1, \ldots, a_n$. Dieser Wert könnte von der speziellen Wahl der Schlüssel $a_1, \ldots, a_n$ abhängen. Man macht jedoch die – meist verschwiegene – folgende Annahme:

Sind $a_1, \ldots, a_n$ und $b_1, \ldots, b_n$ zwei Schlüsselfolgen gleicher Länge und gilt für alle Indexpaare (i, j) die Ungleichung $a_i \leq a_j$ genau dann, wenn $b_i \leq b_j$ gilt, so braucht der Algorithmus zum Sortieren der Folge $a_1, \ldots, a_n$ genau so viele Schlüsselvergleiche wie zum Sortieren der Folge $b_1, \ldots, b_n$.

Diese Annahme ist für die üblichen Algorithmen tatsächlich auch erfüllt.

Der obige Begriff von mittlerem Rechenaufwand ist in allen Anwendungen relevant, in denen jede Anordnung der zu sortierenden Schlüsselfolge in etwa gleich oft vorkommen kann.

16.2.2 Einteilung der Sortierverfahren

Für die Auswahl eines Sortierverfahrens spielt in der Praxis natürlich der Rechenaufwand eine wichtige Rolle, aber auch andere Eigenschaften wie z.B. der Speicherbedarf können für die Auswahl entscheidend sein. Bevor wir Einzelheiten besprechen, geben wir schon mal eine Übersicht an.

16.2.2.1 Einfache interne Sortierverfahren

Sortieren durch Auswahl (Selection Sort)
Sortieren durch Einfügen (Insertion Sort)
Sortieren durch Austausch, speziell Bubble Sort

Alle diese Verfahren arbeiten am Ort und sind für kleine n recht schnell. Ihr Rechenaufwand im mittleren und schlechtesten Fall ist aber in $\Theta(n^2)$.

16.2.2.2 Höhere interne Sortierverfahren

Für das asymptotische Verhalten des Rechenaufwandes gilt

	im besten Fall	im schlechtester Fall	im Mittel	
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Quick Sort	$\Omega(n \log n)$	$\Theta(n^2)$	$\Theta(n \log n)$	
Heap Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	am Ort

16.2.2.3 Externe Sortierverfahren sind Verfahren, die zum Sortieren großer Datenmengen, die auf externen Speichermedien wie Magnetbändern liegen, geeignet sind. Es sind oft Varianten von Merge Sort.

16.2.2.4 Fächersortierverfahren Die obigen Sortierverfahren werden allgemeine Sortierverfahren genannt, weil sie über die Schlüsselmenge S keine besondere Voraussetzungen machen und Vergleiche gemäß der Ordnungsrelation benutzen. Für kleinere Schlüsselmengen S kann man Sortierverfahren verwenden, die auf der Verteilung der Schlüssel auf vorgegebene Fächer beruhen, also auf dem simultanen Vergleich eines Schlüssels mit vielen Werten. Dazu gehören *Bucket Sort* und *Radix Sort*. Ihr Rechenaufwand wächst nur linear mit der Länge n der zu sortierenden Folge. Hat die Menge aller möglichen Schlüssel höchstens m Elemente, so ist der Aufwand von Bucket Sort asymptotisch durch $c \cdot (n + m)$ nach oben beschränkt, der von Radix Sort durch $c \cdot n \log m$ für ein $c > 0$. Ihr Einsatz ist jedoch nur sinnvoll, wenn die Schlüsselmenge S recht klein ist; ansonsten werden der Speicherbedarf und auch die reale Rechenzeit zu groß.

Fazit: Es gibt kein schlechthin bestes Sortierverfahren. Die jeweilige Anwendungssituation ist ausschlaggebend für die Wahl.

Die in Θ verborgenen multiplikativen Konstanten sind bei Merge Sort, Quick Sort und Heap Sort recht klein, so daß die Verfahren schon für kleine n (d.h. etwa $>$ 10) schneller als die einfachen Sortierverfahren sind. Im Mittel ist Quick Sort am schnellsten. Sind jedoch in einer Anwendung die Eingaben nicht gleichwahrscheinlich, so kann Quick Sort langsam werden; so mag Quick Sort in seiner Urform z.B. keine teilweise vorsortierten Eingaben.

In der Praxis will man meist eine Folge von Datensätzen $D_1, \ldots, D_n$ sortieren, in denen eine Komponente den Schlüssel a_j enthält. Weil der sonstige Inhalt der D_j beim Sortieren keine Rolle spielt, werden wir lediglich die Folge der Schlüssel $a_1, \ldots, a_n$ sortieren. Wir gehen davon aus, daß $a_1, \ldots, a_n$ in einem Array `A[1...n]` abgelegt ist und verwenden Arrays mit Indexbereichen, die nicht notwendig bei `1` anfangen. In praktischen Implementierungen können auch andere Datenstrukturen benutzt werden.

16.3 Einfache Sortierverfahren

16.3.1 Sortieren durch Auswahl

```
SelectionSort(A[1...n]) {
    for (i = 1; i < n; i = i + 1) {
        for (j = i + 1; j ≤ n; j = j + 1) {
            if (A[j] < A[i]) {vertausche A[i] und A[j]; }
        }
    }
}
```

16.3.2 Sortieren durch Einfügen

```
InsertionSort( A[1...n] ) {
    for (i = 2; i ≤ n; i = i + 1) {
        a = A[i];
        j = i;
        while (j ≥ 2 und A[j − 1] > a) {
            A[j] = A[j − 1];
            j = j − 1;
        }
        A[j] = a;
    }
}
```

In der **while**-Schleife ist $A[1 \ldots i-1]$ bereits sortiert, und $A[i]$ wird so in $A[1 \ldots i]$ eingefügt, daß $A[1 \ldots i]$ sortiert ist. Die rechts von der neuen Position von $A[i]$ stehenden Elemente werden um eine Position nach rechts verschoben.

16.3.3 Sortieren durch Austausch

Idee: Durchlaufe die Schlüsselfolge $a_1, \ldots, a_n$ und vertausche dabei a_{j-1} und a_j, wenn $a_{j-1} > a_j$. Wiederhole dies, bis kein Austausch mehr möglich ist.

```
BubbleSort1( A[1...n] ) {
    do {
        ausgetauscht = nein;
        for (i = 2; i ≤ n; i = i + 1) {
            if ( A[i − 1] > A[i] ) {
                vertausche A[i − 1] und A[i];
                ausgetauscht = ja;
            }
        }
    } while (ausgetauscht == ja);
}
```

Beweis der Korrektheit: Wenn der Algorithmus terminiert, wurde kein Austausch mehr vorgenommen, und das bedeutet gerade, daß das Array sortiert ist. Es ist also nur zu zeigen, daß der Algorithmus terminiert.
Beachte:
Nach dem 1-ten Durchlauf durch die **while**-Schleife steht in $A[n]$ das Maximum von $A[1 \ldots n]$.
Nach dem 2-ten Durchlauf durch die **while**-Schleife steht in $A[n-1]$ das Maximum von $A[1 \ldots n-1]$ usw. Nach dem $(n-1)$-ten Durchlauf durch die **while**-Schleife steht in $A[2]$ das Maximum von $A[1 \ldots 2]$. Somit wird spätestens beim n-ten Durchlauf kein Austausch mehr vorgenommen, und der Algorithmus terminiert. q.e.d.

Nach obiger Beobachtung enthält das Array $A[n-j+1 \ldots n]$ nach dem j-ten Durchlauf bereits die j größten Elemente in sortierter Reihenfolge; deshalb kann man den Algorithmus ein wenig effizienter machen.

```
BubbleSort2( A[1...n] ) {
    for (j = n; j ≥ 2; j = j − 1) {
        for (i = 2; i ≤ j; i = i + 1)
            if ( A[i − 1] > A[i] ) vertausche A[i − 1] und A[i];
    }
}
```

Durch Verwendung einer Variablen *ausgetauscht* wie in BubbleSort1 kann das Verhalten im besten Fall verbessert werden.

16.3.4 Rechenaufwand Bei den obigen einfachen Sortierverfahren liegt die Anzahl der benötigten Vergleiche im schlechtesten und im mittleren Fall stets in $\Theta(n^2)$. Im besten Fall brauchen manche weniger Vergleiche.

16.3.5 Bemerkung Die obigen einfachen Sortierverfahren arbeiten alle **am Ort**. Für kleine n sind sie den im Folgenden aufgeführten komplexeren Sortierverfahren vorzuziehen. Aber wirklich nur für etwa $n < 20$. Wir wollen auch noch eine Bemerkung über BubbleSort aud dem Buch *Numerical recipes in C* zitieren (S.330): If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

16.4 Merge Sort

Die prinzipielle Gestalt ist folgende:

MergeSort$(A[m \dots k])$ {
  if$(m < k)$ {
    $n = k - m + 1;$
    $l = \lfloor \frac{n}{2} - 1 \rfloor;$
    MergeSort$(A[m \dots m + l]);$
    MergeSort$(A[m + l + 1 \dots k]);$
    verschmelze $A[m \dots m + l]$ und $A[m + l + 1 \dots k]$
    zu dem sortierten Array $A[m \dots k];$
  }
}

16.4.1 Rechenaufwand

Die Anzahl $T(n)$ der Schlüsselvergleiche, die von dem Algorithmus durchgeführt werden, ist unabhängig von den Werten der Glieder der zu sortierenden Folge $a_1, \dots, a_n$. Lediglich ihre Anzahl n spielt eine Rolle. Wie man unmittelbar sieht, gilt für T folgende Rekursionsgleichung

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + M(n), \quad T(1) = 0$$

wobei $M(n)$ die Anzahl der Schlüsselvergleiche ist, die zum Verschmelzen der beiden bereits sortierten Arrays $A[m \dots m + l]$ und $A[m + l + 1 \dots k]$ benötigt werden. Diese Rekursionsgleichung beschreibt also den Aufwand sowohl im besten als auch im schlechtesten Fall.

Wir geben zunächst einen Verschmelzungsalgorithmus an.

Merge$(\ A[1 \dots n], A_1[1 \dots n_1], A_2[1 \dots n_2]\ )$ {
  $i = j = k = 1;$
  while $(\ (i \le n_1)$ und $(j \le n_2)\ )$ {
    if $(A_1[i] < A_2[j])\ \{\ A[k] = A_1[i];\ i = i + 1;\ k = k + 1;\ \}$
    else $\{\ A[k] = A_2[j];\ j = j + 1;\ k = k + 1;\ \}$
  }
  while$(i \le n_1)\ \{\ A[k] = A_1[i];\ i = i + 1;\ k = k + 1;\ \}$
  while$(j \le n_2)\ \{\ A[k] = A_2[j];\ j = j + 1;\ k = k + 1;\ \}$
}

Dieser Algorithmus benötigt im schlechtesten Fall $n_1 + n_2 - 1$ Vergleiche zwischen den Elementen von A_1 und A_2. Innerhalb von MergeSort ist $n_1 = \lfloor \frac{n}{2} \rfloor$ und $n_2 = \lceil \frac{n}{2} \rceil$, also werden im schlechtesten Fall $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil - 1 = n - 1$ Vergleiche ausgeführt. Wir zeigen, daß es nicht besser geht.

16.4.1.1 Lemma Jeder Algorithmus, der zwei beliebige sortierte Folgen $x_1 \le x_2 \le \dots \le x_{\lfloor \frac{n}{2} \rfloor}$ und $y_1 \le y_2 \le \dots \le y_{\lceil \frac{n}{2} \rceil}$ zu einer sortierten Folge $z_1 \le z_2 \le \dots \le z_n$ mischt, benötigt dazu im schlechtesten Fall $n - 1$ Vergleichsoperationen.

Beweis: Wähle die x_i und y_j so, daß $y_1 < x_1 < y_2 < x_2 < \dots < x_{\lfloor \frac{n}{2} \rfloor}$ ($< y_{\lceil \frac{n}{2} \rceil}$, falls n ungerade). Angenommen, der Algorithmus führt weniger als $n - 1$ Vergleiche durch. Dann kann er nicht alle Paare (y_i, x_i) und (x_i, y_{i+1}) vergleichen; denn es gibt

$n-1$ solcher Paare. OBdA habe er y_i und x_i nicht verglichen. Wir tauschen x_i und y_i in den beiden Anfangsfolgen aus. Beide bleiben sortiert! Der Algorithmus führt dieselben Vergleichsoperationen wie bei den ursprünglichen Folgen aus und liefert deshalb dieselbe z-Folge; sie ist aber nicht sortiert, weil jetzt $y_i > x_i$ ist. Dies ist ein Widerspruch! q.e.d.

16.4.1.2 Folgerung Für den Rechenaufwand von Merge Sort gilt

$$T(n) = T(\lfloor \tfrac{n}{2} \rfloor) + T(\lceil \tfrac{n}{2} \rceil) + n - 1, \quad T(1) = 0$$

Diese Rekursionsgleichung hat die Lösung $T(n) = n \cdot \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$, was ein wenig mühsam herzuleiten und zu verifizieren ist. Wir beschränken uns hier auf die Fälle, dass n eine Zweierpotenz ist, also $n = 2^k$ mit $k \in \mathbb{N}_0$ gilt, und setzen $S(k) := T(2^k)$. Dann erfüllt S die Rekursionsgleichung

$$S(k) = 2 \cdot S(k-1) + 2^k - 1, \quad S(0) = 0$$

Für solche Rekursionsgleichungen gibt es eine vollständige Lösungstheorie, auf die wir hier aber nicht näher eingehen wollen. Aus ihr lässt sich herleiten, dass $S(k) = k \cdot 2^k - 2^k + 1$ die (eindeutig bestimmte) Lösung ist. Dass dies stimmt, können wir zumindest überprüfen:

$$\begin{aligned} S(k) - 2 \cdot S(k-1) &= k \cdot 2^k - 2^k + 1 - 2((k-1)2^{k-1} - 2^{k-1} + 1) \\ &= k2^k - 2^k + 1 - k2^k + 2^k + 2^k - 2 \\ &= 2^k - 1 \end{aligned}$$

Mit $k = \log_2 n$ und $T(n) = S(\log_2 n)$ ergibt sich dann

16.4.1.3 Satz Zum Sortieren einer Schlüsselfolge mit Zweierpotenzlänge n benötigt Merge Sort

$$T(n) = n \cdot \log_2 n - n + 1$$

viele Schlüsselvergleiche.

Zusammenfassend ergibt sich die folgende Aussage.

16.4.1.4 Satz Für den Rechenaufwand von Merge Sort gilt

$$T^{best} = T^{average} = T^{worst} \in O(n \log n)$$

16.4.2 Speicherbedarf

Beim Verschmelzen wird ein Hilfsarray benötigt, dessen Länge die Summe der Längen der beiden Teilarrays, im schlechtesten Fall also n ist. Außerdem wird für die Rekursion Speicherplatz in der Größenordnung von $O(\log n)$ benötigt.

Die obige Form von Merge Sort arbeitet also nicht am Ort, sondern benötigt mindestens $2n + \log n$ Speicherplätze.

16.4.3 Bemerkungen

1. Es gibt Modifikationen von Merge Sort, die mit $n + const$ viel Speicherplatz auskommen (siehe Zitate in dem Buch von *Ottmann, Widmayer*).

2. Man kann Merge Sort auch rein iterativ formulieren. Dazu fängt man "unten´´ an (bottom up) d.h. man betrachtet die a_i als 1-gliedrige Folgen und verschmilzt

jeweils a_{2i-1} und a_{2i} zu einer 2-gliedrigen Folge. Diese 2-gliedrigen Folgen werden dann wiederum zu 4-gliedrigen Folgen verschmolzen usw.

3. Der Verschmelzungsschritt in Merge Sort kann leicht so entworfen werden, daß Merge Sort **stabil** ist.

4. Die Strategie von Merge Sort besteht darin, die ursprüngliche Aufgabe in zwei Aufgaben der gleichen Art, aber mit kleineren Eingabemengen aufzuteilen und dann aus den Lösungen dieser „kleineren" Aufgaben die Lösung des ursprünglichen Problems zu konstruieren. Dieses Vorgehen wird rekursiv solange angewendet, bis die entstehenden kleineren Aufgaben trivial oder sehr einfach zu lösen sind. Diese Art von Strategie führt bei vielen, sehr unterschiedlichen Aufgaben zu einer erfolgreichen Lösung. Man nennt sie **Divide-et-Impera**-Strategie oder auch *Teile-und-Herrsche* oder *Divide-and-Conquer*.

16.5 Quick Sort

Quick Sort wurde 1962 von C.A.R. Hoare entwickelt. Es beruht wie Merge Sort auf einer *Divide-et-Impera*-Strategie. Der Unterschied besteht darin, daß die Zerlegung in zwei Teile aufwendiger gestaltet wird und dadurch das Verschmelzen der sortierten Teilfolgen trivial wird.

16.5.1 Idee

(1) Zerlege die Schlüsselfolge $a_1, \ldots, a_n$ in zwei Folgen $b_1, \ldots, b_{n_1}$ und $c_1, \ldots, c_{n_2}$, so daß $b_i \leq c_j$ für alle $i \in \{1, \ldots, n_1\}$ und $j \in \{1, \ldots, n_2\}$.

(2) Sortiere $b_1, \ldots, b_{n_1}$ und $c_1, \ldots, c_{n_2}$.

(3) $b_1, \ldots, b_{n_1}, c_1, \ldots, c_{n_2}$ ist dann eine sortierte Permutation von $a_1, \ldots, a_n$ d.h. das Verschmelzen besteht nur aus einem Aneinanderhängen.

Wesentlich ist der Zerlegungsschritt (1). Er wird bei Quick Sort typischerweise folgendermaßen durchgeführt:

- Wähle ein $p \in \{1, \ldots, n\}$.
- Teile $a_1, \ldots, a_n$ in zwei Teilfolgen auf; die erste bestehe aus allen a_i mit $a_i \leq a_p$, die zweite aus allen a_i mit $a_i > a_p$.

a_p wird **Pivotelement** genannt. Die verschiedenen Varianten von Quick Sort unterscheiden sich vor allem durch unterschiedliche Wahlen des Pivotelements. Wir werden im Folgenden a_1 als Pivatelement wählen. Es gibt aber andere heuristische Wahlen, die in manchen Situationen vorteilhafter sind.

Ist die Schlüsselfolge $a_1, \ldots, a_n$ in einem Array $A[1 \ldots n]$ gespeichert, so kann man dieses Array so umordnen, daß die erste Teilfolge in $A[1 \ldots k-1]$ und die zweite in $A[k+1 \ldots n]$ und a_p in $A[k]$ stehen.

16.5.2 Zerlegungschritt

```
Zerlege( A[1...n] ) {
    i = 1;
    j = n + 1;
    pivot = A[1];
    while ( i < j ) {
        do i = i + 1 while ( A[i] ≤ pivot und i < j );
        do j = j − 1 while ( A[j] > pivot und i ≤ j );
        if ( i < j ) vertausche A[i] und A[j];
    }
    vertausche A[1] und A[j];
}
```

16.5.3 Bemerkungen

1. Bei Beendigung der **while**-Schleife ist $i = j$ oder $i = j + 1$, also $j - i = 0$ oder $j - i = -1$. Da am Anfang $j - i = n$ war, werden insgesamt n oder $n + 1$ Vergleiche ausgeführt.

2. Außer dem Array wird kein zusätzlicher Speicher benötigt.

3. Obiger Zerlegungsschritt erhält nicht die relative Position gleicher Schlüssel.

4. Die Abfragen in den beiden **do**-Schleifen, ob $i < j$ bzw. $i \leq j$ ist, dienen nur dazu, daß der Indexbereich des Arrays nicht überschritten wird. Wenn z.B. $A[k] < pivot$ für alle $k \in \{2, \ldots, n\}$, so würde die erste **do**-Schleife nicht enden. Statt $i < j$ kann man in der ersten **do**-Schleife auch $i < n$ und in der zweiten $j > 1$ prüfen. Man kann die Abfragen sogar ganz weglassen, wenn man das Array um ein Element $A[0]$ und ein Element $A[n + 1]$ so erweitert, daß $A[0] < A[k] < A[n + 1]$ für alle $k \in \{1, \ldots, n\}$.

16.5.4 Quick Sort Der gesamte Algorithmus lautet nun folgendermaßen.

```
    QuickSort( A[l...r] ) {
(1)     i = l;
(2)     j = r + 1;
(3)     pivot = A[l];
(4)     while ( i < j ) {
(5)         do i = i + 1 while ( A[i] ≤ pivot und i < j );
(6)         do j = j − 1 while ( A[j] > pivot und i ≤ j );
(7)         if ( i < j ) vertausche A[i] und A[j];
        }
(8)     if ( l < j − 1 ) QuickSort( A[l...j − 1] );
(9)     if ( j + 1 < r ) QuickSort( A[j + 1...r] );
    }
```

Man kann die do-while-Schleifen in dem Zerlegungsschritt auch umdrehen und erhält dann folgende Variante.

```
        QuickSort( A[l ... r] ) {
(1)         i = l;
(2)         j = r;
(3)         pivot = A[l];
(4)         while ( i < j ) {
(5)             while ( A[i] ≤ pivot und i ≤ j ) do i = i + 1;
(6)             while ( A[j] > pivot und i ≤ j ) do j = j − 1;
(7)             if ( i < j ) vertausche A[i] und A[j];
                }
(8)         if ( l < j − 1 ) QuickSort( A[l ... j − 1] );
(9)         if ( j + 1 < r ) QuickSort( A[j + 1 ... r] );
        }
```

16.5.5 Rechenaufwand

Wir geben hier nur die wichtigsten Resultate an und verzichten auf die Beweise, die etwas mühselig sind, wenn man sie wirklich exakt durchführt.

16.5.5.1 Schlechtester Fall: Es gilt $T^{\text{worst}} \in \Theta(n^2)$

Achtung! Falls als Pivotelement stets das erste Element gewählt wird, hat QuickSort quadratische Laufzeit, wenn die Datenfolge bereits sortiert ist.

16.5.5.2 Verhalten im Mittel: Es gilt $T^{\text{average}} \in O(n \log n)$

16.5.6 Speicherbedarf

Im Zerlegungsschritt braucht QuickSort außer dem zu sortierenden Array nur noch drei zusätzliche Speicherstellen (in Zeile (7) zum Vertauschen von $A[i]$ und $A[j]$ und für die Indizes i und j). Bei der Rekursion wird jedoch Speicherplatz benötigt, um sich die lokalen Variablen l, j, r und die Rücksprungadressen usw. zu merken. Im schlechtesten Fall erreicht er eine Länge in $\Theta(n)$. Deshalb arbeitet QuickSort nicht am Ort.

Es gibt aber Modifikationen, die nur $O(\log n)$ oder sogar nur $O(1)$ zusätzlichen Speicher benötigen, also am Ort arbeiten (zu Lasten der Geschwindigkeit); siehe z.B. das Buch von Ottmann und Widmayer.

16.5.7 Bemerkungen

In der Praxis ist Quick Sort oft etwas schneller als Merge Sort. Allerdings ist eine Laufzeit von $O(n \log n)$ nicht garantiert! Vor allem wenn die zu sortierenden Folgen nicht zufällig sind, sondern z.B. teilweise sortiert sind, kann Quick Sort langsam werden. Dem versucht man durch andere Auswahlen des Pivotelementes zu begegnen, z.B. indem man ein Element zufällig oder aus der Mitte der Folge wählt. Ideal wäre es, als Pivotelement immer den Median der Folge zu wählen; dann wären die beiden Teilfolge nahezu gleich groß. Üblicherweise berechnet man den Median aber gerade dadurch. dass man erst die Folge sortiert und dann das mittlere Element wählt. Es gibt jedoch auch eine andere Divide-et-Impera-Strategie für die Berechnung des Me-

dians (Stichwort: Median der Mediane, siehe z.B. *Ottmann/Widmayer: Algorithmen und Datenstrukturen*, S.181), deren Rechenaufwand linear ist, also in $O(n)$ liegt. Damit kann man erreichen, dass der Rechenaufwand von quickSort tatsächlich immer in $O(n \log n)$ liegt. Allerdings ist die Konstante bei der O-Abschätzung recht groß, so dass sich diese Strategie für übliche Größen von n nicht lohnt.

16.6 Heap Sort

Heap Sort wurde 1964 von J.W.J. Williams und R.W. Floyd entwickelt. Bei Heap Sort wird wie bei Selection Sort immer ein Datensatz mit maximalem (oder mit minimalem) Schlüssel aus der noch nicht sortierten Restfolge ausgewählt. Um die quadratische Rechenzeit von Selection Sort zu reduzieren, wird die Restfolge stets in einer speziellen Form namens Heap (Halde, Haufen) gebracht, wodurch das Finden eines maximalen Elements trivial wird. Ein Heap ist eine Datenstruktur, und zwar ein Binärbaum mit einer Knotenmarkierung mit einer speziellen Eigenschaft. Genaueres wird im Kapitel über Baume ausgeführt. Wir wollen hier jetzt nur einige Resultate zitieren. Heap Sort ist recht gut als Sortierverfahren für den universellen Einsatz geeignet. Trotzdem sollte man in konkreten Anwendungssituationen immer darüber nachdenken, ob nicht Vorkenntnisse über spezielle Eigenschaften der zu sortierenden Folgen vorliegen und eventuell andere Sortierverfahren geeigneter sind.

16.6.1 Aufwandsanalyse

16.6.1.1 Schlechtester Fall: Es gilt $T^{\text{worst}} \in \Theta(n \log n)$

16.6.1.2 Aufwand im Mittel: Es gilt $T^{\text{average}} \in \Theta(n \log n)$

16.6.1.3 Bester Fall: Der Aufwand im besten Fall ist kleiner als $n \log n$. Es gibt Varianten (Smooth Sort), deren Aufwand bei vorsortierten Eingabefolgen linear ist (im Gegensatz zu Quick Sort!).

16.6.1.4 Speicherbedarf: Heap Sort arbeitet **am Ort**, d.h. außer dem Platz für die Eingabefolge wird nur ein konstanter (von der Länge der Eingabefolge unabhängiger) zusätzlicher Platz benötigt. Heap Sort ist nicht stabil.

16.7 Die C-Funktion `qsort`

In der C-Bibliothek ist eine Funktion namens `qsort` zum Sortieren implementiert. Trotz ihres Namens basiert sie nicht unbedingt auf Quick Sort. Sie hat den Prototyp

```
void qsort( void *base, size_t nmemb, size_t size,
            int (*compar)(const void *, const void *));
```

`base` ist eine Zeiger auf den Anfang der Schlüsselfolge, `nmemb` ist ihre Länge und `size` die Speichergröße jedes Schlüssels. `compar` ist ein Zeiger auf eine Funktion, die Schlüssel vergleicht. Diese Funktion muß man selbst bereitstellen, denn sie wird je nach Typ der Schlüssel sehr unterschiedlich aussehen. Die Funktion `qsort` geht von keinem speziellen Schlüsseltyp aus; deshalb ist `base` auch als Zeiger auf `void` deklariert

ebenso wie die Argumente der Vergleichsfunktion. Die Vergleichsfunktion muß einen negativen Wert zurückgeben, wenn der erste Schlüssel kleiner als der zweite ist, Null, wenn beide gleich sind, und einen positiven Wert sonst. Genaueres kann man in der Manualseite nachlesen.

Am leichtesten ist der Gebrauch von `qsort` vielleicht an folgendem Beispiel zu verstehen. Es sortiert eine Folge von `int`, allerdings mit einer selbst gemachten Vergleichsfunktion.

```
/************************************
          qsort - Demo
************************************/

#include <stdio.h>
#include <stdlib.h>

int IntegerVergleich( const void *x, const void *y) {
  return *(int *)x-*(int *)y;
}

int main(void) {
   int i;
   int (* compar)() = IntegerVergleich;
   int A[10] = { 3,4,100,1,2,2,8,9,7,0 };

   qsort(A,10,sizeof(A[0]),compar);

   for(i=0;i<10;i++) printf("%d ", A[i]);
   printf("\n");

   return 0;
}
```

Geschickte Implementierungen von QuickSort und HeapSort findet man in dem Buch *Press, Teukolsky, Vetterling, Flannery: Numerical Recipes in C*, von dem es auch Versionen für andere Programmeirsprachen gibt.

In dem JAVA-Paket `java.util` gibt es die Klassen `Arrays` und `Collections`, die Methoden fürs Sortieren haben.

16.8 Eine untere Laufzeitschranke für Sortierverfahren, die auf Binärvergleichen basieren

Fächersortierverfahren wie *BucketSort* oder *RadixSort* verwenden keine expliziten Vergleiche von Schlüsseln bezüglich der gegebenen Ordnung. Dadurch dass jedoch

der Schlüssel zur Indizierung der Fächer benutzt wird, wird jeder Schlüssel in der Eingabefolge sogar einer Entscheidung zwischen so vielen Alternativen unterworfen, wie es Fächer gibt. Dadurch gelingt es, lineare Laufzeit zu erzielen.

Anders sieht es aus bei Sortierverfahren, die nur die Entscheidung zwischen zwei Alternativen, nämlich $a_i > a_j$ oder $a_i \leq a_j$ oder ähnliches, benutzen. Solche Verfahren nennt man vergleichsbasiert. Genauer meint man damit Algorithmen, die nach dem Start stets durch eine Binärentscheidung (einen Schlüsselvergleich) in den nächsten Zustand kommen, bis sie einen Endzustand erreichen und terminieren. Sie sind durch einen vollständigen Binärbaum beschreibbar, bei dem jeder innere Knoten einer Binärentscheidung entspricht, die je nach Resultat zu einem der beiden Söhne führt. Da das Ergebnis des Sortierens einer Folge $a_1, \ldots, a_n$ eine Permutation von $\{1, \ldots, n\}$ ist, muss jedem Sortieralgorithmus, der Folgen der Länge n sortiert, ein Entscheidungsbaum mit mindestens $n!$ Blättern entsprechen. Ein Binärbaum mit $n!$ Blättern muss mindestens die Höhe $\lfloor \log_2 n! \rfloor$ haben. Daher gibt es mindestens eine Eingabefolge $a_1, \ldots, a_n$, für die der Sortieralgorithmus $\lfloor \log_2 n! \rfloor$ Vergleiche ausführt. Somit gilt

$$\text{Anzahl der Vergleiche im worst case} \geq \lfloor \log_2 n! \rfloor$$

Nach A.2.4 gilt $\log(n!) \in \Theta(n \log n)$.

Damit ergibt sich:

Die worst-case-Laufzeit jedes vergleichsbasierten Sortieverfahrens liegt in $\Omega(n \log n)$.

Anhang A

Mathematische Ergänzungen

A.1 Einige Grundlagen

A.1.1 Die Exponentialfunktion

Die Reihe

$$\sum_{n=0}^{\infty} \frac{x^n}{n!}$$

konvergiert für alle $x \in \mathbb{R}$; ihr Wert wird mit $\exp(x)$ bezeichnet.
Die Zahl $e = \exp(1)$ heißt Eulerzahl. Es gilt $\exp(x) = e^x$. Die Funktion $\exp\colon \mathbb{R} \to \mathbb{R}, x \mapsto e^x$ wird Exponentialfunktion genannt. Sie ist streng monoton wachsend und beliebig oft differenzierbar. Und es gilt auch

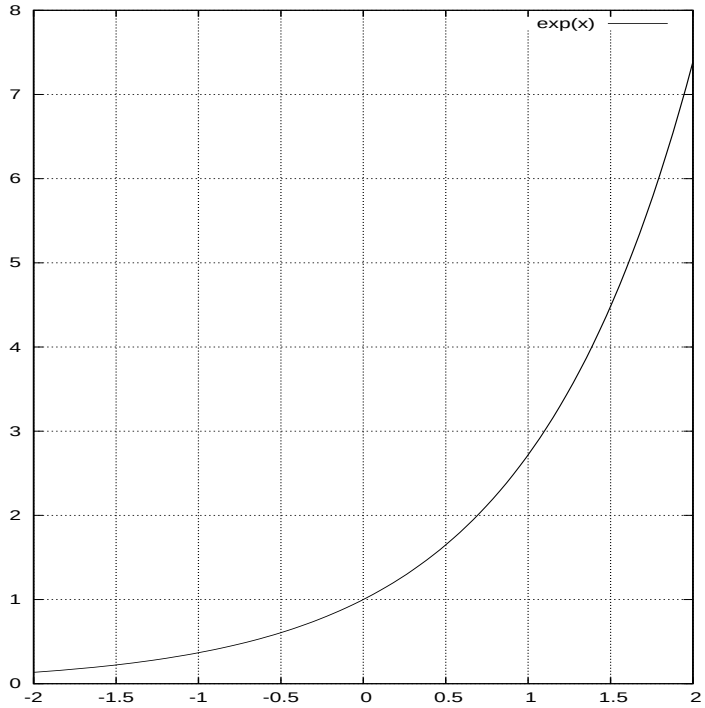
$$e^x = \lim_{n\to\infty} (1 + \frac{x}{n})^n$$

Die Exponentialfunktion besitzt einige besonderen Eigenschaften. Sie erfüllt die Funktionalgleichung

$$e^{x+y} = e^x \cdot e^y \text{ für alle } x, y \in \mathbb{R}$$

und sie ist gleich ihrer Ableitung, d.h. es gilt

$$\frac{d}{dx} \exp = \exp$$



Die Exponentialfunktion ist in allen wissenschaftlichen Bereichen, die mathematische Modellierungen verwenden, von grundlegender Bedeutung.

A.1.2 Die Logarithmusfunktion

Die Umkehrfunktion der Exponentialfunktion heißt die natürliche Logarithmusfunktion

$$\log\colon \mathbb{R}^+ \to \mathbb{R}$$

d.h. es gilt $\log y = x$ genau dann, wenn $y = e^x$ gilt.

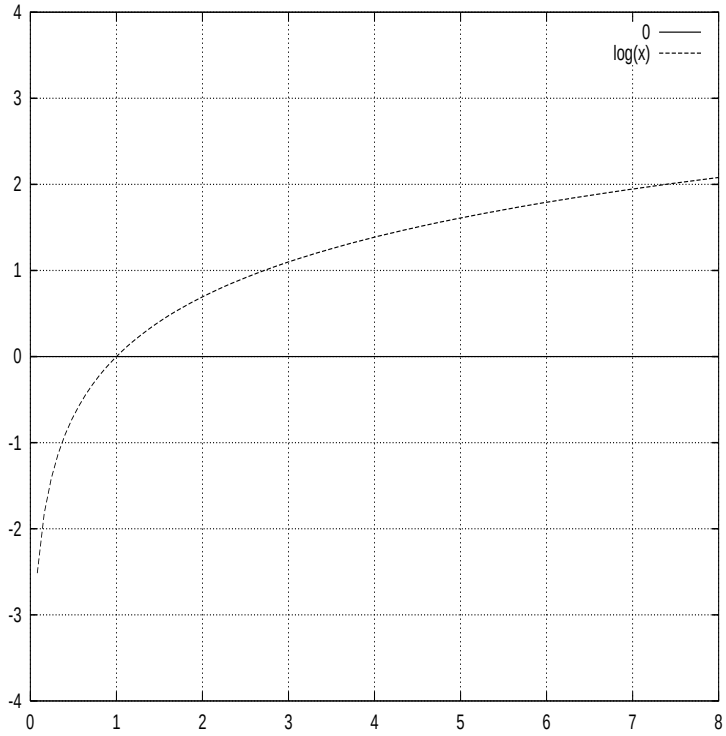
Oft wird sie mit *ln* (*logarithmus naturalis*) bezeichnet.

Die Logarithmusfunktion ist monoton wachsend und stetig, ja sogar beliebig oft differenzierbar. Außerdem gilt $\lim_{x\to 0} \log x = -\infty$ und $\lim_{x\to\infty} \log x = \infty$. Für die Ableitung gilt

$$\log'(x) = \frac{1}{x}$$

und der Logarithmus erfüllt die Funktionalgleichung

$$\log(u \cdot v) = \log u \ + \ \log v \text{ für alle } u, v > 0$$



Es ist üblich, auch Logarithmus- und Exponentialfunktionen bezüglich anderer Basen zu einzuführen. Für $b > 0$ definiert man

$$\log_b(y) = \frac{ln(y)}{ln(b)} \quad \text{und} \quad b^x = e^{x \ln b}$$

Dann gilt

$$y = e^{ln(y)} = e^{ln(b)\cdot \log_b(y)} = b^{\log_b(y)}$$

A.1.3 Die Gaußklammern

A.1.3.1 Definition

Untere Gaußklammer (floor): Für $x \in \mathbb{R}$ sei $\lfloor x \rfloor := \max\{a \in \mathbb{Z} \colon a \leq x\}$.
Obere Gaußklammer (ceiling): Für $x \in \mathbb{R}$ sei $\lceil x \rceil := \min\{a \in \mathbb{Z} \colon a \geq x\}$.

A.1.3.2 Lemma

1. Für $x \in \mathbb{R}$ gilt $\quad x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$
2. Für $a > 0, x \in \mathbb{R}$ gilt $a \cdot \lfloor \frac{x}{a} \rfloor \leq x$ und $a \cdot \lceil \frac{x}{a} \rceil \geq x$
3. Für $x \in \mathbb{R}$ gilt $\lfloor -x \rfloor = -\lceil x \rceil$ und $\lceil -x \rceil = -\lfloor x \rfloor$
4. Für $n \in \mathbb{Z}$ gilt

$$\left\lfloor \frac{n}{2} \right\rfloor = \begin{cases} \frac{n}{2} & \text{, falls } n \text{ gerade} \\ \frac{n-1}{2} & \text{, falls } n \text{ ungerade} \end{cases} \qquad \left\lceil \frac{n}{2} \right\rceil = \begin{cases} \frac{n}{2} & \text{, falls } n \text{ gerade} \\ \frac{n+1}{2} & \text{, falls } n \text{ ungerade} \end{cases}$$

und

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$$

5. Für $n \in \mathbb{N}$ ist $\lfloor \log_2 n \rfloor + 1$ die Länge der Dualdarstellung von n ohne führende Nullen.

A.2 Der Kalkül mit den Symbolen O, Θ, Ω

A.2.1 Definition $D \subset \mathbb{R}^+$ sei eine nichtleere, unbeschränkte Teilmenge.

a) Eine Funktion $f \colon D \to \mathbb{R}$ heißt bei ∞ positiv bzw. nichtnegativ, wenn es ein $x_0 \in \mathbb{R}$ gibt, so daß $f(x) > 0$ bzw. $f(x) \geq 0$ für jedes $x \in D$ mit $x \geq x_0$.

b) Sei $f \colon D \to \mathbb{R}$ bei ∞ nichtnegativ. Dann definiert man

$$O(f) = \{\, g \colon D \to \mathbb{R} \mid \exists\, c > 0\ \exists\, x_0 \in \mathbb{R}\ \forall\, x \in D \\ x > x_0 \Rightarrow 0 \leq g(x) \leq c \cdot f(x) \,\}$$

d.h. $O(f)$ ist die Menge aller Funktionen $g \colon D \to \mathbb{R}$ derart, daß ein $c > 0$ und ein $x_0 \in \mathbb{R}$ existieren, so daß für alle x in D, die größer als x_0 sind, $0 \leq g(x) \leq c \cdot f(x)$ gilt.

$$\Omega(f) = \{\, g \colon D \to \mathbb{R} \mid \exists\, c > 0\ \exists\, x_0 \in \mathbb{R}\ \forall x \in D \\ x > x_0 \Rightarrow 0 \leq c \cdot f(x) \leq g(x) \,\}$$

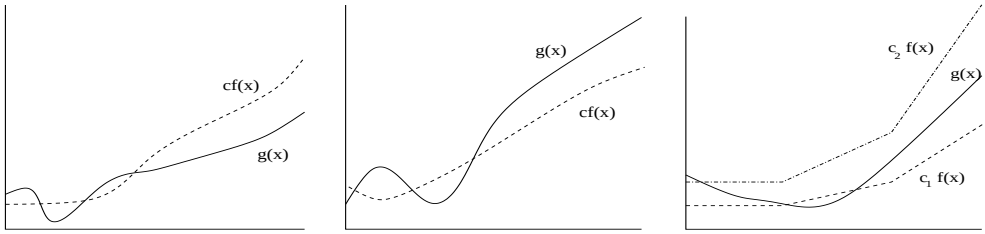
d.h. $\Omega(f)$ ist die Menge aller Funktionen $g \colon D \to \mathbb{R}$ derart, daß ein $c > 0$ und ein $x_0 \in \mathbb{R}$ existieren, so daß für alle x in D, die größer als x_0 sind, $0 \leq c \cdot f(x) \leq g(x)$ gilt.

$$\Theta(f) = \{\, g\colon D \to \mathbb{R} \mid \exists\, c_1, c_2 > 0\ \exists\, x_0 \in \mathbb{R}\ \forall x \in D$$
$$x > x_0 \Rightarrow 0 \leq c_1 \cdot f(x) \leq g(x) \leq c_2 \cdot f(x)\ \}$$

d.h. $\Theta(f)$ ist die Menge aller Funktionen $g\colon D \to \mathbb{R}$ derart, daß $c_1 > 0, c_2 > 0$ und ein $x_0 \in \mathbb{R}$ existieren, so daß für alle x in D, die größer als x_0 sind, $0 \leq c_1 \cdot f(x) \leq g(x) \leq c_2 \cdot f(x)$ gilt.

Die Mengen $O(f), \Omega(f), \Theta(f)$ werden auch Wachstumsklassen genannt.

Die folgenden Diagramme sollen illustrieren, daß das Verhalten für kleine Argumente keine Rolle spielt.



$g \in O(f)$ $g \in \Omega(f)$ $g \in \Theta(f)$

Bemerkung: Bei der Rechenzeitanalyse von Algorithmen ist $D = \mathbb{N}$ und x beschreibt die Problemgröße.

A.2.2 Sprechweisen

$g \in O(f) \iff g$ ist in Groß-O von f
$\iff g$ wächst asymptotisch höchstens so stark wie f
$\iff g$ ist höchstens in der Größenordnung von f

$g \in \Omega(f) \iff g$ ist in Groß-Ω von f
$\iff g$ wächst asymptotisch mindestens so stark wie f
$\iff g$ ist mindestens in der Größenordnung von f

$g \in \Theta(f) \iff g$ ist in Groß-Θ von f
$\iff g$ wächst asymptotisch genau so so stark wie f
$\iff g$ ist genau in der Größenordnung von f

Die Werte von f und g in einem beschränkten Intervall spielen keine Rolle.

A.2.3 Schreibweisen Folgende Schreibweisen sind üblich:
$g = O(f)$ statt $g \in O(f)$, analog $g = \Omega(f)$ statt $g \in \Omega(f)$ und ebenso $g = \Theta(f)$ statt $g \in \Theta(f)$.

Ist f ein Monom, also $f(x) = x^k$ für $x \in D$, so schreibt man meist $O(x^k)$ statt $O(f)$ (entsprechend für Θ und Ω). Ähnlich geht man bei anderen Funktionen vor z.B. $O(x \log x), O(2^x)$. Der Name der Variablen ist unbedeutend; es muß nur klar sein,

bezüglich welcher Variablen die asymptotische Betrachtung angestellt wird. Bei der Rechenzeitanalyse ist $D = \mathbb{N}$, und die Variable wird meist n genannt. Man schreibt dann also $O(n^k), O(n \log n)$ usw.

A.2.4 Beispiele

1) Meist bezeichnet man die Funktion, die konstant gleich einem Wert $a > 0$ ist, einfach wieder mit a. Dann ist $O(a)$ die Menge aller Funktionen $g\colon D \to \mathbb{R}$, die außerhalb eines beschränkten Intervalls beschränkt sind oder etwas lax formuliert bei ∞ beschränkt sind. Der Wert von $a > 0$ spielt keine Rolle, alle $O(a)$ sind gleich; daher wählt man meist $a = 1$ und schreibt $O(1)$.

2) Für $k \leq m$ ist $x^k \in O(x^m)$.
Beweis: Wegen $m - k \geq 0$ ist $\frac{1}{x^{m-k}} \leq 1$ für $x \geq 1$. Mit $c = 1$ und $x_0 = 1$ gilt also $0 \leq x^k \leq cx^m$ für $x \geq x_0$.

3) Für $k > m$ ist $x^k \notin O(x^m)$, aber $x^k \in \Omega(x^m)$.
Beweis: Denn x^{k-m} strebt monoton wachsend gegen ∞ für $x \to \infty$.

4) Für $l < k$ ist $x^l + x^k \in \Theta(x^k)$.
Beweis: Wegen $x^l \in O(x^k)$ gibt es $c > 0$ und $x_0 > 0$, so daß $x^l \leq cx^k$ für $x \geq x_0$. Mit $c_1 = 1$ und $c_2 = 1 + c$ folgt $0 \leq c_1 x^k \leq x^k + x^l \leq c_2 x^k$ für $x \geq x_0$.

4) Mit Hilfe der Regel von de l'Hospital folgt $\log x \in O(x)$, aber $x \notin O(\log x)$, und $x^k \in O(a^x)$ sowie $a^x \notin O(x^k)$ für jedes $k \in \mathbb{N}$ und jedes positive $a \in \mathbb{R}$. Daraus ergibt sich $O(\log x) \subset O(x) \subset O(x \log x) \subset O(x^2)$, wobei die Inklusionen echt sind, also $O(\log x) \neq O(x) \neq O(x \log x) \neq O(x^2)$.

5) Für $D = \mathbb{N}$ gilt $\log(n!) \in \Theta(n \log n)$
Diese Aussage braucht man, um eine untere Schranke für vergleichsbasierte Sortierverfahren zu bestimmen.
Beweis:Für $n > 1$ gilt

$$\log((n-1)!) = \log\left(\prod_{j=1}^{n-1} j\right) = \sum_{j=1}^{n-1} \log j \leq \int_1^n \log t dt = n \log n - n + 1 \leq n \log n$$

wobei die Ungleichung in der Mitte sich daraus ergibt, dass die Summe eine Riemannsche Untersumme für das Integral ist, welches die Stammfunktion $t \log t - t$ hat. Für $n > 1$ folgt

$$\log(n!) = \log(n-1)! + \log n \leq \log(n-1)! + n \leq n \log n + 1 \leq 2n \log n$$

also ist $\log(n!) \in O(n \log n)$.

Statt $\log(n!) \in \Omega(n \log n)$ zu zeigen, kann man auch $n \log n \in O(\log(n!))$ zeigen. Für $n > 1$ ergibt sich auf ähnliche Weise mit Hilfe der Obersumme statt der Untersumme

$$n \log n - n + 1 = \int_1^n \log t dt \leq \sum_{j=1}^{n} \log j = \log\left(\prod_{j=1}^{n} j\right) = \log(n!)$$

Daraus folgt $n \log n \in O(\log(n!) + n - 1) = O(\log(n!))$, weil

$$n - 1 \leq (n-1)\frac{\log 2}{\log 2} = \frac{\log 2^{n-1}}{\log 2} \leq \frac{\log(n!)}{\log 2}$$

A.2.5 Bezeichnungen h, f, f_1, f_2 seien Funktionen, die bei ∞ nichtnegativ sind. Dann setzt man

$$\begin{aligned} h + O(f) &:= \{\, h + g \colon g \in O(f) \,\} \\ O(f_1) + O(f_2) &:= \{g_1 + g_2 \colon g_1 \in O(f_1) \text{ und } g_2 \in O(f_2) \,\} \\ O(f_1) \cdot O(f_2) &:= \{\, g_1 \cdot g_2 \colon g_1 \in O(f_1) \text{ und } g_2 \in O(f_2) \,\} \end{aligned}$$

Entsprechend für die Symbole Ω und Θ und für andere Rechenoperationen als $\cdot$ und $+$.

Eine Gleichung, auf deren beiden Seiten Wachstumsklassen stehen, ist als Gleichheit von Funktionenmengen zu verstehen. Infolgedessen ist die Gleichung $x^2 + O(x) = O(x^2)$ **falsch!** Denn es gilt zwar $x^2 + O(x) \subset O(x^2)$, aber nicht $x^2 + O(x) \supset O(x^2)$, weil z.B. die Funktion $2x^2$ in $O(x^2)$ liegt, aber nicht in der Form $x^2 + h(x)$ mit $h \in O(x)$ dargestellt werden kann.

Achtung! In der Literatur findet sich auch eine andere Interpretation, nämlich daß eine Gleichung zwischen Wachstumsklassen lediglich als Inklusion der linken in der rechten Seite zu verstehen ist; dann ist also z.B. $x^2 + O(x) = O(x^2)$ als $x^2 + O(x) \subset O(x^2)$ zu interpretieren. **Dies ist ein sehr verwirrender Gebrauch des Gleichheitszeichens, denn man darf solche "Gleichungen" stets nur von links nach rechts lesen!** Obwohl diese Verwendung des Gleichheitszeichens sehr üblich ist, werden wir hier meist die klarere Schreibweise mit $\in$ und $\subset$ verwenden.