Algorithmen, Datenstrukturen und Programmierung

Vorlesungsmaterial zum internen Gebrauch (keine Veröffentlichung)

Univ.-Prof. Dr.-Ing. habil. Heiko Vogler
Lehrstuhl Grundlagen der Programmierung
Fakultät Informatik
Technische Universität Dresden

19. Auflage

25. Januar 2023

Vorwort

Das vorliegende Heft faßt den Inhalt der beiden Vorlesungen „Algorithmen und Datenstrukturen“ (V2/Ü2) und „Programmierung“ (V2/Ü2) zusammen. Diese Vorlesungen gehören zu den Pflichtvorlesungen des 1. bzw. 2. Semesters der verschiedenen Studiengänge u. a. der Informatik, Medieninformatik und Informationssystemtechnik an der TU Dresden.

Das Ziel dieser beiden Lehrveranstaltungen ist die Vermittlung verschiedener algorithmischer Konzepte und eine Einführung in die imperative Programmierung und funktionale Programmierung. Die Logik-Programmierung, die nebenläufige Programmierung und die objektorientierte Programmierung werden bewusst ausgespart, da sie Gegenstand anderer Pflicht- bzw. Wahlpflichtvorlesungen sind.

Der zweiseitigen Verankerung der Informatik – nämlich einerseits als formale Strukturwissenschaft und andererseits als anwendungsorientierte Ingenieurwissenschaft – wurde in dieser Vorlesung dadurch Rechnung getragen, dass manche Kapitel (insbesondere bei der Behandlung der Konzepte der Programmierung) einen formalen und mathematischen Charakter haben, andere Kapitel dagegen mehr einen informellen, beschreibenden Anstrich besitzen.

Einführung In Kapitel 1 werden anhand eines einfaches Beispiels die Stationen, die zwischen der Stellung eines Problems und der Erstellung eines Programms, welches das Problem löst, liegen gedanklich durchwandert. Dabei werden einige Fachbegriffe angesprochen (- und nur die wenigsten detailliert erörtert -) und in eine Beziehung zueinander gesetzt. Dieses Kapitel ist informell gehalten und soll den Studierenden zur Orientierung dienen.

Teil I: Kurze Einführung in C (Kapitel 2 bis 7) Im Teil I der Vorlesung stellen wir eine Algorithmenbeschreibungssprache vor, und zwar Pseudo-C. Diese Sprache umfasst einerseits eine an manchen Stellen aus didaktischen Gründen eingeschränkte Version der Programmiersprache C; andererseits sind in Pseudo-C aber auch umgangssprachliche Konstrukte erlaubt. Gleich am Anfang sei hier gesagt: Diese Vorlesung ist **kein** Programmierkurs in C; vielmehr sollen hier die Programmier*konzepte* vermittelt werden.

Teil II: Algorithmische Problemstellungen (Kapitel 8 bis 14) Nach einer Einführung in die wesentlichen Grundlagen der Komplexitätstheorie im Kapitel 8, werden in den Kapiteln 9 bis 13 Algorithmen und Datenstrukturen für verschiedene Problemstellungen behandelt. Im besonderen werden dies Sortier- und Suchverfahren, Hashverfahren, Algorithmen auf Bäumen, Graphen und ein Algorithmus zur Approximation bestimmter Wahrscheinlichkeitsverteilungen (EM-Algorithmus).

Orthogonal zu diesen verschiedenen Problemfeldern zeigen wir exemplarisch verschiedene Prinzipien für die Struktur von Algorithmen auf, nämlich divide-and-conquer, dynamische Programmierung und Backtracking (Kapitel 14).

Teil III: Weitere Programmierkonzepte (Kapitel 15 bis 19) Teil III beginnt mit der Besprechung der Programmiersprache Haskell (Kapitel 15). Danach behandeln wir in diesem Kapitel Konzepte der funktionalen Programmierung auf der Grundlage des λ-Kalküls. In Kapitel 16 geben wir eine kurze Einführung in die Logik-Programmierung. In Kapitel 17 werden wir zeigen, wie man imperative Programmiersprachen implementiert – das werden wir für die Fragmente C_0 und C_1 zeigen. Kapitel 18 stellt einen Kalkül vor, mit dessen Hilfe Eigenschaften von Programmen des Fragments C_0 bewiesen (verifiziert) werden können.

Schließlich betrachten wir im Kapitel 19 eine Verbindung zwischen C_0- und H_0-Programmen; das sind spezielle funktionale Programme, die sogenannten tail-rekursiven funktionalen Programme. Insbesondere werden wir H_0-Programme implementieren und die Transformation von C_0-Programmen in H_0-Programme (und umgekehrt) diskutieren. Kapitel 19 wurde von Herrn Armin Kühnemann freundlicherweise zur Verfügung gestellt.

Anhänge Im Anhang A findet man eine Sammlung der in den Kapiteln 2 bis 7 besprochenen Syntaxdiagramme, im Anhang B werden die wichtigsten mathematischen Grundlagen aus der Mengen- und Funktionenlehre zusammengestellt und das Konzept der vollständigen Induktion besprochen.

Das vorliegende Skript enthält alle in der Vorlesung auftretenden formalen Definitionen, ist aber – was die Ausführlichkeit der Erläuterungen anbelangt – manchmal knapp gehalten. Daher ersetzt das Lesen dieses Heftes auch nicht den Besuch der Vorlesung, ihr aktives *Vorbereiten und Nacharbeiten* und das aktive Arbeiten in den Übungsgruppen. Ebenso wenig ersetzt es ein Literaturstudium anhand von mindestens einem einschlägigen Lehrbuch. Inzwischen gibt es eine sehr große Anzahl von Lehrbüchern sowohl über Algorithmen und Datenstrukturen als auch über Programmierung, die wir hier nicht alle aufzählen wollen. Es seien nur einige Bücher angegeben, auf die sich diese Vorlesung teilweise stützt:

- Algorithmen und Datenstrukturen [OW02, CLR90, CLRS04]
- Programmierung [HSAF94, LNN00]
- funktionale Programmierung [Hut07]
- Logikprogrammierung [SS94]
- Verification of Sequential and Concurrent Programs [AO91, AO97]

Am Ende dieses Skriptes findet sich das Verzeichnis der Literatur, auf die im Skript verwiesen wird.

Für die Hilfe bei der Erstellung dieses Heftes danke ich besonders Matthias Büchse, Tobias Denkinger, Toni Dietze, Kilian Gebhardt, Mathias Hinkel, Hans-Jakob Holtz, Hannes Kluckhuhn, Martin Krebs, Armin Kühnemann, Johannes Osterholzer, Thomas Ruprecht, Lutz Rüdiger, Jana Schubert, Denis Stein, Torsten Stüber, Rainer Vater und Janis Voigtländer.

Dresden, 25. Januar 2023 Heiko Vogler

Inhaltsverzeichnis

1 Vom Problem zum Programm – Ein Überblick

Die Idee zu diesem Kapitel entstammt [Sch93]. Von der eventuell umgangssprachlichen Formulierung eines Problems bis zum Erstellen eines Programms, welches das Problem löst, werden verschiedene Phasen durchlaufen; Abbildung 1.1 zeigt diese Phasen.

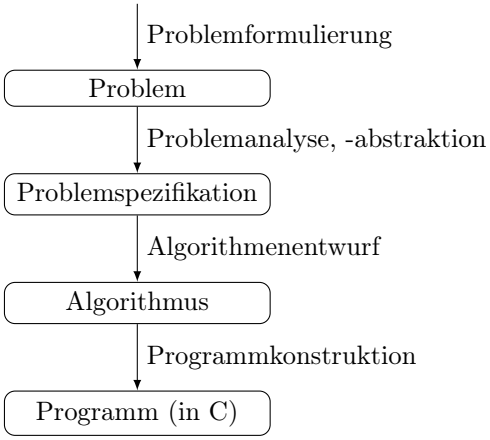


Abbildung 1.1: Vom Problem zur Berechnung der Lösung

1.1 Ein einfaches Beispiel

Anhand eines einfachen Beispiels wollen wir den Ablauf „Vom Problem zur Berechnung einer Lösung“ durchspielen.

1.1.1 Problemformulierung

Wir wollen uns folgendes Problem stellen:

> ***Problem***
> Suche die jüngste Person im Raum.

1.1.2 Problemanalyse, -abstraktion

Bevor wir an die Berechnung einer Lösung denken können, müssen wir zunächst das Problem genauer betrachten und ggf. Ungenauigkeiten ausräumen.

- Gibt es überhaupt eine Lösung?
- Wenn es eine Lösung gibt, gibt es eine eindeutige Lösung?
- Wie soll eine Lösung aussehen?
- Wie sollen die Personen bei der Berechnung einer Lösung repräsentiert werden?
- Wie genau soll das Alter einer Person gezählt werden?

- Was passiert, wenn zwei Personen gleich alt sind?

Wir sehen, dass unsere ursprüngliche Problemformulierung noch recht ungenau ist. Um nun von einer konkreten Situation (728 Studierende sitzen im Hörsaal) abzusehen und auch die Ungenauigkeiten zu beheben, müssen wir verschiedene Abstraktionen durchführen.

Abstraktion

1. Jede Person wird durch eine natürliche Zahl i repräsentiert.
2. Als Alter rechnen wir nur das Alter in Jahren; jede Person i hat also ein Alter a_i (positive, ganze Zahl).
3. Die Zahlen $a_1, a_2, \ldots, a_n$ sind bekannt.
4. „Jüngste“ Person ist eine Person i, falls für jede Person j gilt: $a_i \leq a_j$.
5. Wenn für zwei Personen i und j gilt: $a_i = a_j$, dann wollen wir als Lösung die bzgl. der Folge $a_1, a_2, \ldots, a_n$ erste Person mit Alter a_i als Lösung angeben.

Beschreibt man nun in einer hinreichend formalisierten Sprache die funktionale Beziehung zwischen den Eingabegrößen des Problems und Lösungen des Problems (bei diesen Eingaben), so erhält man ein abstraktes Objekt, welches in unserem Phasenmodell die Grundlage des Algorithmenentwurfs darstellt. Diese Stufe im Softwareentwurfsprozess wird auch als Problemspezifikation bezeichnet.

Problemspezifikation

Gegeben Eine Folge $a_1, a_2, \ldots, a_n$ von ganzen, positiven Zahlen.

Gesucht Der kleinste Positionsindex j mit $a_j = \min\{a_1, \ldots, a_n\}$.

1.1.3 Algorithmenentwurf

Jetzt muss also aus der Problemspezifikation ein Algorithmus zur Berechnung von Ausgaben für vorgelegte Eingaben konstruiert werden.

Ein *Algorithmus* ist eine Vorschrift zur Lösung eines Problems.

Ein Algorithmus hat folgende Eigenschaften:

- muss so präzise formuliert sein, dass er im Prinzip maschinell ausgeführt werden kann,
- ist ein *abstraktes* Objekt,
- ist unabhängig von der Programmiersprache, in der er geschrieben werden soll,
- ist unabhängig vom Computertyp oder der verwendeten Rechnertechnologie,
- ist durch einen endlichen Text beschrieben (*Finitheit*),
- läuft in einzelnen, wohldefinierten Schritten ab (*Effektivität*),
- nach Ausführung jedes Schrittes ist eindeutig festgelegt, welcher Schritt als nächster ausgeführt wird (*Determiniertheit*) und
- kommt bei jeder Eingabe in endlich vielen Schritten zu einem Ende (*Terminiertheit*).

Beachte: Bei manchen Problemstellungen ist es auch angebracht, die eine oder andere Eigenschaft nicht zu fordern (z. B. ist es im Rahmen von Betriebssystemen nicht erwünscht, dass der Algorithmus zur Jobverwaltung irgendwann terminiert).

Erwünschte Eigenschaften eines Algorithmus: korrekt, kurzer Text, schnell, allgemeinverständlich, übersichtlich, leicht veränderbar, vollständig.

Oft widersprechen sich diese erwünschten Eigenschaften; es kommt zu folgenden „trade-offs“:

$$\begin{aligned} \text{kurzer Text} &\longleftrightarrow \text{allgemein verständlich, übersichtlich} \\ \text{schnell} &\longleftrightarrow \text{leicht veränderbar} \end{aligned}$$

Algorithmus 1 löst unser Problem.

Algorithmus 1 MinAlter

Eingabe Eine Folge $a_1, \ldots, a_n$ von positiven, ganzen Zahlen.

Ausgabe der kleinste Positionsindex j mit $a_j = \min\{a_1, \ldots, a_n\}$.

Verfahren Zusätzliche Variablen: x (für das Alter), i (als Zählvariable);

1. (*Initialisierung*) Setze $j := 1, x := a_j$ und $i := 2$.
2. (*Suchlauf*)
   Solange $i \leq n$ gilt, wiederhole:
   falls $a_i < x$, setze $j := i$ und $x := a_j$
   erhöhe i um 1
3. Ausgabe von j als Ergebnis

1.1.4 Programmkonstruktion

Jetzt muss der Algorithmus in eine Programmiersprache übersetzt werden, die der Rechner versteht. Die folgende Tabelle zeigt dabei einige der anstehenden Fragen (linke Spalte) und unsere Antworten (rechte Spalte); natürlich sind auch andere Antworten denkbar.

anstehende Frage	unsere Antwort
Verfügbare Programmiersprachen?	*C*
Wie kommen Daten in den Rechner?	Altersangaben müssen eingegeben werden.
Durch welche Datenstrukturen werden Daten repräsentiert?	Die Folge $a_1, \ldots, a_n$ der Zahlen wird in einem Array (Feld) gespeichert.
Wie wird das Ergebnis ausgegeben?	Ausgabe als verständlicher Text.
Wie wird das Programm gegliedert?	Die Operation „Finde Person mit kleinstem Alter“ wird als Funktion beschrieben.

Das *C*-Programm `JuengstePerson`, welches auf dem Algorithmus MinAlter basiert, ist am Ende dieses Kapitels zu finden.

1.2 Geschichte des Begriffes „Algorithmus“

Der Begriff des Algorithmus ist einer der zentralen, wenn nicht sogar der wichtigste in der Informatik. Wir haben deshalb eine kurze Liste mit historischen Begebenheiten aufgestellt, welche die Entwicklung des Algorithmenbegriffs dokumentieren. Umfangreichere Ausführungen findet man in [MHR80, Wex81].

- ca. 300 v. Chr. Euklids „Elemente“ (Sammlung von Algorithmen)
- ca. 800 n. Chr. Mohamed ibn Musa abu Djafa al Khowarizmi (auch al Khwarizimi, al Choremi u. a.), aus seinem Namen entstand der Begriff „Algorithmus“
- ca. 1200 Raimundus Lullus (Buch „Ars Magna“): hatte Idee zu einer rechnerischen Durchführung aller Beweise, insbesondere Gottesbeweise zur Bekehrung der Heiden (geb. auf Mallorca)
- ca. 1700 G. W. Leibniz (1646–1716): Vision einer kombinatorischen Methode zur Lösung mathematischer und philosophischer Probleme
- J. Jacquard (1752–1834): konstruierte Webstuhl, der, durch Lochkarten gesteuert, verschiedene Muster herstellen konnte.
- H. Hollerith (1860–1929): entwarf eine Maschine, die auf Steckbrettern programmiert wurde (Zweck: amerikanische Volkszählung)

- 1900 Hilberts 10. Problem: Man soll einen Algorithmus angeben, der zu jedem Polynom über ganzen Zahlen entscheidet, ob es eine Nullstelle in den ganzen Zahlen besitzt oder nicht.
- 1931 K. Gödel: Beweissystem für die Zahlentheorie $TH(Nat, *, +)$ ist unvollständig, d. h., es gibt keinen Algorithmus, der bei Eingabe einer beliebigen Formel f aus $TH(Nat, *, +)$ entscheidet, ob f wahr oder falsch ist.
- 1936 Church, Turing, Kleene, Gödel, Herbrand: Präzisierung des Algorithmusbegriffs, Churchsche These.
- 1941 K. Zuse: Entwickelt den elektro-mechanischen Rechner Z3 (Zweck: Mechanisierung von aufwendigen Kalkulationen im Bereich des Vermessungswesens)
- ca. 1940 J. von Neumann: Programme sind auch Daten, die von anderen Programmen manipuliert werden können; Konzept des Computers, wie es auch heute noch verwendet wird.
- 1970 Matjasevic: Hilberts 10. Problem ist nicht algorithmisch lösbar.

```
1  /* JuengstePerson */
2
3  #include <stdio.h>
4
5  #define MAX 100         /* maximale Personenzahl */
6  int a[MAX];
7  int i, j, n;
8
9  int MinIndex(int a[], int n)
10 {/*liefert den Minimalindex in dem Feld a[0]..a[n-1] */
11   int i, j, x;
12   j = 0; x = a[j]; i = 1;
13   while (i < n)
14   { if  (a[i] < x)
15     { j = i;
16       x = a[j];
17     }
18     i = i+1;
19   }
20   return j;
21 }
22
23 int main()
24 { printf("Bitte Anzahl der Personen eingeben: ");
25   do scanf("%d",&n);
26   while ((n < 1) || (n > MAX));
27
28   i = 0;
29   while (i < n)
30   { printf("Bitte Alter der %d. Person eingeben: ", i+1);
31     scanf("%d", &a[i]);
32     i = i+1;
33   }
34   j = MinIndex(a, n);
35   printf("\n\n");
36   printf("Juengste Person ist Nr. %d\n", j+1);
37
38   return 0;
39 }
```

Programm

Teil I

Kurze Einführung in C

Um Algorithmen formulieren und sie danach analysieren zu können, benötigen wir eine Metasprache, in der wir Algorithmen aufschreiben, beschreiben oder spezifizieren können. Deshalb führen wir in diesem Kapitel eine *Algorithmenbeschreibungssprache* ein. Ein Element (oder: Satz) dieser Sprache ist dann ein Algorithmus.

Da wir im Laufe des Vorlesungspaares „Algorithmen und Datenstrukturen“ und „Programmierung“ ebenfalls die konkrete imperative Programmiersprache C einführen werden, haben wir uns für Pseudo-C als Algorithmenbeschreibungssprache entschieden. Das ist im wesentlichen die Programmiersprache C, bei der wir aber auch umgangssprachliche Konstrukte erlauben wollen; diese Konstrukte sind dann überall erlaubt wo *Statements* erlaubt sind.

Es gibt auch ganz anders geartete, von Programmiersprachen unabhängige Algorithmenbeschreibungssprachen, beispielsweise Struktogramme oder Ablauf- oder Flussdiagramme. Diese wollen wir hier jedoch nicht behandeln.

In diesem Teil geben wir eine kurze Einführung in die Sprache C. Wenn man eine Programmiersprache beschreiben will, dann muss man deren Syntax und deren Semantik festlegen. Für die Beschreibung der Syntax von C benutzen wir sogenannte Syntaxdiagramme als Hilfsmittel. Diese eignen sich natürlich auch zur Beschreibung der Syntax einer beliebigen anderen Programmiersprache. Neben dem Konzept der Syntaxdiagramme stellen wir alternativ ebenfalls die sogenannte EBNF vor.

Wir stellen eine Variante von C vor, die an den *ANSI-C*-Standard angelehnt ist, geben aus didaktischen Gründen jedoch eine vereinfachte Syntax wieder, die eine echte Teilmenge von *ANSI-C* beschreibt. Die vollständige Syntax ist in Anhang A aufgelistet.

Die Angabe der Semantik einer Programmiersprache ist ungleich schwieriger und wird deshalb oft vernachlässigt. Wir wählen hier einen Kompromiss und beschreiben im Kapitel 17 die Semantik einer Teilsprache (subset) C_0 von C.

2 Syntax von Programmiersprachen

Jede Sprache, ob natürliche Sprache oder Programmiersprache, hat eine Syntax, die den Aufbau und die Gliederung der Sätze, die zur Sprache gehören sollen, festlegt (vgl. Tabelle 2.1).

Metasprache (Syntax-Beschreibungssprache)	Objektsprache	Element der Objektsprache
Grammatik der deutschen Sprache	natürliche Sprache (z. B. Deutsch)	$\ni$ Satz (z. B. Der Hund läuft schnell.)
- EBNF - Syntaxdiagramme	Programmiersprache (z. B. *C*)	$\ni$ Programm (z. B. `/* JuengstePerson */ ...`)
- Grammatik Typ 2 - Kellerautomaten	formale Sprache (z. B. $L = \{a^n b \mid n \geq 0\}$	$\ni$ Wort (z. B. $w = aaab$)

Tabelle 2.1: Sprachenübersicht

Der Wunsch nach eindeutiger Semantikgebung (zumindest bei Programmiersprachen) erzwingt die formale Definition der Syntax einer Sprache.

Geschichtliches

- Noam Chomsky, amerikanischer Linguist, versuchte die Menge aller syntaktisch korrekten, englischen Sätze mit Hilfe eines Regelsystems zu beschreiben. Der Versuch scheiterte, führte aber zu einer sehr fruchtbaren Forschungsrichtung, insbesondere zur Entwicklung von Grammatiken, welche die Syntax *formaler* Sprachen beschreiben (Chomsky-Grammatiken, Typ-0, Typ-1, Typ-2, Typ-3; siehe auch Vorlesungen *Formale Systeme* und *Theoretische Informatik und Logik*).
- Die Syntax der zu den ältesten Programmiersprachen gehörenden Sprachen FORTRAN und COBOL wurde nicht formal festgelegt; vielmehr wurde sie informell durch Angabe von Beispielen und Gegenbeispielen angegeben.
- John Backus entwickelte im Jahr 1960 eine formale Beschreibung der Syntax der Programmiersprache ALGOL 60; Beschreibungsmittel: Backus-Naur-Form (BNF) (BNF ist äquivalent zur Typ-2-Chomsky-Grammatik).

Zunächst muss hier ein Problem angesprochen werden: Die Definition einer formalen Sprache muss selbst mit formal*sprachlichen* Mitteln erfolgen. Wir unterscheiden deshalb zwischen:

- *Objektsprache* (Sprache, die syntaktisch definiert werden soll).
- *Metasprache* (Sprache, mit deren Hilfe die Objektsprache beschrieben werden soll).

Ebenso ist noch vorab eine Erklärung zum Begriff „Wort“ notwendig. Betrachten wir dazu einen Satz der deutschen Sprache (das ist dann im obigen Sinne die Objektsprache).

Beispiel 2.1.

$$\underbrace{\text{Das } \underbrace{\text{ist}}_{\text{Wort}} \text{ eine T } \underbrace{\text{a}}_{\text{Buchst.}} \underbrace{\text{fel}}_{\text{Silbe}}.}_{\text{Satz}}$$

□

In dem Satz haben wir exemplarisch jeweils einen Buchstaben, eine Silbe und ein Wort gekennzeichnet. Im Gegensatz zur Bedeutung des Begriffs „Wort“ in natürlichen Sprachen steht die Bedeutung dieses Begriffs in Programmiersprachen. Hier ist ein Wort eine *beliebige* endliche Sequenz von aufeinander folgenden Symbolen; ein Symbol kann Buchstabe, Ziffer oder Interpunktionszeichen sein. Betrachten wir dazu ein Element einer (funktionalen) Programmiersprache.

Beispiel 2.2.

```
                    Wort                  Wort
module SchaltJ ( schaltjahr ) where
       Symbol       Symbol
```

□

In der Tat sind Programmiersprachen Instanzen eines ganz allgemeinen Konzeptes, und zwar dem Konzeptes der formalen Sprachen. Wir geben nun einige Definitionen zu diesem Konzept an.

- Ein *Alphabet* ist eine nicht leere, endliche Menge; die Elemente dieser Menge nennen wir *Symbole*.
- Sei Σ ein Alphabet; ein *Wort* (eine Zeichenreihe) über Σ ist eine endliche Folge von Symbolen aus Σ; die Folge kann eine beliebige Länge $k \geq 0$ haben.
- Das *leere Wort*, bezeichnet durch ε, ist das Wort der Länge null.
- Die *Menge aller Wörter über* Σ wird mit Σ^* bezeichnet.
- Die zweistellige Operation der Zeichenreihenverkettung, auch *Konkatenation* genannt, ist die Abbildung $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$. Für jedes $u, v \in \Sigma^*$ definiere

  $$u \cdot v = u_1 u_2 \dots u_m v_1 v_2 \dots v_n,$$

  falls es $m, n \geq 0$ gibt und es für jedes $1 \leq i \leq m$ ein Symbol $u_i \in \Sigma$ gibt und es für jedes $1 \leq j \leq n$ ein Symbol $v_j \in \Sigma$ gibt, so dass $u = u_1 u_2 \dots u_m$ und $v = v_1 v_2 \dots v_n$

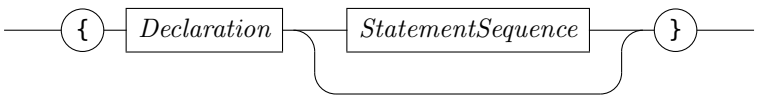
  Der Punkt in „$u \cdot v$“ wird auch oft weggelassen.
- Die *n-te Potenz* eines Wortes $w \in \Sigma^*$, bezeichnet durch w^n, ist induktiv definiert durch $w^0 = \varepsilon$ und $w^{n+1} = w^n \cdot w$ für jedes $n \geq 0$.
- Eine *(formale) Sprache L (über* Σ*)* ist eine Menge von Wörtern über Σ, d. h. $L \in \mathcal{P}(\Sigma^*)$.
- Seien L_1, L_2 Sprachen über Σ: Die *Konkatenation* (oder: das *Komplexprodukt*) *von* L_1 *und* L_2, bezeichnet durch $L_1 \cdot L_2$, ist die Sprache $\{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. Beachte: $\emptyset \cdot L = \emptyset = L \cdot \emptyset$ und $L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$.
- Sei L eine Sprache. Der *Stern von* L, bezeichnet durch L^*, ist die Sprache $\bigcup_{n \geq 0} L^n$, wobei $L^0 = \{\varepsilon\}$, $L^{n+1} = L^n \cdot L$ für jedes $n \geq 0$. Beachte: $\emptyset^* = \{\varepsilon\}$.

Programmiersprachen sind spezielle formale Sprachen.

2.1 Syntaxdiagramme

Eine Möglichkeit zur Beschreibung der Syntax bieten die Syntaxdiagramme. Als einführendes Beispiel zeigen wir das Syntaxdiagramm für *Block* der Programmiersprache *C*.

Block



2.1.1 Aufbau

Ein Syntaxdiagramm besteht aus:

- Ovalen
- Kästchen
- Verbindungen, die aus folgenden Bestandteilen zusammengesetzt sind:
  - Linien (evtl. gebogen)

– Verzweigungen und Zusammenfassungen.

Es gelten die folgenden sieben Regeln für den Aufbau von Syntaxdiagrammen:

1. Jedes Syntaxdiagramm hat einen eindeutigen Namen; der Name ist eine syntaktische Variable.
2. Jedes Kästchen ist mit dem Namen eines Syntaxdiagrammes beschriftet.
3. Jedes Oval ist mit einem Terminalsymbol beschriftet.
4. An jedem Kästchen und an jedem Oval enden genau zwei Linien.
5. Es gibt genau einen Strich, der als Anfang markiert ist.
6. Es gibt genau einen Strich, der als Ende markiert ist.
7. Linien dürfen sich nicht kreuzen.

Die Menge aller Syntaxdiagramme über einer Menge V von syntaktischen Variablen und einer Menge Σ von Terminalsymbolen bezeichnen wir durch SynDia(Σ, V).

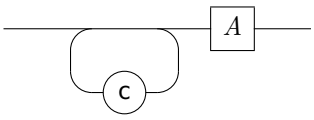
Ein *legaler Weg* durch ein Syntaxdiagramm beginnt am Eingang und folgt in stetiger Bewegung den Linien, wobei Kästchen und Ovale einfach durchquert werden. An Verzweigungen darf eine beliebige Linie gewählt werden, wobei die natürliche Richtung der Verzweigung zu beachten ist.

2.1.2 Algorithmus zur Berechnung der erzeugten Sprache

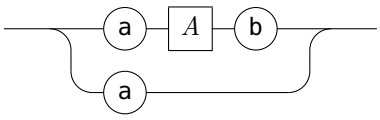
Welche Sprache wird nun durch eine endliche Menge $\mathcal{U}$ von Syntaxdiagrammen beschrieben?

Beispiel 2.3. Es enthalte $\mathcal{U}$ die folgenden Syntaxdiagramme mit S als Startdiagramm:

S



A



□

Dazu konstruieren wir einen Algorithmus, den wir *Rücksprungalgorithmus* (siehe Algorithmus 2) nennen wollen. Er nimmt die Menge $\mathcal{U}$ der Syntaxdiagramme als Eingabe, wobei ein Syntaxdiagramm als das *erste* ausgezeichnet sein muss (Startdiagramm). Bevor der Algorithmus abläuft, wird der Ausgang jedes Kästchens mit einer Marke versehen; wir nennen sie die *Rücksprungadresse* dieses Kästchens. Keine zwei verschiedenen Kästchen dürfen dieselbe Marke haben. Als Hilfsspeicher benutzt der Algorithmus einen Keller. Das ist ein Speicher, der zu jedem Zeitpunkt der Rechnung ein Wort über einem Alphabet enthält. Ein Ende des Worts wird als *Kellerspitze* bezeichnet; hier nehmen wir das linke Ende des Worts. Nun kann der Algorithmus auf der Kellerspitze nachschauen, welches Symbol dort steht und dieses Symbol durch ein anderes Symbol ersetzen. Der Algorithmus kann auch ohne nachzuschauen ein Symbol oben auf den Keller legen oder das oberste Symbol wegnehmen. Als Kelleralphabet wählen wir hier die (endliche) Menge der Rücksprungadressen.

Die von der Menge $\mathcal{U}$ durch den Algorithmus erzeugte Objektsprache ist die Menge aller Wörter w, für die der Erzeugungsprozess des Rücksprungalgorithmus erfolgreich endet.

Bemerkung: In der Tat haben wir hier einen Zusammenhang benutzt, der in der Vorlesung „Formale Systeme“ explizit aufgegriffen wird: die Äquivalenz von kontextfreien Grammatiken und Kellerautomaten. Dabei muss man sich ein System von Syntaxdiagrammen als kontextfreie Grammatik vorstellen und den Rücksprungalgorithmus als Programm des Kellerautomaten. ... Sie werden sehen!

2.2 Extended Backus-Naur-Form (EBNF)

Eine andere Möglichkeit, die Syntax von Programmiersprachen zu definieren, ist durch die sogenannte Extended Backus-Naur-Form (kurz: EBNF) gegeben.

Algorithmus 2 Rücksprungalgorithmus

1. Beginne am Eingang des ersten Syntaxdiagramms von $\mathcal{U}$ (Startdiagramm).
2. Folge den Linien auf einem legalen Weg.
   - Falls dabei der Ausgang erreicht wird, gehe nach Punkt 5.
   - Falls ein Kästchen bzw. Oval erreicht wird, gehe nach Punkt 3.
3. - Falls es sich um ein Oval handelt, notiere das darin enthaltene Terminalzeichen und gehe anschließend zu Punkt 2 zurück.
   - Andernfalls gehe nach Punkt 4.
4. - Falls es sich um ein Kästchen handelt, dann
     – lege eine Kopie der Rücksprungadresse dieses Kästchens oben auf den Keller,
     – suche den Eingang des Diagramms in $\mathcal{U}$ auf, welches den Namen trägt, der in dem erreichten Kästchen steht
     – und arbeite an dem Eingang des neuen Diagramms ab Punkt 2 weiter.
5. - Wenn noch eine Rücksprungadresse *adr* auf dem Keller liegt, dann
     – gehe zur Stelle, die mit *adr* gekennzeichnet ist und
     – nehme *adr* vom Keller und setze die Bearbeitung an dieser Stelle am Punkt 2 fort.
   - Wenn keine Rücksprungadresse auf dem Keller liegt und man sich am Ausgang des Startdiagrammes befindet, dann endet der Erzeugungsprozess hier erfolgreich.

2.2.1 Beispiel einer EBNF-Regel

Jede EBNF-Definition besteht im wesentlichen aus einer endlichen Menge von EBNF-Regeln, welche die Strukturen und Teilstrukturen der Sätze der zu definierenden Sprache festlegen.

Beispiel 2.4. Ein wesentliches Merkmal eines C-Programms sind die Deklarationen von Konstanten und Variablen. Die EBNF-Regel für die Menge aller syntaktisch zulässigen Deklarationen lautet:

$\langle \mathit{Declaration} \rangle ::= \hat{[}\langle \mathit{ConstDeclaration} \rangle\hat{]}\hat{[}\langle \mathit{VarDeclaration} \rangle\hat{]}.$ □

Informell bedeuten die in der EBNF auftretenden Objekte folgendes:

- $\langle \mathit{Declaration} \rangle$, $\langle \mathit{ConstDeclaration} \rangle$, $\langle \mathit{VarDeclaration} \rangle$ heißen *syntaktische Variablen*, sie bezeichnen Mengen von syntaktisch zulässigen Zeichenreihen. Insbesondere bei der Definition von Programmiersprachen werden wir künftig diese Variablen durch spitze Klammern kennzeichnen. $\langle \mathit{Declaration} \rangle$ bezeichnet z. B. die Menge aller syntaktisch zulässigen Deklarationen.
- $\hat{\{} \ldots \hat{\}}$ Wiederholungsklammern; was zwischen den Klammern steht, kann beliebig oft hintereinander gestellt werden. Insbesondere ist darin der Fall enthalten, dass das, was zwischen den Klammern steht, keinmal auftritt. Da die geschweifte Klammer gleichzeitig auch als Terminalsymbol der Sprache C bzw. C_0 auftreten kann, sollen zur besseren Lesbarkeit und ggf. zur Herstellung der Eindeutigkeit von Syntaxdefinitionen künftig die Metazeichen mit einem Dach über dem jeweiligen Symbol gekennzeichnet werden.
- $\hat{[} \ldots \hat{]}$ Optionsklammern; was zwischen den Klammern steht, kann auftreten oder kann weggelassen werden. Man beachte, dass auch hier das Metasymbol (der Einheitlichkeit wegen) mit einem Dach gekennzeichnet ist.
- Wörter ohne spitze Klammern sind Terminalsymbole.
- Außerdem kann in EBNF-Regeln der senkrechte Strich $\hat{|}$ auftreten; hier kann entweder das links vom Strich Stehende auftreten oder das rechts vom Strich Stehende.

Die intuitive Bedeutung dieser EBNF-Regel lautet also: Eine Deklaration beginnt mit einer oder keiner Konstantendeklaration, gefolgt von einer oder keiner Variablendeklaration. Die leere Deklarationsmenge ist somit als Spezialfall enthalten.

2.2.2 EBNF-Definition

Wie gesagt besteht jede EBNF-Definition aus einer endlichen Menge von EBNF-Regeln. Jede EBNF-Regel besitzt eine linke Seite und eine rechte Seite. Die linke Seite besteht aus einer syntaktischen Variablen; die rechte Seite ist ein EBNF-Term.

EBNF-Terme sind aus den syntaktischen Variablen, den Terminalsymbolen und den Metazeichen nach bestimmten Regeln aufgebaut.

Definition 2.5. Sei V eine endliche Menge von syntaktischen Variablen, und sei Σ eine endliche Menge von Terminalsymbolen mit $V \cap \Sigma = \emptyset$. Die *Menge der EBNF-Terme über V und Σ*, bezeichnet durch $T(\Sigma, V)$, ist die kleinste Menge $T \subseteq \left(V \cup \Sigma \cup \left\{\hat{\{}, \hat{\}}, \hat{[}, \hat{]}, \hat{(}, \hat{)}, \hat{|}\right\}\right)^*$, so dass folgende Eigenschaften gelten:

1. $V \subseteq T$.
2. $\Sigma \subseteq T$.
3. Wenn $\alpha \in T$, so auch $\hat{(}\alpha\hat{)} \in T$, $\hat{\{}\alpha\hat{\}} \in T$, $\hat{[}\alpha\hat{]} \in T$.
4. Wenn $\alpha_1, \alpha_2 \in T$, so auch $\hat{(}\alpha_1\hat{|}\alpha_2\hat{)} \in T$, $\alpha_1\alpha_2 \in T$. □

Man beachte: Die Zuordnung eines Symbols zu V bzw. Σ ergibt sich ausschließlich aus den jeweiligen Mengendefinitionen. Da $V \cap \Sigma = \emptyset$ gilt, ist diese Zuordnung immer eindeutig möglich.

Beispiel 2.6. Betrachten wir jetzt die Zeichenreihe, die die Menge aller syntaktisch zulässigen Blöcke in C beschreibt: $\{\langle Declaration\rangle\hat{[}\langle StatementSequence\rangle\hat{]}\}$. In der Tat ist diese Zeichenreihe ein EBNF-Term, wie die folgende Zerlegung zeigt:

$$\underbrace{\underbrace{\underbrace{\{}_{\in\Sigma}\ \underbrace{\langle Declaration\rangle}_{\in V}}_{\in T}\ \underbrace{\underbrace{\hat{[}\ \underbrace{\langle StatementSequence\rangle}_{\in V}}_{\in T}\ \underbrace{\hat{]}}_{\in\Sigma}}_{\in T}}_{\in T}\ \underbrace{\}}_{\in\Sigma}}_{\in T}$$

□

Definition 2.7. Eine *EBNF-Definition* ist ein Tupel $\mathcal{E} = (V, \Sigma, S, R)$, wobei

- V endliche Menge (syntaktische Variablen)
- Σ endliche Menge (Terminalsymbole)
- $S \in V$ (Startsymbol)
- R endliche Menge von EBNF-Regeln der Form $v ::= \alpha$ mit $v \in V$ und $\alpha \in T(\Sigma, V)$. Weiterhin gilt, dass für jede syntaktische Variable v genau eine EBNF-Regel mit v als linker Seite in R enthalten ist. □

Beispiel 2.8. Sei $\mathcal{E} = (V, \Sigma, S, R)$ eine EBNF-Definition mit $V = \{S, A\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und

$$\begin{aligned} R\colon\quad & S ::= \hat{\{}\mathtt{c}\hat{\}}A \\ & A ::= \hat{(}\hat{(}\mathtt{a}A\mathtt{b}\hat{)}\ \hat{|}\ \mathtt{a}\hat{)} \end{aligned}$$

Die EBNF-Definition $\mathcal{E}$ enthält also die zwei syntaktischen Variablen S und A und die Terminalsymbole `a`, `b` und `c`. Des weiteren enthält sie zwei EBNF-Regeln; z. B. ist die rechte Seite der zweiten Regel der EBNF-Term $\hat{(}\hat{(}\mathtt{a}A\mathtt{b}\hat{)}\ \hat{|}\ \mathtt{a}\hat{)}$. Man beachte, dass der Alternativstrich geringere Priorität als die Konkatenation hat. Es ist deshalb üblich, die den Vorrang kennzeichnenden runden Klammern wegzulassen. □

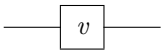
2.2.3 Übersetzung von EBNF-Definitionen in Syntaxdiagramme

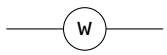
Wir kennen nun zwei verschiedene Syntaxbeschreibungssprachen: Syntaxdiagramme und EBNF. Jetzt wollen wir eine schematische Übersetzung einer EBNF in ein System von Syntaxdiagrammen angeben. Schematisch ist diese Übersetzung in dem Sinne, dass man sie auf jede EBNF-Definition anwenden kann. (Übrigens gibt es auch eine schematische Übersetzung in die andere Richtung, auf die wir aber nicht eingehen wollen.) Die Übersetzung soll so sein, dass die Sprachen, welche von der EBNF-Definition definiert wird (siehe Abschnitt 2.2.4), die gleiche ist wie die, die durch das Syntaxdiagramm erzeugt wird.

Definition 2.9. Sei $\mathcal{E} = (V, \Sigma, S, R)$ eine EBNF-Definition. Sei $v \in V$ und sei $v ::= \alpha$ die EBNF-Regel, bei der v auf der linken Seite steht.

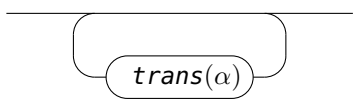
Dann übersetze den EBNF-Term α in ein Syntaxdiagramm aus SynDia(Σ, V). Diesem Syntaxdiagramm ordnen wir den Namen v zu. □

Die *Übersetzung trans*: $T(\Sigma, V) \to$ SynDia(Σ, V) ist induktiv über den Aufbau von EBNF-Termen definiert.[1]

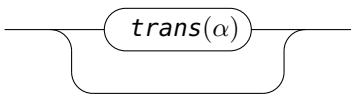
1. Sei $v \in V$; $trans(v) =$ 

2. Sei $\mathtt{w} \in \Sigma$; $trans(\mathtt{w}) =$ 

3. Sei $\alpha \in T(\Sigma, V)$;

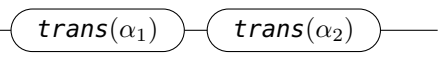
   - $trans(\ \hat{\{}\alpha\hat{\}}\ ) =$ 

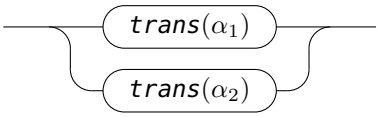
     Beachte: Der Eingang zum Syntaxdiagramm $trans(\alpha)$ ist rechts und der Ausgang links.

   - $trans(\ \hat{[}\alpha\hat{]}\ ) =$ 

   - $trans(\ \hat{(}\alpha\hat{)}\ ) = trans(\alpha)$

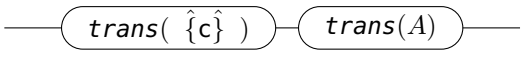
4. Sei $\alpha_1, \alpha_2 \in T(\Sigma, V)$;

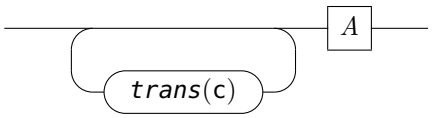
   - $trans(\alpha_1\alpha_2) =$ 

   - $trans(\ \hat{(}\alpha_1\ \hat{|}\ \alpha_2\hat{)}\ ) =$ 

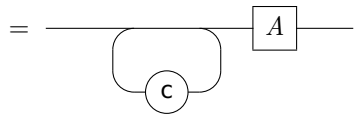
Beispiel 2.10. Gegeben sei die EBNF-Definition $\mathcal{E} = (V, \Sigma, S, R)$, wobei $V = \{S, A\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und R enthalte die Regeln $S ::= \hat{\{}\mathtt{c}\hat{\}}A$ sowie $A ::= \hat{(}\mathtt{a}A\mathtt{b}\ \hat{|}\mathtt{a}\hat{)}$.

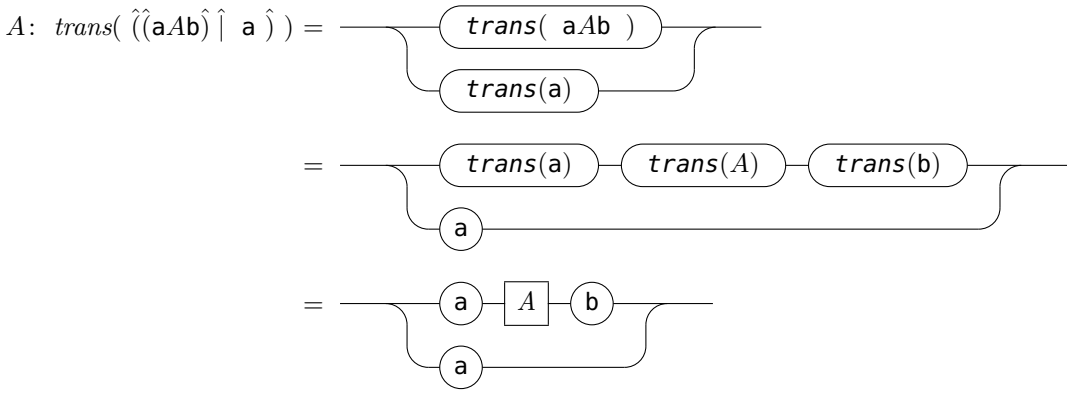
Diese EBNF wollen wir in ein System von Syntaxdiagrammen übersetzen. Zunächst betrachten wir S

S: $trans(\hat{\{}\mathtt{c}\hat{\}}A) =$ 

$=$ 

[1] Beachte, dass die Ovale um $trans(\alpha)$ im Folgenden **nicht** als Zeichen für Terminalsymbole zu verstehen sind, sondern als komplette Syntaxdiagramme, die einzufügen sind indem die entsprechende Regel von *trans* ausgeführt wird.





□

2.2.4 Bedeutung einer EBNF-Definition

Nun erhebt sich die Frage, welche Objektsprache durch eine EBNF-Definition $\mathcal{E}$ beschrieben wird. Wir werden dies hier nur informell angeben.

Wir verbinden mit der EBNF-Definition $\mathcal{E} = (V, \Sigma, S, R)$ die Vorstellung, dass zu jeder syntaktischen Variablen $v \in V$ eine Objektsprache $W(\mathcal{E}, v) \subseteq \Sigma^*$ gehört. $W(\mathcal{E}, v)$ heißt auch *syntaktische Kategorie* von v bezüglich $\mathcal{E}$.

Die rechte Seite α der EBNF-Regel $v ::= \alpha$ beschreibt nun, wie die Wörter in $W(\mathcal{E}, v)$ aussehen können. Um das genauer beschreiben zu können, ordnen wir auch jedem EBNF-Term α eine Objektsprache $[\![\alpha]\!]$ zu, unter der Annahme, dass jedem v bereits eine Sprache zugeordnet wurde. Wir nennen $[\![\alpha]\!]$ auch *Semantik von* α. Genauer gesagt definieren wir eine Funktion

$$[\![.]\!] \colon T(\Sigma, V) \longrightarrow \big((V \to \mathcal{P}(\Sigma^*)) \to \mathcal{P}(\Sigma^*)\big)$$

induktiv über den Aufbau ihres Argumentes. (Für zwei beliebige Mengen A und B bedeutet die Schreibweise $(A \longrightarrow B)$ die Menge aller Funktionen von A nach B.) Statt $[\![.]\!](\alpha)$ schreiben wir $[\![\alpha]\!]$.
Sei also $\alpha \in T(\Sigma, V)$ und $\rho \colon V \to \mathcal{P}(\Sigma^*)$ eine beliebige Funktion, die jedem $v \in V$ eine formale Sprache über Σ zuordnet. Dann definiere $[\![\alpha]\!](\rho)$ wie folgt:

- Wenn $\alpha = v \in V$, dann gilt $[\![\alpha]\!](\rho) = \rho(v)$.
- Wenn $\alpha \in \Sigma$, dann gilt $[\![\alpha]\!](\rho) = \{\alpha\}$.
  Beachte: „{“ und „}“ sind hier übliche Mengenklammern.
- Wenn $\alpha = \hat{(}\alpha_1\hat{)}$, dann gilt $[\![\alpha]\!](\rho) = [\![\alpha_1]\!](\rho)$.
- Wenn $\alpha = \hat{\{}\alpha_1\hat{\}}$, dann gilt $[\![\alpha]\!](\rho) = ([\![\alpha_1]\!](\rho))^*$, d. h. $[\![\alpha]\!](\rho)$ ist der Stern von $[\![\alpha_1]\!](\rho)$.
- Wenn $\alpha = \hat{[}\alpha_1\hat{]}$, dann gilt $[\![\alpha]\!](\rho) = [\![\alpha_1]\!](\rho) \cup \{\varepsilon\}$.
- Wenn $\alpha = \hat{(}\alpha_1\hat{|}\alpha_2\hat{)}$, dann gilt $[\![\alpha]\!](\rho) = [\![\alpha_1]\!](\rho) \cup [\![\alpha_2]\!](\rho)$.
- Wenn $\alpha = \alpha_1\alpha_2$, dann gilt $[\![\alpha]\!](\rho) = [\![\alpha_1]\!](\rho) \cdot [\![\alpha_2]\!](\rho)$, wobei die Objektsprachen $[\![\alpha_1]\!](\rho)$ und $[\![\alpha_2]\!](\rho)$ durch Konkatenation verknüpft sind.

Wenn nun $\mathcal{E}$ die EBNF-Regel $v ::= \alpha$ enthält, dann muss $W(\mathcal{E}, v) = [\![\alpha]\!](\rho)$ gelten, wobei für jedes $u \in V$ gilt: $\rho(u) = W(\mathcal{E}, u)$.

Beispiel 2.11. Wir betrachten die EBNF-Definition des vorangegangenen Beispiels und den EBNF-Term $\alpha = \hat{(}\,\mathtt{a}A\mathtt{b}\,\hat{|}\,\mathtt{a}\,\hat{)}$, der sich in die EBNF-Terme $\alpha_1, \alpha_2, \alpha_3$ und α_4 zerlegen lässt:

$$\alpha = \hat{(}\ \underbrace{\underbrace{\mathtt{a}}_{\alpha_3}\ \underbrace{A\ \mathtt{b}}_{\alpha_4}}_{\alpha_1}\ \hat{|}\ \underbrace{\mathtt{a}}_{\alpha_2}\ \hat{)}$$

Nehmen wir nun an, dass die syntaktischen Kategorien von S und A die Sprachen

$$\rho(S) = W(\mathcal{E}, S) = \{\mathtt{c}^n\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid n, k \geq 0\} \qquad \text{bzw.} \qquad \rho(A) = W(\mathcal{E}, A) = \{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid k \geq 0\}$$

seien. Dann gilt: $W(\mathcal{E}, A) = [\![\hat{(}\,\mathtt{a}A\mathtt{b}\,\hat{|}\,\mathtt{a}\,\hat{)}]\!](\rho)$. Das lässt sich folgendermaßen verifizieren:

$$\begin{aligned}
[\![\hat{(}\,\mathtt{a}A\mathtt{b}\,\hat{|}\,\mathtt{a}\,\hat{)}]\!](\rho) &= [\![\mathtt{a}A\mathtt{b}]\!](\rho) \cup [\![\mathtt{a}]\!](\rho) \\
&= [\![\mathtt{a}]\!](\rho) \cdot [\![A]\!](\rho) \cdot [\![\mathtt{b}]\!](\rho) \cup [\![\mathtt{a}]\!](\rho) \\
&= \{\mathtt{a}\} \cdot [\![A]\!](\rho) \cdot \{\mathtt{b}\} \cup \{\mathtt{a}\} \\
&= \{\mathtt{a}\} \cdot W(\mathcal{E}, A) \cdot \{\mathtt{b}\} \cup \{\mathtt{a}\} \\
&= \{\mathtt{a}\} \cdot \{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid k \geq 0\} \cdot \{\mathtt{b}\} \cup \{\mathtt{a}\} \\
&= \{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid k \geq 1\} \cup \{\mathtt{a}\} \\
&= \{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid k \geq 0\}
\end{aligned}$$

Auf ähnliche Weise beweist man, dass $W(\mathcal{E}, S) = [\![\hat{\{}c\hat{\}}\,A]\!](\rho)$ ist. □

Formal definiert man $W(\mathcal{E}, v)$ als $W(\mathcal{E}, v) = \hat{\rho}(v)$, wobei $\hat{\rho}$ das kleinste $\rho\colon V \to \mathcal{P}(\Sigma^*)$ ist mit

$$\forall (v ::= \alpha) \in R\colon \rho(v) = [\![\alpha]\!](\rho)$$

und die Abbildungen des Typs $V \to \mathcal{P}(\Sigma^*)$ wie folgt (partiell) geordnet sein sollen:

$$\rho_1 \leq \rho_2 \quad \Leftrightarrow \quad \forall v \in V\colon \rho_1(v) \subseteq \rho_2(v)\ .$$

An dieser Stelle muss man beweisen, dass $\hat{\rho}$ wohldefiniert ist, d. h., dass es existiert und eindeutig ist. Dafür bedient man sich des Fixpunktsatzes von Tarski (siehe appendix B.8) und nutzt die folgende äquivalente Formulierung der Definition: Sei $f\colon (V \to \mathcal{P}(\Sigma^*)) \to (V \to \mathcal{P}(\Sigma^*))$ definiert für jedes $v \in V$ durch $f(\rho)(v) = [\![\alpha]\!](\rho)$ wenn $(v ::= \alpha) \in R$. Dann ist $\hat{\rho}$ der kleinste Fixpunkt von f. (Ein Fixpunkt einer Funktion f ist ein Element ρ, welches unter f auf sich selbst abgebildet wird, d. h., $f(\rho) = \rho$.) Nun lässt sich zeigen, dass unsere Ordnung auf $V \to \mathcal{P}(\Sigma^*)$ und die Funktion f die Eigenschaften erfüllen, die im Fixpunktsatz von Tarski gefordert werden. Wir erhalten das Ergebnis, dass der kleinste Fixpunkt von f eindeutig bestimmt ist, und zwar mit

$$\forall v \in V\colon \hat{\rho}(v) = \textstyle\bigcup_{i \geq 0} f^i(\bot)(v)\ ,$$

wobei $\bot$ jedes Element in V auf die leere Sprache abbildet, d. h. $\bot(v) = \emptyset$ für jedes $v \in V$.

Beispiel 2.12. Für die EBNF-Definition aus Beispiel 2.8 wollen wir jetzt mit Hilfe der Fixpunktsemantik die syntaktischen Kategorien $W(\mathcal{E}, S)$ und $W(\mathcal{E}, A)$ ermitteln. Dazu schreiben wir die Abbildungen des Typs $V \to \mathcal{P}(\Sigma^*)$ als Spaltenmatrizen auf, wobei der obere Eintrag das Bild von S enthält, der zweite das Bild von A.

Dann berechnen wir wie folgt.

$$\binom{\emptyset}{\emptyset} \overset{f}{\longmapsto} \binom{\emptyset}{\{\mathtt{a}\}} \overset{f}{\longmapsto} \binom{\{\mathtt{c}^n\mathtt{a} \mid n \geq 0\}}{\{\mathtt{a}\mathtt{a}\mathtt{b}\} \cup \{\mathtt{a}\}} \overset{f}{\longmapsto} \binom{\{\mathtt{c}^n\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid n \geq 0, 0 \leq k \leq 1\}}{\{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid 0 \leq k \leq 2\}}\ .$$

Allgemein gilt für jedes $i \geq 2$:

$$\binom{\emptyset}{\emptyset} \overset{f^i}{\longmapsto} \binom{\{\mathtt{c}^n\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid n \geq 0,\ 0 \leq k \leq i-2\}}{\{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid 0 \leq k \leq i-1\}}\ .$$

Diese Behauptung lässt sich durch vollständige Induktion beweisen, siehe Beispiel B.43. Somit erhalten wir:

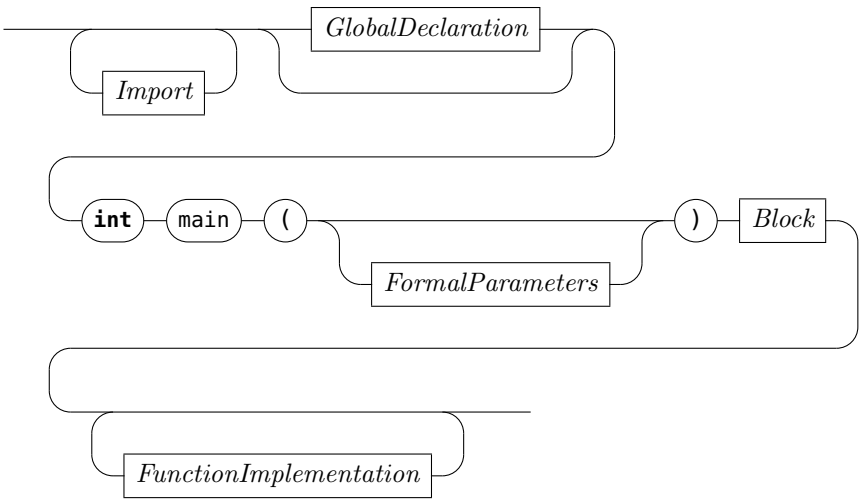
$$\binom{W(\mathcal{E}, S)}{W(\mathcal{E}, A)} = \binom{\bigcup_{i \geq 0} f^i(\bot)(S)}{\bigcup_{i \geq 0} f^i(\bot)(A)} = \binom{\{\mathtt{c}^n\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid n, k \geq 0\}}{\{\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid k \geq 0\}}\ ,$$

insbesondere ist also die Sprache der EBNF: $W(\mathcal{E}, S) = \{\mathtt{c}^n\mathtt{a}^k\mathtt{a}\mathtt{b}^k \mid n, k \geq 0\}$. □

3 Aufbau eines C-Programms

Ein Programm in der Programmiersprache C ist folgendermaßen aufgebaut:

Program



Import siehe Abschnitt 7.1

Wir starten mit einem kleinen Beispielprogramm von C, mit dessen Hilfe die Summe der ersten n Quadratzahlen berechnet werden kann.

```
1  /* Summation */
2  #include <stdio.h>
3
4  int n, s;
5
6  int main()
7  { int i;
8
9    scanf("%d",&n);
10   i = 1;
11   s = 0;
12   while (i <= n)
13   { s = s+i*i;
14     i = i+1;
15   }
16   printf("%d",s);
17   return 0;
18 }
```

3.1 Erste Bemerkungen

- Signifikantes Merkmal eines C-Programms ist die Funktion `main()`, die in jedem Programm genau einmal existieren muss. Eine Funktion kann Argumente (eine Liste formaler Parameter, siehe Kapitel 5) haben; in unserem Beispiel ist die Funktion `main()` parameterlos.

- Ein *C*-Programm kann mit `#include` auf Bibliotheken (z. B. auf `stdio`) zugreifen, d. h. dort deklarierte Konstanten, Typen, Variablen und Funktionen importieren. `stdio` ist in der Modulbibliothek standardmäßig vorhanden. `#include <stdio.h>` erlaubt z. B. die Verwendung von Lese-und Schreibfunktionen (importiert aus der Bibliothek `stdio`). Das Kommando `scanf("%d",&n);` liest eine ganze Zahl (`INTEGER`) als Eingabe im Dezimalformat von der Tastatur und speichert den Wert unter der Adresse der Variablen `n`. Das Kommando `printf("%d",s);` gibt die in `""` eingeschlossene Zeichenfolge auf dem Bildschirm aus und setzt dabei an die Stelle des „Platzhalters“ `%d` den Wert der Variablen `s` im Dezimalformat.
- In einem *C*-Programm können (optional) globale Konstanten, Typen, Variablen und Funktionen (in unserem Beispiel die `INTEGER`-Variablen `n` und `s`) deklariert werden.
- Der Block einer Funktion beginnt mit `{` und endet mit `}` und enthält (optional) die Deklaration lokaler Konstanten, Typen und Variablen (in unserem Beispiel die `INTEGER`-Variable `i`) sowie eine Folge (Sequenz) von Statements.
- Die Implementation von (immer global gültigen) Funktionen kann auch hinter der Funktion `main()` erfolgen (siehe Funktionskonzept, Kapitel 5; dort gehen wir auch auf diese zwei unterschiedlichen Implementierungsmöglichkeiten ein).

In Programmen können folgenden Objekte auftreten:

- Bezeichner (Identifier) (in unserem Beispiel `i`, `n`, `s` und `main`)
- Zahlen (z. B. 0, 1)
- Schlüsselwörter (z. B. `int`, `while` und `return`)
- Operatoren und Begrenzer (z. B. `=`, `+`, `{`, `}`, ...)
- Kommentare (`/* Summation */`)
  - Kommentare werden durch das Doppelzeichen `/*` eingeleitet und durch das Doppelzeichen `*/` abgeschlossen,
  - sind wichtig für die Lesbarkeit und Wartbarkeit von Programmen,
  - enthalten Erläuterungen und Hinweise für den Menschen, haben aber keinerlei Auswirkungen auf die Programmausführung auf einem Computer,
  - dürfen wie Leerzeichen überall zwischen zwei Symbolen stehen und
  - werden nicht in den Syntaxdiagrammen berücksichtigt.

Beim Entwickeln von Programmen in *C* müssen (wie in anderen Programmiersprachen auch) einige Bedingungen beachtet werden, wie z. B.

- Schlüsselwörter werden immer *klein* geschrieben.
- Objekte müssen *vor* ihrer erstmaligen Verwendung deklariert werden. Ausnahme: Objekte, die aus Bibliotheken importiert werden oder die vordefiniert sind.
- Die Schreibweise von Bezeichnern ist verbindlich, d. h. es wird zwischen Groß- und Kleinschreibung unterschieden.
- Bezeichner dürfen (mit gleicher Schreibweise) in einer Deklaration nur einmal deklariert werden.

Auf einige weitere Bedingungen, die besonders im Zusammenhang mit der Verwendung von Statements zu beachten sind, wird bei der Betrachtung dieser Statements eingegangen.

Schlüsselwörter

Schlüsselwörter (oder: keywords) sind Wörter mit fester Bedeutung, die mit einem Kleinbuchstaben oder _ beginnen. Schlüsselwörter in *C* sind z. B. :

```
break    case     char     const    do       double   else     enum
float    for      if       int      long     return   short    struct
switch   typedef  union    unsigned void     while
```

Operatoren und Begrenzer

Operatoren und Begrenzer (oder: punctuators) in *C* sind z. B.:

```
{   }   (   )   /   %   &   &&   ->   +   *   #   |   ||   ;   -
```

Präprozessor-Token

Präprozessor-Token (oder: preprocessor tokens) sind Wörter, die vom Präprozessor verarbeitet werden, dazu zählen z. B. `define` und `include`.

Bezeichner (Identifier)

Jedes Programm arbeitet mit verschiedenen Objekten. Eine eindeutige Identifizierung der Objekte wird durch eine Namensgebung erreicht. Diese Namen werden Bezeichner (oder: Identifier) genannt. Bezeichner sind Zeichenfolgen, die unter Einhaltung der oben bereits genannten und der folgenden Bedingungen gebildet werden dürfen:

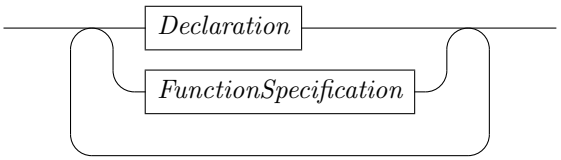
- Bezeichner dürfen Zeichenfolgen beliebiger Länge sein; allerdings sind für die Unterscheidung von zwei Bezeichnern nur die ersten 32 Zeichen signifikant.
- Bezeichner dürfen große und kleine Buchstaben (keine Umlaute!), Ziffern und den Unterstrich (_) enthalten.
- Bezeichner dürfen *nicht* mit einer Ziffer beginnen.
- Schlüsselwörter und einige reservierte Namen dürfen nicht als Bezeichner verwendet werden.

Einige Bezeichner haben in *C* eine besondere Bedeutung (sog. spezielle Bezeichner), z. B. `main`.

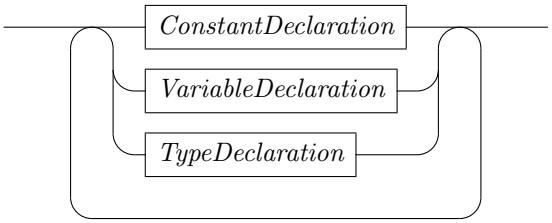
3.2 Deklarationen

Wie wir bereits gesehen haben, müssen Objekte, die nicht vordefiniert sind und nicht importiert werden, im Programm deklariert werden. Deklarationen können global oder innerhalb eines Blocks (siehe Abschnitt 3.3) erfolgen. Ausnahme: Funktionen dürfen nur im globalen Deklarationsteil oder im Teil *FunctionImplementation* (siehe Kapitel 5) spezifiziert werden.

GlobalDeclaration



Declaration



Wir werden an dieser Stelle nur die Deklaration von einfachen Konstanten und Variablen sowie die prinzipielle Deklaration von Typen angeben. Auf die Spezifikation von Funktionen werden wir im Kapitel 5 und auf die Deklaration von benutzerdefinierten Konstanten, Typen und Variablen im Kapitel 6 eingehen.

3.2.1 Konstantendeklaration

Zur Deklaration von Konstanten sind zwei Wege möglich:

Die Definition eines konstanten Wertes kann über die Definition eines Makros erfolgen:

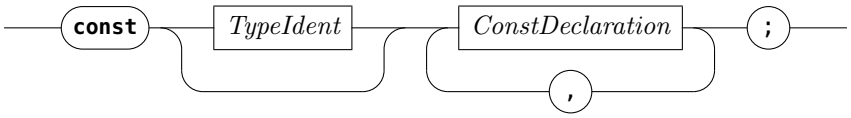
```
#define MAX 40
```

In diesem Fall wird bereits beim Compilieren überall wo der Bezeichner `MAX` auftaucht, dieser durch den Wert 40 ersetzt (bzw. allg. durch den Wert des Ausdrucks, der bei der Definition rechts vom Makro-Bezeichner steht). Damit kann der so definierte Wert auch in weiteren Deklarationen verwendet werden, z. B. in

```
int feld[MAX];
```

Alternativ kann die Deklaration einer Konstanten mit dem Schlüsselwort `const` eingeleitet werden; danach können die Angabe eines Typs sowie eine oder mehrere durch Kommata getrennte Konstantendeklarationen folgen.

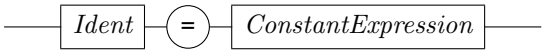
ConstantDeclaration



In diesem Fall wird eine schreibgeschützte Variable mit einem Initialwert erzeugt, die eine Adresse besitzt und deren Wert demzufolge erst zur Laufzeit des Programms abgerufen werden kann. Fehlt der Typbezeichner, wird der Typ aus dem angegebenen Wert abgeleitet, andernfalls wird der angegebene Typ verwendet.

Eine Konstantendeklaration besteht aus einem Identifier, dem Terminalsymbol '=' und einem konstanten Ausdruck, dessen Wert der Konstanten zugeordnet wird. Die Werte von so definierten „Konstanten" können zur Laufzeit des Programms nicht verändert werden.

ConstDeclaration



Beispiel 3.1.

```
const Zahl1 = 4, Zahl2 = 12;
const Zeichen = 'A';
const double Wert = 27.9;
const float Pi = (float) 3.1415;
```

□

Mit `const` deklarierte Identifier können *nicht* für weitere Deklarationen verwendet werden, da sie beim Compilieren noch nicht zur Verfügung stehen.

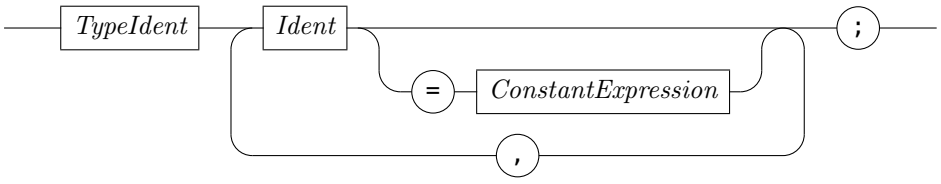
```
const max = 40;
int feld[max];    /* falsch!!! */
```

Auf die Deklaration von Konstanten selbst definierter Typen werden wir bei der Beschreibung dieser Typen (Kapitel 6) eingehen.

3.2.2 Variablendeklaration

Die Deklaration von Variablen besteht aus der Angabe des Typs der zu deklarierenden Variablen und einem oder mehreren durch Kommata getrennten Bezeichnern. Alle angegebenen Bezeichner bezeichnen Variablen vom gleichen Typ. Jede Variable kann mit einem Initialwert belegt werden. Geschieht das nicht, dann ist ihr Initialwert unbestimmt.

VariableDeclaration



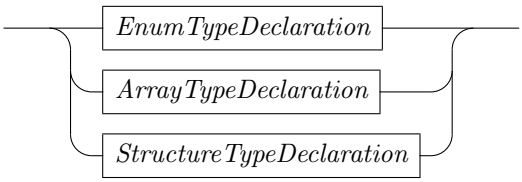
Beispiel 3.2. `int n, s = -3, l = 27;  /* n = ?, s = -3, l = 27 */`

□

3.2.3 Typdeklaration

Auf die Deklaration benutzerdefinierter Typen wollen wir an dieser Stelle nur kurz eingehen, da sie im Kapitel 6 genauer beschrieben werden. Folgende Typdeklarationen sind möglich:

TypeDeclaration



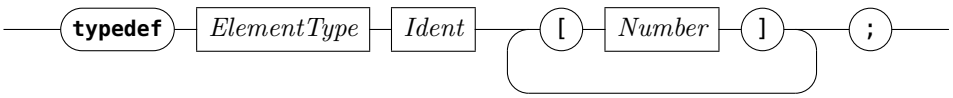
Ein Typ (oder genauer: Datentyp) ist eine Menge von Werten; diese Menge erhält einen Namen, den wir Typbezeichner nennen.

Typen werden deklariert, um danach unter Verwendung des Typbezeichners (auch an unterschiedlichen Stellen des Programms) Variablen deklarieren zu können. Die Deklaration von benutzerdefinierten Typen erfolgt im Deklarationsteil eines Blocks oder (wenn sie z. B. für formale Parameter in Funktionsdefinitionen verwendet werden sollen) im globalen Deklarationsteil.

Die Typdeklaration wird mit dem Schlüsselwort `typedef` eingeleitet, danach folgt die Beschreibung des zu deklarierenden Typs (typabhängig etwas unterschiedlich) und die Angabe des Typ-Bezeichners.

Wir wollen hier beispielhaft die Deklaration eines Array-Typs angeben:

ArrayTypeDeclaration



Als Beispiel sei das folgende Programmfragment angegeben.

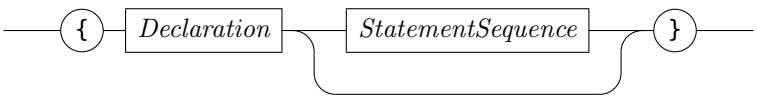
```
1 | typedef int feld[20];     /* deklariert einen ARRAY-Typ mit
2 |                                 20 Elementen des Typs int;
3 |                                 der Bezeichner des Typs ist feld */
4 |
5 | feld A, B;                /* zwei Variablen des Typs feld */
```

Auf die Deklaration anderer Typen (z. B. Record-Typen, Pointer-Typen) gehen wir im Kapitel 6 ein.

3.3 Block einer Funktion

Nachdem wir nun einfache Objekte in einem Programm (global zur Funktion `main`) deklarieren können, wollen wir abschließend den Verarbeitungsteil kennenlernen. Der Verarbeitungsteil eines Programms bzw. einer Funktion wird *Block* genannt; er ist durch geschweifte Klammern kenntlich gemacht.

Block

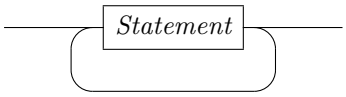


Wie wir sehen, gibt es auch hier den bereits besprochenen Deklarationsteil. Allerdings handelt es sich hier ausschließlich um die Definition lokaler Objekte, d. h. um Objekte, die nur innerhalb dieses Blocks bzw. der zugehörigen Funktion gültig sind.

Achtung: Es sei noch einmal darauf hingewiesen, dass an dieser Stelle *keine* Funktionen deklariert werden dürfen.

Nach den (lokalen) Deklarationen folgt die Beschreibung der Verarbeitung der Objekte. Dieser Teil besteht aus einer Sequenz von Verarbeitungsschritten (Statements), ausgedrückt durch das Nichtterminalsymbol *StatementSequence*:

StatementSequence

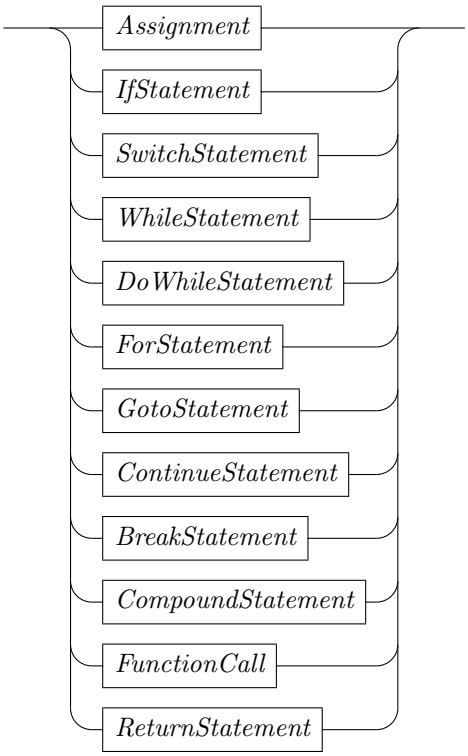


Welche konkreten Ausprägungen nun diese Verarbeitungsschritte (Statements) haben können, d. h. welche Möglichkeiten uns zur Formulierung eines Algorithmus zur Verfügung stehen, wird im Kapitel 4 beschrieben.

4 Einfache Kontrollstrukturen von C

Ein C-Programm legt eine Menge von Folgen von Zuweisungen (assignments) fest. Eine Folge von Zuweisungen heißt auch Ablauf. Die Programmiersprache bietet dem Programmierer verschiedene Möglichkeiten an, diese Abläufe zu definieren und zu strukturieren; die Möglichkeiten nennt man *Kontrollstrukturen*. Sie werden durch die verschiedenen Ausprägungen der syntaktischen Variablen *Statement* beschrieben. Es folgt das Syntaxdiagramm für *Statement*:

Statement

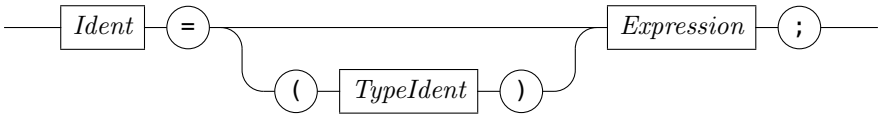


Man kann zwischen einfachen Kontrollstrukturen und den für die Unterprogrammtechnik eingesetzten Kontrollstrukturen (das sind *FunctionCall* und *ReturnStatement*) unterscheiden.

Wir wollen nun zunächst die Kontrollstrukturen *Assignment*, *IfStatement*, *SwitchStatement*, *WhileStatement*, *DoWhileStatement*, *ForStatement*, *BreakStatement* und *CompoundStatement* erläutern. Die Statements *FunctionCall* und *ReturnStatement* werden im Kapitel 5 besprochen. Auf die explizite Behandlung von *GotoStatement* und *ContinueStatement* verzichten wir.

Assignment. Die Zuweisung (*Assignment*) weist der durch *Ident* bezeichneten Variablen explizit den Wert zu, der sich durch die Auswertung des auf der rechten Seite angegebenen Ausdrucks ergibt. *Ident muss* eine Variable bezeichnen! An der Position von *Ident* kann auch ein Name stehen, der ein Element eines anderen Typs bezeichnet (z. B. *Ident für Pointer, Ident[Index] für Feldvariablen, Ident.Ident für Structure-Variablen usw., siehe Kapitel 6). Wenn der Typ des Ausdrucks sich vom Typ der Variablen unterscheidet, kann durch einen cast-Ausdruck auf der rechten Seite der Zuweisung eine Typ-Konvertierung vorgenommen werden.

Assignment



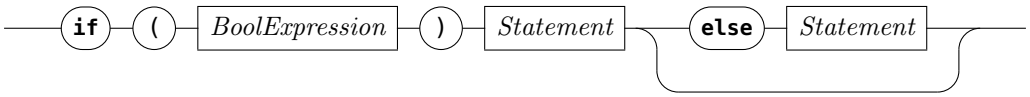
Beispiel 4.1.

```
int n, s;
float k;
. . .
k = (float) (n * s);
```
□

Das Ergebnis des Produkts von `n * s` ist zunächst vom Typ **`int`**, wird aber vor der Zuweisung durch den cast-Ausdruck **`(float)`** in den Typ **`float`** konvertiert.

IfStatement. Durch die Anwendung des *IfStatement*s wird die bedingte Ausführung von Statements realisiert. Wenn die Auswertung von *BoolExpression* den logischen Wert `true` (`1`) ergibt, wird das erste Statement ausgeführt, bei `false` (`0`) das zweite Statement. Die Entscheidung kann auch nur einseitig (Fehlen des 'else'-Zweiges) sein; dann wird bei `false` die Arbeit nach dem *IfStatement* fortgesetzt.

IfStatement



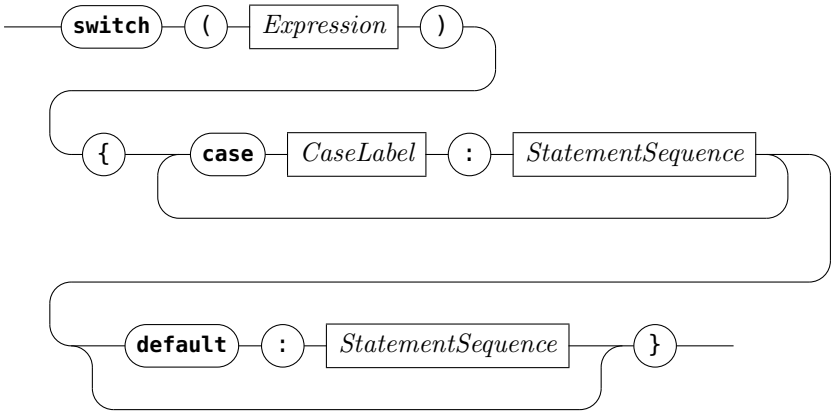
Beispiel 4.2. `if (h == tail) { h = p; q = r; }` □

Beispiel 4.3. `if (h == tail) { h = p; q = r; } else { h = r; q = p; }` □

Hinweis: Auf logische Werte und Ausdrücke vom Typ *BoolExpression* werden wir im Kapitel 6 im Zusammenhang mit dem Datentyp **`int`** genauer eingehen.

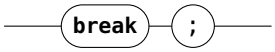
SwitchStatement. Des öfteren ist eine Fallunterscheidung mit *mehr* als zwei Fällen notwendig. Dafür wird das *SwitchStatement* verwendet.

SwitchStatement



Innerhalb des *SwitchStatements* (und zwar am Ende einer der *StatementSequence*s) wird gewöhnlich das *BreakStatement* verwendet.

BreakStatement



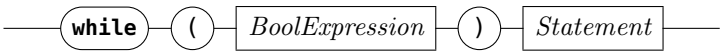
Beispiel 4.4. Sei `exp` eine *SimpleExpression* vom Typ **`int`**.

```
switch (exp)
{ case 0:   x = 0;      a = b;      break;
  case 1:   x = x + 1;  a = 2 * b;  break;
  default:  x = 1;      a = 0;
}
```

Diese Fallunterscheidung legt fest, welche Aktionen in Abhängigkeit des Wertes von `exp` durchzuführen sind. Hat `exp` den Wert `n`, dann wird die Ausführung hinter dem Label **`case`** `n:` fortgesetzt. Gibt es kein solches Label, dann wird an die Stelle **`default`**: gesprungen. Beim Erreichen von **`break`** wird das *SwitchStatement* beendet. Steht am Ende einer Anweisungsfolge kein **`break`**, so werden alle weiteren Anweisungen bis zum nächsten **`break`** bzw. bis zum Ende des *SwitchStatement* abgearbeitet (bei nachfolgenden **`case`**-Labels erfolgt dann kein Gleichheitstest mehr). □

WhileStatement. Das *WhileStatement* realisiert die wiederholte Ausführung eines Statements (Schleifenrumpf). *Vor* der Ausführung des Schleifenrumpfes wird getestet, ob die Schleifenbedingung *BoolExpression* den logischen Wert `true` (`1`) oder `false` (`0`) ergibt. Nur bei `true` wird der Schleifenrumpf ausgeführt und danach *BoolExpression* erneut getestet. Wenn sich gleich beim ersten Aufruf des *WhileStatements* `false` ergibt, wird der Rumpf *nicht* ausgeführt.

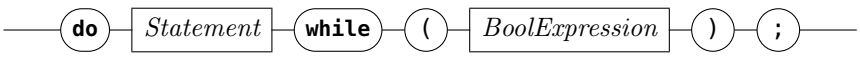
WhileStatement



Beispiel 4.5. `while (s - 0.5*s > eps) { s = 0.5*s; }` □

DoWhileStatement. Das *DoWhileStatement* ist vergleichbar mit dem *WhileStatement*, nur dass die Schleifenbedingung nicht *vor*, sondern *nach* der Ausführung des Schleifenrumpfes getestet wird. Insbesondere wird also der Rumpf *mindestens einmal* ausgeführt.

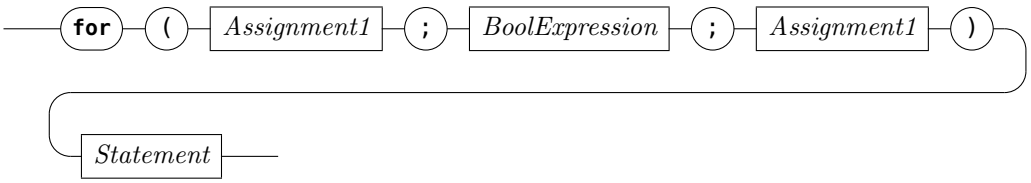
DoWhileStatement



Beispiel 4.6. `do { s = 0.5*s; } while (s - 0.5*s > eps);` □

ForStatement. Oft steht schon *vor* Ausführung der Schleife fest, wie oft eine bestimmte Aktionsfolge ausgeführt werden muss. In diesem Fall benutzt man eine For-Schleife statt einer While-Schleife.

ForStatement



Assignment1 ist vom Strukturaufbau wie *Assignment*, hat jedoch kein abschließendes Semikolon. Das erste Vorkommen von *Assignment1* dient in der Regel dazu, der Zählvariablen einen Anfangswert zuzuweisen (z. B. `i=1`). Der Ausdruck *BoolExpression* wird *vor* jedem Schleifendurchlauf getestet. Wenn er den Wert `false` (`0`) ergibt, wird der Schleifenrumpf nicht mehr durchlaufen (wie beim *WhileStatement*), sonst doch. Das zweite Vorkommen von *Assignment1* wird zur Berechnung des nächsten Wertes der Zählvariablen *nach* jedem Schleifendurchlauf ausgeführt.

Hinweis: Im ANSI-C-Standard sind der Kopf des *WhileStatement*, des *DoWhileStatement* und des *ForStatement* etwas allgemeiner definiert (*Expression* anstatt *BoolExpression* oder *Assignment1*).

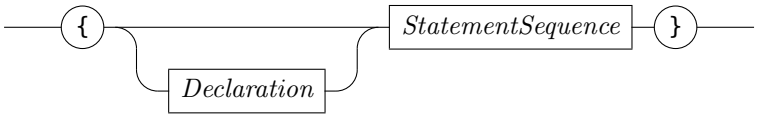
Beispiel 4.7. `for (i = 1; i <= 10; i = i + 1) printf("%d\n", i);` □

Hinweis: Das Assignment `i = i + 1` wird in C/C^{++} oft auch durch `++i` abgekürzt (`++` ist der Inkrementationsoperator). Bei `++i` handelt es sich um das sogenannte Präinkrement. Neben dem Präinkrement gibt es auch noch das Postinkrement `i++`. Der Unterschied zwischen den beiden Inkrementoperationen besteht lediglich im Rückgabewert der Ausdrücke:

```
int a, b, x, y;
a = b = 42;

x = ++a;  // x hat jetzt den Wert 43
y = b++;  // y hat jetzt den Wert 42
```

CompoundStatement. Das *CompoundStatement* wird verwendet, um die Abarbeitung einer *StatementSequence* anstelle eines einzelnen Statements in anderen Anweisungen zu ermöglichen. Die Syntax eines *CompoundStatement* entspricht der eines *Block*s; wir verwenden jedoch bewusst das *CompoundStatement*, um die Abarbeitung einer *StatementSequence* zu bezeichnen, während wir unter einem *Block* den Rumpf einer Funktion verstehen wollen.

CompoundStatement



Im *CompoundStatement* können lokale Objekte deklariert werden, die dann nur innerhalb dieses *CompoundStatement*s Gültigkeit besitzen. Dabei ist allerdings Vorsicht geboten, wie das folgende Beispiel zeigt:

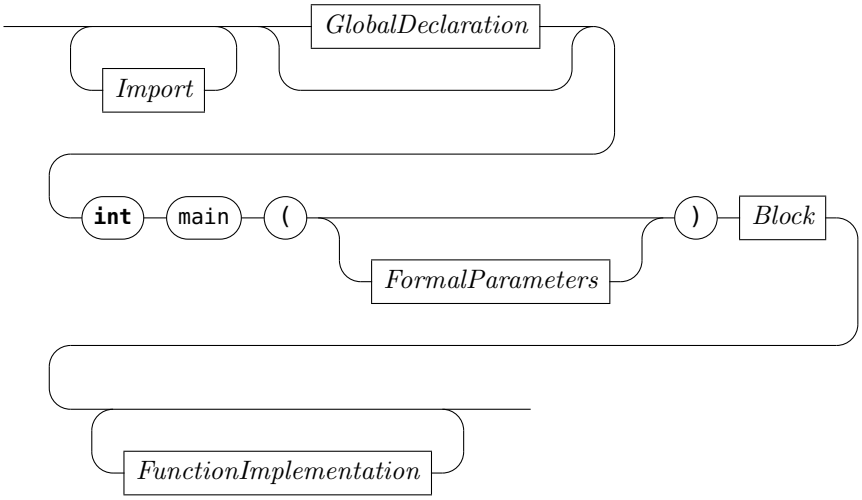
```
 1 #include <stdio.h>        /* Endlosschleife */
 2
 3 int i;
 4
 5 int main()
 6 { i = 5;
 7   while (i > 0)           /* Endlosschleife, da innerhalb des CompoundStatements */
 8   { int i = 8;            /* nur die lokal deklarierte Variable i sichtbar ist,  */
 9     printf("i = %d\n",i); /* für Test auf Abbruch aber die globale Variable i    */
10     i = i-1;              /* verwendet wird. Außerdem erreicht das innere i nie  */
11   }                       /* den Wert 0, da es am Anfang eines jeden Schleifen-  */
12   return 0;               /* durchlaufs erneut auf 8 gesetzt wird.               */
13 }
```

5 Funktionskonzept

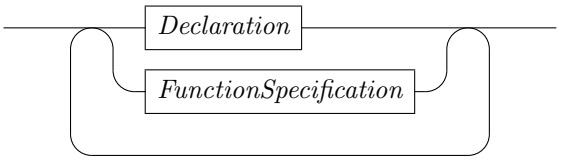
Die in diesem Kapitel eingeführte Unterprogrammtechnik dient dazu, den Kontrollfluss zu strukturieren. Dabei müssen Funktionen in *C* grundsätzlich global deklariert werden.

Wir wiederholen zunächst die Syntaxdiagramme von *Program* und *GlobalDeclaration* aus Kapitel 3:

Program



GlobalDeclaration



Beispiel 5.1. Wir zeigen den Zusammenhang von *FunctionHeading*, *FunctionImplementation* und *FunctionSpecification* für die Funktion f:

	FunctionHeading	*FunctionImplementation*
Mathematik	$f\colon \mathbb{Z} \to \mathbb{Z}$	$f\colon \mathbb{Z} \to \mathbb{Z}$ $f(x) = 3x^2 + 4$

C	`int f (int x);`	`int f (int x)`
		`{ return (3*x*x + 4);`
		`}`

□

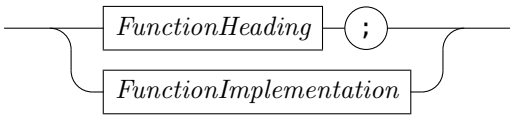
5.1 Deklaration von Funktionen

Beispiel 5.2. Auf Basis der Taylorentwicklung von sin können wir eine Näherungsfunktion `sinus` in *C* angeben. Die Funktion `sinus` kann dann beliebig oft im Code des Programms verwendet werden.

```
1  #include <stdio.h>
2
3  /* Näherung des Sinus, basierend auf der Taylorentwicklung 5-ten Grades */
4  float sinus(float x)      /* Funktionsimplementation */
5  { return x  -  x * x * x / 6  +  x * x * x * x * x * x / 120; }
6
7  int main()
8  { float x1, x2, y1, y2;
9
10   scanf("%f", &x1);
11   y1 = sinus(x1);         /* Funktionsaufruf */
12   printf("sinus(%f) = %f\n", x1, y1);
13
14   scanf("%f", &x2);
15   y2 = sinus(x2);         /* Funktionsaufruf */
16     printf("sinus(%f) = %f\n", x2, y2);
17
18   return 0;
19 }
```

□

FunctionSpecification



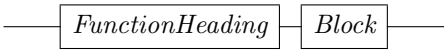
Wenn als Funktionsspezifikation nur die *FunctionHeading* angegeben ist, dann muss der zugehörige *Block* an anderer Stelle (z. B. hinter der `main`-Funktion oder in einem anderen File (siehe Modularisierungskonzept)) angegeben werden. Dann muss allerdings die *FunctionHeading* unmittelbar vor dem zugehörigen *Block* wiederholt werden. Beispiel 5.2 könnte demzufolge auch so aussehen:

```
1  #include <stdio.h>
2
3  /* Näherung des Sinus, basierend auf der Taylorentwicklung 5-ten Grades */
4  float sinus(float x);     /* Funktionskopf*/
5
6  int main()
7  { float x1, x2, y1, y2;
8
9    scanf("%f", &x1);
10   y1 = sinus(x1);         /* Funktionsaufruf */
11   printf("sinus(%f) = %f\n", x1, y1);
12
13   scanf("%f", &x2);
14   y2 = sinus(x2);         /* Funktionsaufruf */
15     printf("sinus(%f) = %f\n", x2, y2);
16
17   return 0;
18 }
19
20 float sinus(float x)      /* Funktionsimplementation */
21 { return x  -  x * x * x / 6  +  x * x * x * x * x * x / 120; }
```

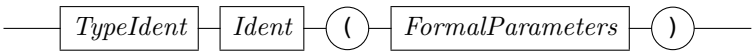
```
1  int x, y;
2
3  int P(int a, int b, int *c)
4  { . . .
5    *c = *c + 1;
6    . . .
7    return . . .;
8  }
9
10 int main()
11 { . . .
12   x = 5 + P(3, 4 + y, &y);
13   . . .
14 }
```

Abbildung 5.1: Funktion vom Ergebnistyp **int** mit Aufruf.

FunctionImplementation



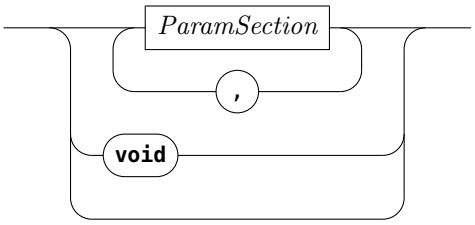
FunctionHeading



Der Kopf einer Funktion muss mit einem Typ-Identifier (*TypeIdent*) eingeleitet werden, der den Ergebnistyp angibt. Bei Funktionen, die kein Ergebnis liefern sollen, wird als *TypeIdent* **void** (leerer Typ) verwendet. Die Klammern `()` müssen zur Identifikation einer Funktion angegeben werden, auch wenn die Funktion keine Argumente besitzt.

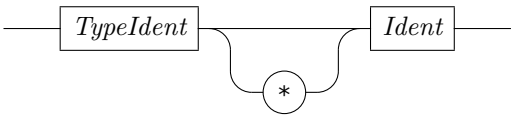
Beispiel 5.3. Die *FunctionHeading* unseres Beispiels lautet: **float** sinus (**float** x) (TypeIdent, Ident, FormalParameters) □

FormalParameters



Die Parameterliste (*FormalParameters*) enthält entweder eine Aufzählung der beim Aufruf der Funktion zu übergebenden Parameter, das Schlüsselwort **void** oder sie ist leer. In *C* wird eine leere Parameterliste (im Gegensatz zu *C*++) als unbestimmte Anzahl von Parametern interpretiert. Deshalb sollte **void** verwendet werden, um eindeutig festzulegen, dass eine Funktion definitiv *keine* Argumente besitzt.

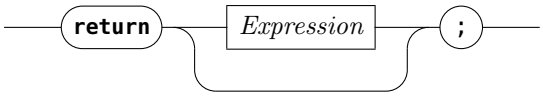
ParamSection



ParamSection enthält die Angabe für jeweils einen der formalen Parameter mit zugehörigem Typ. Es muss zwischen Variablenparametern (*mit* vorangestelltem *, call by reference) und Wertparametern (*ohne* vorangestelltem *, call by value) unterschieden werden. Innerhalb des Blocks einer Funktion muss beim Zugriff auf den Wert, der durch einen Variablenparameter referenziert wird, ein * vor den Bezeichner des Parameters gesetzt werden. In Beispiel 5.2 ist `x` ein Wertparameter vom Typ **float**.

Funktionen, die *nicht* den Ergebnistyp **void** (leer) haben (also z. B. eine Funktion mit Ergebnistyp **int** wie `P` in Abbildung 5.1), müssen ein *ReturnStatement* enthalten; der Wert des Ausdrucks, welcher **return** folgt, liefert den Wert an die aufrufende Stelle zurück. *Expression* muss dabei einen Wert des Typs ergeben, der als Ergebnistyp der Funktion angegeben ist:

ReturnStatement



- Eine Funktion liefert einen Wert, deshalb muss bei ihrer Deklaration vor dem Funktionsnamen der Typ des Ergebnisses angegeben werden. Ggf. ist dieser Typ **`void`**, also leer.
- Innerhalb (typischerweise am Ende) der Anweisungsfolge einer Funktion muss (außer beim Ergebnistyp **`void`**) eine **`return`**-Anweisung angegeben werden. Bei Funktionen vom Typ **`void`** ist die **`return`**-Anweisung am Ende optional, kann aber durchaus innerhalb der Anweisungsfolge (ohne *Expression*) auftreten, um z. B. die Funktion vorzeitig zu beenden. Die **`return`**-Anweisung beendet immer die Abarbeitung der Anweisungsfolge einer Funktion.

5.2 Gültigkeitsbereich von Deklarationen

Der Gültigkeitsbereich der Deklarationen von Objekten ist eine statische Eigenschaft, d. h. er lässt sich am Programm ablesen, ohne dass es ausgeführt werden muss.

Folgende zwei Fälle können auftreten:

- Ein Objekt ist in der Parameterbeschreibung (*FormalParameters*) oder im Block einer Funktion deklariert; dann handelt es sich um einen formalen Parameter bzw. ein lokales Objekt. Der Gültigkeitsbereich dieses Objektes ist ab Deklarationsstelle der gesamte restliche Block der Funktion.
- Ein Objekt ist im globalen Deklarationsteil eines Programmes deklariert (z. B. Konstanten, Variablen, Typen, Funktionen). Dann ist der Gültigkeitsbereich dieses Objekts ab Deklarationsstelle das gesamte restliche Programm, außer der Gültigkeitsbereich der lokal deklarierten Objekte gleichen Namens.

Also: Formale Parameter von Funktionen und die im Block deklarierten Variablen gelten (oder: sind „sichtbar“) nur innerhalb des Blocks dieser Funktion. (Der Vollständigkeit halber sei gesagt, dass dem *Block* das *CompoundStatement* gleichgestellt ist.) Falls innerhalb einer Funktion Objekte mit gleichem Namen wie im globalen Deklarationsteil deklariert sind, so sind die globalen Objekte innerhalb dieser Funktion nicht sichtbar.

In Abbildung 5.2 zeigen wir anhand eines Beispielprogramms den Gültigkeitsbereich der verschiedenen Objekte, wobei der Index P auf das Gesamtprogramm hinweist.

5.3 Pulsierender Speicher bei Aufruf von Funktionen

Der Aufruf einer *Funktion* (*FunctionCall*) erfolgt z. B. in einem Ausdruck (siehe Beispiel `MinIndex`); der durch die **`return`**-Anweisung zurückgelieferte Wert tritt dann in dem Ausdruck an die Stelle des Funktionsaufrufs.

Der Aufruf einer Funktion wird durch Angabe des Namens und aktueller Werte für die formalen Parameter realisiert. Dabei müssen die aktuellen Parameter typmäßig mit den entsprechenden formalen Parametern übereinstimmen.

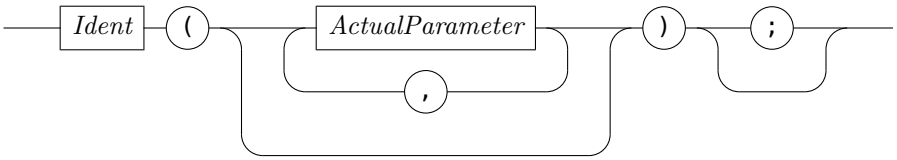
Wenn ein Variablenparameter vorliegt, dann muss der aktuelle Parameter ebenfalls vom Typ Zeiger sein, ggf. muss vor die entsprechende Variable ein Referenzierungsoperator `&` gesetzt werden. Soll innerhalb des Blocks der Funktion auf den Wert des Parameters zugegriffen werden, muss zum Dereferenzieren ein $*$ vor den Bezeichner des Variablenparameters gesetzt werden. Bei einem Wertparameter darf der aktuelle Parameter selbst ein beliebiger Ausdruck des entsprechenden Typs sein.

Funktionen des Typs **`void`** dürfen *nicht* in Ausdrücken verwendet werden, da sie keinen Wert liefern. Sie werden mit ihrem Namen, den Klammern und ggf. mit einer Liste von aktuellen Parametern aufgerufen. Nur in diesen Fällen wird der Funktionsaufruf mit einem Semikolon abgeschlossen. Auch Funktionen eines anderen Typs (also verschieden von **`void`**) dürfen so aufgerufen werden, wenn ihr Wert nicht verwendet werden soll. Auf der Ebene der Syntaxdiagramme wird dies nicht unterschieden; die Unterscheidung ist Teil der statischen Semantik.

```
 1  int y, i;                              |   |
 2                                         |   |
 3  void C()                               |   |   |
 4  { i = i - 1; }                         |   |   |
 5                                         |   |   |
 6  void B(int a, int b)                   |   |   |   |
 7  { int y, z;                                |   |   |           |   |   |   |
 8    y = 1;                                   |   |   |           |   |   |   |
 9    C();                                     |   |   |           |   |   |   |
10  }                                          |   |   |           |   |   |   |
11                                         |   |   |   |           aB  bB  yB  zB
12  void A()                               |   |   |   |   |
13  { int j;                               |   |   |   |   |   |
14    y = 2 * y; j = 5;                    |   |   |   |   |   |
15    if (i < 4)                           |   |   |   |   |   |
16    { i = i - 1;                         |   |   |   |   |   |
17      B(i, 6 * j);                       |   |   |   |   |   |
18    }                                    |   |   |   |   |   |
19  }                                      |   |   |   |   |   |
20                                         |   |   |   |   |   jA
21  int main() /* Hauptprogramm */         |   |   |   |   |
22  { i = 1; y = 2;                        |   |   |   |   |
23    A();                                 |   |   |   |   |
24    return 0;                            |   |   |   |   |
25  }                                      |   |   |   |   |
26                                         yP  iP  CP  BP  AP
```

Abbildung 5.2: Gültigkeitsbereich verschiedener Objekte.

FunctionCall



Die Liste der aktuellen Parameter enthält dabei für jeden in der Funktionsdeklaration angegebenen formalen Parameter einen Ausdruck (bei Variablenparametern nur eine Variable) des entsprechenden Typs.

Betrachten wir jetzt einmal die folgende Funktion P.

```
1  int i, j;
2
3  void P(int a, int b, int *c)
4  { int d, e;
5    . . .
6    *c = *c + 1;
7    . . .
8  }
9
10 int main()
11 { i = 1;
12   P(4 + 2 * i, 5, &i);
13   . . .
14   return 0;
15 }
```

Bezogen auf den Speicher bewirkt der Aufruf dieser Funktion `P` folgendes:

1. Anlegen von Speicherplätzen für die formalen Parameter von `P`, für die Rücksprungadresse und für die in `P` lokal deklarierten Variablen (*Erweiterung* der Umgebung). Da der formale Parameter `c` als Referenzparameter (siehe Abschnitt 5.4) deklariert ist, belegt er einen Speicherplatz, dessen Inhalt auf einen Speicherplatz verweist.
2. Aktivierung der Anweisungsfolge, die im Block von `P` enthalten ist (Speichertransformation)
3. Nach Abarbeiten der Anweisungsfolge: Freigabe der Speicherplätze (Rückkehr zur Aufrufumgebung)

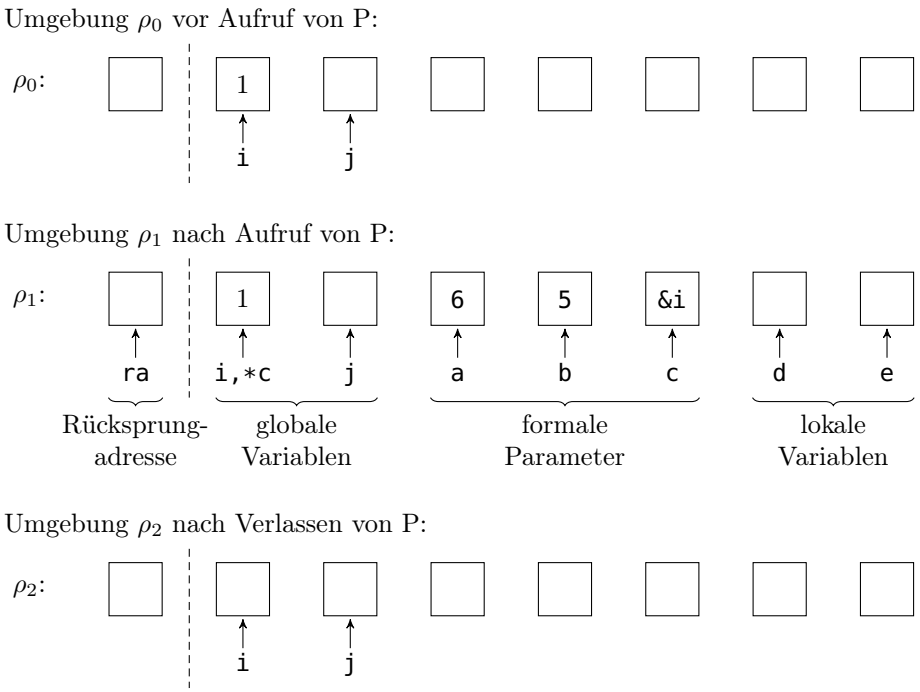


Abbildung 5.3: Veränderung der Umgebung bei Aufruf einer Funktion.

Das heißt, dass der Aufruf einer Funktion die Umgebung (engl.: environment), in der das Programm ausgeführt werden soll, verändert (siehe Abbildung 5.3 zum Konzept der Umgebung): Vor dem Aufruf von P sind die Variablenbezeichner i und j an Speicherplätze gebunden, nach dem Aufruf sind die Variablenbezeichner i, j, a, b, c, d und e an Speicherplätze gebunden, nach Verlassen von P bleibt nur noch die Speicherplatzbindung von i und j erhalten. Dieser Zusammenhang wird in Abbildung 5.3 verdeutlicht. Weiterhin erkennt man, dass der benutzte Speicherbereich beim Aufruf und beim Verlassen einer Funktion quasi pulsiert. Deshalb sprechen wir vom *pulsierenden* Speicher. Dieses Konzept werden wir in der Vorlesung „Programmierung“ formal behandeln.

5.4 Parameterübergabe

Man unterscheidet zwei verschiedene Parameterübergabetechniken: call-by-value und call-by-reference. Im Folgenden sollen uns kleine Programme dabei helfen, die Wirkprinzipien dieser Parameterübergaben zu erkennen. Insbesondere wollen wir für gewünschte Programmstellen die während des Ablaufs bestehenden Variablenbelegungen angeben. Um nun genau auf diese gewünschten Programmstellen Bezug nehmen zu können, fügen wir in den Programmcode Haltepunkte (`label1`, `label2`, ...) als Kommentare ein; immer dann, wenn der Ablauf an einer solchen Marke vorbeikommt, hält das Programm (Haltepunkt) und wir können die aktuelle Variablenbelegung ablesen. Hat eine Variable noch keinen Wert durch eine Zuweisung erhalten, so soll anstelle des Wertes ein `?` angegeben werden. Haben wir in unserem Programm Funktionsaufrufe, so müssen wir auch noch die Rücksprungmarken protokollieren. Zunächst erhält jeder Funktionsaufruf im Programmcode eine eindeutige Rücksprungmarke, bezeichnet durch die Kommentare `$1`, `$2`, ..., `$n`, und unser Speicherbelegungsprotokoll wird durch die Spalte RM (Rücksprungmarkenkeller) erweitert. Nach dem Vorbild des bereits bekannten Rücksprungalgorithmus kann nun der Rücksprungmarkenkeller während eines Programmablaufes auf- und abgebaut werden.

Mit einem konkreten Haltepunkt und einer Rücksprungmarkenfolge kann nun ein ganz spezifischer Ablaufzeitpunkt im Programm identifiziert werden, und für diesen geben wir dann auch die Belegung aller zu diesem Zeitpunkt in der Umgebung (Speicherplätze 1, 2, 3, ...) vorhandenen Variablen an. Ein Trennstrich dient zur besseren Unterscheidung zwischen globalen und lokalen Variablen. Treten hierbei Situationen auf, wo Variablen noch keine Wertzuweisungen erhalten haben, so wollen wir den Speicherinhalt dieser Variablen jeweils durch ein `?` kenntlich machen. Der Wert n eines Variablenparameters steht

für Speicherplatz n.

Wertparameter (call-by-value)

- Anlegen eines Speicherplatzes mit dem Namen des formalen Parameters
- Speicherplatz erhält als Startwert den Wert des zugeordneten aktuellen Parameters (wenn dies ein Ausdruck ist, dann muss dieser vorher berechnet werden)
- Rechnungen auf dem Speicherplatz des formalen Parameters haben *keinen* Einfluss auf den aktuellen Parameter.

Beispiel 5.4.

```
1  int x;
2
3  void unwirksam(int a, int b)
4  { /*label1*/
5    a = a + b;
6    /*label2*/
7  }
8
9  int main()
10 { x = 3;
11   /*label3*/
12   unwirksam(x, 4); /*$1*/
13   /*label4*/
14   return 0;
15 }
```

Hier das zugehörige Speicherbelegungsprotokoll:

Haltepunkt	RM	Umgebung		
		1	2	3
label3	–	x 3		
label1	1	x 3	a 3	b 4
label2	1	x 3	a 7	b 4
label4	–	x 3		

Variablenparameter (call-by-reference)

- Anlegen eines Speicherplatzes für den formalen Parameter zur Aufnahme eines Zeigers (Zeigervariable).
- Speicherplatz erhält die Adresse des aktuellen Parameters, d. h. der formale Parameter „zeigt auf“ den Speicherplatz des aktuellen Parameters.
- Durch Dereferenzierung der Zeigervariablen kann auf den Inhalt des Speicherplatzes des aktuellen Parameters zugegriffen werden.
- Mit Hilfe des formalen Parameters kann der Inhalt des Speicherplatzes des aktuellen Parameters verändert werden; diese Wertänderung bleibt auch nach dem Ende der Funktion erhalten und kann durch den aktuellen Parameter abgerufen werden.

Beispiel 5.5.

```
1  int x;
2
3  void wirksam(int *a, int b)
4  { /*label1*/
5    *a = *a + b;
6    /*label2*/
7  }
8
9  int main()
10 { x = 3;
11   /*label3*/
12   wirksam(&x, 4); /*$1*/
13   /*label4*/
14   return 0;
15 }
```

Hier das zugehörige Speicherbelegungsprotokoll:

Haltepunkt	RM	Umgebung		
		1	2	3
label3	–	x 3		
label1	1	x 3	a 1	b 4
label2	1	x 7	a 1	b 4
label4	–	x 7		

5.5 Gültigkeitsbereich in rekursiven Funktionen

Werden mehrere Funktionen deklariert, die sich rekursiv aufrufen, so gilt das Prinzip des static scope. Es besagt folgendes: Beim Aufruf einer Funktion P sind die Objekte gültig, die zum Zeitpunkt der Deklaration von P gültig sind, und *nicht* die Objekte, die zum Zeitpunkt ihres Aufrufs gelten. Das soll durch das folgende Beispiel veranschaulicht werden.

Beispiel 5.6.

```
1  /*  StaticScope */
2  #include <stdio.h>
3
4  int x, i;
5
6  void A();
7
8  void B()
9  { int x;
10
11   x = 1;
12   printf("%d\n", x);
13    /*label1*/
14   A(); /*$2*/
15    /*label2*/
16   printf("%d\n", x);
17 }
18
19 void A()
20 {  /*label3*/
21   x = 2*x;
22   printf("%d\n", x);
23   if (i < 4)
24   { i = i+1;
25      /*label4*/
26     B(); /*$3*/
27   }
28    /*label5*/
29   printf("%d\n", x);
30 }
31
32 int main() /* Hauptprogramm */
33 { i = 1;
34   x = 2;
35    /*label6*/
36   A(); /*$1*/
37    /*label7*/
38   return 0;
39 }
```

Haltepunkt	RM	Umgebung 1	2	3	4	5
label6	–	x	i			
		2	1			
label3	1	x	i			
		2	1			
label4	1	x	i			
		4	2			
label1	3 : 1		i	x		
		4	2	1		
label3	2 : 3 : 1	x	i			
		4	2	1		
label4	2 : 3 : 1	x	i			
		8	3	1		
label1	3 : 2 : 3 : 1		i		x	
		8	3	1	1	
label3	2 : 3 : 2 : 3 : 1	x	i			
		8	3	1	1	
label4	2 : 3 : 2 : 3 : 1	x	i			
		16	4	1	1	
label1	3 : 2 : 3 : 2 : 3 : 1		i			x
		16	4	1	1	1
label3	2 : 3 : 2 : 3 : 2 : 3 : 1	x	i			
		16	4	1	1	1
label5	2 : 3 : 2 : 3 : 2 : 3 : 1	x	i			
		32	4	1	1	1
label2	3 : 2 : 3 : 2 : 3 : 1		i			x
		32	4	1	1	1
label5	2 : 3 : 2 : 3 : 1	x	i			
		32	4	1	1	
label2	3 : 2 : 3 : 1		i		x	
		32	4	1	1	
label5	2 : 3 : 1	x	i			
		32	4	1		
label2	3 : 1		i	x		
		32	4	1		
label5	1	x	i			
		32	4			
label7	–	x	i			
		32	4			

Ausgaben des Programms:

$$4, 1, 8, 1, 16, 1, 32, 32, 1, 32, 1, 32, 1, 32$$

Achtung! Die Zahlen werden vom Programm zeilenweise, untereinander ausgedruckt.

Anhand eines letzten Beispiels soll für ein gegebenes *C*-Programm nochmals die praktische Vorgehensweise der Erstellung eines Ablaufprotokolls für die Gewinnung gewünschter Variablenbelegungen verdeutlicht werden.

Beispiel 5.7. Das folgende Programm soll für die Eingabe `e = 1` ausgeführt werden.

```
1  #include <stdio.h>
2  void g(int x, int y, int *z);
3
4  void f(int x, int *y)
5  { int u;
6     /*label1*/
7    if (x > 0)
8    { f(x-1, &u);   /*$2*/
9       /*label7*/
10     g(x-1, u, y); /*$3*/
11      /*label8*/
12   }
13   else *y = 1;
14    /*label2*/
15 }
16
17 void g(int x, int y, int *z)
18 { int u;
19    /*label3*/
20   if (x > 0)
21   { f(x-1, &u);   /*$4*/
22      /*label9*/
23     *z = u+y;
24   }
25   else *z = 1;
26    /*label4*/
27 }
28
29 int main()
30 { int e, a;
31   scanf("%d", &e);
32    /*label5*/
33   f(e, &a);       /*$1*/
34   printf("a = %d\n", a);
35    /*label6*/
36   return 0;
37 }
```

Halte-punkt	RM	Umgebung 1	2	3	4	5	6	7	8	9
label5	–	e	a							
		1	?							
label1	1			x	y	u				
		1	?	1	2	?				
label1	2 : 1						x	y	u	
		1	?	1	2	?	0	5	?	
label2	2 : 1						x	y	u	
		1	?	1	2	1	0	5	?	
label7	1			x	y	u				
		1	?	1	2	1				
label3	3 : 1						x	y	z	u
		1	?	1	2	1	0	1	2	?
label4	3 : 1						x	y	z	u
		1	1	1	2	1	0	1	2	?
label8	1			x	y	u				
		1	1	1	2	1				
label2	1			x	y	u				
		1	1	1	2	1				
label6	–	e	a							
		1	1							

(Beachte: `label9` wird bei unserem Ablauf nicht erreicht!)

6 Datenstrukturen

Die Programmiersprache *C* stellt verschiedene einfache Typen standardmäßig zur Verfügung (char, int, float, double). Diese Typen können teilweise durch Modifikatoren (signed, unsigned, short, long) verändert werden. Außerdem können weitere Typen durch Strukturierungsarten vom Programmierer konstruiert werden. In Abbildung 6.1 sind häufig verwendete Typen von *C* in einer Übersicht zusammengefasst.

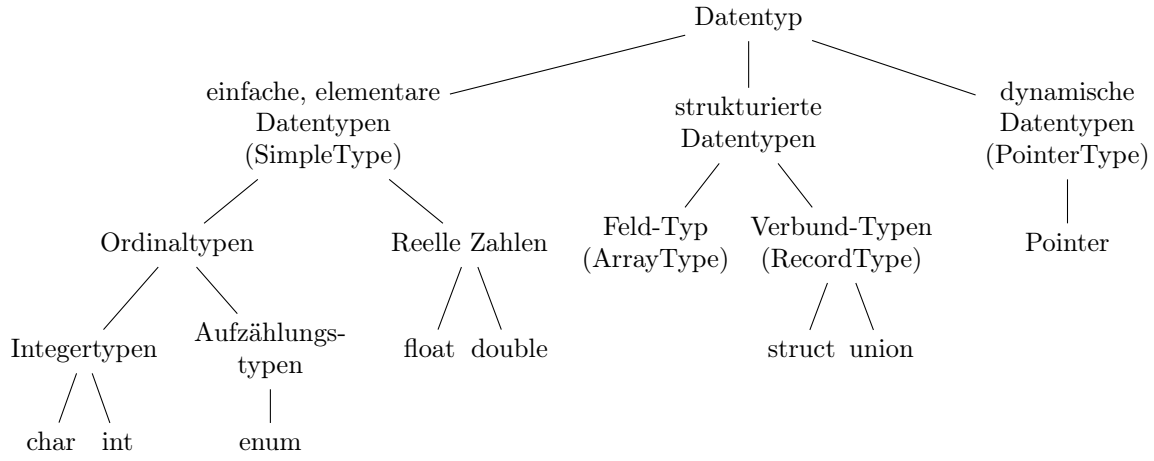


Abbildung 6.1: Übersicht über Datentypen in *C*.

Den prinzipiellen Aufbau einer Typdeklaration haben wir bereits im Kapitel 3 kennengelernt und am Beispiel eines Array-Typs demonstriert.

Jeder Typ bezeichnet eine Menge von Werten. Einer Variable dieses Typs kann dann ein Wert zugeordnet werden.

Im folgenden werden wir uns nur mit den internen Datenstrukturen beschäftigen. Diese Datenstrukturen werden im Hauptspeicher gehalten und haben somit höchstens die Lebensdauer des zugehörigen Programms. Bezüglich der externen Datenstrukturen (Datentyp File) verweisen wir auf einschlägige Literatur.

Folgende Datentypen wollen wir besprechen:

- einfache, elementare Datentypen (SimpleType)
- Feld-Typen (ArrayType)
- Verbund-Typen (RecordType)
- dynamische Datentypen (PointerType)

6.1 Einfache, elementare Datentypen

Die einfachen, unstrukturierten (oder: elementaren) Datentypen lassen sich in die Ordinaltypen und in die reellen Typen aufteilen (siehe Abbildung 6.1). Grob gesagt bezeichnen Ordinaltypen Mengen, die man aufzählen kann (was ja bei den reellen Zahlen nicht der Fall ist). Eine andere Aufteilung der einfachen, unstrukturierten Datentypen ergibt sich dadurch, ob ein solcher Typ standardmäßig zur Verfügung steht oder erst vom Programmierer spezifiziert werden muss. Zur ersten Sorte gehören dann nämlich char, int, float und double, und zur zweiten Sorte gehören die Aufzählungstypen.

Im folgenden gehen wir die Typen anhand der ersten Aufteilung durch, d. h. bezüglich der Abbildung 6.1 von links nach rechts.

6.1.1 Integer-Typen

Diese Typen bezeichnen jeweils einen Teilbereich der ganzen Zahlen. Die Grundtypen heißen **`char`** und **`int`**.

Typ `char`

`char` ist ein Datentyp, dessen Werte die ganzen Zahlen zwischen −128 und +127 umfassen. Für die Speicherung dieser Werte benötigt man ein Byte (standardmäßig **`signed char`**). Man kann auch **`unsigned char`** spezifizieren; in diesem Fall ist der Zahlenbereich 0 bis 255. Da die druckbaren Zeichen auf einem Computer ebenfalls mit einem Byte codiert werden, wird der Datentyp **`char`** auch zur Behandlung von Zeichen (Characters) verwendet und hat daher auch seinen Namen.

Ein gebräuchlicher Code, der jedem darstellbaren Zeichen eine natürliche Zahl zwischen 0 und +127 zuordnet, ist der ASCII-Code (American Standard Code for Information Interchange).

Beispiel 6.1. Verwendung des Datentyps **`char`** in *C*:

```
1 . . .
2 const c = 'A';
3 char d;
4
5 d = c;
6 d = '+';
7 d = 12;    /* !!! */
8 . . .
```

□

Zu beachten ist, dass **`char`** intern ein *numerischer* Datentyp ist.

Operationen: Alle Operationen für ganze Zahlen, siehe dazu Datentyp `int` (nächster beschriebener Datentyp).

ASCII-Tabelle: Den im ASCII-Code verfügbaren Zeichen sind die ihnen entsprechenden (dezimalen) Ordinalzahlen gegenübergestellt.

0	NUL	16	DLE	32		48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Beispiel 6.2. Das folgende Programmfragment liest eine Folge s von Symbolen von der Eingabe und wandelt s' in eine ganze Zahl (gespeichert auf x) um, wobei s' der längste Präfix von s ist, der nur aus Ziffern besteht.

```
1  . . .
2  char ch;
3  int x;
4
5  int main()
6  { x = 0;
7    scanf("%c", &ch);
8    while (('0' <= ch) && (ch <= '9'))
9    { x = 10 * x + ch - '0';
10     printf("x = %d\n", x);
11     scanf(" %c", &ch);
12   }
13   return 0;
14 }
```

Bei diesem Programmbeispiel ergibt sich für die Eingabefolge `027b` die folgende Rechnung bzw. Ausgabe:

Taste	ch	x	Ausgabe
0	'0'	$10 * 0 + 48 - 48 = 0$	x = 0
2	'2'	$10 * 0 + 50 - 48 = 2$	x = 2
7	'7'	$10 * 2 + 55 - 48 = 27$	x = 27
b	'b'		

□

Typ `int`

Der Typ **`int`** kann mit den Typmodifikatoren **`short`** bzw. **`long`** modifiziert und zusätzlich mit **`unsigned`** bzw. **`signed`** versehen werden, so dass sich insgesamt die folgenden Integer-Typen ergeben:

```
signed short int
unsigned short int
signed int
unsigned int
signed long int
unsigned long int
```

Dabei wird **`signed`** verwendet, wenn weder **`signed`** noch **`unsigned`** angegeben wurde (default-Wert). Werden Modifikatoren verwendet, darf die Angabe des Grundtyps **`int`** auch weggelassen werden. Der Compiler nimmt dann automatisch **`int`** an. Demzufolge bezeichnen **`signed short int`**, **`short int`**, **`signed short`** und **`short`** ein und denselben Typ. Aus Gründen der Lesbarkeit sollte man jedoch auf diese Varianten verzichten.

Der Datentyp **`short int`** belegt 2 Byte Speicherplatz (d.h. 2^{16} Zahlen sind darstellbar).

```
MIN_short_int = -32768
MAX_short_int =  32767
```

Die Typen **`int`** und **`long int`** besitzen den gleichen Wertebereich `MIN_int` bis `MAX_int` mit

```
MIN_int = -2147483648
MAX_int =  2147483647
```

Der Speicherplatzbedarf dieser Datentypen ist implementationsabhängig. Dabei belegt der Datentyp **`long int`** 4 Byte (bzw. 8) Speicherplatz, d.h. 2^{32} Zahlen sind darstellbar, während der Speicherplatzbedarf des Datentyps **`int`** architekturabhängig ist und je nach Datenbusbreite 2, 4 oder 8 Byte betragen kann (2 Byte bei 16-Bit-Architekturen, dann aber nur Wertebereich von **`short int`**). Sollen Programme portierbar sein, sollte man deshalb auf den Datentyp **`int`** verzichten und **`short int`** (bzw. **`short`**) oder **`long int`** (bzw. **`long`**) verwenden. In unseren Beispielen werden wir aus Gründen der Übersichtlichkeit jedoch nur **`int`** verwenden.

Deklaration von Integer-Konstanten: Der Wert einer Integer-Konstanten kann durch Verwendung der Suffixe **L** oder **l** für **`long`** und **U** oder **u** für **`unsigned`** in beliebiger Schreibweise und Reihenfolge angegeben werden, für die Speicherung werden jedoch in *C* (im Gegensatz zu C^{++}) immer 4 Byte verwendet.

Beispiel 6.3. Deklaration von Integer-Konstanten:

```
const a = 4, b = 3865;    /* Werte sind vom Typ '(signed) int' */
const c = 48726;          /* (signed) long int */
const c = 48726u;         /* unsigned int */
const d = -124L;          /* (signed) long int */
const d = 324528375u;     /* unsigned long int */
```
□

Operatoren:

- **Arithmetische Operationen** (liefern bei ganzzahligen Operanden *immer* ein ganzzahliges Ergebnis):
  - **unäre Operatoren**:
    `++` (Inkrementierung)
    `--` (Dekrementierung)
  - **binäre Operatoren**:
    - **additive Operatoren**:
      `+` (Addition)
      `-` (Subtraktion)
    - **multiplikative Operatoren** (besitzen einen höheren Rang als additive Operatoren):
      `*` (Multiplikation)
      `/` (Ergebnis der ganzzahligen Division)
      `%` (Rest der ganzzahligen Division)
    - **bitweise Operationen** (besitzen einen niedrigeren Rang als additive Operationen):
      `<<`, `>>` (bitweises Verschieben nach links bzw. nach rechts)
      `&`, `|`, `^` (bitweise Konjunktion, Disjunktion und Alternative)
- **Vergleichsoperationen**: `==`, `<`, `>`, `<=`, `>=`, `!=`
  Vergleichsoperationen liefern einen Wahrheitswert.
- **Boolesche Operationen** (s. u.)

Die üblichen Integer-Operationen müssen wir jetzt durch Operationen auf Booleschen Ausdrücken erweitern. Da in *C* der Datentyp BOOLEAN nicht existiert, werden die Wahrheitswerte „true“ und „false“ durch die Integer-Werte 1 bzw. 0 repräsentiert. Boolesche Ausdrücke sind somit Verknüpfungen von Integer-Typen; ist der Wert eines Operanden ungleich 0, dann wird er als 1 („true“) gewertet. Das Ergebnis einer Booleschen Operation ist dann immer 0 oder 1 („false“ oder „true“).

Es stehen folgende Operationen für *logische* Verknüpfungen zur Verfügung:

`!`	Negation
`&&`	Konjunktion (UND-Verknüpfung)
`||`	Disjunktion (ODER-Verknüpfung)

Insbesondere gilt: $\texttt{!x} = \begin{cases} 0 & \text{wenn } \texttt{x} \neq 0 \\ 1 & \text{wenn } \texttt{x} = 0 \end{cases}$

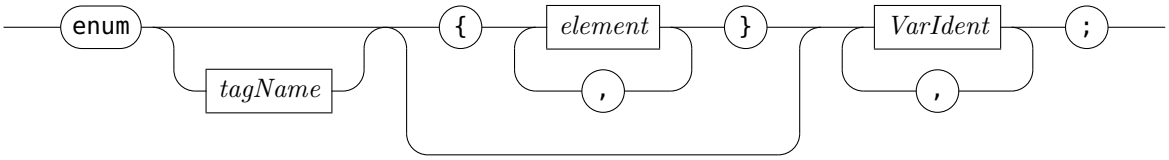
Beispiel 6.4. Operationen auf **`int`**-Zahlen in *C*:

```
1  int a=4; int b=7; int c=0; int d=-3;
2  int e;
3
4  e = !d;      /* e erhält den Wert 0 */
5  e = !a;      /* e erhält den Wert 0 */
6  e = !c;      /* e erhält den Wert 1 */
7  e = !!d;     /* e erhält den Wert 1 */
8  e = a && b;  /* e erhält den Wert 1 */
9  e = a && c;  /* e erhält den Wert 0 */
10 e = !a && c; /* e erhält den Wert 0,
11                 Negation hat Vorrang */
12 e = a || c;  /* e erhält den Wert 1 */
```
□

6.1.2 Aufzählungstypen (Enumerate)

Die zulässigen Werte eines Aufzählungstyps sind Bezeichner (*Elemente*) und werden explizit aufgezählt. In *C* werden dabei intern lediglich Integer-Konstanten (beginnend bei 0) deklariert. Variablen des Enumerate-Typs sind „ganz normale“ Integer-Variablen, denen jedes Element des festgelegten Typs, aber auch jeder andere Integer-Wert zugewiesen werden kann. Das folgende Syntaxdiagramm gilt für die Deklaration von Enumerate-Variablen, die durch *VarIdent* bezeichnet werden:

EnumType



tagName kann verwendet werden, um dem Enumerate-Typ eine Bezeichnung zuzuordnen. Diese kann später zur Deklaration von Variablen verwendet werden. Beachte: Die Umgehung von {...} im Diagramm *EnumType* darf nur genutzt werden, wenn bereits der “Geradeausweg“ für denselben *tagName* ausgeführt wurde.

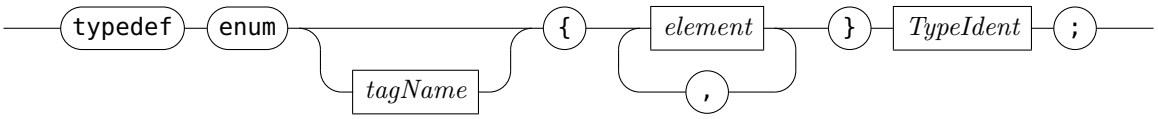
Beispiel 6.5. Deklaration von Enumerate-Variablen in *C*:

```
enum {schwarz, weiss} f;
enum colour {rot,gelb,blau} f1,f2;
enum colour farbe;
```

Die Variable `farbe` hat den selben Typ wie die Variablen `f1` und `f2`. □

Bei Typ-Deklarationen bezeichnet *TypeIdent* den deklarierten Typ. Ist ein *tagName* angegeben, so kann dieser in Verbindung mit **`enum`** benutzt werden, um z. B. Konstanten oder Variablen zu deklarieren. Meist wird aber *TypeIdent* als Typ-Bezeichner verwendet; in diesem Fall kann auch *tagName* entfallen:

EnumTypeDeclaration



Beispiel 6.6. Verwendung von Aufzählungstypen in *C*:

```
1  . . .
2  typedef enum Tage {Mo, Di, Mi, Do, Fr, Sa, So} Wochentage;
3  const enum Tage Sonntag = So;  /* oder: const Wochentage Sonntag = So; */
4                                 /* aber auch: const Sonntag = So;
5                                         bzw.: const int Sonntag = So; */
6  Wochentage anyday, f;          /* oder auch: enum Tage anyday, f; */
7  int x;
8
9  . . .
10
11 anyday = Di;
12 anyday++;           /*  (anyday == Mi) */
13 f = anyday;         /*  (f == Mi)  */
14 x = Do;             /*  (x == 3, automatische Typkonversion von Wochentage zu int) */
15 f = x + Mi;         /*  (f == Sa) */
16 f = Do + Sa         /*  (f == 8) */
17 f = (Wochentage) 4; /*  (f == Fr) */
18 f = 12;             /*  (f == 12) */
19 Sonntag = x;        /*  Falsch! Sonntag als Konstante deklariert! */
```

□

Operationen: alle Operationen, die für Integer-Typen erlaubt sind

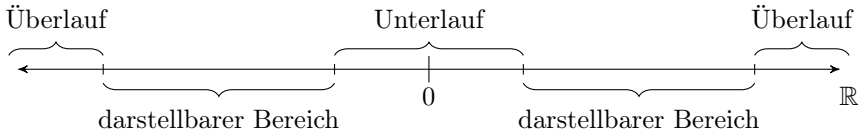
6.1.3 Reelle Zahlen

Dieser Typ bezeichnet eine endliche Teilmenge von $\mathbb{R}$. Die Grundtypen sind **`float`** und **`double`**, **`double`** kann zu **`long double`** modifiziert werden.

Beispiel 6.7. Darstellungsweisen für Gleitkommazahlen:

- `37.52`
- `0.0`
- `7.35E13`
- `-0.375E-7`
- $\pm\texttt{0.}\underbrace{a_1 \ldots a_n}_{\text{Mantisse}}\texttt{E}\pm\underbrace{b_1 \ldots b_k}_{\text{Exponent}}$ mit $a_1 \neq 0, n \geq 1, k \geq 1, a_i, b_j \in \{0, \ldots, 9\}$ □

Die Genauigkeit der Darstellung einer reellen Zahl ist begrenzt, weil nur endlich viele Werte dargestellt werden können. Insbesondere gibt es den Unterlauf- und den Überlaufbereich.



Darstellung, Wertebereich und Genauigkeit:

Typ	Byte	Min.	Max.	Genauigkeit
float	4	±3.4E-38	±3.4E38	≥ 6 Ziffern
double	8	±1.7E-308	±1.7E308	≥ 10 Ziffern
long double	10	±1.2E-4932	±1.2E4932	≥ 10 Ziffern

Genauigkeit bezieht sich hier auf die Anzahl der signifikanten Ziffern der Mantisse einer reellen Dezimalzahl. Sollten z. B. durch eine Operation über die signifikante Ziffernanzahl eines Typs hinausgehende Ziffern entstanden sein, so werden diese bei der Zuweisung zur entsprechenden Variablen abgeschnitten.

Gleitkommakonstanten:

Gleitkommakonstanten ohne Suffix sind vom Typ **`double`**, jedoch kann auch hier durch Verwendung der Suffixe **L** bzw. **l** für **`long`** und **F** bzw. **f** für **`float`** ein anderer Typ festgelegt werden.

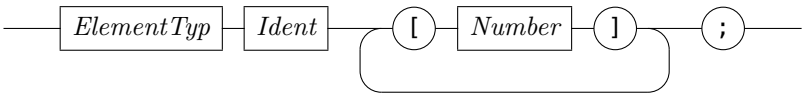
6.2 Strukturierte Datentypen

Oft lassen sich die Objekte, mit denen man bei einem konkreten Problem umgeht, nicht auf einfache Weise als Zahlen darstellen, weil sie strukturiert sind. Für diese Situationen stehen in *C* strukturierte Datentypen zur Verfügung. Dazu zählt man das Feld und den Verbund.

6.2.1 Feld (Array)

Ein Feld ist eine Folge (endlicher Länge) von Daten (Komponenten) desselben Typs (*ElementTyp*). Der Zugriff auf Komponenten des Feldes erfolgt über Indizes, wobei jeder Index ein Element der (endlichen) Indexmenge dieser Folge sein muss. Die Deklaration *einer* Feldvariablen wird syntaktisch durch folgendes Syntaxdiagramm festgelegt:

ArrayType



ElementTyp bezeichnet dabei den Typ jeder Komponente des Feldes, *Number* die Anzahl der Komponenten und *Ident* den Bezeichner der Feldvariablen. Die Zählung des Index beginnt grundsätzlich bei 0.

Beispiel 6.8. Deklaration von Feldvariablen:

```
int Feld[4];
char Letter[6];
```

Dann können z. B. folgende Zuweisungen im Programm auftreten:

```
Feld[2] = 7;
Letter[0] = 'A';
```

Die Deklarationen

```
int Feld[4] = {2, 7, 0, -4};
char Letter[6] = {'A', 'B', 'C', 'D', 'E', 'F'};
```

erzeugen initialisierte Feldvariablen. □

Beispiel 6.9. Deklaration von Feldkonstanten:

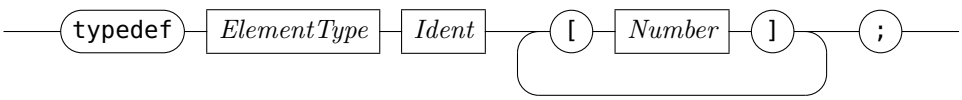
```
const int Feld[4] = {2, 7, 0, -4};
const char Letter[6] = {'A', 'B', 'C', 'D', 'E', 'F'};
```

Dabei können **int** und **char** auch weggelassen werden. □

Deklaration von Feldtypen:

Das Syntaxdiagramm für die Deklaration von Array-Typen haben wir im Kapitel 3 bereits beispielhaft verwendet:

ArrayTypeDeclaration



Beispiel 6.10. Deklaration von Array-Typen in *C*:

```
typedef int array1[4];
typedef char array2[6];

array1 Feld;       /* Feld ist eine Variable des Typs array1 */
array2 Letter;     /* Letter ist eine Variable des Typs array2 */
```

□

Beispiel 6.11. Es soll die Menge aller zweidimensionalen Felder deklariert werden, bei denen der Index in der ersten Dimension aus der Indexmenge `[0..99]` und der zweite Index aus der Indexmenge `{rot, gruen, blau}` gewählt werden soll; die Feldeinträge sollen den Typ **float** haben.

```
enum farben {rot, gruen, blau} color;
   /* color ist eine Variable des Typs enum farben */
float x[100][3];
. . .
color = gruen;
x[87][color] = (float) 3.7;
x[45][rot] = (float) -46.4E-12;
```
□

Beispiel 6.12. Als zweites Beispiel zeigen wir ein Programm, mit dessen Hilfe zwei Matrizen miteinander multipliziert werden können. Seien $A(m,n)$ und $B(n,q)$ zwei Matrizen. Genauer:

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nq} \end{pmatrix}.$$

Durch die Multiplikation entsteht die (m,q)-Matrix C:

$$C = \begin{pmatrix} c_{11} & \cdots & c_{1q} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mq} \end{pmatrix} \text{ mit } c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} .$$

Wir verdeutlichen die Definition der Matrixmultiplikation an dem folgenden Beispiel, bei dem eine 2×3-mit einer 3×2-Matrix multipliziert wird. Das Ergebnis ist dann eine 2×2-Matrix.

$$\begin{pmatrix} 5 & 0 & -2 \\ 1 & -3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 & -1 \\ 5 & 3 \\ 4 & -6 \end{pmatrix} = \begin{pmatrix} 5\cdot 2 + 0\cdot 5 + (-2)\cdot 4 & 5\cdot(-1) + 0\cdot 3 + (-2)\cdot(-6) \\ 1\cdot 2 + (-3)\cdot 5 + 4\cdot 4 & 1\cdot(-1) + (-3)\cdot 3 + 4\cdot(-6) \end{pmatrix}$$
$$= \begin{pmatrix} 10 + 0 + (-8) & -5 + 0 + 12 \\ 2 + (-15) + 16 & -1 + (-9) + (-24) \end{pmatrix} = \begin{pmatrix} 2 & 7 \\ 3 & -34 \end{pmatrix}$$

Der entsprechende Programmausschnitt könnte wie folgt aussehen:

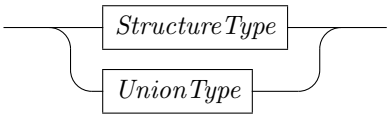
```
1  . . .
2  const m = 5, n = 3, q = 4;
3
4  float a[5][3],    /* m, n ; in C nur konstante Ausdruecke zugelassen! */
5        b[3][4],    /* n, q */
6        c[5][4];    /* m, q */
7  int i, j, k;
8  float s;
9
10 for (i = 0; i < m; i++)
11   for (j = 0; j < q; j++)
12   { s = 0;
13     for (k = 0; k < n; k++)
14       s = s + a[i][k] * b[k][j];
15     c[i][j] = s;
16   }
17 . . .
```

Der algorithmische Aufwand ist von der Ordnung $O(n * m * q)$, also kubisch, da $O(n * m * q) = O(p^3)$ mit z. B. $p = \max\{n, m, q\}$. Es gibt aber auch bessere Multiplikationsalgorithmen; z. B. der nach seinem Erfinder (oder: Konstrukteur) benannte Strassen-Algorithmus: $O(p^{2.81})$. Der Weltrekord bei der Matrixmultiplikation liegt bei $O(p^{2.373})$. □

6.2.2 Verbund (Structure, Union)

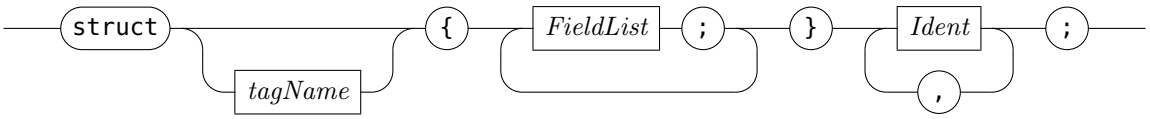
Im Gegensatz zum Feld können bei einem Verbund die verschiedenen Komponenten unterschiedliche Typen haben. Ein typisches Beispiel ist das Konzept der Person mit ihren verschiedenen Attributen. In C können zwei unterschiedliche Verbund-Typen deklariert werden:

RecordType

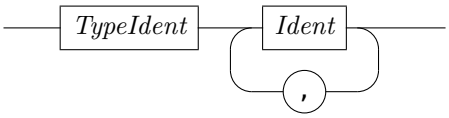


Beim Structure-Typ wird für jede in *FieldList* angegebene Definition Speicherplatz innerhalb der Struktur angelegt. Der gesamte Platzbedarf für eine Structure-Variable ergibt sich damit aus der Summe des Platzbedarfs aller Datenfelder. Der Zugriff auf die Datenfelder (die in *FieldList* angegebenen Elemente) einer Structure-Variable erfolgt durch den `.`-Operator: Ist z. B. `structVar` eine solche Variable und `element` ein Datenfeld, welches für ihren Strukturtyp definiert wurde, dann kann man auf dieses Feld durch den Ausdruck `structVar.element` zugreifen.

StructureType



FieldList



Wenn *tagName* weggelassen wird, erhält man unbenannte oder anonyme Strukturen, d. h. man kann zwar Variablen (*Ident* in *StructureType*) des unbenannten Structure-Typs deklarieren, kann aber den Typ später nicht wieder verwenden, um weitere Variablen des gleichen Typs oder Funktionsparameter dieses Typs zu deklarieren.

Beispiel 6.13. In der Variablendeklaration **`struct`** `{ ... } a, b, c;` gehören die Variablen `a`, `b` und `c` dem unbezeichneten Typ **`struct`** `{ ... }` an. □

Sollen später weitere Variablen des gleichen Typs, z. B. `x, y, z`, deklariert werden, so *muss* ein *tagName* verwendet werden, der dann in Verbindung mit dem Schlüsselwort **`struct`** den Typ bezeichnet.

Beispiel 6.14.

```
struct beispiel { ... } a, b, c;
struct beispiel x, y, z;
```

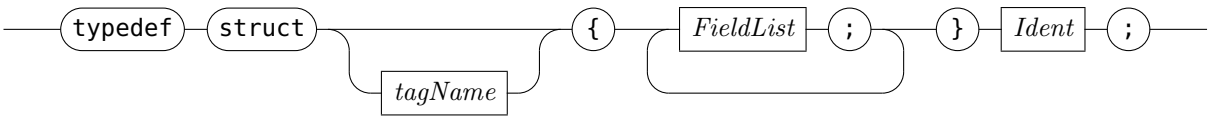
Hier ist `beispiel` der *tagName*. Die Variablen `x, y, z` sind typgleich mit `a, b, c`. □

Beispiel 6.15. **`struct`** `beispiel_2 {`**`int`** `k, l;` **`float`** `m;} p, q;`
Hier sind `p,q` Variablen des Strukturtyps **`struct`** `beispiel_2`. Der Strukturtyp **`struct`** `beispiel_2` enthält die drei Komponenten `k`, `l` und `m` vom Typ **`int`**, **`int`** und **`float`**. □

Innerhalb der Structure-Deklaration (d. h. in *FieldList*) darf der eben deklarierte Structure-Typ *nur* für Pointer verwendet werden. In diesem Fall *muss* auf *tagName* Bezug genommen werden; *tagName* muss also vergeben worden sein.

Wird ein Structure-Typ mittels einer *Typdeklaration* deklariert, so kann als Bezugnahme auf diesen Structure-Typ der *tagName* (falls vorhanden) in Verbindung mit **`struct`** oder der am Ende der Deklaration stehende Typbezeichner (*Ident*) verwendet werden. Beide bezeichnen den deklarierten *Typ* und sind gleichwertig (siehe auch das folgende Beispiel).

StructureTypeDeclaration



Beispiel 6.16. Gegeben sei die folgende Typdeklaration:

```
typedef struct beispiel_3 { ... } mytype;
```

Dann haben die folgenden beiden Zeilen dieselbe Bedeutung:

```
struct beispiel_3 x, y, z;
mytype x, y, z;
```

□

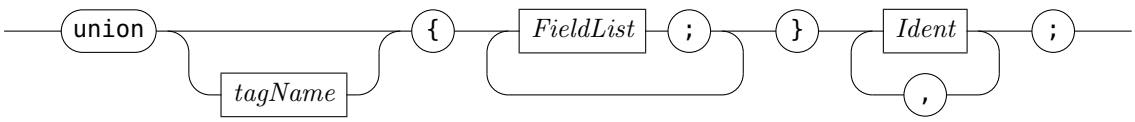
Beispiel 6.17. Verwendung von strukturierten Datentypen:

```
1  /* Beispiel für structure */
2  . . .
3  typedef struct personal { char name[30];
4                            enum {m, w, i} geschlecht;
5                            enum {verh, led, gesch, verw} famstand;
6                            unsigned int gehalt;
7                            struct {short int tag, monat, jahr;} gebdat;
8                          } person;
9
10 person egon;
11 . . .
12 egon.gehalt = 8000;
13 strcpy(egon.name, "Maier"); /* kopiert die Zeichen (ASCII-Code) der String-Kon-
14                               stanten "Maier" nacheinander auf Adressen ab der
15                               Adresse der Variablen egon.name und schreibt auf
16                               die nächstfolgende Adresse den Wert 0 */
17 egon.geschlecht = m;
18 egon.famstand = led;
19 egon.gebdat.tag = 22;
20 egon.gebdat.monat = 12;
21 egon.gebdat.jahr = 1960;
22 . . .
```

□

Beim Union-Typ werden die Einträge der *FieldList* als exklusive Alternativen betrachtet, von denen immer nur genau eine ausgewählt wird. Deshalb wird auch nur Speicherplatz entsprechend des Bedarfes des größten der Datenfelder aus *FieldList* angelegt. Bei Verwendung der Union-Variablen wird der Inhalt dann entsprechend interpretiert.

UnionType



Für Typdeklarationen, für die Verwendung des .-Operators zum Zugriff auf Datenfelder sowie für die Verwendung des *tagName* gilt das Gleiche wie beim Structure-Typ.

Beispiel 6.18. Wir zeigen ein Beispiel für die Verwendung von Union-Typen. Hier ist Vorsicht geboten, da z. B. gilt, dass `auto1.eigenschaft.sitzplaetze` und `auto1.eigenschaften.zuladung` die selbe Speicherstelle adressieren und die dort abgelegte Bitfolge einmal als **`int`**- und einmal als **`float`**-Datum interpretiert wird. Von jeder Variable des Typs `kfz` sollte deshalb höchstens eins der Felder `sitzplaetze`, `vmax` und `zuladung` benutzt werden!

```
1  /* Beispiel für union */
2  . . .
3  typedef struct kfz_typ { char hersteller[30];
4                           enum {pkw, bus, lkw} art;
5                           float preis;
6                           union { short int vmax;
```

```
7                                   short int sitzplaetze;
8                                   float zuladung; } eigenschaft; } kfz;
9
10 kfz auto1, auto2, auto3;
11 . . .
12 auto1.art = pkw;
13 auto1.eigenschaft.vmax = 180;
14
15 auto2.art = bus;
16 auto2.eigenschaft.sitzplaetze = 45;
17
18 auto3.art = lkw;
19 auto3.eigenschaft.zuladung = 25.5f;
20 . . .
```

□

6.3 Dynamische Datentypen

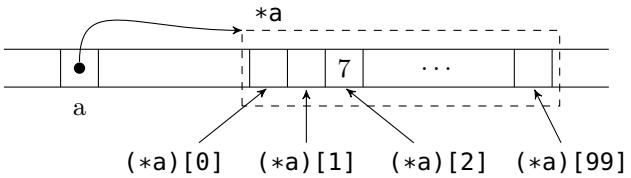
Durch die Deklaration von Variablen lässt sich der Programmierer Speicherplatz bereitstellen. Dieser Speicherplatz ist in seiner Größe bis zum Ende des Programmablaufs (oder dem Verlassen der entsprechenden Funktion) fest. Manchmal lässt sich mit dieser eingeschränkten Technik ein Algorithmus aber nur schwer entwerfen, beispielsweise wenn der Umfang der zu verarbeitenden Daten am Anfang noch nicht feststeht. Deshalb wollen wir jetzt dem Programmierer erlauben, das Anlegen und die Freigabe von Speicherplätzen durch Programmcode selbst zu veranlassen, und dadurch die Strukturierung des Speicherbereichs dynamisch (d. h. zur Laufzeit) verändern zu können. Dazu führen wir das Konzept der Zeigervariablen (Pointer) ein.

6.3.1 Zeigervariable

Werte einer Zeigervariablen sind Verweise (Referenzen, Adressen) auf andere Speicherbereiche. Zeigertypen und Zeigervariablen werden im globalen Deklarationsteil eines Programms oder innerhalb einer Funktion deklariert.

Beispiel 6.19. Verwendung von dynamisch bereitgestelltem Speicher:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef int feld[100];
5  typedef feld *P_feld;
6  P_feld a;
7
8  int main()
9  {
10   a = (P_feld) malloc(sizeof(feld));
11   . . .
12   (*a)[2] = 7;
13   . . .
14   free(a);
15   . . .
16 }
```



□

In unserem Beispiel bezeichnet `P_feld` einen Zeigertyp. Die Programmvariable `a` ist vom Typ `P_feld`, und sie bezeichnet einen Speicherplatz, der als Wert die Anfangsadresse eines Speicherbereichs - wir sprechen von einem Zeiger auf diesen Speicherbereich - aufnehmen kann. Dieser Speicherbereich ist ein Feld mit 100 Plätzen und wird durch `*a` bezeichnet. Die einzelnen Plätze werden (wie üblich bei Feldern) durch `(*a)[i]` mit $0 \leq i \leq 99$ bezeichnet. Also bezeichnet `(*a)[2]` den dritten Speicherplatz von `*a`.

Die Deklaration einer Zeigervariablen bewirkt nur das Anlegen eines Speicherplatzes für die Zeigervariable, *nicht* aber das Anlegen eines Speicherbereichs vom Zieltyp der Zeigervariable. Diese Speicherbereitstellung geschieht dynamisch durch den Ausdruck

```
malloc(size)
```

Um `malloc` zu verwenden, muss mit `#include <stdlib.h>` die Standardbibliothek `stdlib` eingebunden werden. `malloc` reserviert so viele Bytes an Speicherplatz, wie als Argument angegeben wird. Man sollte also *mindestens* so viele Bytes angeben, wie für den dynamischen Typ benötigt wird.

Durch die Zuweisung

```
a = (P_feld) malloc(sizeof(feld));
```

wird also

- Speicherbereich der Größe `sizeof(feld)` angelegt,
- durch den cast-Ausdruck `(P_feld)` dem zunächst typfreien, angelegten Platz der Typ `P_feld` zugeordnet und
- in den Speicherplatz mit der Bezeichnung `a` die Adresse des neuen, getypten Speicherbereichs eingetragen.

Die Speicherfreigabe erfolgt über die Anweisung

```
free(a);
```

Diese Anweisung gibt den für das Feld angelegten Speicherbereich wieder frei (nicht aber die Variable `a`!), d. h. auf `*a` darf danach nicht mehr zugegriffen werden. Der freigegebene Speicherbereich kann damit für mit `malloc` neu erzeugte Variablen wieder verwendet werden, d. h. der Speicher wird dynamisch verwaltet.

Der Speicherbereich, der durch `malloc` angelegt wird, wird *nicht* auf dem Laufzeitkeller, sondern in einem gesonderten Speicherbereich, der Halde (heap), abgelegt und kann deshalb z. B. auch über die aktive Phase der Funktion, die den `malloc`-Befehl enthält, hinaus benutzbar bleiben.

Wir wollen hier ein Beispiel für den Zugriff auf Zeigervariablen angeben, weisen aber darauf hin, dass der verwendete Programmierstil (gemischter Zugriff auf lokale und globale Variablen, Änderung einer globalen Variablen direkt aus einer Funktion heraus) allgemein nicht angewendet werden sollte.

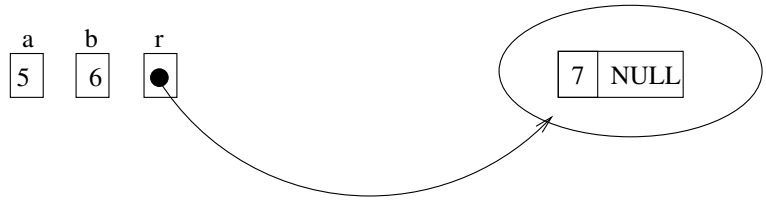
Beispiel 6.20. Zugriff auf Zeigervariablen:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct ele *zeiger;
5  typedef struct ele { int zahl;
6                       zeiger next;
7                     } element;
8
9  int a, b; zeiger r;
10
11 void A(int x, int y, int *z)
12 { int hilf; zeiger p;
13
14   hilf = (x + y) * *z;
15   p = (zeiger) malloc(sizeof(element));
16   p->zahl = hilf;      /* indirekter Zugriff, Dereferenzierung,  */
17   p->next = r;         /* gleichbedeutend mit: (*p).zahl = hilf; */
18   r = p;
19
```

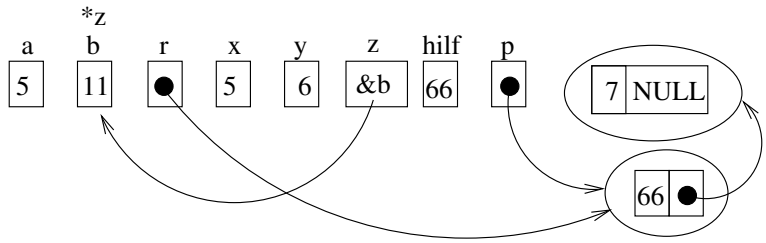
```
20    if (hilf < 100)
21    { *z = *z + 5;       /* z ist Referenzparameter, deshalb hier Zugriff auf den Wert
22                            mit *,                                                  */
23      A(x + y, 10, z);   /* aber hier rekursiver Aufruf von A mit drittem Parameter als
24                            Referenzparameter, also Übergabe einer Adresse, die in z
25                            gespeichert ist!                                        */
26      *z = 2 * *z;
27    }
28  }
29
30  int main()
31  { a = 5;
32    b = 6;
33    r = (zeiger) malloc(sizeof(element));
34    r->zahl = 7;         /* gleichbedeutend mit (*r).zahl = 7; */
35    r->next = NULL;
36    A(a, b, &b);
37    return 0;
38  }
```

Wir geben nun für dieses Programm ein Ablaufprotokoll an, aus dem die Struktur der Daten deutlich wird.

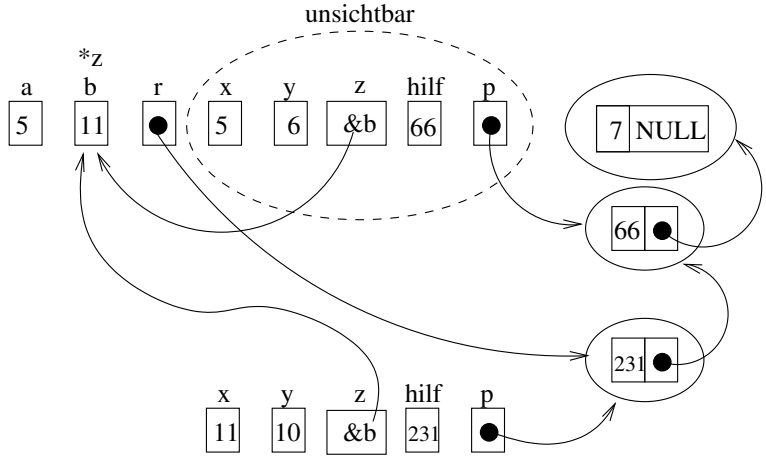
Im Hauptprogramm



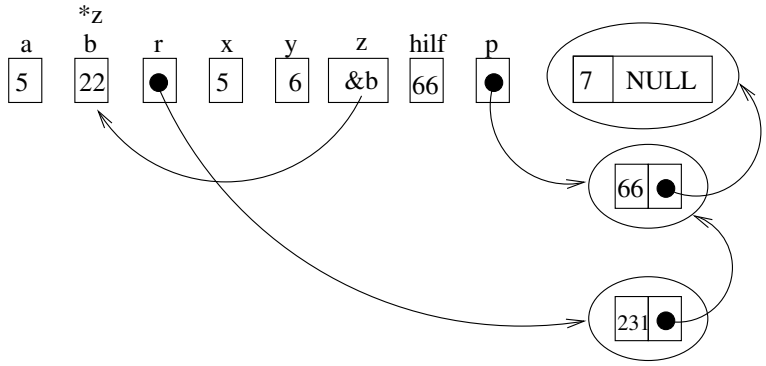
vor dem 2. Aufruf von A



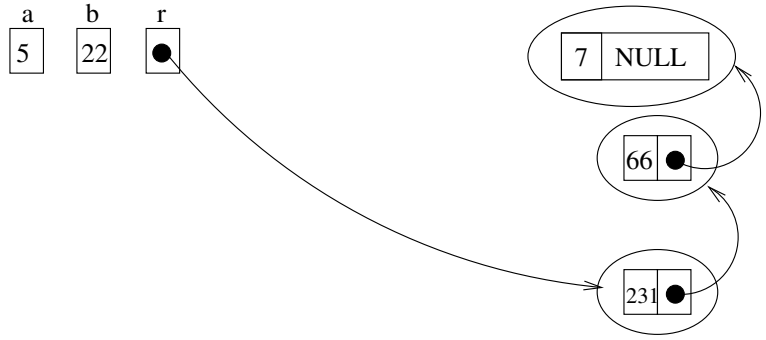
vor dem 1. Rücksprung



nach Verlassen des 2. Aufrufs von A



nach Verlassen des 1. Aufrufs von



□

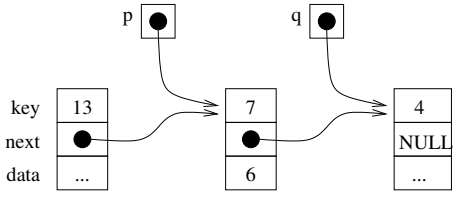
6.3.2 Einfach-verkettete Listen

Eine oft benutzte Datenstruktur, die mit Hilfe des Zeigerkonzepts realisiert werden kann, ist die einfach-verkettete Liste. Dabei enthält jedes Element der Liste neben dem Schlüsselwert (`key`) und evtl. zugehörigen Daten (`data`) einen Zeiger auf das nachfolgende Listenelement (`next`):

```
typedef struct nodeelem *Ptr;
typedef struct nodeelem { int key;
                          Ptr next;
                          int data;
                        } node;
```

Beispiel 6.21.

```
Ptr h,p,q; int n,i;

p->data = 6;
q = p->next;
```



□

Aufbau einer verketteten Liste

Das folgende Programmstück liest vier Integer-Zahlen ein und baut eine einfach-verkettete Liste aus vier Instanzen des Typs `node` auf, in denen jeweils in der ersten Komponente (key) die aktuell eingelesene Integer-Zahl gespeichert wird.

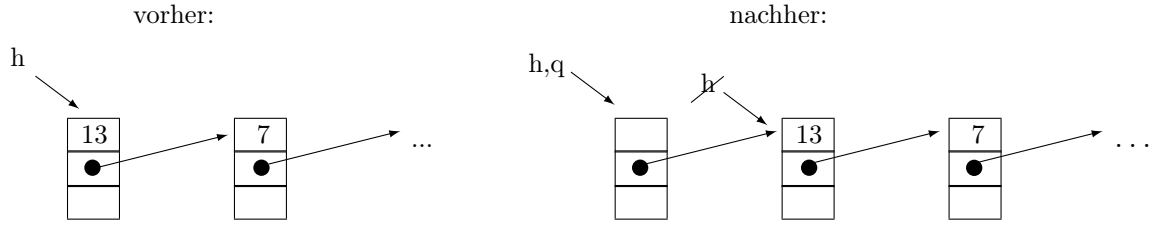
```
1  scanf("%d", &n);
2  q = (Ptr) malloc(sizeof(node));
3  h = q;                          /* h haelt den Listenanfang fest */
4  q->key = n;
5  q->next = NULL;
6  for (i = 1; i <= 3; i++)
7  { scanf("%d", &n);
8    p = (Ptr) malloc(sizeof(node));
9    p->key = n;
10   p->next = NULL;
11   q->next = p;
12   q = p;                        /* q zeigt auf das letzte Element */
13 }
```

Im folgenden wollen wir uns mit dem Einfügen und Löschen von Elementen in solche Listen beschäftigen.

Einfügen in die verkettete Liste ...

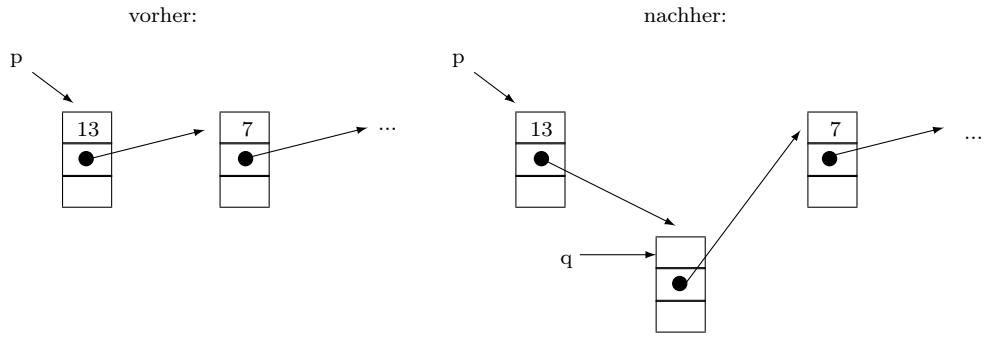
1. ... an den Anfang (auf den h zeigt):

```
1  q = (Ptr) malloc(sizeof(node));
2  q->next = h;
3  h = q;
```



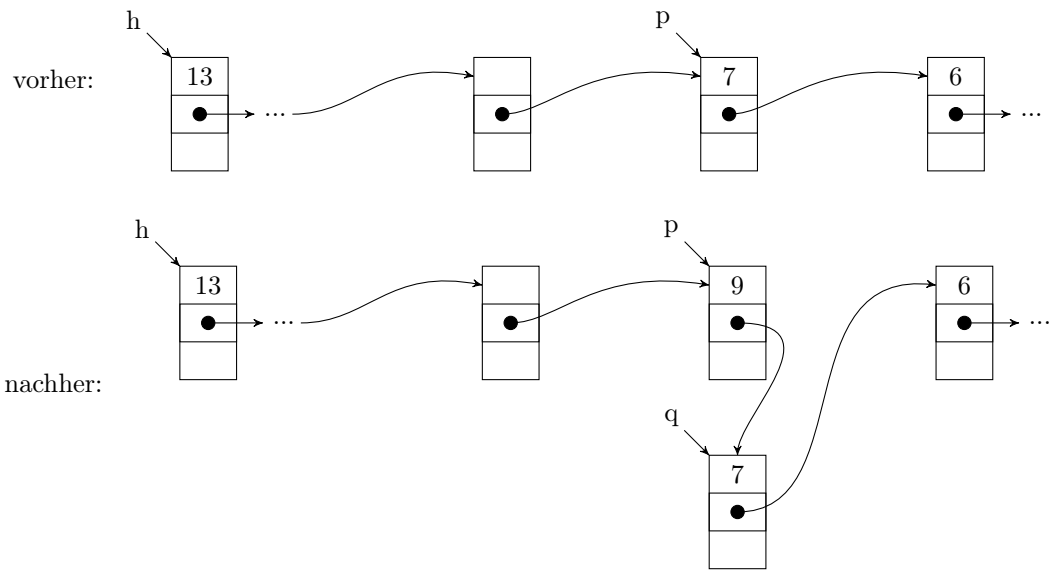
2. ... hinter ein durch einen Zeiger p bezeichnetes Objekt:

```
1  q = (Ptr) malloc(sizeof(node));
2  q->next = p->next;
3  p->next = q;
```



3. ... vor ein durch einen Zeiger p bezeichnetes Objekt:

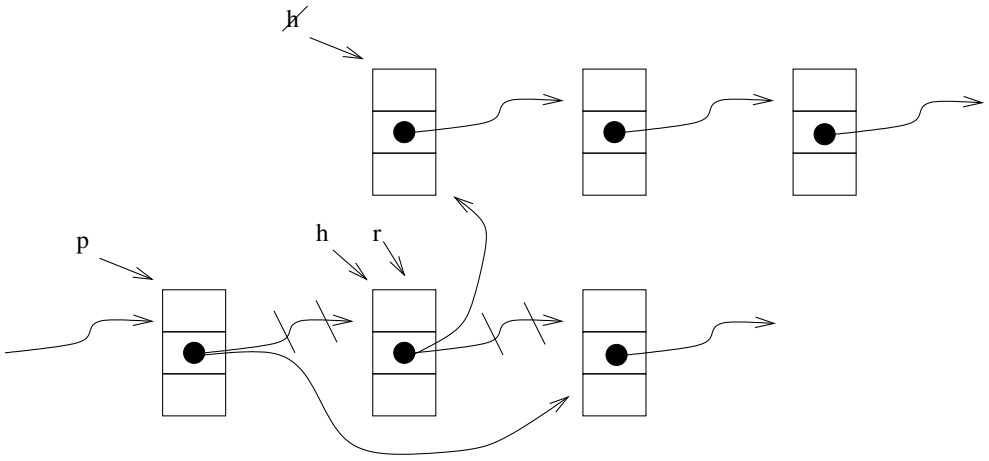
```
1  q = (Ptr) malloc(sizeof(node));
2  q->key = p->key;
3  q->next = p->next;
4  q->data = p->data;
5  p->next = q;
6  p->key = 9;      /* *p ist neues Datenobjekt */
```



Ausketten und Archivieren von Elementen

Annahme: Zwei Listen; aus der zweiten Liste den Nachfolger eines durch p bezeichneten Datenobjekts an den Anfang der ersten Liste, bezeichnet durch h, setzen.

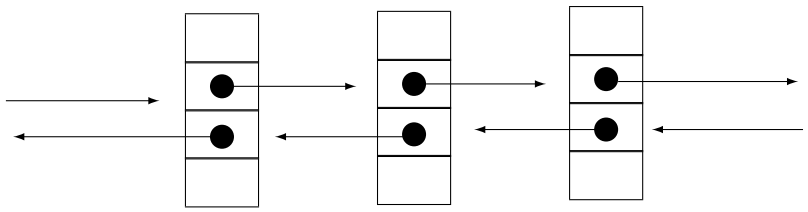
```
1 | r = p->next;
2 | p->next = r->next;
3 | r->next = h;
4 | h = r;
```



6.3.3 Doppelt-verkettete Listen

Zur Lösung mancher Probleme ist es nützlich, wenn die Daten nicht nur in einer Richtung durchlaufen werden können, sondern in beiden Richtungen. Dazu dient die doppelt-verkettete Liste.

```
typedef struct nodeelem *LPtr;
typedef struct nodeelem { int key;
                          LPtr next;
                          LPtr prev;
                          ... data;
                        } node;
LPtr  h, p, q;
```



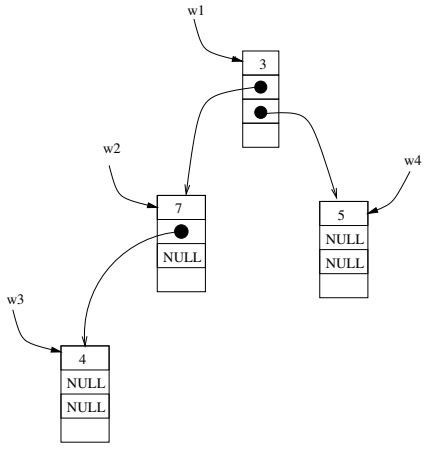
Aufbau einer doppelt-verketteten Liste

```
1  scanf("%d", &n);
2  q = (LPtr) malloc(sizeof(node));
3  q->key = n;
4  q->prev = NULL;              /* erstes Element hat keinen Vorgaenger */
5  h = q;                       /* Listenanfang                         */
6  for (i = 1; i <= 10; i++)
7  { scanf("%d", &n);
8    p = (LPtr) malloc(sizeof(node));
9    p->key = n;
10   q->next = p; p->prev = q;
11   q = p;
12   q->next = NULL;            /* letztes Element hat keinen Nachfolger */
13 }
```

6.3.4 Bäume

Oft haben die Daten eines Problems keine lineare, sondern eine verzweigte Struktur. Mit Hilfe von Bäumen lässt sich eine große Klasse von verzweigten Strukturen erfassen. Wir verwenden die folgende Typdefinition für Binärbäume, bei denen jeder Knoten einen Schlüsselwert (`key`) und Daten (`data`) enthält.

```
typedef struct nodeelem *BPtr;
typedef struct nodeelem { int key;
                          BPtr left, right;
                          ... data;
                        } node;
```



Beispiel 6.22. Wir wollen eine Funktion `hoehe` spezifizieren, welche die Berechnung der Höhe (Anzahl der Knoten auf dem längsten Pfad von der Wurzel zu einem Blatt) eines Baumes des Typs `node` berechnen soll. Die hierbei genutzte Funktion `max` berechnet aus zwei Integer-Zahlen das Maximum.

```
1  int hoehe(BPtr wz)
2  { int h1, h2;
3     /* label1 */
4    if (wz == NULL) return 0;
5    h1 = hoehe(wz->left);  /* $1 */
6     /* label2 */
7    h2 = hoehe(wz->right); /* $2 */
8     /* label3 */
9    return max(h1, h2)+1;
10 }
```

Beim Aufruf erhält die Funktion `hoehe` als aktuellen Parameter einen Zeiger auf die Wurzel des zu berechnenden Baumes. Durch rekursive Aufrufe auf dem jeweils linken und rechten Teilbaum werden deren Höhen `h1` bzw. `h2` ermittelt und dann der Wert $\texttt{max}\{\texttt{h1}, \texttt{h2}\} + 1$ als Höhe des gesamten Baumes zurückgeliefert. Der Ausstieg aus dieser Rekursion erfolgt in den Blattknoten; die unter den Blattknoten liegenden Teilbäume sind leer und haben per Definition die Höhe 0.
Wir wollen nun die Arbeitsweise (Speicherbelegungsprotokoll) von `hoehe` anhand des obigen Baumes und eines Programms `laengsterPfad` demonstrieren. Für die Darstellung des Speicherbelegungsprotokolls nutzen wir das bekannte Konzept des pulsierenden Speichers, werden uns aber nur auf die Funktion `hoehe` konzentrieren (label1 bis label3).

```
1  /*laengsterPfad*/
2  typedef struct nodeelem *BPtr;
3  typedef struct nodeelem { int key;
4                            BPtr left, right;
5                            ... data;         } node;
6  . . .
7  int hoehe(. . .){. . .}        /* Berechnet die Höhe eines binären Baumes.    */
8  void eingabe(BPtr *wz){. . .}  /* Realisiert Eingabe eines binären Baumes,
9                                    Zeiger auf die Wurzel wird zurückgeliefert. */
10 int main()
11 { int h; BPtr w;
12   eingabe(&w);
13   h = hoehe(w); /* $3 */
14   /* label4 */
15   . . .
16 }
```

Haltepunkt	RM	Umgebung											
		1	2	3	4	5	6	7	8	9	10	11	12
label1	3	h1	h2	*wz*									
		?	?	*w1*									
label1	1 : 3				h1	h2	*wz*						
		?	?	*w1*	?	?	*w2*						
label1	1 : 1 : 3							h1	h2	*wz*			
		?	?	*w1*	?	?	*w2*	?	?	*w3*			
label1	1 : 1 : 1 : 3										h1	h2	*wz*
		?	?	*w1*	?	?	*w2*	?	?	*w3*	?	?	*NULL*
label2	1 : 1 : 3							h1	h2	*wz*			
		?	?	*w1*	?	?	*w2*	0	?	*w3*			
label1	2 : 1 : 1 : 3										h1	h2	*wz*
		?	?	*w1*	?	?	*w2*	0	?	*w3*	?	?	*NULL*
label3	1 : 1 : 3						*wz*	h1	h2	*wz*			
		?	?	*w1*	?	?	*w2*	0	0	*w3*			
label2	1 : 3				h1	h2	*wz*						
		?	?	*w1*	1	?	*w2*						
label1	2 : 1 : 3							h1	h2	*wz*			
		?	?	*w1*	1	?	*w2*	?	?	*NULL*			
label3	1 : 3				h1	h2	*wz*						
		?	?	*w1*	1	0	*w2*						
label2	3	h1	h2	*wz*									
		2	?	*w1*									
label1	2 : 3				h1	h2	*wz*						
		2	?	*w1*	?	?	*w4*						
label1	1 : 2 : 3							h1	h2	*wz*			
		2	?	*w1*	?	?	*w4*	?	?	*NULL*			
label2	2 : 3				h1	h2	*wz*						
		2	?	*w1*	0	?	*w4*						
label1	2 : 2 : 3							h1	h2	*wz*			
		2	?	*w1*	0	?	*w4*	?	?	*NULL*			
label3	2 : 3				h1	h2	*wz*						
		2	?	*w1*	0	0	*w4*						
label3	3	h1	h2	*wz*									
		2	1	*w1*									

Der Vollständigkeit halber sei gesagt, dass nach Rückkehr von `hoehe` zu `main`, also z. B. bei label4, die Variable `h` den Wert 3 besitzt. Somit ist die Höhe unseres Beispielbaums gleich 3. □

Beispiel 6.23. Jetzt wollen wir beispielhaft für die in Kapitel 2 angegebene BNF-Definition (d.h. keine EBNF-Definition) $\mathcal{E}'$ die Definition eines dynamischen Datentyps angeben, mit dessen Hilfe Ableitungsbäume von $\mathcal{E}'$ beschrieben werden können.
Wir wiederholen noch einmal die entsprechenden Regeln und versehen dabei jede Regel mit einem Label ($r1$ bis $r6$).

$r1\colon S ::= CA$ $\quad r2\colon S ::= A$ $\quad r3\colon C ::= \texttt{c}C$ $\quad r4\colon C ::= \texttt{c}$ $\quad r5\colon A ::= \texttt{a}A\texttt{b}$ $\quad r6\colon A ::= \texttt{a}$

Dann können wir Ableitungsbäume von $\mathcal{E}'$ mit Hilfe von Pseudocode folgendermaßen definieren:

```
typedef enum {S, A, C, a, b, c} set_of_symbols;
typedef enum {r1, r2, r3, r4, r5, r6, no} set_of_rules;

typedef struct nodeelem *pointer_to_node;
typedef struct trailelem *pointer_to_trail;

typedef struct nodeelem { set_of_symbols label;
                          set_of_rules rule;
                          pointer_to_trail next; } node;

typedef struct trailelem { pointer_to_node item;
                           pointer_to_trail next; } trail;
```

Im Knoten `nodeelem` ist das Symbol (syntaktische Variable oder Terminalsymbol) enthalten. Außerdem ist dort die an diesem Knoten angewandte Regel gespeichert, welche `no` ist, falls am Knoten ein Terminalsymbol gespeichert ist. Über den Zeiger `next` erreicht man eine Liste von Zeigern, die jeweils auf einen Nachfolgerknoten zeigen. Natürlich enthält dieser Datentyp auch Werte, d. h. Bäume, die keine Ableitungsbäume sind. Abbildung 6.2 stellt einen Baum des Datentyps `node` graphisch dar. □

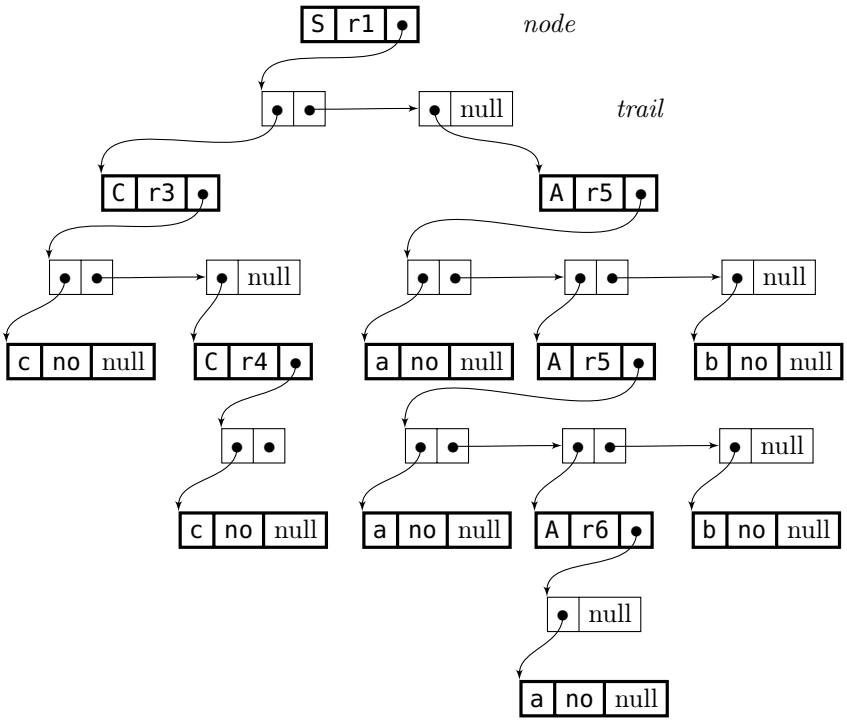


Abbildung 6.2: Ableitungsbaum

7 Modularisierungskonzept

Große Softwareprojekte machen es notwendig, dass zeitgleich ein Team von Mitarbeitern an der Erstellung der Software arbeitet. Wir wollen jetzt die Voraussetzungen dieses arbeitsteiligen Softwareentwurfs besprechen.

Jeder Mitarbeiter muss hier eine genau definierte Teilaufgabe erhalten, dann aber relativ eigenständig die Lösung gestalten können. Die Eigenständigkeit darf aber nur soweit gehen, dass gewährleistet bleibt, dass letztlich die Lösungen der Teilaufgaben als Bausteine des Gesamtprojektes in gewünschter Weise zusammenarbeiten. Da die Teilaufgaben i. allg. in sehr enger Beziehung zueinander stehen, d. h. auf gemeinsame Ressourcen (z. B. konsistente Datenbestände) zugreifen sollen, kommt der Bezugnahme auf gemeinsame Ressourcen eine besondere Bedeutung zu. Softwaretechniker bezeichnen die Gestalt dieser Bezugnahme als *Schnittstelle*; diese Schnittstelle ist wesentlicher Teil der definierten Teilaufgabe. Die Schnittstelle spezifiziert also insbesondere, welche äußeren Ressourcen die zugeordnete Teilaufgabe nutzen darf bzw. soll und in welcher Form das Ergebnis an die (Software-)Umgebung geliefert werden soll. Innerhalb dieser durch die Schnittstelle fixierten Rahmenbedingungen kann nun die Mitarbeiterin nach eigenem Ermessen die Lösung der Teilaufgabe gestalten. Oft ist es sogar erwünscht, dass diese Detaillösungen verborgen bleiben (*information hiding*).

Diese eben skizzierte Art des „Programmierens im Großen“ wird in *C* durch das Modulkonzept unterstützt. Wir haben bereits – allerdings unkommentiert und zwangsweise – von diesem Konzept Gebrauch gemacht. Bestandteil jedes besprochenen *C*-Programms war u. a. die Anweisung `#include <stdio.h>`, ein Befehl für das Einbinden eines Standard-Moduls, der uns speziell die Einlese- und Ausgabefunktionen bereitstellen sollte. Diesen Spezialfall der Modularisierung müssen wir jetzt nur noch erweitern, und zwar

- auf die Vielfalt von benutzbaren Standard-Modulen (Standard-Bibliotheksmodule)
- und auf die selbst definierbaren Module (selbstdefinierte Bibliotheksmodule).

In diesem Sinne wollen wir unter einem *Programm* ein Modul mit genau einer Funktion namens `main()` und beliebigen Importen von (für die Abarbeitung notwendigen) Bibliotheksmodulen (also Standard- und selbstdefinierte Module) verstehen.

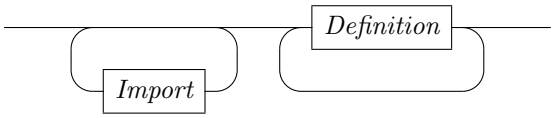
Vom Aufbau her sind beide Modulkategorien gleich: Sie bestehen jeweils aus einem Definitionsmodul (auch Header-File genannt), bezeichnet mit *filename*`.h`, und einem Implementierungsmodul, bezeichnet mit *filename*`.c`. Beide Modulteile müssen denselben Namen tragen und bilden eine logische Einheit.

Standard-Bibliotheksmodule werden durch den Import des entsprechenden Definitionsmoduls im Programm verfügbar gemacht (wie bereits mit `#include <stdio.h>` praktiziert); sowohl Definitionsmodule als auch Implementierungsmodule sind schon programmiert und können, wenn sie hilfreich sind, vom Programmierer genutzt werden. Die Einbindung (Import) selbstdefinierter Module geschieht genauso, nur muss hier der Programmierer zunächst seine Definitionsmodule mit zugehörigen Implementierungsmodulen selbst programmieren, dann übersetzen und schließlich in seiner Nutzerbibliothek ablegen.

7.1 Definitionsmodul

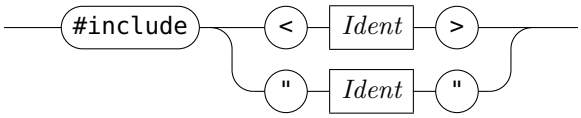
Im Definitionsmodul (oder: Header-File) *filename*`.h` wird die Schnittstelle des Moduls festgelegt. Das geschieht durch die Angabe der Objekte (d. h. Konstanten, Typen, Variablen und Funktionen), die von außen sichtbar und nutzbar sein sollen (siehe Syntaxdiagramm *Definition*). Man sagt, dass diese Objekte *exportiert* werden. Vom Standpunkt eines anderen Moduls können diese (und nur diese) Objekte mit `#include "`*filename*`.h"` importiert und benutzt werden. Die importierten Objekte werden im importierenden Modul nicht einzeln aufgelistet. Natürlich kann das Modul selbst auch Objekte von anderen Modulen importieren (siehe Syntaxdiagramm *Import*).

DefinitionModule



- Der Definitionsmodul besteht aus einer Liste von `#include`-Anweisungen (Importen) und Definitionen.

Import

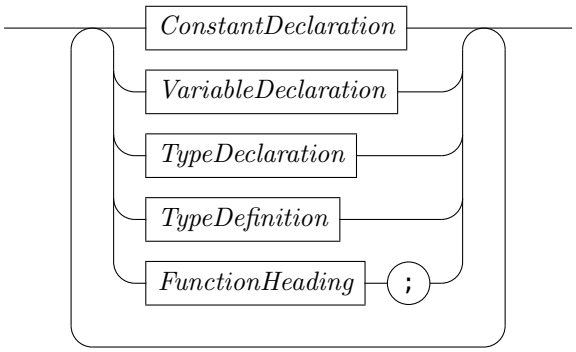


- Das Schlüsselwort `#include` wird gefolgt vom Namen des Header-Files des Moduls, aus dem Objekte importiert werden sollen, eingeschlossen in spitze Klammern (`<>`) oder Anführungszeichen (`""`). Der Name muss dabei immer das Header-File des zu importierenden Moduls sein.
- Die unterschiedliche Syntax der beiden `#include`-Anweisungen hat folgende Auswirkung:

  Das Einschließen des Bibliotheksnamens in `< >` (wie bei `#include <stdlib.h>`) bewirkt, dass die betreffende Bibliothek nur in den voreingestellten include-library-paths gesucht wird (Standardbibliotheken).

  Dagegen werden Bibliotheken, die in `" "` eingeschlossen sind (wie bei `#include "stacks.h"` im später folgenden Beispiel `stacks`), zuerst im Default-Directory gesucht, also in dem Directory, welches das Projekt enthält. Damit ist auch die Umdefinition bzw. Erweiterung von Bibliotheken möglich.

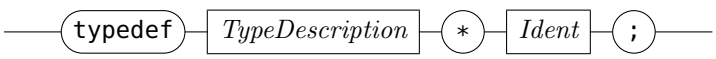
Definition



- Es können Konstanten-, Typ-, Variablen- und Funktionsdeklarationen exportiert werden.
- Die Deklaration von Konstanten, Typen und Variablen erfolgt wie in Kapitel 3 bzw. Kapitel 6 beschrieben. Pointertypen müssen nicht vollständig beschrieben werden (siehe *TypeDefinition*); in diesem Fall muss die vollständige Typdeklaration später im Implementierungsteil erfolgen.
- Bei Funktionen wird nur die *FunctionHeading* (d. h. Ergebnistyp, Name der Funktion, Parameterliste (ggf. leer)) angegeben.

Definition von Pointertypen:

TypeDefinition



- Eine Typdefinition definiert einen Pointertyp. Sie ist ähnlich zu einer Typdeklaration aufgebaut. Allerdings kann die Beschreibung des Typs (*TypeDescription*), auf den ein Pointer vom Typ *Ident* zeigt, erst im Implementierungsmodul erfolgen und so vor dem Nutzer verborgen werden. In diesem Fall spricht man von einem undurchsichtigen oder opaken Typ (Stichwort: abstrakte Datentypen). Die *TypeDescription* ist also ein syntaktischer Teil der entsprechenden Typdeklaration.

Beispiel 7.1. Nehmen wir an, dass einer der Mitarbeiter unseres fiktiven Softwareprojekts die Datenstruktur „Keller“ mit Zugriffsoperationen zu programmieren und den anderen Projektmitarbeitern zur Verfügung zu stellen hätte. Dann könnte das Definitionsmodul, das schon zu Projektbeginn festgelegt wird, wie folgt aussehen:

```
1  /* Header-File pushdown.h */
2
3  typedef struct ele *pushdown;
4
5  void CreatePushdown(pushdown *s);    /* erzeugt einen leeren Keller           */
6  void Push(pushdown *s, int x);       /* legt ein Element x auf dem Keller ab  */
7  void Pop(pushdown *s, int *x);       /* entfernt das oberste Element vom
8                                          Keller und speichert den Wert auf *x  */
9  int Empty (pushdown s);              /* testet, ob pushdown leer ist          */
```

Dieses Definitionsmodul enthält keine Import-Anweisungen. Es bietet nach außen den opaken Datentyp `pushdown` (Keller) und die auf ihm arbeitenden Funktionen

- für das Anlegen eines neuen Kellers (`CreatePushdown`),
- für das Auflegen eines Elements auf einen Keller (`Push`),
- für das Wegnehmen eines Elements von einem Keller (`Pop`) und
- für den Test auf Leerheit (`Empty`) des Kellers

an. Vom Datentyp `pushdown` ist nur bekannt, dass es ein Pointertyp (Pointer auf einen Strukturtyp) ist.

Das Modul `pushdown` könnte dann von einem anderen Modul beispielsweise folgendermaßen importiert und benutzt werden.

```
1  . . .
2  #include "pushdown.h"
3
4  . . .
5  pushdown t, u;
6  int x;
7  . . .
8  { . . .
9    CreatePushdown(&u);
10   CreatePushdown(&t);
11   Push(&u, 25);
12   Push(&t, 7);
13   . . .
14   if (!Empty(u))
15   { Pop(&u, &x);
16     Push(&t, x);
17   }
18   . . .
19 }
```

Das Zusammenwirken des Moduls `pushdown` und des anderen Moduls wird in Abbildung 7.1 veranschaulicht. □

7.2 Implementierungsmodul

Im Implementierungsmodul werden die für den Export im Definitionsmodul angekündigten Objekte programmiert. Diese Programmierung bleibt für andere Module unsichtbar und ist nicht zugreifbar oder veränderbar (Stichwort: Datenkapselung, information hiding).

- Das Implementierungsmodul (*filename*`.c`) muss das Definitionsmodul (*filename*`.h`) ebenfalls mit `#include "`*filename*`.h"` importieren.

Modul 1

Definitionsmodul

```
1  /* Header-File pushdown.h */
2
3  typedef struct ele *pushdown
4
5  void CreatePushdown(pushdown *s); /* erzeugt einen leeren Keller */
6  void Push(pushdown *s, int x);    /* legt ein Element x auf dem Keller ab*/
7  void Pop(pushdown *s, int *x);    /* entfernt das oberste Element vom Keller
8                                       und speichert den Wert auf *x */
9  int Empty(pushdown *s);           /* testet, ob pushdown leer ist */
```

Implementierungsmodul

```
1  /* Implementierungsmodul pushdown.c*/
2
3  #include <stdlib.h>
4  #include "pushdown.h"
5
6  typdef struct ele {int key; pushdown next;} element;
7
8  void CreatePushdown(pushdown *s)
9  { *s=NULL;
10 }
11
12 void Push(pushdown *s, int x)
13 { pushdown top;
14   top= (pushdown) malloc (sizeof(element)) /* erzeuge neues oberes Kellerelement*/
15   top->key= x;
16   top->next= *s;
17   *s=top;
18 }
19
20 void Pop(pushdown *s, int *x)
21 { pushdown top;
22   top= *s;
23   *x= (*s)->key;
24   *s= (*s)->next;
25   free(top);                          /* entferne oberes Kellerelement*/
26 }
27
28 int Empty(pushdown s)
29 { return s==NULL;
30 }
```

Modul 2

Definitionsmodul

```
1  ...
2
3  ...
```

Import

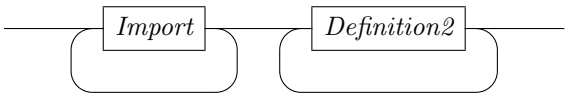
Implementierungsmodul

```
1  ...
2  #include "pushdown.h"
3
4  ...
5  pushdown t,u;
6  int x;
7  ...
8  {
9   CreatePushdown(&u);
10  CreatePushdown(&t);
11  Push(&u, 25);
12  Push(&t, 7);
13  ...
14  if (!Empty(u))
15  { Pop(&u, &x);
16    Push(&u, x);
17  }
18  ...
19 }
```

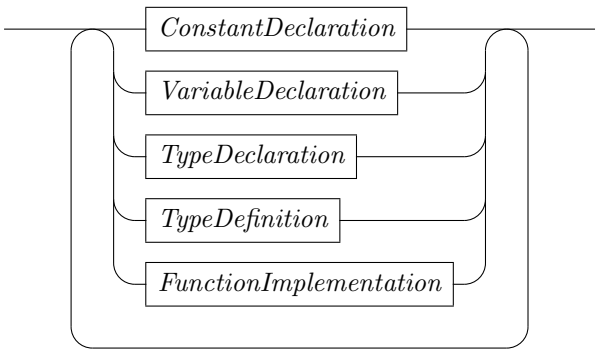
Abbildung 7.1: Zusammenwirken im Modul `pushdown`

- Im Implementierungsmodul müssen alle Beschreibungen opaker Datentypen und alle Funktionsdeklarationen zu den im Definitionsmodul enthaltenen Funktionsköpfen enthalten sein.
- Alle anderen im Definitionsmodul enthaltenen Namen (von Konstanten, Variablen, sichtbaren (d. h. nicht opaken) Datentypen) dürfen *nicht* mehr erscheinen.

ImplementationModule



Definition2



Beispiel 7.1 (Fortsetzung)**.** Im Implementierungsmodul `pushdown.c` werden der opake Datentyp `pushdown` sowie die Funktionen `CreatePushdown`, `Push`, `Pop` und `Empty` programmiert.

```
1  /* Implementierungsmodul pushdown.c */
2
3  #include <stdlib.h>
4  #include "pushdown.h"
5
6  typedef struct ele { int key;
7                       pushdown next;} element;
8
9  void CreatePushdown(pushdown *s)
10 { *s = NULL;
11 }
12
13 void Push(pushdown *s, int x)
14 { pushdown top;
15
16   top = (pushdown) malloc(sizeof(element));
17   top->key = x;
18   top->next = *s;
19   *s = top;
20 }
21
22 /* 'Pop' nimmt an, dass 's' kein leerer 'pushdown' ist. */
23 void Pop(pushdown *s, int *x)
24 { pushdown top;
25
26   top = *s;
27   *x = (*s)->key;
28   *s = (*s)->next;
29   free(top);
30 }
31
32 int Empty(pushdown s)
33 { return s==NULL;
34 }
```

Dieses Implementierungsmodul importiert das Bibliotheksmodul `stdlib`, um die Funktionen `malloc` und `free` zum Anlegen bzw. Löschen von Speicherplätzen eines dynamischen Datentyps sowie die Pointerkonstante `NULL` verfügbar zu machen. □

Teil II

Algorithmische Problemstellungen

In diesem Teil der Vorlesung wollen wir für verschiedene Problemstellungen Algorithmen entwerfen und analysieren.

In Kapitel 1 haben wir grob definiert, was ein Algorithmus ist (siehe Abschnitt 1.1.3). Jetzt wollen wir Maße angeben, mit deren Hilfe man die Komplexität von Algorithmen messen kann (Kapitel 8). In den Kapiteln 9 bis 13 formulieren wir dann in Pseudo-*C* einzelne Algorithmen der Problem- und Themenfelder Sortieren, Suchen, Bäume, Graphalgorithmen bzw. einen Algorithmus zur Approximation von Wahrscheinlichkeitsverteilungen. Im Kapitel 14 werden wir Konstruktionsmethoden von Algorithmen vorstellen und ihre Wirksamkeit an Beispielen belegen.

8 Komplexität von Algorithmen

Da ein Algorithmus ein Problem lösen soll, muss der Algorithmus auf jeden Fall die Eigenschaft haben, dass er korrekt ist. Das heißt, wenn etwa das Problem darin besteht, den Funktionswert einer Funktion f zu einem beliebig vorgegebenen Argument x zu berechnen, dann muss der Algorithmus auch tatsächlich $f(x)$ berechnen.

Im allgemeinen wird es zu einem Problem verschiedene korrekte Algorithmen geben.

Beispiel 8.1. Ein aus unserem Alltag bekanntes Problem ist das Auffinden eines Namens im Telefonbuch. Zwei intuitiv bekannte Suchalgorithmen sollen den Stellenwert einer guten Lösungsidee verdeutlichen.

Algorithmus 3 Lineares Suchen

```
Beginnend mit der ersten Seite wird in der Reihenfolge der Seiten das
Telefonbuch durchsucht.
```

Beurteilung: korrekt, aber sehr ineffizient; „linearer Aufwand (in der Anzahl der Einträge)“.

Algorithmus 4 Binäres Suchen

```
Setze l (erste Seite eines Seitenbereichs)  = Anfangsseite des Telefonbuches
Setze r (letzte Seite eines Seitenbereichs) = Schlussseite des Telefonbuches

Wiederhole folgende Schritte bis Name gefunden bzw. nicht gefunden
   * Besteht der Seitenbereich aus nur einer Seite, dann durchsuche diese.
       Wird Name gefunden, dann Telefonnummer merken und Ende des Suchens.
       Wird Name nicht gefunden, dann Feststellung, dass Name nicht im
        Telefonbuch steht und Ende des Suchens.
   * Schlage Telefonbuch etwa in der Mitte des Seitenbereichs l...r auf; die
     aufgeschlagene Seite habe die Nummer m.
   * Wenn gesuchter Name im Bereich l...m liegen müsste, dann setze r=m,
     andernfalls setze l=m und arbeite mit diesem neuen Bereich weiter.
```

Beurteilung: korrekt und effizient; „logarithmischer Aufwand“. □

In der Regel interessieren wir uns in der Menge der Lösungsalgorithmen für den effizientesten, d. h. den Algorithmus, der am schnellsten die Lösung berechnet und dabei nach Möglichkeit auch noch am wenigsten Speicherplatz verwendet. Den Zeit- und Speicherplatzbedarf eines Algorithmus nennt man auch seine Komplexität (Zeitkomplexität, Platzkomplexität).

Intuitiv ist klar, dass das lineare Suchen bezüglich der Zeitkomplexität schlechter (d. h. ineffizienter) als das binäre Suchen ist. Das ist uns klar, obwohl wir weder eine konkrete Programmiersprache noch eine konkrete Rechnertechnologie (heimischer PC versus Hochleistungsrechner) angegeben haben. In der Tat abstrahiert man bei der Komplexitätsanalyse eines Algorithmus von diesen Randbedingungen und geht dazu über, die *Anzahl von bestimmten Operationen*, die der Algorithmus ausführt, als seine Laufzeit anzusehen und *nicht* die konkrete Anzahl von Millisekunden, die der Rechner zur Ausführung benötigt. Wenn man beispielsweise einen Such- oder Sortieralgorithmus auf seine Zeitkomplexität untersuchen will, so kann man die Vergleiche von Schlüsseln zählen, oder wenn ein arithmetischer Algorithmus vorliegt, so kann man die ausgeführten arithmetischen Operationen zählen.

Zur Abstraktion von der Rechnertechnologie kommt noch eine *zweite Abstraktion*: Für einen gegebenen Algorithmus und eine Eingabe w ist man nicht an der genauen Zahl der Vergleichsoperationen oder

arithmetischen Operationen *für diese konkrete Eingabe w* interessiert, vielmehr interessiert man sich nur für das *Wachstum* dieser Anzahl bei Vergrößerung des Problems (d. h. hier: Verlängerung des Wortes w).

Bei der Komplexität eines Algorithmus unterscheidet man (nach der 2. Abstraktion) zwischen seinem Verhalten im besten Fall (best-case), durchschnittlichen Fall (average-case) und im schlechtesten Fall (worst-case). Zur Ermittlung dieser Komplexitäten betrachtet man für eine beliebige aber feste Problemgröße n alle Probleme der Größe n und berechnet das Minimum, den Durchschnitt bzw. das Maximum der entsprechenden Laufzeiten. Im Rahmen dieser Vorlesung werden wir nicht die average-case, häufig auch nur die worst-case Komplexität angeben; auch werden wir uns auf die Zeitkomplexität konzentrieren.

Beispiel 8.2. Betrachten wir noch einmal unseren Algorithmus MinAlter von Seite 11:

Algorithmus MinAlter (Wiederholung von Algorithmus 1)

Eingabe: Eine Folge $a_1, \ldots, a_n$ von positiven, ganzen Zahlen.

Ausgabe: der kleinste Positionsindex j mit $a_j = \min\{a_1, \ldots, a_n\}$.

Verfahren: Zusätzliche Variablen: x (für das Alter), i (als Zählvariable);

1. (*Initialisierung*) Setze $j = 1, x = a_j$ und $i = 2$.
2. (*Suchlauf*)
   Solange $i \leq n$ gilt, wiederhole:
   falls $a_i < x$, setze $j = i$ und $x = a_j$
   erhöhe i um 1
3. Ausgabe von j als Ergebnis

Die beiden Abstraktionsschritte sind in Abbildung 8.1 veranschaulicht, wobei $\mathbb{N}^{\{0,\ldots,400\}}$ die Menge aller Hörsaalbelegungen mit 401 Plätzen bezeichnet (pro Platz: eine Altersangabe). Wir können uns $\mathbb{N}^{\{0,\ldots,400\}}$ als die Menge aller Wörter $w = (a_0, \ldots, a_{400})$ mit $a_i \in \mathbb{N}$ (für $0 \leq i \leq 400$) vorstellen.

Nun lassen wir den Algorithmus gedanklich auf einer beliebigen, aber festen Folge $w = a_1, ..., a_n$ von positiven, ganzen Zahlen ablaufen. Für die Analyse der Zeitkomplexität von MinAlter interessiert uns die Anzahl $T(w)$ aller *nach* Ablauf ausgeführten Zuweisungen und Vergleiche. Entsprechend der zweiten Abstraktion betrachtet man nun eine Eingabelänge n und dann *alle* Folgen der Länge n und definiert

- den besten Fall (best-case):

$$T_{\text{best-case}}(n) = \min\{T(w) \mid w \in \mathbb{N}^n, \text{ mit verschiedenen Zahlen}\}$$

- den durchschnittlichen Fall (average-case):

$$T_{\text{average-case}}(n) = \text{erwartete Anzahl bei Gleichverteilung aller Eingaben der Länge } n$$

- den schlechtesten Fall (worst-case):

$$T_{\text{worst-case}}(n) = \max\{T(w) \mid w \in \mathbb{N}^n, \text{ mit verschiedenen Zahlen}\}$$

Zur genauen Bestimmung dieser Werte füllen wir die folgende Tabelle aus wobei eine Zuweisung und ein Vergleich jeweils den Zeitbedarf E hat.

Operation	Aufwand	Wie oft wird Operation ausgeführt?
Setze $j = 1$	$1\,E$	1
Setze $x = a_j$	$1\,E$	1
Setze $i = 2$	$1\,E$	1
Teste $i \leq n$	$1\,E$	n
Teste $a_i < x$	$1\,E$	$n-1$
Setze $j = i, x = a_j$	$2\,E$	? ($n-1$ bis 0)
Setze $i = i + 1$	$1\,E$	$n-1$
Ausgabe	$1\,E$	1

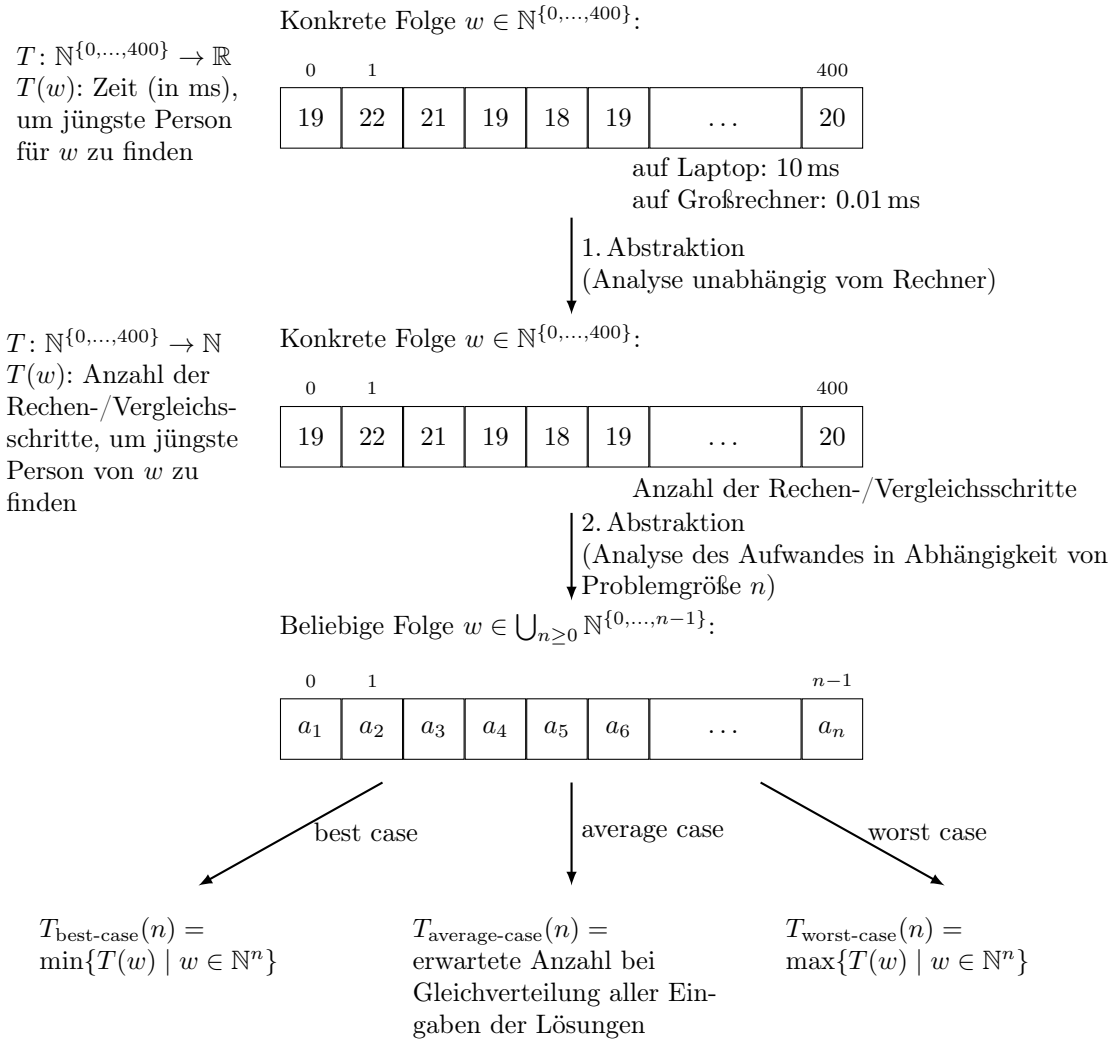


Abbildung 8.1: Analyse des Algorithmus MinAlter

Während die Ermittlung von $T_{\text{best-case}}(n)$ und $T_{\text{worst-case}}(n)$ ohne Zusatzkenntnisse möglich ist, werden für $T_{\text{average-case}}(n)$ Kenntnisse aus der Wahrscheinlichkeitstheorie benötigt.

$$\begin{aligned} T_{\text{best-case}}(n) &= (3n+2) \\ T_{\text{worst-case}}(n) &= 5n \\ T_{\text{average-case}}(n) &= (3n+2) + 2\textstyle\sum_{i=2}^{n} \frac{1}{i} \end{aligned}$$

$T_{\text{average-case}}(n)$ gilt unter der Annahme, dass alle Anordnungen (Permutationen) von $a_1, \ldots, a_n$ gleich wahrscheinlich und auch alle a_i mit $(1 \leq i \leq n)$ voneinander verschieden sind. Unter den Zahlen $a_1, ..., a_i$ ist dann nämlich a_i mit der Wahrscheinlichkeit $\frac{1}{i}$ das kleinste Element. □

Die Komplexitätstheorie wird in der Vorlesung Theoretische Informatik und Logik vertieft.

Bei der Ermittlung der Komplexitäten geht man noch einen Schritt weiter; hier kommt es nicht auf konstante Faktoren und additive Konstanten an. Man faßt also Komplexitätsfunktionen zusammen, die sich nur durch einen konstanten Faktor und eine additive Konstante unterscheiden: Sei $f\colon \mathbb{N} \to \mathbb{N}$, dann definiere

$$O(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \text{es gibt } c_1 > 0, c_2 > 0 \text{ und } n_0\colon \text{für jedes } n \geq n_0\colon g(n) \leq c_1 \cdot f(n) + c_2\}$$

für die Abschätzung der von Funktionen nach oben (*Groß-O-Notation*; f „ist obere Schranke“ für alle

$g \in O(f)$) und

$$\Omega(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \text{es gibt } c > 0 \text{ und } n_0 > 0\colon \text{für jedes } n > n_0\colon g(n) \geq c \cdot f(n)\}$$

für die Abschätzung der der Funktion nach unten (*Groß-Omega-Notation*; f ist „untere Schranke“ für alle $g \in \Omega(f)$). Beispielsweise gilt für die Funktion n^4 folgendes:

$$n^4 \in (O(n^6) \cap \Omega(n^2))$$
$$n^4 \in (O(n^5) \cap \Omega(n^3))$$

Lassen sich untere und obere Aufwandsschranke, d. h. $\Omega(f_1)$ und $O(f_2)$, für einen Algorithmus angeben, und es gilt $f_1 = f_2 = f$, dann sagt man auch, der Algorithmus hat die *genaue Komplexität* $\Theta(f)$ (*Groß-Theta*).

Noch einige abschließende Bemerkungen zu den Abschätzungen $O(f)$ und $\Omega(f)$. Soll für einen Algorithmus der Aufwand nach oben abgeschätzt werden, also der worst-case betrachtet werden, dann sucht man eine obere Schrankenfunktion $f\colon \mathbb{N} \to \mathbb{N}$, wobei man bemüht ist, eine möglichst kleine obere Schranke zu finden; optimal wäre die *kleinste* obere Schranke. Die Bestimmung der worst-case-Komplexität wird also im Regelfall auf die Berechnung einer möglichst kleinen oberen Schrankenfunktion f hinaus laufen; daraus ergibt sich $O(f)$ nach obiger Definition. Soll der Aufwand nach unten abgeschätzt werden, so sucht man eine untere Schrankenfunktion $f\colon \mathbb{N} \to \mathbb{N}$, wobei man hier bemüht ist, die *größte* untere Schranke f zu finden. Bei der Bestimmung der best-case-Komplexität ist also im Regelfall die größte untere Schranke f interessant, also ein Mindestaufwand, der niemals unterschritten werden kann. Daraus ergibt sich dann $\Omega(f)$ nach obiger Definition. Beide Schranken sind asymptotische Abschätzungen, das heißt: Die relativen Abweichungen vom realen Verhalten des Algorithmus werden um so kleiner, je größer das Problem wird. Haben wir nun einen gegebenen Algorithmus, so wird der worst case bei extrem ungünstiger Eingabeinformation auftreten, der best case bei der Günstigsten.

Nehmen wir zum Beispiel unseren Algorithmus „Binäres Suchen“ von Seite 75, so würde der worst case eintreten, wenn der gesuchte Name auf der ersten Seite im Telefonbuch wäre. Wir müssten etwa $\log_2 n$ (n ist die Anzahl der Seiten des Telefonbuches) mittlere Seiten aufschlagen und die jeweils aufgeschlagene Seite auswerten. Für den Suchaufwand erhalten wir somit als worst-case-Komplexität $O(\log_2 n) \cdot O(c) = O(c \cdot \log_2 n) = O(\log_2 n)$, wobei $O(c)$ der maximale Aufwand ist, um eine Telefonbuchseite zu durchsuchen bzw. auszuwerten. Dieser ist, wie wir wissen, nicht vom Umfang n des Telefonbuchs abhängig, sondern nur vom Inhalt einer Seite und natürlich dem (hier nicht spezifizierten) Suchalgorithmus. In jedem Fall kann der Aufwand durch eine Konstante c abgeschätzt werden.

Der best-case würde eintreten, wenn der gesuchte Name auf der ersten aufgeschlagenen Seite stünde, hier vielleicht sogar in der 1. Zeile. Ein nicht zu unterschreitender Mindestaufwand wäre somit $\Omega(c)$ mit $c \in \mathbb{N}$.

Beispiel 8.3. Die Funktion $f(n) = 4n^2 - 34n + 1024$ ist in $O(g)$ und auch in $\Omega(g)$, wobei $g(n) = n^2$ ist.
Denn: $4n^2 - 34n + 1024 \leq 4n^2 + 1024 = 4 * g(n) + 1024$.
Wählt man also $c_1 = 4$ und $c_2 = 1024$, so lässt sich mit g eine Majorante für f angeben, somit $f = O(g)$.
Weiterhin gilt: $4n^2 - 34n + 1024 > n^2$ für $n \in \mathbb{N}$.
Wählen wir nun $c = 1$ und $n_0 = 1$, so haben wir mit g bzw. n^2 eine Minorante gefunden, also $f = \Omega(g)$.
(Beachte: $f(n) - n^2$ besitzt *keine* reellen Nullstellen!) □

Somit lässt sich in diesem Fall auch leicht die *genaue* Komplexität angeben, nämlich $f = \Theta(g)$. Oft schreibt man nur $O(n^2)$, $\Omega(n^2)$ oder $\Theta(n^2)$ und sagt entsprechend:

- Der Algorithmus hat *höchstens* quadratisches Wachstum,
- der Algorithmus hat *mindestens* quadratisches Wachstum bzw.
- der Algorithmus hat *genau* quadratisches Wachstum.

Wichtige und häufig auftretende Wachstumsklassen sind:

- logarithmisches Wachstum: $O(\log n)$,
- lineares Wachstum: $O(n)$,
- $n \cdot \log n$-Wachstum: $O(n \cdot \log n)$,
- polynomielles Wachstum: $O(n^k)$ für ein $k \in \mathbb{N}$,

- exponentielles Wachstum: $O(2^n)$

Die Abbildung 8.2 soll das Wachstumsverhalten der o. g. Funktionsklassen deutlich machen. Da Komplexitätsbetrachtungen insbesondere das Lösungsverhalten für große n (Problemgrößenparameter) untersuchen sollen, wird vordergründig das asymptotische Verhalten interessieren.

Algorithmen mit exponentiellem Wachstum gelten (i. Allg.) als für die Praxis wertlos. Das ergibt sich daraus, dass für große Probleme – und gerade für die Lösung *großer* Probleme benutzen wir Rechner – der Zeitaufwand gigantisch ist. Zweitens haben diese Algorithmen die Eigenschaft, dass selbst bei verbesserter Rechnertechnologie (wenn etwa die Rechengeschwindigkeit um den Faktor 100 steigt) der Zeitaufwand nicht linear sinkt.

Als praxistauglich werden die Algorithmen eingestuft, die ein niedrigeres Wachstum als exponentiell haben. Aber Vorsicht: Die multiplikativen Konstanten können so groß sein, dass der Algorithmus letztlich doch nur akademischen Wert hat.

Wie auch immer: Der Informatiker sollte zur Lösung eines Problems einen Algorithmus schaffen, der erstens korrekt und zweitens effizient ist. Das sind die Mindestanforderungen. In der Vorlesung Softwaretechnologie werden noch andere Anforderungen diskutiert, die sich hauptsächlich auf das „Programmieren im Großen“ beziehen.

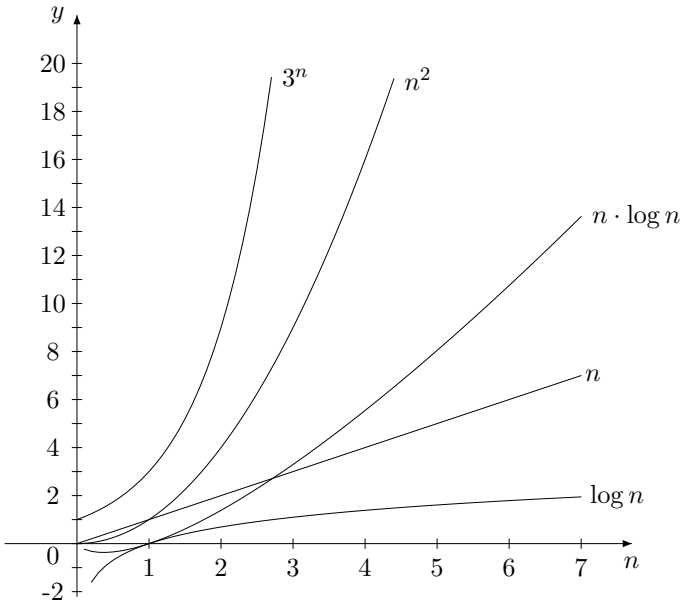


Abbildung 8.2: Wachstumsverhalten von Funktionsklassen

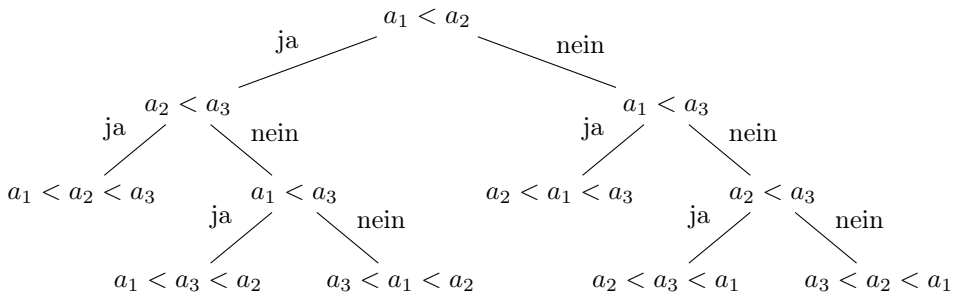
9 Sortieren

In diesem Kapitel werden wir verschiedene Algorithmen vorstellen, die zum Sortieren einer Sequenz von Zahlen eingesetzt werden können. In der Praxis wird man üblicherweise nicht nur Zahlen, sondern auch Datensätze sortieren wollen. Dabei erfolgt die Sortierung nach einem Schlüsselwert (das ist z. B. eine natürliche Zahl, die ein Objekt eindeutig bestimmt), der in jedem Datensatz enthalten ist. Die restlichen Daten eines Datensatzes werden während des Sortiervorgangs zusammen mit dem Schlüssel umgeschichtet, bei großen Datensätzen wird häufig nur ein Feld von Zeigern sortiert. Da die Schlüssel im allgemeinen ganze Zahlen sind, werden wir uns hier auf das Sortieren von Zahlen beschränken und abstrahieren von möglichen Satellitendaten.

Häufig werden sogenannte *vergleichende Sortieralgorithmen* benutzt, welche die Reihenfolge der Elemente der Eingabe nur auf Grund von Vergleichen zwischen den Eingabeelementen bestimmen. Diese Algorithmen können nicht auf andere Weise Information über ihre Eingabe erlangen. Zunächst stellen wir uns die Frage, wie schnell überhaupt mit vergleichenden Sortieralgorithmen sortiert werden kann. Wir suchen also eine Funktion g, so dass für jeden Algorithmus H, der eine Folge von Zahlen vergleichsbasiert sortiert, gilt, dass die Zeitkomplexität von H von der Art $\Omega(g)$ ist, d. h. mindestens so schnell wie g wächst. Dabei betrachten wir als Eingabe n verschiedene, positive, ganze Zahlen $a_1, \ldots, a_n$ und die natürliche $<$-Ordnung. Gesucht ist dann eine Permutation $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ mit $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$.

Da wir vergleichende Sortieralgorithmen betrachten, zählen wir den Vergleich und den Austausch von zwei Zahlen als Basisoperation. Nach einer Sequenz von Vergleichen hat man soviel Information über die eingegebenen Zahlen gewonnen, dass man die entsprechende Permutation berechnen kann.

Alle diese Sequenzen lassen sich in einem sogenannten *Entscheidungsbaum* zusammenfassen. Für $n = 3$ und a_1, a_2, a_3 könnte dieser Baum folgendermaßen aussehen:



Aus den Blattbeschriftungen lässt sich dann leicht die zugehörige Permutation ableiten. Beispielsweise entspricht das Blatt des Pfades (ja, nein, ja) der Permutation $\pi(1) = 1$, $\pi(2) = 3$, $\pi(3) = 2$.

Da die eingegebene Zahlenfolge beliebig sein kann, muss jeder Sortieralgorithmus einen solchen Entscheidungsbaum konzipieren. Für jede Eingabe besteht dann der Ablauf des Sortieralgorithmus im Auffinden und Durchlaufen eines Pfades durch den Entscheidungsbaum von der Wurzel zu einem Blatt. Die Länge der Pfade bestimmt also auch die Mindestlaufzeit eines Sortieralgorithmus. Hier, für $n = 3$, sind dies drei bzw. vier Vergleiche, und im Allgemeinen entspricht die Höhe des Baumes, also die Länge seines längsten Pfades von der Wurzel zu einem seiner Blätter, der Anzahl an Vergleichsoperationen, die der Algorithmus im schlechtesten Fall auszuführen hat.

Betrachten wir nun die Anzahl an Blättern in dem Entscheidungsbaum für die Sortierung von n vorgegebenen Zahlen. Für die Anordnung dieser Zahlen gibt es $n!$ Permutationen. Da zu jeder möglichen Permutation genau ein Blatt in dem Entscheidungsbaum gehören muss (denn der Algorithmus soll jede mögliche Reihenfolge der Eingabe berücksichtigen können), gibt es also genau $n!$ Blätter (für $n = 3$ sind es also mindestens $3! = 6$ Blätter).

Wie hoch ist ein binärer Baum mit $n!$ oder mehr Blättern? Er hat mindestens die Höhe $\log_2(n!)$. Diese Zahl lässt sich nach der Stirlingschen Formel abschätzen und ist von der Art $\Omega(n \cdot \log_2 n)$. Also: Jeder vergleichende Sortieralgorithmus für n Zahlen benötigt mindestens $n \cdot \log_2 n$ Rechenschritte.

Es gibt jedoch auch Sortieralgorithmen, die nicht vergleichsbasiert sind (zum Beispiel *Bucketsort* oder *Radixsort*) und die unter bestimmten Voraussetungen an die Eingabe in linearer Zeit laufen. Wir wollen uns in diesem Kapitel jedoch nur auf vergleichende Sortieralgorithmen beschränken.

Betrachten wir nun konkrete Sortierverfahren. Man unterscheidet zwischen:

- externem Sortieren (Daten befinden sich auf externem Speicher mit sequentiellem Zugriff) und
- internem Sortieren (Daten befinden sich im Hauptspeicher mit beliebigem Zugriff).

sowie

- in-place Sortieralgorithmen (benötigen eine konstante Anzahl an zusätzlichen Speicherplätzen) und
- out-of-place Sortieralgorithmen (die Größe des zusätzlich benötigten Speicherplatzes hängt von der Größe der Eingabe ab)

Hier werden wir nur interne in-place Sortierverfahren besprechen, und zwar: *Quicksort* und *Heapsort*.

In jedem Fall sind die zu sortierenden Zahlen in einem Feld `a` abgelegt mit der Deklaration:

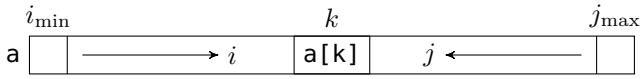
```
int a[n];  /* a[0] ... a[n-1] */
```

9.1 Quicksort

Der Sortieralgorithmus *Quicksort* wurde von C.A.R. Hoare im Jahre 1962 entwickelt. Die Algorithmenidee folgt der Lösungsmethode *Divide-and-Conquer* (siehe auch Kapitel 14) und vollzieht die (notwendige) Zerlegung des jeweils (noch) zu sortierenden Feldes in zwei Teilfelder mit Hilfe einer Mengenpartitionierung. Ein beliebig gewähltes Element dieses noch zu sortierenden (Teil-)Feldes dient hierbei als Teilungselement, und *eine* Felddurchmusterung genügt nun, um mit Hilfe von Vertauschungen der Feldelemente sicherzustellen, dass alle Elemente links vom gewählten Teilungselement kleiner und rechts davon größer sind. Dabei erfolgt die Felddurchmusterung so, dass möglichst zwei weit entfernte Elemente ausgetauscht werden; dadurch wird ein ineffizientes lokales Verschieben vermieden.

Der Prozess des Teilens und Partitionierens wird solange ausgeführt, bis alle Teilfelder sortiert sind (ein- oder nullelementig). Obwohl man bei einem beliebig gewählten Teilungselement keineswegs eine (optimale) Halbierung des Feldumfanges erreicht, zeigt sich, dass auch diese Zerlegung im statistischen Mittel nach $O(\log n)$ Schritten beendet ist. Eine Ausprägung dieses Prozesses ist folgender Algorithmus:

```
 1  void quicksort(int a[], int L, int R)  // L und R bezeichnen die linke bzw. rechte
 2                                         // Grenze des zu sortierenden Teils von a
 3  { int i, j, w, x, k;
 4
 5    i = L;  j = R;                       // i und j durchlaufen a von links bzw. rechts
 6    k = (L+R) / 2;  x = a[k];            // x wird Pivotelement genannt
 7
 8    do
 9    { while (a[i] < x) i = i + 1;
10      while (a[j] > x) j = j - 1;
11      if (i <= j)
12      { w = a[i];                        //
13        a[i] = a[j];                     // hier werden a[i] und a[j] getauscht
14        a[j] = w;                        //
15        i = i + 1;  j = j - 1;
16      }
17    }
18    while (i <= j);
19
20    if (L < j) quicksort(a, L, j);
21    if (R > i) quicksort(a, i, R);
22  }
```



Übrigens ist der Test `(i <= j)` auf Zeile 11 *nicht* überflüssig: Bei Eingabe von `4 5 6 3 7 8 9` liefert der Test (bei der zweiten Ausführung) den Wahrheitswert *false*.

Beispiel 9.1. Aufruf: `quicksort(a, 0, 6);`
$i = 0,\ j = 6,\ k = 3,\ x = 5$

nach Zeile 9:

a:	7	21	9	5	2	3	14
	↑					↑	
	i					j	

$a[0] = 7 \not< 5 = x$
$a[5] = 3 \not> 5 = x$

nach Zeile 9:

a:	3	21	9	5	2	7	14
		↑			↑		
		i			j		

$a[1] = 21 \not< 5 = x$
$a[4] = 2 \not> 5 = x$

nach Zeile 9:

a:	3	2	9	5	21	7	14
			↑	↑			
			i	j			

$a[2] = 9 \not< 5 = x$
$a[3] = 5 \not> 5 = x$

nach Zeile 9:

a:	3	2	5	9	21	7	14
			↑	↑			
			j	i			

Aufrufe: `quicksort(a, 0, 2)`, `quicksort(a, 3, 6)` usw. □

Die Zeitkomplexität von Quicksort ist

- im günstigsten Fall $\Omega(n \cdot \log n)$,
- im Mittel $O(n \cdot \log n)$,
- im schlechtesten Fall $O(n^2)$. Dieser Fall tritt ein, wenn das gewählte Pivotelement in jedem Schritt das kleinste bzw. größte Element der zu sortierenden Teilfolge ist. Für die Wahl des Pivotselements, wie sie in `quicksort` definiert ist, gilt das z.B. für die Folge $2, 6, 4, 1, 3, 5, 7$.

Der Quicksort-Algorithmus funktioniert auch, wenn eine Zahl mehrfach auftritt.

9.2 Heapsort

Wir gehen wiederum von n verschiedenen, positiven, ganzen Zahlen aus, die in einem Feld a gespeichert sind. Für die Beschreibung von *Heapsort* ist es vorteilhaft, sich das Feld a als Binärbaum vorzustellen.

Beispiel 9.2. Wenn z. B. a die Folge

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$	$a[10]$
7	15	14	8	13	18	24	9	5	16	21

ist, dann wird a durch den Binärbaum in Abbildung 9.1 dargestellt. □

Beginnend bei der Wurzel, füllt man also jede Ebene des Baumes von links nach rechts auf; wenn eine Ebene voll ist, dann füllt man die nächste auf. Es gilt dann, dass ein Knoten, der den Eintrag $a[i]$ mit

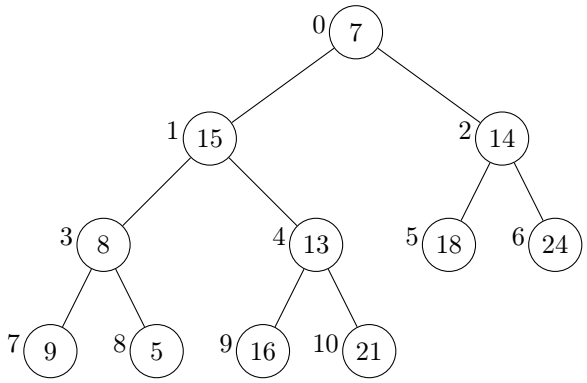


Abbildung 9.1: Binärbaum (kein heap, da z. B. $7 \not> 15$).

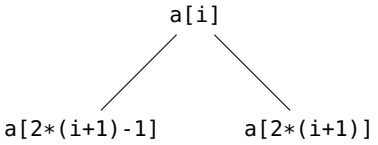


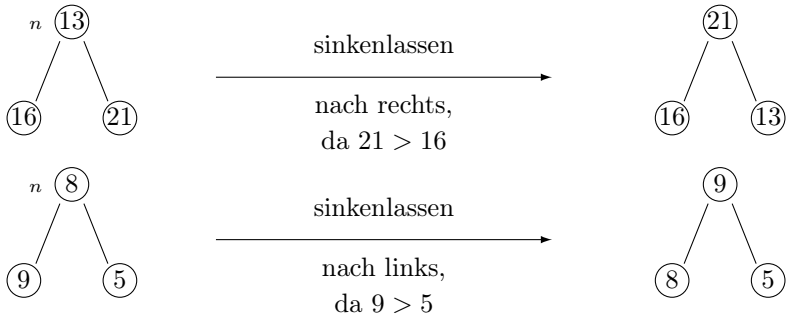
Abbildung 9.2: Veranschaulichung der Nachfolger des Knotens `a[i]` im Binärbaum.

$0 \leq i \leq (n \,\mathrm{DIV}\, 2) - 1$ enthält[1], einen oder zwei Nachfolgerknoten mit den Beschriftungen $a[2 * (i+1) - 1]$ bzw. $a[2 * (i+1)]$ hat (siehe Abbildung 9.2).

Das Ziel von *Heapsort* ist zunächst, aus diesem Binärbaum einen sogenannten „heap“ (Halde) zu konstruieren. Ein *heap* ist ein Binärbaum mit folgenden Eigenschaften:

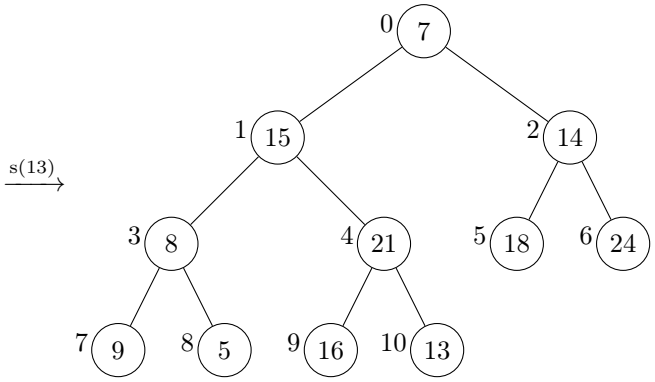
1. Jeder Knoten ist mit einer positiven, ganzen Zahl beschriftet; zwei verschiedene Knoten tragen verschiedene Zahlen.
2. Es gibt eine Ebene t des Baumes, so dass (i) alle auf der Ebene t besetzten Positionen linksbündig angeordnet sind, (ii) alle Positionen auf Ebene $t-1$ besetzt sind und (iii) keine Position der Ebene $t+1$ besetzt ist.
3. Für jeden Knoten n gilt: Wenn n mit h beschriftet ist, dann müssen die Beschriftungen der Nachfolger von n kleiner als h sein (*heap*-Eigenschaft).

Der Binärbaum unseres Beispiels ist also kein *heap*, weil die *heap-Eigenschaft* verletzt ist; die anderen Bedingungen sind erfüllt. Damit der Binärbaum auch die 3. Eigenschaft erfüllt, müssen Knotenbeschriftungen miteinander vertauscht werden. Man beginnt diesen Prozess an der größten Position *li*, die mindestens einen Nachfolger besitzt (d. h. $li = (n \,\mathrm{DIV}\, 2) - 1$, -1 bedingt durch Indexbeginn bei 0 in *C*!), und schreitet zur Position 0 fort. Wenn sich der Prozess nun an einem Knoten n mit Beschriftung h befindet, dann lässt er die Zahl h soweit nach unten (d. h. in Richtung Blätter) sinken, bis beide Nachfolger (wenn vorhanden) mit einer kleineren Zahl beschriftet sind. Das Sinkenlassen erfolgt durch Austausch von h mit der größeren der beiden Beschriftungen der Nachfolger (evtl. auch nur ein Nachfolger) von n.

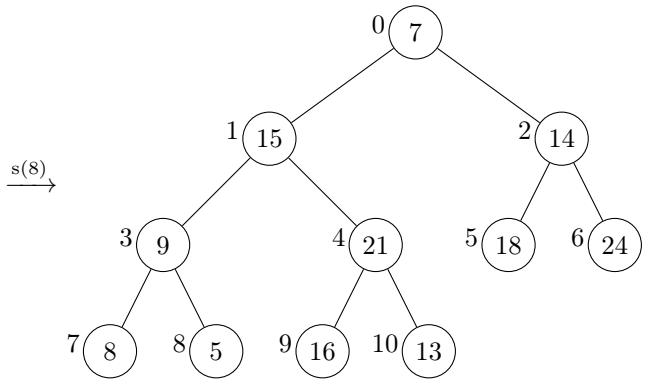


[1] Hier bezeichnet $n \,\mathrm{DIV}\, k$ die ganzzahlige Division von n durch k, also $\lfloor n/k \rfloor$.

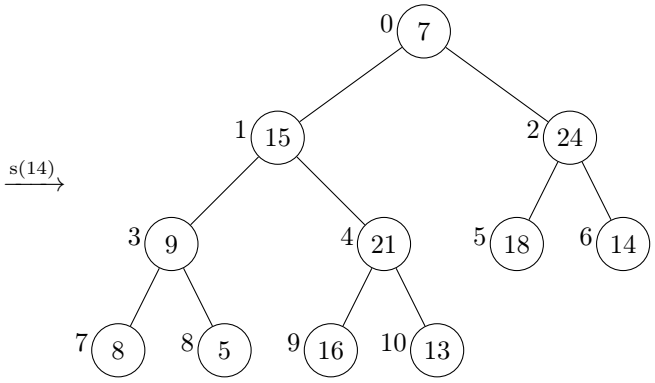
Beispiel 9.2 (Fortsetzung)**.** Führen wir diesen Austauschprozess nun an unserem Beispiel durch. Er beginnt an Position 4 mit Beschriftung 13. Nach Sinkenlassen der 13 entsteht folgender Binärbaum:



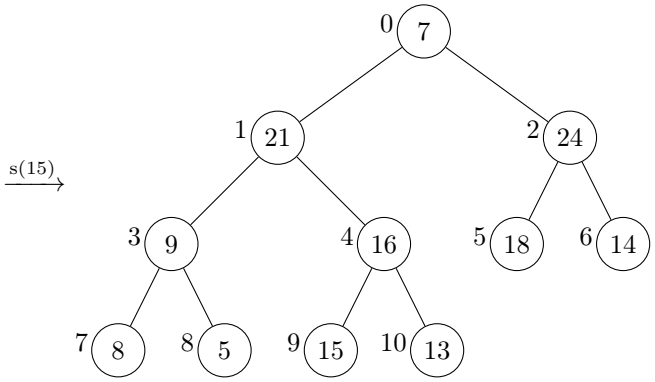
Danach wird die 8 auf Position 3 betrachtet und sinkengelassen.



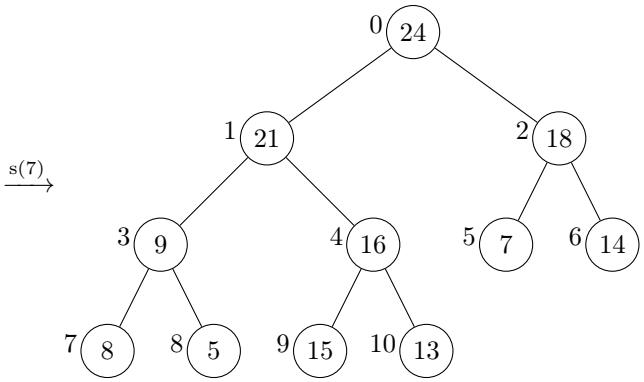
Jetzt die 14 von Position 2:



Nun 15 von Position 1:



Und schließlich die 7 von Position 0:



Hiermit ist die erste Phase von Heapsort abgeschlossen. □

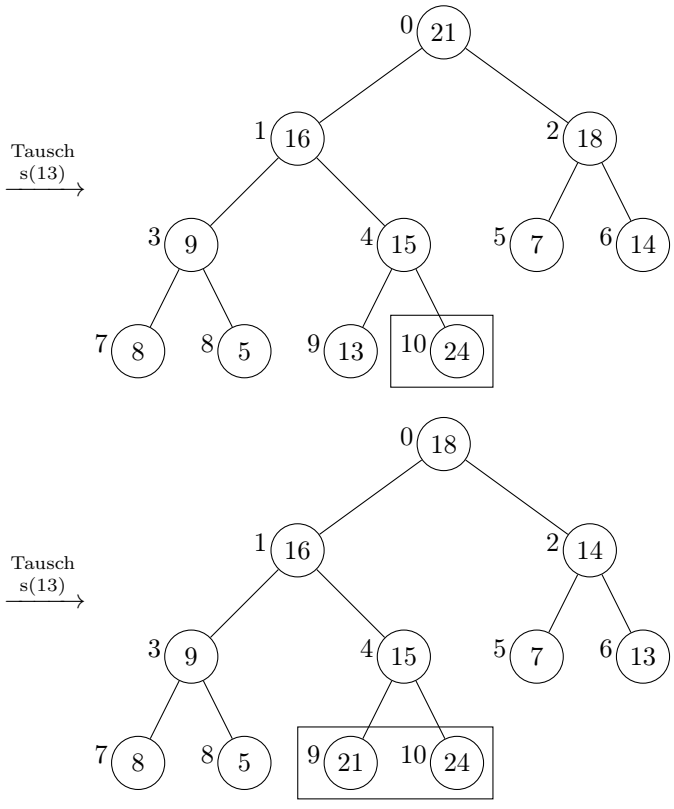
In der zweiten Phase gibt Heapsort die gespeicherten Werte in sortierter Reihenfolge aus. Unter „Ausgabe“ wird hier die Anordnung der Zahlen im Feld a verstanden, so dass das Feld a sortiert ist.

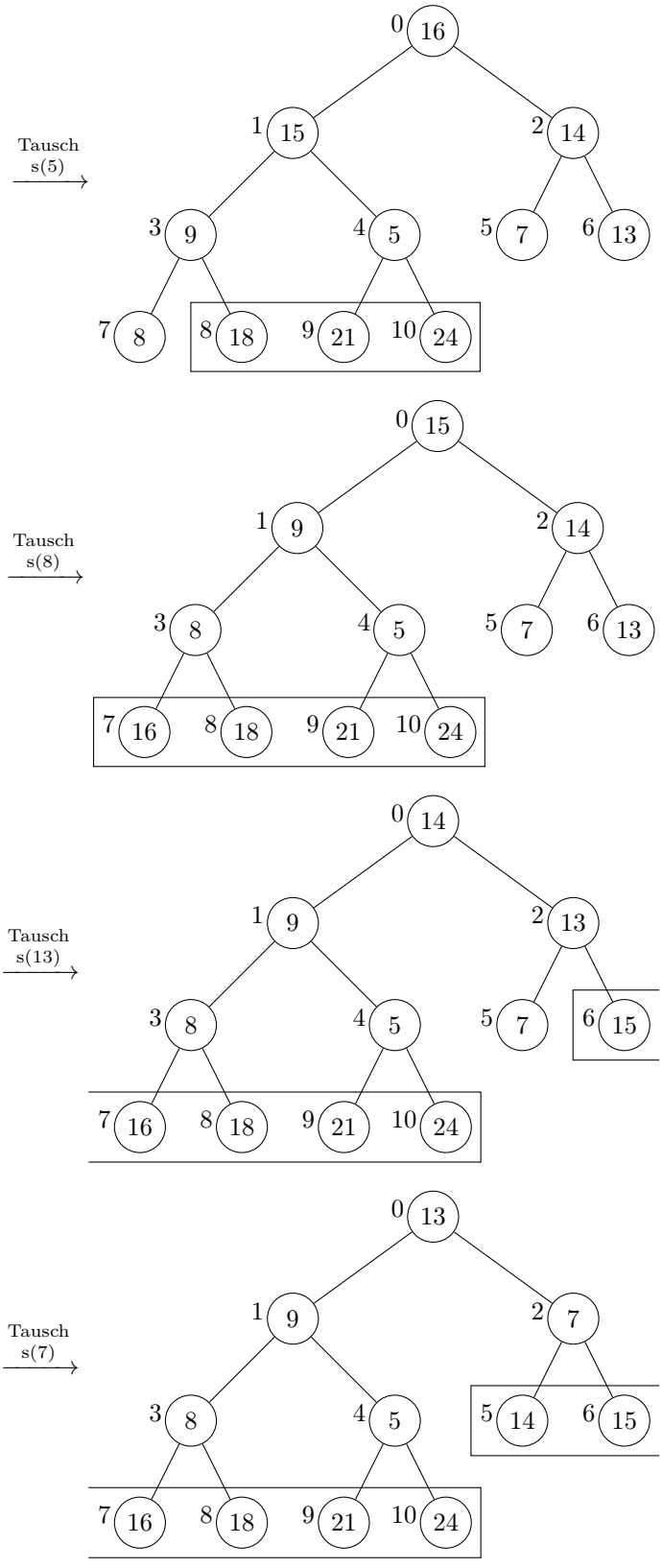
In dieser zweiten Phase wiederholt *heapsort* die Sequenz der folgenden Aktionen (dazu sei $re = n - 1$).

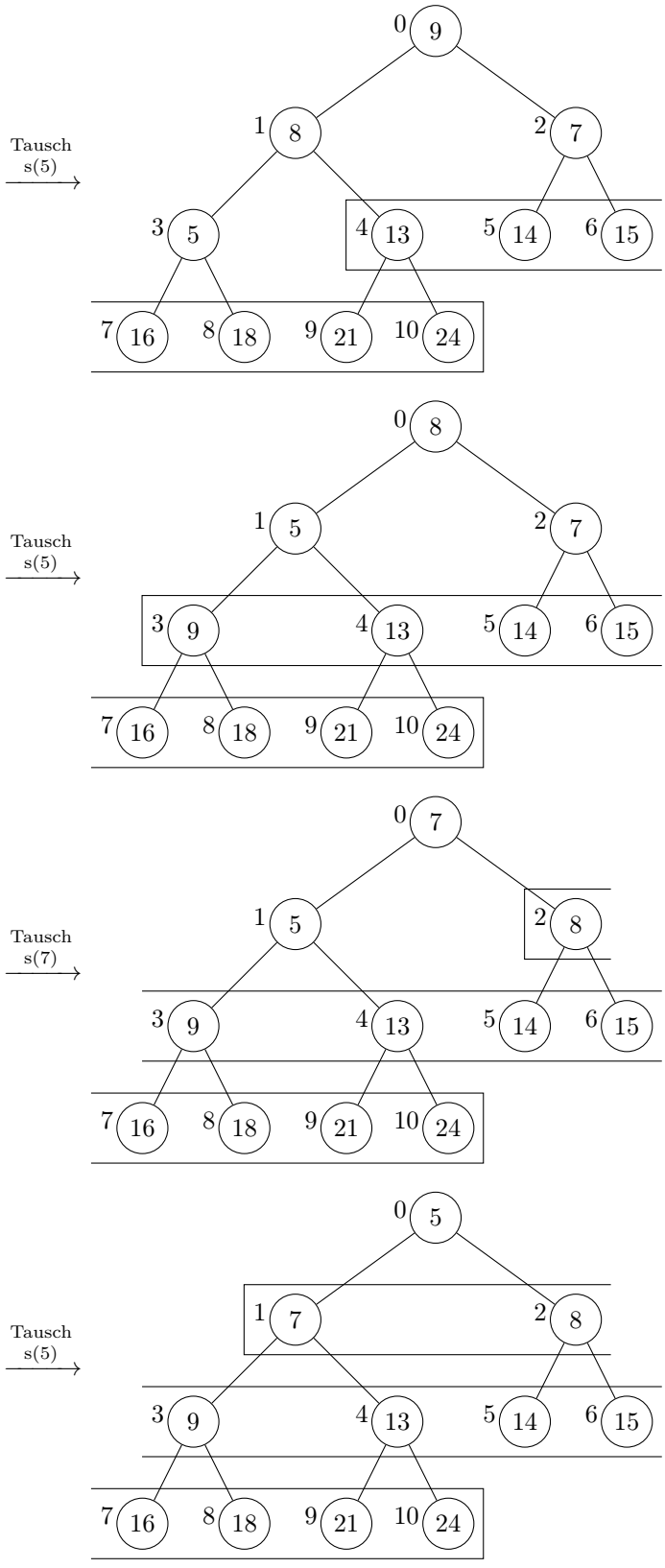
- Austausch der Elemente, die an der Wurzel bzw. an der Position re stehen
- Dekrementieren von re
- Sinkenlassen des Wurzelelements

Die Sequenz dieser drei Aktionen nennen wir einen *Sortierschritt*. Somit wird das sortierte Feld von der Position $n - 1$ beginnend zur Position 0 fortschreitend aufgebaut.

Beispiel 9.2 (Fortsetzung)**.** Führen wir diese zweite Phase an unserem Beispiel durch und betrachten die Zwischenergebnisse nach jedem Sortierschritt. Der „Ausgabeteil“ ist in einem Kasten eingefasst. Die Funktion *sinkenlassen* darf diesen natürlich *nicht* mehr betreten.




Tausch
s(5)
0 16
1 15
2 14
3 9
4 5
5 7
6 13
7 8
8 18
9 21
10 24
Tausch
s(8)
0 15
1 9
2 14
3 8
4 5
5 7
6 13
7 16
8 18
9 21
10 24
Tausch
s(13)
0 14
1 9
2 13
3 8
4 5
5 7
6 15
7 16
8 18
9 21
10 24
Tausch
s(7)
0 13
1 9
2 7
3 8
4 5
5 14
6 15
7 16
8 18
9 21
10 24



Damit ist die zweite Phase abgeschlossen, und das Feld a ist sortiert. □

Zeitkomplexität von Heapsort:

Sei n die Anzahl der zu sortierenden Elemente. Gesucht wird nun eine möglichst kleine obere Schranke f, so dass $O(f)$ die worst-case-Komplexität des Algorithmus beschreibt.

Betrachten wir zuerst Phase 1. Für $\frac{n}{2}$ Knoten wird die Funktion *sinkenlassen* ausgeführt. Der Aufwand der Funktion *sinkenlassen* ergibt sich folgendermaßen: Entlang eines Pfades, der die maximale Länge $\log_2 n$ haben kann (ein (fast) vollständiger Baum hat die Tiefe $\log_2 n$), werden die jeweiligen Austauschoperationen ausgeführt. Wenn man außerdem beachtet, dass $\log_2 n = a \cdot \log n$ mit $a = \frac{1}{\log 2}$, erhält man $O(\log n)$. Für den Gesamtaufwand in Phase 1 ergibt sich somit: $\frac{n}{2} \cdot O(\log n) = O(\frac{n}{2} \cdot \log n) = O(n \cdot \log n)$.

In Phase 2 werden für n Knoten jeweils ein Austauschschritt (mit Aufwand b) und jeweils einmal die Funktion *sinkenlassen* ausgeführt. Daraus ergibt sich: $n \cdot (b + O(\log n)) = n \cdot O(\log n) = O(n \cdot \log n)$.

Für den gesamten Algorithmus ergibt sich die Komplexität als Summe von Phase 1 und 2, also:
$O(n \cdot \log n) + O(n \cdot \log n) = O(n \cdot \log n)$.

Zusatzbemerkung: Da spezifische Eigenschaften der zu sortierenden Folge nur im Aufwand $O(c)$ berücksichtigt werden, gilt die eben berechnete Komplexität von *Heapsort* unabhängig von der Ausprägung der Eingabefolge.

Inbesondere gilt somit: $\Omega(n \cdot \log n) = O(n \cdot \log n) = \Theta(n \cdot \log n)$.

Als letztes geben wir das Programmfragment an, welches den Heapsort-Algorithmus realisiert:

Die Invariante für die zweite Phase lautet:
Für alle re mit $0 \leq re \leq n-1$ gilt

1. $a[re+1] < a[re+2] < ... < a[n-1]$ (Postfix der sortierten Folge)
2. $a[0] < a[re+1]$ falls $re < n-1$
3. für alle j mit $0 \leq j \leq (re+1)$ DIV $2 - 1$ gilt: $a[j] > a[2*j+1]$ und ggf. $a[j] > a[2*j+2]$ (*heap*-Eigenschaft)

```
. . .
#define K 100

/* lasse a[l] in a[l],a[l+1],...,a[r] hineinsinken */
void sinkenlassen(int a[], int l, int r)
{ int i, j, h, loop;

  i = l;
  h = a[i];
  loop = 1;
  while (loop)
  { j = 2*i+1;              /* gehe zum linken Nachfolger von i */
    if (j > r)
      break;

    if (j < r)
      if (a[j] < a[j+1])
        j = j+1;            /* rechter Nachfolger a[j+1] ist       */
                            /* groesser als linker Nachfolger a[j] */
    if (h > a[j])
      break;
    else
    { a[i] = a[j];          /* von j nach i sinkenlassen */
      i = j;
    }
  }
  a[i]= h;
}

void Heapsort(int a[], int n)
{ int li, re, x;

  li = n / 2;
  re = n-1;
  while (li > 0)      /* Phase 1 */
  { li = li-1;              /* rechte Feldgrenze re = n-1 bleibt konstant, */
    sinkenlassen(a, li, re); /* linke Feldgrenze li wird dekrementiert      */
  }
  while (re > 0)      /* Phase 2 */
  { x = a[0];
    a[0] = a[re];
    a[re] = x;
    re = re-1;
    sinkenlassen(a, 0, re);  /* linke Feldgrenze 0 bleibt konstant,       */
  }                          /* rechte Feldgrenze re wird dekrementiert   */
}

int main()
{ int a[K];
  /* Werte fuer a[0] bis a[K-1] eingeben */
  . . .
  Heapsort(a, K);
  . . .
}
```

10 Suchen und Ersetzen

Sobald ein großer Datenbestand gegeben ist, entsteht das Problem, nach einem bestimmten Objekt in diesem Bestand zu suchen. Wir wollen hier zwei Situationen unterscheiden, nämlich dass

- der Datenbestand eine feste Größe hat und
- der Datenbestand eine veränderbare Größe hat.

Die geeigneten Datenstrukturen sind dann vom Typ ARRAY bzw. dynamische Datenstrukturen. Beim Suchen in Datenbeständen fester Größe unterscheiden wir zwischen dem Suchen nach einem Schlüssel (siehe Einführung zu Kapitel 9) oder dem Suchen nach einem Wort in einem Text.

10.1 Suchen von Schlüsseln in festen Datenbeständen

Wir nehmen an, dass die gespeicherten Daten strukturiert sind und es eine Komponente namens `key` gibt, durch die die Daten eindeutig identifizierbar sind.

```
1 typedef struct Feld { int key;
2                       ... contents; } FeldTyp;
3
4 FeldTyp F[Flaenge];
5 int  Wert;
```

Der naive Algorithmus zum Suchen einer Position `i` in `F` mit `F[i].key = Wert` ist das lineare Suchen. Der Aufwand zum Finden einer Position ist also von der Ordnung $O(n)$.

```
1 i = 0;
2 while ((i < Flaenge) && (F[i].key != Wert))
3   i = i+1;
4
5 gefunden = (i < Flaenge);
```

Durch Einführung eines sogenannten Wächterelements kann die Abfrage `(i < Flaenge)` wegfallen.

```
1 FeldTyp F[Flaenge+1];
2 int Wert;
3 . . .
4 i = 0;
5 F[Flaenge].key = Wert;
6 while (F[i].key != Wert) i=i+1;
7 gefunden = (i < Flaenge);
```

Wenn man voraussetzt, dass das Feld `F` bezüglich der Komponente `key` aufsteigend sortiert ist, so lässt sich eine viel effizientere Methode benutzen, das binäre Suchen. Dieses Suchen hat einen Aufwand der Ordnung $O(log\ n)$. Wir geben dazu einen rekursiven und einen iterativen Algorithmus an.

Rekursiver Algorithmus:

```
 1  int SearchRec(FeldTyp F[], int links, int rechts, int wert)
 2  { int pos;
 3
 4    if (links > rechts)
 5      return 0;     /* FALSE */
 6    pos = (links+rechts) / 2;
 7    if (F[pos].key == wert)
 8      return 1; /* TRUE */
 9    if (F[pos].key < wert)
10      return SearchRec(F, pos+1, rechts, wert);
11    else
12      return SearchRec(F, links, pos-1, wert);
13  }
14
15  int main()
16  { int gefunden;
17    . . .
18    gefunden = SearchRec(F,0,Flaenge-1,Wert);
19    . . .
20  }
```

Iterativer Algorithmus:

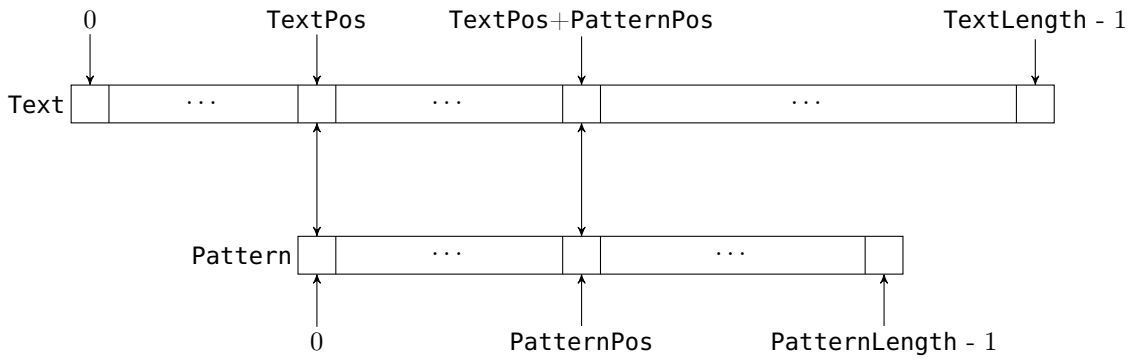
```
 1  gefunden = 0;     /* FALSE */
 2  links = 0; rechts = Flaenge-1;
 3
 4  while ((links <= rechts) && !gefunden)
 5  { pos = (links+rechts) / 2;
 6    if (F[pos].key == Wert)
 7      gefunden = 1;  /* TRUE */
 8    else
 9      if (F[pos].key < Wert)
10        links = pos+1;
11      else
12        rechts = pos-1;
13  }
```

10.2 Suchen von Mustern in Texten

Jetzt sei ein Text (d. h. eine Zeichenreihe) und ein Muster (engl.: Pattern; dies ist ebenfalls eine Zeichenreihe) gegeben. Gesucht ist nun eine Position im Text, an welcher das Pattern zu dem dort beginnenden Text passt; d. h. gesucht ist ein Index j mit $\text{Text}[j+i] = \text{Pattern}[i]$ für alle i mit $0 \leq i \leq \text{Patternlänge} - 1$.

Naiver Algorithmus

Das Pattern wird zeichenweise am Text vorbeigeschoben; in jeder Position werden die Symbole des Patterns und des Textes nacheinander paarweise verglichen bis entweder das gesamte Pattern mit einem Textteil verglichen wurde (Erfolg) oder an einer Position im Pattern eine Ungleichheit festgestellt wird (vgl. Abbildung).




```
1  /* alle skalaren Variablen sind vom Typ int, die Strings Text und Pattern
2     sind ab 0 bis TextLength-1 bzw. PatternLength-1 indiziert.            */
3
4  TextPos = 0; PatternPos = 0;
5
6  while ((PatternPos < PatternLength) && (TextPos+PatternLength <= TextLength))
7  { PatternPos = 0;
8    while ((PatternPos < PatternLength) &&
9             (Pattern[PatternPos] == Text[TextPos + PatternPos]))  /* (*) */
10     PatternPos = PatternPos + 1;
11   TextPos = TextPos + 1;
12 }
13
14 if (TextPos > 0) TextPos = TextPos - 1;
15 if (PatternPos == PatternLength)
16 { gefunden = 1;        /* TRUE */
17   printf("Pattern beginnt an Position: %d", TextPos);
18 }
19 else
20 { gefunden = 0;        /* FALSE */
21   printf("Pattern nicht gefunden");
22 }
```

Man beachte, dass der maximale Index von `Pattern` gleich `PatternLength - 1` ist und dass bei C (hier in der Testbedingung `(*)`) die Auswertung auf dem *kurzen Weg* realisiert wird. D. h. eine Konjunktion wird von links nach rechts solange ausgewertet, bis erstmals der Wahrheitswert *false* auftritt, dann Abbruch der Auswertung und Rückgabe von *false*, andernfalls Rückgabe von *true*.

Aufwand: Wenn der Text die Länge r hat und das Pattern die Länge n, dann werden im ungünstigsten Fall $(r - n + 1) \cdot n$ Vergleiche ausgeführt. Die worst-case-Komplexität dieses naiven Suchverfahrens ist demzufolge: $O(r \cdot n)$.

Algorithmus nach Knuth-Morris-Pratt (KMP)

Die Idee dieses Algorithmus basiert auf dem Wunsch, das Pattern bei Nichtübereinstimmung um möglichst viele, d. h. insbesondere um mehr als eine Position nach rechts zu verschieben. Dazu werden aus dem Pattern bestimmte Verschiebeinformationen hergeleitet; dabei kann ausgenutzt werden, dass bei Nichtübereinstimmung z. B. an der 5. Position des Patterns (`PatternPos` $= 4$) die vier Symbole `Text[TextPos]`, `Text[TextPos + 1]`, `Text[TextPos + 2]`, `Text[TextPos + 3]` bereits bekannt sind.

Beispiel 10.1.

```
Text:    G E G E G E B E N E N F A L L S  _ ...
Pattern: G E G E B E N
             G E G E B E N
```

Offensichtlich kann das Pattern schon direkt um zwei Symbole verschoben werden. □

Die zulässigen Verschiebungen werden ausschließlich aus dem Pattern selbst gewonnen, der Text ist dafür nicht notwendig. Also besitzt der KMP-Algorithmus zwei Phasen:

1. Es wird eine Tabelle aufgebaut, aus der für jede Position im Pattern die Verschiebeinformation bei Unstimmigkeit an dieser Position hervorgeht.
2. Der Text wird durchlaufen und mit den Symbolen des Pattern verglichen, wobei bei Unstimmigkeiten die Verschiebeinformation aus der Tabelle benutzt wird.

Für das Pattern `G E G E B E N` spielen wir zunächst mehrere Texte durch, an denen wir experimentell Gebrauch und Wesensmerkmale der Verschiebeinformationen studieren können. Wir gehen davon aus, dass nach jeder Verschiebung die Textposition und die Patternposition um eins erhöht werden, d. h. dass wir eine fiktive Position -1 im Pattern benötigen.

Tabelle[i]:= j bedeutet: Bei Unstimmigkeit an der Patternposition i, verschiebe das Pattern soweit, dass die Patternposition j auf der aktuellen Textposition steht.

Beispiel 10.2.

```
Textposition:          !
Text:               ...N I C H ...
Pattern:               G E G E B E N
Patternposition:       !
- Tabelle[0] := -1

Textposition:          !
Text:            ... G G E G ...
Pattern:             G E G E B E N
Patternposition:       !
- Tabelle[1] := 0

Textposition:          !
Text:              G E B I R G E ...
Pattern:           G E G E B E N
Patternposition:       !
- Tabelle[2] := -1

Textposition:          !
Text:        ... G E G G E N H E I M ...
Pattern:         G E G E B E N
Patternposition:       !
- Tabelle[3] := 0'

Textposition:             !
Text:          ... G E G E G E B E N E N F A L L S ...
Pattern:           G E G E B E N
Patternposition:          !
- Tabelle[4] := 2

Textposition:                    !
Text:              ... G E G E B G ...
Pattern:               G E G E B E N
Patternposition:               !
- Tabelle[5] := 0

Textposition:                      !
Text:               ...G E G E B E G  ...
Pattern:               G E G E B E N
Patternposition:                   !
- Tabelle[6] := 0
```

Es gibt also folgende Verschiebetabelle

```
Pattern:              G  E  G  E  B  E  N
Position:             0  1  2  3  4  5  6

Tabelle:             -1  0 -1  0  2  0  0
```

□

Allgemein gilt also folgender Zusammenhang:

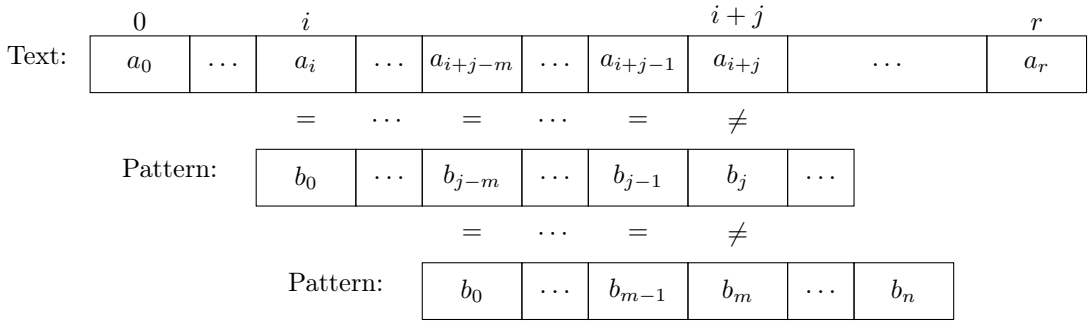
Wenn das Pattern $b_0 \dots b_n$ am Text $a_0 \dots a_r$ an der Textposition i angelegt und verglichen wird, und es besteht Gleichheit bis einschließlich Patternposition $j-1$ und Ungleichheit an Patternposition j (d. h. für alle k mit $0 \leq k \leq j-1$ gilt $b_k = a_{i+k}$ und $b_j \neq a_{i+j}$), dann verschiebe das Pattern so, dass die Patternposition l auf Textposition $i+j$ zu stehen kommt, wobei

$$l = \max(\{-1\} \cup \{m \mid 0 \leq m \leq j-1 \text{ und } (b_0 \dots b_{m-1}) = (a_{i+j-m} \dots a_{i+j-1}) \text{ und } b_m \neq b_j\})$$

Hieraus lässt sich nun das Konstruktionsprinzip für die Patterntabelle ableiten. Für jede Position j der zu erstellenden Verschiebetabelle muss nämlich gelten:

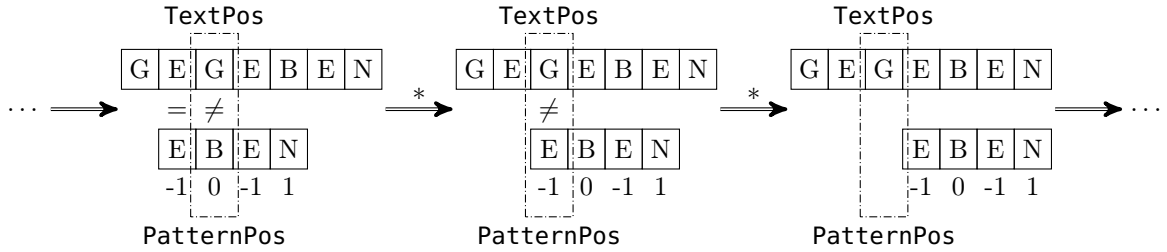
$$\text{Tabelle[j]} = \max(\{-1\} \cup \{m \mid 0 \leq m \leq j-1 \text{ und } (b_0 \dots b_{m-1}) = (b_{j-m} \dots b_{j-1}) \text{ und } b_m \neq b_j\})$$

(Beachte: Für $m = 0$ gilt $(b_0 \dots b_{m-1}) = \varepsilon$, d. h. wir erhalten das leere Wort.)



Das Verschieben kann auch aus mehreren elementaren Verschiebeoperationen bestehen (siehe Beispiel 10.3).

Beispiel 10.3. Bei der Suche des Patterns `EBEN` in `GEGEBEN` kommt es zu zwei direkt aufeinander folgenden elementaren Verschiebeoperationen (jeweils mit $*$ gekennzeichnet) bevor `TextPos` bewegt wird:



□

Funktion zur Berechnung der Tabelleneinträge:

```
1  void TabelleBauen()
2  { int PatPos;          /* durchlaeuft Pattern bis zum
3                            letzten Zeichen */
4    int VglInd;          /* Laenge des linken Teilpatterns,
5                            das mit Patternanfang uebereinstimmt */
6
7    Tabelle[0] = -1;
8    VglInd = 0;
9
10   for (PatPos = 1; PatPos <= PatternLength-1; PatPos = PatPos+1)
11   { if (Pattern[PatPos] == Pattern[VglInd])
12       Tabelle[PatPos] = Tabelle[VglInd];
13     else
14       Tabelle[PatPos] = VglInd;
15
16     while ((VglInd >= 0) && (Pattern[PatPos] != Pattern[VglInd]))
17       VglInd = Tabelle[VglInd];
18
19     VglInd = VglInd+1;
20   }
21 }
```

Nun kann das Hauptprogramm des KMP-Algorithmus aufgeschrieben werden.

```
1  { TabelleBauen();
2    TextPos = 0; PatternPos = 0;
3
4    while ((PatternPos < PatternLength) && (TextPos < TextLength))
5    { while ((PatternPos >= 0) && (Text[TextPos] != Pattern[PatternPos]))
6        PatternPos = Tabelle[PatternPos];
7
8      TextPos = TextPos+1;
9      PatternPos = PatternPos+1;
10   }
11
12   if (PatternPos == PatternLength)
13     printf("Pattern gefunden an Position %d", TextPos-PatternLength);
14   else
15     printf("Pattern nicht gefunden");
16 }
```

Hat unser Pattern eine Länge von n und unser Text eine Länge von r, so benötigt der KMP-Algorithmus höchstens $O(n + r)$ Schritte.
Zur Erinnerung: Bei dem naiven Suchen waren es $O(n \cdot r)$.

10.3 Korrektur von Schreibfehlern

Neben dem Auffinden eines Wortes in einem Text ist die semiautomatische Korrektur von Schreibfehlern eine wichtige Funktion in Textverarbeitungssystemen; diese Funktion nennt man *Spellchecking*. Ein Spellchecker für die natürliche Sprache L nimmt einen Text (d.h. eine Sequenz von Wörtern) und geht die Wörter der Reihe nach durch. Bei jedem Wort w prüft der Spellchecker, ob es ein Element des Vokabulars M der Sprache L ist oder nicht, d.h. ob es richtig geschrieben ist oder nicht. Wenn w in M liegt, dann geht der Spellchecker zum nächsten Wort. Wenn $w \notin M$, dann bietet er dem Benutzer eine Liste von Kandidaten für richtig geschriebene Wörter an; manchmal kann man die Anzahl k der Kandidaten einstellen. Wie kommt der Spellchecker zu dieser Liste? Grob gesagt geschieht das folgendermaßen. Der Spellchecker berechnet für jedes $v \in M$ den Unterschied $d(w, v)$ zwischen w und v. Dann bietet er diejenigen k Wörter aus M als Kandidaten an, die den geringsten Unterschied zu w aufweisen (sofern k solcher Wörter gefunden werden; sonst entsprechend weniger).

Die nächste Frage ist: Was ist der Unterschied zwischen dem *Quellwort* w und dem *Zielwort* v und wie lässt sich dieser quantifizieren? Hierzu kann man die sogenannte *Minimum-Edit-Distance* verwenden:

> Die Minimum-Edit-Distance zwischen w und v ist die minimale Anzahl von Editieroperationen (Insertion, Deletion, Substitution), die gebraucht werden, um von w nach v zu kommen.

Die Editieroperationen leisten folgendes:

- Insertion (i) fügt einen Buchstaben in das Quellwort ein,
- Deletion (d) löscht einen Buchstaben aus dem Quellwort und
- Substitution (s) ersetzt einen Buchstaben des Quellwortes durch einen Buchstaben.

Als Beispiel betrachten wir das Quellwort w = intuition und das Zielwort v = nutrition. Die folgenden drei Tabellen zeigen verschiedene Möglichkeiten, um von w nach v zu kommen; die unterste Zeile enthält die jeweils angewandte Editieroperation; diese Tabellen werden *Alignments* genannt:

1)

i	n	*	t	u	i	t	i	o	n
|	|	|	|	|	|	|	|	|	|
*	n	u	t	r	i	t	i	o	n
d		i		s					

2)

i	n	t	u	i	t	i	o	n
|	|	|	|	|	|	|	|	|
n	u	t	r	i	t	i	o	n
s	s		s					

3)

i	n	t	u	*	*	*	*	i	t	i	o	n
|	|	|	|	|	|	|	|	|	|	|	|
*	*	*	*	n	u	t	r	i	t	i	o	n
d	d	d	d	i	i	i	i					

Die ersten beiden Alignments umfassen drei Editierschritte, das dritte Alignment acht.

Oft gewichtet man die verschiedenen Editieroperationen unterschiedlich, weil man eine Substitution (von a durch b mit $a \neq b$) durch eine Insertion und eine Deletion erhalten kann. Das führt zur sogenannten *(einfachen) Levenshtein-Distanz*; dabei hat die Operation

- Insertion die Kosten 1,
- Deletion die Kosten 1 und
- Substitution (von a durch b mit $a \neq b$) die Kosten 1.

In unserem Beispiel haben also die drei Alignments die Kosten 3, 3 bzw. 8.

> Die Levenshtein-Distanz zwischen w und v, bezeichnet durch $d(w, v)$, ist die Höhe der minimalen Kosten von Editieroperationen (Insertion, Deletion, Substitution), die gebraucht werden, um von w nach v zu kommen.

Da die Kosten für jede Editieroperationen gleich 1 sind, entspricht die Levenshtein-Distanz der Anzahl an Editieroperationen, die gebraucht werden, um von dem Quellwort zum Zielwort zu kommen. In unserem Beispiel gilt $d(\text{intuition}, \text{nutrition}) = 3$.

In der Tat ist die Levenshtein-Distanz eine Metrik im mathematischen Sinne, d.h. es gilt für alle Wörter w und v:

- $d(w, v) = 0$ genau dann wenn $w = v$,
- $d(w, v) = d(v, w)$ und
- $d(w, v) + d(v, u) \geq d(w, u)$ für jedes Wort u.

Im folgenden sei w ein Wort der Länge n und v ein Wort der Länge k. Für jedes j mit $0 \leq j \leq n$ kürzen wir die Sequenz der ersten j Buchstaben von w durch $w_{1,j}$ ab (insbesondere gilt also $w_{1,0} = \varepsilon$ und $w_{1,n} = w$) und für jedes i mit $0 \leq i \leq k$ die Sequenz der ersten i Buchstaben von v durch $v_{1,i}$. Den Buchstaben an Position j von w bezeichnen wir mit w_j; entsprechend benutzen wir v_i.

Nun stellt sich die Frage, wie man die Levenshtein-Distanz zwischen zwei Wörtern w und v berechnet. Die Idee dazu beruht darauf, dass sich die minimalen Kosten einer Folge von Editieroperationen, die das Teilwort $w_{1,j}$ in das Teilwort $v_{1,i}$ überführt, als das Minimum aus

1. $d(w_{1,j}, v_{1,i-1}) + 1$, d.h. den minimalen Kosten einer Folge von Editieroperationen, die $w_{1,j}$ in $v_{1,i-1}$ überführt gefolgt von Insertion von v_i,
2. $d(w_{1,j-1}, v_{1,i}) + 1$, d.h. den minimalen Kosten einer Folge von Editieroperationen, die $w_{1,j-1}$ in $v_{1,i}$ überführt gefolgt von Deletion von w_j,

3. $d(w_{1,j-1}, v_{1,i-1}) + 1$ (falls $w_j \neq v_i$), d.h. den minimalen Kosten einer Folge von Editieroperationen, die $w_{1,j-1}$ in $v_{1,i-1}$ überführt gefolgt von Substitution von w_j durch v_i,
4. $d(w_{1,j-1}, v_{1,i-1})$ (falls $w_j = v_i$), d.h. den minimalen Kosten einer Folge von Editieroperationen, die $w_{1,j-1}$ in $v_{1,i-1}$ überführt,

ergeben. Diese drei Editieroperationen sind in Abbildung 10.1 veranschaulicht. In Formeln aufgeschrieben sieht das so aus, wobei wir $d(w_{1,j}, v_{1,i})$ durch $d(j,i)$ abkürzen (und entsprechend auch andere Abkürzungen verwenden):

$$d(0,i) = i \qquad \text{für jedes } 0 \leq i \leq k,$$

$$d(j,0) = j \qquad \text{für jedes } 0 \leq j \leq n \text{ und}$$

$$d(j,i) = \min\{d(j,i-1)+1,\ d(j-1,i)+1,\ d(j-1,i-1) + \left\{ \begin{array}{cc} 1 & \text{wenn } w_j \neq v_i \\ 0 & \text{sonst} \end{array} \right\}\}$$

für jedes $1 \leq j \leq n$ und jedes $1 \leq i \leq k$.

Wir können d als $((n+1)\times(k+1))$-Matrix auffassen. Jeder Matrixeintrag $d(j,i)$ wird eindeutig durch die drei benachbarten Matrixeinträge $d(j,i-1)$, $d(j-1,i)$ und $d(j-1,i-1)$ bestimmt. Deshalb kann man die Matrix d z.B. zeilenweise oder spaltenweise füllen; man kann aber auch immer die nächste Diagonale berechnen. Hier ist die Matrix für unser Quellwort intuition und Zielwort nutrition.

$d(j,i)$			v								
		n	u	t	r	i	t	i	o	n
	0	→ 1	→ 2	→ 3	→ 4	→ 5	→ 6	→ 7	→ 8	→ 9
	↓	↘	↘	↘	↘	↘		↘		
i	1	1	→ 2	→ 3	→ 4	4	→ 5	→ 6	→ 7	→ 8
	↓	↘	↘	↘	↘	↘↓	↘	↘	↘	↘
n	2	1	→ 2	→ 3	→ 4	→ 5	5	→ 6	→ 7	7
	↓	↓	↘	↘			↘	↘	↘	↘↓
t	3	2	2	2	→ 3	→ 4	→ 5	→ 6	→ 7	→ 8
	↓	↓	↘	↘↓	↘	↘	↘	↘	↘	↘
u	4	3	2	→ 3	3	→ 4	→ 5	→ 6	→ 7	→ 8
w	↓	↓	↓	↘	↘↓	↘		↘		
i	5	4	3	3	→ 4	3	→ 4	→ 5	→ 6	→ 7
	↓	↓	↓	↘	↘	↓	↘			
t	6	5	4	3	→ 4	4	3	→ 4	→ 5	→ 6
	↓	↓	↓	↓	↘	↘	↓	↘		
i	7	6	5	4	4	4	4	3	→ 4	→ 5
	↓	↓	↓	↓	↘↓	↘↓	↘↓	↓	↘	
o	8	7	6	5	5	5	5	4	3	→ 4
	↓	↘↓	↓	↓	↘↓	↘↓	↘↓	↓	↓	↘
n	9	8	7	6	6	6	6	5	4	3

Neben der Levenshtein-Distanz $d(j,i)$ enthält der Matrixeintrag an der Stelle (j,i) noch Pfeile. Jeder Pfeil gibt an, welche der drei Nachbareinträge $d(j,i-1)$, $d(j-1,i)$ und $d(j-1,i-1)$ zum Wert von $d(j,i)$ geführt hat. Zum Beispiel gilt

$$d(4,3) = d(3,3) + 1 = d(4,2) + 1 = d(3,2) + 1 \ .$$

Deshalb zeigen drei Pfeile auf $d(4,3)$. Oder:

$$d(3,3) = d(2,2) < \min\{d(3,2)+1,\ d(2,3)+1\} \ .$$

Deshalb zeigt auf $d(3,3)$ nur ein Pfeil. Die Pfeile lassen sich bequem während des Aufbaus der Matrix mitberechnen.

Wie wir am Beispiel sehen, ist die Höhe der minimalen Kosten $d(9,9)$ einer Folge von Editieroperationen, die das Quellwort intuition in das Zielwort nutrition überführt, der Wert 3. Aus den Pfeilen lassen sich

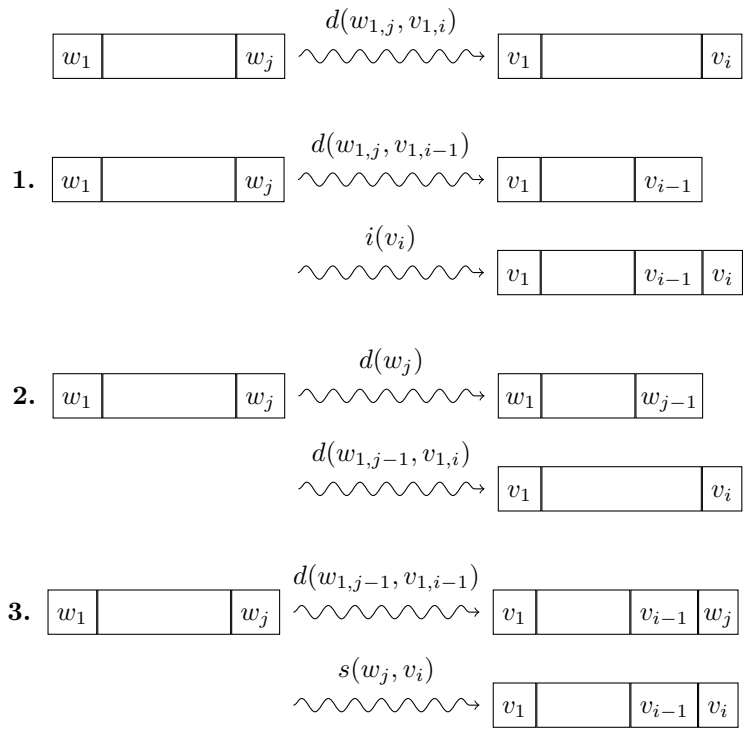


Abbildung 10.1: Editieroperationen für die Berechnung der Levenshtein-Distanz.

nun Alignments konstruieren, die zu diesen minimalen Kosten geführt haben. Dazu starten wir am Matrixeintrag $d(n, k)$ (unten rechts) und verfolgen die Pfeile rückwärts. Wenn wir dabei an eine Gabelung kommen (d.h. der Matrixeintrag $d(j, i)$ hat mehrere eingehende Pfeile), dann können wir irgendeinen dieser Pfeile zurückverfolgen. Man nennt jeden so erhaltenen Weg auch *Backtrace*. Jeder Backtrace ergibt ein Alignment. Umgekehrt entspricht jedem Alignment genau ein Backtrace. Hier sind die zwei verschiedenen Alignments für unser Beispielpaar, welche die minimale Levenshtein-Distanz haben:

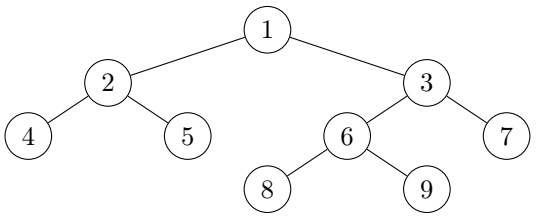
1)
```
i n * t u i t i o n
| | | | | | | | | |
* n u t r i t i o n
  d   i   s
```

2)
```
i n t u i t i o n
| | | | | | | | |
n u t r i t i o n
s s   s
```

11 Bäume

> „Bäume gehören zu den wichtigsten in der Informatik auftretenden Datenstrukturen, Entscheidungsbäume, Ableitungsbäume, Kodebäume, spannende Bäume, baumartig strukturierte Suchräume, Suchbäume und viele andere belegen die Allgegenwart von Bäumen.“ [OW02, Seite 251].

In diesem Kapitel wollen wir zunächst die notwendigen Grundbegriffe einführen und dann verschiedene Ausprägungen betrachten. Die folgende Abbildung zeigt ein Beispiel für einen Baum t.



Knoten 1 heißt *Wurzel* von t; Knoten 4, 5, 8, 9 und 7 sind die *Blätter* von t; jeder Knoten, der kein Blatt ist, heißt *innerer Knoten*; Knoten 6 und 4 sind die *ersten Nachfolger* der Knoten 3 bzw. 2; Knoten 5 und 9 sind die *zweiten Nachfolger* der Knoten 2 bzw. 6; Knoten 6 heißt *Vorgänger* des Knotens 9.

Zu jedem Knoten n gibt es genau einen *Weg (oder: Pfad)* von der Wurzel nach n; z. B. der Weg von der Wurzel zum Knoten 6 lautet $(1, 3, 6)$. Die *Tiefe von n* ist die Anzahl der Kanten auf dem Weg von der Wurzel zum Knoten n. Z. B. haben die Knoten 5 und 6 die Tiefe 2, der Knoten 1 hat die Tiefe 0.

Durch jeden Knoten n wird ein *Teilbaum von t* definiert, z. B. legt der Knoten 3 den Teilbaum fest, der aus den Knoten 3, 6, 7, 8 und 9 besteht.

Oft lassen wir bei der Darstellung von Bäumen die Bezeichnungen der Knoten auch weg.

Sei $d \geq 1$. Ein Baum t hat *die Ordnung d* wenn jeder innere Knoten von t höchstens d Nachfolger hat. Manchmal trifft man in der Literatur auch die strengere Bedingung an, dass jeder innere Knoten *genau d* Nachfolger hat. Unser Beispielbaum hat also die Ordnung 2, sowohl im normalen als auch im strengen Sinn. Bäume der Ordnung $d = 2$ heißen *Binärbäume*, Bäume der Ordnung $d > 2$ nennt man auch *Vielwegbäume*.

Die Höhe $h(t) \in \mathbb{N}$ eines Baumes t ist induktiv über der Struktur von t definiert:

$$h(\bigcirc) = 1 \qquad h\left(\begin{array}{c}\bigcirc\\ t_1 \cdots t_k\end{array}\right) = \max\{h(t_1), \ldots, h(t_k)\} + 1$$

Man faßt die Knoten eines Baumes gleicher Tiefe zu einem *Niveau* zusammen. Ein Baum heißt *vollständig*, wenn auf jedem Niveau die maximal möglichen Knoten existieren und alle Blätter dieselbe Tiefe haben.

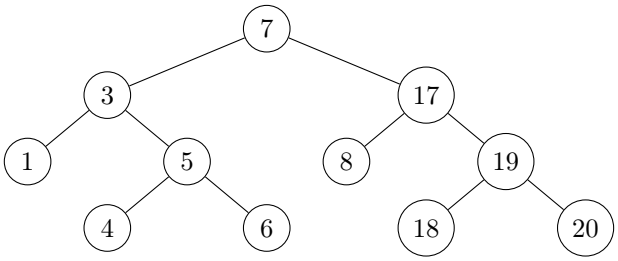
Im Rahmen dieses Kapitels werden die Knoten oft mit Schlüsseln beschriftet sein. Hier wollen wir der Einfachheit halber annehmen, dass die Schlüssel ganze Zahlen sind.

11.1 Suchbäume

Bäume lassen sich sehr gut als Datenstruktur zum Ablegen und Suchen von Objekten, die durch einen Schlüssel identifiziert sind, nutzen. Allgemein können die Knoten eines Baumes beliebige Daten enthalten, die mit einen Schlüssel versehen sind. Wir werden im folgenden der Einfachheit halber nur die Schlüsselwerte betrachten.

Ein Binärbaum t heißt *Suchbaum*, wenn alle Knoten von t mit Schlüsseln versehen sind und wenn zusätzlich folgendes gilt: Sei n ein Knoten mit Schlüssel $s(n)$, seien t_1, t_2 die beiden Teilbäume von n; dann muss jeder in t_1 (bzw. t_2) auftretende Schlüssel kleiner (bzw. größer) als $s(n)$ sein.

Beispiel 11.1. Dieser Baum ist ein Suchbaum:



□

Für vollständige Suchbäume gilt: Der Aufwand für die Suche nach einem Element ist logarithmisch in der Anzahl der Elemente.

Nun zeigen wir

- eine dynamische Datenstruktur in C zur Realisierung von binären Suchbäumen,

```
1  typedef struct Nodeelem *Ptr;
2  typedef struct Nodeelem { int key;
3                            Ptr left, right;
4                            . . .     /* beliebige weitere Daten */ } Node;
```

- eine Funktion zum Auffinden eines Elementes in einem Suchbaum,

```
1  void suche (Ptr t, int x)
2  { if (t == NULL)
3      printf("Element liegt nicht im Baum");
4    else
5      if (t->key == x)
6        printf("Element liegt im Baum");
7      else
8        if (t->key < x)
9          suche(t->right, x);
10       else
11         suche(t->left, x);
12 }
```

- und eine Funktion zum Einfügen eines Elementes in einen Suchbaum.

```
1  void einfuegen (Ptr *t, int x)
2  { Ptr q;
3    if (*t == NULL)
4    { q = (Ptr)malloc(sizeof(Node));
5      q->key = x;
6      q->left = NULL;
7      q->right = NULL;
8      *t = q;
9    }
10   else
11     if ((*t)->key == x)
12       printf("Element liegt schon im Baum");
13     else
14       if ((*t)->key < x)
15         einfuegen(&((*t)->right), x);
16       else
17         einfuegen(&((*t)->left), x);
18 }
```

Ein Aufruf der zweiten Funktion könnte z. B. wie folgt aussehen: `einfuegen(&p, 9);`, wobei `p` eine Variable vom Typ `Ptr` ist. Zur Illustration wichtiger Beziehungen bei der Arbeit mit Zeigern, insbesondere bei Nutzung von Referenzparametern, dient Abbildung 11.1 (man beachte, dass z. B. `(*t)->key` gleichbedeutend mit `(*(*t)).key` ist).

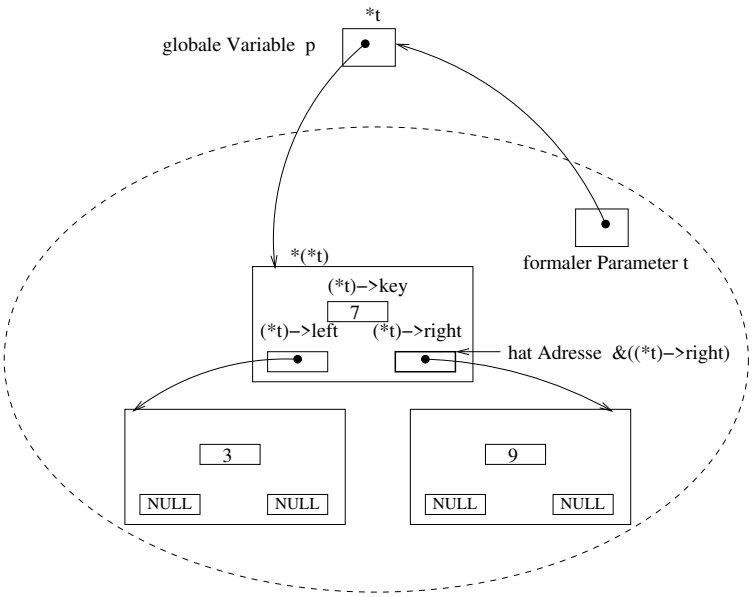


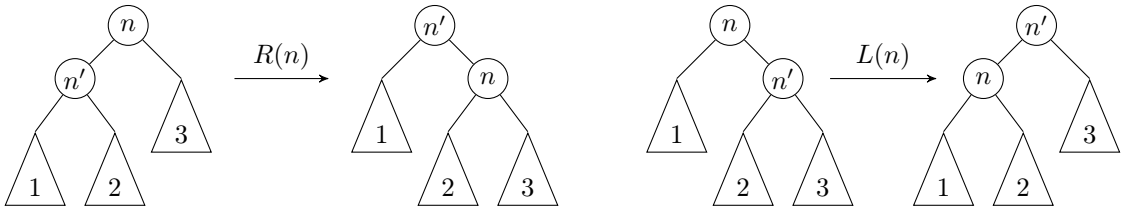
Abbildung 11.1

Günstiger Aufwand bei Suchbäumen wird durch *Vollständigkeit* bewirkt. Für einen Binärbaum, bei dem z. B. nur der rechte Nachfolger jedes Knotens belegt ist (also ein rechtslinearer Kamm), ist der Suchaufwand allerdings proportional zur Anzahl der Knoten.

11.2 Balancierte Bäume

AVL-Bäume

Bei AVL-Bäumen (benannt nach Adelson-Velskij und Landis) ist der Aufwand für die Verwaltung der Elemente (also Suchen, Einfügen, Löschen) immer proportional zum Logarithmus der Anzahl der Elemente der Menge. Allerdings muss gegebenenfalls der Baum beim Einfügen und Löschen eines Elements umstrukturiert werden. Die Umstrukturierung erfolgt durch eine Sequenz von sogenannten Rotationen; dabei gibt es Rechts- und Linksrotation (siehe Abbildung 11.2).

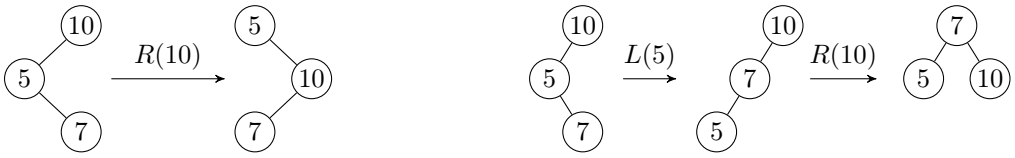


(a) Rechtsrotation um den Knoten n (b) Linksrotation um den Knoten n

Abbildung 11.2: Rechts- bzw. Linksrotation jeweils um den Knoten n.

Manchmal ist eine Doppelrotation notwendig, wie folgendes Beispiel zeigt:

Beispiel 11.2. Durch die in Abbildung 11.3a gezeigte Rechtsrotation ist nichts gewonnen. Deshalb benutzen wir eine Doppelrotation (siehe Abbildung 11.3b). □



(a) ohne Doppelrotation (b) mit Doppelrotation

Abbildung 11.3: Hier ist eine Doppelrotation nötig.

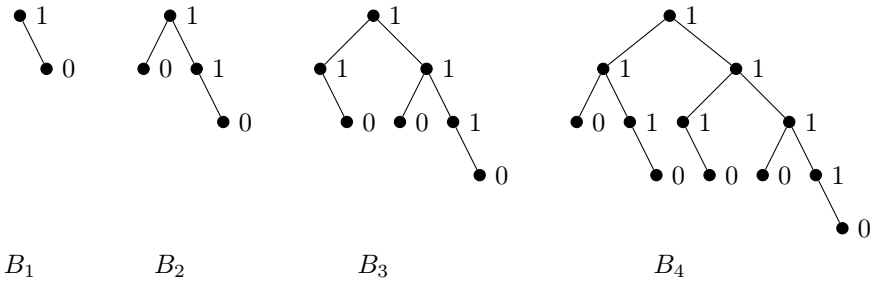


Abbildung 11.4: Beispiele für Binärbäume, die „gerade noch“ AVL-Bäume sind.

Ein *AVL-Baum* ist ein Suchbaum, bei dem an jedem Knoten n gilt $b(n) \in \{-1, 0, 1\}$, wobei $b(n)$ der *Balancefaktor* an dem Knoten n ist.
Dieser ist wie folgt definiert:

$b(n) = h(t_2) - h(t_1)$	wenn t_1 und t_2 der linke bzw. rechte Teilbaum unter n sind,
$b(n) = h(t_2)$	wenn es keinen linken Teilbaum unter n gibt und t_2 der rechte Teilbaum unter n ist,
$b(n) = -h(t_1)$	wenn t_1 der linke Teilbaum unter n ist und es keinen rechten Teilbaum unter n gibt und
$b(n) = 0$	wenn es weder einen linken noch einen rechten Teilbaum unter n gibt.

Beachte: Die Höhe eines leeren Teilbaumes ist 0.

Abbildung 11.4 zeigt binäre Bäume, die „gerade so“ noch AVL-Bäume sind. Diese AVL-Bäume werden nach dem folgenden Gesetz für $n \geq 3$ gebildet:

$$B_n = \begin{array}{c} \bullet \\ \diagup \quad \diagdown \\ B_{n-2} \quad B_{n-1} \end{array}$$

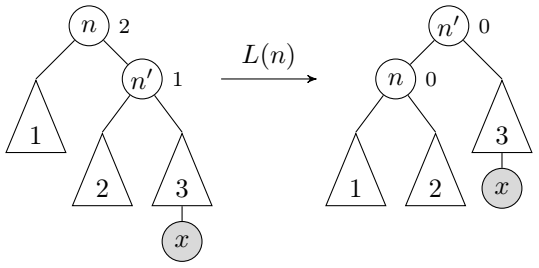
Zwischen der Tiefe t eines AVL-Baumes und der Anzahl k seiner Knoten besteht folgender Zusammenhang: $t \leq 2 \cdot \log_2 k$. Daraus folgt, dass der Suchaufwand bei AVL-Bäumen logarithmisch in der Anzahl der Knoten wächst.

Für das Einfügen in AVL-Bäume gilt allgemein folgendes:

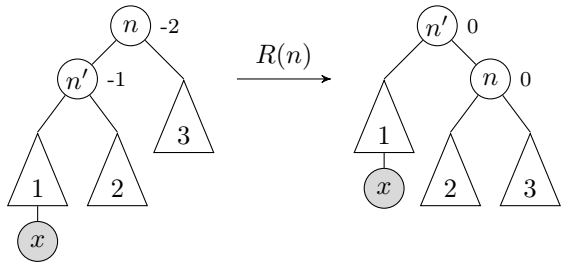
- Der neue Knoten wird (wie bei Suchbäumen) als Blatt an einem Knoten eingefügt, der höchstens einen Nachfolger hat.
- Der Balancefaktor kann nur an Knoten entlang des Suchpfades zum neuen Element verändert worden sein.

Wir werden folgende Datenstruktur nutzen um AVL-Bäume darzustellen:

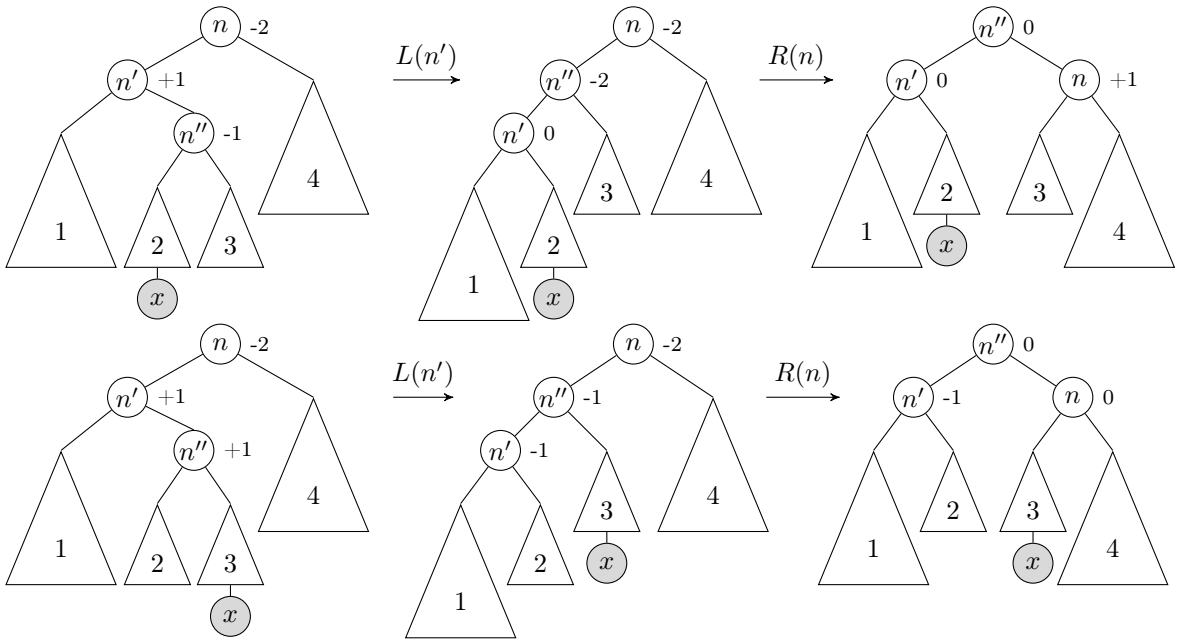
```
1 typedef struct Nodeelem *Ptr;
2
3 typedef struct Nodeelem { int key;
4                           short balance;
5                           Ptr left, right; } Node;
```



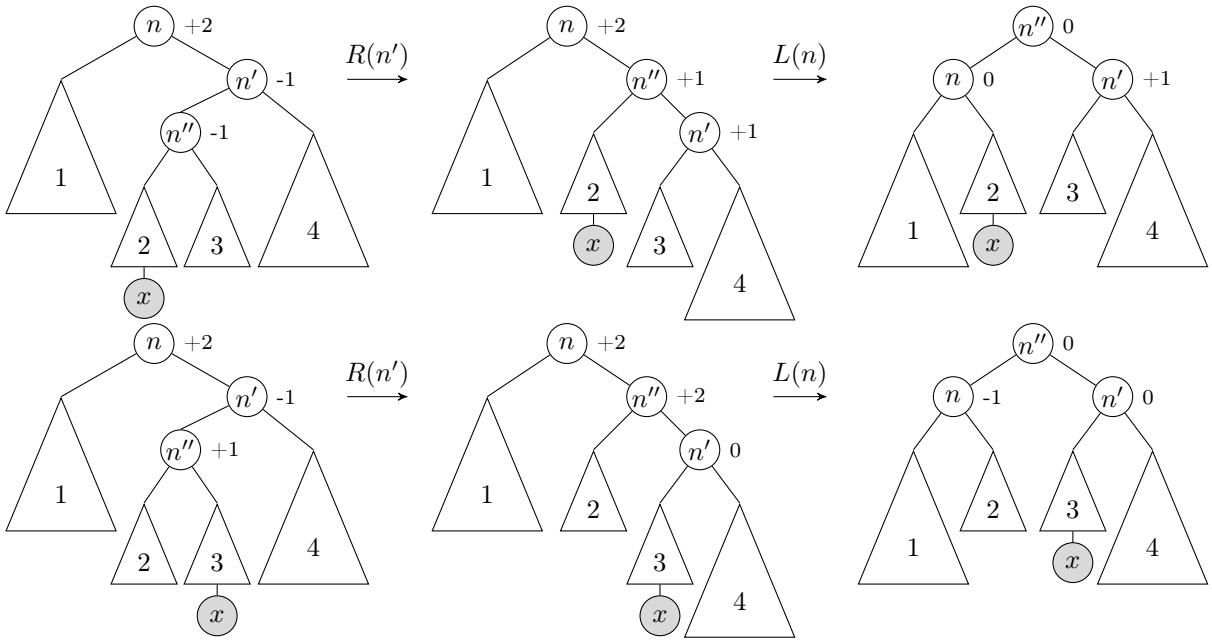
(a) Linksrotation um den Knoten n.



(b) Rechtsrotation um den Knoten n.



(c) Doppelrotation links um den Knoten n' und dann rechts um den Knoten n.



(d) Doppelrotation rechts um den Knoten n' und dann links um den Knoten n.

Algorithmus 5 Einfügen eines Elementes `x` in AVL-Bäume

Eingabe: ein AVL-Baum `t`, ein Element `x`

Ausgabe: ein AVL-Baum, der die Elemente aus `t` und das Element `x` enthält

Verfahren:

1. Füge das neue Element `x` in `t` mit Balancefaktor 0 als direkten Nachfolger des Knotens `n` als Blatt ein, so dass die Suchbaumeigenschaft erfüllt ist. Aktualisiere `n.balance`.
2. Setze `n` auf den Vorgängerknoten von `n`. (Beachte: `n` ist vom Typ `Ptr`!)
   a) Falls `x` im linken Unterbaum von `n` eingefügt wurde
      (i) wenn `n->balance == 1` dann `n->balance = 0` und gehe nach 3.
      (ii) wenn `n->balance == 0`, dann `n->balance = -1` und gehe nach 2.
      (iii) wenn `n->balance == -1` und
      - wenn `n->left->balance == -1` dann Rechtsrotation um `n` (siehe Abbildung 11.5b)
      - wenn `n->left->balance == 1`, dann erst eine Linksrotation um `n->left`, dann eine Rechtsrotation um `n` (siehe Abbildung 11.5c).

   b) Falls `x` im rechten Unterbaum von `n` eingeführt wurde
      (i) wenn `n->balance == -1`, dann `n->balance = 0` und gehe zu 3.
      (ii) wenn `n->balance == 0`, dann `n->balance = 1` und gehe zu 2.
      (iii) wenn `n->balance == 1` und
      - wenn `n->right->balance == 1` dann Linksrotation um `n` (siehe Abbildung 11.5a)
      - wenn `n->right->balance == -1` dann erst eine Rechtsrotation um `n->right`, dann eine Linksrotation um `n` (siehe Abbildung 11.5d).
3. Gehe zurück zur Wurzel von `t`.

Algorithmus 5 fügt ein Element `x` in einen AVL-Baum ein. Beachte: Der beschriebene Algorithmus erzeugt weder eine Balance von -2 bzw. 2 noch benutzt er eine Balance von -2 bzw. 2. Die in den Bildern notierten Balancen von -2 (im Falle von Algorithmenschritt 2(b) wäre es die Balance 2) sind der Korrektheit der statischen Darstellungen geschuldet.

Diese Algorithmenbeschreibung wollen wir jetzt durch die Angabe der *C*-Funktion `einfuegen_AVL` (Seite 107) ergänzen, die diese algorithmischen Teilaufgaben ausführt und sich dabei auf entsprechende Hilfsfunktionen (zur Realisierung von Teilaufgaben) abstützt.

Sei `WURZEL` der Zeiger auf einen bestehenden AVL-Baum, `WERT` der einzufügende Knotenwert, so lautet der Aufruf:

```
f_balance(WURZEL);            /* trägt die Balancewerte in den initialen Baum ein */
einfuegen_AVL(&WURZEL,WERT);
```

Bemerkung: Die hier vorgestellte Lösung repräsentiert nur die Grundideen des Algorithmus und ist deshalb gut nachvollziehbar, hat aber den Mangel, programmtechnisch nicht sehr effizient zu sein. Der versierte Programmierer würde einen wesentlich effizienteren rekursiven Algorithmus verwenden, der allerdings die zugrunde liegenden Ideen nur sehr schwer erkennen ließe (typischer Gegensatz von Effizienz und Transparenz).

```
1  typedef struct Nodeelem *Ptr;          /* Datenstruktur des AVL-Baumes */
2
3  typedef struct Nodeelem { int key;
4                            short balance;
5                            Ptr left, right; } Node;
6
7  typedef enum teilbaum{L,R} zweig;
8
9  void einfuegen(Ptr *t, int x);         /* Funktion siehe Abschnitt 11.1 */
10
11 Ptr zeiger_von_x(Ptr z_Node, int x);   /* Ermittelt den Zeiger des Knotens im AVL-
12                                            Baum, welcher x als Schlüssel hat */
13
14 Ptr vorg(Ptr z_Node, Ptr n, zweig *z); /* Ermittelt den Zeiger auf den Vorgänger des
15                                            Knotens auf den n zeigt und gibt den Zweig
16                                            (Teilbaumseite) an, in dem n liegt, also
17                                            *z == L oder *z == R. Gibt es keinen Vor-
18                                            gängerknoten, so ist die Rückgabe NULL. */
19
20 void rot(Ptr *z_Node, Ptr n, zweig y); /* Führt die Rotation um den Knoten mit Zeiger
21                                            n entsprechend Vorschrift aus: y == 'R'
22                                            Rechts-, y == 'L' Linksrotation. */
23
24 /* Beachte: Dabei kann der Kopfzeiger des (alten) AVL-Baumes geändert werden! */
25
26 void einfuegen_AVL(Ptr *wurzel, int x)
27 { zweig TB; Ptr n;
28   . . .
29   einfuegen(wurzel, x);
30   n = zeiger_von_x(*wurzel, x); n->balance = 0; n = vorg(*wurzel, n, &TB);
31
32   while (n != NULL)
33     if (TB == L)
34       switch (n->balance)
35       { case  1: n->balance = 0; return;
36         case  0: n->balance = -1; n = vorg(*wurzel, n, &TB); break;
37         case -1: switch (n->left->balance)
38                  { case  -1: rot(wurzel, n, R); return;
39                    case   1: rot(wurzel, n->left, L);
40                              rot(wurzel, n, R); return;
41                  }
42       }
43     else   /* hier gilt TB == R */
44       switch (n->balance)
45       { case -1: n->balance = 0; return;
46         case  0: n->balance = 1, n = vorg(*wurzel, n, &TB);break;
47         case  1: switch (n->right->balance)
48                  { case  1: rot(wurzel, n, L); return;
49                    case -1: rot(wurzel, n->right, R);
50                             rot(wurzel, n, L); return;
51                  }
52       }
53 }
```

```
1  void f_balance(Ptr z_Node)
2  { if (z_Node == NULL) return;
3    z_Node->balance = hoehe(z_Node->right) - hoehe(z_Node->left);
4    f_balance(z_Node->right);
5    f_balance(z_Node->left);
6  }
```

12 Graphalgorithmen

12.1 Graphen

Ein gerichteter Graph ist ein Tupel $G = (V, E)$ wobei V eine endliche Menge von Knoten (engl.: *v*ertex) und $E \subseteq V \times V$ eine Menge von Kanten (engl.: *e*dges) ist. Oft wollen wir stillschweigend annehmen, dass $V = \{1, \ldots, n\} \subseteq \mathbb{N}$. Eine Kante $(v, v) \in E$ heißt Schlinge. Ein Graph $G' = (V', E')$ ist ein Teilgraph von G, falls $V' \subseteq V$ und $E' \subseteq E \cap (V' \times V')$. Der Graph G heißt azyklisch, wenn es keinen Knoten $v \in V$ gibt, so dass vE^+v (für die Definition von E^+ siehe Anhang B.3). Insbesondere enthält ein azyklischer Graph keine Schlingen. Ein Weg von v nach v' (wobei $v, v' \in V$) ist eine Folge $(v_1, \ldots, v_n)$ von Knoten $v_1, \ldots, v_n \in V$ mit $n \geq 1$, so dass (i) $v_1 = v$ und (ii) für jedes $1 \leq i \leq n-1$ gilt: $(v_i, v_{i+1}) \in E$ und (iii) $v_n = v'$. Insbesondere ist (v) ein Weg von v nach v. Für zwei Knoten $v, v' \in V$ sagen wir, dass v' von v erreichbar ist, wenn es einen Weg von v nach v' gibt, und dass v' ein (direkter) Nachfolgerknoten von v ist, falls $(v, v') \in E$. Die Knoten $v_2, \ldots, v_{n-1}$ nennen wir innere Knoten des Weges $(v_1, \ldots, v_n)$. Ein Graph G ist zusammenhängend, falls es für alle Knoten v, $v' \in V$ einen Weg von v nach v' oder von v' nach v gibt.

Auch ungerichtete Graphen wollen wir hier betrachten. Wenn für zwei beliebige Knoten v_1, v_2 eines gerichteten Graphen G gilt:

- entweder es gibt keine Kante von v_1 nach v_2 (d. h. $(v_1, v_2) \notin E$) und es gibt keine Kante von v_2 nach v_1 (d. h. $(v_2, v_1) \notin E$),
- oder $(v_1, v_2) \in E$ und $(v_2, v_1) \in E$,

dann ist G ein ungerichteter Graph. Ein ungerichteter Graph ist also ein besonderer gerichteter Graph. Jedes Verfahren, welches wir in diesem Kapitel besprechen und welches auf gerichtete Graphen angewandt wird, kann demnach auch für ungerichtete Graphen verwendet werden. Bei ungerichteten Graphen ersetzen wir die beiden Tupel (v_1, v_2) und (v_2, v_1) in E durch die zweielementige Menge $\{v_1, v_2\}$. D. h. bei einem solchen Graphen ist $E \subseteq \{\{u, v\} \mid u, v \in V\}$. Abbildung 12.1 zeigt ein Beispiel eines gerichteten Graphen mit vier Knoten 1, 2, 3 und 4 und vier Kanten.

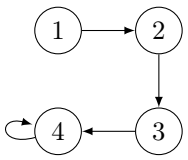


Abbildung 12.1: Beispiel eines gerichteten Graphen.

Graphen können z. B. in einer Booleschen Matrix gespeichert werden. Sei $G = (V, E)$ ein beliebiger Graph mit $n = |V|$ Knoten. Dann definiere die Adjazenzmatrix A_G von G als $(n \times n)$-Matrix über $\{0, 1\}$ durch

$$A_G(i, j) = \begin{cases} 0 & \text{falls } (i, j) \notin E \\ 1 & \text{falls } (i, j) \in E \end{cases}$$

Die Adjazenzmatrix des Graphen G aus Abbildung 12.1 sieht dann so aus:

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	1

Die transponierte Adjazenzmatrix A_G^T von G ist definiert durch $A_G^T(i,j) = A_G(j,i)$. Für ungerichtete Graphen gilt also $A_G = A_G^T$.

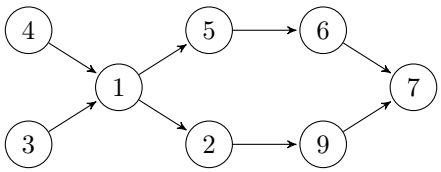
Jeder Baum t (vgl. Kap. 11) kann als gerichteter Graph G_t aufgefasst werden. Die Knoten von G_t sind dabei genau die von t, und für zwei Knoten u und v aus G_t existiert genau dann eine Kante (u,v) im Graphen G_t, wenn v ein direkter Nachfolger von u im Baum t ist.

In der mathematischen Graphentheorie nennt man einen gerichteten Graph einen *gerichteten Baum* falls er einen ausgezeichneten Knoten, den Wurzelknoten, besitzt, so dass es zu jedem Knoten vom Wurzelknoten aus genau einen Weg gibt. Wenn bei einem solchen graphentheoretischen gerichteten Baum für die Nachfolgerknoten jedes Knotens eine Ordnung vorgeben ist, entspricht diese Definition unserer Baumdefinition aus Kap. 11.

12.2 Topologisches Sortieren

In Kapitel 9 wurde nach Eigenschaften eines Elements sortiert, hier wollen wir nach Beziehungen zwischen den Elementen sortieren. Oft ist eine endliche Menge von Daten, Dingen, Vorgängen oder Objekten gegeben, bei denen manche in einer strikten Reihenfolge zu bearbeiten sind, bei anderen spielt die Reihenfolge keine Rolle. Man könnte beispielsweise an das Planen eines Festes denken und die Vorgänge sind dann: Einladungen verschicken, Antworten abwarten, Getränke bestellen, Essen bestellen, Musik bestellen .... Natürlich müssen zuerst die Einladungen verschickt werden und bevor nicht die Antworten eingetroffen sind, sollte man auch nicht das Essen oder die Getränke bestellen, weil man nicht genau weiß, wie viele zu-/absagen werden. Da aber alle Gäste Musik hören wollen (den Gastgeber eingeschlossen), kann die Musik direkt bestellt werden. Wenn man nun das Fest alleine organisiert, dann kann man natürlich die Vorgänge auch nur sequentiell bearbeiten. Es geht also darum, eine Reihenfolge der Vorgänge zu finden, bei der die oben genannten zeitlichen Abhängigkeiten berücksichtigt werden.

Abstrakt gesehen handelt es sich hier um einen gerichteten, azyklischen Graphen, bei dem die Knoten die Vorgänge repräsentieren und die Kanten die kausalen Abhängigkeiten zwischen den Vorgängen, z. B.



Man versucht nun diesen Graphen zu linearisieren, d. h. alle Knoten so in eine Reihenfolge zu bringen, dass für je zwei Knoten v, v' gilt: Wenn es einen Weg von v nach v' gibt, dann kommt v vor v' in der Knotenfolge der Wegnotierung.

z. B.

	4	3	1	2	9	5	6	7
oder	3	4	1	5	2	6	9	7
aber *nicht*	4	1	3	5	2	6	7	9

weil hier 3 *nicht* vor 1 und 9 *nicht* vor 7 steht. Im Allgemeinen lässt sich dieser Vorgang so beschreiben:

> ***Topologisches Sortieren:***
>
> Gegeben sei ein gerichteter, azyklischer Graph $G = (V, E)$. Eine topologische Sortierung von G ist eine bijektive Abbildung $ord : V \to \{1, \ldots, n\}$ mit $n = |V|$, so dass aus $(v, v') \in E$ folgt $ord(v) < ord(v')$.

Das Problem lässt sich mit Hilfe des folgenden Algorithmus in Pseudo-C lösen:

Dazu können wir die in listing 12.1 gezeigte Definition für eine dynamische Datenstruktur verwenden.

```
1  typedef struct leader *LPtr;
2  typedef struct trailer *TPtr;
3
4  typedef struct leader
5                 { int key;
6                   int count;  /* Anzahl der Vorgaenger */
```

Algorithmus 6 Topologisches Sortieren

```
while (Elemente sind noch übrig)
{ Wähle Element aus, welches keinen Vorgänger hat;
  Dekrementiere die Anzahl der Vorgänger in den Nachfolgern des ausgewählten Elements;
  Trage das Element in die gewünschte Ausgabeliste ein;
  Streiche das ausgewählte Element aus der Menge;
}
```

```
 7                 TPtr trail; /* Liste mit Zeigern zu Nachfolgeelementen
 8                                bezuegl. der Halbordnung */
 9                 LPtr next;  /* Zeiger zu Element, welches bzgl. seines
10                                Erscheinens in der Eingabe das naechste ist */
11               } leader;
12
13 typedef struct trailer
14               { LPtr id;
15                 TPtr next;
16               } trailer;
17
18 LPtr p, q, head, tail;
19 TPtr t;
```

Listing 12.1: Datenstruktur zum topologischen Sortieren.

Der gerichtete, azyklische Graph wird als Folge von Paaren eingegeben, jedes Paar repräsentiert eine Kante:

$$(6,7)\,(2,9)\,(9,7)\,(3,1)\,(4,1)\,(5,6)\,(1,5)\,(1,2).$$

Beim topologischen Sortieren gibt es folgende Teilaufgaben:

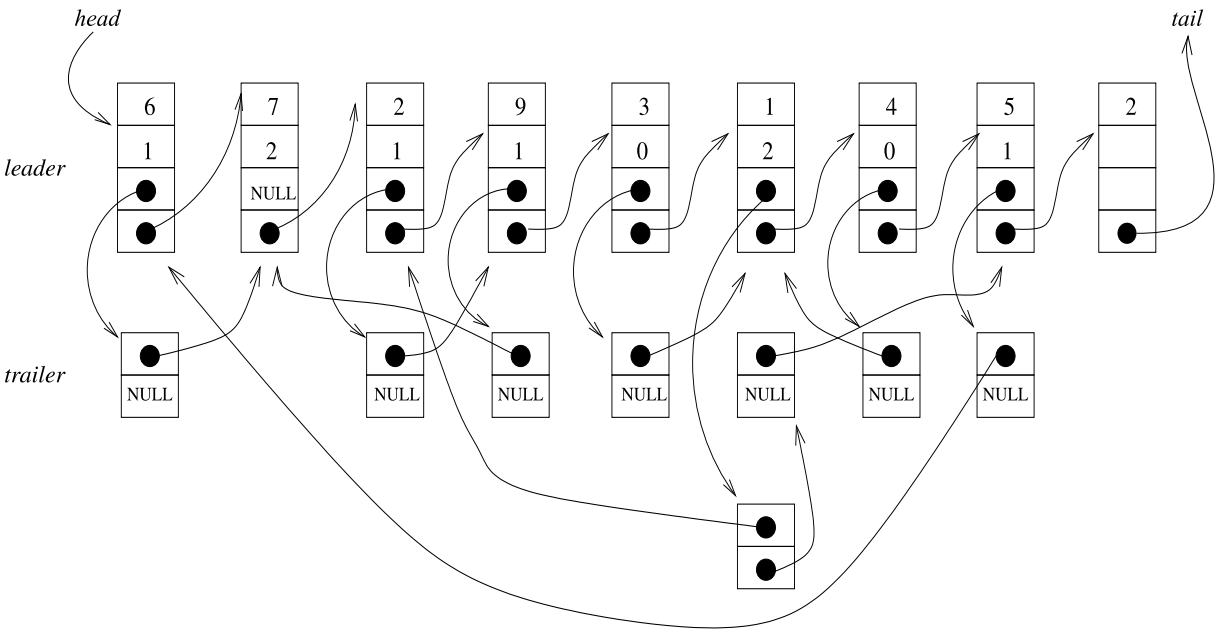
- Einlesen der Paare in Datenstruktur
- Suche Leader ohne Vorgänger
- Ausgabe der topologisch sortierten Folge von Schlüsseln

Einlesen der Paare in Datenstruktur:

```
 1 LPtr find(int w)
 2 { LPtr h;
 3
 4   h = head;  /* head ist erstes Element der Eingabeliste */
 5   tail->key = w;
 6   while (h->key != w) h = h->next; /* Suche Element und setze
 7                                       Zeiger h auf dieses Element */
 8   if (h == tail)
 9   { tail = (LPtr) malloc(sizeof(leader));
10     n = n+1;
11     h->count = 0;
12     h->trail = NULL;
13     h->next = tail;
14   }
15   return h;
16 }
17
18 int main()
19 { int Done = 1;
20     /* Eingabephase, Ende der Eingabe mit "999" als Anfangsknoten */
21   head = (LPtr) malloc(sizeof(leader));
22   tail = head;
```

```
23    n = 0;
24    printf("Anfangsknoten: ");
25    scanf("%d", &x);
26
27    while (Done)
28    { printf("Endknoten: ");
29      scanf("%d", &y);
30      p = find(x);
31      q = find(y);
32      t = (TPtr) malloc(sizeof(trailer));
33      t->id = q;
34      t->next = p->trail;
35      p->trail = t;
36      q->count = q->count + 1;
37      printf("\nAnfangsknoten: ");
38      scanf("%d", &x);
39      if (x == 999) Done = 0;
40    }
41  }
```

Nach dem Einlesen sieht die Datenstruktur wie folgt aus:



Suche Leader ohne Vorgänger

Alle Elemente ohne Vorgänger werden nacheinander an den Anfang einer neuen Liste gestellt. Diese Liste enthält zunächst nur Quellen, d. h. leader–Elemente mit `count=0` (also Elemente ohne Vorgänger).

```
1   p = head;
2   head = NULL;
3   while (p != tail)
4   { q = p;
5     p = q->next;
6     if (q->count == 0)
7     { q->next = head;
8       head = q;
9     }
10  }
```

Ausgabe der topologisch sortierten Folge von Schlüsseln:

Es wird (erreichbar durch `q`) eine Liste mit Elementen dynamisch aufgebaut, deren Vorgängerzähler den Wert 0 haben.

```
q = head;
while (q != NULL)
    /* drucke dieses Element, loesche es dann */
{ printf(" %d ", q->key);
  n = n-1; t = q->trail; q = q->next;
    /* verringere den Vorgaengerzaehler jedes Elementes in der Liste t
       der trailer; wird ein Zaehler 0, so wird dieses Element an den
       Anfang der Liste q der Leader genommen. */
}
```

Durchsuchen der Liste der trailer:

```
while (t != NULL)
{ p = t->id;
  p->count = p->count-1;
  if (p->count == 0)
  { p->next = q;
    q = p;
  }
  t = t->next;
}
```

Das Gesamtprogramm sieht dann wie folgt aus:

```
/* TopSort */
#include <stdio.h>
#include <stdlib.h>

typedef struct leader *LPtr;
typedef struct trailer *TPtr;

typedef struct leader
               { int key;
                 int count;  /* Anzahl der Vorgaenger */
                 TPtr trail; /* Liste mit Zeigern zu Nachfolgeelementen
                                bezuegl. der Halbordnung */
                 LPtr next;  /* Zeiger zu Element, welches bzgl. seines
                                Erscheinens in der Eingabe das naechste ist */
               } leader;

typedef struct trailer
               { LPtr id;
                 TPtr next;
               } trailer;

LPtr p, q, head, tail;
TPtr t;
int x, y, n;

LPtr find(int w)
{ LPtr h;

  h = head;
  tail->key = w;
  while (h->key != w) h = h->next;
  if (h == tail)
  { tail = (LPtr) malloc(sizeof(leader));
    n = n+1;
```

```
35      h->count = 0;
36      h->trail = NULL;
37      h->next = tail;
38    }
39    return h;
40  }
41
42  int main()
43  { int Done = 1;
44      /* Eingabephase, Ende der Eingabe mit "999" als Anfangsknoten */
45    head = (LPtr) malloc(sizeof(leader));
46    tail = head;
47    n = 0;
48    printf("Anfangsknoten: ");
49    scanf("%d", &x);
50
51    while (Done)
52    { printf("Endknoten: ");
53      scanf("%d", &y);
54      p = find(x);
55      q = find(y);
56      t = (TPtr) malloc(sizeof(trailer));
57      t->id = q;
58      t->next = p->trail;
59      p->trail = t;
60      q->count = q->count+1;
61      printf("\nAnfangsknoten: ");
62      scanf("%d", &x);
63      if (x == 999) Done = 0;
64    }
65
66    /* Suche nach Leader ohne Vorgänger */
67    p = head;
68    head = NULL;
69    while (p != tail)
70    { q = p;
71      p = q->next;
72      if (q->count == 0)
73      { q->next = head;
74        head = q;
75      }
76    }
77
78    /* Ausgabephase */
79    q = head;
80    printf("Eingebettete lineare Ordnung:\n");
81    while (q != NULL) /* drucke jeweils ein Element ohne Vorgaenger */
82    { printf(" %d ", q->key);
83      n = n-1;
84      t = q->trail;
85      q = q->next;
86      while (t != NULL)
87      { p = t->id;
88        p->count = p->count-1;
89        if (p->count == 0)
90        { p->next = q;
91          q = p;
92        }
93        t = t->next;
94      }
95    }
96
```

```
97    if (n != 0)
98      printf("\nDiese Liste beschreibt keine partielle Ordnung");
99    printf("\n\n");
100 }
```

12.3 Breiten- und Tiefensuche in Graphen

Im folgenden Abschnitt werden wir uns mit dem Problem der *Graphensuche* befassen, bei dem alle von einem gewissen Startknoten s aus erreichbaren Knoten eines gerichteten Graphen G systematisch durchsucht werden sollen. Dieses Problem besitzt grundlegende Bedeutung, weil sich zahlreiche andere Programmieraufgaben darauf reduzieren lassen: soll zum Beispiel für ein Spiel eine zum Gewinn führende Zugfolge bestimmt werden, so lässt sich das durch das Durchsuchen eines Graphen realisieren, dessen Knoten Spielkonstellationen und dessen Kanten gültige Spielzüge sind. Für ein komplexes Spiel wie z. B. Schach sollte allerdings der entsprechende Graph nicht vollständig im Speicher gehalten, sondern bedarfsgesteuert erstellt werden. Von solchen Implementierungsdetails werden wir aber im Verlauf dieses Abschnitts abstrahieren. Ebenso sehen wir davon ab, nach Knoten mit einer gewissen Eigenschaft zu suchen: es sollen alle vom Startknoten s aus erreichbaren Knoten gefunden werden. Wird zu einem bestimmten Zeitpunkt im Ablauf der Suche von einem Knoten festgestellt, dass er von s aus erreichbar ist, so gilt dieser Knoten ab jenem Zeitpunkt als *besucht*. Die besuchten Knoten, sowie die Ordnung, in der sie besucht wurden, sollen vom Suchverfahren in einem *s-Spannbaum* repräsentiert werden. Ein s-Spannbaum von G ist ein Teilgraph von G, der genau die von s aus erreichbaren Knoten enthält und der ein Baum ist.

Wir können zwei prinzipielle Strategien der Graphensuche unterscheiden: *Tiefen-* und *Breitensuche* (bzw. *Depth-First Search (DFS)* und *Breadth-First Search (BFS)*). Beide Suchverfahren durchsuchen den vorliegenden Graphen G schrittweise, indem sie eine Datenstruktur S verwalten, in der Kanten zu bereits entdeckten Knoten abgespeichert sind. Ein Knoten heißt dann *entdeckt,* wenn er im Verlauf der Suche für ein späteres Besuchen vorgemerkt, aber noch nicht besucht wurde. Am Anfang der Suche ist nur der Startknoten bereits entdeckt. In einem Iterationsschritt wird jeweils ein entdeckter Knoten u von G besucht. Daraufhin wird jeder seiner unbesuchten Nachfolgerknoten v als entdeckt markiert, indem jeweils die Kante (u, v) in S eingefügt wird.

Die Tiefensuche in einem Graphen G untersucht G tiefenorientiert, d. h. wenn es von einem gerade besuchten Knoten u eine ausgehende Kante gibt, dann wird zunächst der Zielknoten dieser Kante besucht (und diese Strategie dann auf diesen Knoten angewendet) bevor die Nachbarknoten von u untersucht werden. (Wenn G ein geordneter Baum wäre, so würde die Tiefensuche genau einem depth-first left-to-right Baumdurchlauf entsprechen.) Einen Spannbaum, welcher durch eine Tiefensuche entstanden ist, nennen wir auch einen *depth-first tree.*

Die Breitensuche hingegen durchsucht G breitenorientiert: ausgehend von einem gerade besuchten Knoten u werden erst alle direkten Nachfolgerknoten von u untersucht, um dann deren Nachfolgerknoten zu untersuchen, dann die Nachfolger jener, usw. (Wäre G wieder ein geordneter Baum, so würde diese Strategie den Baum sukzessive Ebene für Ebene durchlaufen.) Einen Spannbaum, welcher im Laufe einer Breitensuche entstanden ist, nennen wir einen *breadth-first tree.*

Beide Strategien garantieren für zirkuläre Graphen, dass man Knoten nicht mehrfach, sondern höchstens einmal besucht. Darüber hinaus sind beide Strategien nichtdeterministisch: für das Abarbeiten der Nachfolger eines Knotens ist keine feste Reihenfolge vorgegeben, und je nachdem welche Reihenfolge in einem konkreten Suchvorgang gewählt wurde, können sich für den gleichen Eingabegraphen von ihrer Struktur her vollkommen verschiedene Spannbäume ergeben.

Beispiel 12.1. Veranschaulichen wir uns Tiefen- und Breitensuche am Beispielgraphen $G = (V, E)$ aus Abbildung 12.2 mit dem Startknoten 1. Betrachten wir zunächst die Tiefensuche in G, deren Ablauf in Tabelle 12.1a angegeben ist (dabei steht It. für Iterationsschritt). In der zweiten Spalte ist für jeden Iterationsschritt der aktuelle Wert der Datenstruktur S, eine geordnete Sequenz von Kanten $(u, v) \in E$, gegeben. In einer Kante (u, v) ist v der entdeckte Knoten, während der Knoten u ein direkter Vorgängerknoten von v ist, von dem aus v entdeckt wurde. Das Abspeichern von u dient der Konstruktion des Spannbaums.

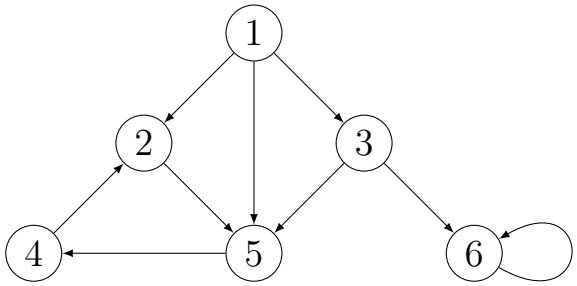
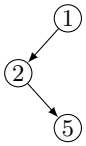
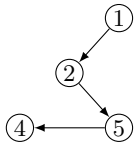
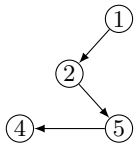
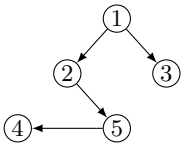
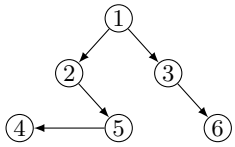
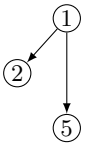
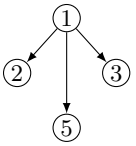
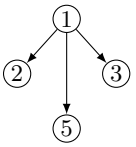
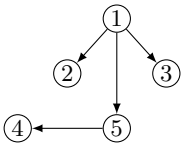
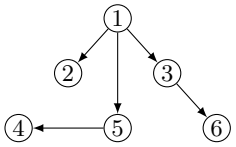


Abbildung 12.2: Beispielgraph G zur Tiefen- und Breitensuche

It.	S (entdeckte Knoten)	t (depth-first tree)
0.	$[(0,1)]$	
1.	$[(1,2),(1,5),(1,3)]$	
2.	$[(2,5),(1,5),(1,3)]$	
3.	$[(5,4),(1,5),(1,3)]$	
4.	$[(1,5),(1,3)]$	
5.	$[(1,3)]$	
6.	$[(3,6)]$	
7.	$[]$	

(a)

It.	S (entdeckte Knoten)	t (breadth-first tree)
0.	$[(0,1)]$	
1.	$[(1,2),(1,5),(1,3)]$	
2.	$[(1,5),(1,3),(2,5)]$	
3.	$[(1,3),(2,5),(5,4)]$	
4.	$[(2,5),(5,4),(3,6)]$	
5.	$[(5,4),(3,6)]$	
6.	$[(3,6)]$	
7.	$[]$	

(b)

Tabelle 12.1: Ablauf einer Tiefen- bzw. Breitensuche auf dem Graphen G aus Abbildung 12.2. In den zweiten Spalten ist die zweite Komponente einer Kante (i,j) der entdeckte Knoten.

In der dritten Spalte ist jeweils der bis zum aktuellen Iterationsschritt aufgebaute Spannbaum t dargestellt. In jedem Schritt der Suche wird das am Anfang der Sequenz (d.h. links) stehende Tupel (u, v) aus S entfernt, und v, falls es nicht bereits besucht wurde, als neuer Kindknoten von u in t eingefügt. Daraufhin werden alle Kanten $(v, w) \in E$, welche zu einem noch nicht besuchten Knoten w führen, an den *Anfang* der Datenstruktur S eingefügt. Diese Knoten w werden also im weiteren Verlauf der Suche *vor* den anderen Knoten in der Datenstruktur verarbeitet werden, und dies entspricht dem oben beschriebenen Vorgehen für die Tiefensuche.

Am Beginn der Suche wird das Tupel $(0, 1)$ in die Datenstruktur eingetragen. Der Wert 0 dient dabei als künstlicher Vorgänger des Knotens 1 und ist für den weiteren Ablauf des Algorithmus irrelevant. Beim ersten Ausführen der Schleife (erste Iteration) wird das Tupel $(0, 1)$ entfernt, der Knoten 1 besucht und in den Spannbaum eingetragen und die drei Nachfolgerknoten 3, 5 und 2 in dieser Reihenfolge entdeckt. In der zweiten Iteration fährt die Tiefensuche fort mit dem Besuchen von Knoten 2 und trägt die Kante $(1, 2)$ im Spannbaum ein. In der fünften Iteration wird die Kante $(1, 5)$ aus der Datenstruktur entfernt, der Spannbaum ändert sich jedoch nicht, da der Knoten 5 bereits im dritten Iterationsschritt besucht wurde. Im sechsten Schritt letztendlich wird der Nachfolger 3 des Knotens 1 bearbeitet, da alle vom Knoten 2 aus erreichbaren Knoten in den vorhergehenden Schritten besucht worden sind.

Der Ablauf einer Breitensuche wurde in Tabelle 12.1b protokolliert. Auch hier enthält die zweite Spalte eine Datenstruktur mit im weiteren Verlauf zu verarbeitenden Knoten zusammen mit ihren direkten Vorgängern, und die dritte Spalte den bisher aufgebauten Spannbaum. In einem Iterationsschritt wird nach wie vor das am weitesten links stehende Tupel (u, v) aus der Datenstruktur entfernt und v unter u im Spannbaum eingefügt. Jedoch werden für die Breitensuche die Kanten mit den noch nicht besuchten Nachfolgern von v nicht am Anfang, sondern am *Ende* der Datenstruktur (d.h. rechts) eingefügt. Es werden durch die Breitensuche also in früheren Iterationsschritten hinzugefügte Knoten bevorzugt behandelt.

Dies macht sich in den Iterationsschritten 2-4 bemerkbar, in denen erst alle Nachfolgerknoten des Startknotens besucht werden. Erst im sechsten bzw. siebten Schritt werden die über jene entdeckten Knoten 4 bzw. 6 besucht. □

Wie wir uns an diesem Beispiel veranschaulichen konnten, besteht der maßgebliche Unterschied zwischen Tiefen- und Breitensuche also allein darin, an welcher Stelle die neu entdeckten Kanten in die Datenstruktur eingefügt werden. Diesen Unterschied kapseln wir in zwei verschiedenen Datenstrukturen.

Definition. Gegeben sei eine Menge A. Der Datentyp *Pushdown*(A) ist die Menge A^* aller endlichen Sequenzen von Elementen aus A, zusammen mit den drei (partiellen) Operationen

$$\begin{aligned}
\mathit{push}\colon\quad & A^* \times A \to A^* & & \\
& (a_1 \cdots a_n, a) \mapsto a a_1 \cdots a_n & & \text{(für alle } n \in \mathbb{N}\text{)} \\
\mathit{pop}\colon\quad & A^* \to A^* & & \\
& a_1 a_2 \cdots a_n \mapsto a_2 \cdots a_n & & \text{(falls } n \geq 1\text{)} \\
\mathit{top}\colon\quad & A^* \to A & & \\
& a_1 \cdots a_n \mapsto a_1 & & \text{(falls } n \geq 1\text{)}
\end{aligned}$$

und dem ausgezeichneten Element $\mathit{empty} = \varepsilon$ aus A^*. □

Definition. Gegeben sei eine Menge A. Der Datentyp *Queue*(A) ist die Menge A^* mit den drei (partiellen) Operationen

$$\begin{aligned}
\mathit{enqueue}\colon\quad & A^* \times A \to A^* & & \\
& (a_1 \cdots a_n, a) \mapsto a_1 \cdots a_n a & & \text{(für alle } n \in \mathbb{N}\text{)} \\
\mathit{dequeue}\colon\quad & A^* \to A^* & & \\
& a_1 a_2 \cdots a_n \mapsto a_2 \cdots a_n & & \text{(falls } n \geq 1\text{)} \\
\mathit{head}\colon\quad & A^* \to A & & \\
& a_1 \cdots a_n \mapsto a_1 & & \text{(falls } n \geq 1\text{)}
\end{aligned}$$

und dem ausgezeichneten Element $\mathit{nil} = \varepsilon$ aus A^*. □

	STORAGE	EMPTY	INSERT	REMOVE	READ
Tiefensuche	*Pushdown*(edge)	*empty*	*push*	*pop*	*top*
Breitensuche	*Queue*(edge)	*nil*	*enqueue*	*dequeue*	*head*

Tabelle 12.2: Instanziierung von STORAGE für Tiefen- und Breitensuche

Beide Datentypen umfassen also endliche Sequenzen über einer Grundmenge A, sowie die Möglichkeit zum Erstellen einer leeren Instanz des Datentyps, zum Einfügen eines Elements (bei Pushdowns am Anfang, bei Queues am Ende des Datentyps), zum Entfernen und zum Lesen des Elements am Anfang der Sequenz. Mithilfe dieser Datentypen können wir nun einen allgemeinen Graphensuche-Algorithmus entwerfen und ihn zu Tiefen- und Breitensuche instanziieren (vgl. Abbildung 12.3, 12.3).

Auch für die Implementierung nehmen wir weiter an, dass die Knotenmenge V von der Form $\{1, 2, \ldots, n\}$ für ein $n \in \mathbb{N}$ ist. Die Beispielimplementierung beruht auf dem Konzept der Modularisierung von C-Programmen (vgl. Kap. 7). Die Header-Dateien in Abbildung 12.3 stellen notwendige Hilfsfunktionen und -datentypen zur Verfügung, u. a. zum Abspeichern von Kanten und zum Umgang mit Graphen bzw. Bäumen (`graph.h`), sowie die oben erwähnten Operationen auf Pushdowns und Queues (`pushdown.h` bzw. `queue.h`). Die zugehörigen Definitionsmodule sind nicht abgebildet.

Die entsprechenden Funktionen in `pushdown.h` und `queue.h` sind mit jeweils gleich lautenden Bezeichnern eines abstrakten Datentyps `STORAGE` deklariert, vgl. dazu Tabelle 12.2. Die Implementierung in `graphsearch.c` (Abbildung 12.4) verwendet diese Bezeichner, somit kann die Graphensuche unabhängig von der zugrunde liegenden konkreten Datenstruktur Pushdown oder Queue spezifiziert werden. In `dfs.c` wird das Modul für Pushdowns vor `graphsearch.c` eingebunden. Der `STORAGE`-Datentyp, der im allgemeinen Algorithmus verwendet wird, wird sich daher wie ein Pushdown verhalten. So wird der Algorithmus zu einer Tiefensuche instanziiert. Analog dazu umgesetzt wird die Instanziierung zu einer Breitensuche in `bfs.c`.

Betrachten wir die Funktion `graphsearch` etwas genauer. Ab Z. 3 von `graphsearch.c` werden zunächst der Spannbaum `t` sowie die Datenstruktur `S` initialisiert und der Startknoten der Suche in `S` eingefügt. Der Iterationsschritt der Suche ist in der äußeren `while`-Schleife (Z. 8-19) implementiert. Solange die Datenstruktur `S` nicht leer ist, wird in jedem Durchlauf ein Tupel aus ihr entnommen, der enthaltene Knoten an der entsprechenden Stelle in den Spannbaum `t` eingefügt und `S` mit all den Nachfolgerknoten angereichert, welche noch nicht in `t` enthalten sind, d. h. noch nicht besucht wurden. Sobald `S` leer ist, sind alle erreichbaren Knoten besucht und der Spannbaum `t` wird zurückgegeben (Z. 20)

Die Laufzeitkomplexität des Algorithmus liegt in $O(|E|)$, wenn wir das uniforme Kostenmaß für Operationen auf dem Graphen und dem Datentyp der entdeckten Knoten annehmen. Die innere `for`-Schleife in Z. 13-17 wird für einen unbesuchten Knoten $v \in V$ genau $|\{v' \mid (v, v') \in E\}|$ mal durchlaufen. Dann gilt v aber als besucht, und die `for`-Schleife wird für den gleichen Knoten v kein weiteres Mal aufgerufen. Die Anzahl der Durchläufe durch die äußere `while`-Schleife (Z. 8-19) liegt in $O(|E|)$, denn immerhin werden der Datenstruktur nur Kanten des Graphen hinzugefügt. Damit ist die Gesamtzahl der Schleifendurchläufe beschränkt durch $|E| + \sum_{v \in V} |\{v' \mid (v, v') \in E\}| = 2 \cdot |E|$, also ist die Laufzeit des Algorithmus in $O(|E|)$.

12.4 Kürzeste Wege

Ein *Distanzgraph* $G = (V, E, c)$ besteht aus einem gerichteten Graphen (V, E) und einer Abbildung $c\colon E \to \mathbb{R}_{\geq 0}$ mit $\mathbb{R}_{\geq 0} = \{r \in \mathbb{R} \mid r \geq 0\}$. Wir nehmen wieder an, dass $V = \{1, \ldots, n\}$ ist. Die Adjazenzmatrix A_G von G ist nun die $n \times n$-Matrix über $\mathbb{R}^{\infty}_{\geq 0} = \mathbb{R}_{\geq 0} \cup \{\infty\}$ mit

$$A_G(u, v) = \begin{cases} c(u, v) & \text{wenn } (u, v) \in E \\ \infty & \text{sonst} \end{cases}$$

Sei nun $s \in V$ ein beliebiger, im folgenden fester Startknoten (Quelle). Wir wollen algorithmisch berechnen, was die kürzeste Entfernung von s nach v ist, für jeden beliebigen Knoten $v \in V$. Dazu diskutieren wir hier den Dijkstra-Algorithmus von Seite 566 aus [OW02].

```c
/* Datentyp zum Abspeichern von Kanten, d.h. Tupeln von Knoten */
typedef struct edge {
  int fst;
  int snd;
} edge;

/* Datentyp zum Abspeichern von gerichteten Graphen */
typedef struct graph { ... } graph;

/* add_edge(&G, e) fuegt dem Graphen G die Kante e, und, falls nicht enthalten, die
 * Knoten e.fst und e.snd hinzu. Ist e.fst == 0, wird ein neuer Graph erzeugt, der
 * als einzigen Knoten e.snd enthaelt. */
void add_edge(graph* G, edge e);

/* empty_graph() erstellt einen neuen, leeren Graphen */
graph empty_graph();

/* contains(G, v) gibt 1 zurueck, falls der Graph G den Knoten v enthaelt, sonst 0 */
int contains(graph G, int v);
```

graph.h

```c
typedef struct Pushdown { ... } STORAGE;
const STORAGE EMPTY = ... ;     /* empty: leerer Pushdown */

/* push: INSERT(S, e) fügt Kante e als oberstes Element in S ein */
STORAGE INSERT(STORAGE S, edge e);

/* pop: REMOVE(S) entfernt oberstes Element eines nichtleeren Pushdowns S */
STORAGE REMOVE(STORAGE S);

/* top: READ(S) liest oberstes Element eines nichtleeren Pushdowns S */
edge READ(STORAGE S);
```

pushdown.h

```c
typedef struct Queue { ... } STORAGE;
const STORAGE EMPTY = ...;      /* nil: leere Queue */

/* enqueue: INSERT(S, e) fügt Kante e als letztes Element in S ein */
STORAGE INSERT(STORAGE S, edge e);

/* dequeue: REMOVE(S) entfernt erstes Element einer nichtleeren Queue S */
STORAGE REMOVE(STORAGE S);

/* head: READ(S) liest erstes Element einer nichtleeren Queue S */
edge READ(STORAGE S);
```

queue.h

Abbildung 12.3: Headerdateien für verallgemeinerte Graphensuche

```
1  /* Funktion zur generalisierten Graphensuche */
2  graph graphsearch(graph G, int s)
3  { graph t = empty_graph();            /* Aufzubauender Suchbaum */
4    STORAGE S = EMPTY;                  /* Datentyp von Kanten zu entdeckten Knoten */
5    edge e = {0, s};                    /* Kein Vorgaengerknoten: e.fst == 0 */
6    S = INSERT(S, e);
7
8    while (S != EMPTY) {
9      e = READ(S);
10     S = REMOVE(S);
11     if (!contains(t, e.snd)) {        /* e.snd noch nicht besucht */
12       add_edge(&t, e);                /* e.snd ist besucht */
13       for (all successors v of e.snd in G)
14         if (!contains(t, v)) {        /* Nur unbesuchte Knoten hinzufuegen */
15           edge f = {e.snd, v};
16           S = INSERT(S, f);           /* v ist entdeckt */
17         }
18     }
19   }
20   return t;
21 }
```

graphsearch.c

```
1  /* Implementierung der Tiefensuche */
2  #include "graph.h"
3  #include "pushdown.h"     /* Instanziierung von STORAGE mit Pushdown */
4  #include "graphsearch.c"
5
6  graph dfs(graph G, int s) { return graphsearch(G, s); }
```

dfs.c

```
1  /* Implementierung der Breitensuche */
2  #include "graph.h"
3  #include "queue.h"        /* Instanziierung von STORAGE mit Queue */
4  #include "graphsearch.c"
5
6  graph bfs(graph G, int s) { return graphsearch(G, s); }
```

bfs.c

Abbildung 12.4: Implementierung der verallgemeinerten Graphensuche

Die Idee des Algorithmus ist folgende: Er verlängert einen bereits bekannten kürzesten Weg p von s nach v um ein Kante (v, v') zu einem kürzesten Weg von s nach v'. Die Möglichkeit für diese inkrementelle Konstruktion kürzester Wege liefert die folgende Optimalitätseigenschaft: für jeden kürzesten Weg $p = (v_0, v_1, \ldots, v_k)$ von v_0 nach v_k ist jeder Teilweg $(v_i, \ldots, v_j)$ mit $1 \leq i < j \leq k$ auch ein kürzester Weg von v_i nach v_j (das kann man leicht durch einen Widerspruchsbeweis zeigen).

Beispiel 12.2. Betrachten wir nun den Distanzgraphen in Abbildung 12.5, wobei wir den Knoten 1 als Quelle wählen. Es gilt als plausible Anfangsfestlegung, dass der kürzeste Weg von 1 nach 1 die Länge 0 hat. Nun sind vom Knoten 1 die Knoten 2, 7 und 6 direkt (d. h. über eine Kante) mit einer Entfernung von 2, 12 bzw. 4 erreichbar; diese nennen wir Randknoten. Diese Information wollen wir als Menge von Tripeln notieren, wobei ein Tripel aus der Knotennummer, der Entfernung von der Quelle und seinem Vorgängerknoten auf dem bisherigen kürzesten Weg besteht. Also: $\{(2, 2, 1), (7, 12, 1), (6, 4, 1)\}$.

Können wir jetzt schon für einen dieser Randknoten v sagen, wie lang der kürzeste Weg von 1 nach v ist? Gehen wir dafür die Randknoten der Reihe nach durch:

- $v = 2$: Sicherlich gibt es für den Randknoten $v = 2$ keinen kürzeren Weg von 1 nach 2 als den Weg $(1, 2)$, der nur aus einer Kante besteht. Denn angenommen, dass z. B. der kürzeste Weg von 1 nach 2 über den Knoten 7 führen würde und dann irgendwie nach 2, dann würde dieser Weg mindestens die Länge der Kante $(1, 7)$, d. h. 12 haben. Dann ist es aber nicht der kürzeste Weg, was ein Widerspruch zur Annahme ist. Für Knoten 6 als Zwischenknoten argumentiert man genauso.
- $v = 7$: Für diesen Randknoten können wir im Augenblick noch keine Aussage darüber machen, wie der kürzeste Weg verläuft: es kann der direkte Weg $(1, 7)$ sein, es könnte aber auch ein Weg sein, der mit der Kante $(1, 2)$ beginnt (was sich auch gleich so herausstellen wird).
- $v = 6$: Hierfür gilt dasselbe wie für den Randknoten 7.

Gut, damit können wir also Knoten 2 in die Menge derjenigen Knoten aufnehmen, für die wir schon einen kürzesten Weg gefunden haben. Damit verändert sich aber die Menge der Randknoten, denn:

1. Jetzt ist auch der Knoten 3 erreichbar, nämlich über den Weg $(1, 2, 3)$, was wir durch die Information $(3, 12, 2)$ codieren (die 12 ergibt sich aus der Summe der Länge des kürzesten Weges von 1 nach 2 und der Länge der Kante $(2, 3)$), und
2. jetzt kann der Randknoten 7 auf kürzerem Weg als direkt über die Kante $(1, 7)$ erreicht werden; deshalb tauschen wir die Information $(7, 12, 1)$ durch $(7, 11, 2)$ aus. □

In dieser Weise fährt der Algorithmus fort. Der Algorithmus teilt also die Knoten von G in drei disjunkte Teilmengen auf:

1. die Menge der gewählten Knoten (d. h. Knoten, für die schon der kürzeste Weg bekannt ist),
2. die Menge R der Randknoten (d. h. direkte Nachbarknoten von gewählten Knoten) und
3. die noch unerreichten Knoten.

Zu Beginn ist nur s gewählt. Der Algorithmus verlängert kürzeste Wege (*sp*: shortest path), wobei die beiden folgenden Eigenschaften gelten:

1. Für beliebige Knoten v, v' und jeden kürzesten Weg $sp(s, v)$ von s nach v gilt: $c(sp(s, v))+c((v, v')) \geq c(sp(s, v'))$ und
2. Für jeden Knoten v' gibt es wenigstens einen Knoten v und einen kürzesten Weg $sp(s, v)$ von s nach v, sodass: $c(sp(s, v)) + c((v, v')) = c(sp(s, v'))$.

Beispiel 12.2 (Fortsetzung)**.** Verfahren wir weiter nach diesem Verfahren, so ergibt sich das folgende Ablaufprotokoll:

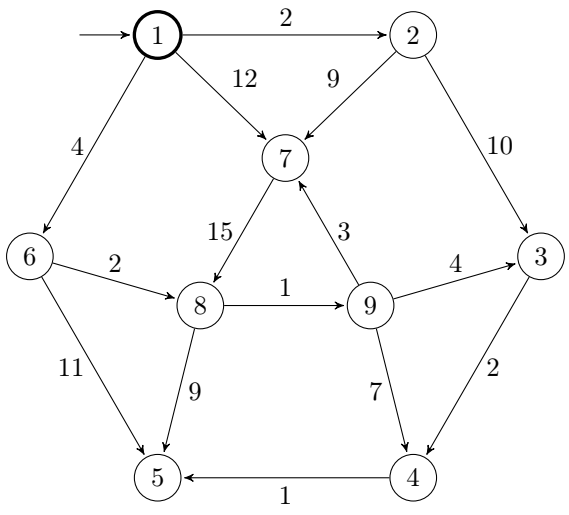


Abbildung 12.5: Beispiel für einen Distanzgraphen.

gewählt	Menge der Randknoten
$(1, 0, -)$	$\{(2, 2, 1), (6, 4, 1), (7, 12, 1)\}$
$(2, 2, 1)$	$\{(3, 12, 2), (6, 4, 1), (7, 11, 2)\}$
$(6, 4, 1)$	$\{(3, 12, 2), (5, 15, 6), (7, 11, 2), (8, 6, 6)\}$
$(8, 6, 6)$	$\{(3, 12, 2), (5, 15, 6), (7, 11, 2), (9, 7, 8)\}$
$(9, 7, 8)$	$\{(3, 11, 9), (4, 14, 9), (5, 15, 6), (7, 10, 9)\}$
$(7, 10, 9)$	$\{(3, 11, 9), (4, 14, 9), (5, 15, 6)\}$
$(3, 11, 9)$	$\{(4, 13, 3), (5, 15, 6)\}$
$(4, 13, 3)$	$\{(5, 14, 4)\}$
$(5, 14, 4)$	$\varnothing$

Aus dem Ablaufprotokoll lassen sich jetzt die kürzesten Wege vom Knoten 1 zu jedem beliebigen anderen Knoten ablesen:

zum Knoten v	kürzester Weg	Länge des Weges
2	(1,2)	2
3	$(1, 6, 8, 9, 3)$	11
4	$(1, 6, 8, 9, 3, 4)$	13
5	$(1, 6, 8, 9, 3, 4, 5)$	14
6	$(1, 6)$	4
7	$(1, 6, 8, 9, 7)$	10
8	$(1, 6, 8)$	6
9	$(1, 6, 8, 9)$	7

□

Die Komplexität des Dijkstra–Algorithmus ist $O(|V|^2)$. Bei Verwendung eines Fibonacci-Heaps zur Implementierung der Prioritätswarteschlange R erreicht man sogar $O(|V| \log |V| + |E|)$.

12.5 Das algebraische Pfadproblem

In diesem Abschnitt diskutieren wir eine Alternative zum Dijkstra-Algorithmus zur Berechnung der kürzesten Wege. Wir werden später sehen, dass sich diese Alternative sehr leicht auf andere Problemstellungen verallgemeinern lässt.

Also: gegeben sei wiederum ein Distanzgraph $G = (V, E, c)$ mit $c\colon E \to \mathbb{R}_{\geq 0}$. Wir nehmen wieder an, dass $V = \{1, \ldots, n\}$ für ein n ist. Außerdem bezeichnen wir die Menge aller Wege von u nach v (für beliebige $u, v \in V$) durch $P_{u,v}$; insbesondere ist (u) ein Weg von u nach u der Länge 0. Schließlich nehmen wir an, dass G keine Schlingen enthält, d.h. $(u, u) \notin E$ für jedes $u \in V$.

Nun geben wir keinen festen Startknoten wie beim Dijkstra-Algorithmus vor, sondern wollen für ein beliebiges Paar $(u, v) \in V \times V$ den kürzesten Weg zwischen u und v bestimmen. Das heißt, dass wir das folgende Problem D_G lösen wollen:

D_G: Für beliebige $u, v \in V$: Wie lang ist der kürzeste Weg p von u nach v?

All diese Werte fassen wir in der sogenannten *kürzesten-Wege-Matrix* zusammen, die wir ebenfalls D_G nennen; das ist die $n \times n$-Matrix D_G über $\mathbb{R}^{\infty}_{\geq 0}$ mit

$$D_G(u, v) = \begin{cases} \min\{c(p) \mid p \in P_{u,v}\} & \text{wenn } P_{u,v} \neq \varnothing \\ \infty & \text{sonst;} \end{cases}$$

für jeden Weg $p = (v_0, \ldots, v_r)$ (mit $r \geq 0$ und $v_0, \ldots, v_r \in V$) ist dessen Länge gleich

$$c(p) = \sum_{l=0}^{r-1} c(v_l, v_{l+1});$$

insbesondere gilt $c((u)) = 0$ für jedes $u \in V$. Also gilt: $D_G(u, u) = 0$ für jedes $u \in V$.

Der nachfolgende Algorithmus, der die kürzeste-Wege-Matrix D_G berechnet, löst zunächst ein etwas schwierigeres Problem $D_G^{(k)}$, welches einen Parameter k hat, der die Werte $0, 1, \ldots, n$ annehmen kann:

$D_G^{(k)}$: Für beliebige $u, v \in V$: Wie lang ist der kürzeste Weg p von u nach v, so dass $p \in P_{u,v}^{(k)}$?

Algorithmus 7 Dijkstra-Algorithmus

Eingabe: gerichteter Distanzgraph $G = (V, E, c)$ und ein Knoten $s \in V$

Ausgabe: für jeden Knoten $v \in V$ ist der kürzeste Weg von s nach v der Weg: $(s, ..., p(p(v)), p(v), v)$

Verfahren:

```
1  Set R;              /* Menge der Randknoten                          */
2  Node u, v;          /* Knoten aus V                                  */
3  PredVector p;       /* ordnet jedem v in V einen Vorgaengerknoten zu */
4  LengthVector d;     /* ordnet jedem v in V einen Abstand (natuerliche
5                         Zahl oder unendlich) zur Quelle zu            */
6
7  /* Initialisierung */
8  for (alle v in V)
9  { d(v) = unendlich;
10   p(v) = undefiniert;
11 }
12 d(s) = 0;
13 p(s) = s;
14 U = V;
15 R = {s};
16
17 while (R nicht leer)
18 { waehle u in R, so dass d(u) = min{ d(v) | v in U }
19   entferne u aus U und aus R;
20
21   for (jedes v in U mit (u,v) in E)
22     if (d(u)+c(u,v) < d(v))
23     { d(v) = d(u)+c(u,v);
24       p(v) = u;
25       fuege v zu R hinzu;
26     }
27 }
```

Dabei ist $P_{u,v}^{(k)}$ die Menge aller der Wege in $P_{u,v}$, deren innere Knoten in der Menge $\{l \mid 1 \leq l \leq k\}$ liegen. (Beachte: für jedes $u \in V$ ist der Weg (u) in $P_{u,u}^{(k)}$ enthalten und zwar für jedes k mit $0 \leq k \leq n$. Außerdem beachte, dass $\{l \mid 1 \leq l \leq 0\} = \varnothing$.) Diese Werte wollen wir in der $n \times n$-Matrix $D_G^{(k)}$ über $\mathbb{R}_{\geq 0}^{\infty}$ zusammenfassen:

$$D_G^{(k)}(u,v) = \begin{cases} \min\{c(p) \mid p \in P_{u,v}^{(k)}\}, & \text{falls } P_{u,v}^{(k)} \neq \varnothing \\ \infty, & \text{sonst.} \end{cases}$$

Es ist klar, dass die Matrizen D_G und $D_G^{(n)}$ gleich sind, denn wenn an einen Weg p von u nach v die Bedingung gestellt wird, dass jeder seiner inneren Knoten aus $\{l \mid 1 \leq l \leq n\} = V$ ist, dann ist das gar keine zusätzliche Bedingung, m.a.W.: $P_{u,v} = P_{u,v}^{(n)}$.

Der Algorithmus berechnet zuerst $D_G^{(0)}$. Da $\{l \mid 1 \leq l \leq 0\} = \varnothing$, enthält $P_{u,v}^{(0)}$ nur Kanten aus G und, wenn $u = v$ ist, auch den Weg (u). D. h. dass $D_G^{(0)}$ explizit so aussieht:

$$D_G^{(0)}(u,v) = \begin{cases} c(u,v) & \text{wenn } u \neq v \text{ und } (u,v) \in E \\ 0 & u = v \\ \infty, & \text{sonst.} \end{cases}$$

Offensichtlich ergibt sich $D_G^{(0)}$ aus der Adjazenzmatrix A_G durch überlagern von 0 in der Diagonale; also $D_G^{(0)} = \min\{A_G, 0_n\}$ wobei 0_n die $n \times n$-Matrix ist, die auf der Diagonalen die 0 enthält und sonst ∞;

genauer:

$$D_G^{(0)}(u,v) = \begin{cases} A_G(u,v) & \text{wenn } u \neq v \\ 0 & u = v, \end{cases}$$

dabei gilt für die Rechnung mit ∞: für jedes $a \in \mathbb{R}_{\geq 0}^{\infty}$ ist $a + \infty = \infty + a = \infty$ und $\min(\infty, a) = \min(a, \infty) = a$. Diese Matrix $\min\{A_G, 0_n\}$ nennen wir *modifizierte Adjanzenzmatrix* und bezeichnen sie durch mA_G. Sie kann einfach von G abgelesen werden.

Wenn nun bereits $D_G^{(k)}$ berechnet ist, dann kann $D_G^{(k+1)}$ nach der folgenden Rekursionsformel berechnet werden:

$$D_G^{(k+1)}(u,v) = \min\{D_G^{(k)}(u,v)\ ,\ D_G^{(k)}(u,k+1) + D_G^{(k)}(k+1,v)\}.$$

Genau diese Berechnung von $D_G^{(0)}, D_G^{(1)}, \ldots, D_G^{(n)}$ führt der Floyd-Warshall-Algorithmus durch.

Algorithmus 8 Floyd-Warshall-Algorithmus

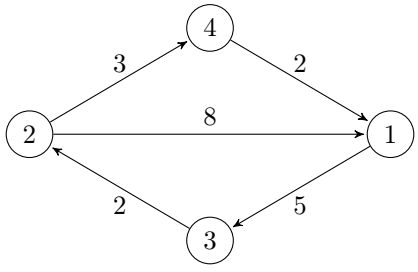
Eingabe: Distanzgraph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$ und $c\colon E \to \mathbb{R}_{\geq 0}$

Ausgabe: $n \times n$-Matrix D_G über $\mathbb{R}_{\geq 0}^{\infty}$

Verfahren:
1 **begin**
2 $\quad D_G^{(0)} := mA_G$;
3 $\quad$ seien $D_G^{(1)}, \ldots, D_G^{(n)}$ $n \times n$-Matritzen
4 $\quad$ **for** $k := 1$ **to** n **do**
5 $\quad\quad$ **for** $u, v \in \{1, \ldots, n\}$ **do**
6 $\quad\quad\quad D_G^{(k)}(u,v) := \min\left\{D_G^{(k-1)}(u,v),\ D_G^{(k-1)}(u,k) + D_G^{(k-1)}(k,v)\right\}$;
7 $\quad D_G := D_G^{(n)}$
8 **end**

Der Floyd-Warshall-Algorithmus hat eine Komplexität von $\Theta(n^3)$.

Beispiel 12.3. Gegeben sei der folgende Distanzgraph G:



Zum Beispiel gilt:
$P_{2,3}^{(0)} = \varnothing$,
$P_{2,3}^{(1)} = \{(2,1,3)\} = P_{2,3}^{(2)}$,
$P_{2,3}^{(3)} = \{(2,1,3), (2,1,3,2,1,3), \ldots\} = \{(2,1,3)^n \mid n \geq 1\}$,
$P_{2,3}^{(4)} = \{(2,1,3), (2,4,1,3), (2,1,3,2,4,1,3), \ldots\} = \{\mathtt{a}_1 \ldots \mathtt{a}_n \mid n \geq 1,\ \mathtt{a}_i \in \{(2,1,3), (2,4,1,3)\}\}$.

G hat die folgende modifizierte Adjazenzmatrix:

$$mA_G = \begin{pmatrix} 0 & \infty & 5 & \infty \\ 8 & 0 & \infty & 3 \\ \infty & 2 & 0 & \infty \\ 2 & \infty & \infty & 0 \end{pmatrix}$$

Es gilt: $D_G^{(0)} = mA_G$. Nun berechnen wir für jedes $1 \leq k \leq 4$ die Matrix $D_G^{(k)}$ (Änderungen gegenüber der Vorgängermatrix sind jeweils kursiv gedruckt).

$$D_G^{(1)} = \begin{pmatrix} 0 & \infty & 5 & \infty \\ 8 & 0 & \mathit{13} & 3 \\ \infty & 2 & 0 & \infty \\ 2 & \infty & \mathit{7} & 0 \end{pmatrix}$$

Beispielsweise gilt:

$D_G^{(1)}(1,3) = \min\{D_G^{(0)}(1,3),\ D_G^{(0)}(1,1) + D_G^{(0)}(1,3)\} = \min\{5,\ 0+5\} = 5$ und
$D_G^{(1)}(2,3) = \min\{D_G^{(0)}(2,3),\ D_G^{(0)}(2,1) + D_G^{(0)}(1,3)\} = \min\{\infty, 8+5\} = 13.$

$$D_G^{(2)} = \begin{pmatrix} 0 & \infty & 5 & \infty \\ 8 & 0 & 13 & 3 \\ \mathit{10} & 2 & 0 & \mathit{5} \\ 2 & \infty & 7 & 0 \end{pmatrix} \qquad D_G^{(3)} = \begin{pmatrix} 0 & \mathit{7} & 5 & \mathit{10} \\ 8 & 0 & 13 & 3 \\ 10 & 2 & 0 & 5 \\ 2 & \mathit{9} & 7 & 0 \end{pmatrix} \qquad D_G^{(4)} = \begin{pmatrix} 0 & 7 & 5 & 10 \\ \mathit{5} & 0 & \mathit{10} & 3 \\ \mathit{7} & 2 & 0 & 5 \\ 2 & 9 & 7 & 0 \end{pmatrix}$$

□

Kommen wir nun zu den Verallgemeinerungen, die wir durch folgende Szenarien beschreiben.

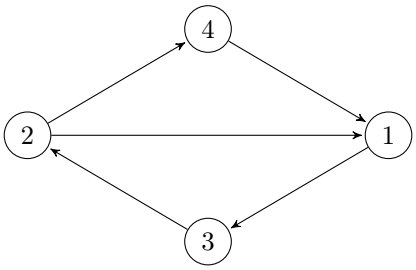
Kapazitätsproblem: Stellen wir uns vor, dass die Knoten des Distanzgraphs G Städte sind und eine Kante (u, v) eine Straßenverbindung zwischen u und v repräsentiert, die man mit höchstens $c(u, v)$ Tonnen Ladung befahren darf. Dann könnte sich beispielsweise eine Spedition die folgende Frage stellen: für zwei beliebige Städte u und v, mit welcher maximalen Last kann man einen Lkw von u nach v schicken?

Erreichbarkeitsproblem: Oder der Spediteur möchte nur ganz einfach wissen, ob er *überhaupt* von u nach v fahren kann.

Zuverlässigkeitsproblem: Jetzt soll der Distanzgraph ein Kommunikationsnetzwerk bestehend aus einer Menge V von Stationen und einer Menge E von Verbindungen repräsentieren. Jede Verbindung $(u, v) \in E$ ist mit einem Wert $c(u, v) \in [0, 1]$ markiert, der besagt, wie groß die Wahrscheinlichkeit einer vollständigen (oder: sicheren) Datenübertragung von u nach v ist. Das verantwortliche Kommunikationsunternehmen möchte dann für jedes $u, v \in V$ die Frage beantworten, mit welcher maximalen Wahrscheinlichkeit die Daten von u nach v in diesem Netzwerk übertragen werden können.

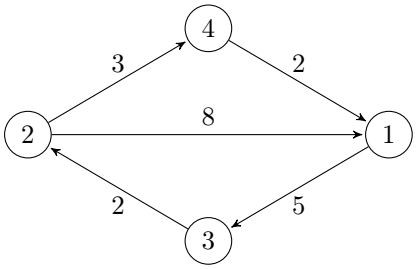
Prozessproblem: Jetzt stellen wir uns vor, dass die Knoten des Distanzgraphen G die Zustände einer technischen Anlage sind; eine Kante $(u, v) \in E$ beschreibt die Möglichkeit, dass die Anlage durch das Ausführen einer Aktion $c(u, v)$ vom Zustand u in den Zustand v übergehen kann. Etwa: Entnahme von Flüssigkeit (das ist die Aktion $c(u, v)$) aus einem Behälter, der nur noch minimal gefüllt ist (das wird durch den Zustand u beschrieben) überführt die Anlage in einen Zustand v, in dem dringend der Behälter wieder aufgefüllt werden muss. Die Verfahrensingenieurin, die für den sicheren Betrieb der Anlage zuständig ist, möchte sich einen Überblick über *alle* auf der Anlage ablaufbaren Prozesse $a_1 a_2 \ldots a_n$, die die Anlage vom Zustand u in den Zustand v überführen, machen und dann gewisse Sicherheitseigenschaften beweisen.

Alle diese Probleme (und noch viele andere) lassen sich durch *einen* Algorithmus berechnen, der eine leichte Verallgemeinerung des Floyd-Warshall-Algorithmus ist. Der Schlüssel zu dieser verblüffenden Möglichkeit liegt in der Analyse der Werte, mit denen in den einzelnen Problemen hantiert wird, und den Operationen darauf. Um diese Analyse durchzuführen, betrachten wir den folgenden Beispielgraphen $G = (V, E)$:



und wichten die Kanten entsprechend der Problemstellung. Dann wollen wir für das Paar $(u, v) = (2, 3)$ das jeweilige Teilproblem lösen. Dazu betrachten wir zwei Wege von 2 nach 3: $p_1 = (2, 4, 1, 3)$ und $p_2 = (2, 1, 3)$. Um die Verallgemeinerung einzusehen, beginnen wir mit dem kürzesten Wegeproblem.

kürzestes Wegeproblem:

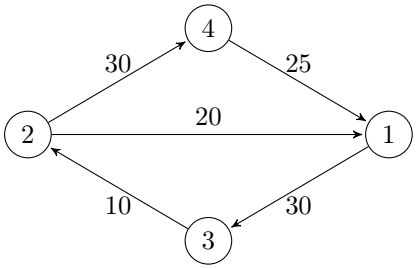


Wie lang ist der kürzeste Weg von 2 nach 3?

Weg p_1: $3 + 2 + 5 = 10$
Weg p_2: $8 + 5 = 13$

also: $\min\{10, 13\} = 10$

Kapazitätsproblem:

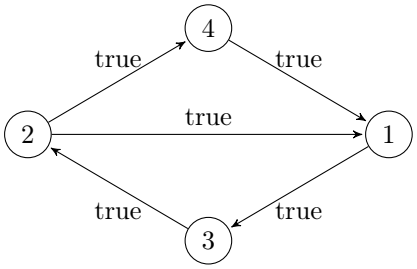


Mit welcher maximalen Tonnage kann man von 2 nach 3 fahren?

Weg p_1: $\min\{30, 25, 30\} = 25$
Weg p_2: $\min\{20, 30\} = 20$

also: $\max\{25, 20\} = 25$

Erreichbarkeitsproblem:

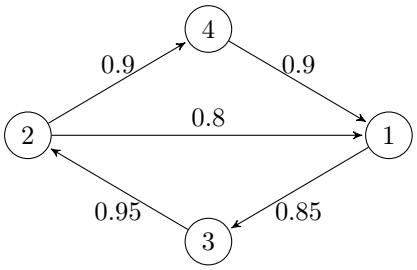


Gibt es eine Verbindung von 2 nach 3?

Weg p_1: $\text{true} \wedge \text{true} \wedge \text{true} = \text{true}$
Weg p_2: $\text{true} \wedge \text{true} = \text{true}$

also: $\text{true} \vee \text{true} = \text{true}$

Zuverlässigkeitskeitsproblem:

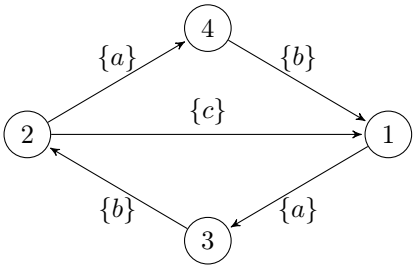


Wie zuverlässig kann die Information von Station 2 zu Station 3 übertragen werden?

Weg p_1: $0.9 \cdot 0.9 \cdot 0.85 = 0.6885$
Weg p_2: $0.8 \cdot 0.85 = 0.68$

also: $\max\{0.6885, 0.68\} = 0.6885$

Prozessproblem:



Wie lautet die Menge aller Prozesse, die die Anlage vom Zustand 2 in den Zustand 3 überführen?

Weg p_1: $\{a\} \circ \{b\} \circ \{a\} = \{aba\}$
Weg p_2: $\{c\} \circ \{a\} = \{ca\}$

also: $\{aba\} \cup \{ca\} = \{aba, ca\}$

(Achtung: Beim Prozessproblem gibt es noch andere Prozesse vom Zustand 2 in den Zustand 3, nämlich z. B. $cabca, cabcabca, \ldots$. Wir werden uns später damit befassen.) Wir erkennen, dass wir in jedem Problem die Werte entlang *eines* Weges mit einer binären Operation $\odot$ verknüpfen, und – wenn für die beiden Wege p_1 und p_2 die Werte $c(p_1)$ bzw. $c(p_2)$ berechnet wurden – die Werte $c(p_1)$ und $c(p_2)$ mit einer anderen binären Operation $\oplus$ verknüpfen:

	Menge S der Werte	$\oplus$	$\odot$	$\mathbf{0}$	$\mathbf{1}$
kürzestes Wegeproblem:	$\mathbb{R}^{\infty}_{\geq 0}$	min	$+$	∞	0
Kapazitätsproblem:	$\mathbb{N}_{\infty}$	max	min	0	∞
Erreichbarkeitsproblem:	{true,false}	$\vee$	$\wedge$	false	true
Zuverlässigkeitsproblem:	$[0, 1]$	max	$\cdot$	0	1
Prozessproblem:	$\mathcal{P}(\Sigma^*)$	$\cup$	$\circ$	$\varnothing$	$\{\varepsilon\}$

Wir nennen $\oplus$ die Akkumulationsoperation und $\odot$ die Pfadoperation. In allen Problemstellungen sind beide Operationen assoziativ. Außerdem ist $\oplus$ kommutativ, d. h. es ist egal in welcher Reihenfolge wir die Werte der Wege in Betracht ziehen. Die Operation $\odot$ ist nicht in jedem Fall kommutativ; beim Prozessproblem ist $\{ab\} = \{a\} \circ \{b\} \neq \{b\} \circ \{a\} = \{ba\}$.

Nun wollen wir ja nicht nur die Wege von u nach v mit $u \neq v$ betrachten, sondern auch mit $u = v$. Dann ist insbesondere (u) ein Weg, der die Länge 0 hat, die Kapazität ∞, Erreichbarkeit true, Zuverlässigkeit 1 und Prozess ε. Es gibt also für jedes Problem einen solchen Wert. Abstrakt bezeichnen wir ihn mit $\mathbf{1}$ und fordern, dass $\mathbf{1}$ neutrales Element für $\odot$ ist, d. h. $s \odot \mathbf{1} = \mathbf{1} \odot s = s$ für jedes $s \in S$.

Ebenso gut kann es sein, dass es keinen Weg von u nach v gibt. Dieser Situation weisen wir die Weglänge ∞, die Kapazität 0, die Erreichbarkeit false, die Zuverlässigkeit 0, und die Prozessmenge $\varnothing$ zu. Abstrakt bezeichnen wir den Wert mit $\mathbf{0}$ und fordern $s \odot \mathbf{0} = \mathbf{0} \odot s = \mathbf{0}$ und $\mathbf{0} \oplus s = s$ für jedes $s \in S$.

Außerdem können wir in unseren Beispielen die folgende Verträglichkeit zwischen $\oplus$ und $\odot$ beobachten:

$$\begin{aligned}
\min\{3 + 2 + 5, 8 + 5\} &= \min\{3 + 2, 8\} + 5 \\
\max\{\min\{30, 25, 30\}, \min\{20, 30\}\} &= \min\{\max\{\min\{30, 25\}, \min\{20\}\}, 30\} \\
(\text{true} \wedge \text{true} \wedge \text{true}) \vee (\text{true} \wedge \text{true}) &= ((\text{true} \wedge \text{true}) \vee (\text{true})) \wedge \text{true} \\
\max\{0.9 \cdot 0.9 \cdot 0.85, 0.8 \cdot 0.85\} &= \max\{0.9 \cdot 0.9, 0.8\} \cdot 0.85 \\
(\{a\} \circ \{b\} \circ \{a\}) \cup (\{c\} \circ \{a\}) &= ((\{a\} \circ \{b\}) \cup \{c\}) \circ \{a\}
\end{aligned}$$

D. h. man kann rechts ausklammern; genauso kann man links ausklammern.

Wir haben gerade von den fünf konkreten Rechenbereichen, sprich: algebraischen Strukturen, oder: Algebren, zu einer ganzen *Klasse* von Algebren abstrahiert, nämlich zur Klasse der *Semiringe*.

Ein *Semiring* ist eine algebraische Struktur $(S, \oplus, \odot, \mathbf{0}, \mathbf{1})$ wobei

- $\oplus$ eine binäre, assoziative und kommutative Operation über S ist (Addition),
- $\odot$ eine binäre, assoziative Operation über S ist (Multiplikation),
- $\mathbf{0}$ ist neutrales Element bzgl. $\oplus$, d. h. $s \oplus \mathbf{0} = s$ für jedes $s \in S$,
- $\mathbf{1}$ ist neutrales Element bzgl. $\odot$, d. h. $s \odot \mathbf{1} = \mathbf{1} \odot s = s$ für jedes $s \in S$,
- $\odot$ ist *distributiv* über $\oplus$, d. h., $s \odot (t \oplus r) = (s \odot t) \oplus (s \odot r)$ und $(s \oplus t) \odot r = (s \odot r) \oplus (t \odot r)$ für jedes $s, t, r \in S$ und

- $\mathbf{0}$ ist ein *Annihilator* für $\odot$, d. h. $\mathbf{0} \odot s = s \odot \mathbf{0} = \mathbf{0}$ für jedes $s \in S$.

Insbesondere ist also $(S, \oplus, \mathbf{0})$ ein kommutatives Monoid und $(S, \odot, \mathbf{1})$ ein Monoid.

In jedem unserer Beispielprobleme haben wir also die Werte in einem speziellen Semiring ausgerechnet; manchmal haben diese auch spezielle Namen:

	$(S, \oplus, \odot, \mathbf{0}, \mathbf{1})$	Semiring
kürzestes Wegeproblem:	$(\mathbb{R}^{\infty}_{\geq 0}, \min, +, \infty, 0)$	tropischer Semiring
Kapazitätsproblem:	$(\mathbb{N}_{\infty}, \max, \min, 0, \infty)$	
Erreichbarkeitsproblem:	$(\{\text{true,false}\}, \vee, \wedge, \text{false,true})$	Boolescher Semiring
Zuverlässigkeitsproblem:	$([0,1], \max, \cdot, 0, 1)$	Viterbi-Semiring
Prozessproblem:	$(\mathcal{P}(\Sigma^*), \cup, \circ, \varnothing, \{\varepsilon\})$	Semiring der formalen Σ-Sprachen
	$(\mathbb{R}^{-\infty}_{\geq 0}, \max, +, -\infty, 0)$	arktischer Semiring
	$(\mathbb{N}, +, \cdot, 0, 1)$	Semiring der natürlichen Zahlen

Weiterhin können wir beobachten, dass jeder Semiring, der zu einem der fünf genannten Probleme gehört, idempotent ist. Ein Semiring $(S, \oplus, \odot, \mathbf{0}, \mathbf{1})$ ist *idempotent* wenn $s \oplus s = s$ für jedes $s \in S$. (Der Semiring der natürlichen Zahlen ist nicht idempotent).

Wir müssen die Analyse der Berechnungen in unseren Beispielproblemen noch etwas fortführen. Für $u = 2$ und $v = 3$ gibt es nicht nur die beiden bisher betrachteten Wege $p_1 = (2, 4, 1, 3)$ und $p_2 = (2, 1, 3)$, sondern es gibt unendlich viele Wege von 2 nach 3:

(2,1,3), (2,4,1,3)

(2,1,3,2,1,3), (2,1,3,2,4,1,3), (2,4,1,3,2,1,3),

...

Das ist insbesondere hier für das Prozessproblem interessant.

Andererseits ist aber $\oplus$ eine zweistellige Operation, d. h. nur für endlich viele Argumente definiert. Schauen wir uns deshalb einmal an, wie in den einzelnen Problemen zu einer Familie $(c(p) \mid p \in P_{u,v})$ (oder allgemeiner: eine Familie[1] $(s_i \mid i \in I)$ mit (beliebiger) Indexmenge I und $s_i \in S$ für jedes $i \in I$) mittels $\oplus$ aufaddiert wird; den entstehenden Wert bezeichnen wir durch $\sum^{\oplus}(s_i \mid i \in I)$ oder kurz: $\sum^{\oplus}_{i \in I} s_i$).

kürzestes Wegeproblem:	$\sum^{\min}_{i \in I} s_i = \inf\{s_i \mid i \in I\}$
Kapazitätsproblem:	$\sum^{\max}_{i \in I} s_i = \sup\{s_i \mid i \in I\}$
Erreichbarkeitsproblem:	$\sum^{\vee}_{i \in I} s_i =$ false wenn alle $s_i =$ false, sonst true
Zuverlässigkeitsproblem:	$\sum^{\max}_{i \in I} s_i = \sup\{s_i \mid i \in I\}$
Prozessproblem:	$\sum^{\cup}_{i \in I} s_i = \bigcup_{i \in I} s_i$

Jetzt kommt wieder eine Abstraktion: Sei $(S, \oplus, \odot, \mathbf{0}, \mathbf{1})$ ein Semiring und $\sum^{\oplus}$ eine Abbildung, die jeder Familie $(s_i \mid i \in I)$ ein Element in S zuordnet. Dann heißt der Semiring $\sum^{\oplus}$-*vollständig*, wenn

- $\sum^{\oplus}$ eine Fortsetzung von $\oplus$ ist, d.h. $\sum^{\oplus}_{i \in \varnothing} s_i = \mathbf{0}$, $\sum^{\oplus}_{i \in \{j\}} s_i = s_j$, $\sum^{\oplus}_{i \in \{j,k\}} s_i = s_j \oplus s_k$,
- $\sum^{\oplus}$ ist assoziativ und kommutativ, d.h. wenn die Indexmenge I partitioniert werden kann durch $(I_j \mid j \in J)$, also $I = \bigcup_{j \in J} I_j$ und $I_l \cap I_k = \varnothing$ für $l \neq k$, dann gilt $\sum^{\oplus}_{j \in J}(\sum_{i \in I_j} s_i) = \sum^{\oplus}_{i \in I} s_i$ und
- $\odot$ ist distributiv über $\sum^{\oplus}$, d. h., $\sum^{\oplus}_{i \in I}(a \odot s_i) = a \odot (\sum^{\oplus}_{i \in I} s_i)$ und $\sum_{i \in I}(s_i \odot a) = (\sum^{\oplus}_{i \in I} s_i) \odot a$ für jedes $a \in S$.

Tatsächlich ist der Semiring

- $(\mathbb{R}^{\infty}_{\geq 0}, \min, +, \infty, 0)$ $\sum^{\min}$-vollständig
- $(\mathbb{N}_{\infty}, \max, \min, 0, \infty)$ $\sum^{\max}$-vollständig

[1] Eine Familie über S mit Indexmenge I ist eine Abbildung $f\colon I \to S$. Wir geben eine Familie durch den Ausdruck $(s_i \mid i \in I)$ an, wobei $s_i = f(i)$.

- $(\{\text{true,false}\}, \vee, \wedge, \text{false,true})$ $\sum^{\vee}$-vollständig
- $([0,1], \max, \cdot, 0, 1)$ $\sum^{\max}$-vollständig
- $(\mathcal{P}(\Sigma^*), \cup, \circ, \varnothing, \{\varepsilon\})$ $\sum^{\cup}$-vollständig.

(Der Semiring der natürlichen Zahlen ist dagegen *nicht* $\sum^{+}$ vollständig, da es zum Beispiel keine natürliche Zahl n gibt, für die $\sum^{+}\{0,1,2,3,\ldots\} = n$ gelten kann.)

Jetzt ist die Analyse abgeschlossen und wir haben eine Klasse von algebraischen Strukturen definiert (nämlich die Klasse aller idempotenten, $\sum^{\oplus}$-vollständigen Semiringe), so dass jedes genannte Beispielproblem in einem passenden Semiring dieser Klasse berechnet wird. Nun verallgemeinern wir

- das kürzeste-Wege-Problem für Distanzgraphen über dem tropischen Semiring zum
- algebraischen Pfadproblem für gewichtete Graphen über einem beliebigen idempotenten, $\sum^{\oplus}$-vollständigen Semiring,

und entsprechend den Floyd-Warshall-Algorithmus zum *Aho-Algorithmus*.

Ein *gewichteter Graph über einem Semiring* $(S, \oplus, \odot, \mathbf{0}, \mathbf{1})$ ist ein Tupel $G = (V, E, c)$ mit $c : E \to S \setminus \{\mathbf{0}\}$. Die Adjazenzmatrix von G ist die $n \times n$-Matrix A_G über S mit

$$A_G(u,v) = \begin{cases} c(u,v) & \text{wenn } (u,v) \in E \\ \mathbf{0} & \text{sonst.} \end{cases}$$

Sei jetzt $G = (V, E, c)$ ein gewichteter Graph über einem $\sum^{\oplus}$-vollständigen, idempotenten Semiring $(S, \oplus, \odot, \mathbf{0}, \mathbf{1})$. Wir nehmen an, dass $V = \{1, \ldots, n\}$. Dann ist das *algebraische Pfadproblem* für G und S die Aufgabe, die $n \times n$-Matrix D_G über S zu berechnen, so dass für jedes $u, v \in V$ gilt:

$$D_G(u,v) = \sum_{p \in P_{u,v}}^{\oplus} c(p),$$

wobei für jeden Weg $p = (v_0, \ldots, v_r)$ mit $r \geq 0$ gilt $c(p) = c(v_0, v_1) \odot c(v_1, v_2) \odot \ldots \odot c(v_{r-1}, v_r)$ (Beachte: $c((u)) = \mathbf{1}$).

Als Vorbereitung für den Aho-Algorithmus definieren wir für jedes $s \in S$ den *Stern von s*. D.h. wir definieren $s^* = \sum_{n \in \mathbb{N}}^{\oplus} s^n$, wobei $s^0 = \mathbf{1}$ und $s^{n+1} = s \odot s^n$. Da S ein $\sum^{\oplus}$-vollständiger Semiring ist, ist s^* wohl-definiert. Für unsere Beispielsemiringe ergibt sich:

$(\mathbb{R}^{\infty}_{\geq 0}, \min, +, \infty, 0)$	$r^* = \sum^{min}\{0,\ r,\ r+r,\ r+r+r, \ldots\} = 0$
$(\mathbb{N}_{\infty}, \max, \min, 0, \infty)$	$k^* = \sum^{max}\{\infty,\ k,\ \min\{k,k\},\ \min\{k,k,k\}, \ldots,\} = \infty$
$(\{\text{true,false}\}, \vee, \wedge, \text{false,true})$	$b^* = \sum^{\vee}\{\text{true}, b, b \wedge b, b \wedge b \wedge b, \ldots\} = \text{true}$
$([0,1], \max, \cdot, 0, 1)$	$s^* = \sum^{max}\{1,\ s,\ s \cdot s,\ s \cdot s \cdot s, \ldots\} = 1$
$(\mathcal{P}(\Sigma^*), \cup, \circ, \varnothing, \{\varepsilon\})$	$L^* = \sum^{\cup}\{\{\varepsilon\}, L, L \circ L, L \circ L \circ L, \ldots\} = L^*$ (der Stern von L wie bereits definiert)

Der Aho-Hopcroft-Ullman Algorithmus [AHU74] (Section 5.6) startet wiederum mit der modifizierten Adjazenzmatrix mA_G. Sie ist die $n \times n$-Matrix $mA_G = A_G \oplus \mathbf{1}_n$, wobei $\mathbf{1}_n$ die $n \times n$-Matrix über S ist, die auf der Diagonalen die $\mathbf{1}$ enthält und sonst die $\mathbf{0}$. Also:

$$mA_G(u,v) = \begin{cases} A_G(u,v) & \text{wenn } u \neq v \\ A_G(u,v) \oplus \mathbf{1} & \text{sonst.} \end{cases}$$

Man sieht leicht, dass der Aho-Hopcroft-Ullman Algorithmus für den tropischen Semiring genau der Floyd-Warshall-Algorithmus ist, denn es gilt:

$$\begin{aligned} & D_G^{(k-1)}(u,v) \oplus \left(D_G^{(k-1)}(u,k) \odot (D_G^{(k-1)}(k,k))^* \odot D_G^{(k-1)}(k,v) \right) \\ &= \min\left\{ D_G^{(k-1)}(u,v),\ D_G^{(k-1)}(u,k) + (D_G^{(k-1)}(k,k))^* + D_G^{(k-1)}(k,v) \right\} \\ &= \min\left\{ D_G^{(k-1)}(u,v),\ D_G^{(k-1)}(u,k) + 0 + D_G^{(k-1)}(k,v) \right\} \end{aligned}$$

Algorithmus 9 Aho-Hopcroft-Ullman-Algorithmus vgl. Alg. 5.5 aus [AHU74]

Eingabe: gewichteter Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$
über einem $\sum^{\oplus}$-vollst., idempotenten Semiring S

Ausgabe: $n \times n$-Matrix D_G über S

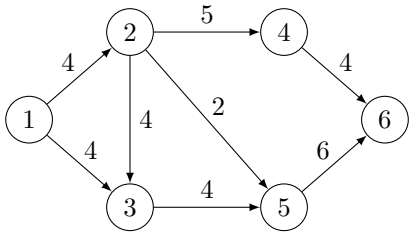
Verfahren:
1 **begin**
2 $D_G^{(0)} := mA_G$;
3 seien $D_G^{(1)}, \ldots, D_G^{(n)}$ $n \times n$-Matritzen
4 **for** $k := 1$ **to** n **do**
5 **for** $u, v \in \{1, \ldots, n\}$ **do**
6 $D_G^{(k)}(u,v) := D_G^{(k-1)}(u,v) \oplus \Big(D_G^{(k-1)}(u,k) \odot (D_G^{(k-1)}(k,k))^* \odot D_G^{(k-1)}(k,v) \Big)$;
7 $D_G := D_G^{(n)}$
8 **end**

$$= \min \Big\{ D_G^{(k-1)}(u,v),\ \ D_G^{(k-1)}(u,k) + D_G^{(k-1)}(k,v) \Big\}$$

Nota bene: Einen gewichteten Graphen (V, E, c) über dem Semiring $(\mathcal{P}(\Sigma^*), \cup, \circ, \emptyset, \{\varepsilon\})$ der formalen Sprachen, bei dem $c((u,v)) \subseteq \Sigma$ für jede Kante $(u,v) \in E$, nennt man auch *endlichen Automat über* Σ. In diesem Fall entspricht der Aho-Hopcroft-Ullman Algorithmus dem Analyseteil des Satzes von Kleene, der besagt, dass jede Sprache, die von einem endlichen Automaten erkannt wird, rational ist.

Abschließend betrachten wir noch zwei weitere Beispiele zum algebraischen Pfadproblem.

Beispiel 12.4. Ein Wegenetz sei durch einen Graphen G modelliert, welcher in der folgenden Abbildung gegeben ist.



Wir wollen nun das Kapazitätsproblem für diesen Graphen lösen und berechnen dazu die Matrizen $D_G^{(i)}$:

$$D_G^{(0)} = \begin{pmatrix} \infty & 4 & 4 & 0 & 0 & 0 \\ 0 & \infty & 4 & 5 & 2 & 0 \\ 0 & 0 & \infty & 0 & 4 & 0 \\ 0 & 0 & 0 & \infty & 0 & 4 \\ 0 & 0 & 0 & 0 & \infty & 6 \\ 0 & 0 & 0 & 0 & 0 & \infty \end{pmatrix} \qquad D_G^{(1)} = D_G^{(0)}$$

$$D_G^{(2)} = \begin{pmatrix} \infty & 4 & 4 & \underline{4} & \underline{2} & 0 \\ 0 & \infty & 4 & 5 & 2 & 0 \\ 0 & 0 & \infty & 0 & 4 & 0 \\ 0 & 0 & 0 & \infty & 0 & 4 \\ 0 & 0 & 0 & 0 & \infty & 6 \\ 0 & 0 & 0 & 0 & 0 & \infty \end{pmatrix} \qquad D_G^{(3)} = \begin{pmatrix} \infty & 4 & 4 & 4 & \underline{4} & 0 \\ 0 & \infty & 4 & 5 & \underline{4} & 0 \\ 0 & 0 & \infty & 0 & 4 & 0 \\ 0 & 0 & 0 & \infty & 0 & 4 \\ 0 & 0 & 0 & 0 & \infty & 6 \\ 0 & 0 & 0 & 0 & 0 & \infty \end{pmatrix}$$

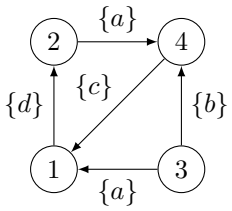
$$D_G^{(4)} = \begin{pmatrix} \infty & 4 & 4 & 4 & 4 & \underline{4} \\ 0 & \infty & 4 & 5 & 4 & \underline{4} \\ 0 & 0 & \infty & 0 & 4 & 0 \\ 0 & 0 & 0 & \infty & 0 & 4 \\ 0 & 0 & 0 & 0 & \infty & 6 \\ 0 & 0 & 0 & 0 & 0 & \infty \end{pmatrix} \qquad D_G^{(5)} = \begin{pmatrix} \infty & 4 & 4 & 4 & 4 & 4 \\ 0 & \infty & 4 & 5 & 4 & 4 \\ 0 & 0 & \infty & 0 & 4 & \underline{4} \\ 0 & 0 & 0 & \infty & 0 & 4 \\ 0 & 0 & 0 & 0 & \infty & 6 \\ 0 & 0 & 0 & 0 & 0 & \infty \end{pmatrix} = D_G^{(6)} = D_G$$

Zum Beispiel

$$\begin{aligned} D_G^{(2)}(1,4) &= \max\left\{D_G^{(1)}(1,4), \min\left\{D_G^{(1)}(1,2), D_G^{(1)}(2,2)^*, D_G^{(1)}(2,4)\right\}\right\} \\ &= \max\left\{0, \min\left\{4, \infty, 5\right\}\right\} \\ &= 4 \end{aligned}$$

□

Beispiel 12.5. Es sei das Prozessproblem für den folgenden Graphen zu berechnen.



Die Matrizen $D_G^{(i)}$ berechnen wir folgendermaßen:

$$D_G^{(0)} = \begin{pmatrix} \{\varepsilon\} & \{d\} & \varnothing & \varnothing \\ \varnothing & \{\varepsilon\} & \varnothing & \{a\} \\ \{a\} & \varnothing & \{\varepsilon\} & \{b\} \\ \{c\} & \varnothing & \varnothing & \{\varepsilon\} \end{pmatrix} \qquad D_G^{(1)} = \begin{pmatrix} \{\varepsilon\} & \{d\} & \varnothing & \varnothing \\ \varnothing & \{\varepsilon\} & \varnothing & \{a\} \\ \{a\} & \{ad\} & \{\varepsilon\} & \{b\} \\ \{c\} & \{cd\} & \varnothing & \{\varepsilon\} \end{pmatrix}$$

$$D_G^{(2)} = \begin{pmatrix} \{\varepsilon\} & \{d\} & \varnothing & \{da\} \\ \varnothing & \{\varepsilon\} & \varnothing & \{a\} \\ \{a\} & \{ad\} & \{\varepsilon\} & \{b, ada\} \\ \{c\} & \{cd\} & \varnothing & \{\varepsilon, cda\} \end{pmatrix} \qquad D_G^{(3)} = D_G^{(2)}$$

$$D_G^{(4)} = \begin{pmatrix} \{dac\}^* & \{dac\}^* \circ \{d\} & \varnothing & \{dac\}^* \circ \{da\} \\ \{acd\}^* \circ \{ac\} & \{acd\}^* & \varnothing & \{acd\}^* \circ \{a\} \\ \{a, bc\} \circ \{dac\}^* & \{a, bc\} \circ \{dac\}^* \circ \{d\} & \{\varepsilon\} & \{b, ada\} \circ \{cda\}^* \\ \{cda\}^* \circ \{c\} & \{cda\}^* \circ \{cd\} & \varnothing & \{cda\}^* \end{pmatrix}$$

Zum Beispiel

$$\begin{aligned} D_G^{(4)}(1,2) &= D_G^{(3)}(1,2) \cup \left(D_G^{(3)}(1,4) \circ D_G^{(3)}(4,4)^* \circ D_G^{(3)}(4,2)\right) \\ &= \{d\} \cup \left(\{da\} \circ \{\varepsilon, cda\}^* \circ \{cd\}\right) \\ &= \{dac\}^* \circ \{d\} \end{aligned}$$

□

13 EM-Algorithmus

Der *Expectation-Maximization-Algorithmus* (kurz: *EM-Algorithmus*) ist eine weit verbreitete statistische Methode, welche insbesondere im Bereich der künstlichen Intelligenz als Lernverfahren angewandt wird.

13.1 Lernverfahren

Was ist unter dem Begriff „Lernverfahren“ zu verstehen? Betrachten wir dazu einmal wie ein Mensch lernt, also neues Wissen erlangt. Im Wesentlichen kann man zwei Lernformen unterscheiden.

Lernen durch Aufnahme von Fakten. Diese Form des Lernens findet zum Beispiel dann statt, wenn sich ein Student auf eine Prüfung vorbereitet und zu diesem Zweck Vorlesungen besucht bzw. Bücher liest. Auf diese Weise erwirbt der Student klares Wissen in einem kurzen Zeitraum.

Empirisches Lernen. Dieser Form liegt eine *wiederholte* Betrachtung der Umwelt zu Grunde. Aus einer Analyse sich wiederholender Verhaltensmuster in den Beobachtungen werden neue Erkenntnisse abgeleitet. Da Wahrnehmungen oberflächlich und widersprüchlich sein können, führt diese Lernform nur zu unscharfem Wissen und erfordert in der Regel viele Beobachtungen (und demnach viel Zeit), um zu vertrauenswürdigen Erkenntnissen zu gelangen.

Die erste Lernform kann nur dann Anwendung finden, wenn Fakten über den betrachteten Wissensbereich vorliegen. Oftmals ist es aber zu aufwändig oder unmöglich, solche Fakten bereitzustellen. In diesem Fall muss man auf das empirische Lernen ausweichen. So ist es zum Beispiel einfach, Menschen an ihren Stimmen zu erkennen, wenn man diese bereits häufig gehört hat. Es ist jedoch unmöglich, einer Freundin *durch Erklärungen beizubringen*, wie die Stimmen zu unterscheiden sind (wenn wir annehmen, dass es keine solch einfachen Erkennungsmerkmale wie Stimmlage oder Dialekt gibt). Stattdessen kann die Freundin dieses Wissen nur empirisch durch mehrfaches Hören der Stimmen erlangen.

Intelligente Systeme benötigen Wissen über ihre Umwelt, um mit dieser geeignet interagieren zu können. Auch hier gibt es zwei Vorgehensweisen, das System mit dem nötigen Wissen auszustatten: entweder die Informatikerin baut es fest in das System in Form von Fakten ein oder sie stattet das System mit der Fähigkeit aus, sich die Kenntnisse durch wiederholte Beobachtungen anzueignen, wodurch sich die genannten Vorteile des empirischen Lernens übertragen. Die letztgenannte Fähigkeit wird durch den EM-Algorithmus bereit gestellt.

Das empirische Lernen ist insbesondere im Bereich der Verarbeitung natürlicher Sprachen von großer Bedeutung. Betrachten wir dazu folgendes Beispiel. Angenommen, Sie finden beim Umgraben Ihres Gartens eine Tontafel mit den in Abbildung 13.1 aufgelisteten zwölf Satzpaaren (siehe [Kni97]).

Sie vermuten, dass es sich bei den Satzpaaren um Übersetzungen außerirdischer Sprachen handelt, nennen wir sie *Centauri* und *Arcturan*. Da in jedem der Satzpaare die beiden Sätze ungefähr gleichviel Worte haben, nehmen Sie an, dass die Sprachen sehr ähnlich sind und Wort für Wort (mit variabler Wortreihenfolge) übersetzt werden.

Voller Begeisterung über Ihren Fund möchten Sie ein Centauri-Arcturan-Wörterbuch erstellen. Die zwölf beobachteten Satzpaare geben dieses nicht direkt preis, außerdem gibt es verschiedene Unsicherheiten, die Ihre Aufgabe erschweren. Zum einen kann es sein, dass ein Wort ein Füllwort ist und innerhalb eines Satzpaares keinen Partner in der Übersetzung hat (ein solches Wort muss es im Satzpaar elf geben, da der erste Satz dort sechs, der zweite jedoch nur fünf Wörter hat). Zum anderen kann ein Wort mehrdeutig sein und in verschiedenen Satzpaaren auch verschieden übersetzt werden. Solche Unsicherheiten müssen Sie in den Erstellungsprozess einbeziehen.

Ein möglicher Eintrag, den Sie in Ihr Wörterbuch einfügen können, ist beispielsweise das Übersetzungspaar „ghirok – hilat“, da das Wort „ghirok“ in den Satzpaaren 3 und 10 vorkommt, das einzige gemeinsame Wort in den jeweiligen Übersetzungen aber nur „hilat“ ist. Auf diese Weise können Sie weitere sinnvolle

1a.	ok-voon ororok sprok .
1b.	at-voon bichat dat .
2a.	ok-drubel ok-voon anok plok sprok .
2b.	at-drubel at-voon pippat rrat dat .
3a.	erok sprok izok hihok ghirok .
3b.	totat dat arrat vat hilat .
4a.	ok-voon anok drok brok jok .
4b.	at-voon krat pippat sat lat .
5a.	wiwok farok izok stok .
5b.	totat jjat quat cat .
6a.	lalok sprok izok jok stok .
6b.	wat dat krat quat cat .
7a.	lalok farok ororok lalok sprok izok enemok .
7b.	wat jjat bichat wat dat vat eneat .
8a.	lalok brok anok plok nok .
8b.	iat lat pippat rrat nnat .
9a.	wiwok nok izok kantok ok-yurp .
9b.	totat nnat quat oloat at-yurp .
10a.	lalok mok nok yorok ghirok clok .
10b.	wat nnat gat mat bat hilat .
11a.	lalok nok crrrok hihok yorok zanzanok .
11b.	wat nnat arrat mat zanzanat .
12a.	lalok rarok nok izok hihok mok .
12b.	wat nnat forat arrat vat gat .

Abbildung 13.1: Zwölf Satzpaare außerirdischer Sprachen (siehe [Kni97])

Annahmen treffen, Übersetzungspaare finden und Möglichkeiten einschränken, so dass Sie schließlich zu dem folgenden oder einem ähnlichen Wörterbuch gelangen.

anok	–	pippat
brok	–	lat
clok	–	bat
crrrok	–	*keines* (?)
drok	–	sat
enemok	–	eneat
erok	–	totat
farok	–	jjat
ghirok	–	hilat
hihok	–	arrat
izok	–	vat / quat
jok	–	krat
kantok	–	oloat
lalok	–	wat / iat
mok	–	gat
nok	–	nnat
ok-drubel	–	at-drubel
ok-voon	–	at-voon
ok-yurp	–	at-yurp
ororok	–	bichat
plok	–	rrat
rarok	–	forat
sprok	–	dat
stok	–	cat
wiwok	–	totat
yorok	–	mat
zanzanok	–	zanzanat

Anstatt das Wörterbuch unter großem Aufwand per Hand zu konstruieren, kann man dafür den EM-Algorithmus nutzen. Wir werden im Folgenden kurz die theoretischen Grundlagen erläutern, die nötig sind, um den EM-Algorithmus einzuführen.

13.2 Zufallsexperimente

Definition 13.1. Ein *Zufallsexperiment* besteht aus einer endlichen Menge X, der *Ergebnismenge*, und einer Funktion $p : X \to [0,1]$ mit der Eigenschaft $\sum_{x \in X} p(x) = 1$. Die Funktion p heißt *Wahrscheinlichkeitsverteilung über X*. Die Menge aller Wahrscheinlichkeitsverteilungen über X wird mit $\mathcal{M}(X)$ bezeichnet. Sei $\mathcal{M}$ eine Menge von Wahrscheinlichkeitsverteilungen über X, also $\mathcal{M} \subseteq \mathcal{M}(X)$. Dann nennt man $\mathcal{M}$ ein *Wahrscheinlichkeitsmodell über X*. $\mathcal{M}(X)$ heißt *unbeschränktes Wahrscheinlichkeitsmodell über X*. □

Beispielsweise ist beim Zufallsexperiment „Werfen einer Münze“ die Ergebnismenge die Menge $\{K, Z\}$ und $p(a)$ gibt die Wahrscheinlichkeit des Ergebnisses a an.

Zufallsexperimente kann man kombinieren und zu neuen komplexeren Zufallsexperimenten zusammensetzen. Wenn wir zum Beispiel gleichzeitig zwei Münzen werfen, dann führen wir zwei Zufallsexperimente mit den Ergebnismengen $\{K, Z\}$ zu einem Zufallsexperiment mit der Ergebnismenge $\{(K,K), (K,Z), (Z,K), (Z,Z)\}$ zusammen (dabei steht das Ergebnis (K,Z) beispielsweise dafür, dass wir bei der ersten Münze Kopf und bei der zweiten Münze Zahl erhalten haben).

Hier wollen wir annehmen, dass sich die beiden Zufallsexperimente nicht gegenseitig beeinflussen. Dann ist die Wahrscheinlichkeitsverteilung des zusammengesetzten Experimentes eindeutig durch die Wahrscheinlichkeitsverteilungen der beiden Ausgangsexperimente bestimmt. Sind $p^1, p^2 \in \mathcal{M}(\{K, Z\})$ die Wahrscheinlichkeitsverteilungen der ersten bzw. zweiten Münze, dann gilt für die Wahrscheinlichkeitsverteilung $p \in \mathcal{M}(\{(K,K), (K,Z), (Z,K), (Z,Z)\})$ des zusammengesetzten Experiments das Folgende:

$$p(a,b) = p^1(a) \cdot p^2(b) \,. \qquad \text{(für jedes } a, b \in \{K, Z\}\text{)}$$

Wir verallgemeinern diesen Begriff in der folgenden Definition.

Definition 13.2. Gegeben seien zwei Zufallsexperimente mit den Ergebnismengen X_1 bzw. X_2 sowie den Wahrscheinlichkeitsverteilungen $p^1 \in \mathcal{M}(X_1)$ bzw. $p^2 \in \mathcal{M}(X_2)$. Das *unabhängige Produkt* dieser beiden Zufallsexperimente ist dann definiert als das Zufallsexperiment über der Ergebnismenge $X_1 \times X_2$ und der Wahrscheinlichkeitsverteilung $p^1 \times p^2 \in \mathcal{M}(X_1 \times X_2)$, wobei

$$(p^1 \times p^2)(a,b) = p^1(a) \cdot p^2(b) \,. \qquad \text{(für jedes } a \in X_1 \text{ und } b \in X_2\text{)}$$

□

Diese Definition lässt sich auf beliebig viele unabhängige Zufallsexperimente erweitern: wirft man zum Beispiel gleichzeitig drei Münzen und zwei Würfel, so ergibt das ein Zufallsexperiment über der Ergebnismenge $\{K, Z\}^3 \times \{1, \ldots, 6\}^2$. Sind alle Würfe gegenseitig voneinander unabhängig, dann ergibt sich die Gesamt-Wahrscheinlichkeitsverteilung eindeutig aus den einzelnen Wahrscheinlichkeitsverteilungen durch komponentenweise Multiplikation.

13.3 Korpora und Korpuswahrscheinlichkeiten

Oftmals ist die einem Zufallsexperiment unterliegende Wahrscheinlichkeitsverteilung unbekannt und kann nur durch ein empirisches Lernverfahren (welches eine Vielzahl an Beobachtungen, also ein mehrfaches Wiederholen des Experimentes erfordert) ermittelt werden. Die Beobachtungen werden als Korpus formalisiert, und das empirische Lernverfahren versucht die Wahrscheinlichkeitsverteilung zu ermitteln, die die Korpuswahrscheinlichkeit (likelihood) maximiert.

Definition 13.3. Sei X eine Ergebnismenge. Eine Abbildung $h\colon X \to \mathbb{R}^{\geq 0}$ heißt *X-Korpus*, wenn es ein $x \in X$ gibt mit $h(x) > 0$ und die Menge $\mathrm{supp}(h) = \{x \in X \mid h(x) > 0\}$ endlich ist. Wenn $p \in \mathcal{M}(X)$ eine Wahrscheinlichkeitsverteilung ist, dann ist die *Korpuswahrscheinlichkeit* (oder: *Likelihood*) *von h unter p* definiert als

$$L(h,p) = \prod_{x \in X} p(x)^{h(x)} \,.$$

Sei weiterhin $\mathcal{M}$ ein Wahrscheinlichkeitsmodell über X, d. h., $\mathcal{M} \subseteq \mathcal{M}(X)$. Dann ist der *Maximum-Likelihood-Schätzer* von h und $\mathcal{M}$ definiert als

$$\text{mle}(h, \mathcal{M}) = \text{argmax}_{p \in \mathcal{M}} L(h, p) \ ,$$

das heißt, $\text{mle}(h, \mathcal{M})$ ist diejenige Wahrscheinlichkeitsverteilung in $\mathcal{M}$, für die die Likelihood maximal wird: $L(h, \text{mle}(h, \mathcal{M})) \geq L(h, p)$ für jedes $p \in \mathcal{M}$. □

Wenn $\mathcal{M}$ das unbeschränkte Wahrscheinlichkeitsmodell ist, d.h., $\mathcal{M} = \mathcal{M}(X)$, dann lässt sich der Maximum-Likelihood-Schätzer leicht bestimmen: er ist gleich der relativen Häufigkeit von h.

Definition 13.4. Sei h ein X-Korpus. Dann ist die *Größe von* h definiert als $|h| = \sum_{x \in X} h(x)$. Die Funktion $\text{rfe}(h) : X \to [0, 1]$ mit

$$\text{rfe}(h)(x) = \frac{h(x)}{|h|} \qquad \text{(für jedes } x \in X\text{)}$$

wird als *relative Häufigkeit von* h (empirical distribution) bezeichnet. □

Satz 13.5. *Sei X eine Ergebnismenge und h ein X-Korpus.*

1. $\text{rfe}(h)$ *ist eine Wahrscheinlichkeitsverteilung über X, also* $\text{rfe}(h) \in \mathcal{M}(X)$.
2. $\text{rfe}(h) = \text{mle}\big(h, \mathcal{M}(X)\big)$.

Beispiel 13.6. Nehmen wir an, wir werfen eine Münze mit der unbekannten Wahrscheinlichkeitsverteilung $p\colon \{K, Z\} \to [0, 1]$ 30 Mal und erhalten dabei 12 Mal Kopf und 18 Mal Zahl. Das fassen wir in dem $\{K, Z\}$-Korpus $h\colon \{K, Z\} \to \mathbb{R}^{\geq 0}$ mit $h(K) = 12$ und $h(Z) = 18$ zusammen.
Die relative Häufigkeit von h ist $\text{rfe}(h)(K) = \frac{12}{30} = \frac{2}{5}$ und $\text{rfe}(h)(Z) = \frac{18}{30} = \frac{3}{5}$. Nach Satz 13.5 ist das gleich dem Maximum-Likelihood-Schätzer von h und $\mathcal{M}(\{K, Z\})$, d. h.,

$$\text{mle}(h, \mathcal{M}(\{K, Z\})) = \text{rfe}(h).$$

Also interpretieren wir die Beobachtungen so, dass die Wahrscheinlichkeitverteilung p der Münze durch $p(K) = \frac{2}{5}$ und $p(Z) = \frac{3}{5}$ gegeben ist. □

Schwieriger ist die Bestimmung von $\text{mle}(h, \mathcal{M})$ wenn $\mathcal{M} \neq \mathcal{M}(X)$ ist. Diese Situation tritt tatsächlich auf: Sei z. B. $X_1 = X_2 = \{K, Z\}$ und $\mathcal{M} = \{p^1 \times p^2 \mid p^1 \in \mathcal{M}(X_1), p^2 \in \mathcal{M}(X_2)\}$. Dann liegt z. B. die Wahrscheinlichkeitsverteilung

$$p(K, K) = p(Z, Z) = 0 \qquad \text{und} \qquad p(K, Z) = p(Z, K) = 0.5$$

in der Menge $\mathcal{M}(X_1 \times X_2) \backslash \mathcal{M}$, denn es gibt keine $p^1, p^2 \in \mathcal{M}(\{K, Z\})$ mit $p = p^1 \times p^2$. Das lässt sich leicht durch einen Widerspruchsbeweis zeigen. Nehmen wir an, dass $p \in \mathcal{M}$. Dann gibt es $p^1, p^2 \in \mathcal{M}(\{K, Z\})$ mit $p = p^1 \times p^2$. Dann gilt $0 = p(K, K) = p^1(K) \cdot p^2(K)$. Also muss

$$p^1(K) = 0 \quad \text{oder} \quad p^2(K) = 0 \tag{13.1}$$

gelten. Aus $0 = p(Z, Z) = p^1(Z) \cdot p^2(Z)$ folgt, dass

$$p^1(Z) = 0 \quad \text{oder} \quad p^2(Z) = 0 \tag{13.2}$$

gilt. Da $0.5 = p(K, Z) = p^1(K) \cdot p^2(Z)$ gilt

$$p^1(K) \neq 0 \quad \text{und} \quad p^2(Z) \neq 0 \tag{13.3}$$

und weil $0.5 = p(Z, K) = p^1(Z) \cdot p^2(K)$ gilt auch

$$p^1(Z) \neq 0 \quad \text{und} \quad p^2(K) \neq 0. \tag{13.4}$$

Die Aussagen 13.1 und 13.2 widersprechen aber den Aussagen 13.3 und 13.4. Also ist $p \notin \mathcal{M}$
Wenn allerdings das verwendete Wahrscheinlichkeitsmodell $\mathcal{M}$ eine gewisse Struktur hat, dann lässt sich $\text{mle}(h, \mathcal{M})$ dennoch explizit bestimmen.

Satz 13.7. *Seien X_1 und X_2 Ergebnismengen und $\mathcal{M} = \{p^1 \times p^2 \mid p^1 \in \mathcal{M}(X_1), p^2 \in \mathcal{M}(X_2)\}$ ein Wahrscheinlichkeitsmodell über $X_1 \times X_2$. Weiterhin sei h ein $X_1 \times X_2$-Korpus. Dann ist*

$$\mathrm{mle}(h, \mathcal{M}) = \mathrm{rfe}(h^1) \times \mathrm{rfe}(h^2) \ ,$$

wobei h^1 der X_1-Korpus und h^2 der X_2-Korpus ist, die wie folgt definiert sind:

$$h^1(x_1) = \sum\nolimits_{x_2 \in X_2} h(x_1, x_2) \ , \qquad \text{(für jedes } x_1 \in X_1\text{)}$$
$$h^2(x_2) = \sum\nolimits_{x_1 \in X_1} h(x_1, x_2) \ . \qquad \text{(für jedes } x_2 \in X_2\text{)}$$

Den Übergang von h nach h^1 oder h^2 nennt man *Marginalisieren.*

Beispiel 13.8. Nehmen wir an, wir werfen zwei Münzen mit den unbekannten Wahrscheinlichkeitsverteilungen p^1 bzw. p^2 dreißig Mal und erhalten dabei den folgenden $\{K, Z\}^2$-Korpus h:

$$h(K,K) = 5 \ , \qquad h(K,Z) = 10 \ , \qquad h(Z,K) = 5 \ , \qquad h(Z,Z) = 10 \ .$$

Durch Marginalisieren erhalten wir die beide Teilkorpora h^1 und h^2:

$$h^1(K) = h(K,K) + h(K,Z) = 15 \ , \qquad h^2(K) = h(K,K) + h(Z,K) = 10 \ ,$$
$$h^1(Z) = h(Z,K) + h(Z,Z) = 15 \ , \qquad h^2(Z) = h(K,Z) + h(Z,Z) = 20 \ .$$

Demnach haben die erste und die zweite Münze vermutlich die Wahrscheinlichkeitsverteilungen $p^1 = \mathrm{rfe}(h^1)$ bzw. $p^2 = \mathrm{rfe}(h^2)$, also

$$p^1(K) = \frac{h^1(K)}{|h^1|} = 1/2 \qquad p^2(K) = \frac{h^2(K)}{|h^2|} = 1/3$$
$$p^1(Z) = \frac{h^1(Z)}{|h^1|} = 1/2 \qquad p^2(Z) = \frac{h^2(Z)}{|h^2|} = 2/3 \qquad \square$$

An dieser Stelle merken wir an, dass sich Satz 13.7 auch auf mehrfache unabhängige Produkte übertragen lässt. Werfen wir zum Beispiel zwei Würfel und eine Münze und erzeugen dabei den $\{1, \ldots, 6\}^2 \times \{K, Z\}$-Korpus h, dann schätzen wir, dass der erste Würfel die Verteilung $\mathrm{rfe}(h^1)$, der zweite Würfel die Verteilung $\mathrm{rfe}(h^2)$ und die Münze die Verteilung $\mathrm{rfe}(h^3)$ aufweist, wobei die beiden $\{1, \ldots, 6\}$-Korpora h^1 und h^2 und der $\{K, Z\}$-Korpus h^3 wie folgt definiert sind:

$$h^1(x_1) = \sum\nolimits_{(x_2,x_3) \in \{1,\ldots,6\} \times \{K,Z\}} h(x_1, x_2, x_3) \ , \qquad \text{(für jedes } x_1 \in \{1, \ldots, 6\}\text{)}$$
$$h^2(x_2) = \sum\nolimits_{(x_1,x_3) \in \{1,\ldots,6\} \times \{K,Z\}} h(x_1, x_2, x_3) \ , \qquad \text{(für jedes } x_2 \in \{1, \ldots, 6\}\text{)}$$
$$h^3(x_3) = \sum\nolimits_{(x_1,x_2) \in \{1,\ldots,6\}^2} h(x_1, x_2, x_3) \ . \qquad \text{(für jedes } x_3 \in \{K, Z\}\text{)}$$

13.4 Korpora mit unvollständigen Daten

In den vorangegangenen Abschnitten haben wir uns mit der Maximum-Likelihood-Schätzung zu einem X-Korpus h und einem Wahrscheinlichkeitsmodell $\mathcal{M} \subseteq \mathcal{M}(X)$ beschäftigt. Solche Korpora werden als *Korpora mit vollständigen Daten* bezeichnet, da sie jedem einzelnen Ergebnis x in der Ergebnismenge X einen Wert zuordnen.

Korpora werden empirisch erzeugt, zum Beispiel durch die vielfache Ausführung eines Zufallsexperiments. In der Praxis sind Beobachtungen jedoch häufig unvollständig und geben nicht alle Details zum Ausgang des Experimentes preis: verschiedene Ergebnisse können zur gleichen Beobachtung führen; von der Beobachtung kann man dann nicht genau darauf schließen, welches Ergebnis das Experiment genommen hat, da *jedes* Ergebnis in Frage kommt, welches die erfolgte Beobachtung nach sich zieht. Betrachten wir dazu das folgende Beispiel.

Beispiel 13.9. Person A wirft zwei Münzen. Dieses Zufallsexperiment hat die Ergebnismenge $X = \{(K,K),(K,Z),(Z,K),(Z,Z)\}$. Person B kann die Münzen nicht sehen, erfährt jedoch von Person A, wie oft die Kopfseite nach dem Wurf zu sehen ist.

Vom Standpunkt der Person B gesehen, gibt es bei diesem Experiment nur drei mögliche Beobachtungen, nämlich die Werte in der Menge $Y = \{0,1,2\}$. Macht Person B die Beobachtung 0, dann kann sie folgern, dass das Experiment das Ergebnis (Z,Z) genommen hat. Auf die gleiche Art kann sie bei der Beobachtung 2 auf das Ergebnis (K,K) schließen. Bei der Beobachtung 1 weiß sie, dass eines der Ergebnisse (K,Z) oder (Z,K) eintrat, jedoch nicht welches. □

Zufallsexperimente mit unvollständigen Beobachtungen sind also durch eine Ergebnismenge X und eine Beobachtungsmenge Y gekennzeichnet, wobei einem Ergebnis $x \in X$ genau eine Beobachtung $y \in Y$ zugeordnet ist; einer Beobachtung dagegen können ein oder mehrere Ergebnisse zugeordnet sein.

Definition 13.10. Sei X eine Ergebnismenge und Y eine Menge (von *Beobachtungen*). Dann nennen wir eine Funktion $\text{yield}\colon X \to Y$ eine *Beobachtungsfunktion*. Die Umkehrabbildung von yield ist dann die Funktion $A\colon Y \to \mathcal{P}(X)$, die wie folgt definiert ist:

$$A(y) = \{x \in X \mid \text{yield}(x) = y\} \,. \qquad \text{(für jedes } x \in X\text{)}$$

Die Funktion A heißt *Analysator* und ordnet jeder Beobachtung y die Menge der Ergebnisse zu, die zur Beobachtung y führen. Die Elemente in $A(y)$ werden als *Analysen von* y bezeichnet. □

Im Beispiel 13.9 lautet die Beobachtungsfunktion wie folgt:

$$\text{yield}(K,K) = 2\,, \qquad \text{yield}(K,Z) = 1\,, \qquad \text{yield}(Z,K) = 1\,, \qquad \text{yield}(Z,Z) = 0\,.$$

Der zugehörige Analysator ist:

$$A(0) = \{(Z,Z)\}\,, \qquad A(1) = \{(K,Z),(Z,K)\}\,, \qquad A(2) = \{(K,K)\}\,.$$

Führen wir ein Zufallsexperiment mit unvollständigen Beobachtungen mehrfach durch und erzeugen uns auf diese Weise einen Korpus h, dann ist h ein Y-Korpus und kein X-Korpus, denn wir können nur die Beobachtungen und nicht die Ergebnisse des Experimentes zählen. In diesem Fall wird h als ein *Korpus mit unvollständigen Daten* bezeichnet.

Wir befassen uns im Folgenden mit dem Problem, eine Maximum-Likelihood-Schätzung zu einem Korpus mit unvollständigen Daten, also einem Y-Korpus, und einem Wahrscheinlichkeitsmodell über X durchzuführen. Da wir mit dem in den vorherigen Abschnitten vorgestellten Verfahren eine solche Schätzung nur dann durchführen können, wenn der Korpus und das Wahrscheinlichkeitsmodell über der gleichen Menge definiert sind, müssen wir nun unsere Definition auf den Fall der unvollständigen Beobachtungen erweitern.

Definition 13.11. Sei h ein Y-Korpus und $p \in \mathcal{M}(X)$. Die *Korpuswahrscheinlichkeit* (oder: *Likelihood*) *von* h *unter* p ist dann definiert als

$$L(h,p) = \prod_{y \in Y} \Big(\sum\nolimits_{x \in A(y)} p(x) \Big)^{h(y)} \,.$$

Sei $\mathcal{M} \subseteq \mathcal{M}(X)$. Der *Maximum-Likelihood-Schätzer* von h und $\mathcal{M}$ ist dann definiert wie im Fall mit vollständigen Daten, also

$$\text{mle}(h,\mathcal{M}) = \text{argmax}_{p \in \mathcal{M}} L(h,p) \,. \qquad \square$$

Wenn nichts über die Struktur von $\mathcal{M}$ bekannt ist, dann kann man mit Hilfe des *Expectation Maximization Algorithmus* (EM-Algorithmus) (Algorithmus 10) den $\text{mle}(h,\mathcal{M})$ approximieren wie der folgende Satz zeigt. Leider konvergiert der EM-Algorithmus in manchen Fällen nur zu einem lokalen Maximum (und nicht zum globalen Maximum). Der EM-Algorithmus erzeugt dabei eine Sequenz $q_1, q_2, q_3, \ldots$ von Wahrscheinlichkeitsverteilungen.

Algorithmus 10 EM-Algorithmus

Eingabe ein Y-Korpus h;
ein Analysator $A\colon Y \to \mathcal{P}(X)$;
ein Wahrscheinlichkeitsmodell $\mathcal{M} \subseteq \mathcal{M}(X)$ über X;
ein $q_0 \in \mathcal{M}$, so dass $q_0(x) > 0$ für jedes $x \in X$.

Ausgabe eine Sequenz $q_1, q_2, q_3, \ldots$ von Elementen aus $\mathcal{M}$.

1 **für jedes** $i = 1, 2, 3, \ldots$

2 **E-Schritt** berechne den X-Korpus h_i:

$$h_i(x) = h(\text{yield}(x)) \cdot \frac{q_{i-1}(x)}{\sum_{x' \in A(\text{yield}(x))} q_{i-1}(x')}$$

3 **M-Schritt** berechne den Maximum-Likelihood-Schätzer von h_i und $\mathcal{M}$:

$$q_i = \text{argmax}_{p \in \mathcal{M}} L(h_i, p)$$

4 print q_i

Satz 13.12. *Sei $q_1, q_2, q_3, \ldots$ die durch den EM-Algorithmus berechnete Sequenz von Wahrscheinlichkeitsverteilungen über X. Dann gilt*

$$L(h, q_0) \leq L(h, q_1) \leq L(h, q_2) \leq L(h, q_3) \leq \cdots \leq L(h, \text{mle}(h, \mathcal{M})) \ .$$

Beispiel 13.13 (Fortsetzung von Beispiel 13.9)**.** A wirft zwei Münzen 15 mal und teilt B mit, dass bei 4 Würfen 0 Mal Kopf gefallen ist, bei 9 Würfen 1 Mal Kopf und bei 2 Würfen 2 mal Kopf. Das fassen wir in dem folgenden Y-Korpus h zusammen:

$$h(0) = 4 \ , \qquad h(1) = 9 \ , \qquad h(2) = 2 \ .$$

Als Wahrscheinlichkeitsmodell wählen wir $\mathcal{M} = \{p^1 \times p^2 \mid p^1, p^2 \in \mathcal{M}(\{K, Z\})\}$. Die einzelnen Schritte des EM-Algorithmus sind leicht auszuführen. Die Ausführung des Algorithmus hängt vom Startwert p_0 ab. Wir haben in der folgenden Tabelle den Ablauf für vier verschiedene Startwerte zusammengefasst. Um die Tabelle möglichst kompakt darzustellen, notieren wir jede Wahrscheinlichkeitsverteilung $p \in \mathcal{M}$ in der Form (a, b), wobei a und b die Wahrscheinlichkeiten der ersten bzw. zweiten Münze sind, nach einem Wurf Kopf zu zeigen, d. h. $p^1(K) = a$ und $p^2(K) = b$.

	Ablauf 1	Ablauf 2	Ablauf 3	Ablauf 4
(a_0, b_0)	$(0.200, 0.500)$	$(0.900, 0.600)$	$(0.000, 1.000)$	$(0.400, 0.400)$
(a_1, b_1)	$(0.253, 0.613)$	$(0.648, 0.219)$	$(0.133, 0.733)$	$(0.433, 0.433)$
(a_2, b_2)	$(0.239, 0.628)$	$(0.654, 0.213)$	$(0.165, 0.687)$	$(0.433, 0.433)$
(a_3, b_3)	$(0.228, 0.639)$	$(0.658, 0.208)$	$(0.180, 0.679)$	$(0.433, 0.433)$
(a_4, b_4)	$(0.219, 0.648)$	$(0.661, 0.205)$	$(0.188, 0.674)$	$(0.433, 0.433)$
(a_5, b_5)	$(0.213, 0.654)$	$(0.663, 0.204)$	$(0.193, 0.671)$	$(0.433, 0.433)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
(a_{20}, b_{20})	$(0.200, 0.667)$	$(0.667, 0.200)$	$(0.200, 0.667)$	$(0.433, 0.433)$

Betrachten wir die erste Iteration des EM-Algorithmus für Ablauf 1 im Detail, sei also $a_0 = 1/5$, $b_0 = 1/2$ und $i = 1$. D.h. $q_0 = p_0^1 \times p_0^2$ mit $p_0^1(K) = a_0$ und $p_0^2(K) = b_0$. Im E-Schritt ermitteln wir aus dem Korpus $h\colon Y \to \mathbb{R}^{\geq 0}$ über unvollständigen Daten mithilfe des Analysators A und des Wahrscheinlichkeitsverteilung q_0 einen Korpus $h_1\colon X \to \mathbb{R}^{\geq 0}$ über vollständigen Daten. Für die vier Elemente aus X ergibt sich:

$$h_1(K,K) = h(\text{yield}(K,K)) \cdot \frac{q_0(K,K)}{\sum_{x' \in A(\text{yield}(K,K))} q_0(x')} = h(2) \cdot \frac{q_0(K,K)}{\sum_{x' \in \{(K,K)\}} q_0(x')} = 2 \cdot \frac{q_0(K,K)}{q_0(K,K)} = 2$$

$$h_1(K,Z) = h(\text{yield}(K,Z)) \cdot \frac{q_0(K,Z)}{\sum_{x' \in A(\text{yield}(K,Z))} q_0(x')} = h(1) \cdot \frac{q_0(K,Z)}{q_0(K,Z) + q_0(Z,K)}$$
$$= 9 \cdot \frac{1/5 \cdot 1/2}{1/5 \cdot 1/2 + 4/5 \cdot 1/2} = 9 \cdot \frac{1}{5} = 9/5$$

$$h_1(Z,K) = h(\text{yield}(Z,K)) \cdot \frac{q_0(Z,K)}{\sum_{x' \in A(\text{yield}(Z,K))} q_0(x')} = 9 \cdot \frac{4/5 \cdot 1/2}{1/5 \cdot 1/2 + 4/5 \cdot 1/2} = 9 \cdot \frac{4}{5} = 36/5$$

$$h_1(Z,Z) = h(\text{yield}(Z,Z)) \cdot \frac{q_0(Z,Z)}{\sum_{x' \in A(\text{yield}(Z,Z))} q_0(x')} = 4 \cdot \frac{q_0(Z,Z)}{q_0(Z,Z)} = 4$$

Im M-Schritt ermitteln wir den Maximum-Likelihood-Schätzer von h_1 und dem gegebenen Wahrscheinlichkeitsmodell $\mathcal{M}$. Da alle Elemente von $\mathcal{M}$ durch unabhängiges Produkt zweier Wahrscheinlichkeitsverteilungen definiert sind, können wir Satz 13.7 anwenden, es ergibt sich

$$\text{argmax}_{p \in \mathcal{M}} L(h_1, p) = \text{mle}(h_1, \mathcal{M}) \overset{\text{Satz } 13.7}{=} \text{rfe}(h_1^1) \times \text{rfe}(h_1^2)$$

wobei h_1^1 und h_1^2 durch Marginalisierung aus h_1 entstehen. Berechnen wir also zunächst $h_1^[(1)]$ und $h_1^[(2)]$:

$$h_1^1(K) = h_1(K,K) + h_1(K,Z) = 2 + 9/5 = 19/5 \qquad h_1^2(K) = h_1(K,K) + h_1(Z,K) = 2 + 36/5 = 46/5$$
$$h_1^1(Z) = h_1(Z,K) + h_1(Z,Z) = 36/5 + 4 = 56/5 \qquad h_1^2(Z) = h_1(K,Z) + h_1(Z,Z) = 9/5 + 4 = 29/5$$

Nun ermitteln wir die relativen Häufigkeiten von h_1^1 und h_1^2:

$$\text{rfe}(h_1^1)(K) = \frac{19/5}{19/5 + 56/5} = \frac{19}{75} \approx 0.253 =: a_1 \qquad \text{rfe}(h_1^2)(K) = \frac{46/5}{46/5 + 29/5} = \frac{46}{75} \approx 0.613 =: b_1$$
$$\text{rfe}(h_1^1)(Z) = \frac{56/5}{19/5 + 56/5} = \frac{56}{75} \approx 0.747\ (= 1 - a_1) \qquad \text{rfe}(h_1^2)(Z) = \frac{29/5}{46/5 + 29/5} = \frac{29}{75} \approx 0.387\ (= 1 - b_1)$$

Die anderen Werte in der Ablauftabelle ergeben sich auf die gleiche Weise.

Offensichtlich konvergiert der EM-Algorithmus zu einem der Werte $(1/5, 2/3)$, $(2/3, 1/5)$ oder $(13/30, 13/30)$. Das entspricht den folgenden drei Wahrscheinlichkeitsverteilungen $\hat{p}_1$, $\hat{p}_2$, und $\hat{p}_3$ über $\{K,Z\}^2$:

	$\hat{p}_1$	$\hat{p}_2$	$\hat{p}_3$
(K,K)	$1/5 \cdot 2/3 = 2/15$	$2/3 \cdot 1/5 = 2/15$	$13/30 \cdot 13/30 = 169/900$
(K,Z)	$1/5 \cdot 1/3 = 1/15$	$2/3 \cdot 4/5 = 8/15$	$13/30 \cdot 17/30 = 221/900$
(Z,K)	$4/5 \cdot 2/3 = 8/15$	$1/3 \cdot 1/5 = 1/15$	$17/30 \cdot 13/30 = 221/900$
(Z,Z)	$4/5 \cdot 1/3 = 4/15$	$1/3 \cdot 4/5 = 4/15$	$17/30 \cdot 17/30 = 289/900$

Jetzt bestimmen wir die Likelihood dieser drei Wahrscheinlichkeitsverteilungen. Zunächst gilt für beliebiges p:

$$L(h,p) = \prod_{y \in Y} \left(\sum_{x \in A(y)} p(x) \right)^{h(y)}$$

und hier für unser konkretes h und A (siehe Abbildung 13.2):

$$L(h,p) = \left(\sum_{x \in A(0)} p(x) \right)^{h(0)} \cdot \left(\sum_{x \in A(1)} p(x) \right)^{h(1)} \cdot \left(\sum_{x \in A(2)} p(x) \right)^{h(2)}$$
$$= p(Z,Z)^4 \cdot (p(K,Z) + p(Z,K))^9 \cdot p(K,K)^2$$

Dann gilt

$$L(h, \hat{p}_1) = 0.90596 \cdot 10^{-6} \qquad L(h, \hat{p}_2) = 0.90596 \cdot 10^{-6} \qquad L(h, \hat{p}_3) = 0.62305 \cdot 10^{-6}$$

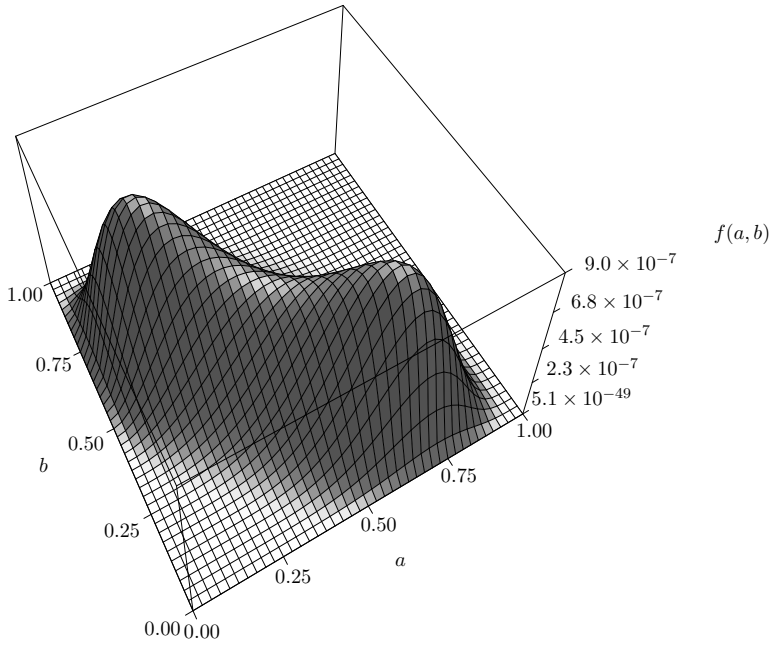


Abbildung 13.2: Funktionsgraph von $L(h, p^1 \times p^2)$ aus Beispiel 13.13 in Abhängigkeit von den Parametern $a = p^1(K)$ und $b = p^2(K)$. Beachten Sie, dass die Funktion in dem dargestellten Bereich zwei lokale Maxima und einen Sattelpunkt hat.

Da $L(h, \hat{p}_3) < L(h, \hat{p}_1) = L(h, \hat{p}_2)$, ist $\hat{p}_1$ (oder $\hat{p}_2$) die gesuchte Wahrscheinlichkeitsverteilung. Da $\hat{p}_1 \in \{p^1 \times p^2 \mid p^1, p^2 \in \mathcal{M}(\{K, Z\})\}$, gibt es $p^1, p^2 \in \mathcal{M}(\{K, Z\})$ mit $\hat{p}_1 = p^1 \times p^2$. Wir ermitteln:

$$p^1(K) = 1/5, \qquad p^1(Z) = 4/5, \qquad p^2(K) = 2/3, \qquad p^2(Z) = 1/3. \qquad \square$$

Es ist leicht zu sehen, dass $\hat{p}_2 = p^2 \times p^1$.

14 Prinzipien für die Struktur von Algorithmen

In den Kapiteln 9 bis 13 haben wir für verschiedene Problemfelder einzelne Algorithmen kennengelernt. Hier wollen wir Prinzipien für die Struktur von Algorithmen diskutieren. Die einzelnen Algorithmen folgen dann dem einen oder anderen (oder mehreren) Prinzip(ien). Manchmal ist die Zuordnung eines Algorithmus zu einem algorithmischen Prinzip allerdings nicht so leicht.

In diesem Kapitel wollen wir drei solche Prinzipien diskutieren:

- Divide-and-Conquer,
- Dynamische Programmierung und
- Backtracking.

14.1 Divide-and-Conquer

Oft lässt sich ein Problem *top-down* in eine Anzahl kleinerer Instanzen desselben Problems (d. h. Teilprobleme) zerlegen. Diese Teilprobleme werden dann (ggf. durch erneutes Zerlegen) rekursiv gelöst. Danach werden die Lösungen der Teilprobleme zu einer Lösung für das Problem zusammengesetzt. Diese Vorgehensweise wird als *Divide-and-Conquer* (teile und herrsche) bezeichnet.

Betrachtet man den Rechenzeitaufwand $A(n)$ für die Berechnung der Lösung eines Problems der Größe n, so lässt sich folgende Rekursionsgleichung für $A(n)$ aufstellen, wenn das Problem in b Teilprobleme der Größe n/d zerlegt wird:

$$A(n) = b \cdot A(n/d) + (\text{Aufwand für das Zusammenfügen}) .$$

Quicksort und binäres Suchen sind Beispiele für Divide-and-Conquer Algorithmen. Hier geben wir drei weitere Algorithmen an, die diesem Prinzip folgen.

14.1.1 Multiplikation zweier großer Zahlen

Beispiel 14.1. Als ein Beispiel für die Methode Divide-and-Conquer betrachten wir die Multiplikation zweier großer positiver, ganzer Zahlen. Nach der Schulmethode würde man die Zahlen 8765 und 4321 folgendermaßen multiplizieren:

$$\begin{array}{r} 8765 \cdot 4321 \\ \hline 35060 \\ 26295 \\ 17530 \\ 8765 \\ \hline 37873565 \end{array}$$

□

Betrachten wir die Rechenzeit dieses Algorithmus. Wenn man für die Multiplikation von zwei einziffrigen Zahlen eine Rechenzeiteinheit veranschlagt und die Addition von einziffrigen Zahlen außer Betracht lässt, dann ergibt sich im wesentlichen eine (in Abhängigkeit von der Länge der zu multiplizierenden Zahlen) quadratische Zeitkomplexität.

Eine Multiplikation nach der Divide-and-Conquer Methode wurde von Karatsuba vorgestellt; sie hat eine Zeitkomplexität von $O(n^{1.59})$. Diese Methode setzt voraus, dass die Anzahl n der Ziffern der beiden Zahlen eine Zweierpotenz ist. Im Verfahren werden die beiden n–ziffrigen Zahlen x und y in zwei gleiche Teile zerlegt:

x in x_1 und x_2,
y in y_1 und y_2,
mit $x = x_1 \cdot 10^{n/2} + x_2$ und $y = y_1 \cdot 10^{n/2} + y_2$.

Multipliziert man diese beiden Darstellungen, so ergibt sich:

$$\begin{aligned} x \cdot y &= (x_1 \cdot 10^{n/2} + x_2) \cdot (y_1 \cdot 10^{n/2} + y_2) \\ &= x_1 y_1 \cdot 10^n + (x_1 y_2 + x_2 y_1) \cdot 10^{n/2} + x_2 y_2 \\ &= x_1 y_1 \cdot 10^n + (x_1 y_1 + x_2 y_2 - (x_2 - x_1)(y_2 - y_1)) \cdot 10^{n/2} + x_2 y_2 \end{aligned}$$

Durch die Aufteilung hat man also die Multiplikation von zwei n-ziffrigen Zahlen auf drei Multiplikationen von $n/2$–ziffrigen Zahlen zurückgeführt (nämlich: $x_1 y_1$, $x_2 y_2$ und $(x_2 - x_1)(y_2 - y_1)$). Das Zusammenfügen der Teilergebnisse (durch Addition, Subtraktion, Multiplikation mit $10^{n/2}$) hat linearen Zeitaufwand. Damit ergibt sich für den Aufwand:

$$A(n) = 3 \cdot A(n/2) + c \cdot n \text{ mit } c > 0 \, .$$

Setzt man $A(1) = a$, so erhält man als Lösung für diese Rekursionsgleichung

$$A(n) = (a + 2c) \cdot n^{\log_2 3} + (-2c)n \, .$$

Damit ist der Rechenaufwand dieses Multiplikationsalgorithmus von der Art $O(n^{\log_2 3})$.

14.1.2 Fibonacci

Im Jahre 1202 hat Leonardo von Pisa (auch Fibonacci genannt) bezogen auf einen idealisierten Lebensraum die Entwicklung der Kaninchenpopulation mathematisch formuliert. Der idealisierte Lebensraum lässt sich durch die folgenden drei Regeln beschreiben:

1. Im ersten Jahr gibt es ein Kaninchenpaar (KP).
2. Jedes KP hat erstmals nach zwei Jahren *ein KP* als Nachwuchs und gebiert dann jährlich *ein KP*.
3. Alle Kaninchen sind unsterblich.

Die Zahl $KP(n)$ der KP nach n Jahren gibt die sogenannte Fibonacci-Folge wieder:

n	0	1	2	3	4	5	6	7	8	9	10
$KP(n)$	0	1	1	2	3	5	8	13	21	34	55

Abbildung 14.1 zeigt die zeitliche Entwicklung der Kaninchenpopulation in den ersten sieben Jahren als Stammbaum der Kaninchenpaar 1 bis 13. Hierbei hat im zweiten Jahr hat das erste KP erstmals Nachwuchs, so dass sich der Stammbaum in zwei Teiläste aufspaltet, der linke Teilbaum für das erste KP und der Teilbaum das neugeborene zweite KP. In Abhängigkeit vom Jahrgang lässt sich KP als Funktion rekursiv beschreiben; die Funktion nennen wir Fibonacci-Funktion:

$KP(0) = 0$
$KP(1) = 1$
$KP(n+2) = KP(n+1) + KP(n)$ wenn $n \geq 0$

Auch hierin ist das Prinzip Divide-and-Conquer enthalten: Das Problem $KP(n+2)$ zu berechnen wird in die Teilprobleme $KP(n+1)$ und $KP(n)$ zerlegt, und deren Lösung wird dann (durch einfache Addition) zusammengefügt.

Die Fibonacci-Funktion kann durch folgende Funktionsprozedur berechnet werden.

```
1 | int fib_rek(int n)
2 | { if (n <= 1) return n;
3 |   else return (fib_rek(n-1) + fib_rek(n-2));
4 | }
```

Wie viele Rechenschritte benötigt der durch die rekursive Funktionsbeschreibung implizierte Algorithmus für die Berechnung von $KP(n)$?

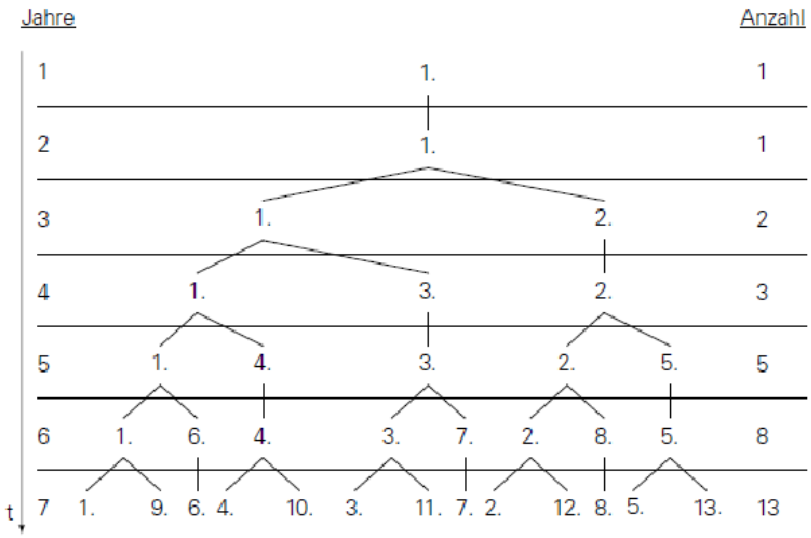


Abbildung 14.1: Entwicklung der Kaninchenpopulation

Beispiel 14.2.

$$\begin{aligned} KP(5) &= KP(4) + KP(3) \\ &= KP(3) + KP(2) + KP(2) + KP(1) \\ &= KP(2) + KP(1) + KP(1) + KP(0) + KP(1) + KP(0) + KP(1) \\ &= KP(1) + KP(0) + KP(1) + KP(1) + KP(0) + KP(1) + KP(0) + KP(1) \\ &= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 \\ &= 5 \end{aligned}$$

Man erkennt, dass für die Berechnung von $KP(5)$ insgesamt fünfmal $KP(1)$ und dreimal $KP(0)$ berechnet (d. h. nachgeschaut) werden muss. □

Man kann beweisen, dass die Funktion, die den Zeitbedarf abschätzt, in der Klasse $O(2^n)$ liegt, d. h. der Algorithmus hat exponentiellen Zeitbedarf. Der Platzbedarf ist linear, d. h. von der Art $O(n)$.

14.1.3 Towers of Hanoi

Bei Towers of Hanoi handelt es sich um ein Ein-Personen-Spiel. Es gibt drei Plätze: A, B und C. Zu Beginn des Spiels liegen auf Platz A n Scheiben unterschiedlicher Größe; die Scheiben sind der Größe nach sortiert, die größte Scheibe liegt unten. Das Ziel des Spiels ist, die n Scheiben von Platz A nach Platz B zu transportieren, wobei zu jedem Zeitpunkt des Spiels (also auch zu Beginn des Spiels) die folgende Bedingung erfüllt sein muss: Es liegt keine Scheibe auf einer kleineren Scheibe. Anschaulich gesprochen heißt das, dass der aus den Scheiben bestehende Stapel einen Kegel bildet.

Als elementaren Spielzug verwenden wir mit $i, j \in \{A, B, C\}$ und $i \neq j$:

$$move(i, j): \text{ transportiere oberste Scheibe von Platz } i \text{ nach Platz } j.$$

Definieren wir die Funktion *towers* wie folgt:
Für alle $n \in \mathbb{N}$ und $i, j, k \in \{A, B, C\}$, wobei $i \neq j$, $j \neq k$ und $k \neq i$:

$$\begin{aligned} towers(n+1, i, j, k) &= towers(n, i, k, j)\ move(i, j)\ towers(n, k, j, i) \\ towers(0, i, j, k) &= \varepsilon \end{aligned}$$

dann erzeugt der Aufruf $towers(n, A, B, C)$ eine Sequenz von elementaren Spielzügen, so dass nach deren Ausführung ein Stapel der Höhe n von A nach B unter Zuhilfenahme von C transportiert worden ist.

Hier haben wir also das Problem $n+1$ Scheiben von A nach B zu transportieren in zwei entsprechende Probleme mit n Scheiben zerlegt.

Die Zeitkomplexität dieses Verfahrens ist exponentiell, d. h. sie liegt in der Klasse $O(2^n)$, die Platzkomplexität ist linear, d. h. sie liegt in der Klasse $O(n)$ (man bedenke, dass man sich Aufrufe von *towers* für die spätere Bearbeitung merken muss).

Ein platzeffizienteres Verfahren sieht folgendermaßen aus:

```
 1 if (n ungerade)
 2   {RICHTUNG = (A -> B -> C -> A)  /* nach rechts */}
 3 else
 4   {RICHTUNG = (A -> C -> B -> A)  /* nach links */}
 5
 6 loop = 1;
 7 while (loop)
 8 { verschiebe kleinste Scheibe um einen Platz in RICHTUNG;
 9   if (Aufgabe erfüllt)
10     break;
11   führe den einzig möglichen Schritt durch, der sich nicht
12     auf die kleinste Scheibe bezieht
13 }
```

Hier ist die Zeitkomplexität exponentiell und die Platzkomplexität konstant. Das Laufzeitverhalten hat sich also nicht verbessert, sondern nur die Platzkomplexität.

14.2 Dynamische Programmierung

Ein anderes Strukturprinzip für Algorithmen ist die dynamische Programmierung. Hier werden zur Lösung eines Problems der Größe n alle relevanten Probleme kleinerer Größe (beginnend bei Problemgröße 1) gelöst und der Reihe nach in eine Tabelle geschrieben. Dann kann bei der Lösung eines Problems der Größe i auf die Lösung aller Probleme mit Größen $1, 2, ..., i-1$ zurückgegriffen werden, d. h. dass jedes Teilproblem höchstens einmal gelöst wird (was ja bei Divide-and-Conquer Algorithmen nicht immer der Fall war). Oft ergibt sich durch diese Tabulierung ein Effizienzgewinn. Da bei diesem Prinzip der Aufbau der Gesamtlösung aus Teillösungen im Vordergrund steht, ordnet man ihm auch das Attribut *bottom-up* zu.

Oft wird die dynamische Programmierung in Algorithmen zur Lösung von Optimierungsproblemen eingesetzt. Bei solchen Problemen gibt es mehrere Lösungen; je nach Anwendungsgebiet liegt eine partielle Ordnung vor, mit deren Hilfe die Güte von Lösungen miteinander verglichen werden kann; eine optimale Lösung ist eine minimale Lösung bzgl. dieser partiellen Ordnung.

Will man einen Algorithmus zur Lösung von Optimierungsproblemen nach dem Prinzip der dynamischen Programmierung aufbauen, dann muss eine Voraussetzung erfüllt sein (Bellmannsches Optimalitätsprinzip): Die optimale Lösung für ein Problem der Größe n muss sich zusammensetzen lassen aus den optimalen Lösungen von Problemen kleinerer Größe (z. B. wie beim Floyd-Warshall- und Aho-Algorithmus).

Als erstes geben wir allerdings ein Beispiel an, bei dem es nicht um Optimierung geht, sondern schlicht um das Ausrechnen eines Funktionswerts. Als zweites beschäftigen wir uns dann mit einem echten Optimierungsproblem.

14.2.1 Fibonacci

In Abschnitt 14.1.2 haben wir die Berechnung von KP nach dem Prinzip Divide-and-Conquer durchgeführt. Es ist leicht, ein effizienteres Verfahren anzugeben, welches nach dem Prinzip der dynamischen Programmierung funktioniert. Es beruht darauf, dass man mit der „Berechnung“ von $KP(0)$ und $KP(1)$ beginnt, $KP(n-2)$ und $KP(n-1)$ als Zwischenergebnisse speichert und dann für die Berechnung von $KP(n)$ benutzt.

$$\begin{aligned} iter(n) &= it(n,\ 0,\ 1) && (n \geq 1) \\ it(1,\ x,\ y) &= y \\ it(n+1,\ x,\ y) &= it(n,\ y,\ x+y) \end{aligned}$$

Es gilt für alle $0 \leq k \leq n-1$:

$$it(n-k,\ KP(k),\ KP(k+1)) \ =\ it(n,\ 0,\ 1) \qquad \text{(Behauptung)}$$

Diese Behauptung lässt sich durch einfache Induktion über k beweisen.

$k = 0$ **(Induktionsanfang)**

$$\begin{aligned} & it(n-k,\ KP(k),\ KP(k+1)) \\ =\ & it(n-0,\ KP(0),\ KP(1)) \\ =\ & it(n,\ 0,\ 1) \end{aligned}$$

$k \to k+1$ **(Induktionsschritt)**

$$\begin{aligned} & it(n-(k+1),\ KP(k+1),\ KP(k+2) \\ =\ & it(n-k-1,\ KP(k+1),\ KP(k+1)+KP(k)) && \text{(nach Def. } KP\text{)} \\ =\ & it(n-k,\ KP(k),\ KP(k+1)) && \text{(nach Definition } it \text{ rückwärts)} \\ =\ & it(n,\ 0,\ 1) && \text{(nach Induktionsvoraussetzung)} \end{aligned}$$

Somit gilt für jedes $n \geq 1$:

$$\begin{aligned} iter(n) &= it(n,\ 0,\ 1) \\ &= it(1,\ KP(n-1),\ KP(n)) && (\text{für } k = n-1) \\ &= KP(n)\ . \end{aligned}$$

Das iterative Verfahren lässt sich leicht in ein Programm übertragen.

```
 1 | int x, y, n, swap;
 2 |
 3 | scanf("%d", &n);
 4 | x = 0;
 5 | y = 1;
 6 | while (n > 1)
 7 | { swap = y;
 8 |   y = x + y;
 9 |   x = swap;
10 |   n = n-1;
11 | }
12 | printf("%d", y);
```

Die Zeitkomplexität dieses iterativen Algorithmus (Programms) ist linear, und die Platzkomplexität ist sogar konstant, da man nur fünf Speicherplätze braucht. Also ist das iterative Verfahren viel zeit- und platzeffizienter als das rekursive Verfahren, allerdings nicht mehr ganz so übersichtlich.

14.2.2 Matrizen-Kettenmultiplikation

Jetzt kommen wir zu einem echten Optimierungsproblem. Es geht um die Multiplikation $M_1 * M_2 * \ldots * M_n$ einer beliebigen Liste $M_1, M_2, \ldots, M_n$ von Matrizen über natürlichen Zahlen, deren Dimensionen zueinander passen, d. h. für jedes $1 \leq i \leq n$ habe M_i die Dimension (p_{i-1}, p_i).

Zunächst ist die Matrixmultiplikation eine binäre Operation. Wenn man die Matrizen M_i und M_{i+1} miteinander multipliziert, dann müssen $p_{i-1} \cdot p_i \cdot p_{i+1}$ elementare Multiplikationen zweier natürlicher Zahlen durchgeführt werden.

Wenn nun die Matrix $M_1 * M_2 * \ldots * M_n$ berechnet werden soll, dann muss also zuerst eine Klammerung gefunden werden, so dass immer zwei Matrizen miteinander multipliziert werden. Beispielsweise ergeben sich für $n = 4$ die folgenden fünf Klammerungsmöglichkeiten:

$$M_1 * (M_2 * (M_3 * M_4))$$

$$M_1 * ((M_2 * M_3) * M_4)$$
$$(M_1 * (M_2 * M_3)) * M_4$$
$$((M_1 * M_2) * M_3) * M_4$$
$$(M_1 * M_2) * (M_3 * M_4)$$

Da die Matrixmultiplikation assoziativ ist, spielt es keine Rolle, welche Klammerung wir der Berechnung zugrunde legen: es kommt immer dasselbe Ergebnis heraus. Allerdings spielt die Klammerung sehr wohl bei der Frage eine Rolle, wie *aufwendig* die Berechnung von $M_1 * M_2 * \ldots * M_n$ ist. Haben beispielsweise die drei Matrizen M_1, M_2, M_3 die Dimensionen $(5, 11), (11, 6), (6, 20)$, dann gilt

Klammerung	Aufwand
$(M_1 * M_2) * M_3$	$5 \cdot 11 \cdot 6 + 5 \cdot 6 \cdot 20 = 930$
$M_1 * (M_2 * M_3)$	$5 \cdot 11 \cdot 20 + 11 \cdot 6 \cdot 20 = 2420$

Es ist also klar, dass man hier zuerst M_1 und M_2 multiplizieren sollte und dann die Matrix $M_1 * M_2$ mit der Matrix M_3. Es liegt hier also das folgende Optimierungsproblem vor:

Gegeben: Matrizen $M_1, M_2, \ldots, M_n$ (repräsentiert durch deren jeweilige Dimensionen)
Gesucht: Klammerung von $M_1 * M_2 * \ldots * M_n$, so dass die Anzahl der durchzuführenden elementaren Multiplikationen minimal ist.

Offensichtlich ist das Bellmannsche-Optimalitätskriterium erfüllt, denn wenn man schon für Abschnitte $M_i, \ldots, M_k$ und $M_{k+1}, \ldots, M_j$ jeweils optimale Klammerungen gefunden hat, dann basiert die optimale Lösung für eine Liste von Matrizen, in der $M_i, \ldots, M_k$ und $M_{k+1}, \ldots, M_j$ auftreten, auf diesen optimalen Lösungen der Teilprobleme. Also können wir den Algorithmus zur Berechnung der optimalen Klammerung nach dem Prinzip der dynamischen Programmierung aufbauen.

Sei $m(i, j)$ die minimale Anzahl der elementaren Multiplikationen, die man braucht, um die Matrix $M_i * \ldots * M_j$ für $i \leq j$ zu berechnen, wobei natürlich $m(i, i) = 0$ ist. Wenn nun die optimale äußere Klammerung von $M_i, \ldots, M_j$ hinter der k-ten Matrix wäre, d. h. $(M_i * \ldots * M_k) * (M_{k+1} * \ldots * M_j)$, dann würde sich die Zahl $m(i, j)$ wie folgt berechnen lassen:

$$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j.$$

Das ergibt sich aus dem Optimalitätskriterium. Wenn man dieses k ermitteln muss, dann braucht man offensichtlich nur eine Minimumsbildung über alle möglichen Werte für k vorzunehmen:

$$m(i, j) = \begin{cases} 0 & \text{wenn } i = j \\ \min_{i \leq k < j}\{m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{wenn } i < j \end{cases}$$

Auf dieser Rekursionsformel basiert der folgende Algorithmus. Er benutzt eine Matrix $m[n][n]$ über integer, wobei allerdings nur die rechte obere Dreiecksmatrix belegt ist. (Beachte: In der zweiten `for`-Schleife muss die Reihenfolge der Paare (i, j) nicht festgelegt werden, sie ist nicht relevant. In unserem imperativen Programmbeispiel wird allerdings durch die Laufanweisung für die Indizes eine Reihenfolge vorgegeben.)

Sei `P` das Feld mit den Dimensionen $p_0, p_1, \ldots, p_n$ der Matrizen $M_1 \ldots M_n$.

```
1  for (i = 1; i <= n; i = i+1) { m[i][i] = 0 };
2
3  for (l = 1; l <= n-1; l = l+1)  /* gehe die Diagonalen der Reihe nach durch */
4    for (i = 1; i <= n; i = i+1)
5      for (j = 1; j <= n; j = j+1)
6        if (j-i == l)
7        { m[i][j] = MaxInteger;
8          for (k = i; k <= j-1; k = k+1)  /* berechne in m[i][j] das Minimum */
9          { q = m[i][k] + m[k+1][j] + P[i-1]* P[k] * P[j];
10           if (q < m[i][j])  m[i][j]=q;
11         }
12       }
```

Hinweis: Um die Übersichtlichkeit zu wahren, wurde die Zählung der Indizes (im Gegensatz zu C) bei 1 begonnen!

Die optimale Klammerung erhält man dadurch, dass man bei der Minimumsbildung sich jeweils das k merkt, für das der Minimumswert angenommen wird.

Beispiel 14.3. Seien fünf Matrizen M_1, M_2, M_3, M_4, M_5 gegeben. Dann berechnet der Algorithmus folgende Matrix m:

Matrix	Dimension
M_1	(5,11)
M_2	(11,6)
M_3	(6,20)
M_4	(20,8)
M_5	(8,9)

m:

$i \diagdown j$	1	2	3	4	5
1	0	330^{1}	930^{5}	1530^{8}	1890^{10}
2	-	0	1320^{2}	1488^{6}	1986^{9}
3	-	-	0	960^{3}	1392^{7}
4	-	-	-	0	1440^{4}
5	-	-	-	-	0

Die oberen Indizes an manchen Einträgen in der Matrix m geben eine mögliche Reihenfolge an, in der der Algorithmus die Einträge berechnet. Dabei werden die folgenden Minimumsbildungen durchgeführt (um die optimale Klammerung nachher konstruieren zu können, protokollieren wir jedes mal den Index k, so dass bei Aufteilung von $M_i, \ldots, M_j$ in $M_i, \ldots, M_k$ und $M_{k+1}, \ldots, M_j$ eine optimale Klammerung entsteht):

$$\begin{aligned} m[1][3] &= \min\{0 + 1320 + 5 \cdot 11 \cdot 20,\ \ 330 + 0 + 5 \cdot 6 \cdot 20\} = \min\{2420, 930\} = 930 \\ k &= 2 \end{aligned}$$

$$\begin{aligned} m[2][4] &= \min\{1320 + 0 + 11 \cdot 20 \cdot 8,\ \ 0 + 960 + 11 \cdot 6 \cdot 8\} = \min\{3080, 1488\} = 1488 \\ k &= 2 \end{aligned}$$

$$\begin{aligned} m[3][5] &= \min\{960 + 0 + 6 \cdot 8 \cdot 9,\ \ 0 + 1440 + 6 \cdot 20 \cdot 9\} = \min\{1392, 2520\} = 1392 \\ k &= 4 \end{aligned}$$

$$\begin{aligned} m[1][4] &= \min\{0 + 1488 + 5 \cdot 11 \cdot 8,\ \ 330 + 960 + 5 \cdot 6 \cdot 8,\ \ 930 + 0 + 5 \cdot 20 \cdot 8\} \\ &= \min\{1928, 1530, 1730\} = 1530 \\ k &= 2 \end{aligned}$$

$$\begin{aligned} m[2][5] &= \min\{0 + 1392 + 11 \cdot 6 \cdot 9,\ \ 1320 + 1440 + 11 \cdot 20 \cdot 9,\ \ 1488 + 0 + 11 \cdot 8 \cdot 9\} \\ &= \min\{1986, 4740, 2280\} = 1986 \\ k &= 2 \end{aligned}$$

$$\begin{aligned} m[1][5] &= \min\{0 + 1986 + 5 \cdot 11 \cdot 9,\ \ 330 + 1392 + 5 \cdot 6 \cdot 9,\ \ 930 + 1440 + 5 \cdot 20 \cdot 9,\ \ 1530 + 0 + 5 \cdot 8 \cdot 9\} \\ &= \min\{2481, 1992, 3270, 1890\} = 1890 \\ k &= 4 \end{aligned}$$

Verfolgt man jetzt die Folge der k's rückwärts, dann kann man daraus schrittweise die optimale Klammerung erzeugen:

$$M_1 * M_2 * M_3 * M_4 * M_5$$

bei $m[1][5]$ ist $k = 4$ (d. h. Trennung nach der Matrix M_4) $\rightarrow$

$$(M_1 * M_2 * M_3 * M_4) * M_5$$

bei $m[1][4]$ ist $k = 2$ $\rightarrow$

$$((M_1 * M_2) * (M_3 * M_4)) * M_5$$ □

Die Zeitkomplexität des Algorithmus liegt in der Klasse $O(n^3)$: Der Algorithmus enthält drei geschachtelte `for`-Schleifen, wobei in jeder der Schleifenrumpf höchstens n-mal durchlaufen wird.

In der Vorlesung *Formale Systeme* wird ein ganz ähnlicher Algorithmus vorgestellt, mit dessen Hilfe sich entscheiden lässt, ob ein vorgelegtes Wort w in einer kontextfreien Sprache ist. Der Algorithmus wurde nach seinen Erfindern Cocke, Younger und Kasami *CYK-Algorithmus* genannt.

14.3 Backtracking

Backtracking ist ein algorithmisches Prinzip, um systematisch nach einer (oder mehreren) Lösungen zu suchen. Für die Beschreibung dieser Methode nutzen wir das Konzept des abstrakten Reduktionssystems. Im Kapitel 14 werden wir uns mit diesem Formalismus genauer beschäftigen; hier nur ein Vorgriff:

Ein abstraktes Reduktionssystem ist ein Tupel $(D, \vdash)$, wobei D die Menge der Zwischenergebnisse und $\vdash \subseteq D \times D$ die Rechenvorschrift ist. Mit Hilfe der Rechenvorschrift kann ein Zwischenergebnis in das nächste umgewandelt werden. Gibt man ein Zwischenergebnis $d \in D$ vor, so repräsentiert eine Folge $d_0, d_1, \ldots, d_k$ mit $d = d_0$ und $(d_i, d_{i+1}) \in \vdash$ für alle $0 \leq i \leq k-1$ das „Ausrechnen von d“.

Für die folgenden Betrachtungen ist es sinnvoll, eine Teilmenge $D' \subseteq D$ als Menge der *Endergebnisse* oder *Lösungen* zu kennzeichnen.

Nun gilt meistens, dass es zu einem Zwischenergebnis $d \in D$ mehr als ein $d' \in D$ mit $d \vdash d'$ gibt. Wenn man alle Rechnungen, die von einem $d \in D$ ausgehen, auf einen Blick sehen möchte, so kann man diese Rechnungen im *Berechnungsbaum von d*, bezeichnet durch $T(d)$, zusammenfassen. Dies ist ein Baum, bei dem die Knoten mit Zwischenergebnissen beschriftet sind; insbesondere ist die Wurzel mit d beschriftet. Wenn ein Knoten x mit d beschriftet ist und $\{d' \mid d \vdash d'\} = \{d_1, \ldots, d_n\}$ für ein $n \geq 0$ die Menge der Folge-Zwischenergebnisse ist, dann hat x n Nachfolgerknoten, die mit $d_1, \ldots, d_n$ beschriftet sind. Natürlich muss vorher unter den Folge-Zwischenergebnissen eine totale Ordnung festgelegt werden.

Das Backtracking mit d als Ausgangspunkt ist nun die *Tiefensuche* durch $T(d)$ (siehe Abschnitt 12.3; allerdings unterscheidet sich die Tiefensuche des Kapitels 12.3 von *Backtracking* dadurch, dass bei der Tiefensuche der Graph vorliegt, während beim *Backtracking* der Baum – als spezieller Graph – während der Tiefensuche erzeugt wird). Dieser Suchlauf kann als rekursive Funktion `backtrack` beschrieben werden, welche einen formalen Parameter hat; dieser kann Knoten von $T(d)$ als Werte annehmen. Anschaulich gesprochen kann man also sagen, dass sich die Funktion `backtrack` immer an einem Knoten x von $T(d)$ befindet. Wenn x die Nachfolgerknoten $x_1, \ldots, x_n$ hat, so besteht der Rumpf von `backtrack` im wesentlichen aus einer `for`-Schleife, in der `backtrack` der Reihe nach auf $x_1, \ldots, x_n$ aufgerufen wird. Die Funktion startet an der Wurzel von $T(d)$ und wird so lange durchgeführt bis ein Endergebnis gefunden wird. Bei Bedarf kann der Suchlauf dort auch fortgesetzt werden, um weitere Endergebnisse zu finden.

Sicherlich kann es im Berechnungsbaum von d auch unendliche Äste geben; das ist dann der Fall, wenn $\vdash$ nicht terminierend ist. Deshalb ist es auch möglich, dass das Backtracking in einen unendlichen Ast läuft, während es sehr wohl eine erfolgreiche Rechnung gibt, da wir den Berechnungsbaum per Tiefensuche durchlaufen (siehe Abbildung 14.2).

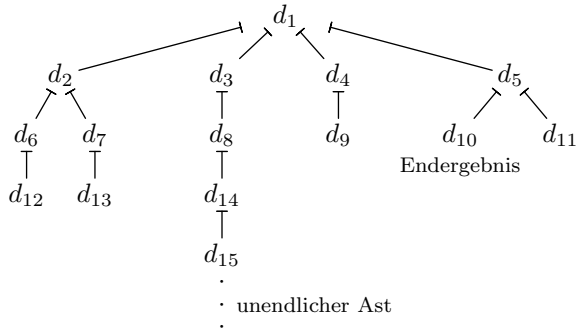


Abbildung 14.2: Berechnungsbaum.

Eine Anwendung des Backtrackings finden wir, wenn wir Lösungen berechnen wollen, wobei jede Lösung aus endlich vielen Komponenten besteht und jede Komponente nur endlich viele Werte annehmen kann.

Dann beginnt Backtracking mit der leeren Teillösung und versucht durch sukzessives Erweitern einer vorliegenden Teillösung schließlich eine Gesamtlösung zu berechnen. Wenn eine Teillösung nicht erweiterbar ist, so verwirft man die zuletzt getroffene Wahl eines Wertes und wählt einen alternativen Wert. Diese Methode lässt sich durch folgendes Programm in Pseudocode schematisch beschreiben.

Algorithmus 11 Backtracking

```
1  void  backtrack (Teillösung)
2  { if (Teillösung == Gesamtlösung)
3      gib Teillösung aus
4    else
5      for (jede Erweiterung der Teillösung)
6        if (Erweiterung zulässig)
7          backtrack (erweiterte Teillösung)
8  }
```

Beispiel 14.4. Wir wollen uns dieses Verfahren nun an einem ganz konkreten Beispiel anschauen, dem Vier-Damen-Problem: ein Brett bestehend aus vier mal vier quadratisch angeordneten Feldern soll so mit vier Damen besetzt werden, dass diese sich nicht gegenseitig bedrohen (im Schach bedrohen sich zwei Damen, wenn sie auf der gleichen Spalte, Zeile oder diagonalen Linie positioniert sind). Da keine zwei Damen in der gleichen Spalte sein dürfen, muss sich in jeder Spalte genau eine Dame befinden. Eine solche Positionierung der vier Damen lässt sich als 4-Tupel (a_1, a_2, a_3, a_4) aufschreiben, wobei $a_i \in \{1, 2, 3, 4\}$ die Nummer der Zeile angibt, in der sich die Dame in der i-ten Spalte befindet. Ein Zwischenergebnis ist dann eine Konfiguration, bei der sich noch nicht alle Damen auf dem Brett befinden; eine solches Zwischenergebnis schreiben wir als 1-, 2-, oder 3-Tupel auf, wobei z. B. ein 2-Tupel bedeutet, dass erst die beiden ersten Spalten mit Damen besetzt wurden.

Wir können mit Hilfe des Backtrackings systematisch nach einer Lösung des Vier-Damen-Problems suchen, indem wir zuerst eine Dame in die erste Spalte setzen und dann versuchen, eine weitere Dame in die nächste Spalte so zu setzen, dass diese sich nicht mit der vorher gesetzten Dame bedroht, usw. Gibt es irgendwann keine Möglichkeit eine weitere Dame zu setzen, so machen wir den letzten Schritt rückgängig und versuchen, an dieser Stelle nach einer weiteren Möglichkeit zu suchen. Dabei entsteht der Berechnungsbaum in Abbildung 14.3.

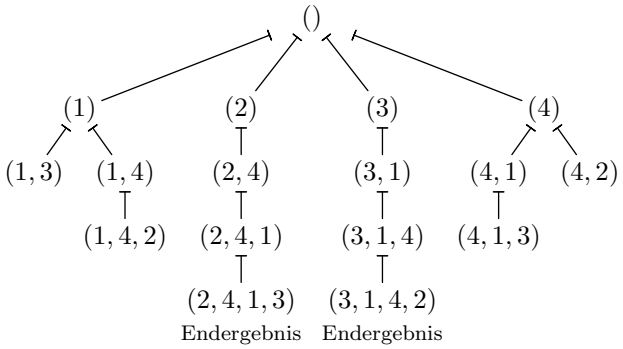


Abbildung 14.3: Berechnungsbaum für das Vier-Damen-Problem.

Da in dem Berechnungsbaum des Vier-Damen-Problems in jedem Ableitungsschritt ein neues Element an eine Liste angehängt wird, können wir uns den Aufwand sparen, jedes mal die vollständige Liste aufzuschreiben. Wenn wir stattdessen nur das neue Element kennzeichnen, so nennen wir einen solchen Baum einen *optimierten Berechnungsbaum* (siehe Abbildung 14.4). □

Beispiel 14.5. Eine Erweiterung des Vier-Damen-Problems ist das bekannte *Eight-Queens-Problem*, welches von dem folgenden Programm gelöst wird. □

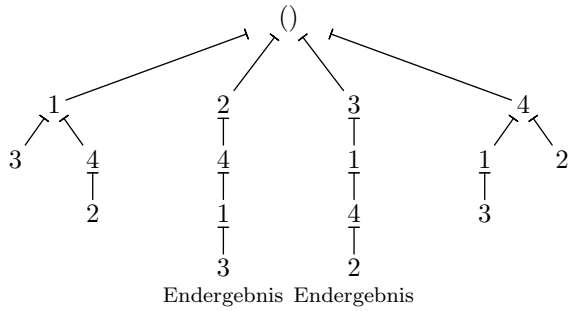


Abbildung 14.4: Optimierter Berechnungsbaum für das Vier-Damen-Problem.

```
 1  /* acht-Damen-Problem */
 2  #include <stdio.h>
 3  #include <math.h>
 4
 5  short a[8];
 6
 7  void drucke()
 8  { short i;
 9    for (i=0; i<=7;i=i+1) printf("%2d", a[i]);
10    printf("\n");
11  }
12
13  int konsistent(int t)
14  { short j;
15
16    for (j = 0; j < t; j = j+1)
17    { if (a[j] == a[t]) return 0;          /* prüft Bedrohung auf der Zeile a[t] */
18      if (abs(a[j] - a[t]) == (t-j)) return 0; /* prüft Bedrohung auf den beiden */
19    }                                          /* Diagonalen durch a[t]          */
20    return 1;
21  }
22
23  void suche(int t)
24  { short i;
25
26    if (t == 8) drucke();
27    else
28      for (i = 0; i <= 7; i = i+1)
29      { a[t] = i;
30        if (konsistent(t)) suche(t+1);
31      }
32  }
33
34  int main ()
35  { suche(0);
36    return 0;
37  }
```

Teil III

Weitere Programmierkonzepte

15 Funktionale Programmierung

Bisher haben wir in der Vorlesung nur ein Programmierparadigma kennengelernt, nämlich das Paradigma der imperativen Programmierung. Dieses Paradigma ist durch das *Zustandskonzept* gekennzeichnet:

- Benutzung von Programmvariablen zur Bezeichnung von Speicherplätzen
- Speicherung von Werten in Speicherplätzen
- Veränderung des „Wertes einer Variablen“ (genauer: des Inhalts des durch die Variable bezeichneten Speicherplatzes) durch Assignment-Anweisungen

Ein imperatives Programm beschreibt, *wie* der Zustand (d. h. die Abbildung von Speicherplätzen auf Werte) verändert werden muss, um zum Ergebnis zu kommen.

Als Beispiel für ein imperatives Programm sei hier die Berechnung der Summe der ersten n Quadratzahlen wiederholt (mit C-Syntax).

```
1  /* Summation */
2  #include <stdio.h>
3
4  int main()
5  { int i, n, s;
6
7    scanf("%d", &n);
8    i = 1;
9    s = 0;
10   while (i <= n)
11   {
12     s = s+i*i;
13     i = i+1;
14   }
15   printf("%d", s);
16   return 0;
17 }
```

Damit spiegelt sich in der imperativen Programmierung das Konzept des von-Neumann Rechners wider, wobei den Programmvariablen die einzelnen Speicherplätze des Rechners entsprechen und die Zuweisung eines Wertes an eine Programmvariable mit dem Speichern eines Wertes in einem Speicherplatz korrespondiert. Alle Programmiersprachen, welche dem Paradigma der imperativen Programmierung folgen, haben diese Charakteristik. Sie werden auch kurz *imperative Programmiersprachen* genannt.

In diesem Teil der Vorlesung wollen wir ein anderes Programmierparadigma kennenlernen: die funktionale Programmierung. Im Unterschied zur imperativen Programmierung gibt es in der funktionalen Programmierung keinen Zustandsraum; für eine Variable i bezeichnet jedes Vorkommen von i denselben Wert.

Formulieren wir die Aufgabe „Summe der ersten n Quadratzahlen“ jetzt als funktionales Programm (mit Haskell-Syntax):

```
1  module Main where
2
3  main :: IO ()
4  main = do
5    n <- readLn
6    print (sumSquares n)
7
8  sumSquares :: Int -> Int
9  sumSquares 0 = 0
10 sumSquares i = i * i + sumSquares (i - 1)
```

Hier bezeichnet `i` keinen Speicherplatz, sondern eine Variable im mathematischen Sinn. Diese kann zwar beliebige Werte aus der Menge `Int` annehmen; aber jedes Vorkommen von `i` in einer Gleichung bezeichnet denselben Wert – wie in mathematischen Formeln. Im funktionalen Programm wird auch nicht mehr angegeben, *wie* der Wert (etwa von `sumSquare 2` berechnet wird (z. B. fehlt die akkumulierende Variable `s` aus dem imperativen Programm völlig); vielmehr wird direkt die gewünschte Funktion spezifiziert. In diesem Sinne ist die funktionale Programmierung abstrakter als die imperative Programmierung; sie abstrahiert vom Zustandsraum und von der Spezifikation der *Berechnung* von Werten, stattdessen spezifiziert sie nur *die Werte selbst*. Noch einmal in anderen Worten:

- In *imperativen Programmen* wird die Berechnung von Werten programmiert („wie berechnet man ...“).
- In *funktionalen Programmen* werden Funktionen programmiert („was berechnet man ...“).

Natürlich legt das gezeigte funktionale Programm nahe, wie der Ausdruck `sumSquare 2` berechnet werden kann: nämlich durch standardmäßige Anwendung der Funktion auf das Argument:

```
  sumSquare 2
= 2 * 2 + sumSquare 1
= 2 * 2 + (1 * 1 + sumSquare 0)
= 2 * 2 + (1 * 1 + 0)
= 5
```

Abschließend zeigen wir noch ein Haskell-Programm, welches einen Eindruck der Kompaktheit der funktionalen Programmierung vermittelt. Das Programm liest mehrere, durch Leerzeichen getrennte Ganzzahlen von der Standardeingabe ein und gibt sie sortiert wieder aus. Zur Sortierung wird der QuickSort-Algorithmus verwendet.

Beispiel 15.1.

```
 1 module Main where
 2
 3 main :: IO ()
 4 main = do
 5   cs <- getContents
 6   putStrLn $ unwords $ map show
 7            $ quicksort $ map read
 8            $ words cs
 9
10 quicksort :: [Int] -> [Int]
11 quicksort [] = []
12 quicksort (x : xs)
13   =  quicksort (filter (x > ) xs)
14   ++ [x]
15   ++ quicksort (filter (x <=) xs)
```

Alle hier nicht explizit definierten Funktionen und Operationen sind in Haskell bereits vordefiniert. Dabei bezeichnen `[]` die leere Liste, `:` eine zweistellige Operation, die das erste Argument vor den Anfang des zweiten Arguments (welches als Liste aufgefasst wird) setzt, `++` eine zweistellige Operation, die zwei Listen l_1 und l_2 als Argumente nimmt und die Konkatenation von l_1 und l_2 als Ergebnis liefert. Die Operation `$` ist rechtsassoziativ und es gilt `f $ x = f x`. Sie dient hier der Einsparung von Klammern, denn `f $ g $ x = f $ (g $ x) = f (g x)`. Die Funktion `filter` verlangt als Argumente ein Prädikat und eine Liste. Als Ergebnis liefert die Funktion eine Liste, in der nur noch die Elemente der ursprünglichen Liste enthalten sind, für die das Prädikat `True` liefert. □

Man erkennt deutlich, dass dieses funktionale Programm nur *die Idee* des QuickSort-Algorithmus darlegt; Berechnungsdetails werden nicht angegeben, vielmehr schreibt man die zu berechnende Funktion *selbst* als Programm auf.

Abschließend ist hier als Vergleich eine bereits aus der Vorlesung Algorithmen und Datenstrukturen bekannte imperative Umsetzung des Quicksort-Algorithmus gegeben (*C*-Programm).

```
1  void quicksort(int a[], int L, int R)
2  { int i, j, w, x, k;
3
4    i = L; j = R; k = (L+R) / 2;
5    x = a[k];
6    do
7    { while (a[i] < x) i = i+1;
8      while (a[j] > x) j = j-1;
9      if (i <= j)
10     { w = a[i];
11       a[i] = a[j];
12       a[j] = w;
13       i = i+1; j = j-1;
14     }
15   } while (i <= j);
16   if (L < j) quicksort(a, L, j);
17   if (R > i) quicksort(a, i, R);
18 }
```

Hintergrund Funktionale Programmiersprachen finden Anwendung vor allem dort, wo komplexe Zusammenhänge zu erfassen sind und besonderes Augenmerk auf der Verlässlichkeit und Wiederverwendbarkeit von Software liegt. Durch modulares, prägnantes Design auf einem hohen Abstraktionsniveau entstehen mit relativ wenig Aufwand Programme, die oftmals eine weitaus größere und direktere Nähe zu den umzusetzenden Problemstellungen und Algorithmen aufweisen als bei Umsetzung in einer imperativen Programmiersprache. Solche „ausführbaren Spezifikationen“ erlauben frühzeitiges Testen, gegebenenfalls auch Vergleich verschiedener Lösungsansätze, und erleichtern die Anpassung an sich ändernde Anforderungen. Sie dienen als Prototypen, die entweder die Umsetzung in einer konventionelleren Programmierumgebung leiten oder selbst zum Endprodukt weiterentwickelt werden können. Durch die Nähe zum mathematischen Funktionskonzept ermöglichen funktionale Sprachen die formale Zusicherung von qualitativen Programmeigenschaften, wodurch das Vertrauen in die entwickelten Systeme erhöht wird.

Lange dem akademischen Bereich verhaftet, finden funktionale Programmiersprachen und -techniken auf Grund dieser Aspekte zunehmend Verwendung im kommerziellen Umfeld. Zahlreiche Beispiele hierfür finden sich unter `http://cufp.galois.com`. Relevante Geschäftsfelder sind etwa die Telekommunikationsbranche (Ericsson mit der eigens entwickelten funktionalen Sprache Erlang), der Bankenbereich (z. B. Credit Suisse und Jane Street Capital), Anwender im Bereich des Hardwareentwurfs (Intel und Bluespec), sicherheits- und informationskritische Anwendungen (Galois und Aetion), sowie Entwicklungen im Betriebssystembereich (Microsoft und Linspire). Neben der direkten Verwendung funktionaler Sprachen finden auch einzelne ihnen entstammende Konzepte und Methoden selektiv Eingang in andere Umgebungen. So führte etwa die Erkenntnis der einfachen Parallelisierbarkeit funktionaler Programme zur Entwicklung der daran angelehnten MapReduce-Technologie zur Verwaltung und Analyse immenser Datenmengen durch Google. Ein ähnlicher Trend ist, dass imperative und objektorientierte Programmiersprachen um wichtige Features vormals nur funktionaler Sprachen erweitert werden. Dies betrifft zum Beispiel die Einführung von Lambda-Ausdrücken und der Abfragesprache LINQ in Visual Basic und C#, oder von Generics in Java.

In den nächsten Kapiteln werden wir einige Phänomene der funktionalen Programmierung behandeln. Dabei besprechen wir anhand der konkreten Programmiersprache Haskell die verschiedenen „Features“ dieses Programmierparadigmas (Abschnitte 15.1–15.7). Abschließend behandeln wir die mathematische Grundlage der funktionalen Programmierung: den λ-Kalkül (Abschnitt 15.8), in dem insbesondere formalisiert wird, wie Berechnungen aussehen.

15.1 Einführung in Haskell

Ein Haskell-Programm besteht im Allgemeinen aus mehreren Quelltextdateien, sogenannten Modulen. Ein Modul besteht aus einer Sequenz von Deklarationen. Deklarationen definieren unter anderem Werte und Datentypen. Auch Funktionen werden als Werte betrachtet.

Namen für Variablen (dazu zählen auch Funktionsnamen) und Typvariablen beginnen mit einem Kleinbuchstaben oder einem Unterstrich. Alle anderen Namen beginnen mit einem Großbuchstaben. Die restlichen Zeichen eines Namens können Buchstaben, Ziffern, Unterstriche oder Hochkommata sein.

Daneben gibt es Operatorsymbole. Ein Operatorsymbol besteht ausschließlich aus Symbolen. Operatorsymbole für Variablen (Funktionen, Operatoren) beginnen *nicht* mit einem Doppelpunkt.

In jedem Modul stehen (wenn nicht explizit ausgeschlossen) die Definitionen des Moduls `Prelude` zur Verfügung, welches mit jedem Haskell-Compiler ausgeliefert wird. Dazu zählen z. B. Funktionen wie `length`, `map` und `foldr`, Operatoren wie `+`, `-`, `*`, `/`, `++` und `.`, und Typen wie `Int`, `Bool`, `Char`, `String` und `Float`.

Jedes Haskell-Programm muss ein Modul namens `Main` definieren. Innerhalb dieses Moduls muss die Funktion `main` definiert werden; sie muss vom Typ `IO ()` sein. Diese Funktion wird bei Ausführung des Haskell-Programms ausgewertet.

Beispiel 15.1 zeigt ein vollständiges Modul. Dabei leitet Zeile 1 das Modul ein und die Zeilen 3 und 9 deklarieren die Typen der jeweils im Anschluss definierten Funktionen.

15.1.1 Einführende Beispiele

Als Einstieg zeigen wir sechs einfache, selbsterklärende Funktionsdeklarationen.

Beispiel 15.2. Die Funktion `square` nimmt ein Argument und gibt das Quadrat des Arguments zurück. Die Funktion `cube` gibt die Kubikzahl ihres Arguments zurück.

```
square :: Int -> Int
square x = x * x

cube :: Int -> Int
cube x = x * square x
```
□

Beispiel 15.3. Die Funktion `allEqual` testet, ob drei Zahlen gleich sind und gibt entsprechend `True` oder `False` zurück.

```
allEqual :: Int -> Int -> Int -> Bool
allEqual x y z = (x == y) && (x == z)
```
□

Beispiel 15.4. Die Funktionen `max` und `max'` bestimmen beide das Maximum zweier Zahlen.

```
max :: Int -> Int -> Int
max x y
  | x >= y    = x
  | otherwise = y

max' :: Int -> Int -> Int
max' x y = if x >= y then x else y
```
□

Beispiel 15.5. Die Funktion `gcd` berechnet den größten gemeinsamen Teiler von zwei Zahlen.

```
gcd :: Int -> Int -> Int
gcd x y
  | x == y    = x
  | y > x     = gcd y x
  | otherwise = gcd (x - y) y
```
□

Beispiel 15.6. Die Funktion `sumEvens` summiert alle geraden Zahlen von `0` bis zu einer gegebenen Zahl (in umgekehrter Reihenfolge).

```
sumEvens 0             = 0
sumEvens n | even n    = sumEvens (n - 2) + n
           | otherwise = sumEvens (n - 1)
```

Die Funktion `even` aus dem Modul `Prelude` prüft, ob eine Zahl gerade ist. Die Konstante `otherwise` aus dem Modul `Prelude` ist definiert als `True`; sie macht den Code hier lesbarer. □

Folgende Besonderheiten gelten bei Haskell-Programmen:

- Kommentare werden durch `--` eingeleitet. Sie reichen bis zum Ende einer Zeile.
- Beliebig lange Kommentare können an beliebiger Stelle mit `{-` eingeleitet und mit `-}` abgeschlossen werden. Diese Kommentare können sich über mehrere Zeilen erstrecken und dürfen verschachtelt werden.
- Alle Funktionen in Haskell sind einstellig. Also bezeichnet z. B. `max` aus Beispiel 15.4 eine Funktion, die ein Argument vom Typ `Int` nimmt und als Ergebnis eine Funktion liefert, die wiederum ein Argument vom Typ `Int` nimmt und (schließlich) eine Zahl liefert. Also ist der Funktionspfeil immer rechtsassoziativ.

  Konvention: Funktionen eines Types der Form $\texttt{a}_1\texttt{ -> a}_2\texttt{ -> ... -> a}_n\texttt{ -> b}$ bezeichnen wir vereinfacht als *n-stellig*, wenn `b` kein Funktionstyp ist. Somit nennen wir die Funktion `max` aus Beispiel 15.4 2-stellig. Wir nutzen diese Konvention, um über die Argumente einer Funktion einzeln reden zu können. Aber im Allgemeinen sind Funktionen in Haskell einstellig; siehe Abschnitt 15.3.2 für eine ausführliche Erklärung.
- Haskell-Programme werden üblicherweise durch verschiedene Einrückungen der Programmzeilen strukturiert. Dabei interpretiert Haskell die Einrückungen wie folgt:
  - Hinter den Schlüsselworten **`where`**, **`let`**, **`do`** und **`of`** beginnt jeweils ein neuer Block.
  - Ist eine Zeile weiter eingerückt als die vorhergehende Zeile, setzt diese Zeile die vorhergehende fort.
  - Ist eine Zeile genauso weit eingerückt wie eine ihrer vorangegangenen Zeilen, beginnt ein neues Element im entsprechenden Block.
  - Ist eine Zeile kürzer eingerückt als vorherige Zeilen, werden die entsprechend weiter eingerückten Blöcke beendet.

  Diese Interpretation der Einrückungen wird „layout rule“ oder „off-side rule“ genannt[1]. Haskell unterstützt, ähnlich wie in der Programmiersprache C, auch die explizite Programmstrukturierung mittels geschweifter Klammern und Semikolons; die Einrückungen werden in diesem Fall ignoriert. In der Praxis ist diese explizite Strukturierung allerdings unüblich. Das folgende Beispiel zeigt zwei äquivalente Programme; das linke ist mit Hilfe von Einrückungen, das rechte mittels geschweifter Klammern und Semikolons strukturiert.

```
 1 module Main where
 2 f :: Int -> Int
 3 f 0 = 0
 4 f n = let inc, dec :: Int -> Int
 5           inc n = n + 1
 6           dec n = n - 1
 7       in if even n
 8          then dec n
 9          else inc n `div` 2
10
```

```
 1 module Main where
 2 { f :: Int -> Int
 3 ; f 0 = 0
 4 ; f n = let { inc, dec :: Int -> Int
 5             ; inc n = n + 1
 6             ; dec n = n - 1
 7       } in if even n
 8            then dec n
 9            else inc n `div` 2
10 }
```

15.1.2 Funktionsdefinitionen und Berechnung

In Abschnitt 15.1.1 haben wir einige Funktionen programmiert. Offensichtlich gibt es verschiedene Möglichkeiten, Funktionen zu definieren (programmieren, spezifizieren).

- Die Funktionen `square`, `cube`, `allEqual` und `max'` wurden durch eine *einzige Gleichung* spezifiziert (simple equation).
- Die Funktionen `max` und `gcd` wurden durch *Fallunterscheidung* (conditional equation) mit Hilfe so genannter *guards* definiert. Ferner erkennen wir, dass die Funktion `gcd` rekursiv definiert wurde.

[1] `http://www.haskell.org/onlinereport/haskell2010/haskellch2.html#x7-210002.7`

- Die Funktion `max'` wurde durch eine Gleichung spezifiziert, bei der auf der rechten Seite das built-in Konstrukt if-then-else benutzt wurde, um die Fallunterscheidung zu realisieren.
- Die Funktionsdefinition von `sumEvens` nutzt *guards* und *pattern matching*. Mittels pattern matching kann man prüfen, ob Funktionsargumente einer bestimmten, vorgegebenen Form entsprechen.

Im Allgemeinen besteht die Definition einer Funktion `f` aus beliebig vielen aufeinander folgenden Blöcken von jeweils einer der folgenden Formen (die Zeilenumbrüche gehören nicht zur Syntax; sie dienen hier nur der besseren Übersicht):

```
f p1 p2 ... pn = e
```

oder

```
f p1 p2 ... pn
  | g1 = e1
  | g2 = e2
  ...
  | gk = ek
```

Dabei gilt folgendes:

- `p1`, `p2`, ..., `pn` sind *patterns*. Ein *pattern* ist ein Variablenbezeichner oder ein Literal (z. B. eine Zahl, ein Zeichen oder eine Zeichenkette). Mit der Einführung von algebraischen Datentypen (siehe Abschnitt 15.2.3) werden wir noch weitere Arten von patterns kennenlernen.
- `g1`, `g2`, ..., `gk` sind *guards*. Eine Form von *guards* sind Boolesche Ausdrücke (*boolean guards*). Es gibt auch noch andere Formen von guards[2], auf die hier nicht weiter eingegangen werden soll.
- `e1`, `e2`, ..., `ek`, `e` sind Ausdrücke vom Ergebnistyp der Funktion `f`.

Die erste Form ist äquivalent zu folgender Formulierung:

```
f p1 p2 ... pn | True = e
```

Zur Definition der Auswertung einer Funktionsapplikation reicht es also, nur Funktionsdefinitionen zu betrachten, die aus mehreren Blöcken in der zweiten Form bestehen. Bei der Auswertung wird zunächst geprüft, ob die Argumente zu den patterns des ersten Blocks passen; diesen Vorgang nennt man auch *pattern matching*. Passen die patterns, werden der Reihe nach die guards ausgewertet, bis ein guard `True` ergibt; der Wert des Ausdruck rechts vom „`=`“ ist dann der Wert der Funktionsapplikation. Passt ein pattern nicht oder ergeben alle guards `False`, wird mit dem nächsten Block genauso fortgefahren. Führt kein Block zum Erfolg, ist der Wert der Funktion für die gegebenen Argumente undefiniert und Haskell löst einen Fehler aus.

Wenn nun ein Ausdruck vorgegeben ist, dann lässt er sich auf ganz natürliche Weise ausrechnen; die Berechnung ist eine Folge von Ausdrücken („Zwischenergebnissen“), z. B.:

```
  sumEvens (1 + 2)          -- (0 == 1 + 2)   =   (0 == 3)   =   False
                            -- even 3   =   False
                            -- otherwise   =   True
= sumEvens (3 - 1)          -- (0 == 3 - 1)   =   (0 == 2)   =   False
                            -- even 2   =   True
= sumEvens (2 - 2) + 2      -- (0 == 2 - 2)   =   (0 == 0)   =   True
= 0 + 2
= 2
```

Wie man an den Kommentaren sieht, sind ggf. Zwischenrechnungen notwendig, um zur richtigen Definitionszeile zu gelangen.

Lokale Definitionen mit `where`-Klauseln: Zur besseren Strukturierung von Funktionsdefinitionen können sogenannte *lokale Definitionen* verwendet werden. Zum Beispiel:

[2] `http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-830004.4.3`

```
maxSquare n m
  | sqN > sqM = sqN
  | otherwise = sqM
  where
    sqN = square n
    sqM = square m
    square :: Int -> Int
    square z = z * z
```

In dieser Definition werden lokal drei Objekte definiert: die Werte `sqN` und `sqM` und die Funktion `square`. Diese lokalen Definitionen sind nur im Bereich der Funktionsdefinition `maxSquare` bekannt, deshalb heißen sie auch lokal. Lokale Definitionen (wie etwa die von `square`) können auch *vor* ihrem Auftreten (etwa in der Zeile `sqM = square m`) benutzt werden.

Im allgemeinen Fall sieht ein Block einer Definition mit lokalen Definitionen wie folgt aus (die Zeilenumbrüche gehören nicht zur Syntax; sie dienen hier nur der besseren Übersicht):

```
f p1 p2 ... pn = e
  where
    list
```

oder

```
f p1 p2 ... pn
  | g1 = e1
  | g2 = e2
  ...
  | gk = ek
  where
    list
```

wobei `list` eine Liste von Funktionsdefinitionen oder Konstantendefinitionen ist, deren Namen in den `gj` und `ej` und in `e` auftreten dürfen.

Schauen wir uns nun die Berechnung des Ausdrucks `maxSquare 4 (cube 2)` an.

```
  maxSquare 4 (cube 2)    -- (sqN > sqM)
                          -- where
                          -- sqN = square 4
                          --     = 4 * 4
                          --     = 16
                          -- (16 > sqM)
                          -- where
                          -- sqM = square (cube 2)
                          --     = square (2 * square 2)
                          --     = square (2 * 2 * 2)
                          --     ...
                          --     = 64
                          -- (16 > 64) = False
= sqM
= 64
```

Hier sehen wir, dass Nebenrechnungen geschachtelt sein können. Auch kann man erkennen, dass `sqM` nur *einmal* ausgerechnet werden muss.

Lokale Definitionen mit `let`-Ausdrücken: Eine weitere Möglichkeit lokale Definitionen zu notieren sind `let`-Ausdrücke. Sie haben folgende Syntax:

```
let list in ausdruck
```

Dabei ist `list` eine Liste von Funktions- oder Wertdefinitionen, die in `ausdruck` verwendet werden dürfen. Da **`let`**-Ausdrücke, wie der Name sagt, Ausdrücke sind, können sie im Gegensatz zu **`where`**-Klauseln innerhalb anderer Ausdrücke verwendet werden, z. B.:

```
pythagoras x y = sqrt (let square z = z * z in square x + square y)
```

Dabei ist `sqrt` eine vordefinierte Funktion, die die Quadratwurzel einer Zahl berechnet.

Auch in **`let`**-Ausdrücken können mehrere Funktionen oder Werte definiert werden, z. B.:

```
maxSquare n m = let square z = z * z
                    sqN = square n
                    sqM = square m
                in if sqN > sqM then sqN else sqM
```

Gültigkeitsbereiche von Variablenbezeichnern: Das folgende unvollständige Beispiel soll die Gültigkeitsbereiche von Variablenbezeichnern veranschaulichen. In Kommentaren werden die an der jeweiligen Stelle gültigen Bezeichner angegeben.

```
f ... = ... {- f, g -}

g a = ... {- f, g, a -} ... (let d = ... {- f, g, a, d -} in ... {- f, g, a, d -})
g b
  | ... {- f, g, h ,b -} = ... {- f, g, h, b -}
  | ... {- f, g, h, b -} = ... {- f, g, h, b -} ...
                           (let e = ... {- f, g, h, b, e -} in ... {- f, g, h, b, e -})
  where
    h c = ... {- f, g, h, b, c -}
```

Zu beachten ist, dass Variablenbezeichner lokal neu belegt werden können. Man sagt, eine lokale Definition *überdeckt* die ursprüngliche Definition. Im folgenden Beispiel ist in Kommentaren angegeben, wofür ein Bezeichner an der jeweiligen Stelle steht.

```
x = 1

x' = x {- x und x' bezeichnen den Wert 1 -}

f x = x {- x bezeichnet das Argument von f, nicht den Wert 1 -}

g x {- x bezeichnet das Argument von g -}
  = x {- x bezeichnet den Wert 2 -}
  + (let x = 3 in x {- x bezeichnet den Wert 3 -})
  where x = 2
```

Die Funktion `g` liefert also unabhängig von ihrem Argument den Wert `5`, obwohl der Anfang der Funktionsdefinition `g x = x ...` etwas anderes suggeriert. Deshalb sollten Überdeckungen im allgemeinen vermieden werden, um die Übersicht nicht zu beeinträchtigen.

15.2 Datentypen

15.2.1 Basistypen

In Haskell stehen die folgenden Basistypen mit Operationen aus dem Modul `Prelude` zur Verfügung:

Menge der ganzen Zahlen `Int`:

Beispiele: `12`, `0`, `-34562`
Operationen: `+`, `*`, `^` („zur Potenz“), `-`, `div`, `mod`, `abs`, `negate` („Umkehrung des Vorzeichens“).
Boolesche Vergleiche: `==`, `>`, `>=`, ...

Boolesche Werte `Bool`:

Die Werte: `True`, `False`
Operationen: `&&` (Konjunktion), `||` (Disjunktion), `not` (Negation)

Zeichen (character) `Char` und Zeichenreihen (Strings) `String`:

Beispiel für Zeichen: `'A'`, `'a'`, `'8'`

Es gibt eine Standardcodierung von Zeichen als Zahlen (Unicode) mit den entsprechenden Umwandlungsfunktionen:

```
toEnum   :: Int  -> Char
fromEnum :: Char -> Int
```

Zeichen können mit dem Operator `>` verglichen werden. Dabei basiert der Vergleich auf dem Vergleich der jeweiligen Ordnungszahlen im Unicode-Standard.

Beispiele für Zeichenreihen: `"hallo"`, `"ja, so ist das"`, `"654nnein"`, `"0.434"`

Operation auf Zeichenreihen: `++` (Konkatenation)

Beispiel: `"Guten " ++ "Tag"` = `"Guten Tag"`

Gleitkommazahlen (Floating Point) `Float`:

Beispiele: `0.3215`, `-23.56`, `451.0`
Operationen: (entnommen aus S. Thompson, Haskell – The Craft of Functional Programming, Addison-Wesley, Seite 47)

Zeichen für die Funktion	Typ der Funktion	Beschreibung
`+`, `*`, `-`	`Float -> Float -> Float`	Addition, Multiplikation, Subtraktion
`/`	`Float -> Float -> Float`	Division
`**`	`Float -> Float -> Float`	Exponentiation: x ** $y = x^y$
`==`, `/=`, `<=`, ...	`Float -> Float -> Bool`	(Un-)Gleichheit, Vergleichsoperationen
`abs`	`Float -> Float`	absoluter Wert
`sin`, `cos`, ...	`Float -> Float`	trigonometrische Funktionen
`ceiling`, `floor`, `round`	`Float -> Int`	Umwandlung einer Gleitk.zahl (Aufrunden, Abrunden, Runden)
`fromInt`	`Int -> Float`	Umwandl. ganze Zahl in eine Gleitk.zahl
`read`	`String -> Float`	Umwandl. Zeichenreihe in eine Gleitk.zahl
`show`	`Float -> String`	Umwandl. Gleitk.zahl in eine Zeichenreihe
`signum`	`Float -> Int`	-1, 0 oder 1 wenn die Zahl negativ, gleich 0 bzw. positiv ist
...		

Die Typen dieser Funktionen sind eigentlich allgemeiner als hier angegeben. Die allgemeinen Formen kann man jedoch erst verstehen, wenn man mit Polymorphie (Abschnitt 15.4) und Typklassen (Abschnitt 15.5) vertraut ist. Deshalb wurde hier auf die Angabe der allgemeinen Typen verzichtet.

15.2.2 Komposite Typen

Genau so wie in der imperativen Programmierung möchte man auch in der funktionalen Programmierung Daten strukturieren können.

Tupel `(a1, a2, ..., an)`:

Ein Beispiel für einen Tupeltyp ist `(Int, Int)`, welcher die Menge aller Paare von ganzen Zahlen als Wertebereich hat.

Es ist in Haskell möglich, Typdefinitionen aufzuschreiben; damit wird ein Alias für einen Typ definiert. Dieses Alias muss mit einem Großbuchstaben beginnen.

Beispiel 15.7. Im folgenden Programm sieht man, wie man mehrstellige Funktionen mit Hilfe von Tupeltypen programmieren kann.

```
type Triple = (Int, Int, Int)

addThree :: Triple -> Int
addThree (first, second, third) = first + second + third
```
□

Beispiel 15.8. Ein anderes Beispiel zeigt das Umklammern von Ausdrücken:

```
shift :: ((Int, Int), Int) -> (Int, (Int, Int))
shift ((x, y), z) = (x, (y, z))
```
□

Listen `[a]`:

Wenn `a` ein Typ ist, dann bezeichnet `[a]` den Typ „Liste mit Elementen vom Typ `a`“.

Beispiel 15.9. Einige Listen verschiedener Typen:

```
[1, 2, 3, 4]      :: [Int]
[True]            :: [Bool]
[]                :: [Int]
[[2, 3], [], [4]] :: [[Int]]
```
□

Für eine nicht leere Liste `[x1, x2, ..., xn]` wird `x1` als Kopf (head) der Liste und `[x2, ..., xn]` als die Restliste (tail) bezeichnet. Man schreibt die Liste dann auch wie folgt auf: `x1 : [x2, ..., xn]`; das Symbol `:` wird cons-Operator genannt und auch durch `cons` bezeichnet. Der cons-Operator ist rechtsassoziativ definiert, d. h. `x1 : x2 : xs = x1 : (x2 : xs)`. Beispielsweise ist der Kopf der Liste `[2, 5, 1]` die Zahl `2`, die Restliste ist `[5, 1]`; `[2, 5, 1]` lässt sich als `2 : [5, 1]` schreiben.

Es ist in Haskell üblich, dass Bezeichner von Listen und Elementen von Listen folgende Form haben, z. B. Element `x` und Liste `xs`, oder Element `d` und Liste `ds`. Der Listenbezeichner wird durch Anhängen eines `s` gebildet, als stünde der Elementbezeichner in der (englischen) Pluralform, denn die Liste enthält im allgemeinen mehrere Werte vom Typ des Elements. Auch wenn gar kein Elementbezeichner benötigt wird, enden Listenbezeichner meist mit `s`.

Die folgenden Beispiele verwenden pattern matching auf Listen. Das pattern `[]` passt ausschließlich auf leere Listen. Dagegen passt das pattern `(x : xs)` ausschließlich auf Listen, die mindestens ein Element enthalten, wobei der Kopf der Liste an `x` und die Restliste an `xs` gebunden wird.

Beispiel 15.10. Eine Listen verarbeitende Funktion:

```
sumList :: [Int] -> Int
sumList []       = 0
sumList (x : xs) = x + sumList xs
```

oder:

```
type Intlist  = [Int]

double :: Intlist -> Intlist
double []       = []
double (x : xs) = (2 * x) : double xs
```

Eine Beispielrechnung:

```
  double [2, 3]
= (2 * 2) : double [3]
= (2 * 2) : ((2 * 3) : double [])
= 4 : ((2 * 3) : double [])
= 4 : (6 : double [])
= 4 : (6 : [])
= 4 : [6]
= [4, 6]
```
□

Eine Standardfunktion für Listen ist die Verkettung `++`:

```
[2, 3] ++ [3, 4] = [2, 3, 3, 4].
```

Nur Listen gleichen Typs dürfen verkettet werden. Des Weiteren ist der Operator `++` rechtsassoziativ und hat die gleiche Bindungsstärke wie `:`.

Die Verkettungsfunktion lässt sich allerdings auch „von Hand“ programmieren:

```
append :: [Int] -> [Int] -> [Int]
append []       ys = ys
append (x : xs) ys = x : append xs ys
```

Beispiel 15.11. Als Beispiel für ein etwas größeres Haskell-Programm wollen wir jetzt eine Liste von Zahlen sortieren (siehe S. Thompson). Dazu verwenden wir den folgenden rekursiven Algorithmus `iSort`:

> Wenn `[x1, ..., xn]` die zu sortierende Liste ist, dann sortiere die Liste `[x2, ..., xn]` und füge das Element `x1` an der passenden Stelle ein.

```
iSort :: [Int] -> [Int]
iSort []       = []
iSort (x : xs) = ins x (iSort xs)

ins :: Int -> [Int] -> [Int]
ins x []      = [x]
ins x (y : ys)
  | x <= y    = x : y : ys
  | otherwise = y : ins x ys
```

Beispielrechnung:

```
  iSort [4, 9, 1]
= ins 4 (iSort [9, 1])
= ins 4 (ins 9 (iSort [1]))
= ins 4 (ins 9 (ins 1 (iSort [])))
= ins 4 (ins 9 (ins 1 []))
= ins 4 (ins 9 [1])                 -- (9 <= 1) = False
= ins 4 (1 : ins 9 [])
= ins 4 (1 : [9])
= ins 4 [1, 9]                      -- (4 <= 1) = False
= 1 : ins 4 [9]                     -- (4 <= 9) = True
= 1 : 4 : 9 : []
= 1 : 4 : [9]
= 1 : [4, 9]
= [1, 4, 9]
```

Auf der Seite 187 werden wir die Korrektheit dieses Programms beweisen. □

Funktionen (a -> b):

Auch Funktionen können mit Hilfe von `type` vereinbart werden:

```
type Loc   = Int
type State = Loc -> Int
```

15.2.3 Algebraische Datentypen

Ausgehend von den Basistypen können mit Hilfe der kompositen Typen neue Datenstrukturen erzeugt werden. Leider enthalten aber alle diese Datenstrukturen letztlich die Basistypen als elementare Einheiten, d. h. der Programmierer muss seine Daten letztlich als Zahlen oder Buchstaben codieren. Deshalb ist es – wie auch in der imperativen Programmierung – erwünscht, *problemspezifische* Datenkonstruktoren benutzen zu können. In *C* wurde das z. B. durch die Aufzählungstypen programmiertechnisch unterstützt. In funktionalen Sprachen benutzt man dafür algebraische Datentypen. Sie werden durch das Schlüsselwort `data` gekennzeichnet.

Beispiel 15.12.

```
data Season = Spring | Summer | Autumn | Winter
data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

dies entspricht in C:

```
typedef enum Sn {Spring, Summer, Autumn, Winter} Season;
typedef enum Ds {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} Day;
```

□

Die Elemente, die rechts vom Gleichheitszeichen stehen, nennt man Datenkonstruktoren oder einfach: Konstruktoren. Funktionen können z. B. durch pattern matching auf Konstruktoren definiert werden:

Beispiel 15.13.

```
isWorkingDay :: Day -> Bool
isWorkingDay Saturday = False
isWorkingDay Sunday   = False
isWorkingDay _        = True
```

Diese Funktion bildet alle Arbeitstage auf `True` und alle Wochenendtage auf `False` ab. □

Der Unterstrich _ in Beispiel 15.13 heißt *wildcard* und darf auf der linken Seite einer Definition überall da stehen, wo auch ein Variablenbezeichner stehen dürfte, ggf. auch mehrfach. Wie eine Variable akzeptiert der Unterstrich jeden übergebenen Wert, ohne den Wert jedoch an einen Bezeichner zu binden. Damit ist für den Leser einer Funktion sofort erkennbar, wenn ein ihr übergebener Wert gar nicht verwendet wird.

Die in den obigen Beispielen genannten Konstruktoren sind nullstellig in dem Sinne, dass sie nicht Bezug auf andere Datentypen nehmen. Im allgemeinen ist das sehr wohl möglich wie das Beispiel der arithmetischen Ausdrücke zeigt.

Beispiel 15.14.

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
```

Offensichtlich ist der Konstruktor `Add` zweistellig, weil er auf zwei Datentypen (nämlich zweimal `Expr`) Bezug nimmt, und der Konstruktor `Lit` ist einstellig.
Dann ist z. B. `(Add (Sub (Lit 9) (Lit 12)) (Lit 7))` ein Element dieses Datentyps; es bezeichnet den Ausdruck $(9 - 12) + 7$. □

Im Allgemeinen hat ein algebraischer Datentyp die folgende Form:

```
data Typename
  = Con1 t1,1 ... t1,k1
  | Con2 t2,1 ... t2,k2
  | ...
  | Conr tr,1 ... tr,kr
```

wobei

- `Typename` ein Name ist, der mit einem großen Buchstaben beginnt.
- `Con1`, ..., `Conr` Datenkonstruktoren sind, die alle mit einem großen Buchstaben beginnen.
- Die $\texttt{t}_{\texttt{i,j}}$ Typnamen sind, die ebenfalls alle mit einem großen Buchstaben beginnen.
- `ki` die Stelligkeit des Konstruktors `Coni` ist.

Auch bei algebraischen Datentypen mit Konstruktoren, deren Stelligkeit größer als 0 ist, kann man mit pattern matching arbeiten.

Beispiel 15.15.

```
eval :: Expr -> Int
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2

height :: Expr -> Int
height (Lit _)     = 1
height (Add e1 e2) = 1 + max (height e1) (height e2)
height (Sub e1 e2) = 1 + max (height e1) (height e2)
```

Die Funktion `eval` rechnet den Wert eines Ausdrucks aus, die Funktion `height` die Höhe des Syntaxbaumes eines Ausdrucks. □

Beispiel 15.16. Auch besteht die Möglichkeit, dass sich verschiedene algebraische Datentypen rekursiv aufrufen. Stellen wir uns dazu folgende EBNF-Definition *ohne Wiederholungsklammern* (siehe Abschnitt 2.2) mit Variablen S, A und B und den folgenden EBNF-Regeln vor:

$$S ::= AB \qquad A ::= \hat{(}aAb \mid ab\hat{)} \qquad B ::= \hat{(}bB \mid b\hat{)}$$

Dann lässt sich die Menge der abstrakten Syntaxbäume von Wörtern dieser EBNF-Definition durch den folgenden algebraischen Datentyp programmieren:

```
data TreeS = S_AB  TreeA TreeB
data TreeA = A_aAb TreeA  |  A_ab
data TreeB = B_bB  TreeB  |  B_b
```

Es werden die algebraischen Datentypen `TreeS`, `TreeA` und `TreeB` definiert. Bei diesem gegenseitigen Aufruf von algebraischen Datentypen spricht man auch von *mutual recursion*. Die EBNF-Regeln sind in Konstruktoren (`S_AB`, `A_aAb`, `A_ab`, `B_bB`, `B_b`) umgewandelt; die Stelligkeit eines Konstruktors entspricht der Anzahl der Variablen auf der rechten Seite der EBNF-Regel.

Der abstrakte Syntaxbaum von *aabbbb* lautet: `(S_AB (A_aAb A_ab) (B_bB B_b))`. □

`case`-Ausdrücke Da algebraische Datentypen in Haskell eine wichtige Rolle spielen, ist pattern matching ein häufig benötigtes Konzept. Bisher haben wir pattern matching jedoch nur auf der linken Seite von Funktionsdefinitionen gesehen (links vom `=`). Das soll sich mit der Einführung von `case`-*Ausdrücken* ändern.

Beispiel 15.17. Mit einem `case`-Ausdruck kann man die Funktion `height` aus Beispiel 15.15 auch wie folgt definieren.

```
height :: Expr -> Int
height e = 1 + case e of
                 Lit _     -> 0
                 Add e1 e2 -> max (height e1) (height e2)
                 Sub e1 e2 -> max (height e1) (height e2)
```

□

Allgemein haben `case`-Ausdrücke die folgende Form.

```
case exp0 of
  p1 -> exp1
  p2 -> exp2
  ...
  pk -> expk
```

Dabei sind `exp0`, `exp1`, ..., `expk` Ausdrücke und `p1`, `p2`, ..., `pk` patterns (`k` ≥ 1). Variablen, die in einem pattern `pi` ($1 \leq$ `i` $\leq$ `k`) gebunden werden, können im Ausdruck `expi` verwendet werden. Zur Auswertung des `case`-Ausdruckes werden die patterns der Reihe nach auf dem Wert von `exp0` getestet. Ist `pi` das erste passende pattern, dann ist der Wert des `case`-Ausdruckes der Wert von `expi`.

Laut Haskell-Standard sind in `case`-Ausdrücken auch guards und `where`-Klauseln erlaubt[3]; wir beschränken uns hier jedoch auf den beschriebenen einfachen Fall.

[3] `http://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-460003.13`

15.3 Funktionen höherer Ordnung

Funktionen können Argumente und Resultate haben, die selber wieder Funktionen sind. Dann spricht man von Funktionen höherer Ordnung. Solche Funktionen können als Kontrollstrukturen aufgefasst werden, wie sich an den folgenden Beispielen zeigen wird. Sie sind aber – im Gegensatz zu *while-* und *for-Konstrukten* von *C* – benutzerdefiniert.

15.3.1 Beispiele

Beispiel 15.18.

```
map :: (Int -> Int) -> [Int] -> [Int]
map f []       = []
map f (x : xs) = f x : map f xs
```

Die Funktion `map` nimmt ein Argument vom Typ `Int -> Int` und liefert eine Funktion vom Typ `[Int] -> [Int]` als Ergebnis. Damit lässt sich sehr einfach aus einer Liste von Zahlen eine Liste der mit 2 multiplizierten Zahlen konstruieren:

```
double :: Int -> Int
double x = 2 * x

  map double [2, 1, 4]
= double 2 : map double [1, 4]
= (2 * 2) : map double [1, 4]
= 4 : double 1 : map double [4]
...
= 4 : 2 : 8 : map double []
= 4 : 2 : 8 : []
= [4, 2, 8]
```

Möchte man jetzt auf die Listenelemente eine andere Funktion anwenden, so braucht man diese nur zu definieren und als erstes Argument von `map` einzugeben. Wir sehen also, dass `map` wie eine Art von benutzerdefinierter Kontrollstruktur wirkt (vergleichbar zur `for`-Schleife über einem Feld). □

Beispiel 15.19. Die Funktion `foldr` geht über `map` hinaus, indem sie die Ergebnisse von `f` *faltet*, d. h. sie übergibt `f` neben jedem Listeneintrag das bisherige Zwischenergebnis als zweites Argument:

```
foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldr _ z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Nun kann ganz einfach die Summe oder das Produkt der Zahlen einer Liste programmiert werden:

```
sumList :: [Int] -> Int
sumList xs = foldr (+) 0 xs

prodList :: [Int] -> Int
prodList xs = foldr (*) 1 xs
```

□

Beispiel 15.20. Sehr hilfreich ist es auch, eine Funktion `filter` zu haben, die aus einer Liste von Zahlen solche mit einer bestimmten Eigenschaft herausfiltert, d. h. die Zahlen der Argumentliste, welche die Eigenschaft besitzen, werden in die Ergebnisliste geschrieben:

```
filter :: (Int -> Bool) -> [Int] -> [Int]
filter _ [] = []
filter p (x : xs)
  | p x       = x : filter p xs
  | otherwise =     filter p xs
```

Beispielsweise möchte man aus einer Liste von Zahlen alle diejenigen herausfiltern, die geradzahlig sind.

```
even :: Int -> Bool
even x = mod x 2 == 0

  filter even [1, 4, 2]            -- even 1 = mod 1 2 == 0
                                   --        = 1 == 0
                                   --        = False
= filter even [4, 2]               -- even 4 = mod 4 2 == 0
                                   --        = 0 == 0
                                   --        = True
= 4 : filter even [2]              -- even 2 = mod 2 2 == 0
                                   --        = 0 == 0
                                   --        = True
= 4 : 2 : filter even []
= 4 : 2 : []
= [4, 2]
```

□

Beispiel 15.21. Eine weitere sehr nützliche Funktion höherer Ordnung ist die Funktionskomposition, die sich wie folgt sehr einfach definieren lässt:

```
compose :: (Int -> Int) -> (Int -> Int) -> Int -> Int
compose f g x = f (g x)
```

Die Funktionskomposition ist im Modul `Prelude` bereits als Operator definiert: `(f . g) x = f (g x)`.

□

15.3.2 Partielle Applikation von Funktionen und Operatoren

Damit eine Haskell-Funktion eine Funktion zurück gibt, muss man an der Funktionsdefinition im allgemeinen nichts ändern, da der Funktionspfeil `->` in Typangaben rechtsassoziativ ist.

Zur Verdeutlichung sei `f` eine Funktion mit dem Typ `Int -> Int -> Bool`. Die Funktionsanwendung `f 2 3` liefert dann einen Wert vom Typ `Bool`. Da der Typ von `f` gedanklich von rechts geklammert wird (d. h. `Int -> (Int -> Bool)`), liefert `f 2` einen Wert vom Typ `Int -> Bool`. Man spricht in so einem Fall von einer *partiellen Applikation* von `f`, oder man sagt `f` ist *unterversorgt*. So kann `f 2` ohne Umstände beispielsweise an die Funktion `filter` aus dem vorherigen Abschnitt übergeben werden: `filter (f 2) [1, 2, 3]`.

Auch bei Operatoren besteht die Möglichkeit der partiellen Applikation. Dabei kann man sich sogar aussuchen, welche Seite des Operators man unterversorgt. Während `(1 / 2)` den Quotienten aus `1` und `2` bildet, ist `(1 /)` eine Funktion, die das Reziproke eines Wertes liefert, und `(/ 2)` eine Funktion, die einen Wert halbiert. Unterversorgt man beide Seiten eines Operators, wandelt man den Operator effektiv in eine gewöhnliche Funktion um: `(/) 1 2 = 1 / 2`.

Beispiel 15.22. Als ein weiteres Beispiel für die Benutzung von Funktionen höherer Ordnung und von partieller Applikation programmieren wir den Quicksort-Algorithmus zum Sortieren einer Liste von Zahlen.

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x : xs)
  = quicksort (filter (x >) xs) ++ [x] ++ quicksort (filter (x <=) xs)
```

Die Abarbeitung von `quicksort [3, 1, 6]` geschieht wie folgt:

```
  quicksort [3, 1, 6]
= quicksort (filter (3 >) [1, 6]) ++ [3] ++ quicksort (filter (3 <=) [1, 6])
             -- (3 >) 1 = True
= quicksort (1 : filter (3 >) [6]) ++ [3] ++ quicksort (filter (3 <=) [1, 6])
                  -- (3 >) 6 = False
= quicksort (1 : filter (3 >) []) ++ [3] ++ quicksort (filter (3 <=) [1, 6])
= quicksort (1 : []) ++ [3] ++ quicksort (filter (3 <=) [1, 6])
...
```

```
= [1] ++ [3] ++ quicksort (filter (3 <=) [1, 6])
                          -- (3 <=) 1 = False
= [1] ++ [3] ++ quicksort (filter (3 <=) [6])
                          -- (3 <=) 6 = True
= [1] ++ [3] ++ quicksort (6 : filter (3 <=) [])
= [1] ++ [3] ++ quicksort (6 : [])
...
= [1] ++ [3] ++ [6]
= [1, 3, 6]
```
□

15.3.3 Anonyme Funktionen

Es ist in Haskell möglich Funktionen zu definieren, ohne ihnen einen konkreten Bezeichner zu geben. Sie werden *anonyme Funktionen* genannt. Ihre Syntax ist wie folgt definiert:

```
\ p1 ... pn -> ausdruck
```

Eine solche Definition akzeptiert sequentiell `n` Argumente und liefert das durch `ausdruck` beschriebene Ergebnis, wobei `p1`, ..., `pn` für pattern (im einfachsten Fall nur Variablenbezeichner) stehen und `ausdruck` für einen Ausdruck, in dem die Variablenbezeichner aus den pattern verwendet werden dürfen, steht. Im Gegensatz zu nicht anonymen Funktionen können hier nicht mehrere Definitionszeilen mit verschiedenen pattern angegeben werden.

Eine anonyme Funktion kann in einem Ausdruck überall da genutzt werden, wo auch ein Funktionsbezeichner stehen kann, wie die folgenden Beispiele zeigen.

Beispiel 15.23. Die Funktion `addPairs` verlangt eine Liste von Paaren vom Typ `(Int, Int)` und erstellt eine Liste mit den Summen der Komponenten der Paare. Wir modifizieren zunächst den Typ von `map`, behalten aber die Gleichungen bei:

```
map :: ((Int, Int) -> Int) -> [(Int, Int)] -> [Int]

addPairs :: [(Int, Int)] -> [Int]
addPairs = map (\ (x, y) -> x + y)
```
□

Beispiel 15.24. Die Funktion `elem` überprüft, ob ein gegebener Wert vom Typ `Int` in einer ebenfalls gegebenen Liste vom Typ `[Int]` vorkommt, und gibt entsprechend einen Wert vom Typ `Bool` zurück. Entsprechend verändern wir den Typ von `foldr`:

```
foldr :: (Int -> Bool -> Bool) -> Bool -> [Int] -> Bool
foldr _ b []       = b
foldr f b (x : xs) = f x (foldr f b xs)

(||) :: Bool -> Bool -> Bool
True  || _ = True
False || x = x

elem :: Int -> [Int] -> Bool
elem x = foldr (\ y b -> x == y || b) False
```

Die Abarbeitung von `elem 3 [1, 3, 2]` geschieht wie folgt:

```
  elem 3 [1, 3, 2]
= foldr (\ y b -> 3 == y || b) False [1, 3, 2]
= (\ y b -> 3 == y || b) 1 (foldr (\ y b -> 3 == y || b) False [3, 2])
= 3 == 1 || foldr (\ y b -> 3 == y || b) False [3, 2]
= False || foldr (\ y b -> 3 == y || b) False [3, 2]
= foldr (\ y b -> 3 == y || b) False [3, 2]
= (\ y b -> 3 == y || b) 3 (foldr (\ y b -> 3 == y || b) False [2])
= 3 == 3 || foldr (\ y b -> 3 == y || b) False [2]
= True || foldr (\ y b -> 3 == y || b) False [2]
= True
```
□

Anonyme Funktionen werden auch Lambda-Abstraktionen genannt, da sie vom Lambda-Kalkül inspiriert sind. Diesen stellen wir in Abschnitt 15.8 vor.

15.4 Typpolymorphie und Typüberprüfung

15.4.1 Polymorphie

Beispiel 15.25. Nehmen wir an, dass wir die Länge einer Liste von Zahlen programmieren müssten. Dann wäre das folgende Programm dafür geeignet:

```
length' :: [Int] -> Int
length' []       = 0
length' (_ : xs) = 1 + length' xs
```

Nehmen wir nun an, dass sich dieselbe Aufgabe für Listen von Booleschen Werten stellt. Dann wäre folgendes Programm geeignet:

```
length'' :: [Bool] -> Int
length'' []       = 0
length'' (_ : xs) = 1 + length'' xs
```

Wir sehen, dass es zwischen den beiden Programmen nur den Unterschied im Argumenttyp gibt, genauer gesagt: Der Typ der Listenelemente unterscheidet sich. Aber offensichtlich ist es der Funktion `length` egal, ob sie eine Liste mit Zahlen, Booleschen Werten oder etwa Listen von Paaren aus Zahlen und Booleschen Werten als Argument bekommt. In solchen Fällen wollen wir eine programmtechnische Möglichkeit haben, direkt das Essentielle der Funktion `length` hinzuschreiben. Dafür benutzen wir sogenannte *Typvariablen*

```
length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

Hier ist `a` eine Typvariable, die mit einem beliebigen Typ instanziiert werden kann. Im Unterschied zur Typbezeichnung beginnt der Name der Typvariablen immer mit einem kleinen Buchstaben. Der Typ von `length` heißt *polymorpher Typ*, weil er verschiedene Gestalten annehmen kann. Wird `length` in einem konkreten Kontext benutzt, so wird die Typvariable `a` entsprechend instanziiert. □

Auch andere nützliche Listenfunktionen hängen nicht vom Typ der Elemente ab und sollten sofort polymorph definiert werden.

```
append :: [a] -> [a] -> [a]
append []       ys = ys
append (x : xs) ys = x : append xs ys

zip :: [a] -> [b] -> [(a, b)]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
zip _        _        = []

map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = f x : map f xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

Auch algebraische Datentypen können die Polymorphie nutzen.

Beispiel 15.26.

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Im rekursiv definierten algebraischen Datentyp `Tree` tritt die Typvariable `a` auf, die an einen beliebigen Typ gebunden werden kann.

```
height :: Tree a -> Int
height Nil            = 1
height (Node n t1 t2) = 1 + max (height t1) (height t2)
```

Die Funktion `height` berechnet die Höhe (d. h. maximale Länge von Pfaden) eines vorlegten Baums. Welchen Typ die Knotenwerte haben spielt hierbei natürlich keine Rolle.

```
collapse :: Tree a -> [a]
collapse Nil            = []
collapse (Node n t1 t2) = collapse t1 ++ [n] ++ collapse t2
```

Die Funktion `collapse` durchläuft den vorgelegten Baum „inorder“ und gibt die Knotenbeschriftungen aus. □

Ein wichtiger im Modul `Prelude` bereits vordefinierter polymorpher algebraischer Datentyp ist `Maybe`. Mit ihm lassen sich z. B. Berechnungen modellieren, die fehlschlagen können.

```
data Maybe a = Nothing | Just a

safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (div x y)  -- div ist die ganzzahlige Division.
```

Weitere Anwendungsbeispiele von `Maybe` können im Abschnitt 15.6 nachgelesen werden

15.4.2 Typüberprüfung

Beispiel 15.27. Betrachten wir nun die drei folgenden Funktionen:

```
f :: (t, Char) -> (t, [Char])
f (...) = ...

g :: (Int, [u]) -> Int
g (...) = ...

h = g . f
```

Der Punkt „`.`“ in der Definition der Funktion `h` ist ein Standardoperator in Haskell und steht für die Funktionskomposition. Er hat den Typ `. :: (b -> c) -> (a -> b) -> (a -> c)`.

Welchen Typ hat `h`? Um das herausfinden zu können, müssen offensichtlich zunächst die Typausdrücke `(t, [Char])` und `(Int, [u])` in Übereinstimmung gebracht werden. Wenn das gelingt, dann sagen wir auch, dass die Typen *unifizierbar* sind. Um diese Frage algorithmisch beantworten zu können, wandeln wir zunächst die Typausdrücke in Typterme um. □

Typausdrücke

Die Menge *TypA* der Typausdrücke ist die kleinste Menge von Ausdrücken (d. h. Zeichenreihen), die die folgenden fünf Bedingungen erfüllt:

- `Int`, `Bool`, `Float`, `Char` und `String` sind Typausdrücke.
- Jede Typvariable ist ein Typausdruck.
- Wenn `e` ein Typausdruck ist, dann ist der Listentyp `[e]` auch ein Typausdruck.
- Wenn `e1`, ..., `en` Typausdrücke sind, dann ist auch der Tupeltyp `(e1, ..., en)` ein Typausdruck.
- Wenn `e1` und `e2` Typausdrücke sind, dann ist auch der Funktionstyp `(e1 -> e2)` ein Typausdruck.

Typterme

Die Menge $TypT$ der Typterme ist die Menge der Terme über Σ_T indiziert mit X, wobei

$$\Sigma_T = \{()^n \mid n \geq 1\} \cup \{[]^{(1)}, \to^{(2)}\} \cup \{Int^{(0)}, Char^{(0)}, Float^{(0)}, Bool^{(0)}\}$$

und X die Menge der Typvariablen ist (zur Definition von Termen siehe Anhang B6).

Jeder Typausdruck `e` kann in einen Typterm $trans($`e`$)$ umgewandelt werden. Dabei ist die Übersetzung $trans\colon TypA \to TypT$ induktiv über den Aufbau von $TypA$ wie folgt definiert:

- $trans($`Int`$) = Int$, $trans($`Bool`$) = Bool$, $trans($`Float`$) = Float$ und $trans($`Char`$) = Char$
- Wenn `e` = `t` eine Typvariable ist, dann ist $trans($`e`$) = t$.
- Wenn `e` = `[e']`, dann ist $trans($`e`$) = [\,](trans($`e'`$))$.
- Wenn `e` = `(e1, ..., en)`, dann ist $trans($`e`$) = ()^n(trans($`e1`$), \ldots, trans($`en`$))$.
- Wenn `e` = `(e1 -> e2)`, dann ist $trans($`e`$) = \to(trans($`e1`$), trans($`e2`$))$.

Die Operatoren Int, $Bool$, $Float$, $Char$, t, $[\,]$, $()^n$ und $\to$ nennt man auch Typkonstruktoren.

Beispiel 15.27 (Fortsetzung)**.**

$$trans(\texttt{(t, [Char])}) = ()^2(t, [\,](Char))$$
$$trans(\texttt{(Int, [u])}) = ()^2(Int, [\,](u))\,.$$

Ob die beiden Typausdrücke `(t,[Char])` und `(Int,[u])` in Übereinstimmung miteinander gebracht werden können, ist gleichbedeutend mit der Frage, ob die beiden Typterme $trans($`(t,[Char])`$)$ und $trans($`(Int,[u])`$)$ unifizierbar sind. Wir wollen im folgenden von dem speziellen Rangalphabet Σ_T abstrahieren und die Unifikation von Termen über beliebigen Rangalphabeten betrachten. □

Unifikation von Termen

Sei $X = \{x_1, x_2, x_3, \ldots\}$ eine Menge von Variablen und Σ ein *beliebiges* Rangalphabet. Eine *Variablenbelegung* ist eine Abbildung $\varphi\colon X \to T_\Sigma(X)$, bei der $\{x \in X \mid \varphi(x) \neq x\}$ endlich ist. Die Erweiterung $\widetilde{\varphi}\colon T_\Sigma(X) \to T_\Sigma(X)$ von φ auf Terme durch „Einsetzen“ definiert ist:

- für jede Variable x gilt $\widetilde{\varphi}(x) = \varphi(x)$ und
- für jeden k-stelligen Konstruktor $\delta \in \Sigma$ und Terme $t_1, \ldots, t_k \in T_\Sigma(X)$ gilt $\widetilde{\varphi}(\delta(t_1, \ldots, t_k)) = \delta(\widetilde{\varphi}(t_1), \ldots, \widetilde{\varphi}(t_k))$.

Zwei Terme t_1 und t_2 über einem beliebigen Rangalphabet Σ sind *unifizierbar*, wenn es eine Variablenbelegung φ gibt, so dass $\widetilde{\varphi}(t_1) = \widetilde{\varphi}(t_2)$. Die Abbildung φ heißt *Unifikator* von t_1 und t_2.

Seien φ_1 und φ_2 zwei Unifikatoren von t_1 und t_2. φ_1 heißt *allgemeiner als* φ_2, wenn es eine Variablenbelegung ν gibt, so dass $\varphi_2(x) = \widetilde{\nu}(\varphi_1(x))$ für jede Variable x. Ein Unifikator φ heißt *allgemeinster Unifikator*, wenn φ allgemeiner als jeder Unifikator von t_1 und t_2 ist.

Beispiel 15.27 (Fortsetzung)**.** Für die beiden Typterme

$$t_1 = [\,](()^3(t, Char, \to(v, Int))) \qquad \text{und} \qquad t_2 = [\,](()^3(Char, u, w))$$

seien zwei Variablenbelegungen φ_1 und φ_2 wie folgt definiert:

$$\varphi_1(t) = Char\,, \qquad \varphi_1(u) = Char\,, \qquad \varphi_1(w) = \to(v, Int)\,,$$
$$\varphi_2(t) = Char\,, \qquad \varphi_2(u) = Char\,, \qquad \varphi_2(w) = \to(Int, Int)\,, \qquad \varphi_2(v) = Int\,.$$

Dann sind φ_1 sowie φ_2 Unifikatoren von t_1 und t_2, jedoch ist φ_1 allgemeiner als φ_2, da es eine Variablenbelegung ν gibt, so dass $\varphi_2 = \nu \circ \varphi_1$ gilt (diese ist definiert als $\nu(v) = Int$). Somit gilt die Gleichheit $\widetilde{\nu}(\varphi_1(w)) = \widetilde{\nu}(\to(v, Int)) = \to(Int, Int) = \varphi_2(w)$. φ_1 ist der allgemeinste Unifikator von t_1 und t_2. □

Algorithmus 12 Unifikationsalgorithmus

Eingabe: Terme s und t über einem Rangalphabet Σ und einer Variablenmenge X.

Ausgabe: Wenn s und t unifizierbar sind, dann wird deren allgemeinster Unifikator ausgegeben. Wenn s und t nicht unifizierbar sind, dann wird dieser Sachverhalt ausgegeben.

Vorgehen:

1. Setze $M := \{\binom{s}{t}\}$ und wende so lange eine der vier folgenden Regeln an, bis keine mehr anwendbar ist.
   - *Dekompositionsregel*: Wenn M ein Paar der Form $\binom{\delta(s_1,\ldots,s_k)}{\delta(t_1,\ldots,t_k)}$ enthält, wobei $\delta \in \Sigma$ ein k-stelliger Konstruktor ist und $s_1, \ldots, s_k, t_1, \ldots, t_k$ Terme über Konstruktoren und Variablen sind, dann lösche das Paar $\binom{\delta(s_1,\ldots,s_k)}{\delta(t_1,\ldots,t_k)}$ aus M und füge die Paare $\binom{s_1}{t_1}, \ldots, \binom{s_k}{t_k}$ hinzu. Hinweis: ist δ nullstellig, dann wird durch Anwendung dieser Regel das Paar $\binom{\delta}{\delta}$ aus der Menge M entfernt, ohne ein neues Paar in M einzufügen.
   - *Elimination trivialer Gleichungen*: Wenn M ein Paar $\binom{x}{x}$ für eine Variable x enthält, dann lösche dieses Paar aus M.
   - *Vertauschung*: Wenn M ein Paar $\binom{t}{x}$ enthält und t ist keine Variable, dann lösche dieses Paar aus M und füge das Paar $\binom{x}{t}$ hinzu.
   - *Substitution von Variablen*: Wenn M das Paar $\binom{x}{t}$ enthält und x kommt in t nicht vor (occur check), dann ersetze in jedem anderen Paar von M die Variable x durch den Term t.
2. Ist M von der Form $\{\binom{u_1}{t_1}, \ldots, \binom{u_k}{t_k}\}$, so dass $u_1, \ldots, u_k$ paarweise verschiedene Variablen und $t_1, \ldots, t_k$ Terme über Konstruktoren und Variablen sind, in denen die u_is nicht mehr vorkommen, dann gibt es einen allgemeinsten Unifikator φ von t_1 und t_2. Dabei ist $\varphi(u_i) = t_i$, und $\varphi(x) = x$ für alle Variablen x die nicht in u_1, …, u_n vorkommen. Wenn M nicht diese Form hat, dann sind t_1 und t_2 nicht unifizierbar.

Wenn zwei Terme unifizierbar sind, so ist der allgemeinste Unifikator (bis auf Variablenumbenennung) eindeutig bestimmt.

Es ist algorithmisch entscheidbar, ob zwei Terme unifizierbar sind oder nicht. Der Algorithmus heißt Unifikationsalgorithmus (Algorithmus 12). Zur besseren Überschaubarkeit verwenden wir hier die Schreibweise $\binom{t_1}{t_2}$ für das Paar (t_1, t_2).

Beispiel 15.27 (Fortsetzung)**.** Wenden wir nun den Unifikationsalgorithmus auf ein Beispiel an; es entsteht eine Sequenz von Mengen von Paaren, wobei die $(i+1)$-te Menge aus der i-ten Menge durch Anwendung einer der vier Regeln hervorgeht.

$$M_1 = \left\{ \binom{()^2(t, [\,](Char))}{()^2(Int, [\,](u))} \right\}$$

$$M_2 = \left\{ \binom{t}{Int}, \binom{[\,](Char)}{[\,](u)} \right\} \qquad \text{(durch Dekomposition mit } \delta = ()^2\text{)}$$

$$M_3 = \left\{ \binom{t}{Int}, \binom{Char}{u} \right\} \qquad \text{(durch Dekomposition mit } \delta = [\,]\text{)}$$

$$M_4 = \left\{ \binom{t}{Int}, \binom{u}{Char} \right\}. \qquad \text{(durch Vertauschung)}$$

Also ist φ mit $\varphi(t) = Int$ und $\varphi(u) = Char$ der allgemeinste Unifikator von $()^2(t, [\,](Char))$ und $()^2(Int, [\,](u))$. □

15.5 Typklassen

Im vorangegangenen Abschnitt haben wir die Typpolymorphie kennengelernt. Es stellt sich jedoch heraus, dass die bisherigen Konzepte für eine angenehme Programmierung noch nicht ausreichend sind.

Sei `elem` eine Funktion, die mittels des Operators `==` prüft, ob ein gegebener Wert in einer ebenfalls gegebenen Liste vorkommt, und entsprechend ein Ergebnis vom Typ `Bool` liefert. Welchen Typ soll `elem` haben? Der Typ `Int -> [Int] -> Bool` ist unbefriedigend, schließlich möchte man auch für Werte vom Typ `Float` prüfen können, ob der Wert in einer Liste vom Typ `[Float]` vorkommt. Der polymorphe Typ `a -> [a] -> Bool` ist dagegen zu allgemein, denn es ist nicht beschränkt, durch welche Typen die Typvariable `a` instanziiert werden kann; so könnte `a` durch `Int -> Int` instanziiert werden, doch der Operator `==` kann keine Funktionen vergleichen.

Um die Möglichkeiten der Instanziierung von Typvariablen einzuschränken gibt es das Konzept der *Typklassen* und deren *Instanzen*. Für unser Beispiel benötigen wir die in Haskell bereits vordefinierte Typklasse `Eq`.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

Eine Typklasse definiert zunächst die Typen einiger polymorpher Funktionen, für die wir die Instanziierung der Typvariablen einschränken wollen (hier sind es die Operatoren `==` und `/=`). Außerdem ist es möglich, für jede dieser Funktionen eine *Standardimplementierungen* anzugeben (hier für beide Operatoren angegeben; eine Erläuterung folgt). Durch die Deklaration von *Instanzen* legt man fest, durch welche Typen die in der Klassendeklaration genannte Typvariable (hier `a`) instanziiert werden kann. Beispielsweise kann `a` mit folgendem Programmstück durch `Int` instanziiert werden:

```
instance Eq Int where
  x == y = eqInt x y
```

Dabei soll `eqInt :: Int -> Int -> Bool` als Platzhalter für den Vergleich für Werte vom Typ `Int` stehen, um hier nicht auf Details des Vergleichs eingehen zu müssen. Mit einer Instanzdeklaration legt man die Implementierung der durch die Typklasse deklarierten Funktionen für einen konkreten Typen (hier `Int`) fest. Es fällt auf, dass hier keine Implementierung für `/=` angegeben wurde. Das ist hier nicht notwendig, da bereits in der Typklasse eine Standardimplementierung dieses Operators gegeben wurde, nämlich mithilfe von `==`. Eine Standardimplementierung darf in einer Instanzdeklaration auch überschrieben werden, wie es hier für `==` geschehen ist. In unserem Beispiel muss sogar für jede Instanz mindestens eine Standardimplementierung überschrieben werden, weil ansonsten die Berechnung der Operationen für die jeweilige Instanz nicht terminieren würde.

Nun können wir den Typ für unsere Funktion `elem` festlegen: `elem :: (Eq a) => a -> [a] -> Bool`. Damit wird die Instanziierung der Typvariable `a` auf solche Typen eingeschränkt, die Instanz der Typklasse `Eq` sind (das wird durch die Bedingung `(Eq a)` erzwungen). Somit ist es mit unseren bisherigen Deklarationen möglich, mittels `elem` zu prüfen, ob ein Wert vom Typ `Int` in einer Liste vom Typ `[Int]` vorkommt.

Eine Typklasse wird natürlich erst nützlich, wenn sie mehrere Instanzen hat. Für unser Beispiel können wir z. B. noch den Test auf Gleichheit für Listen implementieren.

```
instance (Eq a) => Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _        == _        = False
```

Damit Listen auf Gleichheit getestet werden können, müssen offenbar auch deren Elemente auf Gleichheit getestet werden können. Diese Bedingung wird hier ähnlich wie bei Funktionstypen durch `(Eq a) =>` angegeben. Die letzten drei Zeilen schreiben die Instanzdeklaration der Funktion `==` für Listen fest; sie benutzt die Deklaration der Funktion `==` auf Elementen der Liste.

Die gezeigte Instanzdeklaration ist auch rekursiv anwendbar, d. h. wenn ein Typ `a` Instanz der Typklasse `Eq` ist, dann ist auch `[a]` Instanz von `Eq`, dann ist auch `[[a]]` Instanz von `Eq`, und so weiter. Nun kann `elem` also auch für alle Typen `a`, die Instanz der Typklasse `Eq` sind, prüfen, ob eine Liste vom Typ `[a]`

in einer Liste vom Typ `[[a]]` vorkommt. So kann der Typ von `elem` z. B. wie folgt instanziiert werden: `elem :: [Int] -> [[Int]] -> Bool`, `elem :: [[Int]] -> [[[Int]]] -> Bool`, und so weiter.

Die hier betrachtete Typklasse `Eq` und die Funktion `elem` sind im Modul `Prelude` bereits vordefiniert. Das folgende Beispiel zeigt die ebenfalls schon vordefinierte Typklasse `Functor`. Es verdeutlicht, dass man auch Typklassen für Typen, die einen Typparameter haben, definieren kann.

Beispiel 15.28. Wir möchten die Funktion `fmap` definieren, die es uns erlaubt, auf die Elemente von Strukturen wie Listen oder Bäumen eine Funktion anzuwenden. Außerdem möchten wir mit dem Operator `<$` alle Elemente einer solchen Struktur durch denselben Wert ersetzen können (z. B. für eine Liste: `5 <$ [2, 3, 4] = [5, 5, 5]`).

Zur Lösung der Aufgabe definieren wir die Typklasse `Functor`. Dabei ist `const x _ = x` vordefiniert und hat den Typ `a -> b -> a`.

```
class Functor f where
  fmap :: (b -> a) -> f b -> f a
  (<$) :: a -> f b -> f a
  (<$) = fmap . const
```

Die Typvariable `f` steht hier für die Strukturierung (z. B. in Listen oder Bäumen) von Werten eines Typs. Während für `fmap` keine Implementierung angegeben ist – diese muss für jede Instanz separat erstellt werden –, wurde für den Operator `<$` eine Standardimplementierung auf Basis von `fmap` angegeben.

Eine Struktur, für die die angegebenen Funktionen auf jeden Fall definiert sein sollten, sind Listen.

```
instance Functor [] where
  fmap _ []       = []
  fmap g (x : xs) = g x : fmap g xs
```

Für den Operator `<$` wird hier die Standardimplementierung benutzt.

Eine andere Struktur, die sich als Instanz von `Functor` anbietet, sind Bäume.

```
data Tree a = Nil | Node a (Tree a) (Tree a)

instance Functor Tree where
  fmap _ Nil            = Nil
  fmap g (Node x t1 t2) = Node (g x) (fmap g t1) (fmap g t2)
```

Damit lässt sich nun einfach die Funktion `setZero` definieren, die alle Werte innerhalb einer Liste oder eines Baumes auf `0` setzt:

```
setZero :: (Functor f) => f a -> f Int
setZero x = 0 <$ x
```

Eine Beispielberechnung kann wie folgt aussehen:

```
  setZero [1, 2]
= 0 <$ [1, 2]                       -- Definition setZero
= (fmap . const) 0 [1, 2]           -- Standardimplementierung <$
= fmap (const 0) [1, 2]             -- Definition .
= const 0 1 : fmap (const 0) [2]    -- Definition fmap aus instance Functor []
= 0 : fmap (const 0) [2]            -- Definition const
= 0 : const 0 2 : fmap (const 0) [] -- Definition fmap aus instance Functor []
= 0 : 0 : fmap (const 0) []         -- Definition const
= 0 : 0 : []                        -- Definition fmap aus instance Functor []
= [0, 0]
```

Ähnlich leicht lässt sich die Funktion `double` definieren, die die Werte innerhalb einer Liste oder eines Baumes verdoppelt:

```
double :: (Functor f, Num a) => f a -> f a
double = fmap (2 *)
```

Hier muss auch die Instanziierung der Typvariable `a` eingeschränkt werden, da die Operation $*$ nur für Typen, die Instanz der Typklasse `Num` sind, definiert ist. □

Die folgende Tabelle zeigt einige im Modul `Prelude` definierte Typklassen, einige Funktionen, die diese definieren, und einige Typen, die Instanz der Typklasse sind.

Typklasse	definierte Funktionen	Instanzen
`Eq`	`==`, `/=`	`Bool`, `Char`, `Float`, `Int`, `(Eq a) => [a]`, ...
`Ord`	`compare`, `<`, `>=`, `>`, `<=`, `min`, `max`	`Bool`, `Char`, `Float`, `Int`, `(Ord a) => [a]`, ...
`Num`	`+`, `*`, `-`, `negate`, `abs`, ...	`Double`, `Float`, `Int`, `Integer`, ...
`Fractional`	`/`, `recip`, `fromRational`	`Double`, `Float`, ...
`Functor`	`fmap`	`[]`, `Maybe`, ...

In den vorangegangenen Abschnitten wurden viele Funktionen vorgestellt, die im Modul `Prelude` bereits definiert sind; allerdings wurden die Typen meist vereinfacht angegeben, da die Konzepte der Typpolymorphie und der Typklassen noch nicht bekannt waren. Deshalb sollen hier nun ihre tatsächlichen Typen angegeben werden.

```
(++)    :: [a] -> [a] -> [a]
(.)     :: (b -> c) -> (a -> b) -> a -> c
const   :: a -> b -> a
elem    :: (Eq a) => a -> [a] -> Bool
filter  :: (a -> Bool) -> [a] -> [a]
fmap    :: (Functor f) => (a -> b) -> f a -> f b
foldr   :: (a -> b -> b) -> b -> [a] -> b
length  :: [a] -> Int
map     :: (a -> b) -> [a] -> [b]
product :: (Num a) => [a] -> a
sum     :: (Num a) => [a] -> a
zip     :: [a] -> [b] -> [(a, b)]
```

15.6 Monaden

Im folgenden wollen wir uns mit dem Konzept der *Monaden* und dessen Vorzügen für die funktionale Programmierung beschäftigen. Monaden haben ihren theoretischen Ursprung in der Kategorientheorie [Mac71, Pie91], einem sehr abstrakten, aber auch grundlegenden, Teilgebiet der Algebra.

Sei T eine Monade, und A ein Objekt, das wir uns als Grundmenge von Werten vorstellen können. Dann bezeichnet das Objekt TA, das A von T zugeordnet wird, die *Berechnungen* über A. Solche Berechnungen können durch eine Grundoperation der Monade miteinander komponiert werden.

Verschiedene Monaden können verschiedene Formen der Berechnung darstellen. So könnte zum Beispiel die Monade T *nichtdeterministische* Berechnungen, zusammen mit einer entsprechenden Komposition, abbilden, während T' Berechnungen mit *Seiteneffekten* kapselt, wieder mit einer Möglichkeit, solche Berechnungen hintereinander auszuführen. Monaden erlauben es also, von verschiedenen Formen der Berechnung und ihrer Komposition zu abstrahieren. Dies spielt eine große Rolle fürs elegante Programmieren in Haskell und soll an konkretem Beispielcode veranschaulicht werden.

15.6.1 `Maybe` und Berechnungen mit möglichem Fehlschlag

Der in Abbildung 15.1 dargestellte Stammbaum Alberts soll im folgenden in ein Haskell-Programm umgesetzt werden.[4] Wir definieren dazu zwei Funktionen `vater` und `mutter`, die einer `Person` den entsprechenden Elternteil zuordnen. Die Tatsache, dass manche Eltern im Stammbaum Alberts unbekannt sind, berücksichtigen wir mit dem polymorphen algebraischen Datentyp `Maybe`: ist zum Beispiel der Vater von `p` bekannt, sagen wir er ist `v`, dann hat `vater p` den Wert `Just v`, und sonst den Wert `Nothing`.

```
type Person = String

vater :: Person -> Maybe Person
```

[4] Das Beispiel ist inspiriert durch das Wikibook auf `http://en.wikibooks.org/wiki/Haskell/Understanding_monads`.

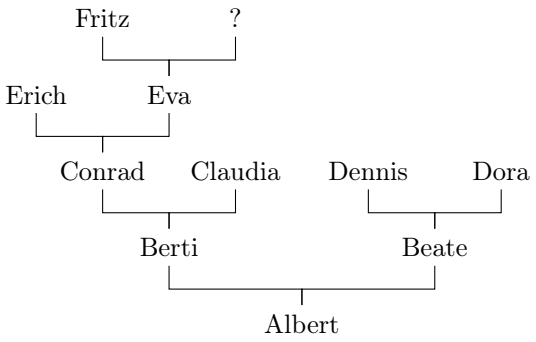


Abbildung 15.1: Alberts Stammbaum

```
vater "Albert" = Just "Berti"
vater "Berti"  = Just "Conrad"
vater "Beate"  = Just "Dennis"
vater "Conrad" = Just "Erich"
vater "Eva"    = Just "Fritz"
vater _        = Nothing

mutter :: Person -> Maybe Person
mutter "Albert" = Just "Beate"
mutter "Berti"  = Just "Claudia"
mutter "Beate"  = Just "Dora"
mutter "Conrad" = Just "Eva"
mutter _        = Nothing
```

Wir wollen nun eine Funktion `urgrossmuttervs` implementieren, welche, für gegebene Funktionen `vater` und `mutter`, jeder Person ihre Urgroßmutter (groß-)väterlicherseits zuordnet. Dabei müssen wir darauf achten, dass die Rückgabewerte von `vater` und `mutter` in `Maybe` gekapselt sind, wir müssen also für jeden Aufruf Pattern-Matching betreiben.

```
urgrossmuttervs :: Person -> Maybe Person
urgrossmuttervs p = case vater p of
                      Nothing -> Nothing
                      Just v  -> case vater v of
                                   Nothing -> Nothing
                                   Just gv -> mutter gv
```

Das ist sicherlich nicht die eleganteste Möglichkeit, eine solche Funktion zu implementieren! Man kann sich vorstellen, dass für kompliziertere Funktionen noch wesentlich mehr Code zum Pattern-Matching geschrieben werden muss, und damit die Leserlichkeit des Programmstücks noch weiter sinkt.

Um dieses Problem zu lösen, erinnern wir uns an unsere Erkenntnis aus der Einleitung: Monaden sind Abstraktionen von Berechnungen mit geeigneter Komposition. Im vorliegenden Fall sind die von uns betrachteten Berechnungen nichts weiter als Funktionen, die (mittels `Nothing`) fehlschlagen können, also Funktionen vom polymorphen Typ `a -> Maybe b`. In `urgrossmuttervs` machen wir dann letzten Endes nichts anderes, als die Funktionen `vater`, `vater`, und `mutter` von diesem Typ nacheinander auszuführen. Vielleicht können wir ja das dafür nötige Pattern-Matching direkt in die Komposition der Berechnungen verlagern – definieren wir also einen Operator, der das leisten soll.

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >>= f = Nothing
(Just w) >>= f = f w
```

Der Operator `>>=` (oft als *bind* bezeichnet) erwartet als Argumente einen Wert `ma` vom Typ `Maybe a`, beispielsweise das Ergebnis einer bereits vorangegangenen Berechnung, sowie eine Berechnung `f` mit Typ `a -> Maybe b`. Im Fall, dass `ma` den Wert `Nothing` hat, die bisherige Berechnung also fehlgeschlagen ist, wird diese Information einfach weitergereicht (Zeile 2). Andernfalls wird `f` auf den Wert in `ma` angewandt, um einen `Maybe b`-Wert zu erhalten (Zeile 3). Nun brauchen wir noch eine Funktion, mit der wir konstante

Berechnungen erzeugen können, also Berechnungen, die nichts weiter machen, als den angegebenen Wert zu berechnen. Diese Funktion heißt in Haskell `return`.[5]

```
return :: a -> Maybe a
return a = Just a
```

Im Falle von Berechnungen mit `Maybe` verpackt `return` einfach den übergebenen Wert in den Konstruktor `Just`. Damit können wir `urgrossmuttervs` folgendermaßen implementieren:[6]

```
urgrossmuttervs :: Person -> Maybe Person
urgrossmuttervs p = return p >>= vater >>= vater >>= mutter
```

Die Tatsache, dass `urgrossmuttervs` nur eine Komposition von drei Berechnungen ist, tritt nun klar zu Tage. Das notwendige Pattern-Matching findet im Hintergrund statt und verstellt nicht mehr den Blick aufs Wesentliche.

15.6.2 Nichtdeterminismus mit Listen

Nun kann Albert mit Hilfe von Haskell also in seinem Stammbaum seine Urgroßmutter berechnen lassen. Doch was ist, wenn er zum Beispiel an der Liste *aller* seiner Urgroßeltern interessiert ist? Versuchen wir, dies zu implementieren. Zuerst definieren wir eine Funktion `eltern`, mit der man die Eltern einer Person bestimmt.

```
eltern :: Person -> [Person]
eltern p = maybeToList (vater p) ++ maybeToList (mutter p)
  where maybeToList Nothing  = []
        maybeToList (Just a) = [a]
```

Dabei dient die Hilfsfunktion `maybeToList` dazu, `Maybe`-Werte in Listen umzuwandeln. Nun also die Funktion `urgrosseltern`:

```
urgrosseltern :: Person -> [Person]
urgrosseltern p = elternVon (elternVon (eltern p))
  where elternVon :: [Person] -> [Person]
        elternVon []     = []
        elternVon (e:es) = eltern e ++ elternVon es
```

Für ein Argument `p` berechnen wir erst die Liste der `eltern` von `p`. Die Hilfsfunktion `elternVon` wendet dann auf jedes Element dieser Liste die Funktion `eltern` an und konkateniert die Ergebnislisten, um die Liste der Großeltern von `p` zu erhalten. Durch zweimaliges Anwenden von `elternVon` werden so also die Urgroßeltern bestimmt.

Diese vorläufige Lösung stellt uns wiederum nicht wirklich zufrieden – immerhin versteckt sich hinter `elternVon` nichts weiter als die Konkatenation der Ergebnisse der elementweisen Anwendung der Funktion `eltern` auf die Eingabeliste. Das mag in diesem Fall noch ohne Weiteres von der Hand gehen, aber für jede ähnliche Funktion auf dem Stammbaum müsste auch eine neue Hilfsfunktion implementiert werden, obwohl deren Struktur durch `elternVon` letztlich schon vorgegeben ist.

Auch hier können wir uns mit dem Konzept der Monaden behelfen. Als Berechnungen fassen wir nun Funktionen mit dem polymorphen Typ `a -> [b]` auf. Oft werden solche Funktionen von Haskell-Programmierern als *nichtdeterministische* Funktionen bezeichnet, da sie für einen Eingabeparameter sozusagen mehrere verschiedene Ergebnisse liefern können. Die Komposition zweier solcher nichtdeterministischer Berechnungen wendet dann die zweite Berechnung auf jedes Ergebnis der ersten an, und konkateniert die so entstehenden Ergebnislisten:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
[]     >>= f = []
(x:xs) >>= f = f x ++ (xs >>= f)
```

Die Funktion `return` gibt in diesem Fall als Ergebnis eine einelementige Liste zurück.

[5] Man sollte sich vom Namen aber nicht verwirren lassen: `return` beendet nicht etwa wie in C die aktuelle Funktion!

[6] Wir definieren `>>=` als linksassoziativ, der Ausdruck ließe sich also äquivalent als `((return p >>= vater) >>= vater) >>= mutter` schreiben.

```
return :: a -> [a]
return a = [a]
```

Somit können wir `urgrosseltern` wesentlich prägnanter fassen:

```
urgrosseltern :: Person -> [Person]
urgrosseltern p = return p >>= eltern >>= eltern >>= eltern
```

Die Funktion ist also nichts weiter als die dreimalige monadische Komposition der Berechnung `eltern`. Für die Urgroßeltern von Albert ergibt sich die Berechnung

```
  urgrosseltern "Albert"
= return "Albert" >>= eltern >>= eltern >>= eltern
= ["Albert"] >>= eltern >>= eltern >>= eltern
= (eltern "Albert" ++ []) >>= eltern >>= eltern
= ["Berti", "Beate"] >>= eltern >>= eltern
= (eltern "Berti" ++ eltern "Beate" ++ []) >>= eltern
= ["Conrad", "Claudia", "Dennis", "Dora"] >>= eltern
= (eltern "Conrad" ++ eltern "Claudia" ++ eltern "Dennis" ++ eltern "Dora")
= ["Erich", "Eva"]
```

15.6.3 Zwischenspiel: Die Typklasse `Monad` und die `do`-Notation

Bisher haben wir also gesehen, wie Berechnungen mit möglichem Fehlschlag sowie mit Nichtdeterminismus in Haskell darstellbar sind und wie sie miteinander komponiert werden können. Es liegt nun nahe, von diesen konkreten Berechnungsarten zu abstrahieren und eine allgemeine Repräsentation von Monaden in Haskell zu finden. Dies erreichen wir mit den in Abschnitt 15.5 eingeführten *Typklassen.*

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Eine Monade ist also ein Typkonstruktor `m`, zusammen mit einem Operator `(>>=)` für die Komposition von Berechnungen und einer Funktion `return`, die konstante Berechnungen erzeugt.[7]

Zwei Instanzen der Typklasse `Monad`, mit den entsprechenden Implementierungen von `>>=` und `return`, haben wir schon kennen gelernt, nämlich die Typkonstruktoren `Maybe` sowie `[]` aus den obigen zwei Abschnitten.

Bevor wir uns weiteren Instanzen der `Monad`-Typklasse zuwenden, wollen wir aber noch eine hilfreiche Schreibweise für die Komposition monadischer Berechnungen kennen lernen. Nehmen wir an, zusätzlich zum Stammbaum aus dem obigen Abschnitt haben wir eine Funktion `rothaarig :: Person -> Bool`, die genau dann `True` als Ergebnis haben soll, wenn die übergebene Person rote Haare hat. Es soll nun eine Funktion `grosselternrh` implementiert werden, die die Liste der Eltern aller rothaarigen Eltern einer Person berechnet.

```
grosselternrh :: Person -> [Person]
grosselternrh p = eltern p >>= (\e ->
                  eltern e >>= (\ge ->
                  if rothaarig e then return ge
                                 else [] ))
```

Da die Entscheidung, ob ein `ge` ausgegeben wird, erst in der `if`-Klausel fällt, muss der Wert von `e` festgehalten werden. Dies erreichen wir hier mit anonymen Funktionen. Wesentlich eleganter ist allerdings die folgende Schreibweise in der sogenannten `do`-Notation:

[7] Dabei müssen für `m`, `(>>=)`, sowie für `return` gewisse Gesetze gelten, unter anderem Assoziativität von `(>>=)`, Verträglichkeit zwischen `(>>=)` und `return`, und ein Gesetz für die Verträglichkeit von Funktionskomposition mit `m`. Wir wollen diese hier aber nicht aufschreiben, sondern verweisen auf die Fachliteratur, vgl. z. B. [Mac71]. Die Typklasse umfasst außer den erwähnten noch einige andere Funktionen, die aber nicht wesentlich für die Funktionalität und das Verständnis sind. Auch hier wollen wir auf die Literatur verweisen, vgl. [OGS08].

```
grosselternrh' :: Person -> [Person]
grosselternrh' p = do e  <- eltern p
                      ge <- eltern e
                      if rothaarig e then return ge
                                     else []
```

In der Tat sind beide Varianten zueinander äquivalent. Die `do`-Notation, die uns sehr an den Stil der imperativen Programmierung erinnert, ist nichts weiter als *syntaktischer Zuckerguss* für die monadische Komposition von Berechnungen des obigen Typs.[8] Um zu zeigen, wie diese Notation *entzuckert* wird, müssen wir erst formal definieren, welche Form sie annehmen kann. Ein *`do`-Ausdruck* ist ein Ausdruck von der Form

```
do stmt1
   ...
   stmtk
   exp
```

wobei $\texttt{k} \geq 0$, *`exp`* ein Ausdruck ist, und jedes Statement *`stmti`*, $1 \leq \texttt{i} \leq \texttt{k}$, von einer der folgenden Formen ist:

- *`stmti = exp`*, wobei *`exp`* wieder ein beliebiger Ausdruck ist,[9]
- *`stmti = pat <- exp`*, dabei ist *`pat`* ein Pattern und *`exp`* ein Ausdruck,
- *`stmti = let decls`*, hierbei ist *`decls`* eine Liste von Deklarationen, wie wir sie auch aus herkömmlichen `let`-Ausdrücken kennen.

Für solche `do`-Ausdrücke können wir nun eine Funktion *desugar* definieren, welche sie in herkömmlichen, semantisch äquivalenten, monadischen Code umwandelt. Es sei

```
desugar(do exp)                 =   exp

desugar(do exp                      exp >>= (\_ ->
           stmt2                              desugar(do stmt2
           ...                  =                     ...
           stmtk)                                     stmtk)
                                              )
desugar(do x <- exp                 exp >>= (\x ->
           stmt2                              desugar(do stmt2
           ...                  =                     ...
           stmtk)                                     stmtk)
                                              )
desugar(do pat <- exp               let ok pat =  desugar( do stmt2
           stmt2                                               ...
           ...                  =                             stmtk )
           stmtk)                       ok _ = fail "Pattern not matched!"
                                    in exp >>= ok
desugar(do let decls                let decls in
           stmt2                        desugar(do stmt2
           ...                  =                  ...
           stmtk)                                  stmtk).
```

Dabei ist `exp` ein Ausdruck, *`stmt2`*, ..., *`stmtk`* Statements der obigen Form, und *`decls`* ein Block von Deklarationen. Bei der Umwandlung von Statements von der Form *`pat <- exp`*, bei der das Ergebnis der Berechnung *`exp`* an das Pattern *`pat`* gebunden werden soll, unterscheiden wir zwei Fälle. Im ersten Fall, in der dritten definierenden Gleichung, ist das Pattern eine Variable *`x`*, das Matchen mit solchen

[8] Aufgrund dessen, dass in imperativen Sprachen meist Semikolons für das Hintereinanderausführen von Statements verwendet werden, und da die Komposition von Berechnungen je nach Art der zugrunde liegenden Monade verschiedene Gestalt haben kann, sprechen Haskell-Programmierer bei Monaden auch oft von *programmierbaren Semikolons.*

[9] Insbesondere könnte `exp` wieder ein `do`-Ausdruck sein, man kann `do`-Ausdrücke also verschachteln!

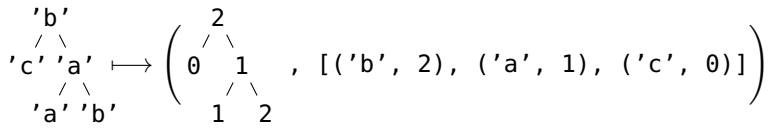


Abbildung 15.2: Anwendung von `intify` auf einen Baum `t :: Tree Char`

Patterns gelingt natürlich immer. Wir können also den Ausdruck *exp* einfach mittels `>>=` auswerten und das Ergebnis an die Variable *x* binden.

Im zweiten Fall, in der vierten Gleichung, nehmen wir an, dass *pat* keine bloße Variable, sondern ein zusammengesetztes Pattern ist. Hier müssen wir darauf achten, dass das Matchen eventuell fehlschlagen kann. In unserer syntaktischen Transformation führen wir daher eine neue Funktion `ok` ein, deren Bezeichner im Programm noch nicht verwendet wird. Dann können wir die schon bekannte Syntax zur Definition von Funktionen nutzen, um mit *pat* zu matchen. Für den Fall, dass die Unifikation fehlschlägt, wird die Hilfsfunktion `fail :: Monad m => String -> m a` aufgerufen. Diese bricht die laufende monadische Berechnung ab, und gibt womöglich – je nach verwendeter Monade – die übergebene Fehlermeldung aus.

Die `do`-Notation erlaubt es in bestimmten Fällen, monadischen Code wesentlich prägnanter zu fassen. Unter Verwendung von Funktionen höherer Ordnung und fortgeschrittenen Konzepten wie *applikativen Funktoren* kann aber auch oft auf den imperativen Stil der `do`-Blöcke verzichtet werden [MP08]. Im folgenden Abschnitt werden wir uns jedoch der `do`-Notation bedienen.

15.6.4 Berechnungen mit Zustand

Die effiziente algorithmische Lösung mancher Programmierprobleme erfordert Zugriff auf einen globalen Zustand, der während der Ausführung beliebig ausgelesen und verändert werden kann. Aufgrund von Haskells *referentieller Transparenz* können wir diesen nicht einfach wie in imperativen Programmiersprachen in einer globalen Variable ablegen und bei Belieben modifizieren, immerhin können wir in einem Haskell-Programm nur Datentypen sowie Funktionen über ihnen (und als Sonderfall Konstanten) definieren. Bisher haben wir uns in solchen Fällen damit beholfen, den nötigen Zustand in einem weiteren Parameter der entsprechenden Funktion mitzuführen und ihn explizit durchzuschleifen. Dass dies jedoch schnell unhandlich werden kann, zeigt das folgende Beispiel aus einem größeren Haskell-Projekt.

Es sei dabei ein binärer Baum mit Labels von einem beliebigen Typ `a`, dargestellt als Wert eines polymorphen algebraischen Datentyps

```
data Tree a = Leaf a | Branch a (Tree a) (Tree a)
```

gegeben. Im betrachteten Projekt müssen zeitgleich Millionen von solchen Bäumen möglichst speichereffizient verarbeitet werden. Um diese Anforderung zu erfüllen, auch wenn die Typvariable `a` mit einem besonders speicherhungrigen Datentyp instanziiert wird, soll eine Funktion

```
intify :: Eq a => Tree a -> (Tree Int, [(a, Int)])
```

implementiert werden, die jedem Baum `t` ein Tupel `(t', tab)` zuordnet, so dass

- `t'` von der selben Struktur wie `t` ist,
- die Labels aus `t` in `t'` durch Ganzzahlen ersetzt sind, so dass zwei Labels in `t` genau dann gleich sind, wenn die entsprechenden Zahlen in `t'` gleich sind,[10] und
- in `tab` genau diese Beziehung zwischen den Labels in `t` und `t'` festgehalten ist; `tab` gibt also an, welches Label durch welche Zahl ersetzt wurde.

Eine beispielhafte Anwendung von `intify` findet sich in Abbildung 15.2, hier wurden in `t` die Labels vom Typ `Char` durch Ganzzahlen ersetzt. Die Liste in der zweiten Komponente des Tupels gibt die Assoziation zwischen `Char`s und `Int`s an, so wurde zum Beispiel `'a'` durch `1` ersetzt.

Wir wollen nun versuchen, `intify` zu implementieren.

[10] In diesem Vergleich liegt die Notwendigkeit der Typklasse `Eq` in der Typsignatur von `intify`.

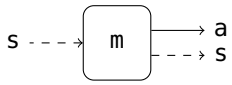


Abbildung 15.3: Eine Berechnung von `State s`

```
intify :: Eq a => Tree a -> (Tree Int, [(a, Int)])
intify t = (t', tab')
  where (t', _, tab') = intify' t 0 []
        intify' :: Eq a => Tree a -> Int -> [(a, Int)] -> (Tree Int, Int, [(a, Int)])
        intify' (Leaf x) i tab
          = case lookup x tab of
              Nothing -> (Leaf i, i+1, (x, i):tab)
              Just j  -> (Leaf j, i, tab)
        intify' (Branch x t1 t2) i0 tab0
          = case lookup x tab2 of
              Nothing -> (Branch i2 t1' t2', i2+1, (x,i2):tab2)
              Just j  -> (Branch j t1' t2', i2, tab2)
            where (t1', i1, tab1) = intify' t1 i0 tab0
                  (t2', i2, tab2) = intify' t2 i1 tab1

lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup a []   = Nothing
lookup a ((x1, x2):xs)
  | a == x1   = Just x2
  | otherwise = lookup a xs
```

Die Funktion `lookup` wurde dem *Prelude* entnommen und dient der Suche in Assoziationslisten vom Typ `[(a,b)]`. So gibt `lookup a tab` den ersten mit `a` assoziierten Wert in `tab`, verpackt in `Just`, zurück; findet es kein entsprechendes Tupel, dann jedoch `Nothing`.

Wie wir sehen, übernimmt den Großteil der Arbeit jedoch die Helferfunktion `intify'`. Diese hat zusätzlich zum Eingabebaum zwei weitere Parameter: einerseits eine möglicherweise unvollständige Zuordnungstabelle `tab :: [(a,Int)]`, aus der wir entnehmen können, welche Labels wir bereits welchen Ganzzahlen zugeordnet haben. Darüber hinaus speichern wir in einem weiteren `Int`-Parameter die nächste zu vergebende Zahl. Man beachte, dass sich diese Parameter auch im Rückgabetyp von `intify'` wieder finden müssen – so muss zum Beispiel nach einem rekursiven Aufruf der Funktion auf einen Teilbaum zurück kommuniziert werden, welche neuen Assoziationen zwischen Labels hinzugekommen sind; vergleiche dazu die `where`-Klauseln in Zeile 12–13.

Bei Übergabe eines `Leaf`s sucht `intify'` mittels `lookup` dessen Label in der mitgeführten Assoziationsliste `tab`. Sofern dieses gefunden wird, wird es mit dem entsprechenden Ganzzahlwert ersetzt, andernfalls wird der nächstmögliche Wert verwendet, und der Ausgabebaum, zusammen mit der um diese Information angereicherten Assoziationsliste und dem neuen nächstmöglichen Wert, zurückgegeben.

Bei `Branch`es wird ähnlich verfahren, zusätzlich müssen jedoch die enthaltenen Teilbäume verarbeitet und die dazu nötigen Zustandsinformationen explizit verwaltet werden. Im vorliegenden Fall einer *postorder traversal* heißt das: Erst wird der linke Teilbaum, ausgehend von der übergebenen Zustandsinformation, umgewandelt. Der dabei zurückgegebene Zustand wird zur Transformation des rechten Teilbaums verwendet, und der Ausgabezustand dieser Operation fließt ein in die Umbenennung des Labels des betrachteten `Branch`es. Der resultierende Zustand muss dann wiederum zurückgegeben werden.

Wollte man nun den Baum stattdessen in einer anderen Reihenfolge, zum Beispiel in *preorder*, durchlaufen, so müsste auch das explizite Durchreichen der Zustandsinformationen abgewandelt werden – hier wird offensichtlich, dass solcher Code schwer zu warten ist. Daher wollen wir diese lästige Zustandsverwaltung mithilfe von Monaden *implizit* machen – an welche Funktionen der Zustand weiter gegeben wird, ist dann durch die Reihenfolge der Komposition dieser Funktionen bestimmt.

Wie sehen die Berechnungen einer solchen Zustandsmonade aus? Ein Blick auf die Signatur von `intify'` hilft uns weiter: Eine Berechnung mit Zustand ist eine Funktion, die Zustand (sagen wir vom Typ `s`) entgegen nimmt, und einen Wert (Typ `a`), zusammen mit modifizierter Zustandsinformation (wieder

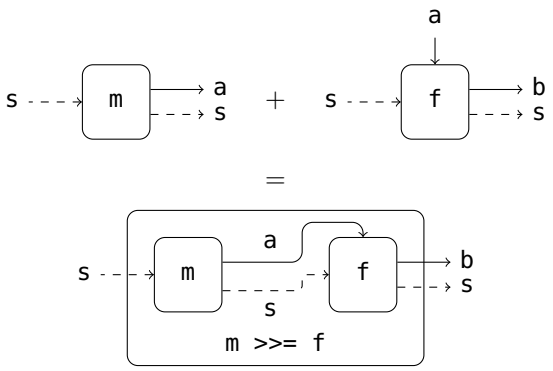


Abbildung 15.4: Komposition in `State s`

vom Typ `s`), ausgibt. Damit wir solche Funktionen später identifizieren können, verpacken wir sie im Konstruktor `State`, wir erhalten also

```
data State s a = State (s -> (a, s))
```

als Typ der Berechnungen der Monade. Zusätzlich definieren wir die Hilfsfunktion

```
runState :: State s a -> (s -> (a, s))
runState (State f) = f
```

um die in `State` enthaltene Funktion wieder extrahieren zu können.

Man beachte, dass `State` sowohl im Typ der Zustandsinformationen als auch im Typ des Rückgabewerts polymorph ist. Gegeben einen Typ `s`, verwenden wir im folgenden den partiell applizierten Typ `State s` als Monade, die Berechnungen mit implizitem Zustand aus `s` beschreibt. In Abbildung 15.3 ist eine Berechnung dieser Monade als Box mit durchgeschleiftem Zustand vom Typ `s` und Ausgabetyp `a` dargestellt.

Wenn wir zwei solcher Berechnungen verknüpfen, muss durch `>>=` im Hintergrund der Zustand weitergereicht werden – wie können wir das realisieren? Hier hilft uns ein Blick auf den allgemeinen Typ `m a -> (a -> m b) -> m b` von `>>=` im obigen Abschnitt weiter. Ersetzen wir `m` durch `State s`, erhalten wir

```
>>= :: State s a -> (a -> State s b) -> State s b
```

als Typ. Einen Ausdruck `m >>= f` können wir uns also vorstellen wie in Abbildung 15.4 veranschaulicht: `m` ist eine Berechnung der Monade mit Ausgabetyp `a`, und `f` eine Berechnung mit Ausgabetyp `b`, die zusätzlich abhängig von einer Eingabe vom selben Typ `a` ist – zur Komposition übergeben wir also einfach die Ausgabe `a` von `m` an `f` und schleifen das Zustandsverhalten durch `m` und `f a`. Wir erhalten somit

```
m >>= f = State (\s -> let (a, s') = runState m s
                       in  runState (f a) s'
                )
```

für die Komposition. Der Aufruf `return a` erzeugt eine Berechnung, die den aktuellen Zustand unverändert propagiert und als Ausgabe den Wert `a` hat:

```
return a = State (\s -> (a, s))
```

Wir brauchen noch zwei kleine Hilfsfunktionen. Die Funktion

```
get :: State s s
get = State (\s -> (s, s))
```

erlaubt in einer Sequenz von komponierten Berechnungen Zugriff auf den aktuellen Zustand, während dieser mittels

```
put :: s -> State s ()
put s = State (\_ -> ((), s))
```

modifiziert werden kann.

Wie nutzen wir diese Monade nun für unser Programmierproblem? Zuallererst führen wir einen polymorphen Typ `S` ein, der unsere Zustandsinformationen, die Assoziationsliste und die nächste zu vergebende Ganzzahl, enthalten soll:

```
data S a = S Int [(a, Int)]
```

Im folgenden werden wir uns also in der Monade `State (S a)` bewegen. Die folgende Berechnung `getKey` dient dazu, den bereits einem Label zugeordneten `Int`-Wert auszugeben, oder einen neuen zu vergeben und die Assoziationsliste dementsprechend zu aktualisieren:

```
getKey :: Eq a => a -> State (S a) Int
getKey a = do
  S i tab <- get
  case lookup a tab of
    Nothing -> do
      put (S (i + 1) ((a, i):tab))
      return i
    Just j -> return j
```

Die Funktion nutzt in Zeile 3 zuerst `get`, um den aktuellen Zustand auszulesen und sucht dann nach einem Eintrag für das Label `a`. Wird dieser nicht gefunden (Zeile 5), so wird mittels `put` der Zustand aktualisiert. Sowohl in diesem Fall, als auch in dem dass ein Eintrag gefunden wurde (Zeile 8), wird der entsprechende Wert durch `return` ausgegeben.

Die Berechnung `intifyWorker` nutzt `getKey`, um die Transformation eines Baums durchzuführen.

```
intifyWorker :: Eq a => Tree a -> State (S a) (Tree Int)
intifyWorker (Leaf x) = getKey x >>= (return . Leaf)
intifyWorker (Branch x t1 t2) = do
  t1' <- intifyWorker t1
  t2' <- intifyWorker t2
  i <- getKey x
  return (Branch i t1' t2')
```

Die Zustandsübergabe von `intifyWorker t1` nach `intifyWorker t2` und `getKey` ist nun ganz durch deren Reihenfolge gegeben. Wollte man den Baum in *preorder* durchlaufen, wäre die einzige notwendige Code-Änderung, `getKey` *vor* den rekursiven Aufrufen auszuführen.

Unsere verbesserte monadische Implementierung von `intify` ruft `intifyWorker` mit einem Startzustand auf, um nach Ausführung dessen Ausgabebaum, zusammen mit der extrahierten Assoziationsliste, auszugeben.

```
intifySt :: Eq a => Tree a -> (Tree Int, [(a, Int)])
intifySt t = (t', tab) where (t', S _ tab) = runState (intifyWorker t) (S 0 [])
```

15.6.5 Eingabe und Ausgabe mit `IO`

Ein Beispiel für die `do`-Notation haben wir bereits in unseren ersten Haskell-Programmen (am Beginn von Kapitel 15) kennen gelernt, wo wir damit in der Funktion `main` Werte einlasen und ausgaben. In der Tat versteckt sich hinter diesem `do` auch nur eine bestimmte Monade, die `IO`-Monade. Mit dieser lassen sich in Haskell Berechnungen mit *Seiteneffekten* ausdrücken, wie etwa Operationen zur Ein- und Ausgabe, zum Zugriff aufs Dateisystem und auf Rechnernetze, Prozess- und Speichermanagement und vielem mehr.

Grob kann man sich die `IO`-Monade als eine Verallgemeinerung der Zustandsmonade vorstellen: Machte `State` aber nur einen bestimmten Typ von Zustand in der laufenden Berechnung implizit, so kapselt `IO` den *gesamten* aktuellen Zustand des Rechners und seiner Umwelt. Dieser Zustand umfasst, unter anderem, die aktuelle Systemzeit, den derzeitigen Eingabepuffer, die auf Festspeichermedien abgelegten Daten, gepufferte IP-Pakete, den Zustand der Peripherie des Rechners (z.B. eines Druckers) usw.

Die Typen der Berechnungen dieser Monade stellen dabei zusammen mit Haskells Typsystem sicher, dass solche Berechnungen mit Seiteneffekten vom seiteneffektfreien Rest des Programms klar getrennt sind.

Insbesondere ist es nicht möglich, aus einer Funktion heraus, welche keine `IO`-Berechnung ist, eine Funktion mit Seiteneffekten aufzurufen. Wenn wir Eigenschaften eines Haskellprogramms beweisen wollen, ist das ein großer Vorteil, denn beim Beweis für den rein funktionalen Teil des Programms sind wir so nicht gezwungen, den Zustand des Rechners mit einzubeziehen. Wir müssen für diesen Teil also nur Gleichungen umformen, wie zum Beispiel in einem Induktionsbeweis, siehe Abschnitt 15.7. Beweise für Code aus `IO` (und allgemeiner, für imperative Programme) sind im Vergleich dazu wesentlich verzwickter. Daher ist es gute Programmierpraxis, möglichst viel Funktionalität des Programms in Funktionen außerhalb der `IO`-Monade zu verlagern.

In der `IO`-Monade finden wir, neben zahlreichen anderen, die folgenden Funktionen:

- `putChar :: Char -> IO ()` und `putStr :: String -> IO ()` zur Ausgabe von Zeichen und Strings,
- `getChar :: IO Char` und `getLine :: IO String` zum Einlesen von einzelnen Zeichen und ganzen Zeilen,
- `getContents :: IO String` zum Einlesen der gesamten Standardeingabe,
- `readFile :: FilePath -> IO String` und `writeFile :: FilePath -> String -> IO ()` zum Lesen und Schreiben von Dateien,
- `getCurrentTime :: IO UTCTime` zur Bestimmung der aktuellen Zeit sowie
- `sendTo :: HostName -> PortID -> String -> IO ()` und `recvFrom :: HostName -> PortID -> IO String` zum Verschicken und Empfangen von Nachrichten über ein Rechnernetz.

Wir verweisen auf [OGS08] für eine Einführung in die Systemprogrammierung mit Haskell, und auf die Seite `http://haskell.org/hoogle/` für die Dokumentation weiterer nützlicher Funktionen (nicht nur) aus `IO`.

15.7 Beweis von Programmeigenschaften

In diesem Kapitel werden wir für funktionale Programme individuelle Eigenschaften beweisen; dabei spielt oft die Induktion eine entscheidende Rolle.

15.7.1 Beweise von elementaren Eigenschaften

Beispiel 15.29. Gegeben sei die Funktion `swap`, die die Reihenfolge der Elemente eines Tupels vertauscht.

```
swap :: (Int, Int) -> (Int, Int)
swap (x, y) = (y, x)
```

Nun ist es sehr leicht zu zeigen, dass das zweimalige Anwenden von `swap` die Identität ist.

```
  swap (swap (x, y))
= swap (y, x)
= (x, y)
```
□

15.7.2 Beweis durch Induktion über Listen

Listen sind in der funktionalen Programmierung eine zentrale Datenstruktur. Betrachten wir den Basistyp `T` und die Liste `xs :: [T]`. Dann kann `xs` eine Eigenschaft haben, z. B. dass ihre Länge geradzahlig ist, oder dass `rev (rev xs) = xs`, wobei die Funktion `rev` (reverse) die Reihenfolge der Elemente der Argumentliste umdreht. Wenn man zeigen will, dass eine Eigenschaft für jede Liste `xs :: [T]` gilt, kann man das Prinzip der Induktion auf Listen anwenden. Dabei handelt es sich um eine Verallgemeinerung des Prinzips der vollständigen Induktion über den natürlichen Zahlen:

Prinzip der Induktion über Listen:

Sei `T` ein Typ und sei P: `[T]` $\to \{0,1\}$ eine Eigenschaft (oder: Prädikat). Für `xs :: [T]` sagen wir, P(`xs`) *gilt*, wenn P(`xs`) $= 1$, sonst *gilt* P(`xs`) *nicht*.

Wenn

- (Induktionsanfang:) das Prädikat P für die leere Liste gilt (d.h. P(`[]`) gilt) und
- (Induktionsschritt:) für jede Liste `xs'` und jeden Eintrag `x` die folgende Implikation gilt: wenn P(`xs'`) gilt (Induktionsvoraussetzung) , dann gilt auch P(`x : xs'`),

dann gilt P für jede Liste, d. h. P(`xs`) gilt für jedes `xs :: [T]`.

Im Anhang B.7 wird die Korrektheit dieses Prinzips formal bewiesen. Das Prinzip der vollständigen Induktion ergibt sich aus dem Prinzip der Induktion über Listen, indem man die natürliche Zahl n mit der Liste `[x, ..., x]` mit n mal `x` identifiziert (für einen beliebigen Wert `x`), also wird insbesondere 0 mit der leeren Liste identifiziert.

Beispiel 15.30. Als Beispiel für die Anwendung des Prinzips der Induktion über Listen betrachten wir die beiden einfachen Listen verarbeitende Funktionen `sumList` und `double`.

```
sumList :: [Int] -> Int
sumList []       = 0
sumList (x : xs) = x + sumList xs

double :: [Int] -> [Int]
double []       = []
double (x : xs) = (2 * x) : double xs
```

Nun gilt offensichtlich (wegen der Distributivität der Multiplikation über die Addition im Ring der ganzen Zahlen) folgender Zusammenhang zwischen diesen Funktionen:

Für jede Liste `xs :: [Int]` gilt:

$$\texttt{sumList (double xs)} = 2 * \texttt{sumList xs}.$$

Dann formulieren wir die Eigenschaft P: `[Int]` $\to \{0,1\}$, wobei P(`xs`) $= 1$ genau dann gilt, wenn `sumList (double xs) = 2 * sumList xs`, und beweisen mit Hilfe des Prinzips der Induktion über Listen, dass P(`xs`) für jedes `xs :: [Int]` gilt:

Induktionsanfang: `xs = []`

```
  sumList (double xs)
= sumList (double [])
= sumList []
= 0
= 2 * 0
= 2 * (sumList [])
= 2 * (sumList xs)
```

Induktionsvoraussetzung: Sei `xs' :: [Int]` eine Liste. Wir nehmen an, dass P(`xs'`) gilt.

Induktionsschritt: Für `xs = (x : xs')` und `x :: Int`

```
  sumList (double xs)
= sumList (double (x : xs'))
= sumList ((2 * x) : double xs')
= (2 * x) + sumList (double xs')
= (2 * x) + (2 * sumList xs')                    (laut I. V.)
= 2 * (x + sumList xs')
= 2 * sumList (x : xs')
= 2 * sumList xs                                          □
```

Beispiel 15.31. Als nächstes wollen wir die Richtigkeit des bereits benutzten Programms `iSort` (siehe Seite 165) mit Hilfe der Induktion über Listen beweisen.

Sei `xs :: [Int]`. Wir definieren nun die folgende Aussage:

$$P(\texttt{xs}) \text{ gilt genau dann, wenn für jedes } i \text{ mit } 0 \leq i < |\texttt{xs}| - 1 \text{ gilt: } \texttt{xs}(i) \leq \texttt{xs}(i+1),$$

d. h. $P(\texttt{xs})$ ist genau dann wahr, wenn die Liste `xs` sortiert ist.

Hierbei ist $|\texttt{xs}|$ die Länge der Liste `xs` und $\texttt{xs}(i)$ das i-te Listenelement von `xs` (beginnend mit $i = 0$); im folgenden werden wir noch die Vereinbarungen $\max(\texttt{xs}) = \max\{\texttt{xs}(0), \ldots, \texttt{xs}(|\texttt{xs}| - 1)\}$ und $\min(\texttt{xs}) = \min\{\texttt{xs}(0), \ldots, \texttt{xs}(|\texttt{xs}| - 1)\}$ benutzen, wobei $\max(\texttt{[]}) < \texttt{x}$ für jedes `x :: Int` gelten soll.

Behauptung: Für jedes `xs :: [Int]` gilt $P(\texttt{iSort xs})$.

Wir wollen den Beweis unter Nutzung dreier Hilfsbehauptungen durchführen.

- **Hilfsbehauptung 1 (HB1):** Seien `ys, zs :: [Int]`, `x :: Int`. Wenn $\max(\texttt{ys}) < \texttt{x}$, dann gilt auch $Q_1(\texttt{ys})$ mit $Q_1(\texttt{ys}) :=$ (`ins x (ys ++ zs) = ys ++ ins x zs`).

  Beweis von HB1 durch Induktion über `ys`:

  Induktionsanfang:

  Für `ys = []` gilt: `ins x ([] ++ zs) = ins x zs = [] ++ ins x zs`

  Induktionsvoraussetzung:

  Sei `ys' :: [Int]` eine Liste, für die HB1 gilt.

  Induktionsschritt: `ys = (y : ys')`

  Sei $\max(\texttt{y : ys'}) < \texttt{x}$, dann gilt:

  ```
    ins x ((y : ys') ++ zs)
  = ins x (y : (ys' ++ zs))
  = y : ins x (ys' ++ zs)
  = y : (ys' ++ ins x zs)                    (laut I. V.)
  = (y : ys') ++ ins x zs
  ```

- **Hilfsbehauptung 2 (HB2):** Für jedes `xs :: [Int]` gilt $Q_2(\texttt{xs})$ mit

  $$Q_2(\texttt{xs}) := (\text{für jedes } \texttt{x :: Int}, \text{ wenn } \texttt{x} \leq \min(\texttt{xs}), \text{ dann gilt: } \texttt{ins x xs = (x : xs)}) .$$

  Beweis von HB2 durch Induktion über `xs`.

- **Hilfsbehauptung 3 (HB3):** Für jedes `xs :: [Int]` gilt $Q_3(\texttt{xs})$ mit

  $$Q_3(\texttt{xs}) := (\text{für jedes } \texttt{x :: Int} \text{ gilt: } P(\texttt{xs}) \Longrightarrow P(\texttt{ins x xs})) .$$

  Beweis von HB3: Seien `xs :: [Int]`, `x :: Int` und gelte $P(\texttt{xs})$.
  Dann existieren `ys, zs :: [Int]` derart, dass:
  – `xs = ys ++ zs`
  – $\max(\texttt{ys}) < \texttt{x}$ und $\texttt{x} \leq \min(\texttt{zs})$
  – $P(\texttt{ys})$ und $P(\texttt{zs})$ gelten.

  Nun gilt: `ins x xs = ins x (ys ++ zs)` $\overset{\text{HB1}}{=}$ `ys ++ ins x zs` $\overset{\text{HB2}}{=}$ `ys ++ [x] ++ zs`.
  Da $P(\texttt{ys})$ und $P(\texttt{zs})$ gelten und $\max(\texttt{ys}) < \texttt{x}$ und $\texttt{x} \leq \min(\texttt{zs})$, folgt dass $P(\texttt{ys ++ [x] ++ zs})$ gilt und damit auch $P(\texttt{ins x xs})$.

Setzen wir nun unseren Hauptbeweis durch Induktion über `xs` fort:

Induktionsanfang:

Sei `xs = []`, so gilt $P(\texttt{iSort []})$ $(= P(\texttt{[]}))$.

Induktionsvoraussetzung:

Sei `xs' :: [Int]` eine Liste so dass $P(\texttt{iSort xs'})$ gilt.

Induktionsschritt: `xs = (x : xs')`

Aus der Induktionsvoraussetzung folgt nach HB3, dass auch $P(\texttt{ins x (iSort xs')})$ gilt. Nach Definition von `iSort` folgt schließlich die Gleichheit $P(\texttt{ins x (iSort xs')}) = P(\texttt{iSort (x : xs')})$ und damit die Behauptung. □

Beispiel 15.32. Ein weiteres Beispiel zeigt, dass es auch hier zweckmäßig ist, zunächst eine Hilfsbehauptung zu beweisen. Gegeben sei die Funktion `rev`, die die Reihenfolge der Elemente einer Liste umdreht:

```
rev :: [a] -> [a]
rev []       = []
rev (x : xs) = rev xs ++ [x]
```

Offensichtlich muss für jede Liste `xs` gelten: `rev (rev xs)` = `xs`. Dazu beweisen wir zunächst die folgende Behauptung:

(∗) Für jede Liste `xs :: [a]` und jedes Listenelement `x :: a` gilt: `rev (xs ++ [x])` = `x : rev xs`.

Beweis von (∗) durch Induktion über Listen:

Induktionsanfang:

```
  rev ([] ++ [x])
= rev [x]
= rev (x : [])
= rev [] ++ [x]
= [] ++ [x]
= [x]
= x : []
= x : rev []
```

Induktionsvoraussetzung: Sei `xs' :: [a]` eine Liste, für die (∗) gilt.

Induktionsschritt: `xs = (x' : xs')`

```
  rev ((x' : xs') ++ [x])
= rev (x' : (xs' ++ [x]))
= rev (xs' ++ [x]) ++ [x']
= (x : rev xs') ++ [x']                                  (laut I. V.)
= x : (rev xs' ++ [x'])
= x : rev (x' : xs')
```

Wir haben beim Induktionsschritt einige weitere offensichtliche Eigenschaften von Listen benutzt, wie z. B.: für jede Liste `xs'` und jede Einträge `x, x'` gilt: `(x' : xs') ++ [x]` = `x' : (xs' ++ [x])`.

Nun können wir die Eigenschaft `rev (rev xs)` = `xs` beweisen:

Induktionsanfang:

```
  rev (rev [])
= rev []
= []
```

Induktionsvoraussetzung: Sei `xs' :: [a]` eine Liste, für die gilt: `rev (rev xs')` = `xs'`.

Induktionsschritt: `xs = (x : xs')`

```
  rev (rev (x : xs'))
= rev (rev xs' ++ [x])                          (wegen Def. von rev)
= x : rev (rev xs')                                     (wegen ∗)
= x : xs'                                           (wegen I. V.)
= xs                                                            □
```

15.7.3 Beweis durch strukturelle Induktion

Beim Beweis von Eigenschaften von Funktionen, die über algebraischen Datentypen definiert sind, müssen wir das Prinzip der Induktion noch einmal verallgemeinern.

Prinzip der strukturellen Induktion:

Sei P: `T` $\to \{0, 1\}$ eine Eigenschaft des algebraischen Datentyps `T`.

Wenn

- (Induktionsanfang:) für jeden Konstruktor `D` von `T`, für den *kein* Argumenttyp gleich `T` ist, das Prädikat P gilt und

- (Induktionsschritt:) folgende Implikation für jeden k-stelligen Konstruktor `C` mit Ergebnistyp `T` mit $k \geq 1$ und alle Werte `t1`, ..., `tk` (vom passenden Argumenttyp) gilt: wenn P für jedes `ti` vom Typ `T` gilt (Induktionsvoraussetzung), dann gilt P für `(C t1 ... tk)`,

dann gilt P für jeden Wert von `T`.

Beispiel 15.33. `data Tree = C Int Tree Tree | A`
Der Konstruktor `C` ist dreistellig. Beim Induktionsschritt nehmen wir nur für sein zweites und drittes Argument an, dass `P` gilt, denn nur diese beiden Argumente sind vom Typ `Tree`. □

Im Anhang B.7 wird die Korrektheit des Prinzips der strukturellen Induktion formal bewiesen.

Beispiel 15.34. Betrachten wir folgende Funktionen:

```
map :: (a -> a) -> [a] -> [a]
map f []       = []
map f (x : xs) = f x : map f xs

data Tree a = Nil | Node a (Tree a) (Tree a)

mapTree :: (a -> a) -> Tree a -> Tree a
mapTree f Nil            = Nil
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1) (mapTree f t2)

collapse :: Tree a -> [a]
collapse Nil            = []
collapse (Node x t1 t2) = collapse t1 ++ [x] ++ collapse t2
```

Die Funktion `mapTree` verallgemeinert die Funktion `map` von Listen auf algebraische Datentypen. Offensichtlich gilt für jeden Wert `t :: Tree a` die folgende Eigenschaft:

$$P(\texttt{t}) = 1 \quad \text{gdw} \quad \text{für jede Funktion } \texttt{f :: a -> a} \text{ gilt: } \texttt{map f (collapse t) = collapse (mapTree f t)}$$

welche wir jetzt durch strukturelle Induktion beweisen (Abbildung 15.5 zeigt ein Beispiel). Dabei benutzen wir die Eigenschaft, dass für jede Funktion `g` und jede der Listen `ys`, `zs` gilt:

```
map g (ys ++ zs) = map g ys ++ map g zs .
```

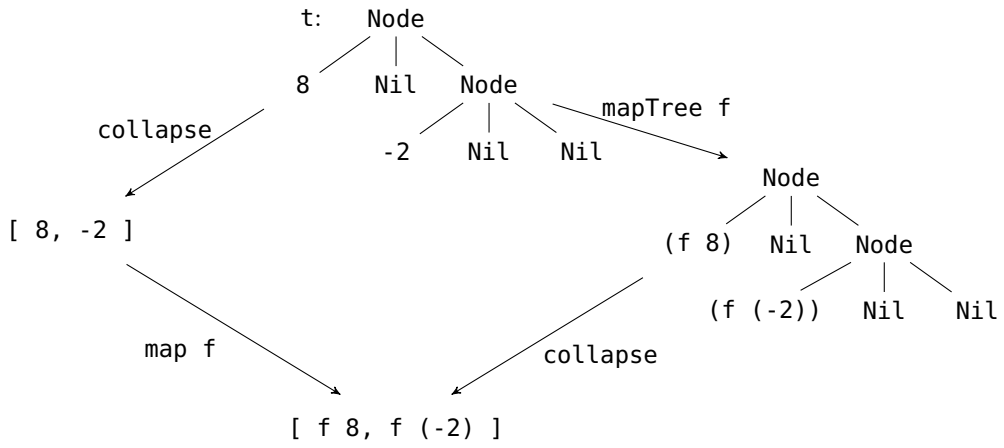


Abbildung 15.5: Beispiel dafür, dass `map f (collapse t) = collapse (mapTree f t)` für jede Funktion `f :: a -> a` und jeden Baum `t :: Tree a`, wobei `a` mit `Int` instanziiert ist.

Induktionsanfang:

```
  map f (collapse Nil)
= map f []
= []
= collapse Nil
= collapse (mapTree f Nil)
```

Induktionsvoraussetzung: Seien `t1, t2 :: Tree a` beliebige Bäume. Wir nehmen an, dass für jede Funktion `f :: a -> a` gilt:

`map f (collapse t1) = collapse (mapTree f t1)` und
`map f (collapse t2) = collapse (mapTree f t2)`

Induktionsschritt:

```
  map f (collapse (Node x t1 t2))
= map f (collapse t1 ++ [x] ++ collapse t2)
= map f (collapse t1) ++ map f [x] ++ map f (collapse t2)
= collapse (mapTree f t1) ++ [f x] ++ collapse (mapTree f t2)
= collapse (Node (f x) (mapTree f t1) (mapTree f t2))
= collapse (mapTree f (Node  a t1 t2))
```

□

15.8 Der λ-Kalkül

In diesem Abschnitt wollen wir eine weitere funktionale Programmiersprache einführen, den λ-Kalkül. Die Programme dieser Sprache heißen λ-Terme. Ein Beispiel für einen λ-Term ist:

$$\lambda x.\,((+x)\,x)$$

Dieser Term beschreibt eine anonyme Funktion, welche für eine gegebene Zahl n den Wert $n+n$ berechnet. In ihm kommt die Variable x, das Symbol $+$ (welches wir als Addition interpretiert haben), die Applikation (z.B. wird die unterversorgte Addition $(+x)$ auf x angewendet), und die Abstraktion $\lambda x.t$ vor. Schauen wir uns als Nächstes den syntaktischen Aufbau der λ-Terme formal an.

Definition 15.35. Sei Σ eine Menge von Symbolen mit $X \cap \Sigma = \emptyset$. Die Menge der λ_Σ-*Terme*, bezeichnet durch $\lambda(\Sigma)$, ist die kleinste Menge W von Wörtern über $X \cup \Sigma \cup \{(,), \lambda, .\}$, so dass gilt:

(i) $X \cup \Sigma \subseteq W$ („Atome“).

(ii) Wenn t_1 und t_2 in W, dann $(t_1\ t_2) \in W$ („Applikation“).

(iii) Wenn t in W und $x \in X$, dann $(\lambda x.t) \in W$ („Abstraktion“). □

Bei der Applikation wird ein λ-Term t_1 auf einen zweiten λ-Term t_2 angewendet. Hiermit kann die Anwendung einer Funktion (repräsentiert durch t_1) auf ein Argument (repräsentiert durch t_2) beschrieben werden. Die Abstraktion erlaubt die Bildung von neuen Funktionen aus bereits vorliegenden λ-Termen. Ist z. B. t ein λ-Term und x eine Variable, so heißt $(\lambda x.t)$ die „Abstraktion von t nach x“ und bezeichnet die einstellige Funktion mit der Variablen x.

Beispiel 15.36. Sei $\Sigma = \{1, 3, +, *, squ, a, f\}$ eine Menge von Symbolen. Dann sind folgende Wörter λ_Σ-Terme:

$$3, \qquad ((+3)1), \qquad (\lambda x.((*1)x)) \text{ und} \qquad ((\lambda x.((*x)a))(f(squ\ 3))).$$

Aber auch (xx) und $(x(\lambda y.*))$ sind λ_Σ-Terme. Die letzten beiden λ_Σ-Terme kann man nicht direkt interpretieren. □

Schreibkonventionen. Zur Vermeidung von unübersichtlichen Klammerungen vereinbaren wir,

- dass die Applikation linksassoziativ ist, d. h. für $t, t_1, t_2, \ldots, t_n \in \lambda(\Sigma)$ kann der Term $(\ldots((t\ t_1)t_2)\ldots t_n)$ durch $t\ t_1 t_2 \ldots t_n$ abgekürzt werden,
- dass mehrere Abstraktionen $(\lambda x_1.(\lambda x_2.\ldots.(\lambda x_n.t)\ldots))$ durch $(\lambda x_1 \ldots x_n.t)$ abgekürzt werden können, und
- dass die Applikation Priorität vor der Abstraktion hat, d. h. $(\lambda x.xy)$ kürzt den Term $(\lambda x.(xy))$ ab und *nicht* $((\lambda x.x)y)$.

Also z. B. statt $((+3)1)$ schreiben wir jetzt $+\ 3\ 1$, und statt $(\lambda x.(\lambda y.((yx)(\lambda z.z))))$ jetzt: $(\lambda xy.yx(\lambda z.z))$.

Der λ-Term $(\lambda x.+\ x\ x)$ kann durchaus mit der C-Funktion

```
int malZwei(int x)
{ return x+x; }
```

verglichen werden.

Nun wollen wir ein Verfahren beschreiben, mit dessen Hilfe wir einen λ-Term „ausrechnen“ können. Dazu definieren wir eine binäre Rechenrelation $\Rightarrow \subseteq \lambda(\Sigma) \times \lambda(\Sigma)$ auf der Menge $\lambda(\Sigma)$ der λ-Terme. Wenn dann ein konkreter λ-Term vorliegt, z. B. $(\lambda x.\,(+x)\,x)\,3$, dann können wir einen Rechenschritt ausführen:

$$(\lambda x.\,(+x)\,x)\,3 \ \Rightarrow (+3)\,3$$

Hier wird also jedes freie Vorkommen von x in $(+x)\,x$ durch 3 ersetzt. Eine Rechnung mittels $\Rightarrow$ kann auch mehrere Schritte umfassen, z.B.

$$((\lambda x.\,(\lambda y.\,(+x)\,y))\,3)\,4 \Rightarrow (\lambda y.\,(+3)\,y)\,4 \Rightarrow (+3)\,4.$$

Für die formale Definition von $\Rightarrow$ müssen wir als erstes die Begriffe freies Vorkommen einer Variablen und, dual dazu, gebundenes Vorkommen einer Variable klären.

Definition 15.37. Sei $t \in \lambda(\Sigma)$. Die Mengen der *freien Vorkommen von Variablen in* t und der *gebundenen Vorkommen von Variablen in* t, bezeichnet durch $FV(t)$ bzw. $GV(t)$, sind induktiv über den Aufbau von t definiert.

(i) Falls $t = x$ für ein $x \in X$, dann $FV(t) = \{x\}$ und $GV(t) = \emptyset$;
falls $t = \sigma$ für ein $\sigma \in \Sigma$, dann $FV(t) = GV(t) = \emptyset$.

(ii) Falls $t = (t_1 t_2)$ für $t_1, t_2 \in \lambda(\Sigma)$, dann $FV(t) = FV(t_1) \cup FV(t_2)$ und $GV(t) = GV(t_1) \cup GV(t_2)$.

(iii) Falls $t = (\lambda x.t')$ für $t' \in \lambda(\Sigma)$, dann $FV(t) = FV(t') \setminus \{x\}$ und $GV(t) = GV(t') \cup \{x\}$. □

Die Menge $Var(t)$ der Variablen eines λ-Terms t ist die Vereinigung der Mengen $FV(t)$ und $GV(t)$.

Beispiel 15.38. Im λ_Σ-Term $(\lambda x.xy(\lambda z.y))$ kommt y zweimal frei vor; x und z kommen gebunden vor. Manchmal sprechen wir auch etwas salopp von „freier“ oder „gebundener“ Variable. Dabei ist immer Vorsicht geboten, denn eine Variable kann in einem λ-Term sowohl frei als auch gebunden vorkommen, z. B. im Term $(\lambda x.y(\lambda y.xy))$ kommt y einmal frei und einmal gebunden vor. □

Definition 15.39. Sei $t \in \lambda(\Sigma)$. Dann heißt t *geschlossener λ-Term*, falls $FV(t) = \emptyset$. Ein geschlossener λ-Term heißt auch *Kombinator*. □

Wir haben am Beispiel gesehen, dass die Rechenrelation $\Rightarrow$ auf dem Ersetzen von freien Vorkommen von Variablen durch einen λ-Term basiert.

Beispiel 15.40. Sei die Funktion $f\colon \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ beschrieben durch den λ-Term $(\lambda x.(\lambda y.+xy))$. Dann sollte die Applikation $(f\ 3)$ zu $(\lambda y.+\ 3\ y)$ reduziert werden können. Dabei haben wir den Rumpf von f genommen, also $(\lambda y.+xy)$, und jedes freie Vorkommen von x durch 3 ersetzt. Diese Substitution einer Variablen durch einen Term formalisieren wir als nächstes. □

Definition 15.41. Sei $x \in X$ und seien $t, s \in \lambda(\Sigma)$. Die *Substitution von s für jedes freie Vorkommen von x in t*, bezeichnet durch $t[x/s]$, ist induktiv über den Aufbau von t definiert.

(i) Falls $t = x$, dann $t[x/s] = s$;
falls $t = y$ für ein $y \in X$ mit $y \neq x$, dann $t[x/s] = t$;
falls $t \in \Sigma$, dann $t[x/s] = t$.

(ii) Falls $t = (t_1\ t_2)$ für $t_1, t_2 \in \lambda(\Sigma)$, dann $t[x/s] = (t_1[x/s]\ t_2[x/s])$.

(iii) Falls $t = (\lambda y.t')$ für $y \in X$ und $t' \in \lambda(\Sigma)$ und

a. falls $x = y$, dann $t[x/s] = t$,

b. falls $x \neq y$, dann $t[x/s] = (\lambda y.t'[x/s])$. □

Beispiel 15.42. $((+\ y)(+x((\lambda y.y)a)))[y/s] = ((+\ y)[y/s]\ (+x((\lambda y.y)a))[y/s]) = ((+s)(+x((\lambda y.y)a)))$
Oder: Sei $t = (\lambda x.y(\lambda y.xy))$ und sei s ein λ_Σ-Term. Dann ist $t[y/s] = (\lambda x.s(\lambda y.xy))$.
Oder: Sei $t = (\lambda x.z)$, dann ist $t[z/x] = (\lambda x.x)$. Hier wird also ein freies Vorkommen der Variablen z in ein gebundenes Vorkommen der Variablen x umgewandelt. Das müssen wir – wie wir gleich sehen werden – vermeiden. □

Diese Substitution wird in der β-Reduktion von λ-Termen benutzt.

Definition 15.43. Die *β-Reduktion*, bezeichnet durch $\rightarrow_\beta$, ist die binäre Relation auf $\lambda(\Sigma)$, die für alle $t, s \in \lambda(\Sigma)$ wie folgt definiert ist: Wenn $GV(t) \cap FV(s) = \emptyset$, dann $(\lambda x.t)\ s \rightarrow_\beta t[x/s]$. □

Die β-Reduktion beschreibt also die Reduktion der Applikation $(\lambda x.t)\ s$ einer λ-Abstraktion $(\lambda x.t)$ auf einen Argumentterm s. Diese Reduktion darf nur dann erfolgen, wenn keine der freien Variablen von s gleichzeitig eine gebundene Variable von t ist, denn sonst könnte die freie Variable zu einer gebundenen werden.

Beispiel 15.44. Sei $\Sigma = \{3, a\}$. Dann gilt

$$(\lambda x.\underbrace{+\ x\ 3)}_{t}\ \underbrace{(\lambda z.a)}_{s} \rightarrow_\beta \underbrace{+\ (\lambda z.a)\ 3}_{t[x/s]}$$

denn $GV(t) = FV(s) = \emptyset$. □

Dagegen lässt sich die Applikation

$$r = (\lambda x.\underbrace{(\lambda y.+\ x\ y))}_{t}\ \underbrace{y}_{s}$$

nicht zu

$$(\lambda y.+\ y\ y)$$

reduzieren, da $FV(s) = \{y\}$ und $GV(t) = \{y\}$ nicht disjunkt sind. In der Tat würde der Verzicht auf die Bedingung $(GV(t) \cap FV(s) = \emptyset)$ dazu führen, dass zwei *verschiedene* Ergebnisse berechnet werden können. Das zeigt das folgende Beispiel, in dem wir den Term t benutzen.

Beispiel 15.45. Für den λ-Term $(\lambda y.r\ 3)\ 4$ mit $r = (\lambda x.(\lambda y.+\ x\ y))\ y$ gibt es dann zwei verschiedene Ableitungen:

$$(\lambda y.\ \underbrace{(\lambda x.(\lambda y.+\ x\ y))\ y}_{r}\ 3)\ 4 \not\rightarrow_\beta (\lambda y.\ (\lambda y.+\ y\ y)\ 3)\ 4 \rightarrow_\beta (\lambda y.+\ y\ y)\ 3 \rightarrow_\beta +\ 3\ 3 \text{ und}$$

$$(\lambda y.\ \underbrace{(\lambda x.(\lambda y.+\ x\ y))\ y}_{r}\ 3)\ 4 \rightarrow_\beta (\lambda x.\ (\lambda y.+\ x\ y))\ 4\ 3 \rightarrow_\beta (\lambda y.+\ 4\ y)\ 3 \rightarrow_\beta +\ 4\ 3\ .$$

Die Analyse beider Ableitungen zeigt, dass in der ersten die Umwandlung der freien Variablen y in eine gebundene ungerechtfertigt ist; und genau das wird durch die Bedingung $(GV(t) \cap FV(s) = \emptyset)$ ausgeschlossen. □

Die Bedingung $(GV(t) \cap FV(s) = \emptyset)$ an die β-Reduktion ist also nicht immer erfüllt. Sie lässt sich aber mit Hilfe der α-Konversion herstellen. Die α-Konversion entspricht der konsistenten Umbenennung von formalen Parametern in Funktionen.

Definition 15.46. Die *α-Konversion*, bezeichnet durch $\rightarrow_\alpha$, ist die binäre Relation auf $\lambda(\Sigma)$, die für alle $t \in \lambda(\Sigma)$ wie folgt definiert ist. Wenn $z \notin FV(t) \cup GV(t)$, dann $(\lambda x.t) \rightarrow_\alpha (\lambda z.t[x/z])$. □

Nun könnte im vorangegangenen Beispiel zunächst die α-Konversion

$$(\lambda y.+\ x\ y) \rightarrow_\alpha (\lambda z.+\ x\ z)$$

vorgenommen werden und dann wie folgt durch β-Reduktion gerechnet werden:

$$(\lambda y.\ (\lambda x.(\lambda z.+\ x\ z))\ y\ 3)\ 4 \rightarrow_\beta (\lambda y.\ (\lambda z.+\ y\ z)\ 3)\ 4 \rightarrow_\beta (\lambda z.+\ 4\ z)\ 3 \rightarrow_\beta +\ 4\ 3$$

Allerdings haben wir jetzt etwas vorgegriffen, denn wir haben eben die α-Konversion auf einen *Teil*term angewendet, was nach Definition nicht möglich ist. Diesen Mangel wollen wir schnell beheben und schließlich die Rechenvorschrift des λ-Kalküls definieren.

Definition 15.47. Sei $t \in \lambda(\Sigma)$ und $x \in X \setminus (FV(t) \cup GV(t))$. Ein *Kontext von t*, bezeichnet durch $C[x]$, ist ein Element von $\lambda(\Sigma)$, so dass

- x genau einmal in $C[x]$ auftritt und
- es ein $s \in \lambda(\Sigma)$ gibt, so dass $C[x][x/s] = t$.

Statt $C[x][x/s]$ schreiben wir auch $C[s]$. □

Definition 15.48. Die *Rechenvorschrift des λ-Kalküls* ist die binäre Relation $\Rightarrow \subseteq \lambda(\Sigma) \times \lambda(\Sigma)$, so dass

$$\Rightarrow\ = \Rightarrow_\alpha \cup \Rightarrow_\beta.$$

Die *α-Rechenvorschrift* ist die binäre Relation $\Rightarrow_\alpha \subseteq \lambda(\Sigma) \times \lambda(\Sigma)$, so dass $t \Rightarrow_\alpha s$ genau dann wenn

- es einen Kontext $C[x]$ von t gibt und
- es λ-Terme r und r' gibt,

so dass $t = C[r]$, $r \rightarrow_\alpha r'$ und $s = C[r']$.
Die *β-Rechenvorschrift* ist die binäre Relation $\Rightarrow_\beta \subseteq \lambda(\Sigma) \times \lambda(\Sigma)$, die genau wie $\Rightarrow_\alpha$ definiert ist, nur dass in der Definition $\rightarrow_\alpha$ durch $\rightarrow_\beta$ ersetzt werden muss. □

Also gilt im vorangegangenen Beispiel:

$$(\lambda y.\ (\lambda x.(\lambda y.+\ x\ y))\ y\ 3)\ 4 \Rightarrow (\lambda y.\ (\lambda x.(\lambda z.+\ x\ z))\ y\ 3)\ 4 \Rightarrow (\lambda y.\ (\lambda z.+\ y\ z)\ 3)\ 4 \Rightarrow +\ 4\ 3$$

Zum Beispiel gilt im ersten Schritt:

- $C[u] = (\lambda y.(\lambda x.u)\, y\, 3)4$
- $r = (\lambda y.+\, x\, y)$
- $r' = (\lambda z.+\, x\, z)$
- $r \rightarrow_\alpha r'$.

Wenn beliebig viele (inklusive 0) Schritte mit Hilfe der Rechenvorschrift gerechnet werden sollen, so schreibt man dafür auch $t \Rightarrow^* s$. Also gilt z. B.

$$(\lambda y.\ (\lambda x.(\lambda y.+\ x\ y))\ y\ 3)\ 4 \Rightarrow^* +\ 4\ 3$$

Definition 15.49. Wenn $t \Rightarrow^* s$ und es gibt kein $s_1, s_2 \in \lambda(\Sigma)$ mit $s \Rightarrow_\alpha^* s_1 \Rightarrow_\beta s_2$, dann heißt s *(β-)Normalform* von t. □

Es ist klar, dass man für einen Term mit mindestens einer λ-Abstraktion beliebig viele α-Konversionen hintereinander ausführen kann. Also – bezogen auf die Terminologie des vorangegangenen Abschnitts – ist $\Rightarrow$ nicht terminierend.

Der λ-Kalkül hat aber die angenehme Eigenschaft, dass seine Rechenvorschrift konfluent ist.

Lemma 15.50. *Die Rechenvorschrift $\Rightarrow$ ist konfluent, d. h. für alle $t, t_1, t_2 \in \lambda(\Sigma)$ gilt: wenn $t \Rightarrow^* t_1$ und $t \Rightarrow^* t_2$, dann gibt es ein $s \in \lambda(\Sigma)$ mit $t_1 \Rightarrow^* s$ und $t_2 \Rightarrow^* s$.*

Wegen der Konfluenz der Rechenvorschrift ist die Normalform eines λ-Terms t eindeutig bestimmt, vorausgesetzt diese Normalform existiert überhaupt. Wenn sie existiert, dann bezeichnen wir sie mit $\mathrm{nf}(t)$.

Nun gibt es auch λ-Terme, die keine Normalform besitzen, weil es eine nicht terminierende Folge von durchführbaren β-Reduktionen gibt. Diese Möglichkeit tritt durch die Typfreiheit im ungetypten λ-Kalkül und dem damit verbundenen Phänomen der Selbstanwendung auf: ein λ-Term t kann auf sich selbst angewendet werden, also $(t\ t)$ ist auch ein korrekter Term. Betrachten wir dazu den Term

$$f_\perp = (\lambda y.yy)(\lambda y.yy).$$

Dann kann für einen beliebigen Term t die folgende Berechnung durchgeführt werden:

$$f_\perp = (\lambda y.yy)(\lambda y.yy) \Rightarrow (\lambda y.yy)(\lambda y.yy) \Rightarrow (\lambda y.yy)(\lambda y.yy) \Rightarrow \ldots$$

Man beachte, dass alle Rechnungsschritte durch β-Reduktion hervorgerufen werden.

Der Term $(\lambda y.yy)(\lambda y.yy)$ reproduziert sich. Wir haben also insbesondere hiermit bewiesen, dass die Rechenvorschrift des λ-Kalküls nicht terminierend ist.

Wir haben bisher nichts über die Semantik der Symbole aus Σ gesagt. Im ersten Beispiel dieses Abschnitts hatten wir zwar $+$ als Addition interpretiert, aber wie würde die Interpretation von $\oplus$ oder σ lauten, wenn $\oplus, \sigma \in \Sigma$?

Wir werden im Folgenden diese Frage dadurch überflüssig machen, indem wir $\Sigma = \emptyset$ setzen. Das klingt zunächst merkwürdig, wir werden aber sehen oder zumindest andeuten, dass man in der Programmiersprache $\lambda(\emptyset)$ alle berechenbaren Funktionen programmieren kann. Beginnen wir mit der Darstellung der natürlichen Zahlen in $\lambda(\emptyset)$.

Die natürlichen Zahlen lassen sich wie folgt als λ-Terme darstellen (Church-numerals):

- 0 wird repräsentiert durch den Term $(\lambda xy.y)$; um kurze Schreibweisen zu ermöglichen, bezeichnen wir diesen Term durch $\langle 0\rangle$.
- 1 wird repräsentiert durch den Term $\langle 1\rangle = (\lambda xy.xy)$;
- 2 wird repräsentiert durch den Term $\langle 2\rangle = (\lambda xy.x(xy))$;

und allgemein

- n wird repräsentiert durch den Term $\langle n\rangle = (\lambda xy.\underbrace{x(x\ldots(x}_{n}\,y)\ldots))$.

Man beachte, dass jeder Term $\langle n\rangle$ in Normalform ist.

Ganz allgemein werden wir im weiteren λ-Terme durch Abkürzungen in spitzen Klammern $\langle\ldots\rangle$ bezeichnen. Zum einen ist dadurch die Struktur eines zusammengesetzten λ-Terms besser erkennbar, zum anderen lassen sich damit Fehlinterpretationen, insbesondere zu Variablennamen, vermeiden.

Wir testen die Güte der Repräsentationen natürlicher Zahlen, indem wir einen λ-Term $\langle succ\rangle$ angeben, der die Nachfolgeroperation realisiert, d. h. für jede natürliche Zahl n soll die Normalform des λ-Terms $\langle succ\rangle\langle n\rangle$ die Zahl $n+1$ repräsentieren, also: $nf(\langle succ\rangle\langle n\rangle) = \langle n+1\rangle$.

Dazu definieren wir $\langle succ\rangle = (\lambda z.(\lambda xy.x(zxy)))$ und berechnen $\langle succ\rangle\langle n\rangle$:

$$\begin{aligned}
\langle succ\rangle\langle n\rangle &= (\lambda z.(\lambda xy.x(zxy)))\ (\lambda xy.\underbrace{x(x\ldots(x}_{n}\,y)\ldots)) \\
&\Rightarrow (\lambda xy.x(\ (\lambda xy.\underbrace{x(x\ldots(x}_{n}\,y)\ldots))\ xy)) \\
&\Rightarrow (\lambda xy.x(\ (\lambda y.\underbrace{x(x\ldots(x}_{n}\,y)\ldots))\ y)) \\
&\Rightarrow (\lambda xy.x(\ \underbrace{x(x\ldots(x}_{n}\,y)\ldots)\ )) \\
&= (\lambda xy.\underbrace{x(x\ldots(x}_{n+1}\,y)\ldots)\ ) \\
&= \langle n+1\rangle
\end{aligned}$$

Analog zu $\langle succ \rangle$ lässt sich nun auch der λ-Term $\langle pred \rangle$ angeben, der die Vorgängeroperation realisiert, d. h. für jede natürliche Zahl n mit $n \geq 1$ soll $\langle pred \rangle \langle n \rangle \Rightarrow^* \langle n-1 \rangle$ gelten.
Hierzu definieren wir:

$$\langle pred \rangle = (\lambda k.k(\lambda pu.u(\langle succ \rangle (p\ \langle true \rangle)))(p\ \langle true \rangle))(\lambda u.u\langle 0 \rangle \langle 0 \rangle)\ \langle false \rangle),$$

wobei hierin die folgenden λ-Terme für die Booleschen Werte verwendet werden:

$$\langle true \rangle = (\lambda xy.x) \qquad \text{und} \qquad \langle false \rangle = (\lambda xy.y)\ .$$

Offensichtlich gilt für alle λ-Terme s_1, s_2 mit $FV(s_1) = FV(s_2) = \emptyset$

$$\langle true \rangle\ s_1\ s_2 \Rightarrow^* s_1 \qquad \text{und} \qquad \langle false \rangle\ s_1\ s_2 \Rightarrow^* s_2\ .$$

Mithilfe dieser Definitionen lässt sich die Korrektheit von $\langle pred \rangle$ nachprüfen. Außerdem sieht man leicht, dass $\langle pred \rangle \langle 0 \rangle \Rightarrow^* \langle 0 \rangle$ gilt.
Mittels $\langle true \rangle$ und $\langle false \rangle$ können wir als Boolesche Bedingung fragen, ob eine natürliche Zahl gleich 0 ist. Zu diesem Zweck definieren wir den Term:

$$\langle iszero \rangle = (\lambda z.z\ (\langle true \rangle \langle false \rangle)\ \langle true \rangle).$$

Dann gilt:

$$\begin{aligned}
\langle iszero \rangle \langle 0 \rangle &= (\lambda z.z\ (\langle true \rangle\ \langle false \rangle)\ \langle true \rangle)\ (\lambda xy.y) \\
&\Rightarrow (\lambda xy.y)\ (\langle true \rangle\ \langle false \rangle)\ \langle true \rangle \\
&\Rightarrow (\lambda y.y)\ \langle true \rangle \\
&\Rightarrow \langle true \rangle
\end{aligned}$$

und

$$\begin{aligned}
\langle iszero \rangle \langle n+1 \rangle &= (\lambda z.z\ (\langle true \rangle \langle false \rangle)\ \langle true \rangle)\ (\lambda xy.\underbrace{x(x \ldots (x}_{n+1} y) \ldots)\ ) \\
&\Rightarrow (\lambda xy.\underbrace{x(x \ldots (x}_{n+1} y) \ldots)\ )\ (\langle true \rangle \langle false \rangle)\ \langle true \rangle \\
&\Rightarrow (\lambda y.\underbrace{(\langle true \rangle \langle false \rangle)((\langle true \rangle \langle false \rangle) \ldots ((\langle true \rangle \langle false \rangle)}_{n+1} y) \ldots)\ )\ \langle true \rangle \\
&\Rightarrow \underbrace{(\langle true \rangle \langle false \rangle)((\langle true \rangle \langle false \rangle) \ldots ((\langle true \rangle \langle false \rangle)}_{n+1} \langle true \rangle) \ldots) \\
&\Rightarrow^* \langle false \rangle.
\end{aligned}$$

Die Reduktion in der letzten Zeile ist wegen $\langle true \rangle s_1 s_2 \Rightarrow^* s_1$ gerechtfertigt. Schließlich wollen wir noch die Verzweigung mit Hilfe des *if-then-else*-Konstruktes betrachten. Die Verzweigung lässt sich durch den folgenden Term beschreiben:

$$\langle ite \rangle = (\lambda bxy.bxy)$$

Der λ-Ausdruck $\langle ite \rangle\ t_1\ t_2\ t_3$, wobei t_1, t_2 und t_3 ebenfalls λ-Terme sind mit $FV(t_1) = FV(t_2) = FV(t_3) = \emptyset$, verhält sich dann wie folgt:

$$\langle ite \rangle\ t_1\ t_2\ t_3 \Rightarrow^* \begin{cases} t_2 & \text{wenn } t_1 \Rightarrow^* \langle true \rangle \\ t_3 & \text{wenn } t_1 \Rightarrow^* \langle false \rangle \end{cases}$$

Nun haben wir in den einleitenden Beispielen zur funktionalen Programmierung gesehen, dass auch die Rekursion als Mechanismus zum Aufbau von funktionalen Programmen erlaubt ist, z. B. wurde `quicksort` rekursiv programmiert. Auf der anderen Seite ist die Rekursion in λ-Termen nicht explizit enthalten. Sie lässt sich aber simulieren; dabei nutzen wir die Eigenschaft aus, dass die Berechnungsvorschrift $\Rightarrow$ nicht terminierend ist. Wie wir das machen, zeigen wir an einem Beispiel.

Gegeben sei das folgende funktionale Programm, welches die Addition zweier natürlicher Zahlen x und y ausführt.

```
add x y = if x = 0 then y else 1 + add (x - 1) y
```

Diese wollen wir als λ-Term darstellen. Ersetzen wir hier das if-then-else-Konstrukt, den Test auf Null, die Nachfolgerfunktion und die Vorgängerfunktion durch die uns bereits bekannten λ-Terme, beachten außerdem noch die im λ-Kalkül übliche Präfix-Schreibweise, so können wir bereits mit unserem jetzigen Wissen diese rekursive Definitionsgleichung in die folgende rekursive Definition umschreiben:

$$\texttt{add} = (\lambda xy.\langle ite\rangle\ (\langle iszero\rangle\ x)\ \ y\ \ (\langle succ\rangle(\texttt{add}\ (\langle pred\rangle\ x)\ y)))$$

Nun besteht das Problem darin, aus dieser rekursiven Definition einen λ-Term $\langle add\rangle$ zu gewinnen, so dass $\langle add\rangle\langle n\rangle\langle k\rangle \Rightarrow^* \langle n+k\rangle$ für alle natürlichen Zahlen n und k.
Dazu abstrahieren wir im Term

$$(\lambda xy.\langle ite\rangle(\langle iszero\rangle x)\ \ y\ \ (\langle succ\rangle(\texttt{add}\ (\langle pred\rangle x)\ y)))$$

von `add`, d. h. wir fassen das Symbol `add` als variabel auf und setzen hierfür eine „frische“ Variable, z. B. die Variable z. Dadurch entsteht der folgende λ-Term:

$$\lambda z.(\lambda xy.\langle ite\rangle(\langle iszero\rangle x)\ \ y\ \ (\langle succ\rangle(z\ (\langle pred\rangle x)\ y)))$$

Um die Nähe zur Funktion *add* auszudrücken, wollen wir diesen λ-Term mit $\langle Add\rangle$ bezeichnen, also:

$$\langle Add\rangle = (\lambda zxy.\langle ite\rangle(\langle iszero\rangle x)\ \ y\ \ (\langle succ\rangle(z\ (\langle pred\rangle x)\ y)))$$

Beachte: Dies ist nun *keine* rekursive Gleichung mehr, sondern die Festlegung, dass $\langle Add\rangle$ einen bestimmten λ-Term bezeichnet.
Ganz intuitiv gesagt, enthält $\langle Add\rangle$ alle Informationen der Additionsfunktion. Formal gesagt ist $\langle Add\rangle$ eine Funktion, welche eine Funktion f als Argument nimmt und die Funktion

$$(\lambda xy.\langle ite\rangle(\langle iszero\rangle x)\ \ y\ \ (\langle succ\rangle(f\ (\langle pred\rangle x)\ y)))$$

als Ergebnis liefert.
Wir müssen jetzt nur noch einen Operator (λ-Term) finden, mit dessen Hilfe wir diese Funktion sooft es die Rekursion erfordert erzeugen können. Betrachten wir hierzu den λ-Term:

$$\langle Y\rangle = (\lambda z.(\lambda u.z(uu))(\lambda u.z(uu))) \in \lambda(\emptyset)\ ,$$

der auch *Fixpunktkombinator* heißt. Wir werden nun zeigen, dass wir $\langle add\rangle = \langle Y\rangle\,\langle Add\rangle$ setzen können. Für die Korrektheit dieser Behauptung beweisen wir zunächst die folgende Aussage durch vollständige Induktion über n:

$$\text{Für jedes } n,k \geq 0 \text{ gilt: } \langle YAdd\rangle\langle n\rangle\langle k\rangle \Rightarrow^* \langle n+k\rangle\ ,$$

wobei $\langle YAdd\rangle$ den λ-Term $(\lambda u.\langle Add\rangle\ (u\,u))(\lambda u.\langle Add\rangle\ (u\,u))$ abkürzt. Mit Hilfe von $\langle YAdd\rangle$ lässt sich jetzt jede beliebige Anzahl von $\langle Add\rangle$‘s erzeugen:

$$\begin{aligned}\langle YAdd\rangle &= (\lambda u.\langle Add\rangle(uu))(\lambda u.\langle Add\rangle(uu))\\ &\Rightarrow \langle Add\rangle\langle YAdd\rangle\\ &\Rightarrow \langle Add\rangle(\langle Add\rangle\langle YAdd\rangle)\\ &\Rightarrow \langle Add\rangle(\langle Add\rangle(\langle Add\rangle\langle YAdd\rangle))\\ &\Rightarrow \ldots\end{aligned}$$

Induktionsanfang für $n = 0$ und jedes k gilt

$$\begin{aligned}&\langle YAdd\rangle\langle 0\rangle\langle k\rangle\\ \Rightarrow\ &\langle Add\rangle\langle YAdd\rangle\langle 0\rangle\langle k\rangle\\ \Rightarrow\ &(\lambda xy.\langle ite\rangle(\langle iszero\rangle x)\ y\ (\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle x)\ y)))\langle 0\rangle\langle k\rangle\\ \Rightarrow\ &(\lambda y.\langle ite\rangle(\langle iszero\rangle\langle 0\rangle)\ y\ (\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle\langle 0\rangle)\ y)))\langle k\rangle\end{aligned}$$

$$
\begin{aligned}
&\Rightarrow (\langle ite\rangle(\langle iszero\rangle\langle 0\rangle)\ \langle k\rangle\ (\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle\langle 0\rangle)\ \langle k\rangle))) \\
&\Rightarrow (\langle ite\rangle\langle true\rangle\langle k\rangle(\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle\langle 0\rangle)\langle k\rangle))) \\
&\Rightarrow^* \langle k\rangle \\
&= \langle 0+k\rangle\ .
\end{aligned}
$$

Induktionsschluss $n \to n+1$:

$$
\begin{aligned}
&\langle YAdd\rangle\langle n+1\rangle\langle k\rangle \\
\Rightarrow\ &\langle Add\rangle\langle YAdd\rangle\langle n+1\rangle\langle k\rangle \\
\Rightarrow\ &(\lambda xy.\langle ite\rangle(\langle iszero\rangle x)\ y\ (\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle x)\ y)))\langle n+1\rangle\langle k\rangle \\
\Rightarrow\ &(\lambda y.\langle ite\rangle(\langle iszero\rangle\langle n+1\rangle)\ y\ (\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle\langle n+1\rangle)\ y)))\langle k\rangle \\
\Rightarrow\ &(\langle ite\rangle(\langle iszero\rangle\langle n+1\rangle)\ \langle k\rangle\ (\langle succ\rangle(\langle YAdd\rangle(\langle pred\rangle\langle n+1\rangle)\ \langle k\rangle))) \\
\Rightarrow\ &\langle ite\rangle\langle false\rangle\langle k\rangle(\langle succ\rangle(\langle YAdd\rangle\langle n\rangle\ \langle k\rangle)) \\
\Rightarrow^*\ &\langle succ\rangle(\langle YAdd\rangle\langle n\rangle\ \langle k\rangle) \\
\Rightarrow^*\ &\langle succ\rangle(\langle n+k\rangle) \qquad \text{(nach Induktionsvoraussetzung)} \\
\Rightarrow^*\ &\langle n+1+k\rangle\ .
\end{aligned}
$$

Schließlich gilt:

$$
\begin{aligned}
&\langle Y\rangle\langle Add\rangle\langle n\rangle\langle k\rangle \\
=\ &(\lambda h.(\lambda u.h(uu))(\lambda u.h(uu)))\langle Add\rangle\langle n\rangle\langle m\rangle \\
\Rightarrow\ &\underbrace{(\lambda u.\langle Add\rangle(uu))(\lambda u.\langle Add\rangle(uu))}_{\langle YAdd\rangle}\langle n\rangle\langle m\rangle \\
\Rightarrow^*\ &\langle n+k\rangle\ .
\end{aligned}
$$

Also ist tatsächlich $\langle Y\rangle\,\langle Add\rangle$ der gesuchte Term $\langle add\rangle$, d. h.: $\langle add\rangle = \langle Y\rangle\,\langle Add\rangle$.

Zur Festigung der erworbenen Kenntnisse wollen wir noch zwei Beispiele angeben:
Die Multiplikation *mult* zweier natürlicher Zahlen n und k und die Fakultätsfunktion *fac*.
Die Multiplikation lässt sich wie folgt durch ein funktionales Programm definieren:

```
mult x y = if x == 0 then 0 else add y (mult (x - 1) y)
```

Offensichtlich ist diese Funktion durch folgende rekursive Definition

$$mult = (\lambda xy.\ \langle ite\rangle\ (\langle iszero\rangle\ x)\ \ \langle 0\rangle\ \ (\langle add\rangle\ y\ (mult\ (\langle pred\rangle\ x)\ y)))$$

beschreibbar. Hier benutzen wir die Kenntnis, dass die Addition bereits als λ-Term definiert ist.
Somit ist

$$\langle Mult\rangle = (\lambda zxy.\ \langle ite\rangle\ (\langle iszero\rangle\ x)\ \ \langle 0\rangle\ \ (\langle add\rangle\ y\ (z\ (\langle pred\rangle\ x)\ y)))$$

und letztlich der gewünschte λ-Term $\langle mult\rangle = \langle Y\rangle\,\langle Mult\rangle$.
Die Fakultätsfunktion kann durch folgendes funktionale Programm definiert werden:

```
fac x = if x == 0 then 1 else x * fac (x - 1)
```

Die zugehörige rekursive Definition für diese Funktion ist dann:

$$fac = (\lambda x.\ \langle ite\rangle\ (\langle iszero\rangle\ x)\ \ \langle 1\rangle\ \ (\langle mult\rangle\ x\ (fac\ (\langle pred\rangle\ x))))$$

Auch hier benutzen wir den bereits bekannten λ-Term $\langle mult\rangle$.
Mit

$$\langle Fac\rangle = (\lambda zx.\ \langle ite\rangle\ (\langle iszero\rangle\ x)\ \ \langle 1\rangle\ \ (\langle mult\rangle\ x\ (z\ (\langle pred\rangle\ x))))$$

erhalten wir die gesuchte Darstellung $\langle fac\rangle = \langle Y\rangle\langle Fac\rangle$.

Es folgt eine Übersicht der Definition und Wirkung einiger λ-Terme:

$$\begin{aligned}
\langle 0\rangle &= (\lambda xy.y)\\
\langle 1\rangle &= (\lambda xy.xy)\\
\langle 2\rangle &= (\lambda xy.x(xy))\\
\langle n\rangle &= (\lambda xy.\underbrace{x(x\ldots(x}_{n}\,y)\ldots))
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{true}\rangle &= (\lambda xy.x)\\
\langle \mathit{false}\rangle &= (\lambda xy.y)
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{succ}\rangle &= (\lambda z.(\lambda xy.x(zxy)))\\
\langle \mathit{succ}\rangle\langle n\rangle &\Rightarrow^* \langle n+1\rangle
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{pred}\rangle &= (\lambda k.k(\lambda pu.u(\langle \mathit{succ}\rangle(p\ \langle \mathit{true}\rangle))(p\ \langle \mathit{true}\rangle))(\lambda u.u\langle 0\rangle\langle 0\rangle)\ \langle \mathit{false}\rangle)\\
\langle \mathit{pred}\rangle\langle 0\rangle &\Rightarrow^* \langle 0\rangle\\
\langle \mathit{pred}\rangle\langle n\rangle &\Rightarrow^* \langle n-1\rangle \qquad \text{wenn } n \geq 1
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{Add}\rangle &= (\lambda zxy.\langle \mathit{ite}\rangle(\langle \mathit{iszero}\rangle x)\ \ y\ \ (\langle \mathit{succ}\rangle(z\ (\langle \mathit{pred}\rangle x)\ y)))\\
\langle \mathit{add}\rangle &= \langle Y\rangle\,\langle \mathit{Add}\rangle\\
\langle \mathit{add}\rangle\langle n_1\rangle\langle n_2\rangle &\Rightarrow^* \langle n_1+n_2\rangle
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{Sub}\rangle &= (\lambda zxy.\langle \mathit{ite}\rangle(\langle \mathit{iszero}\rangle y)\ \ x\ \ (\langle \mathit{pred}\rangle(z\ x\ (\langle \mathit{pred}\rangle y))))\\
\langle \mathit{sub}\rangle &= \langle Y\rangle\,\langle \mathit{Sub}\rangle\\
\langle \mathit{sub}\rangle\langle n_1\rangle\langle n_2\rangle &\Rightarrow^* \begin{cases}\langle n_1-n_2\rangle & \text{wenn } n_1 \geq n_2\\ \langle 0\rangle & \text{sonst}\end{cases}
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{Mult}\rangle &= (\lambda zxy.\ \langle \mathit{ite}\rangle\ (\langle \mathit{iszero}\rangle\ x)\ \ \langle 0\rangle\ \ (\langle \mathit{add}\rangle\ y\ (z\ (\langle \mathit{pred}\rangle\ x)\ y)))\\
\langle \mathit{mult}\rangle &= \langle Y\rangle\,\langle \mathit{Mult}\rangle\\
\langle \mathit{mult}\rangle\langle n_1\rangle\langle n_2\rangle &\Rightarrow^* \langle n_1\cdot n_2\rangle
\end{aligned}$$

$$\langle \mathit{mod}\rangle\langle n_1\rangle\langle n_2\rangle \Rightarrow^* \langle z\rangle \qquad \text{wobei } 0 \leq z < n_2 \text{ mit } z = n_1 - i\cdot n_2 \text{ für ein } i \in \mathbb{N}$$

$$\begin{aligned}
\langle \mathit{ite}\rangle &= (\lambda bxy.bxy)\\
\langle \mathit{ite}\rangle\ s\ s_1\ s_2 &\Rightarrow^* \begin{cases}s_1 & \text{wenn } s \Rightarrow^* \langle \mathit{true}\rangle\\ s_2 & \text{sonst}\end{cases}
\end{aligned}$$

$$\begin{aligned}
\langle \mathit{iszero}\rangle &= (\lambda z.z(\langle \mathit{true}\rangle\langle \mathit{false}\rangle)\langle \mathit{true}\rangle)\\
\langle \mathit{iszero}\rangle\ s &\Rightarrow^* \begin{cases}\langle \mathit{true}\rangle & \text{wenn } s \Rightarrow^* \langle 0\rangle\\ \langle \mathit{false}\rangle & \text{sonst}\end{cases}
\end{aligned}$$

$$\langle Y\rangle = (\lambda h.(\lambda y.h\ (y\ y))\ (\lambda y.h\ (y\ y))) \qquad \text{(Fixpunktkombinator)}$$

16 Kleine Einführung in die Logik-Programmierung

(In Englisch)

Logic programming is based on first-order predicate logic; in its basic form the logic is restricted to Horn clauses, i.e., universal quantified disjunctions of literals with at most one positive literal. The operational semantics is based on the resolution principle of Robinson (see [Rob65]).

A prominent example of such a programming language is Prolog (Kowalski and Colmerauer, 1972, see [Kow74, CKRP73]). See [SS94, Llo87] for nice introductions to logic programming. Here we will discuss a subset of Prolog, which we call Prolog$^-$. You can find an online Prolog interpreter under `http://swish.swi-prolog.org/`.

16.1 First examples and syntax of Prolog$^-$

Beispiel 16.1. We consider the unary predicate **`nat`** which contains all non-negative integers in unary representation, e.g., `s(s(0))` for 2. Rather than writing `s(s(0))` $\in$ **`nat`**, we write **`nat`**`(s(s(0)))`.

Here is a small Prolog$^-$-program which describes the non-negative integers:

```
nat(0).
nat(s(X)) :- nat(X).
```

The first line is a *fact* telling that `0` is a non-negative integer. The second line reads: if `X` is a non-negative integer, then so is `s(X)`. (So one might call the symbol combination `:-` "if".)

Now we can ask the following *query* (or, in other words, we can state the following *goal*): Is 2 a natural number? Formally:

```
?- nat(s(s(0))).
```

Applying the second **`nat`**-rule while binding the variable `X` to `s(0)` we obtain the goal

```
?- nat(s(0)).
```

Applying this rule again we obtain

```
?- nat(0).
```

Finally, this is a fact due to the first **`nat`**-rule; thus, we derive the goal

```
?-.
```

We call such a sequence of goals an *SLD-derivation*; this particular sequence of goals is even an *SLD-refutation*, because it ends with the particular goal **`?-.`**. SLD is an abbreviation and stands for SL-resolution of definite goals; and SL stands for linear resolution with selection function. □

Definition 16.2 (Prolog$^-$ Program)**.** The context-free syntax of Prolog$^-$ is given by the following rules of an EBNF-definition:

$$
\begin{aligned}
\langle prog \rangle &::= \langle pred \rangle \,\hat{\{}\, \langle pred \rangle \,\hat{\}} \\
\langle pred \rangle &::= \langle clause \rangle \,\hat{\{}\, \langle clause \rangle \,\hat{\}} && (\textit{predicate definitions}) \\
\langle clause \rangle &::= \langle lit \rangle \texttt{ :- } \langle lit \rangle \,\hat{\{}\, \texttt{ , } \langle lit \rangle \,\hat{\}}\, \texttt{ .} && (\textit{rules}) \\
&\quad \hat{|}\, \langle lit \rangle \texttt{ .} && (\textit{facts})
\end{aligned}
$$

$$\langle lit\rangle ::= \langle predid\rangle \mathbin{\hat{|}} \langle predid\rangle \texttt{(} \langle pat\rangle \hat{\{} \texttt{,} \langle pat\rangle \hat{\}} \texttt{)} \qquad (\textit{(positive) literals})$$

$$\langle pat\rangle ::= \langle varid\rangle \mathbin{\hat{|}} \langle conid\rangle \mathbin{\hat{|}} \langle conid\rangle \texttt{(} \langle pat\rangle \hat{\{} \texttt{,} \langle pat\rangle \hat{\}} \texttt{)} \qquad (\textit{patterns})$$

$$\langle conid\rangle ::= \hat{(}\texttt{a} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{z}\hat{)}\,\hat{\{}\texttt{a} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{z}\hat{\}} \qquad (\textit{constructors})$$

$$\langle predid\rangle ::= \hat{(}\texttt{a} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{z}\hat{)}\,\hat{\{}\texttt{a} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{z}\hat{\}} \qquad (\textit{predicate symbols})$$

$$\langle varid\rangle ::= \hat{(}\texttt{A} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{Z}\hat{)}\,\hat{\{}\texttt{A} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{Z} \mathbin{\hat{|}} \texttt{a} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{z} \mathbin{\hat{|}} \texttt{0} \mathbin{\hat{|}} \ldots \mathbin{\hat{|}} \texttt{9}\hat{\}} \qquad (\textit{variables})$$

Let $P \in W(\langle prog\rangle)$. For every rule L_0 `:-` $L_1, \ldots, L_n$. in P, the part L_0 (the part $L_1, \ldots, L_n$, respectively) is called *left-hand side* or *head* (*right-hand side* or *body*, respectively). For every fact L_0. in P, the part L_0 is also called *left-hand side* or *head*.

The set of *goals for* P is defined by adding the following EBNF-rule

$$\langle goal\rangle ::= \texttt{?-}\ \langle lit\rangle \hat{\{} \texttt{,} \langle lit\rangle \hat{\}} \texttt{.} \qquad (\textit{goals})$$

$$\mathbin{\hat{|}}\ \texttt{?-.} \qquad (\textit{empty goal}) \quad \square$$

Here is a more general example of a rule:

```
p(b(X,c),Y) :- q(d(X),Z), r(e(Z),Y).
```

The head of this rule is the literal `p(b(X,c),Y)`; it consists of the binary predicate `p` and the two patterns `b(X,c)` and `Y`. The symbols `b` and `c` denote constructors (of arity 2 and 0, respectively); the symbols `X`, `Y`, and `Z` denote variables. The body of the rule consists of the two literals `q(d(X),Z)` and `r(e(Z),Y)`, where `q` and `r` are again binary predicates.

Beispiel 16.3. We can define the summation of two non-negative integers as ternary predicate `sum` with the intuition that, e.g., `sum(<2>, <3>, <5>)` holds:

```
sum(0, X, X) :- nat(X).
sum(s(X), Y, s(Z)) :- sum(X, Y, Z).
```

Now we can ask the following query: `sum(<2>, <5>, <7>)` where `<n>` abbreviates `s(...s(0)...)` with n times `s`. We might also say: we want to prove that `sum(<2>, <5>, <7>)` holds, or: we have the proof obligation `sum(<2>, <5>, <7>)`.

By applying rules in an appropriate way we obtain the following SLD-refutation:

```
?- sum(<2>, <5>, <7>).
?- sum(<1>, <5>, <6>).
?- sum(<0>, <5>, <5>).
?- nat(<5>).
?- nat(<4>).
...
?- nat(<0>).
?-.
```

Thus we have verified that $2 + 5 = 7$. We can also use the same Prolog$^-$-program to deduce summands from a sum. For instance, we might be interested in solving the equation $x + 1 = 3$. It is possible that an occurrence of a variable `X` in the program also occurs in the goal. In order to avoid clashes, we use variants of rules; a variant of a rule is obtained from the rule by renaming the variables consistently. After renaming, no variable occurs in the goal and in the variant.

```
            ?- sum(X , <1>, <3>).
{X =s(X1)}  ?- sum(X1, <1>, <2>).
{X1=s(X2)}  ?- sum(X2, <1>, <1>).
{X2=0}      ?- nat(<1>).
            ?- nat(<0>).
            ?-.
```

To the left of the symbols `?-` we protocol the matching substitution needed to be able to apply the rule. After having finished the refutation, we iteratively substitute the matching substitutions into itself, starting from the variables of the initial goal. Thus we obtain

$$\texttt{X} = \texttt{s(X1)} = \texttt{s(s(X2))} = \texttt{s(s(0))}.$$

Thus we have proved that `s(s(0))` (i.e., 2) is a solution of the equation.

Indeed, it is possible to perfom another computation starting from the same goal:

```
            ?- sum(X , <1>, <3>).
{X =s(X1)} ?- sum(X1, <1>, <2>).
{X1=s(X2)} ?- sum(X2, <1>, <1>).
{X2=s(X3)} ?- sum(X3, <1>, <0>).
```

But this computation stops without success, because there is no rule applicable although we still have a proof obligation. □

Beispiel 16.4. We can build up lists of objects as follows:

```
list(nil).
list(cons(X, Xs)) :- list(Xs).
```

Here the lists are constructed with a nullary constructor `nil` and a binary constructor `cons`. Instead of `nil` and `cons(X, Xs)` we will use the more intuitive form `[]` and `[X|Xs]`, respectively. Moreover we abbreviate the list `[A|[B|[C|[]]]]` by `[A, B, C]` (and similarly for other lists).

Next we write a Prolog⁻-program which sums up the elements of a list over integers. For this, we define a binary predicate `listsum` for which, e.g., `listsum([2, 4], 6)` holds.

```
listsum([]    , 0).
listsum([X|Xs], Z) :- listsum(Xs, Y), sum(X, Y, Z).
```

Here we can observe a phenomenon of logic programs: the right-hand side may contain variables which do not occur in its left-hand side (in our example: `Y` in the second rule).

Now we can compute the sum of the list `[<2>, <3>, <1>]` as follows:

```
  ?- listsum([<2>, <3>, <1>], U).
  ?- listsum([<3>, <1>], Y), sum(<2>, Y, U).
```

Next we apply the second rule to the literal `listsum ([<3>, <1>], Y)`:

```
                ?-  listsum([<3>, <1>], Y), sum(<2>, Y, U).
                ?-  listsum([<1>], V), sum(<3>, V, Y), sum(<2>, Y, U).
                ?-  listsum([], T), sum(<1>, T, V), sum(<3>, V, Y), sum(<2>, Y, U).
{T=0}           ?-  sum(<1>, 0, V), sum(<3>, V, Y), sum(<2>, Y, U).
{V=s(V1)}       ?-  sum(0, 0, V1), sum(<3>, s(V1), Y), sum(<2>, Y, U).
{V1=0}          ?-  sum(<3>, s(0), Y), sum(<2>, Y, U).
{Y=s(s(s(Y1)))} ?-* sum(0, s(0), Y1), sum(<2>, s(s(s(Y1))), U).
{Y1=s(0)}       ?-  sum(<2>, <4>, U).
{U=s(s(U1))}    ?-* sum(0, <4>, U1).
{U1=<4>}        ?-.
```

Thus `U` = `s(s(U1))` = `s(s(<4>))` = `<6>`.

But we can also ask funny queries like: `listsum(Z, <5>)` which computes the set of all lists which sum up to 5. □

Beispiel 16.5 ([SS94], p.83)**.** We consider formulas of propositional logic (Aussagenlogik) with variables, conjunction (`and`), disjunction (`or`), negation (`neg`), and the constants **`true`** and **`false`**.

The following Prolog⁻-program can be used to verfy whether a formula of propositional logic is satisfiable or not. We use the predicates **`sat`** and **`inv`** for *satisfiable*, and *invalid*, respectively. A formula is satisfiable if there is a truth assignment to its variables that makes it true; it is invalid if there is an assignment that makes it false.

```
     sat(true).
sat(and(X, Y)) :- sat(X), sat(Y).
 sat(or(X, Y)) :- sat(X).
 sat(or(X, Y)) :- sat(Y).
   sat(neg(X)) :- inv(X).

inv(and(X, Y)) :- inv(X).
```

```
inv(and(X, Y)) :- inv(Y).
 inv(or(X, Y)) :- inv(X), inv(Y).
   inv(neg(X)) :- sat(X).
```

Now we can ask whether the formula `and(U, neg(or(neg(U), W)))` (with variables `U` and `W`) has a satisfying variable assignment.

```
            ?- sat(and(U, neg(or(neg(U), W)))).
            ?- sat(U), sat(neg(or(neg(U), W))).
{U=true}    ?- sat(neg(or(neg(true), W))).
            ?- inv(or(neg(true), W)).
            ?- inv(neg(true)), inv(W).
            ?- sat(true), inv(W).
            ?- inv(W).
{W=neg(Z)} ?- sat(Z).
{Z=true}    ?-.
```

Thus we obtain the variable assignment `U = true, W = neg(true)`; this makes the original formula true. □

16.2 SLD-derivations, SLD-refutations, and computed answers

Having seen some examples of refutations, we now want to provide formal definitions for the used notions.

Definition 16.6 (SLD-Resolution)**.** Let P be a Prolog$^-$ program and let $G = (\texttt{?-}L_1, \ldots, L_n.)$ with $n \geq 1$ be a goal for P. If

- there is an $i \in \mathbb{N}$ with $1 \leq i \leq n$,
- there is a variant $C = (M_0\texttt{:-}M_1, \ldots, M_m.)$ of a rule of P (i.e. C is constructed by a renaming of variables), such that G and C have no variables in common, and
- σ is a most general unifier of L_i and M_0,

then

$$G' = (\texttt{?-}\tilde{\sigma}(L_1), \ldots, \tilde{\sigma}(L_{i-1}),\ \tilde{\sigma}(M_1), \ldots, \tilde{\sigma}(M_m),\ \tilde{\sigma}(L_{i+1}), \ldots, \tilde{\sigma}(L_n).)$$

is called a *resolvent of G and C with σ*; the literal L_i is called the *selected literal* in G. □

Definition 16.7 (SLD-Derivation, SLD-Refutation)**.** Let P be a Prolog$^-$ program and let G be a goal for P.

An *SLD-derivation of* (P, G) is a (possibly infinite) sequence $G_0, G_1, G_2, G_3, \ldots$ of goals for P, such that

- $G_0 = G$,
- there is a sequence $C_1, C_2, C_3, \ldots$ of variants of rules of P,
- there is a sequence $\sigma_1, \sigma_2, \sigma_3, \ldots$ of most general unifiers, and
- G_{i+1} is a resolvent of G_i and C_{i+1} with σ_{i+1} for every $i \in \mathbb{N}$.

An *SLD-refutation of* (P, G) is a finite SLD-derivation $G_0, G_1, G_2, \ldots, G_n$ of (P, G) with $n \in \mathbb{N}$ such that $G_n = (\texttt{?-.})$. □

In our previous examples, we have show an SLD-refutation in the form:

$$\begin{array}{r} G_0 \\ \{\sigma_1\}\ G_1 \\ \ldots \\ \{\sigma_n\}\ G_n \end{array}$$

Indeed, we have dropped the unifier σ_i if it only binds variables of the variant C_i of a rule.

Definition 16.8 (Computed Answer)**.** Let P be a Prolog$^-$ program, let G be a goal for P, let $n \in \mathbb{N}$, and let $G_0, G_1, G_2, \ldots, G_n$ be an SLD-refutation of (P, G) where $\sigma_1, \sigma_2, \ldots, \sigma_n$ are the most general unifiers, which are used in the SLD-resolution steps.

The substitution σ, which results from restricting the composition $\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_n$ (first σ_1, second σ_2, and so on) to the variables in G, is called a *computed answer for* (P, G). □

For instance, in Example 16.5 with the propositional formulas, the computed answer for the program P and the goal `and(U, neg(or(neg(U), W)))` is the substitution

$$\sigma(\texttt{U}) = \texttt{true},\ \sigma(\texttt{W}) = \texttt{neg(true)}\ .$$

16.3 Prolog evaluation strategy

In the implementation of the programming language Prolog, a particular evaluation strategy is used for the selection of the literal which is resolved next, and for the selection of rules. Let us consider a goal $G = (\texttt{?-}L_1, \ldots, L_n.)$ with $n \geq 1$. Then, in the definition of SLD-resolution, an arbitrary $i \in \{1, \ldots, n\}$ is selected and the evaluation strategy tries to unify the literal L_i with the head of an arbitrary rule of the program. The Prolog evaluation strategy however, always

1. selects the leftmost literal of the goal, i.e., $i = 1$ (assume that $L_1 = p(t_1, \ldots, t_n)$ for some patterns $t_1, \ldots, t_n$), and
2. traverses the list of rules for the predicate p in the order in which they occur in the program, and performs the unifiability-check between L_1 and the head of the selected rule.
3. If the evaluation gets stuck, then it backtracks to the last choice (first: next rule in the list of rules for p; second: next literal in the goal) and tries again in the same way.

This has consequences for the programmer as we will see. For this, consider the following representation of connections of cities by flights by the predicate `c`:

```
c(dd, ber).
c(dd, muc).
c(dd, fra).
c(fra, lax).
...
```

Then we would like to program reachability within this graph of connections. Here is the first approach:

```
flight(X, Y) :- flight(X, Z), c(Z, Y).
flight(X, Y) :- c(X, Y).
```

Now we consider the goal `?- flight(dd,lax).`. Here is an SLD-refutation:

```
        ?- flight(dd, lax).
        ?- flight(dd, Z), c(Z, lax).
        ?- c(dd, Z), c(Z, lax).
{Z=fra} ?- c(fra, lax).
        ?-.
```

So, it can verify the goal: first fly to Frankfurt, then to Los Angeles. In particular, in the second step we have selected the leftmost literal (i.e., `flight(dd, Z)`) and have used the second rule for `flight`.

Now let us look at a Prolog-evaluation:

```
?- flight(dd, lax).
?- flight(dd, Z), c(Z, lax).
?- flight(dd, Z1), c(Z1, Z), c(Z, lax).
?- flight(dd, Z2), c(Z2, Z1), c(Z1, Z), c(Z, lax).
...
```

The Prolog-evaluation strategy always selects the leftmost literal and checks the rules in the order in which they occur in the program. Since the first rule for `flight` is left-recursive, this leads to an infinite sequence of goals. Thus, the Prolog-evaluation of the goal will not find a solution, although there is one.

The consequence for the Prolog-programmer is to avoid left-recursion and to place facts at the beginning of the list of rules for each predicate. Here is another Prolog-program which takes care of this programming principle:

```
flight(X, Y) :- c(X, Y).
flight(X, Y) :- flight(X, Z), c(Z, Y).
```

Now a Prolog-evaluation (denoted by **P-**) looks as follows:

```
        P- flight(dd, lax).
        P- c(dd, lax).
        % there is no c-rule for which its head is unifiable with c(DD,LAX);
        % go back to the previous choice ...
        P- flight(dd, lax).
        % and try the next possible rule:
        P- flight(dd, Z), c(Z, lax).
        P- c(dd, Z), c(Z, lax).
{Z=fra} P- c(fra, lax).
        P-.
```

Thus, using this Prolog-program, also the Prolog-evaluation finds the solution.

16.4 Relationsship between clauses of Prolog$^-$ and formulas of first-order predicate logic

Every rule

```
L0 :- L1,...,Ln.
```

of a Prolog$^-$-program P represents a formula of first-order predicate logic as follows:

- the literals `L0`,...,`Ln` are implicitly conjunctively connected,
- the symbol **:-** is the logical implication $\leftarrow$, and
- the occurring variables are universally quantified.

Thus the rule `L0 :- L1,...,Ln.` represents the formula

$$\forall(L_0 \leftarrow (L_1 \wedge \ldots \wedge L_n)) \ ,$$

where $\forall$ represents the universal quantification $\forall x_1\ \forall x_2\ \forall x_3 \ldots \forall x_n$, if these are all the variables that occur in the rule. For instance, the rule

```
sat(and(X, Y)) :- sat(X), sat(Y).
```

represents the formula

$$\forall x.\ \forall y.\ \mathbf{sat}(\mathsf{and}(x, y)) \leftarrow (\mathbf{sat}(x)\ \wedge\ \mathbf{sat}(y))$$

If we additionally use the law that $a \leftarrow b$ is logically equivalent to $a \vee \neg b$, where a and b are formulas, $\vee$ is the logical disjunction, and $\neg$ is the logical negation, then we can transform

$$\forall(L_0 \leftarrow (L_1 \wedge \ldots \wedge L_n))$$

into

$$\forall(L_0 \vee \neg(L_1 \wedge \ldots \wedge L_n))$$

and further into

$$\forall(L_0 \vee \neg L_1 \vee \ldots \vee \neg L_n)$$

with the help of DeMorgan's law. In analogy, a fact `L0.` represents the formula $\forall(L_0)$. Hence, rules and facts are universally quantified disjunctions of literals, where exactly one literal is positive.

In general, a *Horn clause* is a universally quantified disjunction of literals, where at most one literal is positive. A Horn clause without a positive literal is called a *goal*. Note that a goal

```
?- L1, ... ,Ln.
```

represents the formula

$$\forall(\neg L_1 \vee \ldots \vee \neg L_n) \ .$$

In particular, the empty goal `?-.` represents the formula

$$\bigvee_{L \in \emptyset} L \leftarrow \bigwedge_{L \in \emptyset} L$$

which is equivalent to the implication $false \leftarrow true$, and hence equivalent to the truth value $false$.
Now we have embedded Prolog$^-$ programs and goals into the first-order predicate logic.

In SLD-refutations we exploit the following important lemma. (Note that this lemma is not restricted to the special formulas which arise in Prolog$^-$.)

Lemma 16.9. *[Llo87, Prop. 3.1] Let P be any finite set of closed formulas of first-order predicate logic. Moreover, let F be a closed formula of first-order predicate logic. Then the following two statements are equivalent:*

1. *F is a logical consequence of P.*
2. *$P \cup \{\neg F\}$ is unsatisfiable.*

For example, we can choose P to be the Prolog$^-$ program P_{sum} for sums, because this is a finite set of closed formulas of first-order predicate logic. As F we choose the formula

$$F \ = \ (\exists x : x + 1 = 3)$$

because we want to prove that F is a logical consequence of P_{sum} (and, of course, also have a way of getting a value for x). According to Lemma 16.9, we consider the logical negation of F, i.e., the fomula

$$\neg F = (\forall x : \neg(x + 1 = 3)) \ ,$$

and prove that $P \cup \{\neg F\}$ is unsatisfiable.
How does $\neg F$ look like in Prolog$^-$? If we represent the relation $x + 1 = 3$ between x, 1, and 3 by the predicate `sum`, then we obtain some itermediate form:

$$\neg F = (\forall x : \neg(\texttt{sum}(x, 1, 3))) \ .$$

We can represent this formula as the goal

```
?- sum(X, <1>, <3>).
```

Next we try to prove that $P \cup \{\neg F\}$ is unsatisfiable. For this we use the following important relation between unsatisfiability and SLD-refutation.

Lemma 16.10. *[Llo87, Cor. 7.2 and Thm. 8.4] Let P be a Prolog$^-$ program and G be a goal. Then the following are equivalent:*

1. *$P \cup \{G\}$ is unsatisfiable.*
2. *There is an SLD-refutation of G.*

Combining Lemmas 16.9 and 16.10 we obtain:

Satz 16.11. *Let P be a Prolog$^-$ program and G be a goal. Let $G = \neg F$. Then the following are equivalent:*

1. *F is a logical consequence of P.*
2. *There is an SLD-refutation of $\neg F$.*

Thus, for example, if there is an SLD-refutation starting from

$$\neg F = (\texttt{?- sum(X, <1>, <3>).})$$

then $F = (\exists x : x + 1 = 3)$ is a logical consequence of P.

17 Implementierung einer einfachen imperativen Programmiersprache

In diesem Kapitel wollen wir – zumindest einführend – zeigen, wie man eine imperative Programmiersprache implementieren kann. Zu einer Implementierung gehören immer

- eine Zielmaschine (abstrakte Maschine, Befehle, Befehlssemantik, Programmsemantik) und
- ein Übersetzer (Compiler), der Programme der imperativen Programmiersprache in Programme der Zielmaschine übersetzt.

Hier beschränken wir uns auf das Fragment (subset) C_1 der Sprache C. In C_1 gibt es als Datentyp nur Integer. Neben der Zuweisung existieren `if`- und `while`-Anweisungen sowie Funktionen mit Wert- und Referenzparametern und Ergebnistyp `void`.

Die Wirkung von diesen Funktionen ohne Rückgabewert erfolgt über Referenzparameter. Man kann sich dieses Kapitel auch als Formalisierung des pulsierenden Speichers bei Funktionsaufrufen (siehe Kapitel 5) vorstellen.

Aus didaktischen Gründen beginnen wir mit einer noch kleineren imperativen Sprache, nämlich C_0, und erweitern dann die dort diskutierten Konzepte geeignet auf C_1.

17.1 Teilsprache C_0

In diesem Abschnitt definieren wir die Teilsprache C_0 von C, in der einerseits schon einige Algorithmen formuliert werden können, die aber andererseits so „klein“ ist, dass wir deren Implementierung studieren können. Wir kehren zu unserem Beispielprogramm „Summation“ aus Kapitel 3 zurück, einem Programm, welches bereits bis auf die Deklaration globaler Variablen die Ansprüche eines C_0-Programms erfüllt. Diesen Mangel können wir leicht beheben, ohne die Spezifikation der Aufgabenstellung ändern zu müssen. Das folgende Programm enthält nur (zur Funktion `main`) lokale Variablen und ist ein C_0-Programm. Es berechnet die Summe $s = \sum_{j=1}^{n} j^2$ der ersten n Quadratzahlen.

```
1  /* Summation */
2  #include <stdio.h>
3
4  int main()
5  { int i, n, s;
6
7    scanf("%d", &n);
8    i = 1;
9    s = 0;
10   while (i <= n)
11   {
12     s = s+i*i;
13     i = i+1;
14   }
15   printf("%d", s);
16   return 0;
17 }
```

Erste Bemerkungen

- Die Funktion `int main()` muss in jedem C_0-Programm genau einmal existieren. Weitere Funktionen sind nicht erlaubt.

- Ein C_0-Programm muss auf die Bibliothek `stdio` durch `#include` zugreifen. Die Bibliothek `stdio` ist in der Modulbibliothek standardmäßig vorhanden. `#include <stdio.h>` erlaubt die Verwendung der Lese- und Schreibprozeduren (importiert aus der Bibliothek `stdio`).
- Als *Datenstrukturen von* C_0 lassen wir nur Variable vom Typ **`int`** zu. Ebenso können in C_0 Konstanten deklariert werden.
- Die *Kontrollstrukturen von* C_0 sind die Ein-/Ausgabebefehle, die Zuweisung, die Sequenz, die Verzweigung und die bedingte Schleife. Sprunganweisungen werden ganz bewusst nicht in die Sprache C_0 aufgenommen.

Für C_0-Programme sollen in diesem Kapitel

1. die Syntax von C_0 und
2. die Implementierung (durch abstrakte Maschine, Befehle und Übersetzer)

formal definiert werden.

17.1.1 Syntax von C_0

Hier würden wir gerne eine EBNF-Definition $\mathcal{E}_{C_0}$ angeben, deren Objektsprache C_0 ist, d. h. $W(\mathcal{E}_{C_0}) = C_0$. Leider lässt sich dieser Wunsch nicht erfüllen, vielmehr können wir mit der Beschreibungssprache EBNF nur eine Obermenge $\hat{C}_0$ von C_0 beschreiben. In der Tat gibt es gar keine EBNF-Definition, deren Objektsprache C_0 ist. Das liegt daran, dass alle C_0-Programme gewisse kontextsensitive Bedingungen berücksichtigen müssen; eine dieser Bedingungen besagt z. B., dass eine Variable, die im Anweisungsteil eines Programms benutzt wird, auch deklariert sein muss. Bezogen auf den Anweisungsteil ist das Vorliegen der Eigenschaft: „Variable x ist deklariert“ also nur durch Betrachtung des Kontextes des Anweisungsteils zu entscheiden; die Eigenschaft ist *kontextsensitiv*.

Kontextsensitive Eigenschaften können in EBNF-Definitionen *nicht* beschrieben werden; dieselbe Situation lag bereits im Kapitel 3 vor. Der Sachverhalt trifft auch für gängige Programmiersprachen wie z. B. *C*, Java, Pascal, PHP zu. Diese Eigenschaften müssen in einer anderen, gegenüber EBNF mächtigeren Beschreibungssprache formuliert werden. Hier benutzen wir die natürliche Sprache; eine alternative, formale Beschreibungssprache sind Attributgrammatiken.

Also: Die folgende EBNF-Definition $\mathcal{E}_{C_0}$ wird eine Objektsprache $W(\mathcal{E}_{C_0}) = \hat{C}_0$ mit $\hat{C}_0 \supseteq C_0$ beschreiben. Zum Beispiel liegt das folgende Programm P in $\hat{C}_0$, aber nicht in C_0, weil die im Anweisungsteil auftretende Variable `j` nicht deklariert ist.

```
1 /* Ungut */
2 #include <stdio.h>
3
4 int main()
5 { int i;
6
7   j = i;
8   return 0;
9 }
```

Im folgenden geben wir die EBNF-Regeln von $\hat{C}_0$ an. Wir verzichten auf die explizite Auflistung der syntaktischen Variablen. Das Startsymbol der EBNF-Definition ist die syntaktische Variable ⟨Program⟩:

⟨Program⟩ ::= `#include <stdio.h>` `int main()` ⟨Block⟩

⟨Block⟩ ::= `{` ⟨Declaration⟩ [⟨StatementSequence⟩] `return 0; }`

⟨Declaration⟩ ::= [⟨ConstDeclaration⟩] [⟨VarDeclaration⟩]

⟨ConstDeclaration⟩ ::= `const` ⟨Ident⟩ `=` [`-`] ⟨Number⟩ { `,` ⟨Ident⟩ `=` [`-`] ⟨Number⟩ } `;`

⟨VarDeclaration⟩ ::= `int` ⟨Ident⟩ { `,` ⟨Ident⟩ } `;`

⟨StatementSequence⟩ ::= ⟨Statement⟩ $\hat{\{}$ ⟨Statement⟩ $\hat{\}}$

⟨Statement⟩ ::= ⟨Assignment⟩ $\hat{|}$ ⟨IfStatement⟩ $\hat{|}$ ⟨WhileStatement⟩ $\hat{|}$ `scanf("%d", &`⟨Ident⟩ `);`
$\hat{|}$ `printf("%d",` ⟨Ident⟩ `);` $\hat{|}$ ⟨CompStatement⟩

⟨Assignment⟩ ::= ⟨Ident⟩ `=` ⟨SimpleExpression⟩ `;`

⟨IfStatement⟩ ::= `if (` ⟨BoolExpression⟩ `)` ⟨Statement⟩ $\hat{[}$ `else` ⟨Statement⟩ $\hat{]}$

⟨WhileStatement⟩ ::= `while (` ⟨BoolExpression⟩ `)` ⟨Statement⟩

⟨CompStatement⟩ ::= `{` ⟨StatementSequence⟩ `}`

⟨BoolExpression⟩ ::= ⟨SimpleExpression⟩ ⟨Relation⟩ ⟨SimpleExpression⟩

⟨SimpleExpression⟩ ::= $\hat{[}$ `+` $\hat{|}$ `-` $\hat{]}$ ⟨Term⟩ $\hat{\{}$ $\hat{(}$ `+` $\hat{|}$ `-` $\hat{)}$ ⟨Term ⟩ $\hat{\}}$

⟨Term⟩ ::= ⟨Factor⟩ $\hat{\{}$ $\hat{(}$ `*` $\hat{|}$ `/` $\hat{|}$ `%` $\hat{)}$ ⟨Factor⟩ $\hat{\}}$

⟨Factor⟩ ::= ⟨Ident⟩ $\hat{|}$ ⟨Number⟩ $\hat{|}$ `(` ⟨SimpleExpression⟩ `)`

Schließlich stellen wir eine Liste mit solchen kontextsensitiven Eigenschaften auf, die sich *nicht* in der Beschreibungssprache EBNF formulieren lassen. Wenn ein Programm in $\hat{C}_0$ diese Bedingungen erfüllt, dann ist es ein C_0-Programm.

- Jeder Bezeichner darf höchstens einmal in ⟨Declaration⟩ deklariert sein.
- Tritt ein Bezeichner in der ⟨StatementSequence⟩ des ⟨Block⟩s auf, so muss er in der ⟨Declaration⟩ des ⟨Block⟩s deklariert sein (als Variable oder Konstante).
- ⟨Ident⟩s in linken Seiten von ⟨Assignment⟩s und in Lese- und Schreibanweisungen dürfen keine Konstantennamen sein.

17.1.2 Abstrakte Maschine AM_0

Die AM_0 ist ein Von-Neumann-Rechner, denn sie weist Speicherplätze und einen Mechanismus auf, mit dessen Hilfe man die dort gespeicherten Werte verändern kann. Die AM_0 ist eine *abstrakte* Maschine, d. h. in Bezug auf konkrete Rechenmaschinen wird von gewissen Gegebenheiten abstrahiert. Zum Beispiel wird die Verbindung (Datenbusse, Steuerleitungen, Stromversorgung etc.) der einzelnen Komponenten der AM_0 auf mathematischem Niveau beschrieben. Ebenfalls werden Daten und Programme getrennt, was bei üblichen Von-Neumann-Rechnern nicht getan wird. Mehr noch: Das AM_0-Programm wird überhaupt nicht gespeichert. Dennoch kann die AM_0 wie in Abbildung 17.1 veranschaulicht werden; sie weist also schon gewisse Ähnlichkeiten mit konkreten Rechenmaschinen auf. Wir haben der AM_0 (zusätzlich) einen Programmspeicher angefügt. Die AM_0 lässt sich folgendermaßen formulieren:

$AM_0 = BZ \times DK \times HS \times Inp \times Out$ mit

$$
\begin{aligned}
BZ &= \mathbb{N} && \text{(Befehlszähler)}\\
DK &= \mathbb{Z}^* && \text{(Datenkeller)}\\
HS &= \{h \mid h\colon \mathbb{N} \dashrightarrow \mathbb{Z}\} && \text{(Hauptspeicher)}\\
Inp &= \mathbb{Z}^* && \text{(Eingabeband)}\\
Out &= \mathbb{Z}^* && \text{(Ausgabeband)}
\end{aligned}
$$

Die Maschine besteht also aus fünf Komponenten. Der Befehlszähler gibt die Adresse des nächsten auszuführenden Befehls an, der Datenkeller wird zur Auswertung von Ausdrücken benötigt und im Hauptspeicher werden Variablenwerte abgelegt. Auf dem Eingabeband stehen zu verarbeitende Daten, und Ergebnisse werden auf dem Ausgabeband abgelegt. Das auszuführende AM_0-Programm wird nicht in der Maschine gespeichert. Ein Zustand der Maschine ist also ein Quintupel, das den jeweiligen Inhalt der fünf Komponenten angibt:

Zustand der AM_0: $s = (m, d, h, inp, out) \in AM_0$

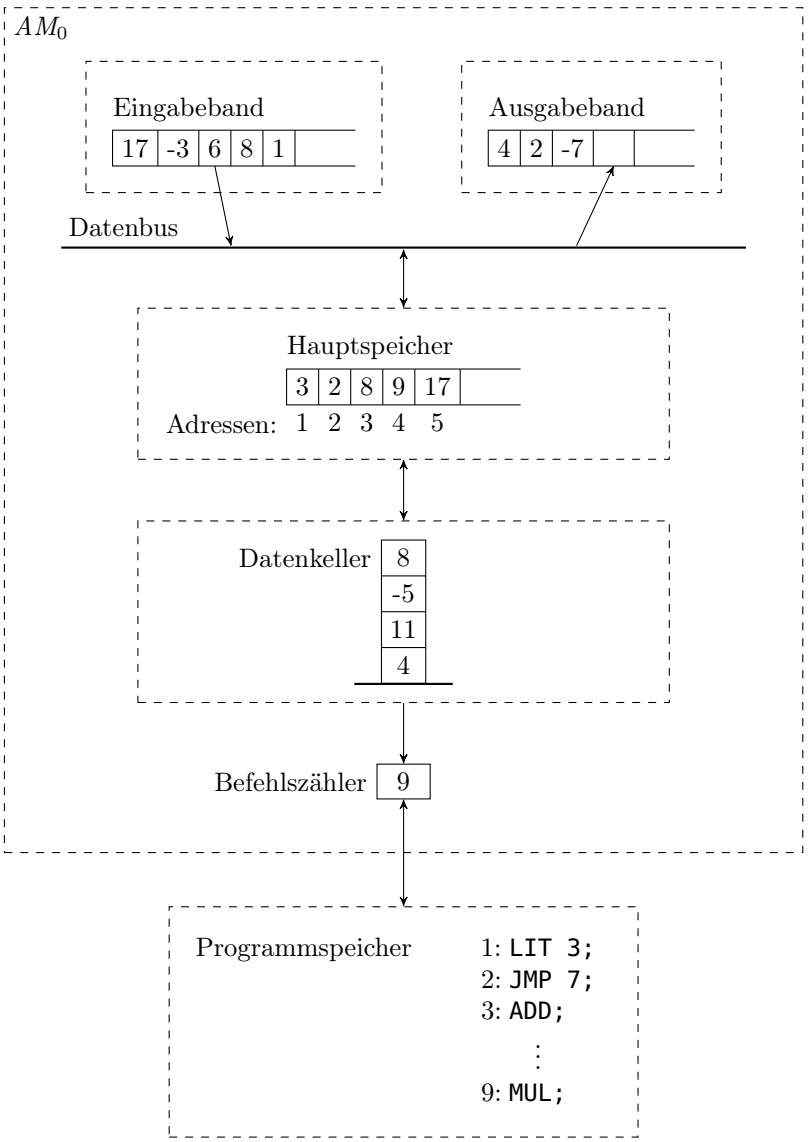


Abbildung 17.1: Die Abstrakte Maschine AM_0.

Der Datenkeller d besteht aus beliebig vielen „übereinander liegenden“ Einträgen $d.1$, $d.2$, ..., $d.n$ (mit $n \geq 0$) von ganzen Zahlen. Dabei ist $d.1$ der oberste Eintrag (falls vorhanden), also die Spitze des Datenkellers. Die Zahl $d.i$ ist somit von der Spitze gesehen der i-te Eintrag. Wir notieren den Datenkeller, indem wir die einzelnen Einträge nebeneinander schreiben und jeweils durch das Symbol : trennen. Als Argument für eine Operation dürfen nur die obersten beiden Kellereinträge benutzt werden; es kann nur der oberste Kellereintrag gelöscht werden; ein neuer Eintrag kann nur oben auf dem Keller abgelegt werden. Die Spitze des Datenkellers ($d.1$) befindet sich also immer links:

$$d = d.1 : d.2 : ... : d.n \qquad \text{mit } d.i \in \mathbb{Z} \text{ für } 1 \leq i \leq n$$

Der Hauptspeicher wird formal durch eine partielle Abbildung $h\colon \mathbb{N} \rightarrow \mathbb{Z}$ beschrieben, die zu einem Speicherplatz seinen jeweiligen Inhalt (falls definiert) angibt.

17.1.3 Befehle der AM_0, deren Semantik und Programmsemantik

Wir bezeichnen die Menge der Befehle der AM_0 durch Γ und unterscheiden vier Arten von Befehlen:

arithmetische und logische Befehle: ADD, MUL, SUB, DIV, MOD, EQ (equal), NE (not equal), LT (less than), GT (greater than), LE (less equal), GE (greater equal). Diese Befehle wirken sich nur auf den Datenkeller aus.

Transportbefehle: LOAD n, STORE n (mit $n \in \mathbb{N}$), LIT z (mit $z \in \mathbb{Z}$). Diese Befehle stellen die Verbindung zwischen Datenkeller und Hauptspeicher her.

Sprungbefehle: JMP n, JMC n (mit $n \in \mathbb{N}$). Diese Befehle verändern den Befehlszähler.

Schreib- und Lesebefehle: WRITE n, READ n (mit $n \in \mathbb{N}$). Diese Befehle verknüpfen den Hauptspeicher mit dem Ausgabe- bzw. Eingabeband.

Programme für die AM_0:

Ein AM_0-Programm P ist eine partielle Funktion $P\colon \mathbb{N} \dashrightarrow \Gamma$, so dass ein $k \in \mathbb{N}$ existiert mit $def(P) = \{1, \ldots, k\}$, wobei $def(P) := \{n \mid P(n) \text{ ist definiert}\}$.

Wir schreiben für $P : \mathbb{N} \dashrightarrow \Gamma$ auch $1\colon \gamma_1;\ \ 2\colon \gamma_2;\ \ \ldots;\ \ k\colon \gamma_k$; mit $k \in \mathbb{N}$ und $\gamma_j = P(j)$ für $1 \leq j \leq k$.

Die Menge aller AM_0-Programme bezeichnen wir durch $Prog_0$.

Jetzt müssen wir die Bedeutung, Auswirkung oder Semantik der einzelnen Befehle beschreiben und definieren, wie ein AM_0-Programm abläuft (Programmsemantik).

Befehlssemantik der AM_0:

Die Befehlssemantik $\mathcal{C}[\![.]\!]\colon \Gamma \to (AM_0 \dashrightarrow AM_0)$ gibt zu jedem Befehl der AM_0 an, welche Zustandstransformation der AM_0 dieser Befehl bewirkt. Dabei schreiben wir statt $\mathcal{C}[\![.]\!](\gamma)$ nun $\mathcal{C}[\![\gamma]\!]$.

- $\mathcal{C}[\![\text{ADD}]\!](m, d, h, inp, out) :=$
  if $d = d.1 : d.2 : d.3 : \ldots : d.n$ mit $n \geq 2$ then $(m+1, (d.2 + d.1) : d.3 : \ldots : d.n, h, inp, out)$

  für MUL analog

  $\mathcal{C}[\![\text{SUB}]\!](m, d, h, inp, out) :=$
  if $d = d.1 : d.2 : d.3 : \ldots : d.n$ mit $n \geq 2$ then $(m+1, (d.2 - d.1) : d.3 : \ldots : d.n, h, inp, out)$

  für DIV und MOD analog

  $\mathcal{C}[\![\text{LT}]\!](m, d, h, inp, out) :=$
  if $d = d.1 : d.2 : d.3 : \ldots : d.n$ mit $n \geq 2$ then $(m+1, b : d.3 : \ldots : d.n, h, inp, out)$
  wobei $b = 1$, falls $d.2 < d.1$, und $b = 0$, falls $d.2 \geq d.1$,
  d. h. für den Wert *true* (bzw. *false*) wird 1 (bzw. 0) abgelegt

  für EQ, NE, GT, LE und GE analog
- $\mathcal{C}[\![\text{LOAD } n]\!](m, d, h, inp, out) :=$
  if $h(n) \in \mathbb{Z}$ then $(m+1, h(n) : d, h, inp, out)$

  $\mathcal{C}[\![\text{LIT } z]\!](m, d, h, inp, out) := (m+1, z : d, h, inp, out)$

  $\mathcal{C}[\![\text{STORE } n]\!](m, d, h, inp, out) :=$
  if $d = d.1 : d'$ then $(m+1, d', h[n/d.1], inp, out)$
  wobei $h[n/d.1](k) = \begin{cases} d.1 & \text{falls } k = n \\ h(k) & \text{sonst} \end{cases}$
- $\mathcal{C}[\![\text{JMP } e]\!](m, d, h, inp, out) := (e, d, h, inp, out)$

  $\mathcal{C}[\![\text{JMC } e]\!](m, d, h, inp, out) :=$
  if $d = 0 : d.2 : \ldots : d.n$ mit $n \geq 1$ then $(e, d.2 : \ldots : d.n, h, inp, out)$
  if $d = 1 : d.2 : \ldots : d.n$ mit $n \geq 1$ then $(m+1, d.2 : \ldots : d.n, h, inp, out)$

  Es wird also zum Befehl mit der Nummer e gesprungen, wenn das oberste Kellerelement gleich 0 ist; die 0 repräsentiert den Wert *false*. Wenn das oberste Kellerelement gleich 1 ist (und damit den Wert *true* repräsentiert), dann wird der Befehlszähler um 1 inkrementiert.
- $\mathcal{C}[\![\text{READ } n]\!](m, d, h, inp, out) :=$
  if $inp = first(inp).rest(inp)$ then $(m+1, d, h[n/first(inp)], rest(inp), out)$
  wobei für jedes $n \in \mathbb{Z}$ und $w \in \mathbb{Z}^*$ gilt: $first(n : w) = n$ und $rest(n : w) = w$

$\mathcal{C}[\![\text{WRITE } n]\!](m, d, h, inp, out) :=$
if $h(n) \in \mathbb{Z}$ then $(m+1, d, h, inp, out : h(n))$

Iterationssemantik der AM_0:

Sei $Prog_0$ die Menge der AM_0-Programme und $P \in Prog_0$.
$\mathcal{I}[\![.]\!] : Prog_0 \to (AM_0 \dashrightarrow AM_0)$.

$$\mathcal{I}[\![P]\!](m, d, h, inp, out) := \begin{cases} \mathcal{I}[\![P]\!](\mathcal{C}[\![P(m)]\!](m, d, h, inp, out)), & \text{falls } m \in def(P), \\ (m, d, h, inp, out), & \text{falls } m \notin def(P) \end{cases}$$

Programmsemantik der AM_0:

$\mathcal{P}[\![.]\!] : Prog_0 \to (Inp \dashrightarrow Out)$

$\mathcal{P}[\![P]\!](inp) := proj_5^{(5)}(\mathcal{I}[\![P]\!](1, \varepsilon, h_\emptyset, inp, \varepsilon))$,
für alle $P \in Prog_0$. Dabei gilt:

1 ist die Befehlsmarke des ersten Befehls,
$h_\emptyset(n)$ ist undefiniert für alle $n \in \mathbb{N}$ und
$proj_5^{(5)} : AM_0 \to Out$ mit $proj_5^{(5)}(m, d, h, inp, out) := out$.

17.1.4 Übersetzung von C_0-Programmen in AM_0-Programme

Wir benötigen für die verschiedenen syntaktischen Konstrukte verschiedene Übersetzungsfunktionen.

Zunächst werden wir die Übersetzung in „AM_0-Programme mit *baumstrukturierten Adressen*“ vornehmen. Diese Programme werden anschließend in die eigentlichen AM_0-Programme übersetzt. Eine baumstrukturierte Adresse hat dabei die folgende Gestalt:

$i_1.i_2.i_3. \ \ldots \ .i_n$ mit $i_j \in \mathbb{N}$ für alle $1 \leq j \leq n$.

Die Menge aller AM_0-Programme mit baumstrukturierten Adressen wird durch $bProg_0$ bezeichnet.

Für die Übersetzung wird eine zusätzliche Menge *Tab* von *Symboltabellen* benötigt, die wie folgt definiert ist:

$$Tab := \{tab \mid tab\colon W(\langle \text{Ident} \rangle) \dashrightarrow (\{\text{const}\} \times \mathbb{Z}) \cup (\{\text{var}\} \times \mathbb{N})\}$$

- **Übersetzung von BoolExpressions:**

  boolexptrans: $\text{W}(\langle \text{BoolExpression} \rangle) \times Tab \dashrightarrow bProg_0$

  boolexptrans$(se_1\ rel\ se_2, tab) :=$
  simpleexptrans(se_1, tab)
  simpleexptrans(se_2, tab)
  REL;
  für alle $se_1, se_2 \in \text{W}(\langle$ SimpleExpression $\rangle)$, $rel \in \{$`==`, `!=`, `<`, `>`, `<=`, `>=`$\}$
  und $tab \in Tab$,
  wobei REL der zu *rel* gehörige Befehl ist, z. B. ist REL = LE, falls $rel =$ `<=`.

- **Übersetzung von SimpleExpressions:**

  simpleexptrans: $\text{W}(\langle \text{SimpleExpression} \rangle) \times Tab \dashrightarrow bProg_0$

  Diese Übersetzung wollen wir nicht mehr formal spezifizieren, vielmehr soll in pragmatischer Weise der Mechanismus benannt und anhand von Beispielen gefestigt werden.

  Ein wesentliches Merkmal der Übersetzung ist das Laden der Operanden des Ausdrucks in der Reihenfolge von links nach rechts. Soll eine Variable geladen werden, so geschieht dies mit dem Befehl LOAD, handelt es sich um eine Konstante, so nutzen wir den Befehl LIT. Des weiteren ist zu beachten, dass die definierten Operationen grundsätzlich in der postfix-Schreibweise durch die entsprechenden Operationssymbole notiert werden und dass bei dieser Notation die Vorrangbeziehungen zwischen den Operationen erhalten bleiben. D. h., die Reihenfolge der Operationssymbole wird nur ausnahmsweise der Reihenfolge im zu übersetzenden Ausdruck folgen.

Übersetzungsbeispiele für SimpleExpressions (Ausdrücke):

Angenommen, die Symboltabelle *tab* hat die folgenden Einträge:

id	a	x	y	z
tab(*id*)	(const, 10)	(var, 1)	(var, 2)	(var, 3)

Dann liefert die Übersetzung der Ausdrücke (a) bis (c) jeweils die angegebenen $bProg_0$-Zeilen:

(a) *simpleexptrans* `(y + z * x, tab)` ⇒ LOAD 2; LOAD 3; LOAD 1; MUL; ADD;

(b) *simpleexptrans*`(x * z - y % a, tab)` ⇒ LOAD 1; LOAD 3; MUL; LOAD 2;
LIT 10; MOD; SUB;

(c) *simpleexptrans*`(z * (x+a) / y - (x-y) * a, tab)` ⇒
LOAD 3; LOAD 1; LIT 10; ADD; MUL; LOAD 2; DIV;
LOAD 1; LOAD 2; SUB; LIT 10; MUL; SUB;

- **Übersetzung von Anweisungen:**

Hier muss noch ein zusätzliches Argument bei den Übersetzungsfunktionen mitgeführt werden, um die nächste freie baumstrukturierte Adresse für einen AM_0-Befehl im zu konstruierenden Programm zu kennen:

stseqtrans: W(⟨StatementSequence⟩) × *Tab* × $\mathbb{N}^*$ -↛ $bProg_0$

stseqtrans($stat_1\ stat_2\ \ldots\ stat_n, tab, a$) :=
sttrans($stat_1, tab, a.1$)
sttrans($stat_2, tab, a.2$)
...
sttrans($stat_n, tab, a.n$)
für alle $stat_1, stat_2, \ldots, stat_n \in$ W(⟨ Statement ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$.

sttrans: W(⟨Statement⟩) × *Tab* × $\mathbb{N}^*$ -↛ $bProg_0$
sttrans(`{` $stat_1\ stat_2\ \ldots\ stat_n$ `}`, tab, a) :=
stseqtrans($stat_1\ stat_2\ \ldots\ stat_n, tab, a$)
für alle $stat_1, stat_2, \ldots, stat_n \in$ W(⟨ Statement ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$.

sttrans(id `=` *exp*`;`, tab, a) :=
if $tab(id) = (\text{var}, n)$ then *simpleexptrans*(*exp*, *tab*) STORE n;
für alle $id \in$ W(⟨ Ident ⟩), *exp* ∈ W(⟨ SimpleExpression ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$,

sttrans(`scanf("%d",&`id`);`, tab, a) :=
if $tab(id) = (\text{var}, n)$ then READ n;
für alle $id \in$ W(⟨ Ident ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$,

sttrans(`printf("%d",`id`);`, tab, a) :=
if $tab(id) = (\text{var}, n)$ then WRITE n;
für alle $id \in$ W(⟨ Ident ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$,

sttrans(`if (` *exp* `)` *stat*, tab, a) :=
boolexptrans(*exp*, *tab*)
JMC $a.1$;
sttrans(*stat*, *tab*, $a.2$)
$a.1$:
für alle *exp* ∈ W(⟨ BoolExpression ⟩), *stat* ∈ W(⟨ Statement ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$,

sttrans(`if (` *exp* `)` $stat_1$ `else` $stat_2, tab, a$) :=
boolexptrans(*exp*, *tab*)
JMC $a.1$;
sttrans($stat_1, tab, a.2$)
JMP $a.3$;
$a.1$: *sttrans*($stat_2, tab, a.4$)
$a.3$:
für alle *exp* ∈ W(⟨ BoolExpression ⟩), $stat_1, stat_2 \in$ W(⟨ Statement ⟩), $tab \in Tab$ und $a \in \mathbb{N}^*$,

sttrans(`while (` *exp* `)` *stat*, *tab*, *a*) :=
a.1 : *boolexptrans*(*exp*, *tab*)
JMC *a*.2;
sttrans(*stat*, *tab*, *a*.3)
JMP *a*.1;
a.2:
für alle *exp* ∈ W(⟨ BoolExpression ⟩), *stat* ∈ W(⟨ Statement ⟩), *tab* ∈ *Tab* und *a* ∈ $\mathbb{N}^*$.

- **Aufbau von Symboltabellen durch Deklarationen:**

  Die Deklaration von Konstanten und Variablen bewirkt entsprechende Einträge in der Symboltabelle:

  update: W(⟨Declaration⟩) × *Tab* -↛ *Tab*

  update(`const` id_1 `=` z_1`, ... ,` id_n `=` z_n`; int` id'_1`,...,`id'_m`;`, *tab*) :=
  $tab[id_1/(\text{const}, z_1), \ldots, id_n/(\text{const}, z_n), id'_1/(\text{var}, 1), \ldots, id'_m/(\text{var}, m)]$
  für alle $id_1, \ldots, id_n, id'_1, \ldots, id'_m \in$ W(⟨ Ident ⟩),
  $z_1, \ldots, z_n \in \{\varepsilon, -\}$·W(⟨ Number ⟩) und *tab* ∈ *Tab*.

- **Übersetzung von C_0-Programmen in $bProg_0$-Programme**

  trans: W(⟨Program⟩) -↛ $bProg_0$

  trans(`#include <stdio.h> int main()` *block*) := *blocktrans*(*block*)
  für alle *block* ∈ W(⟨ Block ⟩),

  blocktrans: W(⟨Block⟩) -↛ $bProg_0$

  blocktrans(`{` *decl statseq* `return 0;}`) := *stseqtrans*(*statseq*, *update*(*decl*, $tab_\emptyset$), 1)
  für alle *decl* ∈ W(⟨ Declaration ⟩) und *statseq* ∈ W(⟨ StatementSequence ⟩),
  wobei $tab_\emptyset \in Tab$ mit *graph*($tab_\emptyset$) = ∅.

Beispiel 17.0 (Fortsetzung)**.** Das C_0-Programm zur Summation von Quadratzahlen soll hier zunächst in ein AM_0-Programm `bprog0` mit baumstrukturierten Adressen übersetzt werden:

trans(`#include <stdio.h> int main () {... return 0;}`)

= *blocktrans*(`{int i,n,s; scanf("%d",&n); ... printf("%d",s); return 0;}`)

= *stseqtrans*(`scanf("%d",&n); ... printf("%d",s);`, *update*(`int i,n,s;`, $tab_\emptyset$), 1)

= *stseqtrans*(`scanf("%d",&n); ... printf("%d",s);`, $\underbrace{tab_\emptyset[i/(\text{var}, 1), n/(\text{var}, 2), s/(\text{var}, 3)]}_{tab_1}$, 1)

= *sttrans*(`scanf("%d",&n);`, tab_1, 1.1)
sttrans(`i=1;`, tab_1, 1.2)
sttrans(`s=0;`, tab_1, 1.3)
sttrans(`while (i<=n) {s=s+i*i; i=i+1;}`, tab_1, 1.4)
sttrans(`printf("%d",s);`, tab_1, 1.5)

= `READ 2;`
simpleexptrans(`1`, tab_1) `STORE 1;`
simpleexptrans(`0`, tab_1) `STORE 3;`
`1.4.1:` *boolexptrans*(`i<=n`, tab_1)
`JMC 1.4.2;`
sttrans(`{s=s+i*i; i=i+1;}`, tab_1, 1.4.3)
`JMP 1.4.1;`
`1.4.2: WRITE 3;`

$=$ `READ 2;`
`LIT 1; STORE 1;`
`LIT 0; STORE 3;`
`1.4.1:` $simpleexptrans(\texttt{i}, tab_1)$ $simpleexptrans(\texttt{n}, tab_1)$ `LE;`
`JMC 1.4.2;`
$stseqtrans\big(\texttt{s=s+i*i; i=i+1;}, tab_1, 1.4.3\big)$
`JMP 1.4.1;`
`1.4.2: WRITE 3;`

$=$ `READ 2;`
`LIT 1; STORE 1;`
`LIT 0; STORE 3;`
`1.4.1: LOAD 1; LOAD 2; LE;`
`JMC 1.4.2;`
$sttrans\big(\texttt{s=s+i*i;}, tab_1, 1.4.3\big)$
$sttrans\big(\texttt{i=i+1;}, tab_1, 1.4.3\big)$
`JMP 1.4.1;`
`1.4.2: WRITE 3;`

$=$ `READ 2;`
`LIT 1; STORE 1;`
`LIT 0; STORE 3;`
`1.4.1: LOAD 1; LOAD 2; LE;`
`JMC 1.4.2;`
$simpleexptrans\big(\texttt{s=s+i*i}, tab_1\big)$ `STORE 3;`
$simpleexptrans\big(\texttt{i=i+1}, tab_1\big)$ `STORE 1;`
`JMP 1.4.1;`
`1.4.2: WRITE 3;`

$=$ `READ 2;`
`LIT 1; STORE 1;`
`LIT 0; STORE 3;`
`1.4.1: LOAD 1; LOAD 2; LE;`
`JMC 1.4.2;`
`LOAD 3; LOAD 1; LOAD 1; MUL; ADD; STORE 3;`
`LOAD 1; LIT 1; ADD; STORE 1;`
`JMP 1.4.1;`
`1.4.2: WRITE 3;` □

Nun werden $bProg_0$-Programme mit baumstrukturierten Adressen in $Prog_0$-Programme überführt. Danach besitzt jeder Befehl eine lineare Adresse (oder: Marke), die eine natürliche Zahl ist.

Sei P ein $bProg_0$-Programm. Führe folgende 2 Schritte zur Berechnung des zugehörigen $Prog_0$-Programms P' durch:

1. Nummeriere die Befehle von P, beginnend mit 1 der Reihe nach durch, und merke die Paare (a, a') in einer Menge K, wobei a eine baumstrukturierte Adresse eines Befehls ist, die durch die Übersetzung eines C_0-Programms entstand, und a' die Adresse desselben Befehls ist, die durch die Nummerierung entstand.
2. Ersetze jeden Sprungbefehl JMP a oder JMC b durch JMP a' bzw. JMC b', wenn das Paar (a, a') bzw. (b, b') in K liegt.

Beispiel 17.0 (Fortsetzung)**.** Wendet man den obigen Algorithmus auf unser Beispielprogramm `bprog0` an, so erhält man das folgende Programm `prog0`:

```
 1:  READ 2;          7:  LOAD 2;          13:  MUL;            19:  STORE 1;
 2:  LIT 1;           8:  LE;              14:  ADD;            20:  JMP 6;
 3:  STORE 1;         9:  JMC 21;          15:  STORE 3;        21:  WRITE 3;
 4:  LIT 0;          10:  LOAD 3;          16:  LOAD 1;
 5:  STORE 3;        11:  LOAD 1;          17:  LIT 1;
 6:  LOAD 1;         12:  LOAD 1;          18:  ADD;
```

Der Ablauf des Programms `prog0` auf der abstrakten Maschine AM_0 wird für die Eingabe 2 in Abbildung 17.2 aufgeführt (wobei eine Zeile einem Maschinenzustand entspricht). Dann gilt: $\mathcal{P}[\![\texttt{prog0}]\!](2) = proj_5^{(5)}(\mathcal{I}[\![\texttt{prog0}]\!](1, \varepsilon, h_\emptyset, 2, \varepsilon)) = 5$. □

Eine Nachbetrachtung: Natürlich kann man die operationelle Semantik von C_0 als *Definition* der Semantik von C_0-Programmen auffassen. Üblicherweise definiert man aber zunächst eine denotationelle Semantik und zeigt dann, dass die operationelle Semantik mit der denotationellen Semantik übereinstimmt. Die denotationelle Semantik fungiert also quasi als eine Referenzsemantik, in der auf abstraktem Niveau die Bedeutung von Programmen festgelegt wird. Alle Implementierungen müssen dann korrekt sein bezüglich der denotationellen Semantik.

17.2 $C_1 = C_0$ + Funktionen ohne Rückgabewert

In diesem Abschnitt wollen wir eine Erweiterung von C_0 implementieren, nämlich C_1. Grob gesagt erhält man C_1, wenn man C_0 um Funktionen ohne Rückgabewert erweitert. Wir beschreiben die Implementierung – wie üblich – durch

- die Angabe der Syntax von C_1,
- der Definition einer abstrakten Maschine AM_1 und
- einen Übersetzer von C_1- nach AM_1-Code.

17.2.1 Syntax von C_1

Beispiel 17.1. Als Einführung betrachten wir das folgende C_1-Programm `double1`, welches bei Eingabe von n den Wert $2 * n$ berechnet.

```
 1  #include <stdio.h>
 2  int a, b;
 3
 4  void double(int x, int *y)
 5  { if (x > 0)
 6    { double(x - 1, y);
 7      *y = *y + 2;
 8    } else
 9      *y = 0;
10  }
11
12  void main()
13  { scanf("%d", &a);
14    double(a, &b);
15    printf("%d", b);
16  }
```

□

(BZ,	DK, HS	, Inp,	$\underline{Out}$)
(1,	ε, $h_\emptyset = [\,]$	, 2,	ε)
(2,	ε, $[2/2]$	, ε,	ε)
(3,	1, $[2/2]$	, ε,	ε)
(4,	ε, $[1/1, 2/2]$	, ε,	ε)
(5,	0, $[1/1, 2/2]$	, ε,	ε)
(6,	ε, $[1/1, 2/2, 3/0]$,	ε,	ε)
(7,	1, $[1/1, 2/2, 3/0]$,	ε,	ε)
(8,	$2:1$, $[1/1, 2/2, 3/0]$,	ε,	ε)
(9,	1, $[1/1, 2/2, 3/0]$,	ε,	ε)
(10,	ε, $[1/1, 2/2, 3/0]$,	ε,	ε)
(11,	0, $[1/1, 2/2, 3/0]$,	ε,	ε)
(12,	$1:0$, $[1/1, 2/2, 3/0]$,	ε,	ε)
(13,	$1:1:0$, $[1/1, 2/2, 3/0]$,	ε,	ε)
(14,	$1:0$, $[1/1, 2/2, 3/0]$,	ε,	ε)
(15,	1, $[1/1, 2/2, 3/0]$,	ε,	ε)
(16,	ε, $[1/1, 2/2, 3/1]$,	ε,	ε)
(17,	1, $[1/1, 2/2, 3/1]$,	ε,	ε)
(18,	$1:1$, $[1/1, 2/2, 3/1]$,	ε,	ε)
(19,	2, $[1/1, 2/2, 3/1]$,	ε,	ε)
(20,	ε, $[1/2, 2/2, 3/1]$,	ε,	ε)
(6,	ε, $[1/2, 2/2, 3/1]$,	ε,	ε)
(7,	2, $[1/2, 2/2, 3/1]$,	ε,	ε)
(8,	$2:2$, $[1/2, 2/2, 3/1]$,	ε,	ε)
(9,	1, $[1/2, 2/2, 3/1]$,	ε,	ε)
(10,	ε, $[1/2, 2/2, 3/1]$,	ε,	ε)
(11,	1, $[1/2, 2/2, 3/1]$,	ε,	ε)
(12,	$2:1$, $[1/2, 2/2, 3/1]$,	ε,	ε)
(13,	$2:2:1$, $[1/2, 2/2, 3/1]$,	ε,	ε)
(14,	$4:1$, $[1/2, 2/2, 3/1]$,	ε,	ε)
(15,	5, $[1/2, 2/2, 3/1]$,	ε,	ε)
(16,	ε, $[1/2, 2/2, 3/5]$,	ε,	ε)
(17,	2, $[1/2, 2/2, 3/5]$,	ε,	ε)
(18,	$1:2$, $[1/2, 2/2, 3/5]$,	ε,	ε)
(19,	3, $[1/2, 2/2, 3/5]$,	ε,	ε)
(20,	ε, $[1/3, 2/2, 3/5]$,	ε,	ε)
(6,	ε, $[1/3, 2/2, 3/5]$,	ε,	ε)
(7,	3, $[1/3, 2/2, 3/5]$,	ε,	ε)
(8,	$2:3$, $[1/3, 2/2, 3/5]$,	ε,	ε)
(9,	0, $[1/3, 2/2, 3/5]$,	ε,	ε)
(21,	ε, $[1/3, 2/2, 3/5]$,	ε,	ε)
(22,	ε, $[1/3, 2/2, 3/5]$,	ε,	5)

Abbildung 17.2: Ablauf des Programms `prog0` aus Beispiel 17.0 für die Eingabe 2.

Hier geben wir nur die in Bezug auf C_0 veränderten EBNF-Regeln an.

⟨Program⟩ ::= `#include <stdio.h>`
⟨Declaration⟩
{ `void` ⟨Ident⟩ `(` [⟨FormalParam⟩] `);` }
{ `void` ⟨Ident⟩ `(` [⟨FormalParam⟩] `)` ⟨Block⟩ }
`void main()` ⟨Block⟩

⟨FormalParam⟩ ::= `int` ⟨Ident⟩ { `, int` ⟨Ident⟩ } { `, int *` ⟨Ident⟩ }

Bezüglich C_0 ist also eine globale Deklaration hinzugekommen; außerdem können Funktionen (mit leerem Ergebnistyp) vor der Hauptfunktion deklariert werden. Die Hauptfunktion ist nun ebenfalls vom Ergebnistyp void. Die vorangestellte Liste von Funktionsköpfen dient der Forwarddeklaration. Als formale Parameter sind Ganzzahlen sowohl als Wert- als auch als Referenzparameter (Zeiger) möglich, wobei wir zur Vereinfachung der Übersetzung zusätzlich fordern, dass die Wertparameter stets vor den Referenzparametern stehen.

$$\langle \text{Block} \rangle ::= \texttt{\{}\ \langle \text{Declaration} \rangle\ \hat{[}\ \langle \text{StatementSequence} \rangle\ \hat{]}\ \texttt{\}}$$

Im Unterschied zu C_0 enden C_1-Blöcke nicht mit einem return 0; das ist auch klar, weil in C_1 alle Funktionen den Ergebnistyp void haben.

$$\begin{aligned}\langle \text{Statement} \rangle ::= {} & \langle \text{Ident} \rangle\ \texttt{(}\ \hat{[}\langle \text{ValueParam} \rangle\ \texttt{,}\ \langle \text{RefParam} \rangle\ \hat{]}\ \hat{|}\ \langle \text{ValueParam} \rangle\ \hat{[}\ \texttt{,}\ \langle \text{RefParam} \rangle\ \hat{]}\ \texttt{);}\ \hat{|} \\ & \langle \text{Assignment} \rangle\ \hat{|}\ \langle \text{IfStatement} \rangle\ \hat{|}\ \langle \text{WhileStatement} \rangle\ \hat{|}\ \langle \text{CompStatement} \rangle\ \hat{|} \\ & \texttt{scanf("\%d",}\ \hat{[}\texttt{\&}\hat{]}\ \langle \text{Ident} \rangle\ \texttt{);}\ \hat{|}\ \texttt{printf("\%d",}\ \hat{[}\texttt{*}\hat{]}\ \langle \text{Ident} \rangle\ \texttt{);}\end{aligned}$$

$$\langle \text{ValueParam} \rangle ::= \langle \text{SimpleExpression} \rangle\ \hat{\{}\ \texttt{,}\ \langle \text{SimpleExpression} \rangle\ \hat{\}}$$

$$\langle \text{RefParam} \rangle ::= \hat{[}\texttt{\&}\hat{]}\ \langle \text{Ident} \rangle\ \hat{\{}\ \texttt{,}\ \hat{[}\texttt{\&}\hat{]}\ \langle \text{Ident} \rangle\ \hat{\}}$$

Es ist also möglich, Funktionen aufzurufen.

$$\langle \text{Assignment} \rangle ::= \hat{[}\texttt{*}\hat{]}\ \langle \text{Ident} \rangle\ \texttt{=}\ \langle \text{SimpleExpression} \rangle\ \texttt{;}$$

$$\langle \text{Factor} \rangle ::= \hat{[}\texttt{*}\hat{]}\ \langle \text{Ident} \rangle\ \hat{|}\ \langle \text{Number} \rangle\ \hat{|}\ \texttt{(}\ \langle \text{SimpleExpression} \rangle\ \texttt{)}$$

Offensichtlich müssen wir in C_1 zwischen globalen Variablen und lokalen Variablen unterscheiden. Außerdem sind einige zusätzliche kontextsensitive Nebenbedingungen erforderlich:

- Wenn in einer Funktion f eine andere Funktion g aufgerufen wird, dann muss vor der Deklaration von f die Funktion g vollständig deklariert sein oder zumindest deren Funktionskopf angegeben sein, und der Aufruf muss in Anzahl und Art der Parameter der Deklaration entsprechen.
- Als Referenzparameter dürfen nur die Werte von anderen Referenzparametern sowie die mit dem &-Operator ermittelten Adressen von lokalen und globalen Variablen sowie Wertparametern übergeben werden.
- Nur Referenzparameter können mit dem $*$-Operator dereferenziert werden.
- Referenzparametern dürfen keine Werte zugewiesen werden (dem jeweiligen referenzierten Speicherbereich aber sehr wohl).

17.2.2 Abstrakte Maschine AM_1

Wie bei der AM_0 geben wir die AM_1 als kartesisches Produkt von Mengen an.
$AM_1 = BZ \times DK \times LK \times REF \times Inp \times Out$ mit

$$\begin{aligned} BZ &= \mathbb{N} && \text{(Befehlszähler)} \\ DK &= \mathbb{Z}^* && \text{(Datenkeller)} \\ LK &= \mathbb{Z}^* && \text{(Laufzeitkeller)} \\ REF &= \mathbb{N} && \text{(Referenzzeiger)} \\ Inp &= \mathbb{Z}^* && \text{(Eingabeband)} \\ Out &= \mathbb{Z}^* && \text{(Ausgabeband)} \end{aligned}$$

Eine Konfiguration hat die Form $(m, d, h, r, inp, out) \in AM_1$. Man erkennt, dass ein Referenzzeiger r hinzugekommen ist, und der Hauptspeicher ist durch einen Laufzeitkeller (kurz: LK) ersetzt worden. Der Referenzzeiger zeigt auf ein bestimmtes Feld des LK, welcher nun ein Wort

$$h.1 : h.2 : \ldots : h.n$$

mit $n \geq 1$ und $h.i \in \mathbb{Z}$ ist; die Spitze des LK ist rechts. Der LK ist logisch in einen globalen Variablenbereich gv und Aktivierungsblöcke (engl. activation records) $ar_1, \ldots, ar_k$ wie folgt aufgeteilt:

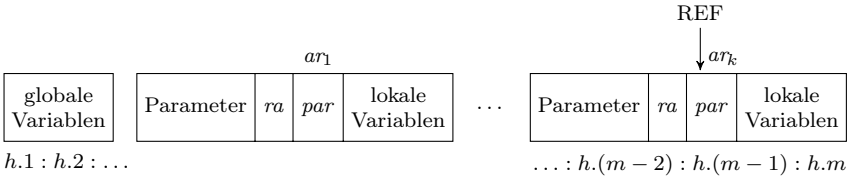
$$gv : ar_1 : \ldots : ar_k$$

genauer:

$$\underbrace{h.1 : h.2 : \ldots : h.i}_{gv} : \underbrace{h.(i+1) : \ldots : h.j}_{ar_1} : \ldots : \underbrace{h.(l+1) : \ldots : h.n}_{ar_k}$$

Im globalen Variablenbereich gv werden die Werte der global deklarierten Variablen gespeichert (im Beispiel 17.1 ist das ein Speicherplatz für `a` und einer für `b`).

Bei jedem Funktionsaufruf wird ein neuer Aktivierungsblock auf LK gelegt (siehe Abbildung 17.3); wenn eine Prozedur abgearbeitet ist, dann wird der oberste (am weitesten rechts stehende) Aktivierungsblock vom LK genommen (vgl. pulsierender Speicher, Abschnitt 5.3).



Auflegen eines neuen Aktivierungsblocks ar_{k+1}:

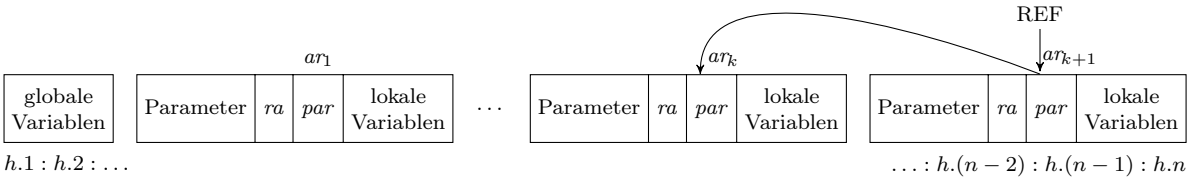


Abbildung 17.3: Aktivierungsblöcke in C_1

Jeder Aktivierungsblock ar_i hat den folgenden logischen Aufbau:

$$\boxed{p_1 : \ldots : p_m : p_{m+1} : \ldots : p_k : ra : par : l_1 : \ldots : l_j}$$

wobei $p_1, \ldots, p_m$ die Werte der Wertparameter der aufgerufenen Funktion sind, $p_{m+1}, \ldots, p_k$ die Werte derer Referenzparameter, ra die Rücksprungadresse ($r$eturn $a$ddress) ist, par ein Verweis auf den vorherigen Aktivierungsblock ($p$revious $a$ctivation $r$ecord) ist, und $l_1, \ldots, l_j$ die Werte der lokalen Variablen der aufgerufenen Funktion sind. Der par eines Aktivierungsblocks ist sein Referenzpunkt. Der Referenzzeiger r zeigt immer auf den Referenzpunkt des obersten Aktivierungsblocks. In jedem Aktivierungsblock zeigt par auf den Referenzpunkt des darunterliegenden Aktivierungsblocks.

17.2.3 Befehle der AM_1, deren Semantik und Programmsemantik

Die AM_1 bietet an:

- arithmetische und logische Befehle ADD, SUB, MUL, DIV, EQ, ...,
- Transportbefehle LOAD(b,o), LOADA(b,o), LOADI(o) STORE(b,o), STOREI(o) und LIT z, wobei $b \in$ {global, lokal} und o und z ganze Zahlen sind,
- Sprungbefehle JMP m und JMC m, wobei m eine natürliche Zahl ist,
- Schreib- und Lesebefehle WRITE(b,o), WRITEI(o), READ(b,o) und READI(o) sowie
- Prozedurbefehle PUSH, CALL adr, INIT n und RET n, wobei n und adr natürliche Zahlen sind,

Befehlssemantik der AM_1:

Die arithmetischen und logischen Befehle, der LIT-Befehl sowie die Sprungbefehle werden semantisch genauso gehandhabt wie bei der AM_0. Deshalb beschreiben wir hier nur die Wirkung der anderen Befehle.

Wir definieren zunächst eine Funktion $adr\colon \mathbb{Z}\times\{\text{lokal}, \text{global}\}\times\mathbb{Z} \to \mathbb{Z}$ wie folgt für jedes $b \in \{\text{lokal}, \text{global}\}$ und $r, o \in \mathbb{Z}$:

$$adr(r,b,o) = \begin{cases} r+o & \text{wenn } b = \text{lokal}, \\ o & \text{wenn } b = \text{global}. \end{cases}$$

$$\begin{aligned}
\mathcal{C}[\![\text{LOAD}(b,o)]\!](m,d,h,r,inp,out) &= (m+1, z:d, h, r, inp, out) \\
\mathcal{C}[\![\text{WRITE}(b,o)]\!](m,d,h,r,inp,out) &= (m+1, d, h, r, inp, out:z) \qquad \text{wobei } z = h.(adr(r,b,o)) \\
\\
\mathcal{C}[\![\text{LOADI}(o)]\!](m,d,h,r,inp,out) &= (m+1, z:d, h, r, inp, out) \\
\mathcal{C}[\![\text{WRITEI}(o)]\!](m,d,h,r,inp,out) &= (m+1, d, h, r, inp, out:z) \qquad \text{wobei } z = h.(h.(r+o)) \\
\\
\mathcal{C}[\![\text{LOADA}(b,o)]\!](m,d,h,r,inp,out) &= (m+1, adr(r,b,o):d, h, r, inp, out) \\
\\
\mathcal{C}[\![\text{STORE}(b,o)]\!](m,d,h,r,inp,out) &= \text{if } d = z:d' \text{ then } (m+1, d', h', r, inp, out) \\
\mathcal{C}[\![\text{READ}(b,o)]\!](m,d,h,r,inp,out) &= \text{if } inp = z:inp' \text{ and } z \in \mathbb{Z} \text{ then } (m+1, d, h', r, inp', out) \\
&\qquad \text{wobei } h' = h[adr(r,b,o)/z] \\
\\
\mathcal{C}[\![\text{STOREI}(o)]\!](m,d,h,r,inp,out) &= \text{if } d = z:d' \text{ then } (m+1, d', h', r, inp, out) \\
\mathcal{C}[\![\text{READI}(o)]\!](m,d,h,r,inp,out) &= \text{if } inp = z:inp' \text{ and } z \in \mathbb{Z} \text{ then } (m+1, d, h', r, inp', out) \\
&\qquad \text{wobei } h' = h[h.(r+o)/z]
\end{aligned}$$

Hinweis: $h[a/z]$ steht für die Ersetzung des a-ten Eintrags auf dem LK h durch z und ist insbesondere nicht als Division im Sinne einer Adressrechnung zu verstehen!

$$\begin{aligned}
\mathcal{C}[\![\text{PUSH}]\!](m,d,h,r,inp,out) &= \text{if } d = z:d' \text{ then } (m+1, d', h:z, r, inp, out) \\
\mathcal{C}[\![\text{CALL } adr]\!](m,d,h,r,inp,out) &= (adr, d, h:(m+1):r, length(h)+2, inp, out) \\
\mathcal{C}[\![\text{INIT } n]\!](m,d,h,r,inp,out) &= (m+1, d, h:\underbrace{0:\ldots:0}_{n}, r, inp, out) \\
\mathcal{C}[\![\text{RET } n]\!](m,d,h,r,inp,out) &= (h.(r-1), d, h.1:\ldots:h.(r-2-n), h.r, inp, out)
\end{aligned}$$

Hinweis: der Referenzzeiger ändert sich nur bei den Befehlen CALL und RET.

Iterationssemantik der AM_1:

Sei $Prog_1$ die Menge der AM_1-Programme. Die *Iterationssemantik* von $Prog_1$ hat den Typ

$$\mathcal{I}[\![.]\!]\colon Prog_1 \to (AM_1 \dashrightarrow AM_1)$$

und ist analog zur Iterationssemantik von $Prog_0$ definiert.

Programmsemantik der AM_1:

$\mathcal{P}[\![.]\!]\colon Prog_1 \to (Inp \dashrightarrow Out)$

$\mathcal{P}[\![P]\!](inp) := proj_6^{(6)}(\mathcal{I}[\![P]\!](1, \varepsilon, \varepsilon, 0, inp, \varepsilon))$, für jedes $P \in Prog_1$. Dabei gilt:

- 1 ist die Befehlsmarke des ersten Befehls,
- $proj_6^{(6)}\colon AM_1 \to Out$ mit $proj_6^{(6)}(0, \varepsilon, h, r, inp, out) := out$.

17.2.4 Übersetzung von C_1-Programmen in AM_1-Programme

Es ist klar, dass wir auch für die Übersetzung von C_1-Programmen eine Symboltabelle benötigen. Allerdings müssen wir jetzt zusätzlich noch Informationen für die deklarierten Prozeduren, Referenzparameter und für die Frage, ob eine Variable global oder lokal deklariert ist, aufnehmen. Wertparameter werden dabei wie lokale Variablen behandelt.

$$\begin{aligned} \mathit{Tab} := \{\mathit{tab} \mid \mathit{tab}\colon W(\langle\text{Ident}\rangle) \text{ -}\!\!\to (\{\text{const}\} \times \mathbb{Z}) \cup (\{\text{var}\} \times \{\text{global}, \text{lokal}\} \times \mathit{Off}) \\ \cup (\{\text{var-ref}\} \times \mathit{Off}) \cup (\{\text{proc}\} \times \mathit{Adr})\} \end{aligned}$$

wobei $\mathit{Off} = \mathbb{Z}$ (Offset) und $\mathit{Adr} = \mathbb{N}^*$ (baumstrukturierte Adresse).

Nachfolgend werden nur die im Vergleich zur Übersetzung in AM_0-Programme veränderten Regeln aufgeführt.

- **Übersetzung von C_1-Programmen in $bProg_1$-Programme**

  trans: W($\langle$Program$\rangle$) -$\!\!\to$ $bProg_1$

  trans(`#include <stdio.h>`
  decl
  fh_1 ... fh_k
  `void f1(int` $I_{1,1}$`, ..., int` I_{1,r_1}`, int *` $J_{1,1}$`, ..., int *` J_{1,s_1}`)` block_1;
  ...
  `void fm(int` $I_{m,1}$`, ..., int` I_{m,r_m}`, int *` $J_{m,1}$`, ..., int *` J_{m,s_m}`)` block_m;
  `void main()` *block*)

  := INIT *size*(*decl*); CALL m+1; JMP 0;
  blocktrans($\mathit{block}_1, \mathit{tab}_1, 1, r_1 + s_1$) ... *blocktrans*($\mathit{block}_m, \mathit{tab}_m, m, r_m + s_m$)
  blocktrans($\mathit{block}, \mathit{tab}, m+1, 0$)

  für jedes *decl* $\in$ W($\langle$Declaration$\rangle$), jede Liste fh_1 ... fh_k von Funktionsköpfen, und jede Folge $\mathit{block}, \mathit{block}_1, \ldots, \mathit{block}_m \in$ W($\langle$Block$\rangle$) von Blöcken. Hierbei ist *size*(*decl*) die Anzahl der deklarierten Variablen und

  $$\mathit{tab}' = [\texttt{f1}/(\text{proc}, 1), ..., \texttt{fm}/(\text{proc}, m)]$$

  $$\mathit{tab} = \mathit{update}(\mathit{decl}, \text{global}, \mathit{tab}')$$

  $$\begin{aligned} \mathit{tab}_i = \mathit{tab}[\quad & I_{i,1}/(\text{var}, \text{lokal}, -(r_i + s_i + 1)), ..., I_{i,r_i}/(\text{var}, \text{lokal}, -(s_i + 2)), \\ & J_{i,1}/(\text{var-ref}, -(s_i + 1)), ..., J_{i,s_i}/(\text{var-ref}, -2) \quad]. \end{aligned}$$

  blocktrans: W($\langle$Block$\rangle$) $\times$ *Tab* $\times$ *Adr* $\times \mathbb{N}$ -$\!\!\to$ $bProg_1$

  blocktrans(`{`*decl statseq*`}`, *tab*, *adr*, *n*)

  := *adr*: INIT *size*(*decl*);
  stseqtrans(*statseq*, *update*(*decl*, lokal, *tab*),*adr*)
  RET *n*;

  für alle *decl* $\in$ W($\langle$ Declaration $\rangle$) und *statseq* $\in$ W($\langle$ StatementSequence $\rangle$); size(*decl*) ist die Anzahl der in *decl* deklarierten Variablen.

- **Aufbau von Symboltabellen durch Deklarationen**

  update: W($\langle$Declaration$\rangle$) $\times$ {global, lokal} $\times$ *Tab* -$\!\!\to$ *Tab*

  update(`const` id_1`=`z_1`, ... ,`id_n `=` z_n`; int` id'_1`, ... ,`id'_m`;`, *b*, *tab*)

  := $\mathit{tab}[\mathit{id}_1/(\text{const}, z_1), \ldots, \mathit{id}_n/(\text{const}, z_n), \mathit{id}'_1/(\text{var}, b, 1), \ldots, \mathit{id}'_m/(\text{var}, b, m)]$

- **Übersetzung von Anweisungen**

  sttrans: W($\langle$Statement$\rangle$) $\times$ *Tab* $\times \mathbb{N}^*$ -$\!\!\to$ $bProg_1$
  definiert für jedes *id* $\in$ W($\langle$ Ident $\rangle$), a $\in \mathbb{N}^*$, und *tab* $\in$ *Tab*:

  sttrans(*id*`=`*exp*`;` , *tab*, *a*)

  := if *tab*(*id*) = (var, *b*, *o*) then *simpleexptrans*(*exp*, *tab*) STORE(*b*,*o*);

  sttrans(`*`*id* `=` *exp*`;` , *tab*, *a*)

:= if $tab(id) = (\text{var-ref}, o)$ then $simpleexptrans(exp, tab)$ STOREI(o);

für jedes $exp \in \text{W}(\langle \text{ SimpleExpression } \rangle)$,

$sttrans($`id(`exp_1`,` $\ldots$ `,`exp_r`,`var_1`,` $\ldots$ `,`var_s`);`$, tab, a)$

:= if $tab(id) = (proc, adr)$ then
$simpleexptrans(exp_1, tab)$ PUSH;
...
$simpleexptrans(exp_r, tab)$ PUSH;
$vartrans(var_1, tab)$ PUSH;
...
$vartrans(var_s, tab)$ PUSH;
CALL adr;

für alle $id \in \text{W}(\langle \text{ Ident } \rangle)$,
$exp_1, ..., exp_r \in \text{W}(\langle \text{SimpleExpression} \rangle)$,
$var_1, ..., var_s \in \{\&, \varepsilon\} \cdot \text{W}(\langle \text{Ident} \rangle)$

$sttrans($`scanf("%d", &`id`);`$, tab, a)$

:= if $tab(id) = (\text{var}, b, o)$ then READ(b, o);

$sttrans($`scanf("%d",`id`);`$, tab, a)$

:= if $tab(id) = (\text{var-ref}, o)$ then READI(o);

$sttrans($`printf("%d",`id`);`$, tab, a)$

:= if $tab(id) = (\text{var}, b, o)$ then WRITE(b, o);

$sttrans($`printf)"%d", *`id`);`$, tab, a)$

:= if $tab(id) = (\text{var-ref}, o)$ then WRITEI(o);

Die Übersetzungen von StatementSequences sowie If-, While- und CompoundStatements entsprechen denen der AM_0-Programme.

- **Übersetzung von Referenzparametern**

  $vartrans(\&id, tab)$

  := if $tab(id) = (\text{var}, b, o)$ then LOADA(b, o);

  $vartrans(id, tab)$

  := if $tab(id) = (\text{var-ref}, o)$ then LOAD(lokal$, o)$;

- **Übersetzung von BoolExpressions und SimpleExpressions**

  BoolExpressions werden analog zu C_0 übersetzt. SimpleExpressions hingegen müssen auf Grund des veränderten Befehlsvorrats anders übersetzt werden. Wie bereits bei den AM_0-Programmen soll dies nur informell dargelegt und durch Beispiele veranschaulicht werden.

  Konstanten werden weiterhin mittels des LIT-Befehls abgebildet. Dem Ladebefehl LOAD wird analog des Speicherbefehls STORE in Zuweisungen zusätzlich mitgeteilt, ob es sich bei der verwendeten Variable `v` um eine globale oder lokale handelt. Außerdem wird angegeben, um die wievielte – angegeben durch den Offset o – lokale Variable innerhalb der Funktion bzw. globale Variable es sich handelt. Der Eintrag in der Symboltabelle sieht folgendermaßen aus: $tab($`v`$) = (\text{var}, \text{global/lokal}, o)$. Für den Zugriff auf durch Referenzparameter übergebene Speicheradressen werden die Befehle LOADI und STOREI verwendet. In der Symboltabelle sind Referenzparameter wie folgt gekennzeichnet: $tab($`v`$) = (\text{var-ref}, o)$.

  Übersetzungsbeispiele für die sonstigen Ausdrücke

  Steht in der Symboltabelle tab für den Bezeichner `c` der Eintrag $(\text{const}, 4)$, so wird dieser einfache Ausdruck mittels `LIT 4;` übersetzt.

  Unter den Voraussetzungen wie sie im C_1-Programm `double1` auf Seite 218 gegeben sind, wird die Zuweisung `*y = *y + 2;` in der Funktion `double` dann folgendermaßen in AM_1-Code übersetzt:

```
LOADI(-2); LIT 2; ADD; STOREI(-2);
```

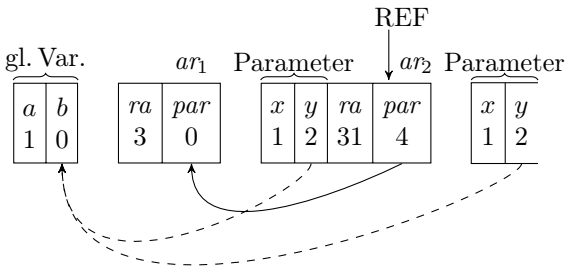
  Da es sich bei der Variablen `y` um einen Referenzparameter handelt und dieser an letzter Stelle im Funktionskopf steht, und folglich unmittelbar vor $(ra : par)$ im Laufzeitkeller liegt, insgesamt also zwei Positionen vor par.

Beispiel 17.1 (Fortsetzung)**.** Für das C_1-Programm `double1` von Seite 218 ergibt sich nachfolgender, bereits linearisierter AM_1-Code. Zusätzlich werden die verschiedenen Symboltabellen aufgeführt.

```
 1:  INIT 2;          tab_Programm = [double/(proc,1), a/(var,global,1), b/(var,global,2)]
 2:  CALL 24;
 3:  JMP 0;
 4:  INIT 0;          tab_double = tab_Programm[x/(var,lokal,−3), y/(var-ref,−2)]
 5:  LOAD(lokal,-3);
 6:  LIT 0;
 7:  GT;
 8:  JMC 21;
 9:  LOAD(lokal,-3);
10:  LIT 1;
11:  SUB;
12:  PUSH;
13:  LOAD(lokal,-2);
14:  PUSH;
15:  CALL 4;
16:  LOADI(-2);
17:  LIT 2;
18:  ADD;
19:  STOREI(-2);
20:  JMP 23;
21:  LIT 0;
22:  STOREI(-2);
23:  RET 2;
24:  INIT 0;          tab_main = tab_Programm
25:  READ(global,1);
26:  LOAD(global,1);
27:  PUSH;
28:  LOADA(global,2);
29:  PUSH;
30:  CALL 4;
31:  WRITE(global,2);
32:  RET 0;
```

Das Abarbeitungsprotokoll des AM_1-Codes für die Eingabe 1 ist in Abbildung 17.5 zu sehen. Damit ergibt sich: $\mathcal{P}[\![\texttt{double1}]\!](1) = proj_6^{(6)}(\mathcal{I}[\![\texttt{double1}]\!](1, \varepsilon, (1:2), 0, \varepsilon, 2)) = 2$.

In Abbildung 17.4 wird beispielhaft der Aufbau des Laufzeitkellers für den Sprung von Befehlszähleradresse 15 nach 4 gezeigt. □



Vervollständigen des neuen Aktivierungsblocks ar_3:

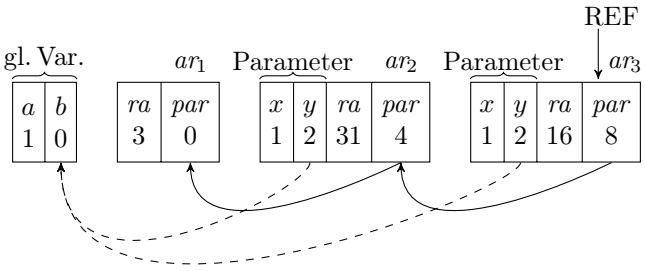


Abbildung 17.4: Aktivierungsblöcke für das C_1-Programm `double1`

(BZ,	DK, LK	, REF,	Inp,	$\underline{Out}$)
(1,	ε, ε	, 0,	1,	ε)
(2,	ε, (0 : 0)	, 0,	1,	ε)
(24,	ε, (0 : 0) : (3 : 0)	, 4,	1,	ε)
(25,	ε, (0 : 0) : (3 : 0)	, 4,	1,	ε)
(26,	ε, (1 : 0) : (3 : 0)	, 4,	ε,	ε)
(27,	1, (1 : 0) : (3 : 0)	, 4,	ε,	ε)
(28,	ε, (1 : 0) : (3 : 0) : (1	, 4,	ε,	ε)
(29,	2, (1 : 0) : (3 : 0) : (1	, 4,	ε,	ε)
(30,	ε, (1 : 0) : (3 : 0) : (1 : 2	, 4,	ε,	ε)
(4,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(5,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(6,	1, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(7, 0 : 1,	(1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(8,	1, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(9,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(10,	1, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(11, 1 : 1,	(1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(12,	0, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(13,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0	, 8,	ε,	ε)
(14,	2, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0	, 8,	ε,	ε)
(15,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2	, 8,	ε,	ε)
(4,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(5,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(6,	0, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(7, 0 : 0,	(1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(8,	0, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(21,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(22,	0, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(23,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4) : (0 : 2 : 16 : 8),	12,	ε,	ε)
(16,	ε, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(17,	0, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(18, 2 : 0,	(1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(19,	2, (1 : 0) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(20,	ε, (1 : 2) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(23,	ε, (1 : 2) : (3 : 0) : (1 : 2 : 31 : 4)	, 8,	ε,	ε)
(31,	ε, (1 : 2) : (3 : 0)	, 4,	ε,	ε)
(32,	ε, (1 : 2) : (3 : 0)	, 4,	ε,	2)
(3,	ε, (1 : 2)	, 0,	ε,	2))
(0,	ε, (1 : 2)	, 0,	ε,	2)

Abbildung 17.5: Abarbeitungsprotokoll des AM_1-Codes aus Beispiel 17.1 für die Eingabe 1

18 Verifikation von Programmeigenschaften

In diesem Kapitel wollen wir einen Kalkül vorstellen, mit dessen Hilfe Eigenschaften imperativer Programme bewiesen werden können. Wir beschränken uns auf C_0-Programme und verweisen auf K. Apt und E.-R. Olderog: „Verification of Sequential and Concurrent Programs“ für eine detaillierte und (viel) weitergehende Studie dieser Verifikationstechnik.

Das Ziel besteht darin, z. B. für unser Summationsprogramm `Summation` von Seite 209 die folgende „Verifikationsformel“ zu beweisen:

$$\{(n \geq 0)\}\ \texttt{Summation}\ \{(s = \sum_{j=1}^{n} j^2)\}$$

Diese Verifikationsformel bedeutet folgendes: Wenn vor der Ausführung von `Summation` die Zusicherung (assertion) $(n \geq 0)$ gilt und `Summation` terminiert, dann gilt nach Ausführung von `Summation` die Zusicherung $(s = \sum_{j=1}^{n} j^2)$, d. h. es wird der von uns gesuchte Wert ausgegeben. Der Hoare-Kalkül stellt nun einen Rahmen zur Verfügung, in dem man solche Verifikationsformeln beweisen kann.

Der Hoare-Kalkül wurde von C.A.R. Hoare in seiner Arbeit „An axiomatic basis for computer programming“ (Comm. Assoc. Comput. Sci. 12 (1969), 576–583) vorgestellt. Er arbeitet mit Aussagen (oder: *Zusicherungen*, *assertions*) und der Veränderung solcher Zusicherungen unter Einwirkung von Programmkonstrukten.

Zu jedem Zeitpunkt des Ablaufs eines Programms befindet „man“ sich an einer bestimmten Programmstelle, und der Speicher hat eine gewisse Belegung. Zu jedem Zeitpunkt lassen sich nun logische Aussagen über die Werte der Programmvariablen treffen (– die Werte werden ja durch die aktuelle Speicherbelegung beschrieben –).

Beispiel 18.1. Betrachten wir das C_0-Programm `Summation`, bei dem wir annehmen, dass nur Zahlen ≥ 0 eingelesen werden.

```
{
      scanf("%d",&n);
```
$\longrightarrow \{(n \geq 0)\}$
```
      i=1;
```
$\longrightarrow \{(n \geq 0) \wedge (i = 1)\}$
```
      s=0;
```
$\longrightarrow \{(n \geq 0) \wedge (i = 1) \wedge (s = 0)\}$

$\Longrightarrow \underbrace{\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}}_{SI}$

```
      while ( i<=n ) /* label 1 */
      {
         s = s + i*i;
         i = i + 1; /* label 2 */
      }
```
$\longrightarrow \{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge (i > n)\}$

$\Longrightarrow \{(s = \sum_{j=1}^{n} j^2)\}$

```
      printf("%d", s);
      return 0;
}
```

Wir protokollieren Schleifendurchläufe für die Belegung `n = 4` an den Stellen, die mit `label 1` und `label 2` markiert sind, um eine allgemeine Beziehung zwischen den Variablen `n`, `s` und `i` zu finden.

nach x Schleifendurchläufen	`label 1` (n, s, i)	$i \leq n$?	`label 2` (n, s, i)
$x = 0$	$(4, 0, 1)$	ja	$(4, 1, 2)$
$x = 1$	$(4, 1, 2)$	ja	$(4, 1 + 2^2, 3)$
$x = 2$	$(4, 1 + 2^2, 3)$	ja	$(4, 1 + 2^2 + 3^2, 4)$
$x = 3$	$(4, 1 + 2^2 + 3^2, 4)$	ja	$(4, 1 + 2^2 + 3^2 + 4^2, 5)$
$x = 4$	$(4, 1 + 2^2 + 3^2 + 4^2, 5)$	nein	—
allg.	$(n, \sum_{j=1}^{i-1} j^2, i)$ wobei $1 \leq i \leq n + 1$		$(n, \sum_{j=1}^{i-1} j^2, i)$ wobei $1 \leq i \leq n + 1$

Nach der `while`-Schleife gilt $\left(s = \sum_{j=1}^{i-1} j^2\right) \wedge (1 \leq i \leq n + 1) \wedge (i > n)$.

In der Abbildung stehen die assertions verschiedener Zeitpunkte des Ablaufs rechts neben dem Programm in geschweiften Klammern; die assertion *SI* ist die Schleifeninvariante. □

Der Hoare-Kalkül ist ein Kalkül für die richtige Verwendung und Transformation solcher Zusicherungen. Dabei geht man von der Vorstellung aus, dass jedes elementare Programmkonstrukt die Zusicherung verändert.

Die Beschreibung der Veränderungen der Zusicherungen erfolgt im Hoare-Kalkül durch *Verifikationsformeln* der Form:

$$\{P\}\mathbf{A}\{Q\}$$

Dabei gilt:

- A ist ein Programmstück,
- P und Q sind Zusicherungen (genauer: prädikatenlogische Ausdrücke), welche Eigenschaften über Variablen beschreiben.
- P heißt *Vorbedingung*, Q heißt *Nachbedingung*.

Die Formel $\{P\}\mathbf{A}\{Q\}$ ist korrekt, wenn folgendes gilt:

> wenn die Variablenwerte vor Ausführung des Programmstücks A die Zusicherung P erfüllen und wenn A terminiert, dann erfüllen die Programmvariablen nach Ausführung von A die Zusicherung Q.

Beachte: Durch Verifikationsformeln wird nur die *partielle Korrektheit* eines Programmstücks bestätigt. Für den Nachweis der *totalen Korrektheit* ist zusätzlich die *Termination* des Programmstücks zu untersuchen.

Je nach Kontext kann eine Aussage der Form $\{P\}\mathbf{A}\{Q\}$ für verschiedenes stehen:

- Ist A ein einzelnes Statement aus C_0, dann kann $\{P\}\mathbf{A}\{Q\}$ als Definition der Semantik von A verstanden werden. Die Gesamtheit der Verifikationsregeln heißt dann die *axiomatische Semantik* von C_0.
- Ist A ein gegebenes Programm, und ist $\{P\}\mathbf{A}\{Q\}$ bereits bewiesen, so kann dies als Schnittstellen-Beschreibung des Programms A aufgefasst werden.
- Ist A ein noch zu konstruierendes Programm, dann ist $\{P\}\mathbf{A}\{Q\}$ als Spezifikation von A aufzufassen.

Das Ziel des Hoare-Kalküls ist es, aus Verifikationsformeln für einzelne Programmteile Verifikationsformeln für zusammengesetzte Programmteile abzuleiten. Dabei geht man von den Verifikationsformeln für Anweisungen aus (Zuweisungsaxiom) bis schließlich eine Verifikationsformel für das gesamte Programm abgeleitet ist. Dieses induktive Aufbauen der Verifikationsformeln wird durch sogenannte *Verifikationsregeln* realisiert.

Hinweis: Ein- und Ausgaben in C_0 werden hier nicht einbezogen; mit anderen Worten: Wir nehmen an, dass das Programm am Anfang des Statementteils seines Blocks ein Lesestatement und am Ende ein Schreibstatement haben kann.

Wir geben nun die Verifikationsregeln der einzelnen Konstrukte von C_0 an:

Im folgenden seien P, Q, R beliebige prädikatenlogische Formeln und $\Pi \in W(\mathcal{E}_{C_0}, \langle \text{BoolExpression} \rangle)$, $\mathbf{A}$, $\mathbf{A_1}$ und $\mathbf{A_2}$ Elemente von $W(\mathcal{E}_{C_0}, \langle \text{Statement} \rangle)$, $\mathbf{S}$, $\mathbf{S_1}$ und $\mathbf{S_2}$ Elemente von $W(\mathcal{E}_{C_0}, \langle \text{StatementSequence} \rangle)$, und $\tau \in W(\mathcal{E}_{C_0}, \langle \text{SimpleExpression} \rangle)$ (siehe Seite 210).

Zuweisungsaxiom (Assignment):

$$\{P^x_\tau\}\ \mathtt{x} = \tau;\ \{P\}$$

Erläuterung: $P^{\mathtt{x}}_\tau$ entsteht aus P, indem jedes (freie) Vorkommen von $\mathtt{x}$ durch τ ersetzt wird. (Zur Erläuterung des Begriffes eines freien Vorkommens einer Variablen sei auf die Vorlesung über Logik und diskrete Strukturen verwiesen.)

Sequenzregel (StatementSequence):

$$\frac{\{P\}\mathbf{S_1}\{Q\} \qquad \{Q\}\mathbf{S_2}\{R\}}{\{P\}\mathbf{S_1}\mathbf{S_2}\{R\}}\ (\text{SR})$$

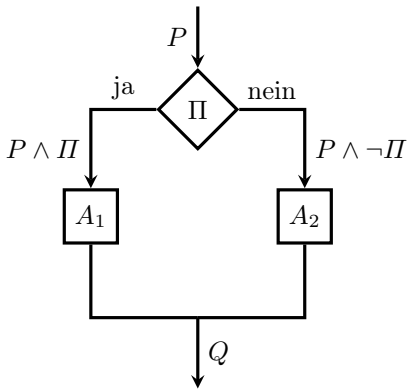
Compregel (CompoundStatement):

$$\frac{\{P\}\ \mathbf{S}\ \{Q\}}{\{P\}\ \{\mathbf{S}\}\ \{Q\}}\ (\text{CR})$$

Alternativregel (IfStatement):

$$\frac{\{P \wedge \Pi\}\mathbf{A_1}\{Q\} \qquad \{P \wedge \neg\Pi\}\mathbf{A_2}\{Q\}}{\{P\}\ \text{if}\ (\mathbf{\Pi})\ \mathbf{A_1}\ \text{else}\ \mathbf{A_2}\{Q\}}\ (\text{A2})$$

$$\frac{\{P \wedge \Pi\}\mathbf{A}\{Q\} \qquad (P \wedge \neg\Pi) \Rightarrow Q}{\{P\}\ \text{if}\ (\mathbf{\Pi})\ \mathbf{A}\{Q\}}\ (\text{A1})$$

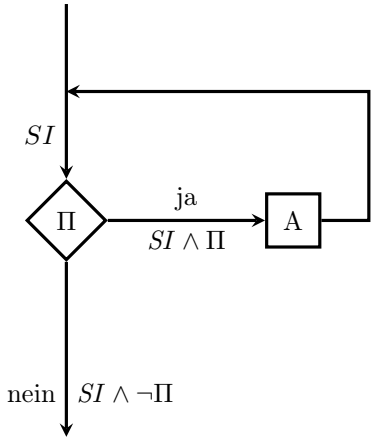


Iterationsregel (WhileStatement):

$$\frac{\{SI \wedge \Pi\}\mathbf{A}\{SI\}}{\{SI\}\ \text{while}\ (\mathbf{\Pi})\ \ \mathbf{A}\ \{SI \wedge \neg\Pi\}}\ (\text{IR})$$

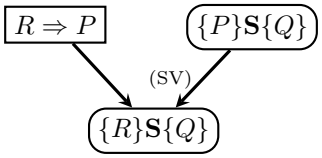
Erläuterung: Die Formel SI heißt *Schleifeninvariante*, da ihre Gültigkeit durch die Ausführung des Rumpfes nicht verändert wird. Die Aussage der Iterationsregel ist folgende:

Wenn *SI* beim Eintritt in die Schleife gilt und wenn *SI* nach einer von der Schleifenbedingung Π zugelassenen Ausführung des Rumpfes immer noch gilt, so gilt *SI* auch nach Ausführung der gesamten Schleife. Die Schleifenbedingung Π gilt dann nicht mehr, weil wir sonst die Schleife nicht hätten verlassen können.

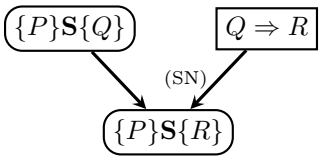


Konsequenzregeln:

Regel von der *stärkeren Vorbedingung*:



Regel von der *schwächeren Nachbedingung*



Satz 18.2. *[Hoare 1969] Gegeben sei ein Programm Prog und eine Spezifikation $S = \{P\}Prog\{Q\}$. Wenn sich durch sukzessive Anwendung der Verifikationsregeln die Formel*

$$\{P\}\ Prog\ \{Q\}$$

herleiten lässt, dann ist Prog bezüglich der Spezifikation S partiell korrekt.

Beispiel 18.3. Wir wollen an Hand des Programms zur Summation von Quadratzahlen (siehe Seite 209) die Verwendung des Hoare-Kalküls demonstrieren:

Bezeichnen wir den Anweisungsteil `i=1; s=0; while (i<=n) {... }` des Programmes mit *prog*, so wollen wir überprüfen, ob *prog* die folgende Spezifikation erfüllt:

$$\{(n \geq 0)\}\ prog\ \{(s = \sum_{j=1}^{n} j^2)\}$$

Anschaulich bedeutet dies:
Wir gehen davon aus, dass der Variablen n durch den Funktionsaufruf `scanf("%d",&n);` bereits ein

Wert zugewiesen wurde, der größer oder gleich 0 ist. Wir wollen dann beweisen, dass der Anweisungsteil *prog* tatsächlich die Summe der Quadratzahlen von 1 bis n berechnet und diesen Wert auf s ablegt.

Als wesentlicher Teil des Beweises ist zu zeigen, dass

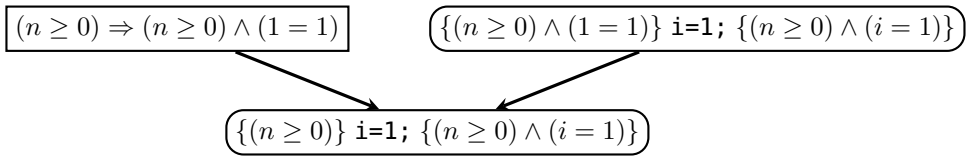
$$(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)$$

eine Schleifeninvariante ist. Dazu müssen wir zeigen, dass diese Zusicherung vor dem ersten Betreten der Schleife und nach jedem Durchlauf (und damit auch nach Verlassen) der Schleife gilt. Wir befassen uns also zunächst mit dem Teil `i=1; s=0;` von *prog*:

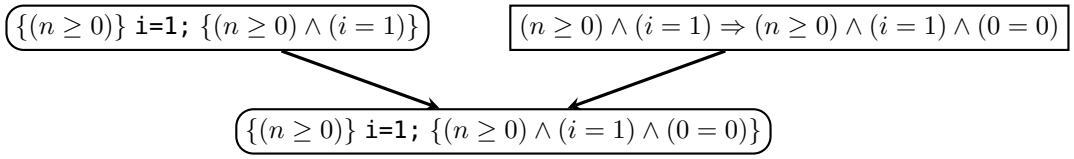
(1) Zuweisungsaxiom:

$$\{(n \geq 0) \wedge (1 = 1)\}\ \texttt{i=1;}\ \{(n \geq 0) \wedge (i = 1)\}$$

(2) stärkere Vorbedingung:



(3) schwächere Nachbedingung:

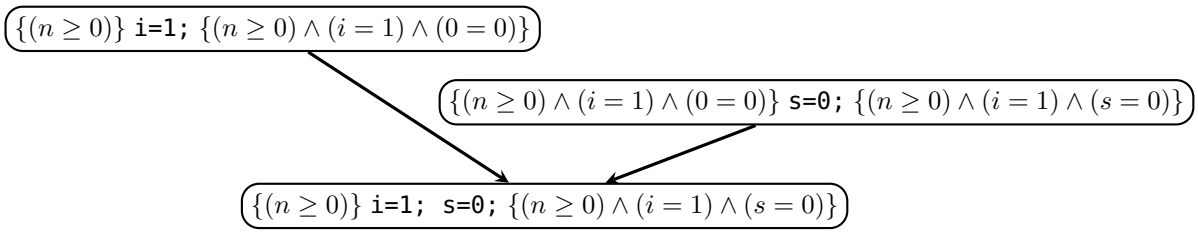


(4) Zuweisungsaxiom:

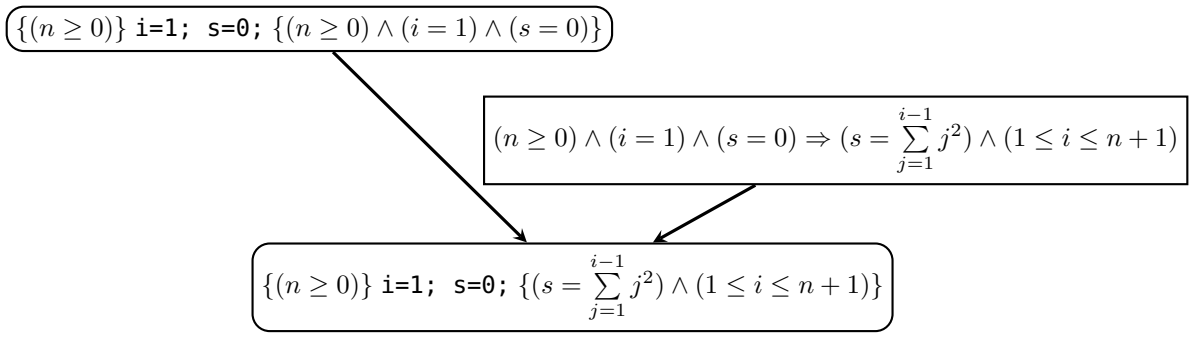
$$\{(n \geq 0) \wedge (i = 1) \wedge (0 = 0)\}\ \texttt{s=0;}\ \{(n \geq 0) \wedge (i = 1) \wedge (s = 0)\}$$

Die Resultate von (3) und (4) können nun mit Hilfe der Sequenzregel verknüpft werden:

(5) Sequenzregel:



(6) schwächere Nachbedingung:

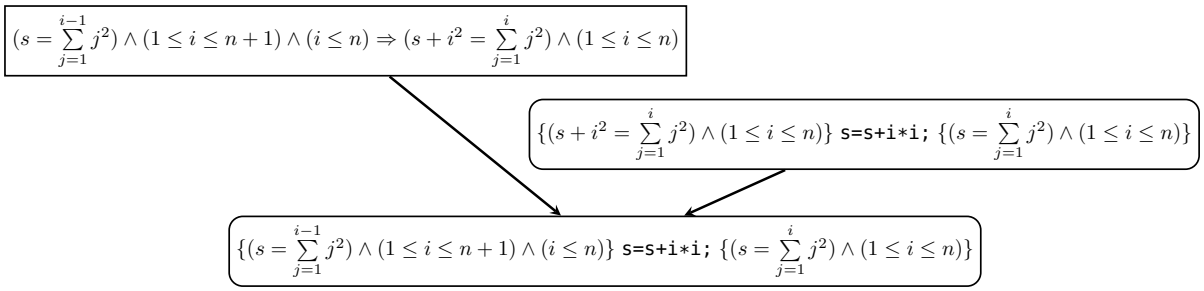


Das heißt, dass die Schleifeninvariante vor Betreten der Schleife gilt. Betrachten wir nun die Statements des Schleifenrumpfes:

(7) Zuweisungsaxiom:

$$\{(s + i^2 = \textstyle\sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)\}\ \texttt{s=s+i*i;}\ \{(s = \textstyle\sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)\}$$

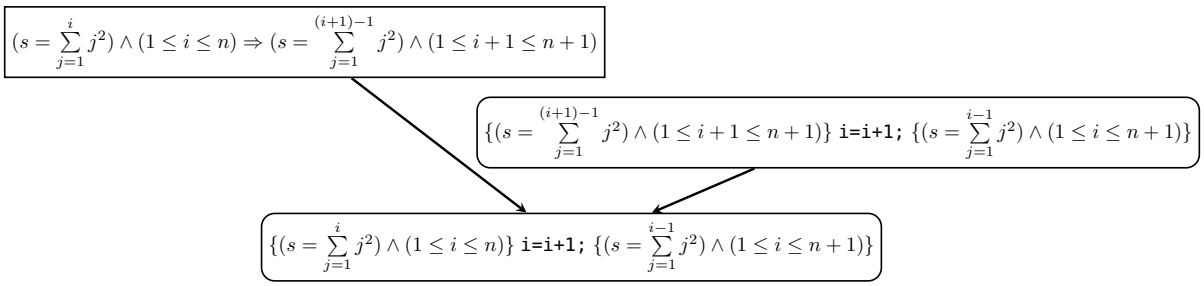
(8) stärkere Vorbedingung: In der folgenden Implikation wird die Voraussetzung $1 \leq i$ benutzt (zum Beispiel ist diese Implikation für $i = -1$ falsch).



(9) Zuweisungsaxiom:

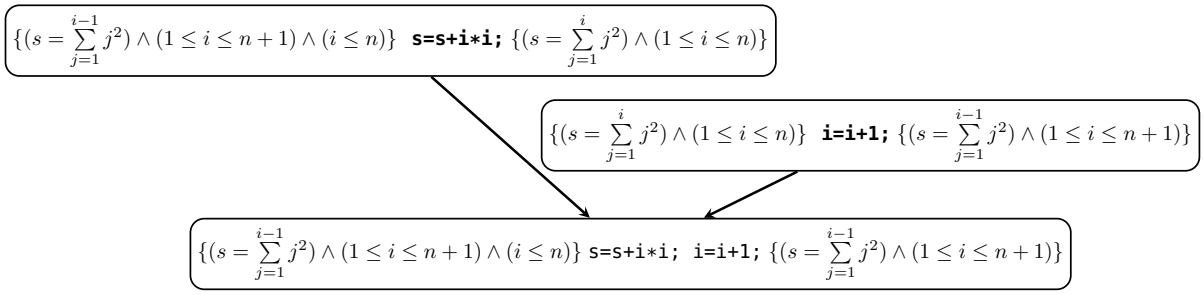
$$\{(s = \textstyle\sum_{j=1}^{(i+1)-1} j^2) \wedge (1 \leq i + 1 \leq n + 1)\}\ \texttt{i=i+1;}\ \{(s = \textstyle\sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n + 1)\}$$

(10) stärkere Vorbedingung:

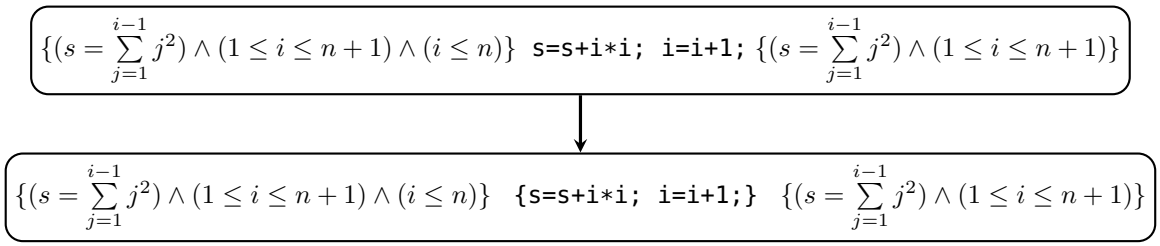


Wir verwenden wiederum die Sequenzregel, um die Resultate von (7) bis (10) zu einem Gesamtresultat für den Schleifenrumpf zusammenzusetzen:

(11) Sequenzregel:

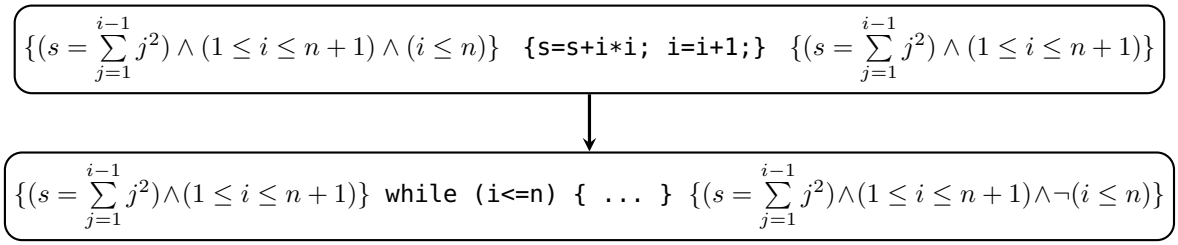


(12) Compregel:



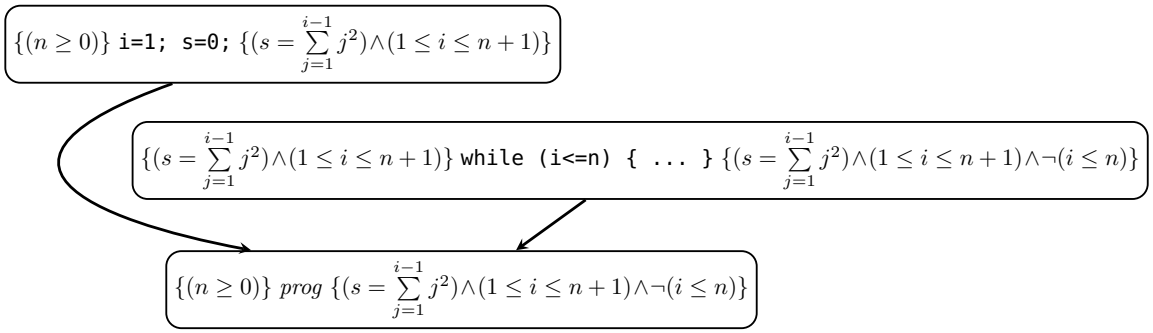
Wir haben nun gezeigt, dass die Wahl unserer Schleifeninvariante richtig war, und können nun die Iterationsregel anwenden:

(13) Iterationsregel:

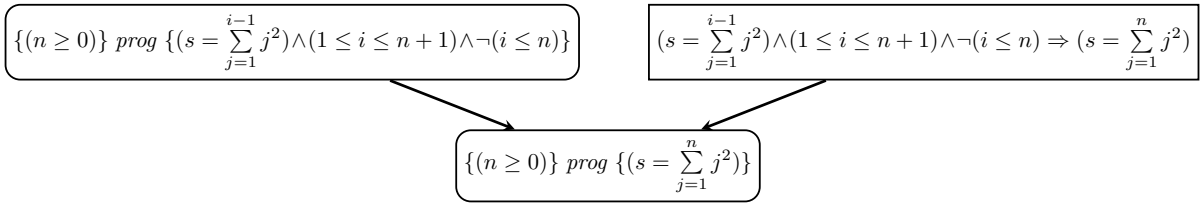


Die Resultate von (6) für die Befehle vor der Schleife und von (13) für die Schleife selbst werden mit Hilfe der Sequenzregel verknüpft:

(14) Sequenzregel:



(15) schwächere Nachbedingung:



Also erfüllt *prog* die Spezifikation, und der Beweis ist beendet. □

Der komplette Beweisbaum befindet sich auf Seite 237.

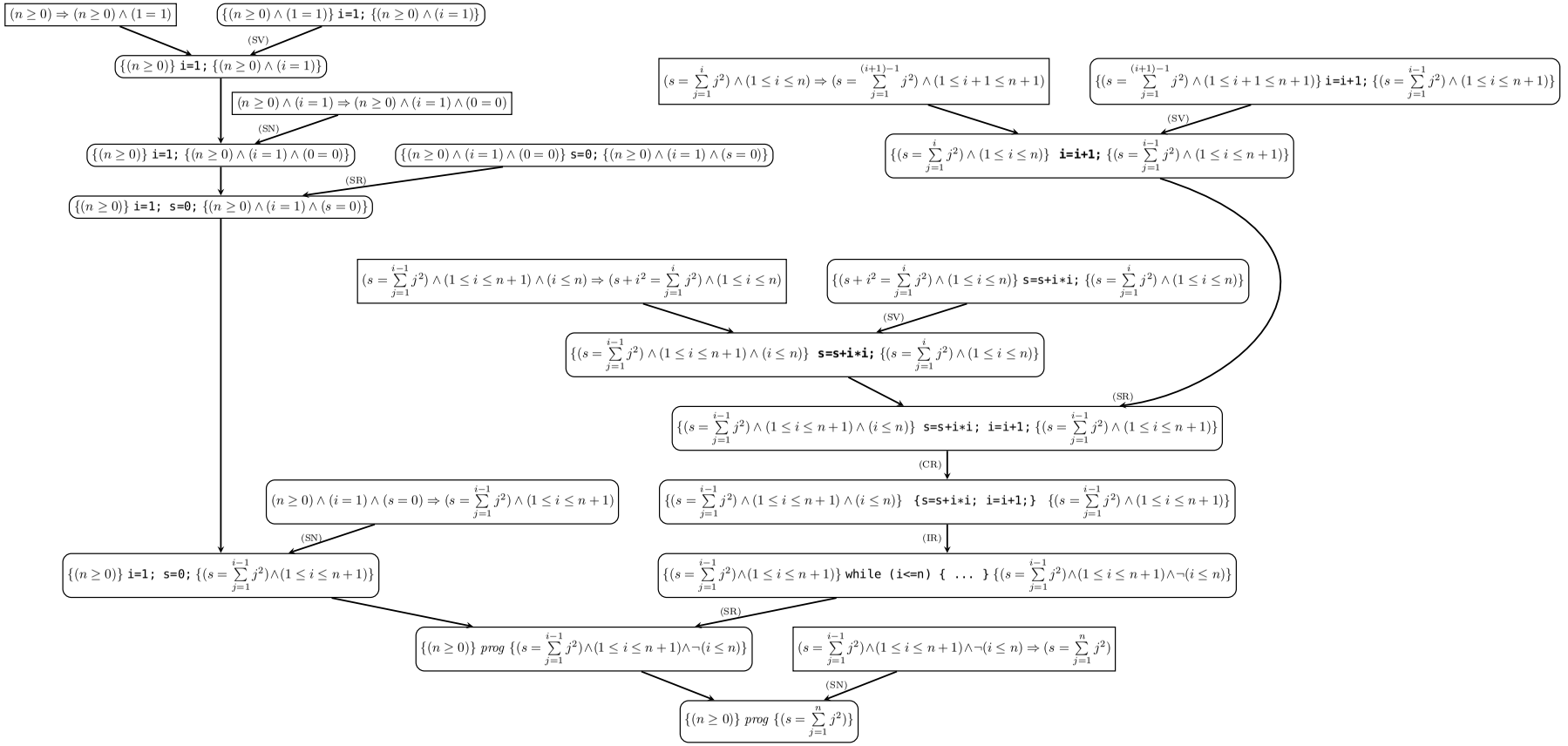
Beispiel 18.4. Wir wollen noch ein weiteres Beispiel angeben. Hier werden wir beweisen, dass die Spezifikation

$$\{(x = z) \wedge (x \geq 0)\}\ \underbrace{\texttt{y = 1;}\ \underbrace{\texttt{while (x > 0) \{ x = x - 1; y = 2 * y; \}}}_{while}}_{A} \{(y = 2^z)\}$$

gilt. Bei diesem Beweis werden wir eine sehr praktikable Methode demonstrieren, die von der Annahme ausgeht, dass diese Spezifikation richtig ist. Mit Hilfe der bekannten Verifikationsregeln wird nun ein Beweisbaum mit dem Ziel entwickelt, in den Blättern ausschließlich Zuweisungsaxiome und Implikationen zu erhalten. Lässt sich dies nicht realisieren, ist unsere Annahme falsch, das heißt die Spezifikation gilt nicht.

Der Beweisbaum (Abbildung 18.1) wird schrittweise von unten (der Wurzel) nach oben aufgebaut. Die jeweils verwendete Verifikationsregel wurde rechts dahinter vermerkt, wobei folgende Abkürzungen gelten: SV=stärkere Vorbedingung, SN=schwächere Nachbedingung, SR=Sequenzregel, CR=Compoundregel, IR=Iterationsregel.

□


$(n \geq 0) \Rightarrow (n \geq 0) \wedge (1 = 1)$
$\{(n \geq 0) \wedge (1 = 1)\}$ `i=1;` $\{(n \geq 0) \wedge (i = 1)\}$
(SV)
$\{(n \geq 0)\}$ `i=1;` $\{(n \geq 0) \wedge (i = 1)\}$
$(n \geq 0) \wedge (i = 1) \Rightarrow (n \geq 0) \wedge (i = 1) \wedge (0 = 0)$
(SN)
$\{(n \geq 0)\}$ `i=1;` $\{(n \geq 0) \wedge (i = 1) \wedge (0 = 0)\}$
$\{(n \geq 0) \wedge (i = 1) \wedge (0 = 0)\}$ `s=0;` $\{(n \geq 0) \wedge (i = 1) \wedge (s = 0)\}$
(SR)
$\{(n \geq 0)\}$ `i=1; s=0;` $\{(n \geq 0) \wedge (i = 1) \wedge (s = 0)\}$
$(s = \sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n) \Rightarrow (s = \sum_{j=1}^{(i+1)-1} j^2) \wedge (1 \leq i+1 \leq n+1)$
$\{(s = \sum_{j=1}^{(i+1)-1} j^2) \wedge (1 \leq i+1 \leq n+1)\}$ `i=i+1;` $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}$
(SV)
$\{(s = \sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)\}$ **`i=i+1;`** $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}$
$(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge (i \leq n) \Rightarrow (s + i^2 = \sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)$
$\{(s + i^2 = \sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)\}$ `s=s+i*i;` $\{(s = \sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)\}$
(SV)
$\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge (i \leq n)\}$ **`s=s+i*i;`** $\{(s = \sum_{j=1}^{i} j^2) \wedge (1 \leq i \leq n)\}$
(SR)
$\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge (i \leq n)\}$ `s=s+i*i; i=i+1;` $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}$
(CR)
$(n \geq 0) \wedge (i = 1) \wedge (s = 0) \Rightarrow (s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)$
$\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge (i \leq n)\}$ `{s=s+i*i; i=i+1;}` $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}$
(IR)
(SN)
$\{(n \geq 0)\}$ `i=1; s=0;` $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}$
$\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1)\}$ `while (i<=n) { ... }` $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge \neg(i \leq n)\}$
(SR)
$\{(n \geq 0)\}$ *prog* $\{(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge \neg(i \leq n)\}$
$(s = \sum_{j=1}^{i-1} j^2) \wedge (1 \leq i \leq n+1) \wedge \neg(i \leq n) \Rightarrow (s = \sum_{j=1}^{n} j^2)$
(SN)
$\{(n \geq 0)\}$ *prog* $\{(s = \sum_{j=1}^{n} j^2)\}$

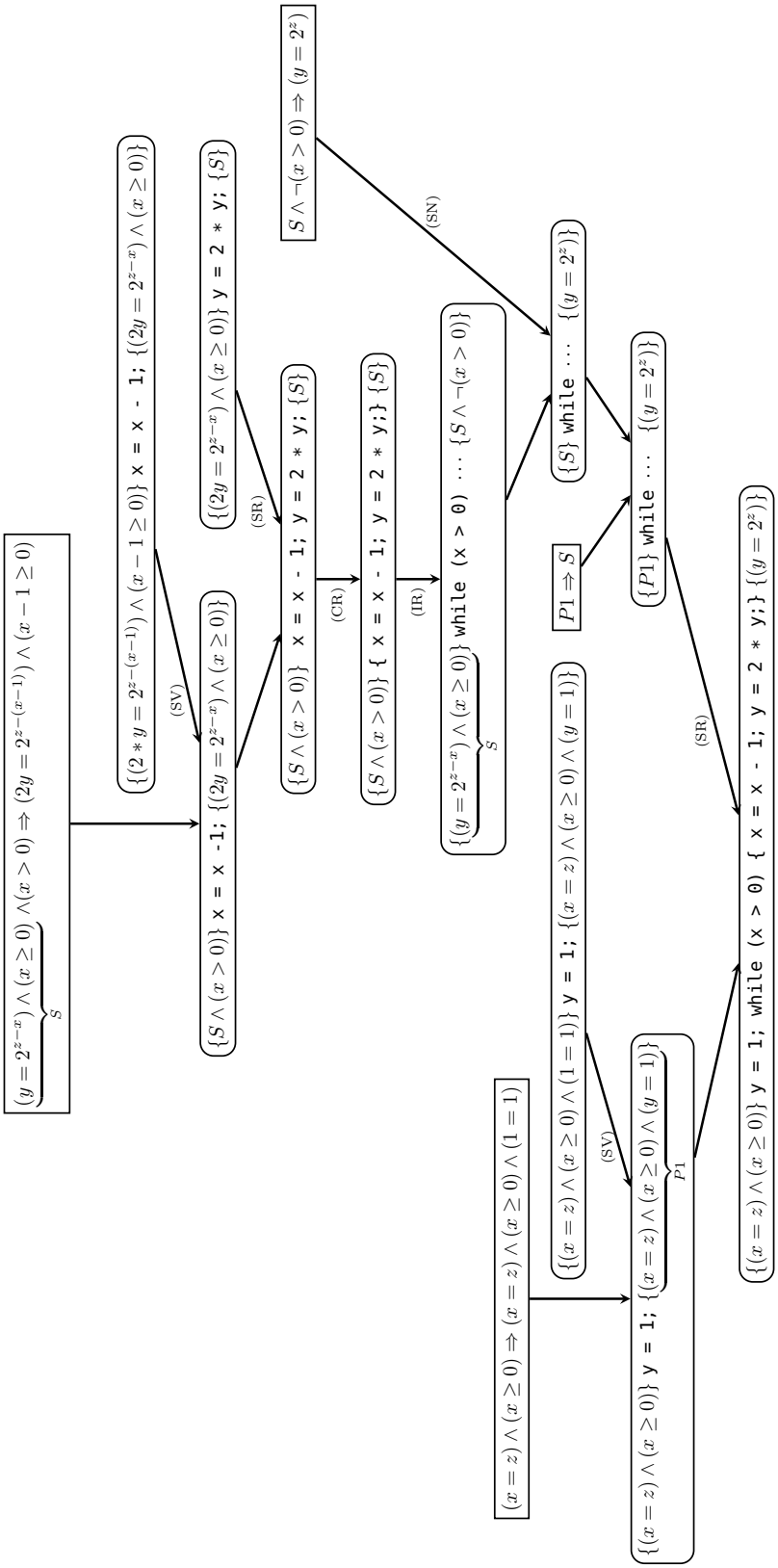


Abbildung 18.1

19 H_0 – Ein einfacher Kern von Haskell

In diesem Kapitel definieren wir die Teilsprache H_0 von Haskell, in der einerseits viele funktionale Programme formuliert werden können, die aber andererseits so „klein“ ist, dass ihre Implementierung auf einer abstrakten Maschine (d. h. die Beschreibung einer operationellen Semantik für H_0) völlig analog zur Implementierung von C_0 erfolgen kann. Darüber hinaus ist diese Implementierung deutlich einfacher und der für ein H_0-Programm erzeugte Code deutlich effizienter als bei den für (das gesamte) Haskell nötigen Implementierungstechniken, da insbesondere auf die Verwendung eines *Laufzeitkellers* (*runtime stack*) zur Verwaltung von rekursiven Funktionsaufrufen verzichtet werden kann (siehe z. B. [Küh03]). Des weiteren gibt es einen engen Zusammenhang zwischen H_0 und C_0, auf den wir im Abschnitt 19.3 eingehen wollen; dort werden wir einige Konzepte der Programmtransformation vorstellen.

In H_0 können ausschließlich sogenannte *tail rekursive* Funktionen definiert werden, bei denen die rechte Seite jeder Funktion entweder (i) keinen Funktionsaufruf enthält oder (ii) genau einen Funktionsaufruf an der „äußersten Position“ enthält (d. h. nicht eingeschachtelt z. B. innerhalb von arithmetischen Ausdrücken) oder (iii) eine Fallunterscheidung (mit `if...then...else`) ist, deren Zweige wiederum nach (i), (ii) oder (iii) aufgebaut sind. Wegen ihres engen Zusammenhanges zu den Schleifen in imperativen Programmen werden tail rekursive Programme auch *iterative* Programme genannt. Als einzigen Datentyp erlauben wir in H_0 die Menge der ganzen Zahlen (`Int`).

Im folgenden geben wir als Beispiele drei H_0-Programme an.

Beispiel 19.1. Das folgende Programm berechnet die Summe zweier Zahlen `x1` und `x2`:

```
module Main where

sum :: Int -> Int -> Int
sum x1 x2 = if x1>0 then sum (x1-1) (x2+1)
                    else x2

main = do x1 <- readLn
          x2 <- readLn
          print (sum x1 x2)
```
□

Beispiel 19.2. Das folgende Programm berechnet die Summe der ersten `x1` Quadratzahlen:

```
module Main where

sumsquare :: Int -> Int -> Int
sumsquare x1 x2 =  if x1==0 then x2
                            else sumsquare (x1-1) (x2+(x1*x1))

main = do x1 <- readLn
          print (sumsquare x1 0)
```
□

Beispiel 19.3. Das folgende Programm berechnet die `x1`-te Fibonaccizahl:

```
module Main where

fib :: Int -> Int -> Int -> Int
fib x1 x2 x3 =  if x1==0 then x2
                         else fib (x1-1) x3 (x2+x3)

main = do x1 <- readLn
          print (fib x1 1 1)
```
□

Für H_0-Programme sollen in diesem Kapitel

1. die Syntax durch eine EBNF-Definition (dadurch Festlegung einer Obermenge $\hat{H}_0$ der syntaktisch zulässigen Programme) und durch kontextsensitive Bedingungen und
2. die operationelle Semantik

formal definiert werden.

19.1 Syntax von H_0

Bevor wir die EBNF-Definition für $\hat{H}_0$ angeben, sollen zunächst die *Terminalsymbole* von $\hat{H}_0$ festgelegt werden:

Bezeichner (Identifier):

Wir unterscheiden die disjunkten Mengen der Variablenbezeichner und Funktionsbezeichner, die als syntaktische Kategorien der syntaktischen Variablen ⟨Varid⟩ bzw. ⟨Funid⟩ definiert sind:

⟨Varid⟩ ::= `x` (`1` | ... | `9`) { `0` | ... | `9` } .

⟨Funid⟩ ::= (`a` | ... | `w` | `y` | `z`) { `a` | ... | `z` | `0` | ... | `9` | `_` } .

Zahlen:

Die Menge der in $\hat{H}_0$ erlaubten Zahlen ist die syntaktische Kategorie von ⟨Number⟩:

⟨Number⟩ ::= 0 | [`-`] (`1` | ... | `9`) { `0` | ... | `9` } .

Schlüsselwörter:

In $\hat{H}_0$ gibt es die folgenden Schlüsselwörter:

```
module   Main   where   main   =     do   <-   readLn
print    (      )       ::     Int   ->   if   then     else
```

Operatoren:

Es stehen die arithmetischen Operatoren

`+`	(Addition),
`-`	(Subtraktion),
`*`	(Multiplikation),
`` `div` ``	(ganzzahlige Division) und
`` `mod` ``	(Rest bei ganzzahliger Division)

und die Vergleichsoperatoren

`==`	(gleich),
`/=`	(ungleich),
`<`	(kleiner als),
`>`	(größer als),
`<=`	(kleiner oder gleich) und
`>=`	(größer oder gleich)

zur Verfügung.

19.1.1 Kontextfreie Syntax von H_0

Im folgenden geben wir die EBNF-Regeln der EBNF-Definition $\mathcal{E}_{H_0}$ von $\hat{H}_0$ an. Wir verzichten auf die explizite Auflistung der syntaktischen Variablen. Das Startsymbol der EBNF-Definition ist die syntaktische Variable ⟨Prog⟩:

⟨Prog⟩ ::= `module Main where`
⟨Fun⟩ $\hat{\{}$ ⟨Fun⟩ $\hat{\}}$
`main = do` $\hat{\{}$ ⟨Varid⟩ `<- readLn` $\hat{\}}$
`print (` ⟨Funid⟩ $\hat{\{}$ ⟨Exp⟩ $\hat{\}}$ `)` .

⟨Fun⟩ ::= ⟨Funid⟩ `::` $\hat{\{}$ `Int ->` $\hat{\}}$ `Int`
⟨Funid⟩ $\hat{\{}$ ⟨Varid⟩ $\hat{\}}$ `=` ⟨Rhs⟩ .

⟨Rhs⟩ ::= `if` ⟨Bexp⟩ `then` ⟨Rhs⟩ `else` ⟨Rhs⟩ $\hat{|}$
⟨Funid⟩ $\hat{\{}$ ⟨Exp⟩ $\hat{\}}$ $\hat{|}$
⟨Exp⟩ .

⟨Bexp⟩ ::= ⟨Exp⟩ $\hat{(}$ `==` $\hat{|}$ `/=` $\hat{|}$ `>` $\hat{|}$ `>=` $\hat{|}$ `<` $\hat{|}$ `<=` $\hat{)}$ ⟨Exp⟩ .

⟨Exp⟩ ::= `(` ⟨Exp⟩ $\hat{(}$ `+` $\hat{|}$ `-` $\hat{|}$ `*` $\hat{|}$ `'div'` $\hat{|}$ `'mod'` $\hat{)}$ ⟨Exp⟩ `)` $\hat{|}$
⟨Varid⟩ $\hat{|}$
⟨Number⟩ .

19.1.2 Kontextsensitive Bedingungen für H_0

Für jede syntaktische Variable v verwenden wir für deren syntaktische Kategorie $W(\mathcal{E}_{H_0}, v)$ im folgenden die kürzere Notation $W(v)$.

Sei $p \in W(\langle\text{Prog}\rangle)$. Wenn p die folgenden kontextsensitiven Nebenbedingungen erfüllt, dann ist p ein H_0-Programm.

- Für jeden Funktionsbezeichner, der in p vorkommt, gibt es genau eine Funktionsdefinition in p.
- Wenn `f ::` ... `g` ... `=` ... $\in W(\langle\text{Fun}\rangle)$ in p vorkommt, dann gilt `f` = `g`.
- Wenn der Typ einer Funktion `f` $\in W(\langle\text{Funid}\rangle)$ in p mit `f ::` $\underbrace{\texttt{Int -> ... -> Int}}_{(n+1)\text{-mal}}$ und $n \geq 0$ festgelegt wird, dann hat
  - die zugehörige Definition von `f` in p die Form `f x1 ... xn = r`, wobei in `r` außer `x1`, ... , `xn` keine weiteren Variablenbezeichner vorkommen dürfen, und
  - jeder Aufruf der Funktion `f` in p die Form `f` $e_1 \ldots e_n$ mit $e_i \in W(\langle\text{Exp}\rangle)$.
- Die Definition der Funktion `main` hat die Gestalt
  `main = do x1 <- readLn ... xn <- readLn print r`,
  wobei `n` ≥ 0 und in `r` außer `x1`, ... , `xn` keine weiteren Variablenbezeichner vorkommen dürfen.

19.2 Operationelle Semantik von H_0

19.2.1 Abstrakte Maschine, Befehle und Programme

Wir verwenden wiederum die abstrakte Maschine AM_0 aus Abschnitt 17.1.2.

19.2.2 Übersetzung von H_0-Programmen in AM_0-Programme

Zunächst werden wir auch hier die Übersetzung in AM_0-Programme mit baumstrukturierten Adressen vornehmen. Diese Programme werden anschließend in die eigentlichen AM_0-Programme übersetzt. Eine baumstrukturierte Adresse hat dabei die folgende Gestalt:

$f.i_1.i_2.i_3. \ldots .i_n$ mit $f \in W(\langle\text{Funid}\rangle)$, $n \in \mathbb{N}$ und $i_j \in \mathbb{N}$ für alle $1 \leq j \leq n$.

Ferner lassen wir 0 als baumstrukturierte Adresse zu, die bei der späteren Linearisierung der Adressen unverändert bleibt. Die Menge aller AM_0-Programme mit baumstrukturierten Adressen wird durch $bProg'_0$ bezeichnet.

Für die Übersetzung wird hier keine Symboltabelle benötigt, da die Variablennamen für die Argumente von Funktionen standardmäßig als `x1`, `x2`, `x3`, ... festgelegt wurden und wir dafür die Speicherplätze $1, 2, 3, \ldots$, respektive, vorsehen.

- **Übersetzung von arithmetischen Ausdrücken:**

$$
\begin{aligned}
&\mathit{exptrans}\colon W(\langle\text{Exp}\rangle) \dashrightarrow \mathit{bProg}'_0 \\
&\qquad \mathit{exptrans}(z) := \text{LIT } z; && \text{für alle } z \in W(\langle\text{Number}\rangle), \\
&\qquad \mathit{exptrans}(\text{x}i) := \text{LOAD } i; && \text{für alle x}i \in W(\langle\text{Varid}\rangle), \\
&\mathit{exptrans}(e_1\ \mathit{op}\ e_2) := \mathit{exptrans}(e_1) \\
&\qquad\qquad\qquad\qquad \mathit{exptrans}(e_2) \\
&\qquad\qquad\qquad\qquad \text{OP}; && \text{für alle } e_1, e_2 \in W(\langle\text{Exp}\rangle) \text{ und } \mathit{op} \in \{\texttt{+}, \texttt{-}, *, \texttt{`div`}, \texttt{`mod`}\} \\
&\qquad\qquad\qquad\qquad \text{wobei OP der dem arithmetischen Operator } \mathit{op} \text{ zugeordnete } AM_0\text{-Befehl ist.}
\end{aligned}
$$

- **Übersetzung von Booleschen Ausdrücken:**

$$
\begin{aligned}
&\mathit{bexptrans}\colon W(\langle\text{Bexp}\rangle) \dashrightarrow \mathit{bProg}'_0 \\
&\mathit{bexptrans}(e_1\ \mathit{rel}\ e_2) := \mathit{exptrans}(e_1) \\
&\qquad\qquad\qquad\qquad \mathit{exptrans}(e_2) \\
&\qquad\qquad\qquad\qquad \text{REL}; && \text{für alle } e_1, e_2 \in W(\langle\text{Exp}\rangle) \text{ und } \mathit{rel} \in \{\texttt{==}, \texttt{/=}, \texttt{>}, \texttt{>=}, \texttt{<}, \texttt{<=}\} \\
&\qquad\qquad\qquad\qquad \text{wobei REL der dem Vergleichsoperator } \mathit{rel} \text{ zugeordnete } AM_0\text{-Befehl ist.}
\end{aligned}
$$

- **Übersetzung von rechten Seiten von Funktionsdefinitionen:** Hier muss ein zusätzliches Argument in der Übersetzungsfunktion mitgeführt werden, um für einen AM_0-Befehl einen konfliktfreien Adressbereich im zu konstruierenden Programm zur Verfügung zu haben:

$$
\begin{aligned}
&\mathit{rhstrans}\colon W(\langle\text{Rhs}\rangle) \times (W(\langle\text{Funid}\rangle) \cdot \mathbb{N}^*) \dashrightarrow \mathit{bProg}'_0 \\
&\mathit{rhstrans}(e, a) := \mathit{exptrans}(e) \\
&\qquad\qquad\qquad\qquad \text{STORE } 1; \\
&\qquad\qquad\qquad\qquad \text{WRITE } 1; \\
&\qquad\qquad\qquad\qquad \text{JMP } 0; && \text{für alle } e \in W(\langle\text{Exp}\rangle) \text{ und } a \in (W(\langle\text{Funid}\rangle) \cdot \mathbb{N}^*) \\
&\mathit{rhstrans}(f\ e_1 \ldots e_n, a) := \mathit{exptrans}(e_1)\ \ldots\ \mathit{exptrans}(e_n) \\
&\qquad\qquad\qquad\qquad \text{STORE } n; \ldots \text{STORE } 1; \\
&\qquad\qquad\qquad\qquad \text{JMP } f; \\
&\qquad\qquad \text{für alle } f \in W(\langle\text{Funid}\rangle),\ e_1, \ldots, e_n \in W(\langle\text{Exp}\rangle) \text{ und } a \in (W(\langle\text{Funid}\rangle) \cdot \mathbb{N}^*) \\
&\mathit{rhstrans}(\texttt{if}\ \mathit{be}\ \texttt{then}\ r_1\ \texttt{else}\ r_2, a) := \quad \mathit{bexptrans}(\mathit{be}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \text{JMC } a.3; \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{rhstrans}(r_1, a.1) \\
&\qquad\qquad\qquad\qquad a.3\colon \mathit{rhstrans}(r_2, a.2) \\
&\qquad\qquad \text{für alle } \mathit{be} \in W(\langle\text{Bexp}\rangle),\ r_1, r_2 \in W(\langle\text{Rhs}\rangle) \text{ und } a \in (W(\langle\text{Funid}\rangle) \cdot \mathbb{N}^*)
\end{aligned}
$$

  Im Fall von $\mathit{rhstrans}(e, a)$ soll das Ergebnis der Auswertung von e vom Datenkeller auf das Ausgabeband übertragen werden. Da es hierfür in der AM_0 keinen speziellen Befehl gibt, wird der Transport über den Umweg des ersten Hauptspeicherplatzes (dessen bisheriger Inhalt nicht mehr relevant ist) vorgenommen. Anschließend wird der Befehlszähler auf 0 gesetzt und damit die Programmausführung beendet.

  Man beachte ferner, dass im Fall von $\mathit{rhstrans}(\texttt{if ...})$ ein weiterer Sprungbefehl am Ende von $\mathit{rhstrans}(r_1, a.1)$ (zum „Überspringen“ von $\mathit{rhstrans}(r_2, a.2)$) nicht notwendig ist, da die Übersetzung $\mathit{rhstrans}(r_2, a.2)$ (gemäß der zwei Fälle $\mathit{rhstrans}(e, a)$ und $\mathit{rhstrans}(f\ e_1 \ldots e_n, a)$) mit dem Befehl JMP endet und damit ohnehin ein unbedingter Sprung durchgeführt wird.

- **Übersetzung von Funktionsdefinitionen:**

$$\begin{array}{l} \mathit{funtrans}\colon W(\langle\mathrm{Fun}\rangle) \text{ -}\!\!\rightarrow bProg_0' \\ \mathit{funtrans}\begin{pmatrix} f\texttt{ :: Int -> ... -> Int} \\ f\texttt{ x1 ... xn = } r \end{pmatrix} := f\texttt{ : } \mathit{rhstrans}(r, f) \end{array}$$

für alle $f \in W(\langle\mathrm{Funid}\rangle)$ und $r \in W(\langle\mathrm{Rhs}\rangle)$

- **Übersetzung von H_0-Programmen:**

$$\begin{array}{l} \mathit{trans}\colon W(\langle\mathrm{Prog}\rangle) \text{ -}\!\!\rightarrow bProg_0' \\ \mathit{trans}\begin{pmatrix} \texttt{module Main where} \\ \mathit{fun}_1 \ldots \mathit{fun}_m \\ \texttt{main = do x1 <- readLn} \\ \qquad\qquad \ldots \\ \qquad\qquad \texttt{xk <- readLn} \\ \qquad\qquad \texttt{print (}f\ e_1\ \ldots\ e_n\texttt{)} \end{pmatrix} := \begin{array}{l} \mathrm{READ}\ 1; \ldots \mathrm{READ}\ k; \\ \mathit{exptrans}(e_1) \ldots \mathit{exptrans}(e_n) \\ \mathrm{STORE}\ n; \ldots \mathrm{STORE}\ 1; \\ \mathrm{JMP}\ f; \\ \mathit{funtrans}(\mathit{fun}_1) \ldots \mathit{funtrans}(\mathit{fun}_m) \end{array} \end{array}$$

für alle $\mathit{fun}_1, \ldots, \mathit{fun}_m \in W(\langle\mathrm{Fun}\rangle)$, $f \in W(\langle\mathrm{Funid}\rangle)$ und $e_1, \ldots, e_n \in W(\langle\mathrm{Exp}\rangle)$

Beispiel 19.2 (Fortsetzung)**.** Das H_0-Programm zur Summation von Quadratzahlen wird zunächst in ein AM_0-Programm mit baumstrukturierten Adressen übersetzt:

$$\begin{array}{l} \mathit{trans}(\texttt{module Main where ... print (sumsquare x1 0)} \\ \\ = \mathrm{READ}\ 1;\ \mathit{exptrans}(\texttt{x1})\ \mathit{exptrans}(\texttt{0}) \\ \quad \mathrm{STORE}\ 2;\ \mathrm{STORE}\ 1;\ \mathrm{JMP}\ \mathrm{sumsquare}; \\ \quad \mathit{funtrans}\begin{pmatrix} \texttt{sumsquare :: Int -> Int -> Int} \\ \texttt{sumsquare x1 x2 = if ... then ... else ...} \end{pmatrix} \\ \\ = \qquad \mathrm{READ}\ 1;\ \mathrm{LOAD}\ 1;\ \mathrm{LIT}\ 0; \\ \qquad \mathrm{STORE}\ 2;\ \mathrm{STORE}\ 1;\ \mathrm{JMP}\ \mathrm{sumsquare}; \\ \quad \mathrm{sumsquare}\colon \mathit{rhstrans}\begin{pmatrix} \texttt{if x1 == 0} \\ \texttt{ then x2} \\ \texttt{ else sumsquare (x1-1) (x2+(x1*x1))} \end{pmatrix}, \mathrm{sumsquare} \\ \\ = \qquad \mathrm{READ}\ 1;\ \mathrm{LOAD}\ 1;\ \mathrm{LIT}\ 0; \\ \qquad \mathrm{STORE}\ 2;\ \mathrm{STORE}\ 1;\ \mathrm{JMP}\ \mathrm{sumsquare}; \\ \quad \mathrm{sumsquare}\colon \mathit{bexptrans}(\texttt{x1 == 0})\ \mathrm{JMC}\ \mathrm{sumsquare}.3; \\ \qquad \mathit{rhstrans}(\texttt{x2}, \mathrm{sumsquare}.1) \\ \quad \mathrm{sumsquare}.3\colon \mathit{rhstrans}(\texttt{sumsquare (x1-1) (x2+(x1*x1))}, \mathrm{sumsquare}.2) \\ \\ = \qquad \mathrm{READ}\ 1;\ \mathrm{LOAD}\ 1;\ \mathrm{LIT}\ 0; \\ \qquad \mathrm{STORE}\ 2;\ \mathrm{STORE}\ 1;\ \mathrm{JMP}\ \mathrm{sumsquare}; \\ \quad \mathrm{sumsquare}\colon \mathit{exptrans}(\texttt{x1})\ \mathit{exptrans}(\texttt{0})\ \mathrm{EQ};\ \mathrm{JMC}\ \mathrm{sumsquare}.3; \\ \qquad \mathit{exptrans}(\texttt{x2})\ \mathrm{STORE}\ 1;\ \mathrm{WRITE}\ 1;\ \mathrm{JMP}\ 0; \\ \quad \mathrm{sumsquare}.3\colon \mathit{exptrans}(\texttt{(x1-1)}) \\ \qquad \mathit{exptrans}(\texttt{(x2+(x1*x1))}) \\ \qquad \mathrm{STORE}\ 2;\ \mathrm{STORE}\ 1;\ \mathrm{JMP}\ \mathrm{sumsquare}; \end{array}$$

$$
\begin{array}{rl}
= & \text{READ 1; LOAD 1; LIT 0;} \\
 & \text{STORE 2; STORE 1; JMP sumsquare;} \\
\text{sumsquare:} & \text{LOAD 1; LIT 0; EQ; JMC sumsquare.3;} \\
 & \text{LOAD 2; STORE 1; WRITE 1; JMP 0;} \\
\text{sumsquare.3:} & \textit{exptrans}(\texttt{x1})\ \textit{exptrans}(\texttt{1})\ \text{SUB;} \\
 & \textit{exptrans}(\texttt{x2})\ \textit{exptrans}(\texttt{(x1*x1)})\ \text{ADD;} \\
 & \text{STORE 2; STORE 1; JMP sumsquare;} \\
= & \text{READ 1; LOAD 1; LIT 0;} \\
 & \text{STORE 2; STORE 1; JMP sumsquare;} \\
\text{sumsquare:} & \text{LOAD 1; LIT 0; EQ; JMC sumsquare.3;} \\
 & \text{LOAD 2; STORE 1; WRITE 1; JMP 0;} \\
\text{sumsquare.3:} & \text{LOAD 1; LIT 1; SUB;} \\
 & \text{LOAD 2; LOAD 1; LOAD 1; MUL; ADD;} \\
 & \text{STORE 2; STORE 1; JMP sumsquare;}
\end{array}
$$

Durch Linearisierung der Adressen erhält man das in Abbildung 19.1 gezeigte Programm p. Bei der Ausführung von p auf der AM_0 mit Eingabe 1 entsteht das in Abbildung 19.2 gezeigte Ablaufprotokoll. Dann gilt: $\mathcal{P}[\![\mathtt{p}]\!](1) = proj_5^{(5)}(\mathcal{I}[\![\mathtt{p}]\!](1, \varepsilon, [\,], 1, \varepsilon)) = proj_5^{(5)}(0, \varepsilon, [1/0, 2/1], \varepsilon, 1) = 1$. □

```
 1:  READ 1;
 2:  LOAD 1;
 3:  LIT 0;
 4:  STORE 2;
 5:  STORE 1;
 6:  JMP 7;
 7:  LOAD 1;
 8:  LIT 0;
 9:  EQ;
10:  JMC 15;
11:  LOAD 2;
12:  STORE 1;
13:  WRITE 1;
14:  JMP 0;
15:  LOAD 1;
16:  LIT 1;
17:  SUB;
18:  LOAD 2;
19:  LOAD 1;
20:  LOAD 1;
21:  MUL;
22:  ADD;
23:  STORE 2;
24:  STORE 1;
25:  JMP 7;
```

Abbildung 19.1: Programm p

	BZ	*DK*	*HS*	*Inp*	*Out*
(	1	ε	$h_\emptyset = [\,]$	1	ε)
(	2	ε	$[1/1]$	ε	ε)
(	3	1	$[1/1]$	ε	ε)
(	4	$0:1$	$[1/1]$	ε	ε)
(	5	1	$[1/1, 2/0]$	ε	ε)
(	6	ε	$[1/1, 2/0]$	ε	ε)
(	7	ε	$[1/1, 2/0]$	ε	ε)
(	8	1	$[1/1, 2/0]$	ε	ε)
(	9	$0:1$	$[1/1, 2/0]$	ε	ε)
(	10	0	$[1/1, 2/0]$	ε	ε)
(	15	ε	$[1/1, 2/0]$	ε	ε)
(	16	1	$[1/1, 2/0]$	ε	ε)
(	17	$1:1$	$[1/1, 2/0]$	ε	ε)
(	18	0	$[1/1, 2/0]$	ε	ε)
(	19	$0:0$	$[1/1, 2/0]$	ε	ε)
(	20	$1:0:0$	$[1/1, 2/0]$	ε	ε)
(	21	$1:1:0:0$	$[1/1, 2/0]$	ε	ε)
(	22	$1:0:0$	$[1/1, 2/0]$	ε	ε)
(	23	$1:0$	$[1/1, 2/0]$	ε	ε)
(	24	0	$[1/1, 2/1]$	ε	ε)
(	25	ε	$[1/0, 2/1]$	ε	ε)
(	7	ε	$[1/0, 2/1]$	ε	ε)
(	8	0	$[1/0, 2/1]$	ε	ε)
(	9	$0:0$	$[1/0, 2/1]$	ε	ε)
(	10	1	$[1/0, 2/1]$	ε	ε)
(	11	ε	$[1/0, 2/1]$	ε	ε)
(	12	1	$[1/0, 2/1]$	ε	ε)
(	13	ε	$[1/1, 2/1]$	ε	ε)
(	14	ε	$[1/1, 2/1]$	ε	1)
(	0	ε	$[1/1, 2/1]$	ε	1)

Abbildung 19.2: Ablaufprotokoll von p für die Eingabe 1

19.3 Zusammenhang der Sprachen H_0 und C_0

Imperative Programmiersprachen (wie C_0) und (tail-rekursive) funktionale Programmiersprachen (wie H_0) erweisen sich in dem Sinne als gleich beschreibungsstark, dass zu jedem imperativen Programm ein semantisch äquivalentes tail-rekursives Programm angegeben werden kann und umgekehrt. Darüber hinaus kann man Algorithmen formulieren, die die jeweilige Programmtransformation schematisch durchführen. Wir werden dazu im folgenden zunächst die Sprache C_0 in Analogie zu H_0 geeignet einschränken und dann einen Algorithmus zur Transformation von C_0-Programmen in H_0-Programme angeben. Die umgekehrte Transformation wollen wir nur beispielhaft behandeln. Im letzten Abschnitt wollen wir kurz beleuchten, warum die Transformation von C_0-Programmen in H_0-Programme auch von praktischem Interesse ist.

19.3.1 Transformation von C_0-Programmen in H_0-Programme

Von nun an sollen die folgenden zusätzlichen Einschränkungen für C_0 gelten (d. h. wir betrachten also streng genommen eine Sprache "C_{00}"), die einerseits (wie in H_0) die Eingabe bzw. Ausgabe nur am Programmanfang bzw. am Programmende erlauben, und andererseits zur Vereinfachung der Transformation die Variablennamen standardisieren und keine Konstanten zulassen:

- Es gibt keine Konstantendeklaration.
- Die Variablendeklaration ist vorhanden und hat die Form `int x1,x2, ... ,xm;` mit `m` ≥ 1.
- Leseanweisungen sind nur am Anfang der Anweisungsfolge des Blocks erlaubt und haben die Form `scanf("%d",&x1); scanf("%d",&x2);... scanf("%d",&xk);` wobei $0 \leq$ `k` $\leq$ `m`.
- Es gibt genau eine Schreibanweisung, und diese ist am Ende der Anweisungsfolge des Blocks (vor dem Befehl `return 0;`).

Das heißt, dass C_0-Programme hier immer die folgende Form haben:

```
#include <stdio.h>

int main()
{ int x1, x2, ..., xm;
  scanf("%d", &x1);
  scanf("%d", &x2);
  ...
  scanf("%d", &xk);

  <stat-seq>

  printf("%d", xi);
  return 0;
}
```

wobei `<stat-seq>` $\in$ W($\langle$StatementSequence$\rangle$), also eine Sequenz von Statements ist. Um im folgenden die Darstellung kurz zu halten, nennen wir ein solches Programm (m, k, i)-C_0-Programm (m Variablen, k Eingabevariablen, xi wird ausgegeben) und geben nur noch `<stat-seq>` an.

Beispiel 19.4. Gegeben sei das folgende $(2, 2, 2)$-C_0-Programm

```
f:  while (x1>0)
    { x1 = x1-1;
      x2 = x2+1;
    }
```

zur Summenberechnung. Dies können wir in das folgende H_0-Programm transformieren:

```
module Main where

f :: Int -> Int -> Int
f x1 x2 = if x1>0 then f (x1-1) (x2+1)
                  else x2
```

```
main = do x1 <- readLn
          x2 <- readLn
          print (f x1 x2)
```
□

Nun wollen wir eine *schematische* Transformation von C_0 nach H_0 angeben. Die wesentliche Idee der Transformation, die zuerst in [McC60] im Rahmen von Flussdiagrammen beschrieben wurde besteht im folgenden: Vor jedes Statement von `<stat-seq>` und vor `printf("%d", xi)` legen wir einen *Ablaufpunkt*. Jedem Ablaufpunkt ordnen wir eine m-stellige Funktion `f` zu, die als Argumente genau die m Programmvariablen `x1,x2, ... ,xm` hat. Dann beschreiben wir den Funktionswert `f(x1,...,xm)` mit Hilfe derjenigen Funktionen, die den als nächstes erreichbaren Ablaufpunkten zugeordnet sind (*continuation semantics*).
In der schematischen Transformation verwenden wir als Funktionsnamen Elemente aus der Menge $Adr = \{f\} \cdot \mathbb{N}^*$.

Beispiel 19.5. Gegeben sei das $(2, 2, 1)$-C_0-Programm aus Abbildung 19.3. In Abbildung 19.4 ist dasselbe Programm als Flussdiagramm dargestellt; hier erkennt man gut die Ablaufpunkte mit den zugeordneten Funktionen. An manchen Ablaufpunkten erscheinen mehrere Funktionen; das liegt daran, dass ein Block `{<stat-seq>}` und die darin enthaltene Sequenz `<stat-seq>` im Flussdiagramm nicht getrennt dargestellt werden.

Um die Wirkung der einzelnen Funktionen beschreiben zu können, übersetzen wir das $(2, 2, 1)$-C_0-Programm in ein AM_0-Programm P, behalten aber die oben eingefügten baumstrukturierten Adressen an den jeweils eindeutig bestimmten Positionen bei.

P:

```
f1:                          LOAD 1; LIT 0; GT;
                             JMC f2;
    f11, f111:               LOAD 1; LIT 1; SUB; STORE 1;
         f112:               LOAD 1; LOAD 2; EQ;
                             JMC f113;
              f1121, f11211: LOAD 2; LIT 1; ADD; STORE 2;
         f113:               LOAD 2; LIT 1; ADD; STORE 2;
```

```
f1:     while (x1 > 0)
f11:    {
f111:     x1 = x1 - 1;
f112:     if (x1 == x2)
f1121:    {
f11211:     x2 = x2 + 1;
          }
f113:     x2 = x2 + 1;
        }
f2:     x1 = x2;
f3:
```

Abbildung 19.3: C_0-Code-Fragment

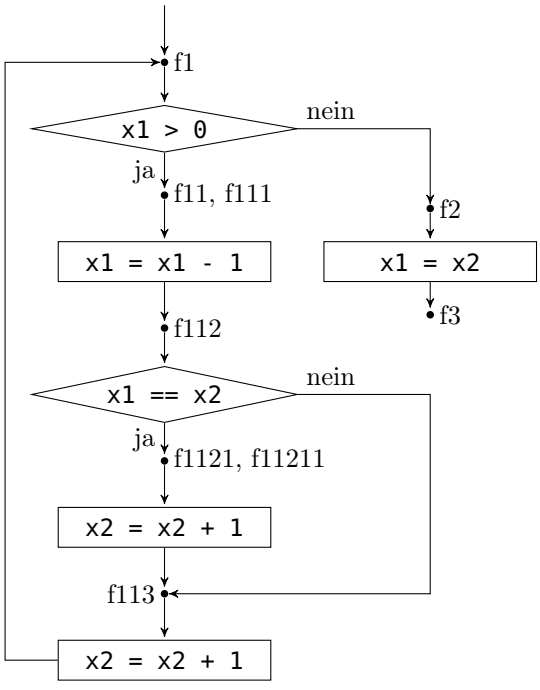


Abbildung 19.4: Flussdiagramm

```
                          JMP f1;
f2:                       LOAD 2; STORE 1;
```
□

Wenn P linearisiert wird, dann gehört zu jeder baumstrukturierten Adresse adr (z. B. f111) eine lineare Adresse; diese nennen wir $\overline{adr}$.

Nun gilt der folgende Zusammenhang zwischen der Funktion adr (wobei adr eine baumstrukturierte Adresse in P ist) und der Semantik von P:

$$adr\ h(1)\ h(2) = proj_5^{(5)}(\mathcal{I}[\![P]\!](\overline{adr}, \varepsilon, h, \varepsilon, \varepsilon))$$

für jede beliebige Hauptspeicherbelegung h. Im allgemeinen hat die Gleichung für ein (m, k, i)-C_0-Programm P die Form:

$$adr\ h(1)\ \ldots h(m) = proj_5^{(5)}(\mathcal{I}[\![P]\!](\overline{adr}, \varepsilon, h, \varepsilon, \varepsilon)).$$

(Man macht sich leicht klar, dass nach Ausführung jeder C_0-Anweisung der Datenkeller leer ist.)

Beispiel 19.5 (Fortsetzung)**.** Kehren wir zur Transformation des $(2, 2, 1)$-C_0-Programms P aus Abbildung 19.3 zurück. Als Ergebnis der Transformation von P nach H_0 erhalten wir die folgenden Funktionen:

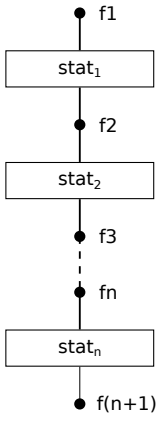
```
f1 x1 x2     = if x1 > 0 then f11 x1 x2
                         else f2 x1 x2
f2 x1 x2     = f3 x2 x2
f3 x1 x2     = x1

f11 x1 x2    = f111 x1 x2
f111 x1 x2   = f112 (x1 - 1) x2
f112 x1 x2   = if x1 == x2 then f1121 x1 x2
                           else f113 x1 x2
f1121 x1 x2  = f11211 x1 x2
f11211 x1 x2 = f113 x1 (x2 + 1)
f113 x1 x2   = f1 x1 (x2 + 1)

main = do x1 <- readLn
          x2 <- readLn
          print (f1 x1 x2)
```
□

Die Transformationsfunktion *sttrans* für Anweisungen hat zwei baumstrukturierte Adressen als Parameter, aus denen die Namen derjenigen zwei Funktionen generiert werden, mit der die Simulation einer Anweisung *stat* begonnen wird bzw. mit der die Simulation der auf *stat* folgenden Anweisung begonnen wird.

Ein weiterer Parameter von *sttrans* gibt die Anzahl m der im C_0-Programm deklarierten Variablen an. Auf Grund der standardisierten Variablennamen wird für die Transformation wiederum keine Symboltabelle benötigt.

- *Transformation von C_0–Programmen:*

  trans: W(⟨Program⟩) -↛ W(⟨Prog⟩)
  trans(
  ```
  #include <stdio.h>
  int main()
  { int x1, ... ,xm;
    scanf("%d",&x1);
    ...
    scanf("%d",&xk);
  ```
  $stat_1\ stat_2\ \ldots\ stat_n$
  ```
    printf("%d",xi);
    return 0;
  }
  ```
) :=



```
module Main where
sttrans(stat_1, 1, 2, m)
sttrans(stat_2, 2, 3, m)
...
sttrans(stat_n, n, n+1, m)
f(n+1) :: Int -> ... -> Int                    (m-mal ->)
f(n+1) x1 ... xm = xi
main = do x1 <- readLn ... xk <- readLn
          print (f1 x1 ... xk 0 ... 0)         ((m − k)-mal 0)
```

für alle $\mathtt{n} \geq 0$, $stat_1, stat_2, \ldots, stat_n \in W(\langle\text{Statement}\rangle)$,
$k \geq 0$, und $i, m \geq 1$ mit $k \leq m$ und $i \leq m$.

- *Transformation von Anweisungen:*

  sttrans: $\mathrm{W}(\langle\text{Statement}\rangle) \times Adr^2 \times \mathbb{N} \dashrightarrow (\mathrm{W}(\langle\text{Fun}\rangle)^* \setminus \{\varepsilon\})$

  sttrans(`xi =` $se, \mathtt{a}, \mathtt{b}, m$) :=

  ```
  fa :: Int -> ... -> Int                (m-mal ->)
  fa x1 ... xm = fb x1 ... x(i − 1) setrans(se) x(i + 1) ... xm
  ```

  für alle $se \in W(\langle\text{SimpleExpression}\rangle)$, $\mathtt{a}, \mathtt{b} \in Adr$ und $i, m \geq 1$ mit $i \leq m$,

  sttrans(`if` (be) $stat, \mathtt{a}, \mathtt{b}, m$) :=

  ```
  fa :: Int -> ... -> Int                (m-mal ->)
  fa x1 ... xm = if betrans(be) then fa1 x1 ... xm
                                else fb   x1 ... xm
  sttrans(stat, a1, b, m)
  ```

  für alle $be \in W(\langle\text{BoolExpression}\rangle)$, $stat \in W(\langle\text{Statement}\rangle)$, $\mathtt{a}, \mathtt{b} \in Adr$ und $m \geq 1$,

  sttrans(`if` (be) $stat_1$ `else` $stat_2, \mathtt{a}, \mathtt{b}, m$) :=

  ```
  fa :: Int -> ... -> Int                (m-mal ->)
  fa x1 ... xm = if betrans(be) then fa1 x1 ... xm
                                else fa2 x1 ... xm
  sttrans(stat_1, a1, b, m)
  sttrans(stat_2, a2, b, m)
  ```

  für alle $be \in W(\langle\text{BoolExpression}\rangle)$, $stat_1, stat_2 \in W(\langle\text{Statement}\rangle)$, $\mathtt{a}, \mathtt{b} \in Adr$ und $m \geq 1$,

  sttrans(`while` (be) $stat, \mathtt{a}, \mathtt{b}, m$) :=

  ```
  fa :: Int -> ... -> Int                (m-mal ->)
  fa x1 ... xm = if betrans(be) then fa1 x1 ... xm
                                else fb   x1 ... xm
  sttrans(stat, a1, a, m)
  ```

  für alle $be \in W(\langle\text{BoolExpression}\rangle)$, $stat \in W(\langle\text{Statement}\rangle)$,
  $\mathtt{a}, \mathtt{b} \in Adr$, $m \geq 1$ (beachte: die 2. Adresse lautet $\mathtt{a}$),

  sttrans(`{` $stat_1\ stat_2\ \ldots\ stat_n$ `}`, $\mathtt{a}, \mathtt{b}, m$) :=

  ```
  fa :: Int -> ... -> Int                (m-mal ->)
  fa x1 ... xm = fa1 x1 ... xm
  sttrans(stat_1, a1, a2, m)
  sttrans(stat_2, a2, a3, m)
  ⋮
  sttrans(stat_n, an, b, m)
  ```

  für alle $stat_1, \ldots, stat_n \in W(\langle\text{Statement}\rangle)$, $\mathtt{a}, \mathtt{b} \in Adr$ und $m \geq 1$.

  Man beachte, dass die Transformation einer Anweisung eine nicht leere Sequenz von Funktionsdefinitionen liefert.

- *Transformation von arithmetischen und Booleschen Ausdrücken:*

  Die oben verwendeten Funktionen *betrans* und *setrans* zur Transformation von Booleschen bzw. arithmetischen Ausdrücken sollen hier nicht formal spezifiziert werden. Bei ihrer Definition sind nur die verschiedene Klammerstruktur und die verschiedenen Bezeichner für Operatoren (`'div'`, `'mod'` und `/=` statt `/`, `%` und `!=`) in H_0 und C_0 zu beachten.

Beispiel 19.4 (Fortsetzung)**.** Durch Anwendung von *trans* auf das C_0-Programm für die Summenberechnung (Seite 245) erhält man das folgende H_0-Programm, bei dem wir auf die Angabe der Funktionstypen verzichtet haben:

```
module Main where

f1   x1 x2 = if x1 > 0 then f11 x1 x2
                       else f2 x1 x2
f11  x1 x2 = f111 x1 x2
f111 x1 x2 = f112 (x1 - 1) x2
f112 x1 x2 = f1 x1 (x2 + 1)
f2   x1 x2 = x2

main = do x1 <- readLn
          x2 <- readLn
          print (f1 x1 x2)
```

□

19.3.2 Transformation von H_0-Programmen in C_0-Programme

Die Simulation eines H_0-Programms p durch ein C_0-Programm gelingt mit nur einer einzigen (!) while-Schleife, die solange durchlaufen wird, wie in p noch rekursive Funktionsaufrufe auszuführen sind. Dazu wird eine neue Variable (im folgenden Beispiel `flag`) verwendet, die die Fälle "noch ein rekursiver Aufruf" und "kein rekursiver Aufruf mehr" unterscheidet.

Im Rumpf der while-Schleife befinden sich die Übersetzungen der einzelnen Funktionsdefinitionen von p. Die Selektion der als nächstes auszuführenden Funktion erfolgt durch eine weitere neue Variable (im folgenden Beispiel `function`), deren Wert in einer geschachtelten if-Anweisung abgefragt wird; (hier wäre ein case-Statement sicherlich angebracht; wir haben aber in C_0 kein case-Statement zur Verfügung und müssen uns deshalb mit geschachtelten if-Anweisungen behelfen) es ist klar, dass `function` genauso viele mögliche Werte annehmen kann, wie es Funktionen in p gibt. Eine dritte neue Variable (im folgenden Beispiel `result`) übernimmt das Rechenergebnis, das nach Beendigung der while-Schleife ausgegeben wird.

Beispiel 19.7. Das nachfolgende H_0-Programm zur Berechnung von $\sum_{i=0}^{x1} i$ (falls $x1$ gerade) bzw. $\prod_{i=1}^{x1} i$ (falls $x1$ ungerade) soll mit Hilfe dieser Idee in ein äquivalentes C_0-Programm überführt werden.

```
 1  module Main where
 2
 3  f1 :: Int -> Int -> Int -> Int
 4  f1 x1 x2 x3 = if x1==0 then x2
 5                         else f2 (x1-1) (x2+x1) (x3*x1)
 6
 7  f2 :: Int -> Int -> Int -> Int
 8  f2 x1 x2 x3 = if x1==0 then x3
 9                         else f1 (x1-1) (x2+x1) (x3*x1)
10
11  main = do x1 <- readLn
12            print (f1 x1 0 1)
```

Betrachten wir beispielhaft einmal den Rechenablauf für die Eingabe `x1` = 3:

	x1	x2	x3
f1	3	0	1
f2	x1-1 =2	x2+x1 =0+3	x3∗x1 =1∗3
f1	x1-1 =1	x2+x1 =0+3+2	x3∗x1 =1∗3∗2
f2	x1-1 =0	x2+x1 =0+3+2+1	x3∗x1 =1∗3∗2∗1
	$\rightsquigarrow \frac{x1*(x1+1)}{2}$		$\rightsquigarrow x1!$

Intuitiv lässt sich hieraus das folgende C-Programm entwickeln:

```
 1  #include <stdio.h>
 2
 3  int main()
 4  { int x1, x2, x3, flag, function,
 5        result;
 6
 7    scanf("%d", &x1);
 8    x2 = 0;
 9    x3 = 1;
10    flag = 1;
11    function = 1;
12    while (flag)
13      switch (function)
14      { case 1:
15          if (x1 == 0)
16          { result = x2;
17            flag = 0;
18          }
19          else
20          { x2 = x2+x1;
21            x3 = x3*x1;
22            x1 = x1-1;
23            function = 2;
24          }
25          break;
26        case 2:
27          if (x1 == 0)
28          { result = x3;
29            flag = 0;
30          }
31          else
32          { x2 = x2+x1;
33            x3 = x3*x1;
34            x1 = x1-1;
35            function = 1;
36          }
37      }
38      printf("%d", result);
39      return 0;
40  }
```

Dieses Programm lässt sich nun leicht in ein C_0-Programm überführen:

```
 1  #include <stdio.h>
 2
 3  int main()
 4  { int x1,x2,x3,flag,function,result;
 5
 6    scanf("%d",&x1);
 7
 8    x2 = 0;         /* Initialisierung für Summe     */
 9    x3 = 1;         /* Initialisierung für Produkt   */
10    flag = 1;       /* flag für Schleifenausführung */
11    function = 1;   /* beginne mit Funktion f1        */
12    while (flag==1)
13      if (function==1)
14        if (x1==0)
15        { result = x2;   /* Resultat ist x2 */
16          flag = 0;      /* Schleifenende   */
17        }
18        else
19        { x2 = x2+x1;
20          x3 = x3*x1;
21          x1 = x1-1;
22          function = 2;      /* gehe zu Funktion f2 */
23        }
```

```
24        else if (function==2)
25          if (x1==0)
26          { result = x3;   /* Resultat ist x3 */
27            flag = 0;      /* Schleifenende   */
28          }
29          else
30          { x2 = x2+x1;
31            x3 = x3*x1;
32            x1 = x1-1;
33            function = 1;    /* gehe zu Funktion f1 */
34          }
35    printf("%d",result);
36    return 0;
37  }
```

□

Für eine schematische Transformation ist zu beachten, dass die Änderung der Parameterwerte (in unserem Beispiel von `x1`, `x2` und `x3`) jeweils zunächst in Hilfsvariablen vorgenommen werden muss, da die Parameterwerte im allgemeinen gegenseitig voneinander abhängen. Im obigen Beispiel ließ sich dies durch eine "geschickte" Reihenfolge der Zuweisungen für `x1`, `x2` und `x3` vermeiden. Ferner ist eine Symboltabelle zu verwenden, die den im allgemeinen beliebigen Funktionsnamen paarweise verschiedene Integerwerte zuordnet, über die dann die Fallunterscheidung erfolgt.

19.3.3 Schleifeninvariante versus Induktionshypothese

In Kapitel 18 hatten wir den Hoare-Kalkül [Hoa69] als formales Hilfsmittel zur Konstruktion von Beweisen für Programmeigenschaften kennengelernt. Dabei hatten wir gesehen, dass die Konstruktion eines Beweises (insbesondere das Finden einer geeigneten Schleifeninvariante) nicht vollständig algorithmisierbar ist. Aus diesem Grunde werden imperative Programme oft in äquivalente funktionale Programme übersetzt, um die für diese Sprachen konzipierten Induktionsbeweiser (vgl. auch Abschnitt 15.7) zu verwenden. Da man aber zeigen kann, dass auch über diesen Umweg eine vollständige Algorithmisierung der Konstruktion eines Beweises nicht gelingen kann, stellt sich die Frage, wo sich auf der Ebene der funktionalen Programme das Problem des Findens einer Schleifeninvariante widerspiegelt:

Wir hatten in Abschnitt 19.3.1 gesehen, dass bei dieser Übersetzung immer tail-rekursive Funktionen entstehen. Diese werden zwar im allgemeinen aus Effizienzgründen gegenüber nicht tail-rekursiven Funktionen bevorzugt, sind aber aus beweistechnischer Sicht meist weniger gut, weil das Finden einer geeigneten Induktionshypothese nicht automatisierbar ist. Der Grund hierfür ist, dass bei tail-rekursiven Programmen die eigentlichen Berechnungen in den Parametern der Funktionen stattfinden müssen, d. h. die Parameterwerte ändern sich von Funktionsaufruf zu Funktionsaufruf. Man spricht von sogenannten *akkumulierenden Parametern*. Oft ist eine Aussage für ganz konkrete Parameterwerte zu beweisen, jedoch erfordert der Beweis selbst eine geeignete *Generalisierung* der Parameterwerte. Dies wollen wir am Beispiel der Funktion `sum` aus dem tail-rekursiven Programm für die Berechnung der Summe zweier Zahlen `x1` und `x2` verdeutlichen:

```
sum x1 x2 = if x1>0 then sum (x1-1) (x2+1) else x2
```

Da die Rekursion durch Abstieg über den Parameter `x1` erfolgt, heißt `x1` *Rekursionsparameter*. Der Parameter `x2` ist ein akkumulierender Parameter.

Es sei nun die (einfache) Aussage "`sum x1 0 = x1` für alle `x1` $\in \mathbb{N}$" zu beweisen. Ein automatischer Beweiser nimmt diese Aussage als Induktionshypothese an. Der Beweis des Induktionsanfangs

```
x1=0:   sum 0 0 = if 0>0 then sum (0-1) (0+1) else 0
                = 0
```

gelingt problemlos, aber im Induktionsschritt

```
x1 -> x1+1:   sum (x1+1) 0 = if (x1+1)>0 then sum ((x1+1)-1) (0+1) else 0
                           = sum x1 1
                           = ?
```

ist die Induktionshypothese nicht anwendbar, da sich der zweite Parameter von 0 auf 1 verändert hat. Es ist eine Generalisierung zur Induktionshypothese "`sum x1 x2 = x1+x2` für alle `x1,x2` $\in \mathbb{N}$" nötig, mit der dann der Induktionsbeweis gelingt. Während die notwendige Generalisierung in unserem Beispiel offensichtlich ist, ist sie im allgemeinen leider nicht automatisierbar.

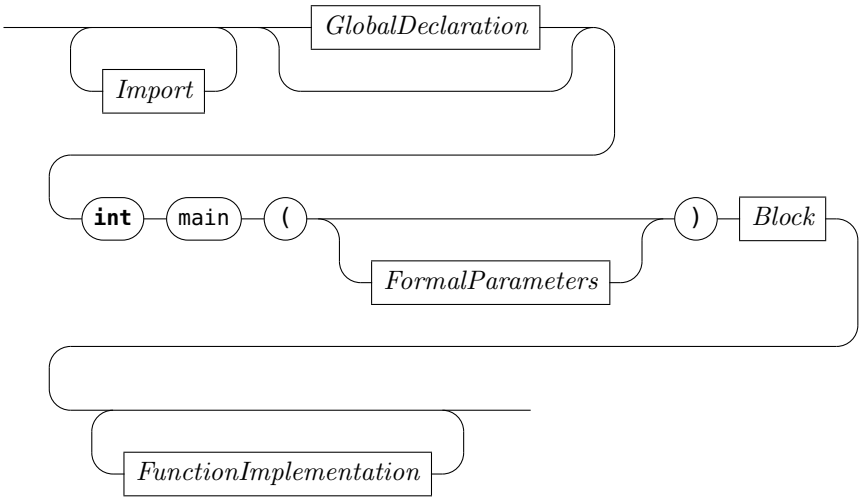
Abschließend möchten wir noch erwähnen, dass die folgende zur Funktion `sum` äquivalente, aber nicht tail-rekursive Funktion

```
sum' x1 x2 = if x1>0 then (sum' (x1-1) x2) + 1 else x2
```

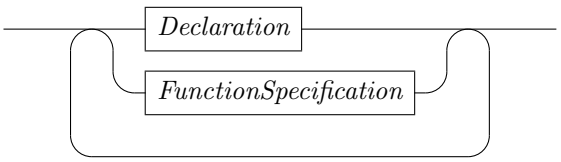
für automatische Beweissysteme geeigneter ist, weil sich der Parameter `x2` im rekursiven Aufruf nicht verändert. Manche tail-rekursiven Programme lassen sich automatisch in geeignete nicht tail-rekursive Programme transformieren. Solche Transformationen werden *Deakkumulation* genannt und sind z. B. in [Gie00, GKV03] zu finden.

A Syntaxdiagramme

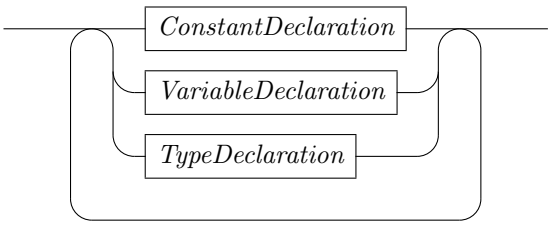
Program



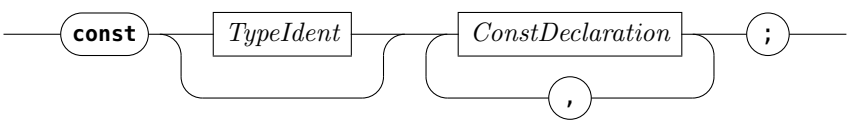
GlobalDeclaration



Declaration

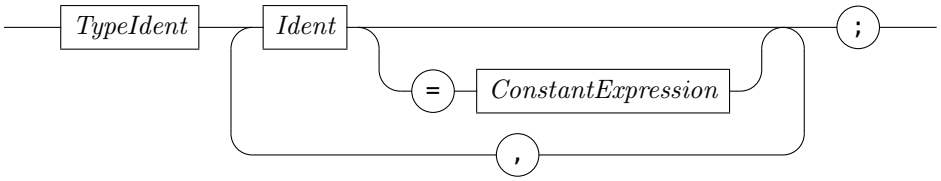


ConstantDeclaration

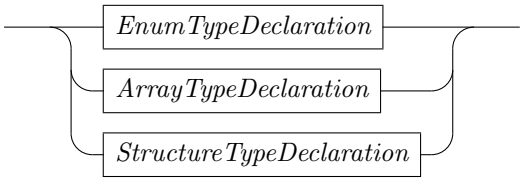


ConstDeclaration

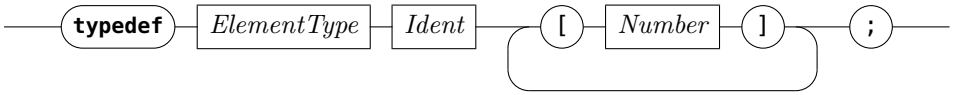
VariableDeclaration



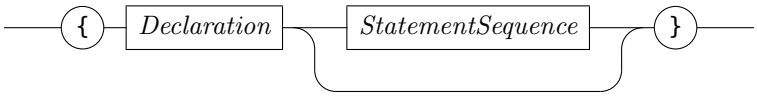
TypeDeclaration



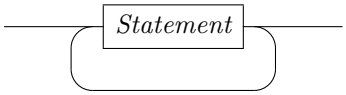
ArrayTypeDeclaration



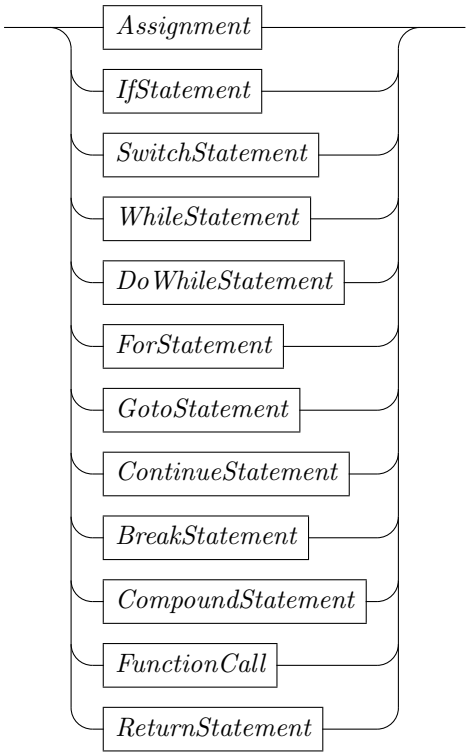
Block



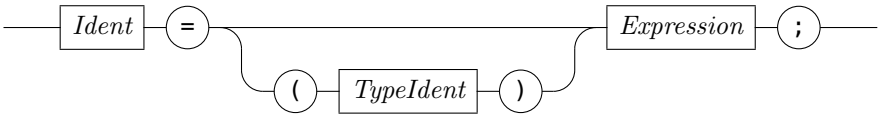
StatementSequence



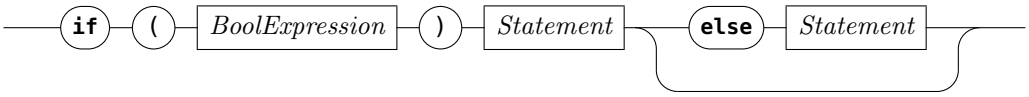
Statement



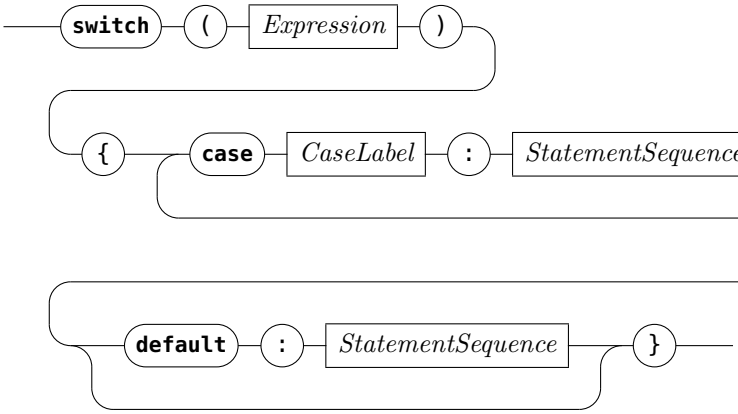
Assignment



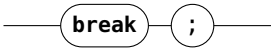
IfStatement



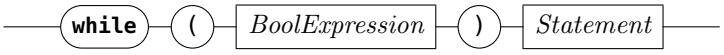
SwitchStatement



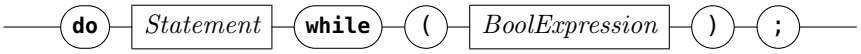
BreakStatement



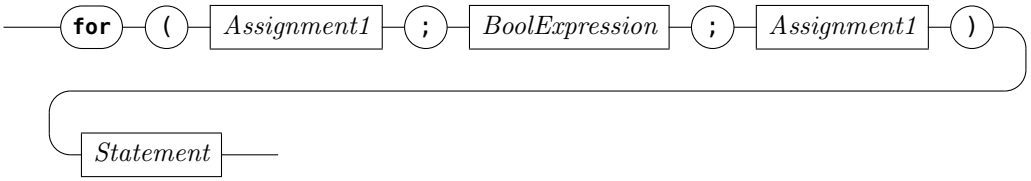
WhileStatement



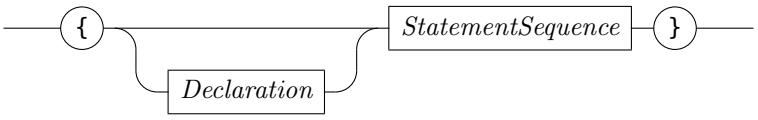
DoWhileStatement



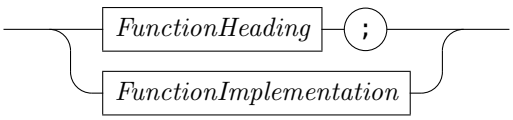
ForStatement



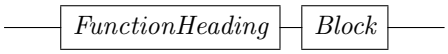
CompoundStatement



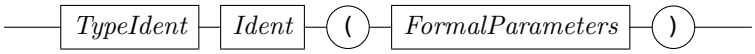
FunctionSpecification



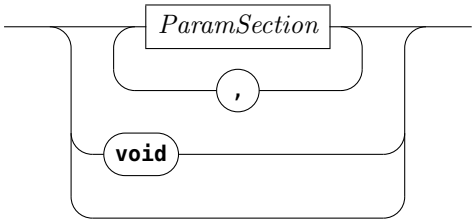
FunctionImplementation



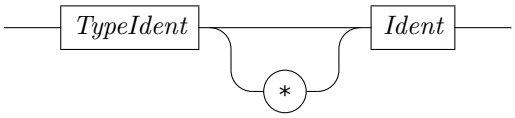
FunctionHeading



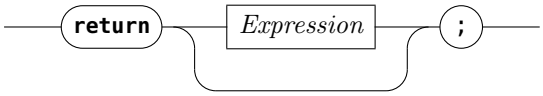
FormalParameters



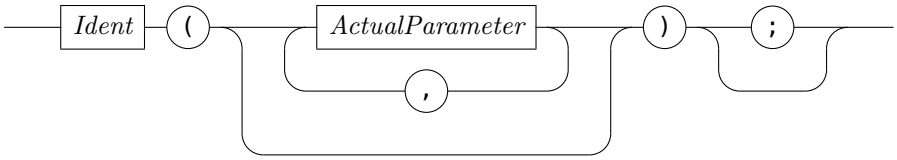
ParamSection



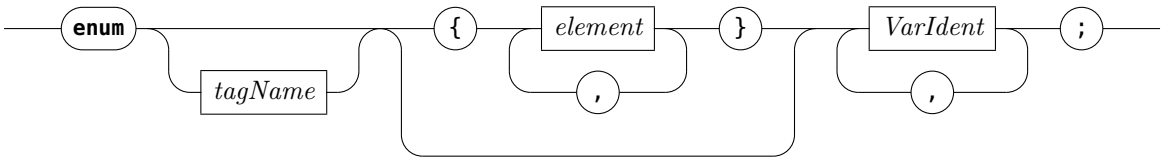
ReturnStatement



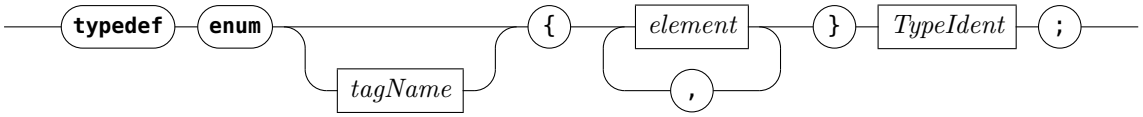
FunctionCall



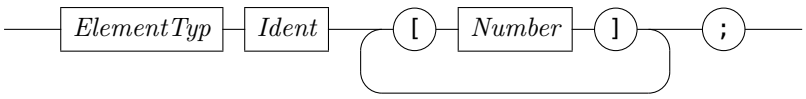
EnumType



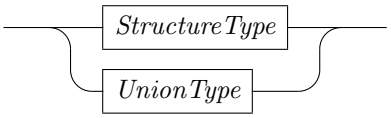
EnumTypeDeclaration



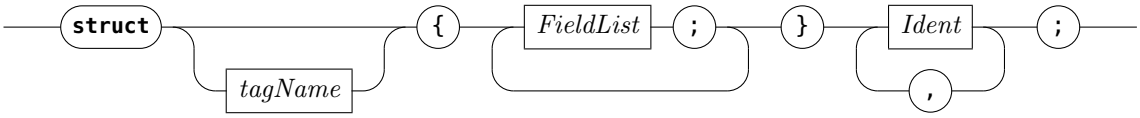
ArrayType



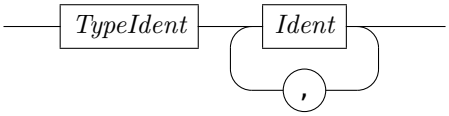
RecordType



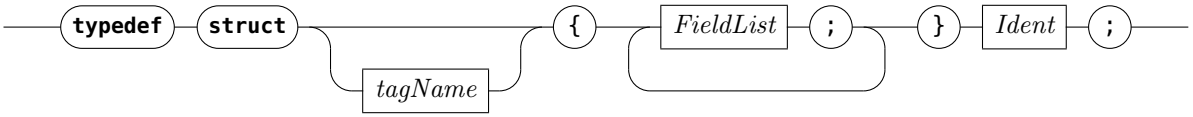
StructureType



FieldList

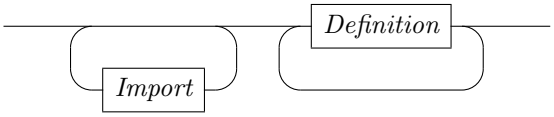


StructureTypeDeclaration

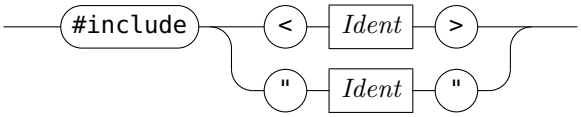


UnionType

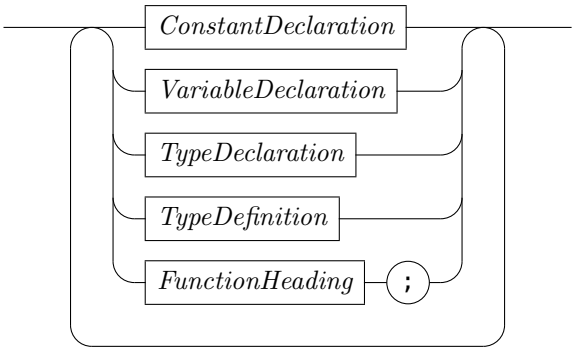
DefinitionModule



Import

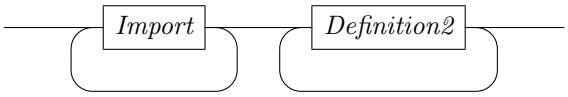


Definition

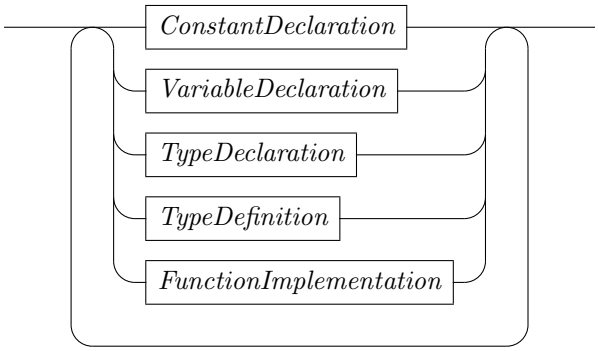


TypeDefinition

ImplementationModule



Definition2



B Mathematische Grundlagen

Hier stellen wir grundlegende mathematische Begriffe und Gesetze vor, die für die Vorlesungen „Algorithmen und Datenstrukturen“ sowie „Programmierung“ von fundamentaler Bedeutung sind und demzufolge als notwendige Voraussetzungen für das Verständnis dieser Vorlesungen angesehen werden müssen.

Die Zielstellung dieses mathematischen Einschubs soll erfüllt sein, wenn der Leser einerseits auf bereits Bekanntes aufmerksam gemacht wird – um es ggf. wieder aufzufrischen – und andererseits evtl. vorhandene Lücken erkennen kann, um diese mit Hilfe geeigneter Lehrbücher schließen zu können.

Damit ist auch gesagt, dass diese kurze Einführung ein entsprechendes Lehrbuch weder ersetzen kann noch soll. Ein Anspruch auf Vollständigkeit kann ebenso wenig erfüllt werden wie eine umfassende didaktische Aufbereitung.

B.1 Einführung in die mathematische Logik

B.1.1 Aussagen und Aussageformen

Definition B.1. Eine *Aussage* A ist ein schriftlich oder sprachlich formulierter Sachverhalt, von dem es sinnvoll ist zu fragen, ob er wahr oder falsch ist. □

Jede Aussage ist somit entweder wahr (Kurzbezeichnung: T oder W oder 1) oder falsch (Kurzbezeichnung: F oder 0).

Beachte: Die genannte Zweiwertigkeit von Aussagen besagt nicht, dass man von jeder Aussage entscheiden kann, ob sie wahr oder falsch ist! Auch ist o. g. Zweiwertigkeit keine zwingende Voraussetzung für den Aufbau einer mathematischen Logik.

Definition B.2. Eine *Aussageform* ist ein schriftlich oder sprachlich formulierter Sachverhalt mit mindestens einer freien Stelle – man spricht auch von einer „freien Variablen“ –, in die man Elemente aus einem Grundbereich einsetzen kann („Belegen der Variablen“). Nach dem Einsetzen ensteht aus der Aussageform eine Aussage. □

Beispiel B.3. „x ist eine natürliche Zahl“ mit dem Grundbereich $\mathbb{R}$ ($\mathbb{R}$: Menge der reellen Zahlen) und der freien Variablen x ist eine Aussageform und könnte in Kurzform mit $A(x)$, $x \in \mathbb{R}$ bezeichnet werden. Je nach Festlegung von x wird jetzt eine wahre oder falsche Aussage entstehen. □

Die Aussagenlogik interessiert sich nun vordergründig dafür, wie sich Wahrheitswerte bei komplizierteren Aussagengebilden, hier mit *Aussagenverbindungen* bezeichnet, aus den Wahrheitswerten der elementaren Aussagen ermitteln lassen.

Um sicherzustellen, dass nur die in der klassischen zweiwertigen Aussagenlogik (nur die soll hier betrachtet werden) definierten Aussagenverbindungen entstehen können, werden elementare Verknüpfungen von Aussagen definiert, die stets wieder zulässige Aussagen mit jeweils definierten Wahrheitswerten erzeugen und somit auch bei wiederholter Anwendung nicht aus der Menge der zulässige Aussagenverbindungen herausführen.

Diese eben genannten Verknüpfungen werden *aussagenlogische Funktoren* oder auch *Junktoren* genannt; es sind die Verknüpfungen $\wedge$ (Konjunktion), $\vee$ (Disjunktion) und $\neg$ (Negation) sowie $\Rightarrow$ (Implikation) und $\Leftrightarrow$ (Äquivalenz). Wenn also z. B. A und B zwei Aussagen sind, so sind $A \wedge B$, $A \vee B$ und $\neg A$ auch Aussagen. Die Zuordnung der Wahrheitswerte zu den durch Anwendung der Verknüpfungen entstehenden neuen Aussagen wird mit Hilfe sogenannter *Aussagenfunktionen* vermittelt.

B.1.2 Aussagefunktionen

Für jeden aussagenlogischen Junktor $\phi \in \{\wedge, \vee, \neg\}$ gibt es eine Aussagefunktion. Sie liefert für Wahrheitswerte der durch ϕ verknüpften Aussagen den Wahrheitswert der zusammengesetzten Aussage.

Seien A und B Aussagen:

- **Negation**: Mit „nicht A“ bzw. $\neg A$ bezeichnet und festgelegt durch die Wertetabelle:

A	$\neg A$
T	F
F	T

- **Konjuktion**: Mit „A und B“ bzw. $A \wedge B$ bezeichnet und festgelegt durch die Wertetabelle:

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

- **Disjunktion**: Mit „A oder B“ bzw. $A \vee B$ bezeichnet und festgelegt durch die Wertetabelle:

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

- **Implikation**: Mit „wenn A, so B“ bzw. $A \Rightarrow B$ bezeichnet und festgelegt durch die Wertetabelle:

A	B	$A \Rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

- **Äquivalenz**: Mit „A genau dann, wenn B“ bzw. $A \Leftrightarrow B$ bezeichnet und festgelegt durch die Wertetabelle:

A	B	$A \Leftrightarrow B$
T	T	T
T	F	F
F	T	F
F	F	T

Diese Aussagenfunktionen und alle Aussagenfunktionen, die man durch Superposition (d. h. Kombinieren) der Aussagenfunktionen erhalten kann, sind extensionale Aussagenfunktionen, d. h. die Wahrheit oder Falschheit der zugeordneten Aussagen hängt nur von der Wahrheit oder Falschheit der zugehörigen Argumente ab und nicht von deren Sinn.

Definition B.4. Eine *n-stellige Aussagenfunktion* (n-stellige Boolesche Funktion) ist eine Abbildung α, durch welche den Wahrheitswerten jedes n-Tupels $(A_1, A_2, \ldots, A_n)$ von Aussagen eindeutig ein Wahrheitswert der resultierenden Aussagenverbindung zugeordnet wird. □

Definition B.5. Zwei Aussagenverbindungen werden *logisch gleichwertig* oder *logisch äquivalent* genannt, wenn bei jeder Aussagenbelegung die Wahrheitswerte der Aussagenfunktionen übereinstimmen. □

Praktisch relevante Beispiele für Paare von logisch äquivalenten Aussageverbindungen sind folgende:

- $A \Rightarrow B$ und $\neg A \vee B$
- $A \Leftrightarrow B$ und $(A \wedge B) \vee (\neg A \wedge \neg B)$
- $A \vee B$ und $\neg(\neg A \wedge \neg B)$, bekannt als de MORGAN'sche Umformung

B.2 Einführung in die Mengenlehre

B.2.1 Mengenbegriff, Mengenbildung

Mengenbegriff von CANTOR:

> „Eine *Menge* ist eine Zusammenfassung bestimmter wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens zu einem Ganzen. Die Objekte in der Menge werden Elemente der Menge genannt."

Um nun entscheiden zu können, ob ein Element zu einer Menge gehört oder nicht, bedient man sich der mathematischen Logik.

Man erklärt einen Grundbereich I von Objekten und eine Aussageform (Eigenschaft) $H(x)$, die für die Objekte aus I definiert ist. Für jedes beliebige $x \in I$ lässt sich somit die Frage stellen, ob $H(x)$ wahr ist oder nicht. Eine Menge M kann nun so definiert werden, dass sie alle Objekte x aus I enthält, für die $H(x)$ eine wahre Aussage ist. Hieraus lässt sich das **Mengenbildungsaxiom** formulieren:

> Zu jeder Aussageform $H(x)$ über I gibt es (genau) eine Menge M, für die gilt: x ist Element von M genau dann, wenn $H(x)$ wahr ist.

Kurz schreibt man diesen Sachverhalt wie folgt auf:

$$M := \{x \mid x \in I \wedge H(x) \text{ ist wahr}\} .$$

Anstatt der mathematischen Mengendefinition wird in der Praxis häufig auch die umgangssprachliche Formulierung dieses Sachverhaltes benutzt.

Beispiel B.6. Gegeben sei der Grundbereich $I = \mathbb{N}$ und die Aussageform $H(x)$: x ist ohne Rest durch die Zahl 3 teilbar. Hiermit erhält man die (abzählbar unendliche) Menge $M = \{0, 3, 6, 9, \dots\}$, also alle Vielfachheiten von 3 (einschließlich der Null) als Elemente von M. In Kurzform schreiben wir $M = \{x \mid x \in \mathbb{N} \text{ und } x \text{ ist ohne Rest durch 3 teilbar }\}$. □

Im folgenden sollen nun einige Festlegungen und Vereinbarungen, die für die Arbeit mit Mengen bedeutungsvoll sind, aufgeschrieben werden.

- $x \in M \Leftrightarrow x$ ist Element von M.
- $x \notin M \Leftrightarrow \neg x \in M$.
- $\{x \mid x \in I \text{ und } x \neq x\} = \emptyset$: *leere Menge*.
- $\{x \mid x \in I \text{ und } x = x\} = I$: *Allmenge* (Grundbereich).
- $\text{card}(M)$ oder $|M|$: *Mächtigkeit* von M; bei endlicher Menge M ist dies die Anzahl der Elemente von M.
- $M_1 = M_2 \Leftrightarrow$ für jedes x gilt: $(x \in M_1 \Leftrightarrow x \in M_2)$: *Extensionalitätsprinzip*, d. h. M_1, M_2 genau dann gleich, wenn sie dieselben Elemente enthalten.
- $\mathbb{N}$: Menge der *natürlichen Zahlen*.
- $\mathbb{N}^+$: $\mathbb{N}^+ := \{x \mid x \in \mathbb{N} \wedge (x > 0)\}$.
- $\mathbb{Z}$: Menge der *ganzen Zahlen*.
- $\mathbb{Q}$: Menge der *rationalen Zahlen*.
- $\mathbb{R}$: Menge der *reellen Zahlen*.

B.2.2 Mengenoperationen, Mengenrelationen

Definition B.7. Seien A und B Mengen über dem Grundbereich I.

- $A \cap B := \{x \mid x \in A \wedge x \in B\}$ heißt der *Durchschnitt* von A und B.
- $A \cup B := \{x \mid x \in A \vee x \in B\}$ heißt die *Vereinigung* von A und B.
- $A \setminus B := \{x \mid x \in A \wedge x \notin B\}$ heißt die *Differenz* von A und B.
- $A \bigtriangleup B := \{x \mid (\ x \in A\ \wedge x \notin B\ ) \vee (\ x \notin A\ \wedge\ x \in B\ )\}$ heißt die *symmetrische Differenz* von A und B. □

Eine gute graphische Veranschaulichung dieser Begriffsbildungen ist mit Hilfe der sogenannten *Venn-Diagramme* möglich und sollte auch vom Leser zum besseren Verständnis praktiziert werden.

Aufbauend auf die eben definierten Mengenbildungen, sollen nun einige wichtige Gesetzmäßigkeiten formuliert werden. Seien A, B und C beliebige Mengen über den Grundbereich I. Dann gilt:

$$\begin{aligned}
A \cup \emptyset &= A\,, & A \cap \emptyset &= \emptyset & \\
A \cup A &= A\,, & A \cap A &= A & \text{(Idempotenz)} \\
A \cup B &= B \cup A\,, & A \cap B &= B \cap A & \text{(Kommutativität)} \\
A \cup (B \cup C) &= (A \cup B) \cup C\,, & A \cap (B \cap C) &= (A \cap B) \cap C & \text{(Assoziativität)} \\
A \cup (B \cap C) &= (A \cup B) \cap (A \cup C), & A \cap (B \cup C) &= (A \cap B) \cup (A \cap C) & \text{(Distributivität)} \\
A \setminus (B \cup C) &= (A \setminus B) \cap (A \setminus C)\,, & A \setminus (B \cap C) &= (A \setminus B) \cup (A \setminus C) & \text{(Regeln von de MORGAN)}
\end{aligned}$$

Als nächstes folgen einige wichtige Grundlagen über *Teilmengen*.

Definition B.8. A und B seien Mengen. A heißt *Teilmenge* oder *Untermenge* von B, geschrieben $A \subseteq B$, wenn jedes Element von A auch Element von B ist. A heißt *echte* Teilmenge oder *echte* Untermenge von B, geschrieben $A \subset B$, wenn gilt: A ist Teilmenge von B und es ist $A \neq B$.

Den Sachverhalt $A \subseteq B$ ($A \subset B$) bezeichnet man auch als Inklusion (bzw. echte Inklusion). □

Die Inklusion von Mengen lässt sich als sogenannte *partielle Ordnung* auffassen, denn es gelten die drei dafür notwendigen Gesetze: Für drei beliebige Mengen A,B und C gilt:

1. $A \subseteq A$ *Reflexivität*,
2. $A \subseteq B \wedge B \subseteq C \;\Rightarrow\; A \subseteq C$ *Transitivität*,
3. $A \subseteq B \wedge B \subseteq A \;\Rightarrow\; A = B$ *Antisymmetrie*.

Beachte:

- Gilt $A \cap B = \emptyset$, dann nennt man A und B *disjunkt* (elementefremd).
- Für jede Menge M gilt: $\emptyset \subseteq M$.
- Mengengleichheit wird oft mit Hilfe der Beziehung $A = B \;\Leftrightarrow\; A \subseteq B \wedge B \subseteq A$ bewiesen.

B.2.3 Weitere Mengenbildungen

Definition B.9. *Mengen zweiter Stufe* haben als Elemente Mengen erster Stufe, d. h. Mengen von Objekten über der Grundmenge I. □

Eine besonders oft genutzte Menge zweiter Stufe ist die *Potenzmenge* einer Menge M.

Definition B.10. Sei M eine Menge. Die Menge aller Teilmengen von M heißt die *Potenzmenge von M* und wird hier mit $\mathcal{P}(M)$ bezeichnet. □

Gilt $\text{card}(M) = n$ mit $n \in \mathbb{N}$, dann hat $\mathcal{P}(M)$ genau 2^n Elemente, d. h. es ist $\text{card}(\mathcal{P}(M)) = 2^n$. Sei zum Beispiel $M = \{1, 2, 3\}$, dann erhält man:

$$\mathcal{P}(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}\,.$$

Definition B.11. A und B seien Mengen. Die Menge $A \times B$ (lies: A Kreuz B) aller geordneten Paare (x, y) mit $x \in A$ und $y \in B$, wobei B nicht notwendig von A verschieden, wird das *kartesische Produkt* (oder *Kreuzprodukt*) von A und B genannt.

Formal: $A \times B := \{(x, y) \mid x \in A \wedge y \in B\}$. □

Sei zum Beispiel $A = \{1, 2\}$ und $B = \{a, b, c\}$, so erhält man:

$$A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}\,.$$

Beachte:

- Im allgemeinen gilt: $A \times B \neq B \times A$.
- $A \times \emptyset = \emptyset \times A = \emptyset$.
- Zwei Paare (x, y) und (s, t) sind genau dann gleich, wenn gilt: $x = s$ und $y = t$.

Definition B.12. $A_1, A_2, \ldots, A_n$ $(n \geq 2)$ seien Mengen (nicht notwendig über dem gleichen Grundbereich).
Die Menge $A_1 \times A_2 \times \cdots \times A_n := \{(x_1, x_2, \ldots, x_n) \mid x_1 \in A_1 \wedge x_2 \in A_2 \wedge \cdots \wedge x_n \in A_n\}$ heißt *Produktmenge von* $A_1, A_2, \ldots, A_n$. □

Definition B.13. Sei A eine Menge.
Die Menge $A \times A \times \cdots \times A := \{(x_1, x_2, \ldots, x_n) \mid x_1 \in A \wedge x_2 \in A \wedge \cdots \wedge x_n \in A\}$ ist ein Spezialfall der eben definierten Produktmenge und heißt *n-te Mengenpotenz von A*. □

Bemerkungen:

- $(x_1, x_2, \ldots, x_n)$ wird *n-Tupel* genannt bzw.
- für $n = 2$ *geordnetes Paar*,
- für $n = 3$ *Tripel*.
- x_i heißt die *i-te Komponente*.

B.3 Relationen

Definition B.14. Eine (binäre) *Relation zwischen einer Menge A und einer Menge B* ist eine Teilmenge R von $A \times B$, kurz: $R \subseteq A \times B$.
Ist $R \subseteq A \times B$ und $(x, y) \in R$, so bezeichnet man y als Bild von x bei R und x als Urbild von y bei R. Weiterhin bezeichnet man

$$D(R) = \{x \mid \text{es gibt ein } y, \text{ so dass } (x, y) \in R\}$$

als *Definitionsbereich von R* und

$$W(R) = \{y \mid \text{es gibt ein } x, \text{ so dass } (x, y) \in R\}$$

als *Wertebereich von R*.
Im Spezialfall $A = B$ wird $R \subseteq A \times A$ mit *Relation auf A* bezeichnet. □

Definition B.15. Ist R eine Relation zwischen A und B, so heißt die Relation $R^{-1} = \{(y, x) \mid (x, y) \in R\}$ die zu R *inverse Relation*. □

Definition B.16. Sei $R \subseteq A \times B$ und $S \subseteq B \times C$.
Unter der *Komposition* (auch mit *Relationenprodukt* bezeichnet) der Relation R und S versteht man die Relation $R \circ S \subseteq A \times C$ (lies: S nach R), die einem $x \in A$ ein $z \in C$ wie folgt zuordnet:

$$(x, z) \in R \circ S \Leftrightarrow \text{ es gibt ein } y \in B \text{ so dass } ((x, y) \in R \wedge (y, z) \in S)$$ □

Beachte:

- Für die Komposition gilt das Assoziativgesetz, aber *nicht* das Kommutativgesetz.
- Es kann $R \circ S = \emptyset$ sein, ohne dass $R = \emptyset$ oder $S = \emptyset$ gilt!

Sei $R \subseteq A \times A$. Die *reflexive, transitive Hülle* von R ist die binäre Relation $R^* = \bigcup_{n \geq 0} R^n$, wobei (i) $R^0 = diag_A$ und $diag_A = \{(a, a) \mid a \in A\}$ und (ii) für $n \geq 0$, $R^{n+1} = R^n \circ R$. Die *transitive Hülle* von R, bezeichnet durch R^+, ist definiert durch $R^+ = \bigcup_{n \geq 1} R^n$.
Eine binäre Relation $R \subseteq A \times A$ heißt

- *reflexiv*, wenn für jedes $a \in A$ gilt $(a, a) \in R$ (oder äquivalent dazu: $diag_A \subseteq R$),
- *symmetrisch*, wenn für jedes $a, b \in A$ gilt: aus $(a, b) \in R$ folgt $(b, a) \in R$ (oder: $R = R^{-1}$),

- *antisymmetrisch*, wenn für jedes $a, b \in A$ gilt: aus $(a, b) \in R$ und $(b, a) \in R$ folgt $a = b$
- *transitiv*, wenn für jedes $a, b, c, \in A$ gilt: aus $(a, b) \in R$ und $(b, c) \in R$ folgt $(a, c) \in R$ (oder: $R \circ R \subseteq R$).

Eine Relation R auf A, die reflexiv, symmetrisch und transitiv ist, nennt man *Äquivalenzrelation*; ist R reflexiv, antisymmetrisch und transitiv, so heißt sie *partielle Ordnung*. Eine partielle Ordnung R heißt *total*, wenn für je zwei Elemente $a, b \in A$ gilt: Wenn $a \neq b$, dann aRb oder bRa.

B.4 Abbildungen

Definition B.17. Seien A und B Mengen. Unter einer *Abbildung oder Funktion* von A in B versteht man eine (binäre) Relation f, so dass es für $x \in A$ *genau* ein Element $y \in B$ gibt, so dass $(x, y) \in f$. Dieses Element y wird im allgemeinen mit $f(x)$ bezeichnet.

Kurzbezeichnung der Funktion: $f : A \to B$. □

Eine Abbildung f von A in B ist somit eine *eindeutige* (binäre) Relation $f \subseteq A \times B$, d. h. jedes Element von A hat höchstens ein Bild. Definitionsbereich $D(f)$ und Wertebereich $W(f)$ werden entsprechend den bei Relationen gemachten Aussagen festgelegt und sind bei der Beschreibung von Abbildungen von grundlegender Bedeutung. So ist eine Abbildung erst vollständig beschrieben, wenn neben der Formulierung der eigentlichen funktionalen Beziehung $f(x)$ auch der zugehörige Definitionsbereich $D(f)$ angegeben wird.

Definition B.18. Sei $f : A \to B$ eine Abbildung. Dann heißt f:

- *injektiv*, wenn es zu jedem $y \in B$ höchstens ein $x \in A$ gibt, so dass $f(x) = y$.
- *surjektiv*, wenn es zu jedem $y \in B$ mindestens ein $x \in A$ gibt, so dass $f(x) = y$.
- *bijektiv*, wenn es zu jedem $y \in B$ genau ein $x \in A$ gibt, so dass $f(x) = y$. □

Bemerkungen:

- Es ist üblich, das Argument x als unabhängige Variable und y als abhängige Variable zu bezeichnen.
- Ist f injektiv, so haben verschiedene Argumente x stets verschiedene Bilder.
- Ist f surjektiv, so gilt $W(f) = B$.
- (f bijektiv) $\Leftrightarrow$ (f injektiv $\wedge$ f surjektiv).
- Zwei Abbildungen f und g sind genau dann gleich, wenn gilt $D(f) = D(g)$ und $f(x) = g(x)$ für alle Argumente x.
- Ist der Definitionsbereich $D(f)$ Teilmenge einer n-fachen Produktmenge $M_1 \times M_2 \times \cdots \times M_n$, so nennt man f eine Abbildung (Funktion) von n Variablen bzw. eine n-stellige Abbildung.
- f heißt reellwertig, falls $D(f)$ eine Teilmenge der reellen Zahlen ist.

Beispiel B.19. Sei $\mathbb{R}^+ = \{x \mid x \in \mathbb{R} \wedge x \geq 0\}$, dann ist

- $f : \mathbb{R} \to \mathbb{R}$ mit $f(x) = x^2$ weder injektiv noch surjektiv,
- $f : \mathbb{R}^+ \to \mathbb{R}$ mit $f(x) = x^2$ injektiv, jedoch nicht surjektiv,
- $f : \mathbb{R} \to \mathbb{R}^+$ mit $f(x) = x^2$ surjektiv, jedoch nicht injektiv,
- $f : \mathbb{R}^+ \to \mathbb{R}^+$ mit $f(x) = x^2$ injektiv und surjektiv, also bijektiv. □

Definition B.20. Eine bijektive Abbildung einer Menge A auf sich selbst wird *Permutation auf A* genannt. □

Definition B.21. Sei $f : A \to B$ bijektiv. Die Abbildung $B \to A$, die jedem $y \in B$ das (eindeutig bestimmte) Element $x \in A$ mit $f(x) = y$ zuordnet, heißt die zu f *inverse Abbildung oder Umkehrabbildung* und wird mit f^{-1} bezeichnet. □

Bemerkungen:

- Bijektive Abbildungen werden auch *umkehrbare oder eineindeutige* Abbildungen genannt.
- Wenn f eine (beliebige) Abbildung ist, dann ist die Relation f^{-1} nicht notwendig eine Abbildung.

Beispiel B.22. Wie bereits festgestellt, ist die Abbildung $f : \mathbb{R}^+ \to \mathbb{R}^+$ mit $f(x) = x^2$ bijektiv, so dass die Voraussetzung für die Bildung der inverse Abbildung erfüllt ist.

Aus der Elementarmathematik ist die gesuchte Umkehrfunktion $f^{-1} : \mathbb{R}^+ \to \mathbb{R}^+$ mit $f(x) = \sqrt{x}$ allgemein bekannt. □

Definition B.23. Seien $f : A \to B$ und $g : B \Rightarrow C$ Abbildungen. Dann ist die *Komposition oder Verkettung* $f \circ g$ eine Abbildung von A in C und durch das Relationenprodukt erklärt.

Kurzbezeichnung: $f \circ g : A \Rightarrow C$ mit $(f \circ g)(x) = g(f(x))$. □

Beachte:

- Auch hier gilt im allgemeinen $f \circ g \neq g \circ f$.
- Sind f und g injektiv bzw. surjektiv bzw. bijektiv, dann ist auch $f \circ g$ injektiv bzw. surjektiv bzw. bijektiv.

B.5 Prinzip der vollständigen Induktion

Die Beweismethodik der vollständigen Induktion geht aus dem Aufbau der Menge der natürlichen Zahlen zurück und nutzt insbesondere deren axiomatische Definition mittels des *PEANO'schen Axiomensystems.*

PEANO'sches Axiomensystem

1. Die Zahl Null ist eine natürliche Zahl.
2. Jede natürliche Zahl besitzt einen eindeutig bestimmten unmittelbaren Nachfolger.
3. Jede natürliche Zahl ist unmittelbarer Nachfolger höchstens einer natürlichen Zahl.
4. Die Zahl Null ist nicht Nachfolger einer natürlichen Zahl.
5. Die Menge der natürlichen Zahlen ist bezüglich der Inklusion die kleinste Menge, die die Zahl Null und mit einer natürlichen Zahl auch deren unmittelbaren Nachfolger enthält.

Bemerkungen:

- Die Menge der natürlichen Zahlen ist somit *abzählbar unendlich.*
- Bezüglich der Operationen $+$ und $*$ gelten die Gesetze der Kommutativität, Assoziativität und Distributivität.
- Die Relation $\geq$ stellt innerhalb $\mathbb{N}$ eine totale Ordnung her.

Das 5. Axiom stellt sicher, dass jede natürliche Zahl von der Null aus erreichbar ist, indem man beliebig oft (aber nur endlich oft) den unmittelbaren Nachfolger bildet. Es wird auch Induktionsaxiom genannt und ist insbesondere die Grundlage für Beweise von Aussagen über natürlichen Zahlen durch vollständige Induktion. Damit sind zugleich Anwendungsbereiche und Grenzen dieses Beweisprinzips genannt.

Sei nun jeder natürlichen Zahl $n \in \mathbb{N}$ eine Aussage $A(n)$ zugeordnet und seien folgende Bedingungen erfüllt:

1. Es gilt $A(0)$ (diese Bedingung wird als *Induktionsanfang* bezeichnet).
2. Für jedes $n \in \mathbb{N}$ folgt aus der Gültigkeit von $A(n)$ stets auch die Gültigkeit von $A(n+1)$ (diese Implikation wird als *Induktionsschluss* oder *Induktionsschritt* bezeichnet).

Dann gilt $A(n)$ für alle $n \in \mathbb{N}$.

Bemerkungen:

- Der Induktionsanfang muss nicht zwingend bei Null liegen.
- Die Gültigkeit von $A(n)$ im Induktionsschluss nennt man *Induktionsvoraussetzung*
- Die Gültigkeit von $A(n+1)$ im Induktionsschluss nennt man *Induktionsbehauptung*

Üblicherweise wird das Prinzip der vollständigen Induktion durch folgenden Satz ausgedrückt.

> Gilt $A(n_0)$ mit $n_0 \in \mathbb{N}$ und folgt für jedes $n \geq n_0$ mit $n \in \mathbb{N}$ aus der Gültigkeit von $A(n)$ stets auch die von $A(n+1)$, so gilt $A(n)$ für alle natürlichen Zahlen $n \geq n_0$.

Abschließend soll noch ein praktikables Beweisschema der vollständigen Induktion angegeben werden:

Induktionsanfang Es ist zu zeigen, dass für ein möglich kleines $n_0 \in \mathbb{N}$ die Aussage $A(n_0)$ gilt; i. allg. wird $n_0 = 0$ oder $n_0 = 1$ sein.

Induktionsschritt Für eine feste natürliche Zahl $n \in \mathbb{N}$, $n \geq n_0$ gelte die Aussage $A(n)$ (*Induktionsvoraussetzung*)

Dann gilt auch die Aussage $A(n+1)$ (*Induktionbehauptung*)

denn ...

Hier ist der mathematische Beweis anzugeben, dass aus $A(n)$ durch Anwendung einer gültigen Transformation schließlich $A(n+1)$ entsteht.

Bei der praktischen Durchführung eines solchen Beweises wird regelmäßig die Schwierigkeit im Finden bzw. Auswerten dieser Transformation stecken.

Beispiel B.24. Aussage: In einem konvexen n-Eck mit $n \geq 4$ ist die Anzahl d_n der Diagonalen:

$$d_n = \frac{n^2 - 3 * n}{2} .$$

1. Feststellung: Es handelt sich um eine Aussage mit $n \in \mathbb{N}$, also vollständige Induktion geeignet zur Beweisführung.
2. Feststellung: n_0 ist in unserem Fall gleich 4, und die Aussage $d_4 = 2$ ist richtig.
3. Durch systematisches Probieren erhält man das Transformationsglied $\Delta = (n-2)+1$ als Zuwachs an Diagonalen beim Übergang eines n-Ecks zu einem $(n+1)$-Eck.

Mit Hilfe dieser Erkenntnisse lässt sich der Beweis problemlos aufschreiben, so dass an dieser Stelle darauf verzichtet werden kann. □

Eine sehr schöne Sammlung von Beweisen durch vollständige Induktion ist in [SGJ86] enthalten.

B.6 Termdefinition

Definition B.25. Ein *Rangalphabet* ist ein Paar (Δ, rk), wobei Δ eine endliche Menge ist und $rk : \Delta \to \mathbb{N}$ jedem Symbol eine Stelligkeit (oder: Rang) zuordnet. □

Für jedes $n \geq 0$ bezeichnen wir die Menge $\{\delta \in \Delta \mid rk(\delta) = n\}$ durch $\Delta^{(n)}$. Wenn für jedes Symbol aus Δ sein Rang aus dem Kontext hervorgeht, dann schreiben wir auch Δ statt (Δ, rk) und benutzen die Abkürzung rk_Δ für den Fall, dass wir doch die Rangfunktion benutzen wollen.

Definition B.26. Sei Δ ein Rangalphabet, und sei A eine beliebige Menge (Indexmenge). Die Menge aller *Terme über* Δ *und* A, bezeichnet durch $T_\Delta(A)$, ist die kleinste Menge $T \subseteq (\Delta \cup A \cup \{(,), ,\})^*$, so dass gilt:

(i) $A \cup \Delta^{(0)} \subseteq T$ und

(ii) für jedes $k \geq 1$, $\delta \in \Delta^{(k)}$ und $t_1, \ldots, t_k \in T$, dann $\delta(t_1, \ldots, t_k) \in T$. □

Statt $T_\Delta(\emptyset)$ schreiben wir auch T_Δ.

Die Elemente von Δ heißen in diesem Kontext Operationssymbole.

B.7 Wohlfundierte Induktion

Wir wollen nun den Begriff der *wohlfundierten Induktion* (oder *Noethersche Induktion*, benannt nach der Mathematikerin Emmy Noether) einführen. Im Anschluss werden wir zeigen, dass sich die vollständige Induktion sowie die strukturelle Induktion von der wohlfundierten Induktion ableiten und die Korrektheit der beiden ersten Induktionen aus der Korrektheit der letzteren folgt.

Definition B.27. Ein *abstraktes Reduktionssystem* ist ein Tupel $(D, \vdash)$, wobei D eine Menge und $\vdash \subseteq D \times D$ eine binäre Relation auf D ist. Anstatt $(d, d') \in \vdash$ schreiben wir $d \vdash d'$ und für ein $d \in D$ bezeichnet $\vdash(d)$ die Menge $\{d' \in D \mid d \vdash d'\}$.

Wir nennen $(D, \vdash)$ *terminierend*, wenn es keine unendliche Folge $d_0, d_1, d_2, \ldots$ mit $d_i \in D$ und $d_i \vdash d_{i+1}$ für jedes $i \in \mathbb{N}$ gibt. □

$(D, \vdash)$ ist also terminierend, wenn es keine unendliche Kette $d_0 \vdash d_1 \vdash d_2 \vdash \cdots$ gibt.

Satz B.28 (Wohlfundierte Induktion)**.** *Sei $(D, \vdash)$ ein terminierendes abstraktes Reduktionssystem und sei $A \subseteq D$. Wenn $\vdash(d) \not\subseteq A$ für jedes $d \in (D \setminus A)$ gilt, dann ist $A = D$.*

Beweis. Sei $\vdash(d) \not\subseteq A$ für jedes $d \in (D \setminus A)$ erfüllt. Nehmen wir aber an, dass $A \neq D$ ist. Da $A \subseteq D$, muss es ein $d_0 \in D$ geben, welches sich nicht in A befindet, es ist also $d_0 \in (D \setminus A)$. Dann ist aber $\vdash(d_0) \not\subseteq A$, es gibt also ein $d_1 \in \vdash(d_0)$, welches sich ebenfalls nicht in A befindet. Mit der gleichen Argumentation finden wir ein $d_2 \in \vdash(d_1)$ mit $d_2 \notin A$, usw. Setzen wir diesen Prozess beliebig fort, erhalten wir eine unendliche Kette $d_0 \vdash d_1 \vdash d_2 \vdash d_3 \vdash \cdots$, was im Widerspruch dazu steht, dass $(D, \vdash)$ terminierend ist. Also gilt $A = D$. ■

Satz B.29 (Vollständigen Induktion)**.** *Sei P eine Eigenschaft auf den natürlichen Zahlen, so dass $P(0)$ gilt und für jedes $n \in \mathbb{N}$ mit der Gültigkeit von $P(n)$ auch die Gültigkeit von $P(n+1)$ folgt. Dann gilt $P(n)$ für jedes $n \in \mathbb{N}$.*

Beweis. Wir definieren das abstrakte Reduktionssystem $(D, \vdash)$ mit $D = \mathbb{N}$ und $\vdash = \{(n+1, n) \mid n \in \mathbb{N}\}$. Durch $\vdash$ wird also jede natürliche Zahl $n \geq 1$ mit ihrem Vorgänger $n-1$ in Relation gesetzt. Es ist leicht zu sehen, dass $(D, \vdash)$ terminierend ist, da es zu jeder natürliche Zahl nur endlich viele kleinere natürliche Zahlen gibt. Wir definieren die Menge $A \subseteq \mathbb{N}$ als $A = \{n \in \mathbb{N} \mid P(n) \text{ gilt}\}$ und wollen zeigen, dass $A = \mathbb{N}$. Unter Nutzung von Satz B.28 genügt es, zu zeigen, dass $\vdash(n) \not\subseteq A$ für jedes $n \in (\mathbb{N} \setminus A)$.

Sei $n \in (\mathbb{N} \setminus A)$. Da $P(0)$ gilt und deshalb $0 \in A$ ist, muss $n \neq 0$ sein. Nehmen wir an, dass $(n-1) \in A$ ist. Dann wäre aber $P(n-1)$ erfüllt und es müsste laut Voraussetzung auch $P(n)$ gelten, was aber $n \in A$ zur Folge hätte, ein Widerspruch. Deshalb ist $(n-1) \notin A$. Da aber $(n-1) \in \vdash(n)$, können wir $\vdash(n) \not\subseteq A$ schlussfolgern. ■

Satz B.30 (Strukturellen Induktion)**.** *Sei Δ ein Rangalphabet, X eine beliebige Menge und P eine Eigenschaft auf der Menge $T_\Delta(X)$ der Terme über Δ und X, so dass $P(x)$ für jedes $x \in X$ gilt und für jedes $n \in \mathbb{N}$, $t_1, \ldots, t_n \in T_\Delta(X)$ und $\delta \in \Delta^{(n)}$ mit der Gültigkeit von $P(t_1)$, $P(t_2)$, …, $P(t_n)$ auch die Gültigkeit von $P(\delta(t_1, \ldots, t_n))$ folgt. Dann gilt $P(t)$ für jedes $t \in T_\Delta(X)$.*

Beweis. Wir definieren

$$\vdash = \{(\delta(t_1, \ldots, t_n), t_i) \mid n \in \mathbb{N},\, t_1, \ldots, t_n \in T_\Delta(X),\, \delta \in \Delta^{(n)} \text{ und } 1 \leq i \leq n\}\,.$$

Durch $\vdash$ wird also jeder Term $\delta(t_1, \ldots, t_n)$ mit seinen Teiltermen $t_1, \ldots, t_n$ in Relation gesetzt. Es ist leicht zu sehen, dass $(T_\Delta(X), \vdash)$ terminierend ist, da es zu jedem Term nur endlich viele Teilterme gibt. Wir definieren die Menge $A \subseteq T_\Delta(X)$ als $A = \{t \in T_\Delta(X) \mid P(t) \text{ gilt}\}$ und zeigen, dass $A = T_\Delta(X)$. Unter Nutzung von Satz B.28 genügt es, zu zeigen, dass $\vdash(t) \not\subseteq A$ für jedes $t \in (T_\Delta(X) \setminus A)$.

Sei $t \in (T_\Delta(X) \setminus A)$. Da für jedes $x \in X$ gilt, dass $P(x)$ erfüllt ist und deshalb $x \in A$ ist, muss $t \neq x$ sein. Also ist t von der Form $\delta(t_1, \ldots, t_n)$ für ein $n \in \mathbb{N}$, $t_1, \ldots, t_n \in T_\Delta(X)$ und $\delta \in \Delta^{(n)}$. Wenn $t_1, \ldots, t_n \in A$ ist, dann gilt $P(t_1)$ bis $P(t_n)$, was laut Voraussetzung die Gültigkeit von $P(t)$ zur Folge hat; dann wäre aber $t \in A$, ein Widerspruch. Deshalb gibt es ein $i \in \{1, \ldots, n\}$ mit $t_i \notin A$. Da aber $t_i \in \vdash(t)$, können wir $\vdash(t) \not\subseteq A$ schlussfolgern. ■

Da Listen in Haskell spezielle Terme auf dem Rangalphabet $\{\texttt{:}^{(2)}, \texttt{[]}^{(0)}\}$ sind, folgt die Korrektheit der Induktion über Listen aus der Korrektheit der strukturellen Induktion.

B.8 Fixpunkttheorem von Tarski

Zunächst wollen wir als Hilfsmittel zur Formulierung dieses Theorems noch einige Begriffe und Definitionen bereitstellen und an Beispielen festigen.

Definition B.31. Sei A eine Menge und $\leq$ eine auf dieser Menge erklärte binäre Relation.

Man nennt $\mathcal{A} = (A, \leq)$ eine *Halbordnung* (partial order), wenn die Relation $\leq$ reflexiv, antisymmetrisch und transitiv ist. □

Definition B.32. Seien $a \in A$ und $T \subseteq A$.

Man nennt a *obere Schranke* von T, wenn gilt: $t \leq a$ für alle $t \in T$. Man nennt a *kleinste obere Schranke* (Supremum) von T, bezeichnet mit $a = \sup T$, falls für jede obere Schranke b von T gilt: $a \leq b$. □

Definition B.33. Seien $a \in A$ und $T \subseteq A$.

Man nennt a *untere Schranke* von T, wenn gilt: $a \leq t$ für alle $t \in T$. Man nennt a *größte untere Schranke* (Infimum) von T, bezeichnet mit $a = \inf T$, falls für jede untere Schranke b von T gilt: $b \leq a$. □

Definition B.34. Gilt für eine Folge $(a_i \mid i \in \mathbb{N})$ mit $a_i \in A$, dass $a_i \leq a_{i+1}$ für jedes $i \in \mathbb{N}$ gilt, so nennt man diese Folge eine *ω-Kette*. □

Definition B.35. Die Halbordnung $\mathcal{A} = (A, \leq)$ wird *strikt* genannt, wenn es ein kleinstes Element $\bot \in A$ gibt, d. h. für jedes $a \in A$ gilt $\bot \leq a$. □

Definition B.36. Die Halbordnung $\mathcal{A} = (A, \leq)$ wird *ω-vollständig* genannt, wenn $\mathcal{A}$ strikt ist und jede ω-Kette von $\mathcal{A}$ ein Supremum hat. □

Definition B.37. Die Halbordnung $\mathcal{A} = (A, \leq)$ wird *vollständiger Verband* genannt, wenn jede Teilmenge von A ein Supremum besitzt. □

Definition B.38. Sei $\mathcal{A} = (A, \leq)$ eine Halbordnung und $f : A \to A$ sei monoton, d. h. es gilt $f(a) \leq f(b)$ für alle $a, b \in A$ mit $a \leq b$. Wenn $\mathcal{A} = (A, \leq)$ ω-vollständig ist, dann heißt f *stetig*, wenn f monoton ist und es gilt:

$$\text{für jede } \omega\text{-Kette } (a_i \mid i \in \mathbb{N}) \text{ mit } a_i \in A \text{ gilt: } f(\sup\{a_i \mid i \in \mathbb{N}\}) = \sup\{f(a_i) \mid i \in \mathbb{N}\}\ . \qquad \square$$

Beispiel B.39. Wenn Δ ein beliebiges Alphabet ist, so ist $(\mathcal{P}(\Delta^*), \subseteq)$ eine Halbordnung.

- Das kleinste Element von $\mathcal{P}(\Delta^*)$ ist $\bot = \emptyset$, somit ist $(\mathcal{P}(\Delta^*), \subseteq)$ strikt.
- Wählen wir $M_i \in \mathcal{P}(\Delta^*)$ mit $i \in \mathbb{N}$ derart, dass für alle i gilt: $M_i \subseteq M_{i+1}$, dann ist $(M_i \mid i \in \mathbb{N})$ eine ω-Kette.
- Sei $(M_i \mid i \in \mathbb{N})$ eine ω-Kette. Mit $\sqcup(M_i \mid i \in \mathbb{N}) = \cup\{M_i \mid i \in \mathbb{N}\}$ erhalten wir das Supremum der ω-Kette, somit ist $(\mathcal{P}(\Delta^*), \subseteq)$ ω-vollständig.
- Für alle Teilmengen $T \subseteq \mathcal{P}(\Delta^*)$ lässt sich berechnen: $\sqcup(M \mid M \in T) = \cup\{M \mid M \in T\}$, somit ist $(\mathcal{P}(\Delta^*), \subseteq)$ ein vollständiger Verband.

  Sei z. B. $\Delta = V \cup \Sigma \cup \{\hat{\{}, \hat{\}}, \hat{[}, \hat{]}, \hat{(}, \hat{)}, \hat{|}\}$, und seien V und Σ beliebige Mengen von syntaktischen Variablen bzw. Terminalsymbolen. Weiterhin sei $f : \mathcal{P}(\Delta^*) \to \mathcal{P}(\Delta^*)$.

  Für $M \in \mathcal{P}(\Delta^*)$ ist die folgende Abbildung:

$$f(M) = V \cup \Sigma \cup \{\hat{(}\alpha\hat{)} \mid \alpha \in M\} \cup \{\hat{\{}\alpha\hat{\}} \mid \alpha \in M\} \cup \{\hat{[}\alpha\hat{]} \mid \alpha \in M\}$$
$$\cup \{\hat{(}\alpha_1\hat{|}\alpha_2\hat{)} \mid \alpha_1, \alpha_2 \in M\} \cup \{\alpha_1\alpha_2 \mid \alpha_1, \alpha_2 \in M\}\ ,$$

  monoton und stetig. □

1. Fixpunktsatz

Satz B.40. *Sei $\mathcal{A} = (A, \leq)$ ein vollständiger Verband und $f : A \to A$ monoton. Dann ist $a_0 = \inf\{a \in A \mid f(a) \leq a\}$ das kleinste Element der Menge $T = \{a \in A \mid f(a) \leq a\}$, und es gilt: $f(a_0) = a_0$. Das Element a_0 ist* kleinster Fixpunkt *von f, bezeichnet durch fix(f).*

Beweis. Wenn $a \in T$, dann gilt: $f(a) \leq a$. Da $a_0 \leq T$ gilt, muss auch für alle $a \in T$ gelten: $a_0 \leq a$. Mit f monoton, gilt $f(a_0) \leq f(a)$ und mit $f(a) \leq a$ wegen der Transitivität von $\leq$ auch $f(a_0) \leq a$. Da $a \in T$ beliebig gewählt ist, gilt $f(a_0) \leq T$. Damit ist $f(a_0)$ eine untere Schranke von T. Da a_0 die größte untere Schranke von T ist, muss gelten: $f(a_0) \leq a_0$ und $a_0 \in T$. Wegen der Monotonie von f folgt auch $f(f(a_0)) \leq f(a_0)$. Also gilt $f(a_0) \in T$. Aus $a_0 \leq T$ folgt nun $a_0 \leq f(a_0)$. Aus den Beziehungen $f(a_0) \leq a_0$ und $a_0 \leq f(a_0)$ folgt auf Grund der Antisymmetrie von $\leq$ die Gleichung $a_0 = f(a_0)$.

Nehmen wir an, es gäbe einen weiteren Fixpunkt b, also $b \in A$ und $f(b) = b$. Wegen der Reflexivität von $\leq$ gilt dann $f(b) \leq f(b) = b$, also auch $f(b) \leq b$, d. h. $b \in T$. Da nun $a_0 \leq T$ gilt, muss auch $a_0 \leq b$ gelten, und somit ist a_0 *kleinster* Fixpunkt von f. ∎

2. Fixpunktsatz

Satz B.41. *Sei $\mathcal{A} = (A, \leq)$ ω-vollständig und $f : A \to A$ stetig. Dann ist $a_0 = \sup\{f^i(\bot) \mid i \in \mathbb{N}\}$ kleinstes Element in der Menge $T = \{a \in A \mid f(a) \leq a\}$, und es gilt: $f(a_0) = a_0$. Das Element a_0 ist* kleinster Fixpunkt *von f, bezeichnet durch fix(f).*

Beweis. Das Element a_0 ist definiert, weil $(f^i(\bot) \mid i \in \mathbb{N})$ wegen der Monotonie von f und $\bot \leq f(\bot)$ eine ω-Kette ist. Da f stetig vorausgesetzt wurde, folgt:

$$\begin{aligned}
f(a_0) &= f(\sup\{f^i(\bot) \mid i \in \mathbb{N}\}) && \text{(Definition } a_0\text{)}\\
&= \sup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\} && \text{(da } f \text{ stetig)}\\
&= \sup\{\bot, \sup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\}\} && \text{(da } \bot \leq a \text{ für jedes } a \in A\text{)}\\
&= \sup\{f^i(\bot) \mid i \in \mathbb{N}\} && \text{(da } \bot = f^0(\bot)\text{)}\\
&= a_0\ .
\end{aligned}$$

Sei $a \in T$, so gilt $f(a) \leq a$. Da f stetig, gilt auch für beliebige $i \in \mathbb{N}$:

$$\bot \leq a \Rightarrow f^i(\bot) \leq f^i(a) \qquad \text{und} \qquad f(a) \leq a \Rightarrow f^i(a) \leq a\ .$$

Wegen der Transitivität von $\leq$ gilt somit für alle $i \in \mathbb{N}$: $f^i(\bot) \leq a$. Da $a_0 = \sup\{f^i(\bot) \mid i \in \mathbb{N}\}$ und $f^i(\bot) \leq a$ für alle $i \in \mathbb{N}$, folgt $a_0 \leq a$. D. h., a_0 ist kleinstes Element in T.

Nehmen wir an, es gäbe einen weiteren Fixpunkt b, also $b \in A$ und $f(b) = b$. Wegen der Reflexivität von $\leq$ gilt dann $f(b) \leq f(b) = b$, also auch $f(b) \leq b$, d. h. $b \in T$. Da nun $a_0 \leq T$ gilt, muss auch $a_0 \leq b$ gelten, und somit ist a_0 *kleinster* Fixpunkt von f. ∎

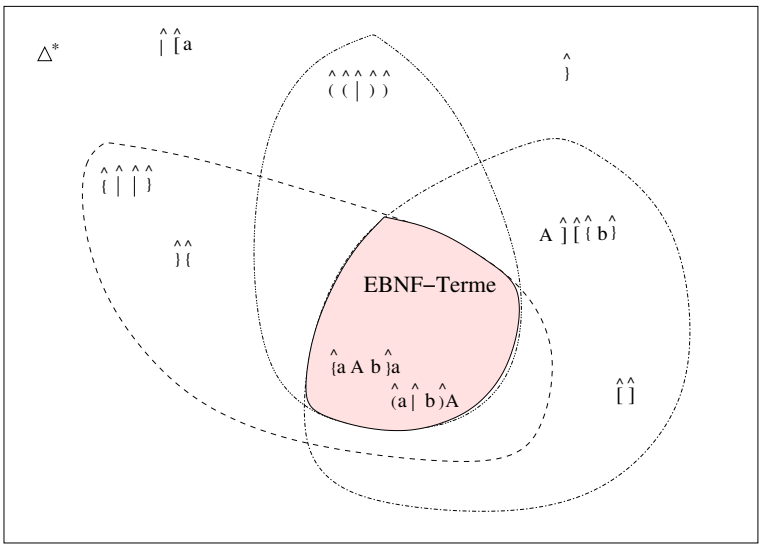
Beispiel B.42. Gegeben sei $\mathcal{A} = (\mathcal{P}(\Delta^*),\ \subseteq)$ mit $\Delta = V \cup \Sigma \cup \{\hat{\{}, \hat{\}}, \hat{[}, \hat{]}, \hat{(}, \hat{)}, \hat{|}\}$ und eine Funktion $f : \mathcal{P}(\Delta^*) \to \mathcal{P}(\Delta^*)$, die wie folgt definiert ist: für $M \in \mathcal{P}(\Delta^*)$ sei

$$\begin{aligned}
f(M) = V \cup \Sigma \cup \{\hat{(}\alpha\hat{)} \mid \alpha \in M\} \cup \{\hat{\{}\alpha\hat{\}} \mid \alpha \in M\} \cup \{\hat{[}\alpha\hat{]} \mid \alpha \in M\}\\
\cup \{\hat{(}\alpha_1\hat{|}\alpha_2\hat{)} \mid \alpha_1, \alpha_2 \in M\} \cup \{\alpha_1\alpha_2 \mid \alpha_1, \alpha_2 \in M\}\ .
\end{aligned}$$

Die Funktion f ist stetig. Die Menge der EBNF-Terme $T(V, \Sigma)$ über V und Σ lässt sich dann wie folgt berechnen:

$$T(V, \Sigma) = \bigcap\{M \in \mathcal{P}(\Delta^*) \mid f(M) \subseteq M\}\ .$$

Wir erhalten also die Menge $T(V, \Sigma)$ als größte Gemeinsamkeit der Mengen $\{M \in \mathcal{P}(\Delta^*) \mid f(M) \subseteq M\}$. Nun legen wir für unser Beispiel $V = \{A\}$ und $\Sigma = \{a, b\}$ fest.



Beginnend mit dem kleinsten Element $\bot = \emptyset$ lässt sich durch wiederholte Anwendung von f die Menge $T(\{A\}, \{a, b\})$ erzeugen:

$$\begin{aligned}
\emptyset &\overset{f}{\longmapsto} V \cup \Sigma = \{a, b, A\} \\
\{a, b, A\} &\overset{f}{\longmapsto} V \cup \Sigma \cup \{\hat{\{}\hat{a}\hat{\}}, \hat{\{}\hat{b}\hat{\}}, \hat{\{}\hat{A}\hat{\}}, \hat{[}\hat{a}\hat{]}, \hat{[}\hat{b}\hat{]}, \hat{[}\hat{A}\hat{]}, \hat{(}\hat{a}\hat{)}, \hat{(}\hat{b}\hat{)}, \hat{(}\hat{A}\hat{)}\} \\
&\quad \cup \hat{(}\hat{b}|\hat{a}\hat{)}, \hat{(}\hat{A}|\hat{a}\hat{)}, \hat{(}\hat{b}|\hat{A}\hat{)}\}, \hat{(}\hat{a}|\hat{b}\hat{)}, \hat{(}\hat{a}|\hat{A}\hat{)}, \hat{(}\hat{A}|\hat{b}\hat{)}\} \\
&\quad \cup \{aa, bb, ab, ba, Aa, aA, bA, Ab, AA\} \\
&\cdots \\
&\cdots \\
&\overset{f}{\longmapsto} T(\{A\}, \{a, b\}) \, .
\end{aligned}$$

□

Beispiel B.43. Sei $\mathcal{E} = (V, \Sigma, S, R)$ eine EBNF-Definition mit $V = \{S, A\}$, $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und R bestehe aus den Regeln

$$\begin{aligned}
S &::= \hat{\{}\hat{\mathtt{c}}\hat{\}}A \, , \\
A &::= \hat{(}\hat{(}\mathtt{a}A\mathtt{b}\hat{)} \, \hat{|} \, \mathtt{a}\hat{)} \, ,
\end{aligned}$$

die wir bereits aus Kapitel 2 kennen. Wir wollen jetzt mit Hilfe der Fixpunktsemantik die syntaktischen Kategorien $W(\mathcal{E}, S)$ und $W(\mathcal{E}, A)$ ermitteln.

Dazu betrachten wir den vollständigen Verband $(\mathcal{P}(\Sigma^*)^2, \subseteq)$ mit $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Dabei gibt in einem Element (L_1, L_2) von $\mathcal{P}(\Sigma^*)^2$ die erste Komponente L_1 die Sprache, die zu S gehört, und die zweite Komponente L_2 die Sprache, die zu A gehört, an. Wir definieren die stetige Abbildung $f : \mathcal{P}(\Sigma^*)^2 \to \mathcal{P}(\Sigma^*)^2$ wie folgt:

$$f((L_1, L_2)) = (\{\mathtt{c}^n w \mid n \geq 0, w \in L_2\}, \{\mathtt{a}w\mathtt{b} \mid w \in L_2\} \cup \{\mathtt{a}\}) \, ,$$

wobei wir für die Mengentupel festlegen: $(L_1, L_2) \subseteq (L_1', L_2')$ genau dann wenn gilt: $L_1 \subseteq L_1'$ und $L_2 \subseteq L_2'$. Das kleinste Element von $(\mathcal{P}(\Sigma^*)^2, \subseteq)$ ist bekanntlich $\bot = (\emptyset, \emptyset)$.

Die Anwendung von f, beginnend mit dem Argument $(\emptyset, \emptyset)$, wollen wir jetzt aufschreiben. Der Übersichtlichkeit wegen benutzen wir statt der Tupel zweizeilige Spaltenmatrizen.

$$\begin{pmatrix} \emptyset \\ \emptyset \end{pmatrix} \overset{f}{\longmapsto} \begin{pmatrix} \emptyset \\ \{\mathtt{a}\} \end{pmatrix} \overset{f}{\longmapsto} \begin{pmatrix} \{\mathtt{c}^n\mathtt{a} \mid n \geq 0\} \\ \{\mathtt{aab}\} \cup \{\mathtt{a}\} \end{pmatrix} \overset{f}{\longmapsto} \cdots \, .$$

Behauptung: Allgemein gilt für jedes $i \geq 0$:

$$\begin{pmatrix} \emptyset \\ \emptyset \end{pmatrix} \overset{f^i}{\longmapsto} \begin{pmatrix} \{\mathtt{c}^n\mathtt{a}^k\mathtt{ab}^k \mid n \geq 0, \ 0 \leq k \leq i-2\} \\ \{\mathtt{a}^k\mathtt{ab}^k \mid 0 \leq k \leq i-1\} \end{pmatrix} \, .$$

Beweis durch vollständige Induktion über i:

$i = 0$

$$f^0(\bot)(S) = \bot(S) = \emptyset = \{c^n a^k a b^k \mid n \geq 0,\ 0 \leq k \leq -2\}$$
$$f^0(\bot)(A) = \bot(A) = \emptyset = \{a^k a b^k \mid 0 \leq k \leq -1\}$$

$i \rightsquigarrow i+1$ Angenommen die Behauptung gilt für i. (I.H.)
Zu zeigen: die Behauptung gilt für $i+1$.

$$\begin{aligned}
\left(f^{i+1}(\bot)\right)(S) &= \left(f\left(f^i(\bot)\right)\right)(S) \\
&= [\![\{c\}A]\!]\left(f^i(\bot)\right) && \text{(Def. von } f \text{ mit } \rho = f^i(\bot)) \\
&= \{c^n \mid n \geq 0\} \cdot [\![A]\!]\left(f^i(\bot)\right) \\
&= \{c^n \mid n \geq 0\} \cdot \left(f^i(\bot)\right)(A) \\
&= \{c^n \mid n \geq 0\} \cdot \{a^k a b^k \mid 0 \leq k \leq i-1\} && \text{(I.H.)} \\
&= \{c^n a^k a b^k \mid n \geq 0,\ 0 \leq k \leq i-1\} \\
&= \{c^n a^k a b^k \mid n \geq 0,\ 0 \leq k \leq (i+1)-2\}
\end{aligned}$$

$$\begin{aligned}
\left(f^{i+1}(\bot)\right)(A) &= \left(f\left(f^i(\bot)\right)\right)(A) \\
&= [\![(aAb|a)]\!]\left(f^i(\bot)\right) \\
&= \{a\} \cdot \left(f^i(\bot)\right)(A) \cdot \{b\} \cup \{a\} \\
&= \{a\} \cdot \{a^k a b^k \mid 0 \leq k \leq i-1\} \cdot \{b\} \cup \{a\} \\
&= \{a^k a b^k \mid 1 \leq k \leq i\} \cup \{a\} \\
&= \{a^k a b^k \mid 0 \leq k \leq i\} \\
&= \{a^k a b^k \mid 0 \leq k \leq (i+1)-1\}
\end{aligned}$$

Somit erhalten wir:

$$\bigcup\{f^i(\bot) \mid i \geq 0\} = \bigcup\{f^i((\emptyset,\emptyset)) \mid i \geq 0\} = \begin{pmatrix}\{\mathtt{c}^n\mathtt{a}^k\mathtt{ab}^k \mid n,k \geq 0\} \\ \{\mathtt{a}^k\mathtt{ab}^k \mid k \geq 0\}\end{pmatrix} = \begin{pmatrix}W(\mathcal{E},S) \\ W(\mathcal{E},A)\end{pmatrix},$$

insbesondere ist also die Sprache der EBNF: $W(\mathcal{E},S) = \mathrm{fix}(f)_1 = \{\mathtt{c}^n\mathtt{a}^k\mathtt{ab}^k \mid n,k \geq 0\}$. □

Liste der Algorithmen

Literaturverzeichnis

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AO91] K. Apt and E.R. Olderog. *Programmverifikation – Sequentielle, parallele und verteilte Programme.* Springer-Verlag, 1991.

[AO97] K. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1997. 2nd edition.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un systeme de communication homme-machine en francais. Technical report, Groupe de rechereche en intelligence artificielle, Université d' Aix-Marseille, 1973.

[CLR90] Th.H. Cormen, Ch.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* The MIT Press, 1990.

[CLRS04] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Algorithmen - Eine Einführung.* Oldenbourg Verlag, 2004.

[Gie00] J. Giesl. Context-moving transformations for function verification. In *1999 International Workshop on Logic-Based Program Synthesis and Transformation, Selected Papers*, volume 1817 of *Lecture Notes in Comput. Sci.*, pages 293–312. Springer-Verlag, 2000.

[GKV03] J. Giesl, A. Kühnemann, and J. Voigtländer. Deaccumulation — Improving provability. In V. Saraswat, editor, *8th Asian Computing Science Conference, ASIAN 2003, Mumbai, India, December 10-13, 2003, Proceedings*, volume 2896 of *Lecture Notes in Comput. Sci.*, pages 146–160. Springer-Verlag, 2003.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. of the ACM*, 12(10):576–583, 1969.

[HSAF94] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Grundlagen von Datenstrukturen in C.* International Thomson Publishing, 1994.

[Hut07] G. Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[Kni97] K. Knight. Automating knowledge acquisition for machine translation. *AI Magazine*, 18(4), 1997.

[Kow74] R. Kowalski. Predicate logic as a programming language. *Information Processing*, 74:569–574, 1974.

[Küh03] A. Kühnemann. Implementation of functional programming languages. Script, Dresden University of Technology, Department of Computer Science, Second Edition, 2003.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LNN00] K.A. Lambert, D.W. Nance, and T.L. Naps. *Introduction to Computer Science with C++.* Brooks/Cole, second edition edition, 2000.

[Mac71] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Comm. of the ACM*, 3:184–195184–195, 1960.

[MHR80] N. Metropolis, J. Howlett, and G. Rota. *A History of Computing in the 20th Century.* Academic Press, 1980.

[MP08] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2008.

[OGS08] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell.* O'Reilly Media, Inc., 1st edition, 2008.

[OW02] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen.* Sprektrum - Akademischer Verlag, 4 edition, 2002.

[Pie91] B. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computer Machinery*, 12:23–41, 1965.

[Sch93] U. Schöning. Vorlesungsskript Informatik I, 5. Auflage. Universität Ulm, 1993.

[SGJ86] I.S. Sominskij, L.I. Golovina, and I.M. Jaglom. *Die vollständige Induktion.* Deutscher Verlag der Wissenschaften, 1986.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1994.

[Wex81] R. Wexelblatt. *History of programming Languages.* Academic Press, 1981.